DiskAccel: Accelerating Disk-Based Experiments by Representative Sampling

Mojtaba Tarihi*
tarihi@ce.sharif.edu

Hossein Asadi*†
asadi@sharif.edu

Hamid Sarbazi-Azad*‡
azad@sharif.edu

* Department of Computer Engineering, Sharif University of Technology, Tehran, Iran
† ICT Innovation Center, Sharif University of Technology, Tehran, Iran
‡ Institute for Research in Fundamental Sciences, Tehran, Iran

ABSTRACT

Disk traces are typically used to analyze real-life workloads and for replay-based evaluations. This approach benefits from capturing important details such as varying behavior patterns, bursty activity, and diurnal patterns of system activity, which are often missing from the behavior of workload synthesis tools. However, accurate capture of such details requires recording traces containing long durations of system activity, which are difficult to use for replay-based evaluation. One way of solving the problem of long storage trace duration is the use of disk simulators. While publicly available disk simulators can greatly accelerate experiments, they have not kept up with technological innovations in the field. The variety, complexity, and opaque nature of storage hardware make it very difficult to implement accurate simulators. The alternative, replaying the whole traces on real hardware, suffers from either long run-time or required manual reduction of experimental time, potentially at the cost of reduced accuracy. On the other hand, burstiness, autocorrelation, and complex spatio-temporal properties of storage workloads make the known methods of sampling workload traces less effective.

In this paper, we present a methodology called DiskAccel to efficiently select key intervals of a trace as representatives and to replay them to estimate the response time of the whole workload. Our methodology extracts a variety of spatial and temporal features from each interval and uses efficient data mining techniques to select the representative intervals. To verify the proposed methodology, we have implemented a tool capable of running whole traces or selective intervals on real hardware, warming up hardware state in an accelerated manner, and emulating request causality while minimizing request inter-arrival time error. Based on our experiments, DiskAccel manages to speed up disk replay by more than two orders of magnitude, while keeping average estimation error at 7.6%.

SIGMETRICS'15, June 15–19, 2015, Portland, OR, USA.
Copyright © 2015 ACM 978-1-4503-3486-0/15/06 ...$15.00.
http://dx.doi.org/10.1145/2745844.2745856 .

1. INTRODUCTION

Storage devices are a key component of any computing environment for intermediate and long-term data storage. Apart from supporting the required capacity, they must meet performance demands at an acceptable cost. As a result, storage devices have long been the subject of many performance optimizations such as prefetching [8], request scheduling [57], caching [47, 23], layout optimizations [33], and hybrid storage [23, 37].

An integral part of designing a high-performance storage device is evaluation of algorithms, cache configurations, and/or other optimization techniques under study. Evaluating such design refinements requires accurate experimentation, which is often very time-consuming. Software-based simulators [6] provide repeatable and closely observable experimental setups without the need for real hardware. However, storage devices are opaque and contain many sophisticated algorithms. Although methodologies exist to automatically extract new device parameters [40], serious issues remain unresolved as shown by our previous experimental studies [49]. In particular, while storage simulators are generally sufficient for preliminary studies of algorithms, they in no way can be used for complete evaluation of algorithms and as such, use of real hardware is essential.

Input workloads, which should be replayed on real hardware, have a strong impact on experiments. These workloads either come from real machines or are generated by benchmarks and synthesis tools. Parameterizable software benchmarks are a popular source of evaluation workloads [20, 30, 9], but are not very representative of real-life workloads. Studies presented in [16, 21] show that while actual workloads are very bursty and exhibit diurnal patterns, widely used benchmarks cannot be configured for these properties. Worse yet, many benchmarks cannot even accurately control inter-arrival times despite its significant influence on performance [3, 27]. Benchmarks results are also influenced by host hardware; a faster processor can decrease I/O inter-arrival time and larger memory can greatly improve operating system caching.

Another obstacle for repeated experiments which can greatly hinder hardware-based experiments is the duration of storage traces ranging from hours to days and weeks. Previous studies have resorted to run traces as fast as possible [50], speed up inter-arrival time [51, 39], manually select portions of the trace [61, 14, 57], and manual selection along with speed-ups [60]. These methods either lack reproducibility or do not preserve key parameters of trace behavior.

Long experimentation times also hinder microprocessor simulations. To alleviate this issue, two main schemes have been proposed which reduce workload size [13, 58]. These methods select certain workload intervals and use their measurements to project the performance of the whole workload. They also keep the state up to date by use of checkpoints or warm-ups. As a result, these schemes can provide accurate estimations while running only a fraction of the original workload which would have taken days or weeks to simulate in full. For I/O workloads, clustering was used in [15] to enhance trace synthesis. In [44], clustering was used to create a classification decision tree for use in controllers of *Solid-State Drives* (SSDs). CART models used in [53] can be used for performance projection and tree-based workload trimming. To the best of our knowledge, none of these works focus on accurate and replay-based reduction of I/O workloads.

In this paper, we present a methodology called DiskAccel that greatly accelerates trace-based storage evaluation on real hardware. DiskAccel employs traces recorded from real servers and uses a variety of properties to select a representative subset of the workload. This process does not use performance numbers obtained from other hardware devices so that the selection becomes independent of hardware. Although this methodology can also be employed with software simulators, due to concerns with accuracy issues noted above, we focus on using real hardware to maximize accuracy. As part of DiskAccel, we have developed a tool to replay the representative intervals on target hardware. Our tool avoids many pitfalls of trace-based experiments by (a) warm-up of hardware cache by replaying an appropriate number of preceding requests, (b) enforcing inter-request dependency by constructing dependency trees, and (c) accurate control of request arrival time by avoiding scheduling delays.

The process of selecting representative intervals divides each trace into multiple intervals, extracts a variety of features from each interval, and then performs clustering. The features used include address and time entropy, request randomness, seek time, and working set locality. We use a weighted variant of the K-Means clustering algorithm similar to [13], and select the point closest to the cluster centroid as the cluster representative. To perform experiments based on this sampling methodology, we have developed a tool to run the representative intervals with timing accuracy and disk state warm-up on real hardware[1]. To increase timing accuracy, we use busy-waiting and real-time Linux facilities [34] to minimize request inter-arrival time error. Warm-up selects requests leading up to the representative intervals such that their footprint is equal to the cache capacity of the target hardware. These requests are then run in an accelerated manner before running the representative interval. We also try to replicate the request dependency by controlling request arrival times in a closed-loop manner. More specifically, we treat read requests as being blocking and consider writes to be non-blocking. This heuristic is essential as almost every publicly existing traces lack essential high level information such as request flags and issuing thread and process identification (ID). To the best of our knowledge, this is the first attempt to present a methodology to accelerate replay-based I/O experiments while keeping error rates at a minimum. In addition, this is the first work which presents efficient warm-up technique for disk drives to speed up I/O experiments.

[1]This tool is available at: http://dsn.ce.sharif.edu/software

To validate DiskAccel, we use the *Microsoft Research Cambridge* (MSRC) workloads [28] to test our methodology against a wide variety of workloads. We perform clustering based on the extracted features and validate our methodology in two steps. First, we use the performance numbers embedded in the traces to compare the whole trace average response time against the weighted response time. This yields an average relative error of 3.6% error for all traces in [28] and a worst-case error of 12.1%. In the experiments, we perform full runs on 11 of the selected traces (each spanning approximately one week) with no acceleration on our hardware testbed to serve as a reference. We then compare the relative error of choosing representative intervals on these traces with the new performance results. This yields an average error of 6.5% and a maximum error of 13.6%. Finally, we perform partial runs on our hardware testbed with an average speed-up of 577x versus full one-week runs and compare the weighted response time against the average response time of the numbers yielded from full one-week runs. This yields an average and maximum relative errors of 7.6% and 17.2%, respectively.

In the following, we discuss the related work in Sec. 2. Our experimental methodology is explained in Sec. 3. Then, we present our methodology of reducing replay time in Sec. 4. Details of methodology for replaying traces is covered in Sec. 5. Next, we validate our methodology on a completely different hardware setup in Sec. 6. We conclude the paper in Sec. 7 and discuss the limitations and future directions of our work.

