

TraceTracker: Hardware/Software Co-Evaluation for Large-Scale I/O Workload Reconstruction

Miryeong Kwon*, Jie Zhang*, Gyuyoung Park*, Wonil Choi†,
David Donofrio‡, John Shalf‡, Mahmut Kandemir† and Myoungsoo Jung*

Computer Architecture and Memory Systems Laboratory, School of Integrated Technology, Yonsei University*,
Pennsylvania State University†, Lawrence Berkeley National Laboratory‡

*mkwon@camelab.org, *jie@camelab.org, *gyuyoung@camelab.org, †wuc138@cse.psu.edu,
‡ddonofrio@lbl.gov, ‡jshalf@lbl.gov, †kandemir@cse.psu.edu, *mj@camelab.org

Abstract—Block traces are widely used for system studies, model verifications, and design analyses in both industry and academia. While such traces include detailed block access patterns, existing trace-driven research unfortunately often fails to find true-north due to a lack of runtime contexts such as user idle periods and system delays, which are fundamentally linked to the characteristics of target storage hardware. In this work, we propose *TraceTracker*, a novel hardware/software co-evaluation method that allows users to reuse a broad range of the existing block traces by keeping most their execution contexts and user scenarios while adjusting them with new system information. Specifically, our *TraceTracker*’s software evaluation model can infer CPU burst times and user idle periods from old storage traces, whereas its hardware evaluation method remasters the storage traces by interoperating the inferred time information, and updates all inter-arrival times by making them aware of the target storage system. We apply the proposed co-evaluation model to 577 traces, which were collected by servers from different institutions and locations a decade ago, and revive the traces on a high-performance flash-based storage array. The evaluation results reveal that the accuracy of the execution contexts reconstructed by *TraceTracker* is on average 99% and 96% with regard to the frequency of idle operations and the total idle periods, respectively.

I. INTRODUCTION

Tracing block accesses is a long-established method to extract and tabulate various system parameters. A set of collected I/O instructions, referred to as a block trace, can provide valuable insights into design tradeoffs and can be used for the implementations of various software subsystems and hardware components in storage stacks. Therefore, many proposals utilize a wide spectrum of block traces for system characterizations, model verifications, and design analyses [10], [15], [18]. Nevertheless, it is non-trivial, and ever-challenging to appropriately record block accesses on various large servers. Thus, open-license block traces, collected on different institutions and server locations, are extensively used in the computer and system communities [27], [9], [16], [11].

While these traces include detailed block access information, they can also lead to wrong results and conclusions for some

simulation-based analyses and design studies. Specifically, time information (i.e., inter-arrival time) on traces is intrinsically connected to the performance characteristics of the target storage. Since modern storage systems are undergoing significant technology shifts, different performance exhibited by new hardware can result in different I/O timing and user application behaviors. Furthermore, open-license block traces are collected on old systems that employ many hard disk drives (HDDs) designed a decade ago, which in turn can make system analysis and evaluation based on such block traces significantly different from the actual results that reflect the real characteristics of modern systems.

Even though the limited timing information is a matter for system research, it is extremely challenging to collect comprehensive information on a variety of servers and large-scale computing systems by incorporating many important (but unpredictable) user scenarios. For example, Microsoft’s exchange server workloads [9], which are one of most popular block traces in system community, recorded detailed I/O patterns across multiple production clusters, which were generated by 5,000 users. Even if one tries to retrace the workloads by constructing such servers with modern storage like solid state drives (SSDs), it is difficult to capture all system delays, idle operations and non-deterministic timing behaviors generated by thousands of users. To address these challenges, some replay methods statically accelerate the old traces to study peak performance [25], [8], [30]. However, these overly-simplified “Acceleration” methods are too imprecise to remaster the time information of the workloads. There also exist dynamic approaches that revise the block traces by issuing actual I/Os to a real system [14], [32], [4]. These “Revision” methods can make the inter-arrival times of workloads more realistic, but they can also lose other important runtime contexts such as user idle periods and system delays.

To be precise, we evaluate the inter-arrival times generated by an acceleration method (*Acceleration* [8]) and a revision method (*Revision* [4]), and the results are shown in Figure 1 in the form of a cumulative distribution function (CDF). In this evaluation, we generated 70 million instructions whose I/O patterns are same as a Microsoft’s network storage file server [9], and are issued to an HDD-based system node (*OLD*) and a SSD-based system node (*NEW*), respectively. Specifically,

This paper is accepted by and will be published at 2017 IEEE International Symposium on Workload Characterization. This document is presented to ensure timely dissemination of scholarly and technical work.

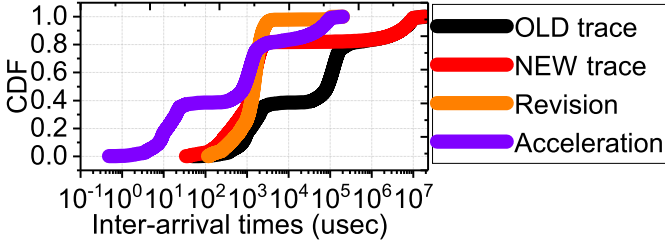


Fig. 1: Cumulative distribution function (CDF) for inter-arrival times observed by different methods and systems.

14 million instructions are issued in an asynchronous fashion, and we injected user idle operations that account for 20% of the total instructions to make I/O access more realistic. The same patterns are collected from both OLD and NEW for a fair comparison. The traces on OLD are used for Acceleration while Revision is implemented by reconstructing the workloads by replaying them on the SSD-based storage node. As shown in the figure, the first half of the distribution curve of Acceleration exhibits shorter inter-arrival times than that of the actual target system (NEW) by 88% on average, while losing 98% of user idle times, compared to the target system. Even though the timing trend of Revision appears similar to that of NEW, it still exhibits longer inter-arrival times at the first half of CDF curve than the ones of NEW by 16%, on average. More importantly, Revision fails to capture 18% of user idle operations and 69% of total idle periods, observed in the real system, NEW.

In this paper, we propose *TraceTracker*, a novel hardware/software co-evaluation method that allows users to reuse a broad range of the existing block traces by keeping most of their execution contexts and user behaviors while adjusting them to the new system information. Specifically, our proposed TraceTracker’s software evaluation model can infer CPU burst times and user idle periods from old-fashioned block traces, whereas its hardware evaluation method remasters the block traces by interoperating the inferred time information as well as renews all inter-arrival times by being aware of the target storage system. The proposed software and hardware co-evaluation methods can be implemented by using publicly-available benchmark tools such as FIO [2].

The main **contributions** can be summarized as follows:

- *Reviving the timing information for diverse workloads.* There are several workloads that provide no specific information or descriptions of the underlying storage trace collection environment. In this work, we analyze a diverse set of large-scale workloads and provide an inference model that estimates the relative time costs of an I/O request service. This inference model evaluates the realistic idle time that can capture the system and user behaviors from the traditional block traces by dividing the arrival time into a channel delay, a device time and an idle time. It decomposes I/O subsystem latency by analyzing the probability density functions and cumulative distribution functions of the inter-arrival times as well as being aware of the given request sizes and operation types.

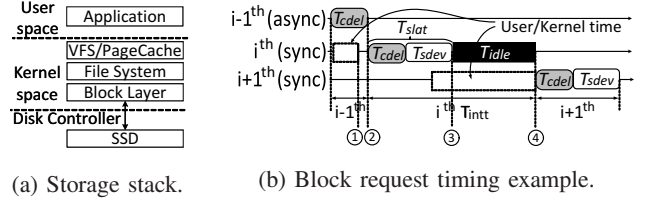


Fig. 2: Storage-level I/O information.