2. RELATED WORK

In this section, we first discuss previous methodologies in synthesis of storage workloads and point out their deficiencies. We then cover trace reduction methodologies used on storage and microprocessor traces and evaluate their usability in storage workloads. Finally, we discuss the use of real disks in storage benchmarking and highlight the important measures that must be taken to ensure high accuracy of performance measurements.

2.1 Trace Synthesis and Benchmarks

Storage traces capture a record of I/O requests sent to the storage subsystem over a period of time. In some scenarios, however, it is essential to scale a workload to the desired hardware configuration. This practice is particularly useful when measuring the maximum potential of underlying hardware or when trying to optimize software or storage subsystem parameters. This is where benchmarks excel the most as the users are able to tune various options and settings.

Benchmark programs attempt to generate a workload similar to a certain application. Benchmark tools such as Postmark [20], Bonnie++ [9], and Iozone [30] are publicly available and have been widely used to evaluate storage algorithms and architectures. However, I/O benchmarks have difficulties in replicating burstiness and auto-correlation [50] despite these behaviors being exhibited widely by I/O workloads [32, 21, 22]. It should also be noted that our proposed methodology can also be used on traces generated by such benchmarks to accelerate experiments.

Trace synthesis programs try to replicate key properties of workloads to yield a portable and tunable replacement for traces. Such approaches include synthesizing traces using Binomial distribution [54, 14, 15] and multi-distribution fits [59] to approaches based on spatio-temporal entropy [52]. Trace shuffling and piece-wise synthesis of traces in [14] demonstrates that although long-range dependence is present, the use of sufficiently large time intervals diminishes its impact. Hence, short-range dependency can represent the inter-arrival times of each interval.

2.2 Trace Reduction

While real-world traces contain key spatio-temporal properties, their long duration must be addressed appropriately. In this section, previous works on reduction and compaction of storage and microprocessor traces will be covered.

Microprocessor simulations are essential for architecture explorations. While various software programs exist to simulate processor behavior at the required level, they tend to be as far as five- to six-orders of magnitude slower than real hardware [58]. This, coupled with a large volume of multi-billion instruction traces makes it very expensive to investigate architectural ideas without resorting to trace size reduction.

Two studies are well-known on the subject of sampling CPU traces, SMARTS [58], which performs systematic sampling of intervals for detailed simulation, and SimPoint [13], which uses per-interval feature extraction and clustering to pick representative intervals. Both studies report massive time savings with negligible error.

Systematic sampling in [58] relies upon assuming a normal distribution of response time in intervals. This expectation can easily be satisfied through the Central Limit Theorem, assuming intervals are numerous and independent. While this assumption does not hold for individual I/O requests, it does hold for request intervals as pointed out by [14]. In Sec. 4, we perform a viability study for using a similar solution for I/O traces.

SimPoint [13] uses clustering algorithms to group similar intervals and select representatives. In earlier releases of this work, intervals had a fixed number of instructions. Later on, variable length intervals were supported by common sequence matching. SimPoint takes advantage of availability of the binaries generating the trace to form much more descriptive and accurate feature vectors based on the code structure. In [24], different feature vectors for this tool were evaluated, of note is the use of accessed memory addresses as the sole source of features which yields poor results. Clustering is performed by a modified version of K-Means which supports weighting different points belonging to each interval. As K-Means needs K, the number of clusters, to be specified, *Bayesian Information Criterion* (BIC) [42] was used to choose the best clustering resulted by different values of K.

The study presented in [14] divides traces into fixed-duration intervals and shows that such divisions can be used and synthesized independent of the history with little impact on accuracy. This study was later used by [15] which extracts temporal information from these intervals and then performs clustering to pick representative intervals as the subject of synthesis. In [44], clustering was used as a step to find out which features can better distinguish key behavior classes. The features used are mostly spatial and the resulting classes specify the main behaviors exhibited, e.g., fully sequential, part-sequential, and part-random. The evaluation for clustering was a goodness of fit measure and the final result is a decision tree which can be used in an SSD controller to optimize for these behaviors. The authors found out that out of the different clustering algorithms, K-Means is the best choice.

A variety of features based on spatial and temporal properties of traces were extracted by [48] for each interval. Interval features were then subjected to a process termed deduplication by the authors to reduce the number of per-interval property sets. These properties were then used to generate benchmark configuration parameters to produce a behavior similar to that of the source trace.

2.3 Replaying I/O Traces

Workload traces offer an opportunity to evaluate hardware devices and algorithms in real-life scenarios. However, proper care must be taken to ensure that meaningful performance results are obtained.

Storage traces record I/O activity from computers where requests often depend on each other. While performing experiments, open-loop as opposed to closed-loop control of requests greatly affects experimental results [41]. Tools such as blktrace/blkparse [5] and fio [4] can perform closed-loop control in a multi-threaded manner provided that this information has been recorded in the trace. TBBT [60] and ROOT [55] try to infer inter-request relationship based on the higher level semantics of the syscalls and *Network File System* (NFS) layers, respectively. Both //TRACE [26] interferes with program activity to discover request dependencies. We cover this subject in Sec. 5.1.

When a certain inter-arrival time is chosen, it must be enforced with high accuracy. Anderson et al. [3] studied the effect of inter-arrival error or drift time on measurement accuracy from trace replay. They subsequently developed a replay tool which kept drift time to a maximum of $100\mu s$.

Another key detail that can affect trace replay results is the state of the storage subsystem, including disks and controller-level cache. To ensure that cache contents are valid for experiments, appropriate warm-up is necessary. For example, the authors in [25, 56] used a portion of a trace to perform warm-up. The latter also tried to match the warm-up of synthetic workloads to the size of hardware state. This will be further discussed in Sec. 5.2.

3. EXPERIMENTAL SETUP

Performance Measure. The performance measure used in this paper is the average request response time which is the time spent between sending a request to the I/O subsystem and receiving operation acknowledgement. Should the operation be blocking, response time indicates the amount of processing time wasted waiting for I/O and a major performance bottleneck for many use cases. In other words, we compare the reference average response time against response time projected by our subset of the workload to measure the accuracy of our methodology. We use *Weighted Response Time* (WRT) calculation to project performance of the whole workload. To compute WRT, as shown in Equation 1, we weight the selected representative intervals by the total number of requests represented by that interval. If we use WRT on the whole workload and assume each interval to only represent itself, the result is the reference average response time. Weighting emphasizes dense intervals over sparse intervals for calculating average request times. There

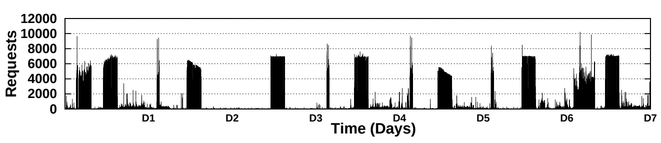

Figure 1: Average per-minute requests of usr_1 trace from [28] (Note the diurnal patterns).

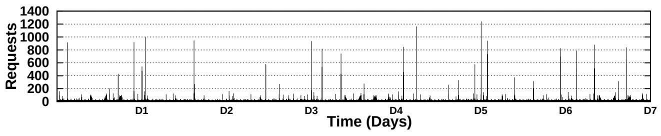

Figure 2: Average per-minute requests of rsrch_0 trace from [28] (Non-diurnal, very bursty, and having no well-defined patterns).