- *Inference automation and hardware/software co-evaluation.* Analyzing extensive out-of-date block traces is non-trivial, and reconstructing the traces is not a one-shot process as the target system will keep shifting its underlying storage technology. In this work, we reify the proposed inference model by automating our graph classification method and steepness analysis, each of which is used to examine massive trace data and speculate the underlying I/O system latency. With the timing information deduced by the proposed inference automation, TraceTracker simulates the old system behaviors and emulates I/O services on a real target system. TraceTracker also performs post-processing to revive asynchronous/synchronous information on new emulated traces. To verify the proposed trace reconstruction method, we introduce several verification metrics such as user idle detection and length. Even in the case of no runtime information being available on the trace collections, our TraceTracker can detect 99% of system delays and idle periods appropriately and secure the corresponding idle periods by 96% of a real execution, on average.

- *Massive trace reconstruction and analysis.* In this work, we reconstructed 577 traces that cover a diverse set of I/O workloads of large-scale computing systems, such as web services, data mining and network file system servers, and performed a comprehensive analysis of the reconstructed block traces. While previous work [8] claimed that 50% of write requests have time intervals that are 2x longer than the effective device operation latency, even after accelerating the block traces (of the same workloads that we tested) by 100x, we observed that the number of time intervals that have idle periods is less than 39% of the total number of I/O requests. Note that the majority of idle periods in all the block traces are found in 1 millisecond, which is also 10% shorter than the one reported by the prior study [8].

II. BACKGROUND

In this section, we first explain the storage-level I/O information from the perspective of a storage stack and block request timing sequence. We then introduce the existing trace revision methods and discuss their limitations.

A. Storage-Level I/O Information

Storage stack. Figure 2a illustrates a typical storage stack from an application to the underlying storage. Once the application makes an I/O request, it is required to switch from

All the traces collected for this paper are available for download from <http://trace.camelab.org>.

the user mode to the kernel mode and jump into an entry-point of a virtual file system (VFS). VFS then copies the corresponding target data from a user buffer to a kernel buffer (referred to as page cache [6]) and forwards the request to the underlying file system. During this time, the mode switch consumes CPU cycles for handling system calls and storing task states in addition to copying the buffers. The file system then looks up the physical locations (indicated by request) and submits this information to the block layer. Finally, the block layer partitions the translated information, including logical block address and request size (in terms of the number of sectors) into multiple packets (or transactions). Note that, before submitting the actual information to the underlying storage, the multiple layers in the storage stack consume CPU cycles for mode switches, data copies and address translations. Note also all open-license block traces are also typically collected underneath the block layer. In cases where there are no system delays or application idleness, the user/kernel specific CPU bursts can overlap with storage bursts, which make the computational cycles that upper software modules consume hide behind the inter-arrival times of multiple I/O requests at the block traces.

Block request timing. Figure 2b shows the timing diagram of block requests, which can be captured from underneath the block layer. There are three requests, denoted by $(i-1)^{th}$, i^{th} and $(i+1)^{th}$. In this example, the $(i-1)^{th}$ request is issued asynchronously, whereas all other requests are issued synchronously. Since the asynchronous block request does not need to wait for the response from the underlying device, there is only a delay caused by the storage interface (i.e., channel) data movement and corresponding data packet. This channel delay is referred to as T_{cdel} . In this work, the I/O subsystem latency, called T_{slat} , consists of T_{cdel} and the actual device time taken by the storage to service the request, denoted by T_{sdev} . The i^{th} request is ready to be submitted to the storage at ①, and therefore, it is issued at ②. Even though T_{slat} is finished at ③, the user/kernel consume some computation cycles, and the $(i+1)^{th}$ request is not available. This in turn leads to T_{idle} by ④. Once the $(i+1)^{th}$ request is prepared by the upper layers, it can be served with its T_{cdel} and T_{sdev} . Note that, in addition to this kind of system delay, T_{idle} represents the time when a user or application does nothing. In this example of block request timings, the inter-arrival time, called T_{intt} , is defined by the time period between ② and ④.

B. Trace Revision

Even though existing block traces can cover different kinds of system configurations and various user scenarios, most publicly-available conventional block traces [1], [27], [9], [16], [11] were collected on HDD-based storage systems around a decade ago. Since then, however, storage systems have been dramatically changed, as modern servers start to adopt flash-based storage to boost performance and most server workloads significantly changed. Since the block traces are intensively studied and used for demonstrating the effectiveness and performance impacts of many system research proposals [31], [8],

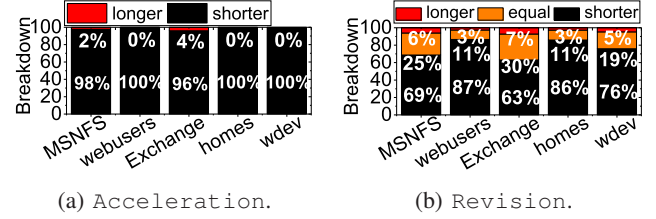


Fig. 3: Differences of inter-arrival times observed by reconstructed traces and real system traces.

[17], [23], these traces need to be mapped to new block traces by considering the new storage system characteristics.

Several approaches exist to reconstruct traditional block traces [25], [8], [30], [14], [32], [4]. First, the acceleration methods (Acceleration) [25], [8], [30] can artificially shorten inter-arrival times to compensate for the low throughput exhibited by HDD-based storage. However, since this method can only resize the inter-arrival times without considering the block request timings, it can remove critical information, such as T_{cdel} and T_{idle} from the traces. For example, if the average T_{intt} of a workload is 50 ms and the acceleration factor is 100, the reconstructed trace exhibits 500 us for its average T_{intt} . This removes the most T_{cdel} and T_{idle} , and can even make T_{sdev} unrealistic as there is no contexts for target device, system, and user behaviors. Instead of simply accelerating inter-arrival times, there is a (Revision) to revise target workloads by replaying the corresponding block traces on a real system [14], [32], [4]. While this would have more realistic T_{cdel} and T_{sdev} , it cannot appropriately accommodate T_{intt} , which varies across all I/O instructions in the trace.

To be precise, we also compare T_{intt} observed in the SSD-based system node (NEW) with the ones generated by Revision and Acceleration, respectively. The evaluation environment and scenario are the same as the test conditions described in Section I, and Acceleration leverages the acceleration degree that [8] uses. We examine different T_{intt} values by executing five open-license block traces (*MSNFS*, *webusers*, *Exchange*, *homes*, *wdev*), which are widely used in the storage community [24], and the results are shown in Figure 3. One can observe from Figure 3a that, 98.6% of T_{intt} reconstructed by Acceleration are shorter than actual T_{intt} observed in NEW. In contrast, as shown in Figure 3b, T_{intt} of Revision is on average 17.8% accurate (i.e., ‘equal’ in the figure). However, most of them (77.8% of total T_{intt} , on average) are shorter than the actual T_{intt} , which means it loses important system delays and user idle periods, T_{intt} . Note that it also exhibits a T_{intt} , that is on average 4.3% longer than the actual T_{intt} . This is because replaying traces drops the mode contexts (i.e., asynchronous/synchronous), which fails to capture the block request timing described by the $(i-1)^{th}$ request (as shown in Figure 2b). Since it is difficult to capture all system delays, idle operations and non-deterministic user behaviors (there is no block trace that offers all such information to the best of our knowledge), block trace reconstruction with limited information is non-trivial and challenging work.