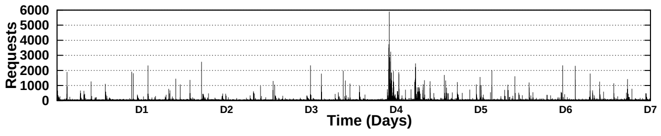

Figure 3: Average per-minute requests of hm_0 trace from [28] (Bursty, activity peaks on a certain day).

are two benefits to this choice. First, dense intervals contain more requests and represent more user activity. Second, the extra strain they put onto the hardware further increases the response time, affecting I/O performance even more [29].

$$RT_{average} = \frac{\sum\limits_{I \in Intervals} RT_{interval}(I) \times Count_{interval}(I)}{\sum\limits_{I \in Intervals} Count_{interval}(I)} \tag{1}$$

Workloads. We used 36 traces, referred to as the *Microsoft Research Cambridge* (MSRC) traces [28], which have been recorded from a variety of servers used for research, web development, terminal services, printing, hardware monitoring, and web proxy services. Each trace is approximately a week long and records requests to a single volume. These traces contain from a few hundred requests to well over a hundred million requests. Burstiness is present throughout and the long duration has allowed them to demonstrate diurnal and weekly patterns. This duration, however, is a serious challenge for replay on real hardware. We show the average per-minute requests for *usr_1*, *rsrch_0*, and *hm_0* in Figure 1, Figure 2, and Figure 3, respectively. It can be seen that these workloads are bursty and while *usr_1* (hosting user data) shows a clear diurnal pattern, the other two are much less predictable. In case of the *hm_0* workload which is used for hardware monitoring, the activity peaks on a single day. These three traces demonstrate the difficulty of representative tracing. To record diurnal and bursty patterns, it is necessary to record long periods of activity but such long traces make replaying difficult.

Testbed. We initially use the same response time numbers embedded on the original traces to test the first part of our methodology, but to fully validate our findings we developed a tool to (a) perform whole one-week runs of a selected number of traces to serve as new references, and (b) perform partial runs including only the representative intervals and compare their WRT against the newly obtained reference numbers. We replay the traces on identical 3.5” 5400RPM 500GB Seagate Momentus Disks of the same batch. We took a number of steps to prevent interferences with our experiments, (a) no file systems were mounted on the target drives and they were directly accessed via their device nodes (/dev/sdXX), (b) device nodes are opened with the O_DIRECT flag to prevent kernel caching on target drives, (c) we used the *RT_PREEMPT* real-time capability patch for Linux [34] and set top priority to the program, and (d) recorded trace data in memory and performed processing beforehand to prevent I/O blocking. The requests are sent with the Linux’s Native Asynchronous I/O [12] which allows us to get the exact request finish times while being able to have multiple requests in flight. These enhancements, however, are not sufficient to minimize drift time and similar to [3] we had to use busy-waiting. This is because *usleep* and *nanosleep* calls must be avoided due to serious inaccuracies [36]. Switching to busy-waiting reduced average drift time by hundreds of microseconds even when the real-time patch was active.

Evaluation Steps. DiskAccel workflow can be seen in Figure 4. DiskAccel selects representative intervals via clustering and then runs them on a system. We took the following steps for evaluating DiskAccel:

1. First, we evaluate the accuracy of DiskAccel at picking representatives (Step 1 in Figure 5). We compare WRT of representative intervals against the whole workload average response time. WRT and average RT are both generated using the original response time values embedded in MSRC traces. In the latter two steps, we will exclusively use response times obtained from our own testbed.

2. To validate Step 1, we first replay a selected number of MSRC traces[2] in *whole* on our testbed at the original pace (one week) and use it as a reference instead of the original response time values. We then use the obtained response time values to validate our evaluation of selecting representatives (Step 2, Figure 5).

3. In Step 3 (Figure 5), we evaluate the accuracy of DiskAccel as a whole. The difference between this step and Step 2 is use of response time results from **partial** testbed runs instead of using whole trace response times used for calculating WRT. It must be noted that the reference average response time is still calculated using the whole, one-week runs similar to Step 2. Partial runs represents both stages of DiskAccel (a) picking representative intervals, evaluated in Step 1 and further validated in Step 2 and (b) replaying on our testbed, including warm-up.

4. TRACE SAMPLING

To reduce traces to a small number of representative intervals, our methodology first performs clustering to choose

[2]Due to the extensive time needed to run each MSRC trace at original speeds, we had to use a subset of these workloads to reduce experimentation time. We tried to pick a single trace from each class for this purpose: *hm_0*, *mds_1*, *prn_0*, *proj_4*, *prxy_0*, *rsrch_0*, *src1_2*, *ts_0*, *usr_2*, *wdev_0*, *web_2*.

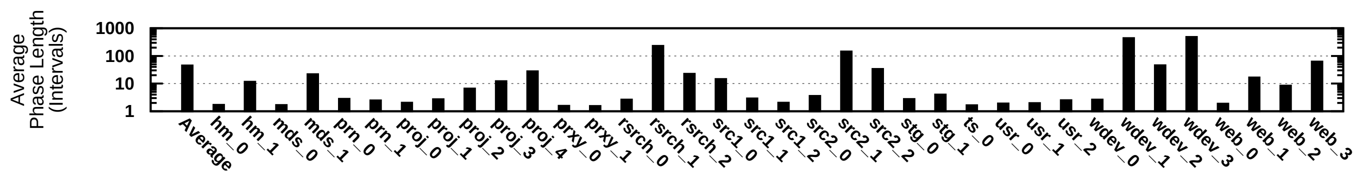

Figure 6: Average phase length across 10-second intervals (A phase is defined as a continuous stretch of intervals where variation between the response time of each two intervals is less than 10%.)

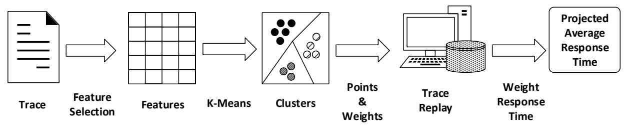

Figure 4: DiskAccel workflow.

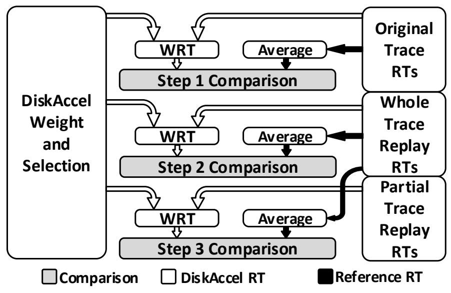

Figure 5: Evaluation steps (RT: Response Time, WRT: Weighted Response Time).

the representative intervals and then replays them while performing warm-up and enforcing request dependency. In this section, we focus on I/O traces and their difference with CPU traces. Then, a systematic sampling based on SMARTS [58] will be evaluated and its shortcomings will be demonstrated. Finally, we describe the process of clustering trace intervals and compare the accuracy and time reduction that can be achieved by choosing representative intervals.

4.1 I/O versus CPU Workloads

Before we engage in application of various sampling techniques, we demonstrate key differences of CPU traces in comparison to I/O traces. First of all, CPUs have much higher activity levels in comparison to I/O devices, meaning an I/O trace of the same activity is going to have much lower number of requests than that of the CPU traces. Second, I/O traces are very bursty [32, 21, 22] and the response time fluctuates wildly between request intervals, while in CPU traces *Cycles per Instruction* (CPI) remains constant for many different intervals [10]. Due to the bursty and sparse nature of storage workloads, unlike microprocessor based methodologies which divide traces into intervals based on the number of requests within them [58, 13], we divide each trace into intervals of a fixed duration.

Burstiness and auto-correlation of storage workloads [32, 21, 22] means that requests cannot be considered independently and any sampling must capture the temporal context of requests. The solution used by [14] was to use a duration-based division of I/O traces which captures temporal context and manages to greatly diminish the impact of long range dependence on performance. We group requests into 10-second intervals. There are a number of reasons behind this selection; first of all, the authors in [14] show that this interval duration diminishes the effect of long range dependence very well. Secondly, as will be demonstrated in Sec. 6.2, run-time of per-interval warm-ups dwarf that of the actual intervals being measured. Nevertheless, a week contains roughly sixty-thousand individual 10-second intervals, a number that allows low error and considerable speed-ups by our proposed methodology.

To demonstrate the difference between I/O and CPU trace behavior, we show the average duration of a "phase" in MSRC traces [28] in Figure 6. Similar to [13], we define a phase as a set of successive intervals where performance fluctuates by less than 10%. Based on the trace, the average phase length can vary from a single interval to multiple hours. It must be noted that long average phase length can also be caused by extreme sparsity, as is the case for the three most dense traces with less than 100 requests per hour on average. It is evident from these results that storage traces cannot be expected to have well-defined phases such as microprocessor traces. However, although I/O response time can affect processor CPI [17], the lack of phase behavior at the I/O request level does not mean that there is no phase behavior at the fine level of detail captured by microprocessor traces. I/O requests are removed from the details of microprocessor activity in such cases.

4.2 Systematic Sampling

In this section, we evaluate the effectiveness of using systematic sampling at representing the performance of the whole trace. SMARTS [58] determines the number of sampled intervals which must run in a detailed manner by using confidence intervals. It then uses systematic sampling with a fixed period to select these intervals and run them in a detailed manner. We evaluate the effectiveness of both these steps using MSRC traces [28] and conclude that this strategy is unsuitable for reducing storage traces.

Similar to SMARTS [58], we first determine the number of intervals using normal distribution confidence intervals. Figure 7 shows the total duration of requests that must be sampled in order to reach a maximum error margin of 15% with a confidence of 90%. This tolerance is much more relaxed than the work in [58] but we believe it is essential due to the bursty nature of storage workloads.

As can be seen from the results in Figure 7, some workloads can be significantly reduced. However, this does not extend to every workload and on average, *500 minutes* of pure replay time (sans the significant overhead of cache warm-up) is needed based on this method. These reductions, while significant, are still impractical for repetitive experiments. This method is also prone to over- and under-reporting the number of intervals. For example, *rsrch_0*, *wdev_1*, and *prxy_0* are reported to require less than *4 samples* but *hm_0* which contains less than 700 requests is reported to need over *27 minutes* of pure interval run-time.

As explained earlier, intervals in [58] are sampled by a fixed period. This requires the specification of a starting offset. This choice is much more challenging for storage workloads as they are bursty and lack well-defined phase behavior. For example, some starting offsets result in entirely empty set of samples for some traces. As noted earlier, this methodology may poorly estimate the number of required samples. To aid systematic sampling, two improvements were tested (a) the minimum number of samples is set to 30 to alleviate the issue of under-sampling (*Min30* versus *NoMin*) (b) since a starting offset is needed for systematic sampling, we choose the single run containing the most requests (*Single*) versus averaging the 10 most populated runs (*Avg10*). Please note that should we use the latter enhancement in practice, it results in 10x slower experiments.

It can be concluded from the experiments (Figure 8) that even with the above enhancements and when using the numbers embedded in the traces themselves, systematic sampling produces significant error. The *Single_30Min* which does not average multiple runs has significant error, even in workloads such as *proj_3*, *rsrch_1*, and *hm_1* where multiple hours worth of samples need to be taken. We attribute this high error to (a) lack of phase behavior in most traces (depicted in Figure 6) and (b) the burstiness of workloads which makes the choice of representative intervals challenging. This is in stark contrast with microprocessor instruction traces where phase behavior is far better defined [46]. We conclude that a guided sampling methodology based on the properties of request intervals is essential for higher accuracy of trace size reduction.

4.3 Feature Extraction and Data Mining

Shortcomings of systematic sampling show that a guided method of selecting representative intervals is essential for reaching higher accuracy. As it is not a good practice to use the performance numbers of a certain hardware configuration when trying to design a general methodology, we refrain from using response time results in clustering and only use these numbers to measure the accuracy of our methodology. Instead, we use properties such as trace request arrival time, address, and type to generate feature vectors for clustering.

Use of *Arrival Time Entropy* as a key property of trace behavior has been proposed in [54, 14, 52]; the latter work also uses address entropy to identify uneven accesses to disk locations. We use the multi-level aggregation scheme shown in Figure 9 which has been used in [54, 14, 15] to estimate the entropy of data points in continuous space. At each aggregation level, buckets have the same size and the next aggregation level is obtained by merging consecutive array buckets in pairs. At every level, a bucket is treated as an alphabet letter in discrete Shannon Entropy [45] calculations and the number of requests within it as the letter frequency.

We use this multi-level aggregation methodology to measure the time and address entropy. While time is continuous and can be divided easily, address is a discrete quantity. Thus, we use 512 bytes as the smallest unit, which is the common denominator of request addresses as it used to be disk sector size for a long time. For both entropy calculations, we use 16 unique levels of aggregation which is higher than what was used in [14].

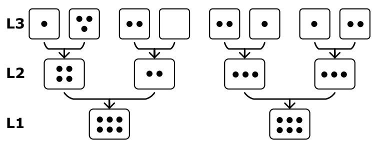

Figure 9: Using different aggregation levels for calculating entropy.

Arrival Time Entropy as described above, is an effective measure of workload burstiness and has been used in several studies [54, 14, 15]. Bursty behavior concentrates many requests in a compact period of time, skewing temporal distribution and reducing entropy. *Address-space entropy* translates to a higher spatial and temporal locality, which can greatly improve cache performance. *Working Set Locality* is defined as the ratio of total interval traffic to the number of unique sectors (working set) accessed in an interval. This measure was named "aggregation ratio" in [15].

We have conducted experiments with multiple features indicative of interval load. We use *Total Traffic*, *Average Request Size*, *Working Set Size*, and Interval *Request Count* as basic features. However, further discrimination between requests is necessary. Read and write requests have a different performance in I/O subsystems [17]. While writes can simply be satisfied by recording data in cache or non-volatile RAM, reads (or more specifically, read misses) should lead to immediate fetch of data, either from storage cache or media. As a result, read operations have a much more pronounced impact on system performance [17] and thus, it is likely beneficial for them to have a higher priority. As such, we also use *Read Ratio* as one of the main features.

Another key property of I/O workloads which affects both *Hard Disk Drives* (HDDs) and *Solid State Drives* (SSDs) is the randomness of I/O requests. Aside from poor cache performance, vastly different implementation details penalize random requests compared to sequential requests. Random accesses force HDDs to perform complex seeks, reducing performance. For SSDs, random accesses generate many more requests to the flash chips, reducing throughput. To recognize random requests, we use a queue-aware distance based scheme similar to what was used in [2]. In this classification, the distance of address of each new request is compared against the address of a number of recent requests. The minimum distance referred to as *Travel Distance* is compared against *Randomness Distance Threshold*.

If the travel distance exceeds this threshold, the request is classified as random. In this paper, this number of recent requests (queue depth) is taken to be 32 requests and Randomness Distance Threshold is 128KB, which seem to be the queue depth and read-ahead length of our surveyed disk drives. *Randomness Ratio* is the feature representing the fraction of random requests within an interval. We also use the sum of request Travel Distance and Average request Travel Distance, as *Total Travel Distance* and *Average Travel Distance* features, respectively. Table 1 shows the features investigated in this study for the construction of feature vectors along with the properties which we think are related to these features.

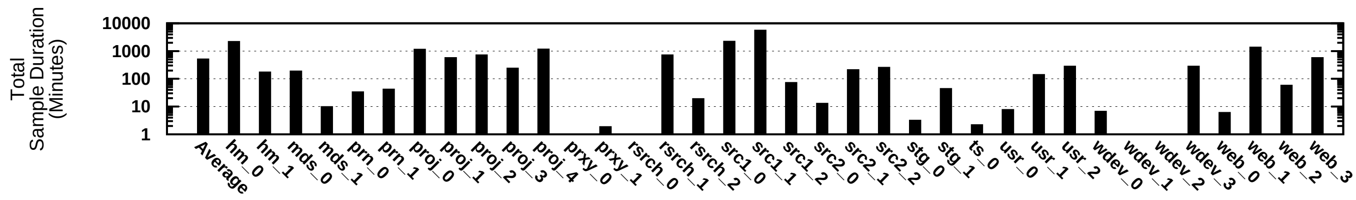

Figure 7: Total duration of intervals selected by confidence-based sampling with a maximum error of 15% and a confidence of 90%. The full duration of the traces is one week (∼10000 minutes).