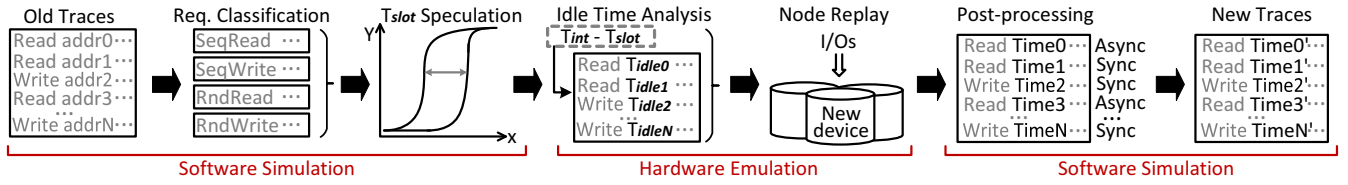


Fig. 4: High-level view of TraceTracker.

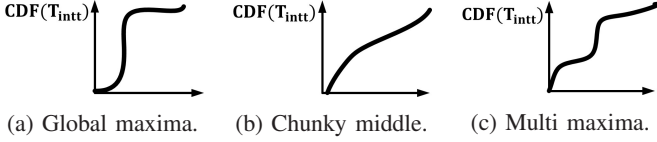


Fig. 5: Types of CDF distribution.

III. TIMING INFERENCE FOR I/O SUBSYSTEMS

Overview of TraceTracker. For individual I/O instructions, it is non-trivial to extract the idle time (i.e., T_{idle}) from old block traces since T_{idle} is affected by multiple unknown system parameters and indeterministic user behaviors at the time of trace collection. Even though the old block traces have no runtime information, including the user behaviors, T_{idle} can be inferred if we can estimate I/O subsystem latency (i.e., T_{slat}), which is composed of the channel delay (i.e., T_{cdel}) and storage device time (i.e., T_{sdev}). Generally speaking, T_{idle} can be simply obtained by subtracting T_{slat} from T_{intt} . Estimating T_{slat} would be relatively easy if most inter-arrival times (i.e., T_{intt}) are similar to each other, which can make the graph, that represents the CDF of T_{intt} (i.e., $CDF(T_{intt})$), steeper. As shown in Figure 5a, the graph rapidly rises in the middle, which exhibits a single maximum on its derivative. Because almost the entire range of $CDF(T_{intt})$ is in the middle of domain and not affected by its tail, T_{intt} at the global maximum of slope of CDF ($CDF(T_{intt})'$) can be considered as T_{slat} . However, there are many block traces whose $CDF(T_{intt})$ exhibits a much smoother slope on (e.g., chunky middle) and/or multiple maxima on its derivative. This is because T_{intt} of each instruction is affected by a different runtime contexts on the target system, which often makes them vary significantly. Considering Figure 5c as an example, this workload exhibits at least two maxima on $CDF(T_{intt})'$, which can render such simple differential analysis difficult to predict T_{slat} of the corresponding trace, appropriately.

Figure 4 summarizes the operation of our proposed TraceTracker. In this work, as shown in the left software simulation of Figure 4, we classify all the I/O instructions traced by a workload into multiple groups based on the request size and operation type. For the multiple groups, we create multiple CDFs for T_{intt} , and estimate the relative time costs of T_{sdev} and T_{cdel} in block request timings by taking into account the different request sizes and types. This relative time cost estimation, in turn, enables us to individually calculate T_{slat}^i for all i numbers of I/O instructions, thereby extracting T_{idle}^i from the target traditional block. Once we secure T_{idle} that varies based on user and system timing behaviors, T_{slat} can be re-evaluated by taking into account the target storage system. Specifically, we emulate the new system by regenerating the



Fig. 6: Finding the coefficients of T_{sdev} (β or η).

each request and issuing it on new storage with estimated T_{idle} . After the target trace emulation, we perform a simple post processing on the trace, which overrides the I/O timing behaviors for asynchronous mode operations by considering the old block trace and regenerated new trace. Further, while the inference logic of TraceTracker extracts the timing behaviors affected by non-deterministic user behaviors and unknown system parameters from the old block trace, the hardware emulation, and post processing parts mimic the system delay and user idle periods on a real system (target) to generate the new block trace.

Inference model. If there is no user idle period or system delay caused by the host-side software modules, T_{intt} can be similar to, or even the same as T_{slat} . In other words, if there is T_{intt} greater than T_{slat} , T_{idle} can be simply inferred by subtracting T_{slat} from T_{intt} . Even though the specific information captured by T_{slat} is also often not recorded and offered by the old block traces, in contrast to T_{idle} , it can be speculated by analyzing the distribution of T_{intt} . As described earlier, there is only one CDF of T_{intt} if all I/O instructions in the target workload exhibit a uniformed request size. This in turn allows us to simply speculate T_{slat} by referring to T_{intt} at the global maxima of $CDF(T_{intt})'$. In cases where the target workload exhibits a wide spectrum of request sizes and types, we speculate T_{slat} by inferring T_{sdev} and T_{cdel} , separately. Since T_{sdev} mainly depends on the underlying storage performance, we assume that T_{sdev} follows a linear model for the sequential accesses; T_{sdev} is inferred by $\beta * r_{size}$ if the type of the request is a read; otherwise, it is speculated by $\eta * r_{size}$. β and η are coefficient values, which will be explained shortly, and r_{size} denotes the size of a request. On the other hand, T_{sdev} on random accesses can be slightly longer than that of sequential accesses as it has a moving delay time, referred to as T_{movd} . T_{movd} typically captures the seek time and rotational latency of the underlying disk [21].

To model T_{movd} , we replay ten FIU workloads [27], [11] on an enterprise disk [29] and measure T_{movd} by calculating the difference between the T_{sdev}^{real} and T_{sdev}^{linear} , each of which is T_{sdev} observed on the real disk by executing random I/O accesses and generated by our linear models with sequential I/O accesses, respectively. We consider the difference between T_{sdev}^{real} and T_{sdev}^{linear} as T_{movd} , and the $CDF(T_{movd})$ results (for each workload) are plotted in Figure 7a. As shown in

the figure, each CDF exhibits a similar magnitude of gradient change with transition of T_{movd} . Motivated by this, we use T_{movd} at the maximum of $CDF(T_{movd})'$ as the representative of the difference between T_{sdev}^{real} and T_{sdev}^{linear} , which is referred to as T_{movd}^{rep} . Consequently, in this work, T_{sdev} for random reads and writes can be expressed by $\beta * r_{size} + T_{movd}^{rep}$ and $\eta * r_{size} + T_{movd}^{rep}$, respectively. Using this inference model, we speculate T_{sdev} , which in turn allow us to infer T_{slat} and T_{idle} . The specific estimation methods for each relative costs in block request timings are described below.

Decomposition of I/O subsystem latency. For each workload, the coefficients of T_{sdev} in our inference model, β and η , can be estimated by using the following disintegration analysis. First, we group all I/O instructions of the workload to reconstruct into three different categories based on i) sequentiality (e.g., sequential vs. random), ii) operation type (e.g., read vs. write) and iii) request size (in terms of sectors). We then create multiple graphs of $CDF(T_{intt})$ for each request size observed in each read or write with the sequential access pattern. The proposed inference model then examines the global maxima of $CDF(T_{intt})'$ for each CDF. Thus, there can be n maxima of $CDF(T_{intt})'$, where n is the number of different I/O request sizes observed in a target workload. It then chooses the two steepest graphs of CDF, which have the two highest magnitudes of T_{intt} changes among the maxima. Let us denote each of the steepest functions as $CDF_{steep1}(T_{intt})$ and $CDF_{steep2}(T_{intt})$, where $CDF_{steep1}(T_{intt})'$ is greater than $CDF_{steep2}(T_{intt})'$. As shown in Figure 6, we can drive $CDF(diff)$ which is CDF difference of $CDF_{steep1}(T_{intt})$ and $CDF_{steep2}(T_{intt})$ for reads and writes separately, and calculate the maximum at $CDF(diff)'$ for each.