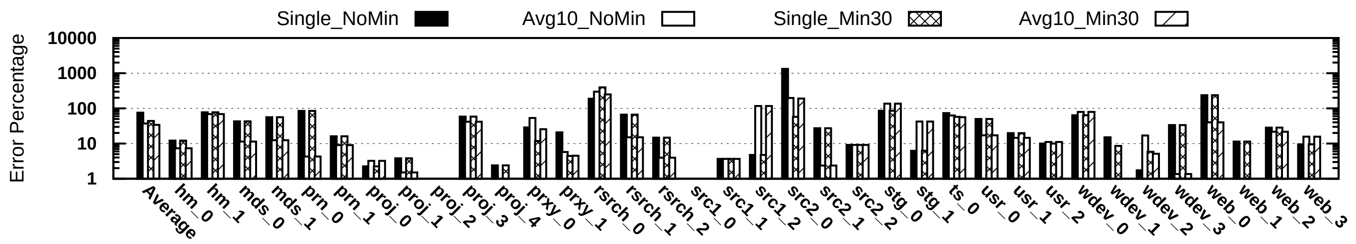

Figure 8: Error percentage of using a SMARTS-like [58] systematic sampling policy on 10-second intervals. AvgN averages the N runs with the highest total request count (Single = Avg1). MinK places a minimum bound of K on the sample count (NoMin = Min1).

4.4 Data Mining

Clustering algorithms group data points based on features similarities. Our aim at data mining, similar to SimPoint [13] is to, (a) group trace intervals into similar clusters and (b) choose a suitable representative for each cluster. We group features together to construct feature vectors as inputs of the clustering algorithm and use WRT error as a measure of accuracy.

As dense request intervals have much more impact on performance [29], the process of clustering and selecting representative intervals must take interval density into account. In other words, the process of clustering must support weighting data points. The authors in [1] conduct a study on weighted clustering where different data points may have different weights assigned. Based on their classification, K-Means is *weight-sensitive* and its clustering behavior is always affected by the repetition of points, noted by them as a naive method of applying weights to data points. SimPoint 3.0 [13] faces a similar challenge due to use of variable length intervals and augments K-Means with support for interval weighting.

We use a weighted variation of K-Means [7] which supports the application of weights to each individual data point. Before feeding the feature vectors to clustering, we perform preprocessing using [19] to center data on zero and set the variance to unity to improve the output of machine learning. We also had to specify K, the number of clusters, in K-Means. Our calculations of error show that the value of K has a great impact on response time estimation error. To select K, we have implemented BIC [42], based on the SimPoint 3.0 implementation in [13] which supports weights, as well as F_k from [31]. Both these methods use distortion and the number of clusters to determine the optimal number of clusters. Based on our experiments, BIC performs better and we omit the results of F_k here due to page limitation. In our methodology, we generate clusterings with K ranging from 2 to 50 with approximately sixty-thousand 10-second intervals as input and choose the clustering with the lowest BIC.

Table 1 shows the final set of features that were extracted for every interval with a short description of each feature. Depending on the selection of features, each interval is represented by a point with the features as its coordinates and the number of requests within it as the weight. Weighted K-Means is then performed on the data with different values of K, forming clusters and reporting the centroids. Intervals represented by points closest to cluster centroid are picked as the cluster representative. We use weighted response time of the representative intervals to measure the error in estimated response time compared to the weighted response time of all intervals. We weight the representative intervals by the total number of requests inside the cluster.

For simplicity, we have used a unified feature vector across all traces. We aim to improve upon this in our future work. The best feature vector for clustering data based on our experiments is *ArqWslRndAntEntTreAte*, which uses Average Request Size, Working Set Locality, Randomness, Address and Time Entropy, Total and Average (Queue-Based) Travel Distance as the feature vector. We choose this name based on the three-word short form in Table 1. Figure 10 shows WRT error by clustering using weighted K-means. Our results show an average and maximum error of 3.6% and 12.1% using the original response time in the traces themselves.

5. TRACE REPLAY

We presented details of our experiment testbed in Sec. 3. In this section, we explain the process of enforcing dependency and the methodology used for warm-up of state before each representative interval is run.

5.1 Enforcing Causality

A very important difference of replaying traces versus running programs is the fact that inter-arrival time of requests is partly influenced by the inter-arrival time of original hardware. This is caused by inter-request dependency in the

Table 1: List of features explored for clustering.

Feature Name	Short Form	Description	Related Property
Random Ratio	*Rnd*	Percentage of Queue-Based Random Requests (See [2])	Temporal and Spatial Locality
Read Ratio	*Rd*	Percentage of Read Requests	Read Frequency
Total Traffic	*Mss*	Sum of Request Sizes	Device Load
Total Request Count	*Cnt*	Total Number of Requests	Device Load
Address Entropy	*Ant*	Multi-level Address Shannon Entropy	Temporal and Spatial Locality
Arrival Time Entropy	*Ent*	Multi-level Arrival Time Shannon Entropy	Burstiness
Working Set	*Wst*	Sum of Unique Locations Accessed	Device Load, Temporal Locality
Working Set Locality	*Wsl*	Ratio of Total Traffic to Working Set	Temporal Locality
Average Request Size	*Arq*	Ratio of Total Traffic to Request Count	Device Load
Travel Distance	*Tre*	Queue-Based Request Offset Travel (See [2])	Spatial Locality, Seek Count
Average Travel Distance	*Ate*	Average Queue-Based Request Offset Travel (See [2])	Spatial Locality, Seek Count

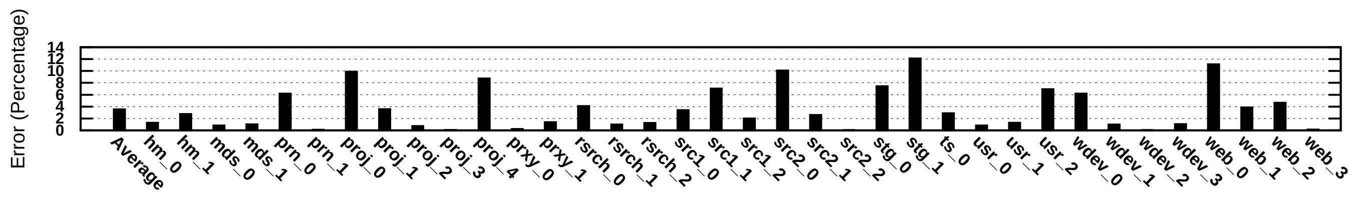

Figure 10: Per-trace WRT error of our chosen feature vector *ArqWslRndAntEntTreAte* with original response times (Step 1 in Figure 5).

original system activity. Ignoring this dependency is referred to as open-loop replaying which causes significant error [41], making closed-loop replays essential. However, most of traces available in public domain fail to record important semantic information and this information must be inferred based on the type and arrangement of requests.

Our replay tool replays traces in a closed-loop manner by constructing I/O request dependency trees. Consider the scenario in Figure 11; in the original replay, request A must finish before B is issued (Top). On the destination hardware, A may take considerably longer (Middle and Bottom). Open-loop replay (Middle) issues request B by Δ_{OL} units of time after A starts, even if it violates the dependency. In contrast, closed-loop replaying waits Δ_{CL} units of time *after* A is finished before sending request B. We believe this method better reflects the relationship between requests.

As noted earlier, enforcement of dependencies is much easier and accurate when detailed traces [55, 26] are present. These tools use rich information to enforce dependency as it was present in the original run. Most available public datasets such as the one used in our study [28] lack detailed information, especially those that record long periods of activity which is more representative. As a result, we have to make approximations of trace behavior for replays.

While it is possible to replay every request as blocking (i.e., controlled similar to A and B in Figure 11), we do not believe it represents real system behavior. Specifically, writes are normally cached by the operating system inside the main memory and are flushed to disk based on operating system behavior. While written data can be flushed to disk by calling *flush()* on Linux and *fcntl(F_FULLSYNC)* on Mac OS X [55], the study in [35] shows that a small percentage of reads are non-blocking and write behavior depends on the workload. As a result, out of the trace requests, we consider reads to be blocking and writes to be non-blocking. As a further support for this decision, it has been observed that microprocessor CPI has a correlation with average I/O read request response time [17], but no such relation exists with average request time for combined read/write requests.

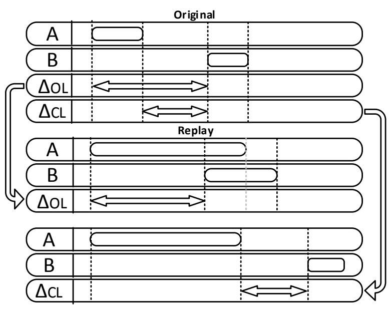

Figure 11: Comparison of open-loop (Middle) and closed-loop (Bottom) replays of a recorded workload (Top). Originally (Top), B depends on A and its start is Δ_{OL} and Δ_{CL} units of time after the start and finish time of A, respectively. On the destination, A takes longer than original and unlike the open-loop replay (Middle), close-loop (Bottom) replay avoids potential conflicts.

We propose a dependency policy such that in the absence of semantic information, read and write operations are assumed to be blocking and non-blocking, respectively. In other words, when calculating request arrival time in our replay tool we have three scenarios, (a) requests following a read requests conclusion will have their arrival time calculated based on its finish (Figure 11, bottom), (b) requests that arrive before a read concluding only depend on its arrival time (Figure 11, middle), and (c) requests that follow a write, whether it concludes or not depend on its arrival time (Figure 11, middle).

An example of this policy can be seen in Figure 12 which

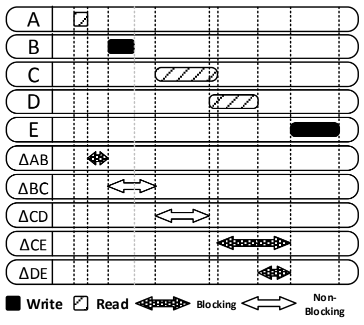

Figure 12: Inferring dependencies by treating reads as blocking and writes as non-blocking.

shows the arrival and finish time in the original trace. Dependencies are inferred based on arrival time and request type. A is a read and is finished before B starts. Thus, it falls under the first scenario and is blocking. B is a write and falls under the third scenario. In other words, we assume that after the program issues A and B, the wait time is only caused by CPU thinking time and not waiting on their results. Request D arrives before C is completed and falls under the second scenario. We assume CPU was actually thinking before issuing D as it did not need C to finish to get issued. Request E arrives after *both* C and D are finished and falls under the first scenario, where both Δ_{CE} and Δ_{DE} have to be preserved. In other words, $Arrival_E = max(Finish_C + \Delta_{CE}, Finish_D + \Delta_{DE})$ where $Arrival_E$ is the calculated arrival time and variables $Finish_C$ and $Finish_D$ describe the finishing time in our testbed. Enforcing both these delta times ensures that even if requests C and D are acknowledged in a different order than the original trace, the assumed constant processing time will be preserved. Since the introduction of *Native Command Queuing* (NCQ) to the Serial ATA standard, out of order acknowledgment should be assumed to be present in all modern storage hardware.

5.2 Accelerated Warm-up

As the goal of DiskAccel is to run storage traces with the greatest possible accuracy, the architectural state of the hardware must be warmed up appropriately. While restoration of architectural state can be performed via checkpoints [13], this capability is rarely available on real hardware. As a result, warm-up of the state should be performed by sending requests.

In [58], this is performed via a step called *Detailed Warm-up*, where a portion of the instruction trace preceding each selected interval is replayed. Between the end of a run and next detailed warm-up, instructions are emulated so the user visible state is always valid, but detailed warm-up should cover the whole architectural state. For I/O traces, an arbitrary portion of the traces has been used to warm up the state [25, 56].

To minimize run-time, we should reduce the time spent on warm-up. However, architectural state of storage subsystems is much larger than that of microprocessors. While current microprocessors have tens of megabytes of cache at most, cache size can be as large as 128MB for HDDs [43], 512MB for SSDs [38], and much more extensive caching is performed in enterprise storage systems.

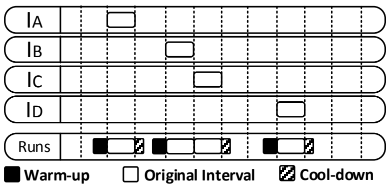

Figure 13: Runs generated (including warm-up and cool-down) for the representative intervals $I_A...I_D$. Since I_B and I_C are continuous, a singular run can perform measurements for both of them.

The warm-up process must select enough requests before each representative interval so the cache is warmed up. As hit ratio of workloads might be variable, we should care about the request footprint and not the sum of request sizes. To track request footprint, we use interval trees to track each unique accessed sector. We then move in backwards from the start of each representative interval and stop when the total capacity of unique disk locations reaches the capacity of the cache. Please note that due to operations such as read-ahead, much fewer requests must be selected to warm up the cache but we choose a pessimistic policy to ensure the accuracy of warm-up. After the last requests in a representative interval is issued, there is a possibility that requests exist that preceded the conclusion of these requests. To increase accuracy, after all requests in a representative interval are issued, we send requests that originally arrived before these requests finished. We refer to this as *cool-down* process.

A high level view of the replaying process for representative intervals can be seen in Figure 13. $I_A...I_D$ are the selected representative intervals and three runs have been generated for replaying them. The striped lines show the interval boundaries; each run is preceded by a warm-up and in its conclusion, a cool-down process is run. I_B and I_C are immediate neighbors and thus, I_B can act as warm-up for I_C and I_C can perform as a cool-down for I_B. This allows them to share the same run.

The warm-up behavior of our solution has important advantages: (a) if the device cache is so large that warm-up of multiple intervals overlaps, the runs are joined and warm-up time is reduced considerably, (b) if the workload is very dense, not all the requests between different representative intervals are run and only the required amount of warm-up is performed.

6. EXPERIMENTAL VALIDATION

As mentioned previously, we use a subset of MSRC [28] in the experiments. All these traces approximately span over a week and have been taken from a variety of servers. While these traces do include the response time of the original server, we use our implementation of a trace runner to (a) validate the results of the clustering process and (b) test the viability and accuracy of running representative trace intervals to estimate the original request response time. We will explain the results of these two experiments in the following subsections.

6.1 Cross-validation of Clustering

In the previous section, we used the response time of the original trace to measure the accuracy of our methodology

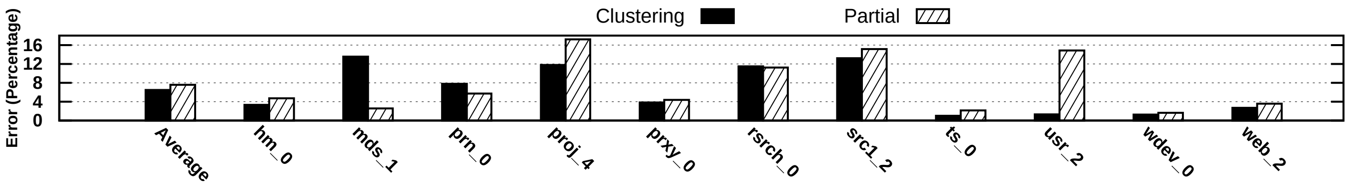

Figure 14: Per-trace WRT error of our chosen feature vector *ArqWslRndAntEntTreAte*, compared against the reference. The "clustering" statistic is the WRT error by clustering whole trace runs (Step 2 in Figure 5) and "partial" refers to the WRT error of partial trace runs (Step 3 in Figure 5). The reference for both cases is the average response time of a whole trace replay.

(Step 1 in Figure 5). To rule out the possibility of our learning methodology being overly tuned to the *Redundant Array of Independent Disks* (RAID) used on the original machines, we use our replay tool described in Sec. 3 to perform the replays on our testbed. Multiple identical and independent disks were used to perform concurrent experiments.