We can then obtain the representative inter-arrival time at the maximum of $CDF(diff)'$ for reads and writes, which are referred to as ΔT_{intt}^{read} and ΔT_{intt}^{write} , respectively. Let us denote the two request sizes, which are used for creating $CDF_{steep1}(T_{intt})$ and $CDF_{steep2}(T_{intt})$, as $size_{r1}$ and $size_{r2}$, respectively. We can estimate the β and η by calculating $\Delta T_{intt}^{read} / |size_{r1} - size_{r2}|$ and $\Delta T_{intt}^{write} / |size_{r1} - size_{r2}|$, respectively. Let us denote the inter-arrival times at the maximum of $CDF_{steep1}(T_{intt})'$ for reads and writes as T_{intt}^{read} and T_{intt}^{write} , respectively. T_{intt}^{read} and T_{intt}^{write} are the best values that explain T_{slat} of the target workload since they exclude the most of the timing effects caused by system delays and user idle periods. Next, we can obtain T_{cdel}^{read} and T_{cdel}^{write} by calculating $T_{intt}^{read} - \beta * size_{r1}$ and $T_{intt}^{write} - \eta * size_{ref}$, where T_{cdel}^{read} and T_{cdel}^{write} are the channel delays for the reads and writes, respectively.

Figure 7b shows the actual T_{cdel} of the FIU workloads that we observed on the disk for each access pattern. One can observe from this figure that, while the difference of T_{cdel} between reads and writes exist to some extent (e.g., *ikki* and *maxmax*), that of T_{cdel} between random and sequential access patterns is not significant (less than 8% and 6%, respectively). Noting that the difference of T_{cdel} is less than T_{sdev} by many orders of magnitude; we believe that estimating the channel delay based the operation type of I/O requests is reasonable.

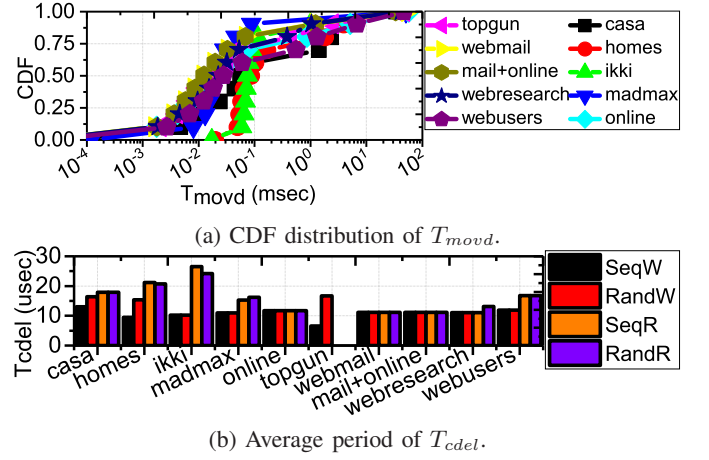


Fig. 7: The time components of T_{slat} (FIU).

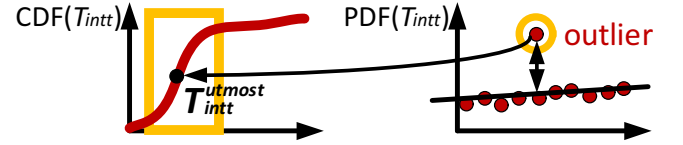


Fig. 8: Checking steepness of CDF distribution.

Lastly, to estimate the relative time cost of T_{movd} , we also need to find $CDF_{steep}^{rand}(T_{intt})$ that has the highest magnitude of gradient change with transition of T_{intt} among multiple CDFs in the group of random accesses, and estimate the inter-arrival time, T_{intt}^{rand} , at the maximum of $CDF_{steep}^{rand}(T_{intt})'$. Then, T_{movd} can be simply calculated by subtracting $T_{cdel}^{read} + \beta * size_{ref}$ (or $T_{cdel}^{write} + \eta * size_{ref}$) from the estimated T_{intt}^{rand} .

IV. IMPLEMENTATION FOR INFERENCE AUTOMATION

Analyzing multiple CDF graphs is important to reconstruct old traces. While categorizing requests based on their types and sizes can be easily automated, the autonomous analysis of CDFs is non-trivial due to their discreteness. In this section, we will detail the implementation of our proposed inference model and explain how to emulate traces with inferred system delays and idle periods (i.e., T_{intt}) to reconstruct the traces.

Graph classification. Since each block trace can exhibit multiple CDFs that are examined by the proposed inference model, it is time consuming to detect the two steepest graphs of $CDF(T_{intt})$, namely, $CDF_{steep1}(T_{intt})$ and $CDF_{steep2}(T_{intt})$, among them. When examining the graph, it would be possible to have a higher degree of the polynomial equation to represent CDF in mathematical expression, which also renders the process of finding $CDF_{steep1}(T_{intt})$ and $CDF_{steep2}(T_{intt})$ for the read and write instruction set of each trace difficult.

One simple but effective method to check the steepness of each graph is to analyze probability density distribution (PDF), instead of examining the derived function of $CDF(T_{intt})$ for a target trace. As shown in Figure 8, the CDF's highest magnitude of gradient change with a transition of T_{intt} can be

Algorithm 1 CDF steepness examination.

```
/** Step 1: Calculate PDF of inter-arrival times ( $T_{intt}$ )
1: for each  $T_i$  in  $T_{intt}$  do
2:    $PDF(T_i) := \text{num}(T_i) / \text{num}(\text{request})$ 
3: end for
/** Step 2: Least Square Regression
4: slope :=  $\text{std}(PDF(T_{intt})) / \text{std}(T_{intt})$ 
5: intercept :=  $\text{mean}(PDF(T_{intt})) - \text{slope} * \text{mean}(T_{intt})$ 
6:  $f(x) := \text{slope} * x + \text{intercept}$ 
/** Step 3: Find outliers
7: margin :=  $\text{var}(PDF(T_{intt})) / 2$ 
8: for each  $T_i$  in  $T_{intt}$  do
9:   distance :=  $PDF(T_i) - f(T_i)$ 
10:  if distance > margin then
11:    outliers.append( $PDF(T_i)$ )
12:  end if
13: end for
/** Step 4: Calculate CDF steepness
14:  $T_{intt}^{utmost} := \max(\text{outliers})$ 
15: steepness :=  $\text{distance}(f(T_{intt}^{utmost}), PDF(T_{intt}^{utmost}))$ 
```

obtained by identifying the utmost outlier on the corresponding PDF. Algorithm 1 outlines how to examine the steepness of the curve of the target CDF through the corresponding PDF. It first calculates the PDF of T_{intt} (cf. lines 1 ~ 3). After that, the algorithm finds the best-fitting straight line through a set of T_{intt} by using linear least squares regression analysis (cf. lines 4 ~ 6). In this algorithm, if there is T_{intt} , which is far from the best fitting straight line by more than a margin, we refer to T_{intt} as an outlier. Note that, as the margin increases, the number of outliers decreases. As the final goal of this PDF analysis is to find the utmost outlier, we setup the margin at half the variance. This PDF analysis algorithm visits all T_{intt} and collects the outliers for all categorized I/O instruction sets described in Section III (e.g., read/write and request sizes). Among the outliers, it first looks for the T_{intt} with the maximum value (denoted by T_{intt}^{utmost}) and returns the distance, which is the difference between the $f(x)$ value of the straight line at the utmost outlier and $PDF(T_{intt}^{utmost})$ (cf. line 14). Then, it compares the distances observed by each $PDF(T_{intt})$ and generates two graphs that have the top two highest T_{intt}^{utmost} values (cf. line 15).

Steepness analysis. It is a challenge to find the highest gradient change with a transition of T_{intt} by analyzing a group of I/O requests with their CDF. Since CDF of T_{intt} is a kind of non-differentiable function due to its discontinuity, the two I/O instruction groups selected by aforementioned graph classification algorithm must convert the discrete results into continuous results. While one can perform a curve fitting on $CDF(T_{intt})$ for the two groups to achieve a differentiable function, there is no perfect function that represents all variances observed in $CDF(T_{intt})$. To address challenge, we interpolate $CDF(T_{intt})$ with piecewise nonlinear curve fitting; two interpolation methods are widely used: i) a special type of piecewise polynomial (called *spline*) interpolation and ii) a piecewise cubic hermite interpolating polynomial (called *pchip*) interpolation. As shown in Figure 9, spline evaluates the coefficient for each interval of data and has two continuous derivatives, whereas pchip has just one derivative, which preserves shape more smoothly than spline. Among $CDF(T_{intt})$ for all old block traces we tested, pchip exhibits the desired appearance of smoothness without oscillation and

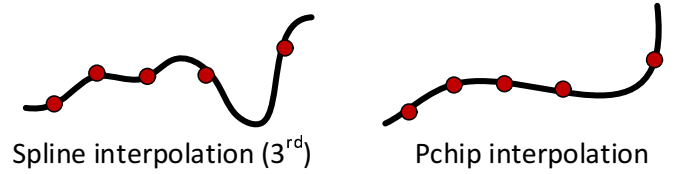


Fig. 9: Different types of interpolations that we tested.

under/overfitting issues that spline has. Once we interpolate $CDF(T_{intt})$ with pchip, we can differentiate the results of interpolation and find the maximum of the differential, which is the highest magnitude of gradient change with a transition of T_{intt} . Note that, the analysis of $CDF(T_{movd})$ described earlier can be processed by the same curve fitting and differential calculation methods applied to $CDF(T_{intt})$.

Hardware emulation and post-processing. Once the relative time costs are estimated, we can derive the T_{sdev} equation, which infers the different device times under the execution of sequential reads/writes and random reads/writes. In cases where there exist n numbers of I/O instruction traced in the target workload, we can denote the idle time, inter-arrival time and device latency of the i^{th} instruction (where $0 < i \leq n$) as T_{idle}^i , T_{intt}^i , and T_{sdev}^i , respectively. We then visit each I/O instruction of an old trace and perform the following trace reconstruction procedure. First, we check the operation type and request size of the old trace's instruction and estimate T_{sdev}^i using the T_{sdev} model (cf. Section III). We also calculate T_{intt}^i by checking the difference between time stamps of the i^{th} and $i + 1^{th}$ instructions, which are given by the old block trace. Thus, T_{idle}^i exists if T_{intt}^i is greater than T_{sdev}^i (e.g., $T_{idle}^i = T_{intt}^i - T_{sdev}^i$). We then delay T_{idle}^i using `sleep()` and issue the i^{th} I/O instruction (composed of the same information of the old block trace) to the underlying brand-new device. We iterate this process for all n I/O instructions. During this phase, we collect the new block trace using `blktrace`, which is a standard block trace tool in Linux [3]. While this hardware emulation mimics the user behaviors, including system delays and idle periods, and incorporates actual channel delays and device times on the real target system, it is not feasible to inject synchronous/asynchronous mode information to each I/O request. Thus, we check the old trace and record all the indices of the instruction whose T_{intt}^i is shorter than T_{sdev}^i . We then examine all the instructions of the new trace (but yet intermittent). In this post-processing, we subtract the new device time (measured by `blktrace`) from the corresponding inter-arrival time and update the next instruction based on the results, if the index of the instruction we are examining is in the range of instruction indices extracted by the old block trace. Note that if workloads provide the T_{sdev} information, we can skip the T_{sdev} inference phase, and immediately perform the hardware emulation and post processing after finding the short T_{intt} .

V. EXPERIMENTAL RESULTS

In this evaluation, we focus on answering the following questions: i) How accurate is our inference model? ii) How

Workload sets	Microsoft Production Server (MSPS)								FIU SRCMap							
Published year	2007								2008							
Workloads	24HR	24HRS	BS	CFS	DADS	DAP	DDR	MSNFS	ikki	madmax	online	topgun	webmail	casa	webresearch	webusers
# of block traces	18	18	96	36	48	48	24	36	20	20	20	20	20	20	28	28
Avg data size (KB)	8.27	28.79	20.73	9.71	28.66	74.42	24.78	10.71	4.64	4.11	4.00	3.87	4.00	4.04	4.00	4.20
Total size (GB)	21.2	178.6	331.2	43.6	44.6	84	44	317.9	25.4	3.8	22.8	9.4	31.2	80.4	13.7	33.6
Workload sets	FIU IODedup								MSR Cambridge (MSRC)							
Published year	2009								2008							
Workloads	mail+online	homes	mds	prn	proj	prxy	rsrch	src1	src2	stg	web	wdev	usr	hm	ts	
# of block traces	21	21	2	2	5	2	3	3	3	2	4	4	3	1	1	
Avg data size (KB)	4.0	5.23	33.0	15.4	29.6	8.6	8.4	35.7	40.9	26.2	7	34	38.65	15.16	9.0	
Total size (GB)	57.1	84.6	208.4	568.8	4780.1	4353	27.63	6516.5	230.6	226.4	625.4	23.7	5506.1	9.24	16.2	

TABLE I: Important characteristics of the publicly-available conventional block traces that we reconstructed.

realistic can our hardware/software make T_{intt} compared to conventional approaches? and iii) What are the system implications based on the revisions of T_{idle} ?

Evaluation node. For the target system where we reconstruct block traces, we build up a storage node that employs an all-flash array by grouping four NVM Express SSDs [7]. The storage capacity of each SSD is 400GB, and a single device consists of 18 channels, 36 dies, and 72 planes. Our storage node can exhibit different levels of parallelism, ranging from an array to channel, channel to die, which in turn can offer read and write bandwidths as much as 9GB/s and 4GB/s, respectively. Our all-flash array is connected to the node’s north-bridge via four PCIe 3.0 slots, each containing four lanes [20] to the storage node.

Target block traces. We reconstruct three workload categories: i) Florida International University (FIU) [27], [11], ii) Microsoft Production Server (MSPS) [9], and iii) Microsoft Research Cambridge (MSRC) [16]. Together, FIU, MSPS, and MSRC contain a total of 577 block traces, which are used for a wide spectrum of simulation-based studies [31], [8], [17], [23]. FIU workloads offer university-scale production server characteristics, which consist of two different types of sub-workloads: SRCMap and IODedup. While SRCMap workloads are collected for an application that optimizes system energy by virtualizing storage, department-level virtual machines for web services and mail, file and version control servers are collected by IODedup. On the other hand, MSPS provides eight different kinds of production server scenarios, and MSRC provides thirteen kinds of data center server scenarios. In MSRC, all workloads contain specific device-level information such as the type of RAID, while the same information for most workloads in MSPS is unknown. In addition, MSPS and MSRC workloads are collected by using an event-based kernel-level tracing facility [19] which can capture detailed information such as issue and completion time stamps; These timestamps are captured when requests are issued from a device driver to the target disk and when the disk completes the I/O operations, respectively. Note that even though all the traces related to the three categories of workloads discussed above include various system configurations and have a wide range of user scenarios, they are all collected around 2007~2009 on disk-based systems. The important characteristics of traces, including the size, and the number of traces per workload, are listed in Table I.