In Step 1 of Figure 5, we used the original response time values in the trace both as the reference and for calculating WRT. In Step 2, we repeat this comparison with different hardware. We use our replay tool to run the thread in full with no acceleration (requiring one week with each individual disk). The resulting response time values were used for calculating both the reference and WRT values yielding an average and maximum error of 6.5% and 13.6%, respectively. Detailed results can be seen in Figure 14 as "clustering" error bars. This error is comparable to those obtained from the original traces and demonstrates that our clustering methodology can successfully select representative intervals that are accurate across different hardware (Single SATA HDDs vs. RAID1 and RAID5 arrays [28]). In the next section, we conduct experiments to perform partial runs (Step 3 of Figure 5).

6.2 Running Representative Intervals

The actual process of DiskAccel only runs the representative intervals on the target hardware and uses WRT to estimate the response time of a complete run. To run each interval, hardware state must be similar to that of the original run at the start of a representative interval. To do so, key properties of the target hardware and list of representative intervals are given to our trace replaying tool.

In our tool, the warm-up of hardware state is performed by running a number of requests preceding each representative interval, so that the hardware cache has the same contents as the complete run. Warm-up requests are selected via an interval tree which ensures that requests are selected by their total footprint and not the sum of request sizes. To speed up the warm-up process, warm-up requests are run as fast as possible.

As cache size varies by different hardware, different number of requests must be sent for warm-up and user must supply cache size according to the hardware on which requests are replayed. Another user-supplied parameter is disk queue depth which is the number of requests that can be concurrently processed by storage hardware. If more requests than queue capacity are to be sent by the hardware, the trace replaying tool must block until at least one request is finished. Cache size and queue capacity for our disks are 8MB and 32 requests, respectively.

We replayed the representative intervals of the selected traces using our replaying tool, weighted each interval by the number of requests within its cluster and calculated WRT. The WRT error, reported by only replaying the representative intervals is reported in Figure 14. The average and maximum response time errors are 7.6% and 17.2%, respectively.

The main aim of the methodology introduced in this paper is to reduce the run-time for replay-based experiments. In Figure 15 we display (a) run-time of the whole trace, (b) partial runs with no accelerated warm-up, calculated based on the original trace arrival time (a lower bound as our testbed is slower), (c) run-time with accelerated warm-up measured on our testbed, and (d) pure run-time of representative intervals with no warm-up. The results show massive speed-ups (the graph is log-scale) compared to both the unaccelerated run and the full run-time with a 577 times reduction on average.

In short, DiskAccel can select representative intervals independent of the hardware configuration using a clustering process that must be run only once for each trace. DiskAccel needs about 20 minutes of run-time for partial runs (including warm-up) of a one-week trace. Our tool also manages to massively reduce run times while keeping average and maximum error across different hardware to a minimum.

6.3 Discussion

As noted in Figure 14, estimation error of our current methodology can be attributed to (a) the process of clustering and choice of representatives, and (b) partial replay and warm-up of state. Our clustering uses a constant feature vector for choosing representative intervals. While we tried alternative approaches such as *DBSCAN* [11] as well as augmenting K-Means with *Principal Component Analysis* (PCA) [18] (as implemented in [19]), we did not achieve better results. We plan to enhance our machine learning with suitable feature selection to improve the results. Based on our investigations, we believe that the error caused by partial runs is mainly caused by discrepencies in low-level behavior of HDD (i.e., cache flush after idleness). We hope to improve this behavior by controlling the timing of requests as the warm-up finishes.

7. CONCLUSION

In this paper, we introduced a methodology, called DiskAccel, to extract key properties of storage traces and use them to select representative intervals by employing the weighted K-Means algorithm. We also developed a tool to effectively run the representative intervals, fill the hardware cache for warm-up, and enforce request dependencies.

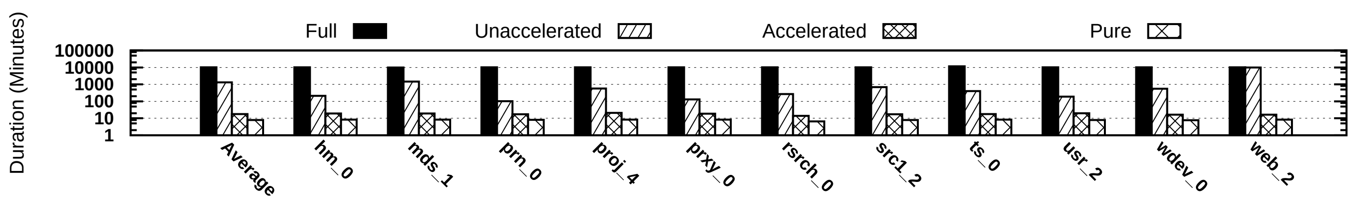

Figure 15: Run time durations of different modes, Whole trace (Full), Partial runs without accelerated warm-up (Unaccelerated), Partial runs with accelerated warm-up (Accelerated), and without warm-up (Pure).

We performed the evaluation of DiskAccel at three different steps. We began by evaluating the ability of the clustering process at accurate estimation of whole trace average response times; first, using the response times recorded in the original trace and then, using response times from replay of whole traces using our tool. We also used our replay tool to only run the representative intervals with the required warm-up in an accelerated manner. Our methodology managed to estimate the average response time by an average and maximum error of %7.6 and %17.2, respectively while speeding up the I/O experiments by 577 times on average.

As a future work, one can further improve the current solution so error and experimentation times could be further reduced. We also limited our study to a single hardware setup due to the time-consuming nature of gathering reference response times at the original rate. We would like to study the behavior of the proposed methodology for more diverse hardware setups, especially SSDs and hybrid hardware architectures in the future.

8. REFERENCES

[1] M. Ackerman, S. Ben-David, S. Branzei, and D. Loker. Weighted clustering. In *AAAI*, pages 858–863, 2012.

[2] I. Ahmad. Easy and Efficient Disk I/O Workload Characterization in VMware ESX Server. In *Proc. IISWC'07*, pages 149–158, Sept.

[3] E. Anderson, M. Kallahalla, M. Uysal, and R. Swaminathan. Buttress: A toolkit for flexible and high fidelity i/o benchmarking. In *Proc. FAST'04*, pages 4–4.

[4] J. Axboe. Fio-flexible io tester. `http://freecode.com/projects/fio`, 2008. [Online; accessed 1-August-2014].

[5] A. D. Brunelle. Block i/o layer tracing: blktrace, 2006.

[6] J. S. Bucy, J. Schindler, S. W. Schlosser, and G. R. Ganger. The disksim simulation environment version 4.0 reference manual (cmu-pdl-08-101). *Parallel Data Laboratory*, page 26, 2008.

[7] J. Burkardt. KMEANS - the K-Means Data Clustering Problem. `http://people.sc.fsu.edu/~jburkardt/m_src/kmeans/kmeans.html`, 2013. [Online; accessed 16-June-2014].

[8] P. Cao, E. W. Felten, A. R. Karlin, and K. Li. Implementation and performance of integrated application-controlled file caching, prefetching, and disk scheduling. *TOCS*, 14(4):311–343, 1996.

[9] R. Coker. Bonnie++. `http://www.coker.com.au/bonnie++/`, 2001. [Online; accessed 20-June-2014].

[10] A. S. Dhodapkar and J. E. Smith. Comparing program phase detection techniques. In *Proc. MICRO'03*, page 217.

[11] M. Ester, H. P. Kriegel, J. Sander, and X. Xu. A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise. In E. Simoudis, J. Han, and U. Fayyad, editors, *Second International Conference on Knowledge Discovery and Data Mining*, pages 226–231. AAAI Press, 1996.

[12] Google. AIOUserGuide. `https://code.google.com/p/kernel/wiki/AIOUserGuide`, 2014. [Online; accessed 1-August-2014].

[13] G. Hamerly, E. Perelman, J. Lau, and B. Calder. Simpoint 3.0: Faster and more flexible program phase analysis. *JILP*, 7(4):1–28, 2005.

[14] B. Hong and T. M. Madhyastha. The relevance of long-range dependence in disk traffic and implications for trace synthesis. In *Proc. MSST'05*, pages 316–326.