Reconstruction techniques. We evaluate five different block

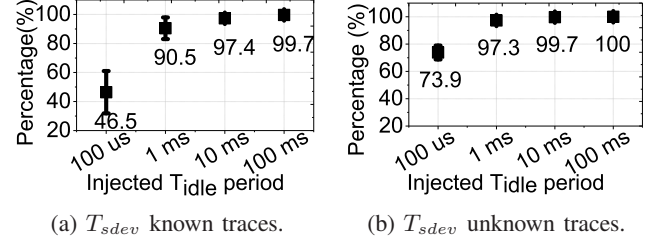


Fig. 10: Verification results, Len(TP).

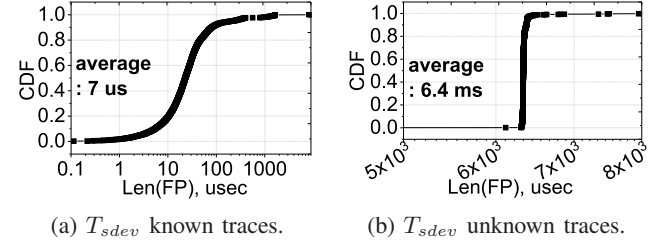


Fig. 11: Verification results, Len(FP).

reconstruction methods:

- **Acceleration:** Reconstruction by shortening T_{intt} [8].
- **Revision:** Replaying block traces on all-flash array [4].
- **Fixed-th:** An advanced revision method by inferring T_{idle} with a fixed threshold.
- **Dynamic:** Reconstructions using our inference model, but with no post-processing.
- **TraceTracker:** Hardware/software co-evaluation for trace reconstruction.

We leverage the value (i.e., 100) that a simulation-based SSD work uses [8] for its T_{intt} acceleration. On the other hand, Fixed-th considers the worst-case device latency of old storage with a fixed threshold value and uses it for inferring T_{idle} . To select a reasonable threshold, we performed a different set of evaluations on a HDD-based node with various thresholds, ranging from 10 ms to 100 ms, and selected 10 ms as Fixed-th’s optimal threshold. In contrast, Dynamic injects different T_{idle} per I/O instruction by speculating it over our inference model, but without the post-processing component of TraceTracker.

A. Verification

Metrics. The results of this verification evaluation can be either positive or negative, each of which may be true or false. If the inference model speculates that there is T_{idle} , it can be classified as positive, and otherwise, the result is negative. Being negative or positive can be tested per I/O instruction.

On the other hand, if the existence of T_{idle} is same in both target and reconstructed block traces, one can call this as true. Otherwise, it is false. Therefore, the results of the inference model test can be represented by four different statistics: true positive (TP), false positive (FP), false negative (FN), and true negative (TN). For verification, we will use four functions as follows: i) $Detection(TP) = \text{number of TP} / \text{number of } T_{idle}^{injected}$, ii) $Detection(FP) = \text{number of FP} / \text{total number of I/O instructions}$, iii) $Len(TP) = T_{idle}^{estimated} / T_{idle}^{injected}$, and iv) $Len(FP) = T_{idle}^{estimated}$, where $T_{idle}^{injected}$ and $T_{idle}^{estimated}$ are the idle times that were injected into the target block traces and speculated by our inference model, respectively. The first two functions capture the ratios of the number of TP/FP and the number of corresponding I/O instructions, whereas $Len(TP)$ and $Len(FP)$ indicate how much our inference model speculates accurate T_{idle} based on the result of a prediction hit or miss, respectively. Note that $Len(TP)$ is the ratio of the speculated idle periods and actual idle periods, whereas $Len(FP)$ is the actual period that the inference model mispredicts.

Results. Since the block traces have no information on T_{idle} , we inject T_{idle} in random places with various idle periods, ranging from 100 us to 100 ms. In this evaluation, injected T_{idle} accounts for 10% of the total I/O instructions of the target block traces. We compare the injected T_{idle} with the T_{idle} predicted by our inference model. We select two different groups of traces. One includes the traces that contain no timing information (e.g., FIU), and the other has I/O submission and completion time information (e.g., MSPS), which can be considered as T_{sdev} . In this evaluation, we denote the former and the latter as T_{sdev} known traces and T_{sdev} unknown traces, respectively.

Figure 10 shows the results of $Len(TP)$ observed by two trace groups that TraceTracker reconstructed. If the injected T_{idle} is longer than 1 ms, TraceTracker shows 90.5% and 97.3% accuracy of TP for T_{sdev} known traces and T_{sdev} unknown traces, respectively. If the injected T_{idle} is close to 100 us, the accuracy of TP declines compared to other cases by 46.5% and 73.9% for T_{sdev} known traces and T_{sdev} unknown traces, respectively. This is because the injected T_{idle} is in a range of the latency that new storage (in our case, Intel NVMe 750) exhibits. While this blurring boundary issue could make it difficult for our inference model to distinguish between device latency and idle time, most of the actual microsecond-scale system delays and idle periods are revived by our inference model. In addition, we observed that the results of $Detection(TP)$ is in the range of 82.2% ~ 99.7% across all the block traces that TraceTracker built. On the other hand, $Detection(FP)$ is, on average, 6% and 26% while $Len(FP)$ is on average 7 us and 6.4 ms for T_{sdev} known traces and T_{sdev} unknown traces, respectively. However, the distribution of $Len(FP)$ observed by the reconstructed traces tells a different story. As shown in Figure 11, more than 98% of $LenFP$ for T_{sdev} known traces and T_{sdev} unknown traces are shorter than 1 ms and 6 ms, respectively. Considering the high accuracy of TP and low impact of FP, we can conclude

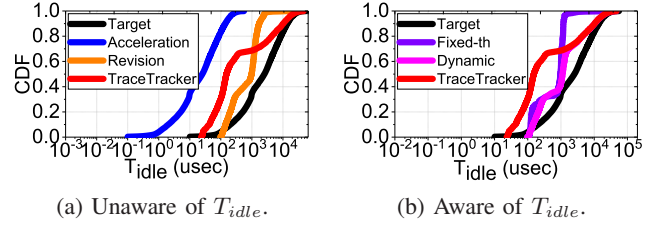


Fig. 12: CDF distribution of T_{intt} (MSNFS).

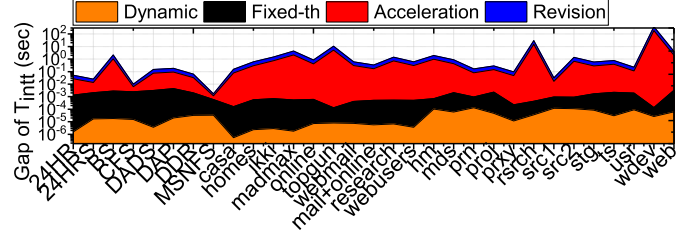


Fig. 13: T_{intt} differences among the different kinds of trace reconstruction techniques and TraceTracker method.

that TraceTracker is within the confidence interval to reconstruct T_{intt} .

Comparisons. In this section, we analyze the accuracy of TraceTracker compared to other reconstruction methods by inspecting the details of the T_{intt} . To this end, we compare TraceTracker's CDF of T_{intt} with two different groups of methods, each being unaware of and aware of T_{idle} ; the results are shown in Figures 12a and 12b, respectively. In these figures, Target shows, the CDF of T_{intt} brought by the original block traces collected on HDD-based nodes. One can observe from Figure 12a that, Acceleration just shifts the CDF of Target from the right to the left as much as the acceleration factor indicates (e.g., 100x), which eliminates all the useful information to simulate target systems. On the other hand, Revision reflects the characteristics of the underlying new storage. However, compared to TraceTracker, it removes T_{cdel} and T_{idle} by around 70% and 30%, respectively. As shown in Figure 12b, while Fixed-th and Dynamic behave more realistically than Acceleration, unfortunately, Fixed-th loses 65% of the T_{idle} and Dynamic exhibits 30% longer T_{intt} than TraceTracker as it also loses asynchronous/synchronous mode information and is unable to capture T_{cdel} appropriately.

Figure 13 plots the average difference between TraceTracker and other trace reconstruction methods in terms of T_{intt} for all the workloads we tested. One can observe from the figure that Acceleration and Revision methods that possess no information for T_{idle} to reconstruct traces differ by 7.08 and 7.15 seconds from TraceTracker, respectively. Considering the worst-case latency of the underlying SSD accesses (around 2 ms), losing such idle times that include system delays and user behaviors can have a great impact on diverse simulation-based studies. While Fixed-th and Dynamic have less T_{intt} differences compared to Acceleration or Revision, the difference between TraceTracker and their T_{intt} is as high as 1.3 ms and 0.035 ms, respectively. This means that, even though

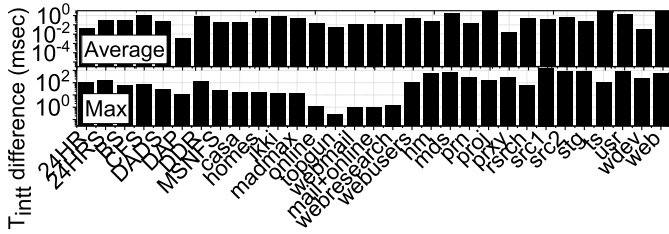


Fig. 14: T_{intt} differences between the target block traces and TraceTracker traces.

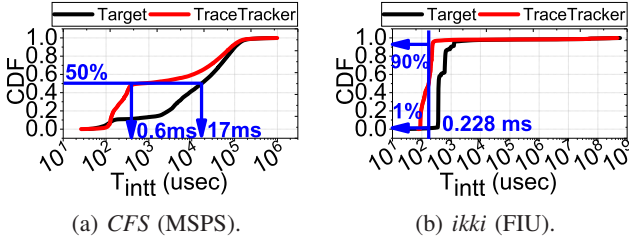


Fig. 15: Distribution differences between the target block traces and `TraceTracker` traces.

Fixed-th and Dynamic can capture the underlying storage characteristics, the actual time behaviors, including T_{cdel} and T_{idle} , are omitted. As a result, they can exhibit different system behaviors with inaccurate T_{intt} values.

B. System implications

Overall analysis of inter-arrival times. The top and bottom of Figure 14 plot the average and maximum T_{intt} differences between the target block traces and traces reconstructed by TraceTracker. As shown in the figure, T_{intt} of the TraceTracker traces is 0.677 ms shorter, on average, than that of the target block traces. The T_{intt} implies that system analysis and evaluation studies that use the T_{intt} of target block traces should consider the TraceTracker traces instead since the time budget to perform foreground/background tasks can be tightened when the storage system is changed. For example, the *ts* workload (MSRC) has an average of 3 ms shorter T_{intt} in TraceTracker traces than in the target block traces. In addition, the median values of T_{intt} are 2 ms and 0.02 ms for target block traces and TraceTracker, respectively. Note that, the average T_{intt} differs among the 31 workloads because of the impact of the specific workload characteristics such as request size and type.

To analyze T_{intt} differences between the two traces in detail, we plot the CDF distribution of T_{intt} , as shown in Figure 15, only for the *CFS* (MSPS) and *ikki* (FIU), which have the maximum T_{intt} differences among the same workload categories (MSPS, FIU). As shown in the figures, the T_{intt} distribution of the `TraceTracker` traces leans towards the short time period and the average differences are 1 ms and 0.823 ms, respectively. For instance, 50 % of T_{intt} in the target block traces are less than 17 ms while that of the `TraceTracker` is 0.601 ms in Figure 15a. In addition, as shown in Figure 15b, 1 % of T_{intt} in the target block traces



Fig. 16: Average time period of T_{idle} .

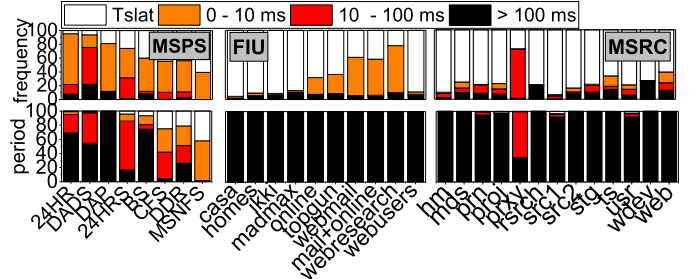


Fig. 17: Breakdown of T_{idle} .

are less than 0.228 ms, while 90 % of the T_{intt} are less than the same value in the TraceTracker traces.

Details of idle times. T_{idle} can be a representative workload characteristic, and the estimated T_{idle} was injected when the traces are reconstructed on the target storage system. Since the T_{idle} periods should be same in reconstructed traces, the T_{idle} that we estimated can be immediately used for other conventional block traces. Figure 16 shows the T_{idle} period estimated by TraceTracker. As shown in the figure, the average T_{idle} of MSPS is 0.27 s, and that of FIU is 2.80 s remove *madmax* workload has 20.5 s of longest T_{idle} among the FIU workloads. MSRC has an average T_{idle} value of 2.25 s, except for *rsrch* and *wdev* which have 69.2 s and 403.1 s of T_{idle} , respectively.

To check the detailed T_{idle} patterns of the 31 workloads, we analyze the breakdown of total T_{intt} duration by grouping these into T_{slat} , T_{idle} ($0 \sim 10$ ms), T_{idle} ($10 \sim 100$ ms), and T_{idle} (longer than 100 ms). The top and bottom parts of Figure 17 focus on the frequency and period, respectively. The frequency refers to the total number of requests per group while the period means the total time duration of each group. As shown in the figure, the MSPS workloads have larger T_{idle} breakdown in terms of frequency, compared to other workloads; the average T_{idle} breakdown is 70%, 31%, and 26% for MSPS, FIU, and MSRC, respectively. In contrast to the frequency, the average breakdown of period per workload categories is 87%, 99.8%, and 99.2% for MSPS, FIU, and MSRC, respectively. In other words, although the FIU and the MSRC workloads have low T_{idle} frequency, most of the T_{intt} is T_{idle} at around 90%. In addition, as shown in the figure, most of the T_{idle} is longer than 100 ms in the FIU and MSRC workloads. Similar to the average T_{idle} period shown in Figure 16, the breakdown pattern of MSPS workloads varies compared to other workloads. In the MSPS, the average frequency is 30%, 47.7%, 15%, and 6.7% for each group, while the average period breakdown is 12.6%, 18.3%, 26%, and 42.7%, respectively. Since the MSPS workloads have

short T_{idle} , it is harder for them to utilize inter-arrival times, compared to other workloads.