[15] B. Hong, T. M. Madhyastha, and B. Zhang. Cluster-based input/output trace synthesis. In *Proc. IPCCC'05*, pages 91–98.

[16] W. W. Hsu, A. J. Smith, and H. C. Young. Characteristics of production database workloads and the tpc benchmarks. *IBM Systems Journal*, 40(3):781–802, 2001.

[17] B. Jacob, S. Ng, and D. Wang. *Memory Systems: Cache, DRAM, Disk.* 2010.

[18] I. Jolliffe. *Principal Component Analysis.* John Wiley & Sons, Ltd, 2005.

[19] E. Jones, T. Oliphant, P. Peterson, et al. SciPy: Open source scientific tools for Python. `http://www.scipy.org/`, 2001–. [Online; accessed 1-August-2014].

[20] J. Katcher. Postmark: A new file system benchmark. Technical report, Technical Report TR3022, Network Appliance, 1997. www.netapp.com/tech_library/3022.html, 1997.

[21] S. Kavalanekar, D. Narayanan, S. Sankar, E. Thereska, K. Vaid, and B. Worthington. Measuring database performance in online services: a trace-based approach. In *Performance Evaluation and Benchmarking*, pages 132–145. 2009.

[22] S. Kavalanekar, B. Worthington, Q. Zhang, and V. Sharda. Characterization of storage workload traces from production windows servers. In *Proc. IISWC'08*, pages 119–128.

[23] T. Kgil and T. Mudge. Flashcache: a nand flash memory file cache for low power web servers. In *Proc. CASES'06*, pages 103–112.

[24] J. Lau, S. Schoemackers, and B. Calder. Structures for phase classification. In *Proc. ISPASS'04*, pages 57–67.

[25] C. Maltzahn, K. J. Richardson, and D. Grunwald. Reducing the disk i/o of web proxy server caches. In *Proc. ATC'99*, pages 225–238.

[26] 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 *Proc. FAST'07*.

[27] N. Mi, G. Casale, L. Cherkasova, and E. Smirni. Injecting realistic burstiness to a traditional client-server benchmark. In *Proc. ICAC'09*, pages 149–158.

[28] D. Narayanan, A. Donnelly, and A. Rowstron. Write off-loading: Practical power management for enterprise storage. *ACM ToS*, 4(3):10:1–10:23, Nov. 2008.

[29] D. Narayanan, A. Donnelly, E. Thereska, S. Elnikety, and A. Rowstron. Everest: Scaling down peak loads through i/o off-loading. In *Proc. OSDI'08*, December.

[30] W. D. Norcott and D. Capps. Iozone filesystem benchmark. `http://www.iozone.org/`, 2006. [Online; accessed 20-June-2014].

[31] D. T. Pham, S. S. Dimov, and C. Nguyen. Selection of k in k-means clustering. *Proceedings of the Institution of Mechanical Engineers, Part C: Journal of Mechanical Engineering Science*, 219(1):103–119, 2005.

[32] A. Riska and E. Riedel. Disk drive level workload characterization. In *Proc. ATC'06*, pages 97–102.

[33] M. Rosenblum and J. K. Ousterhout. The design and implementation of a log-structured file system. *TOCS*, 10(1):26–52, 1992.

[34] S. Rostedt and D. V. Hart. Internals of the rt patch. In *Proceedings of the Linux symposium*, 2007.

[35] C. Ruemmler and J. Wilkes. Unix disk access patterns. In *USENIX Winter*, volume 93, pages 405–420, 1993.

[36] R. Saikkonen. Linux i/o port programming mini - howto. `www.tldp.org/HOWTO/pdf/IO-Port-Programming.pdf`, 2000. [Online; accessed 19-November-2014].

[37] R. Salkhordeh, H. Asadi, and S. Ebrahimi. Operating system level data tiering using online workload characterization. *The Journal of Supercomputing*, 71(4):1534–1562, 2015.

[38] Samsung Corporation. *Samsung SSD 840 PRO Series*, August 2013. Rev. 1.2.

[39] P. Scheuermann, G. Weikum, and P. Zabback. Data partitioning and load balancing in parallel disk systems. *The VLDB Journal*, 7(1):48–66, 1998.

[40] J. Schindler and G. R. Ganger. Automated disk drive characterization (poster session). *SIGMETRICS Perform. Eval. Rev.*, 28(1):112–113, June 2000.

[41] B. Schroeder, A. Wierman, and M. Harchol-Balter. Open versus closed: A cautionary tale. In *Proc. NSDI'06*, pages 18–18, 2006.

[42] G. Schwarz. Estimating the dimension of a model. *The Annals of Statistics*, 6:461–464, 1978.

[43] Seagate Corporation. *Enterprise Capacity 3.5 HDD Data Sheet*, April 2014.

[44] B. Seo, S. Kang, J. Choi, J. Cha, Y. Won, and S. Yoon. Io workload characterization revisited: A data-mining approach. *IEEE TC*, 63(12):3026–3038, Dec 2014.

[45] C. E. Shannon. A note on the concept of entropy. *Bell System Tech. J*, 27:379–423, 1948.

[46] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *Proc. ASPLOS'02*, pages 45–57.

[47] A. J. Smith. Disk cache-miss ratio analysis and design considerations. *TOCS*, 3(3):161–203, 1985.

[48] V. Tarasov, S. Kumar, J. Ma, D. Hildebrand, A. Povzner, G. Kuenning, and E. Zadok. Extracting flexible, replayable models from large block traces. In *Proc. FAST'12*, pages 22–22, 2012.

[49] M. Tarihi. Serious problems using dixtrac extracted parameters. `https://sos.ece.cmu.edu/pipermail/disksim-users/2013-July/000821.html`, 2013. [Online; accessed 16-June-2014].

[50] A. Traeger, E. Zadok, N. Joukov, and C. P. Wright. A nine year study of file system and storage benchmarking. *ACM TOS*, 4(2):5, 2008.

[51] 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 *Proc. FAST'11*, pages 12–12.

[52] M. Wang, A. Ailamaki, and C. Faloutsos. Capturing the spatio-temporal behavior of real traffic data. *Performance Evaluation*, 49(1):147–163, 2002.

[53] M. Wang, K. Au, A. Ailamaki, A. Brockwell, C. Faloutsos, and G. Ganger. Storage device performance prediction with cart models. In *Proc. MASCOTS'04*, pages 588–595, Oct.

[54] M. Wang, T. Madhyastha, N. H. Chan, S. Papadimitriou, and C. Faloutsos. Data mining meets performance evaluation: Fast algorithms for modeling bursty traffic. In *Proc. ICDE'02*, pages 507–516.

[55] Z. Weiss, T. Harter, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. ROOT: Replaying Multithreaded Traces with Resource-Oriented Ordering. In *Proc. SOSP'13*, November 2013.

[56] T. M. Wong and J. Wilkes. My cache or yours? making storage more exclusive. In *Proc. ATEC'02*, pages 161–175, 2002.

[57] B. L. Worthington, G. R. Ganger, and Y. N. Patt. Scheduling algorithms for modern disk drives. *SIGMETRICS Perform. Eval. Rev.*, 22(1):241–251, may 1994.

[58] R. E. Wunderlich, T. F. Wenisch, B. Falsafi, and J. C. Hoe. SMARTS: Accelerating microarchitecture simulation via rigorous statistical sampling. In *Proc. ISCA'03*, pages 84–95.

[59] J. Zhang, A. Sivasubramaniam, H. Franke, N. Gautam, Y. Zhang, and S. Nagar. Synthesizing Representative I/O Workloads for TPC-H. In *Proc. HPCA'04*, pages 142–142, Feb.

[60] N. Zhu, J. Chen, T.-c. Chiueh, and D. Ellard. TBBT: Scalable and Accurate Trace Replay for File Server Evaluation. In *Proc. SIGMETRICS'05*, pages 392–393, 2005.

[61] Q. Zhu, Z. Chen, L. Tan, Y. Zhou, K. Keeton, and J. Wilkes. Hibernator: Helping disk arrays sleep through the winter. In *Proc. SOSP'05*, pages 177–190.