VI. RELATED WORKS

There exist many prior studies that proposed to modify the conventional block traces to adjust to new storage systems [8], [25], [30], [4], [5], [14], [32]. For example, [8], [25], [30] tried to simply accelerate the inter-arrival times with a fixed scaling factor. On the other hand, [4], [5], [14], [32] replayed I/O requests on the real storage system by injecting an extra delay or zero (no-idle) between two consecutive requests. In contrast, [14] was aware of the behaviors of parallel applications, which are widely used in scientific or business environments, and reflected these onto target block traces by injecting different idle times per I/O instruction. As there are multiple nodes that execute parallel applications, this work calculates the duration of an extra delay by taking into account the computing time and synchronization time which is required for each node to ensure data synchronization; the input data of one node is another node's output. While all the above methods did not consider user behaviors and I/O execution mode, TraceTracker can reconstruct the old block traces irrespective of application types and can classify the inter-arrival times into I/O subsystem latency and extra delay (idle times) by including both system and user behaviors. In TraceTracker, the idle times are decided by modeling the performance of the target trace's storage system.

Early studies on storage performance modeling [12], [13], [22], [26], [28] try to capture the performance of new storage systems by identifying the target workload's characteristics. For example, [28] used Classification and Regression Trees (CART), which is learning-based black box modeling technique. However, CART does not understand the input features and generates a multidimensional function of the model. Thus, for the storage performance, [28] utilized the request information (e.g., inter-arrival times, logical block number, request type, and data size) as features of the CART algorithm. Unfortunately, the main problem of machine-learning based modeling is that it is hard to explain how the model can be achieved. While [28] creates performance model without understanding the inter-arrival times, our TraceTracker analytically models storage performance by decomposing the inter-arrival times and detecting the short inter-arrival times for asynchronous I/O execution.

VII. ACKNOWLEDGEMENT

This research is mainly supported by NRF 2016R1C1B2015312. This work is also supported in part by DOE DE-AC02-05CH 11231, IITP-2017-2017-0-01015, NRF-2015M3C4A7065645, and MemRay grant (2015-11-1731). Kandemir is supported in part by NSF grants 1439021, 1439057, 1409095, 1626251, 1629915, 1629129 and 1526750, and a grant from Intel. Myoungsoo Jung is the corresponding author.

VIII. CONCLUSION

TraceTracker is a new approach to reconstruct existing block traces to new traces which is being aware of the target storage system only with inter-arrival time information of target workloads. To maintain the important workload's characteristics such as system and user behaviors in the new traces, TraceTracker estimates the idle times by automatically inferencing the performance of storage system from target block traces. We can detect 99% of system delays and idle periods appropriately and secure the corresponding idle periods by 96% of a real execution, on average.

REFERENCES

- [1] O. Application, "I/o. umass trace repository," *Technical report*, [Http://traces.cs.umass.edu](http://traces.cs.umass.edu), 2007.
- [2] J. Axboe, "Flexible i/o tester," *Freshmeat project website*, 2011.
- [3] A. D. Brunelle, "Block i/o layer tracing: blktrace," 2006.
- [4] S. Chen, A. Ailamaki, M. Athanassoulis, P. B. Gibbons, R. Johnson, I. Pandis, and R. Stoica, "Tpc-e vs. tpc-c: Characterizing the new tpc-e benchmark via an i/o comparison study," *ACM SIGMOD Record*, 2011.
- [5] G. R. Ganger and Y. N. Patt, "Using system-level models to evaluate i/o subsystem designs," *IEEE TC*, 1998.
- [6] K. Harty and D. R. Cheriton, "Application-controlled physical memory using external page-cache management," *ACM ASPLOS*, 1992.
- [7] Intel, "Intel ssd 750 series," *URL: http://www.intel.com/content/www/us/en/solid-state-drives/*, 2015.
- [8] J. Jeong, S. S. Hahn, S. Lee, and J. Kim, "Lifetime improvement of nand flash-based storage systems using dynamic program and erase scaling," in *FAST*, 2014.
- [9] S. Kavalanekar, B. Worthington, Q. Zhang, and V. Sharda, "Characterization of storage workload traces from production windows servers," in *IISWC*, 2008.
- [10] Y. Kim, R. Gunasekaran, G. M. Shipman, D. A. Dillow, Z. Zhang, and B. W. Settemyer, "Workload characterization of a leadership class storage cluster," in *PDSW*, 2010.
- [11] R. Koller and R. Rangaswami, "I/o deduplication: Utilizing content similarity to improve i/o performance," *TOS*, 2010.
- [12] A. Merchant and P. S. Yu, "Analytic modeling of clustered raid with mapping based on nearly random permutation," *TC*, 1996.
- [13] M. Mesnier, M. Wachs, B. Salmon, and G. R. Ganger, "Relative fitness models for storage," *SIGMETRICS*, 2006.
- [14] M. P. Mesnier, M. Wachs, R. R. Simbasivan, J. Lopez, J. Hendricks, G. R. Ganger, and D. R. O'hallaron, "trace: parallel trace replay with approximate causal events," in *FAST*, 2007.
- [15] V. Mohan, T. Siddiqua, S. Gurumurthi, and M. R. Stan, "How i learned to stop worrying and love flash endurance," in *HotStorage*, 2010.
- [16] D. Narayanan, A. Donnelly, and A. Rowstron, "Write off-loading: Practical power management for enterprise storage," *TOS*, 2008.
- [17] D. Narayanan, A. Donnelly, E. Thereska, S. Elnikety, and A. Rowstron, "Migrating server storage to ssds: analysis of tradeoffs," in *EuroSys*, 2009.
- [18] D. Narayanan, A. Donnelly, E. Thereska, S. Elnikety, and A. I. Rowstron, "Everest: Scaling down peak loads through i/o off-loading," in *OSDI*, 2008.
- [19] I. Park and M. K. Raghuraman, "Server diagnosis using request tracking," in *DSN*, 2003.
- [20] PCI-SIG, "Pci express base 3.0 specification," *Addison-Wesley Publishing Company*, 2010.
- [21] C. Ruemmler and J. Wilkes, "An introduction to disk drive modeling," *Computer*, 1994.
- [22] E. Shriver, A. Merchant, and J. Wilkes, "An analytic behavior model for disk drives with readahead caches and request reordering," in *SIGMETRICS*, 1998.
- [23] G. Soundararajan, V. Prabhakaran, M. Balakrishnan, and T. Wobber, "Extending ssd lifetimes with disk-based write caches," in *FAST*, 2010.

- [24] Storage Networking Industry Association, “Snia trace repository.”
- [25] B. Trushkowsky, P. Bodík, A. Fox, M. J. Franklin, M. I. Jordan, and D. A. Patterson, “The scads director: Scaling a distributed storage system under stringent performance requirements.” in *FAST*, 2011.
- [26] M. Uysal, G. A. Alvarez, and A. Merchant, “A modular, analytical throughput model for modern disk arrays,” in *MOSCOTS*, 2001.
- [27] A. Verma, R. Koller, L. Useche, and R. Rangaswami, “Srcmap: Energy proportional storage using dynamic consolidation.” in *FAST*, 2010.
- [28] M. Wang, K. Au, A. Ailamaki, A. Brockwell, C. Faloutsos, and G. R. Ganger, “Storage device performance prediction with cart models,” in *MASCOTS*, 2004.
- [29] WD, “Western digital (wd) blue,” URL:<https://www.wdc.com/wd-blue-pc-desktop-hard-drive.html>, 2011.
- [30] C. Weddle, M. Oldham, J. Qian, A.-I. A. Wang, P. Reiher, and G. Kuenning, “Paraid: A gear-shifting power-aware raid,” *TOS*, 2007.
- [31] Y. Zhang, G. Soundararajan, M. W. Storer, L. N. Bairavasundaram, S. Subbiah, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, “Warming up storage-level caches with bonfire,” in *FAST*, 2013.
- [32] N. Zhu, J. Chen, T.-C. Chiueh, and D. Ellard, “Tbbt: scalable and accurate trace replay for file server evaluation,” in *SIGMETRICS*, 2005.