On the Accuracy and Scalability of Intensive I/O Workload Replay

Alireza Haghdoost and Weiping He, *University of Minnesota;*
Jerry Fredin, *NetApp;* David H.C. Du, *University of Minnesota*

https://www.usenix.org/conference/fast17/technical-sessions/presentation/haghdoost

This paper is included in the Proceedings of
the 15th USENIX Conference on
File and Storage Technologies (FAST '17).

February 27–March 2, 2017 • Santa Clara, CA, USA

ISBN 978-1-931971-36-2

On the Accuracy and Scalability of Intensive I/O Workload Replay

Alireza Haghdoost*, Weiping He*, Jerry Fredin†, and David H.C. Du*

*University of Minnesota Twin-Cities, †NetApp Inc.

Abstract

We introduce a replay tool that can be used to replay captured I/O workloads for performance evaluation of high-performance storage systems. We study several sources in the stock operating system that introduce the uncertainty of replaying a workload. Based on the remedies of these findings, we design and develop a new replay tool called *hfplayer* that can more accurately replay intensive block I/O workloads in a similar unscaled environment. However, to replay a given workload trace in a scaled environment, the dependency between I/O requests becomes crucial. Therefore, we propose a heuristic way of speculating I/O dependencies in a block I/O trace. Using the generated dependency graph, *hfplayer* is capable of replaying the I/O workload in a scaled environment. We evaluate *hfplayer* with a wide range of workloads using several accuracy metrics and find that it produces better accuracy when compared with two exiting available replay tools.

1. Introduction

Performance evaluation of a storage system with realistic workloads has always been a desire of storage systems developers. Trace-driven evaluation is a well-known practice to accomplish this goal. It does not require installation of a real system to run applications and does not expose a production systems to potential downtime risk caused by performance evaluation experiments. However, the lack of an accurate trace replay tool makes it less appealing and draws some skepticism of using trace-driven methods for performance evaluation of block storage devices and systems [19, 23].

An I/O trace typically includes the timestamps of when each I/O request is issued and completed. These timestamps can be used to replay the workload on a similar unscaled environment. However, the block I/O trace does not usually include any information about the dependencies between I/O requests. Therefore, these timestamps cannot be directly used in a workload replay on a scaled environment where the issue time of a latter I/O request may be determined by the completion time of a former request in a dependency chain. However, a latter I/O request may be issued earlier before the completion time of a former I/O request if there is no dependency between the two. Some examples of a scaled environment can mean a server speed is increased, the number of disk drives is doubled, or a faster type of drives is considered.

The I/O dependency information is available only in the file system or application layers. However, the intrinsic software overhead and high degree of parallelism that embedded in these layers reduce the capability of the workload replay to stress-test a modern block storage system when replaying I/O traces at file system or application layer. For example, Weiss et al. [24] demonstrates Can a scalable file system replay tool that can barely hit 140 IOPS, while modern storage systems are capable of driving intensive I/O workloads with hundreds of thousands of IOPS and a few milliseconds response-time [4].

Typical block I/O replay tools ignore I/O dependencies between I/O requests and replay them as fast as possible (AFAP) [15]. The AFAP approach cannot accurately replay a workload since it overlooks the intrinsic computation and wait time in the original application. Therefore, the characteristics of replayed workload may be different from that of an original application in terms of throughput, response-time and request ordering.

We believe a more accurate replay tool for scaled environments should try to mimic the behavior of the real application on the scaled environments and respect the existing dependencies in the I/O trace. This is possible by speculating the dependencies of I/O requests and trying to propagate I/O-related performance gains along the dependency chains. However, it is challenging to discover I/O dependencies simply based on a given block I/O workload trace without accessing the application source code.

In this work, we propose *hfplayer*, a replay tool that tries to infer potential I/O dependencies from the block I/O traces and replay I/O intensive workloads with more accuracy in both scaled and unscaled replay modes. In the scaled mode, the arrival of new requests depends on the completion of the previous requests, while in the unscaled mode each request is issued independently at a scheduled time [14].

The goal of this replay tool is to ensure that replayed I/O requests arrive the storage device (e.g., SAN controller) at the right time and order. We develop methods to ensure that the right number of I/O requests being issued from the user level and these requests traverse all the OS layers (the entire path from user space to device controller) with minimal interference on the workload throughput, I/O response time and ordering.

The intended use of *hfplayer* is for performance evaluation, debugging and validating different block storage devices and systems with realistic workloads. For the set of our experiments, it replays realistic I/O workload on both scaled and unscaled storage systems with less than 10% average error.

Our main contributions in the paper are summarized as follows: 1) Uncover and propose remedies to Linux I/O stack limitations and its non-deterministic I/O behaviors which are sources of measurement uncertainty. 2) Design and implement a workload replay engine which can more accurately replay a block I/O workload that has been captured in a similar unscaled environment. 3) Design and implement a scalable workload replay engine which can speculate I/O request dependencies and replay a realistic workload in a scaled environment. Our workload replay tool is open-source and available for download at: https://github.com/umn-cris/hfplayer.

The rest of this paper is structured as follows. In the next section, we describe several sources of non-deterministic behaviors for block I/O workload replay in the stock operating system. We also describe our approaches to remedy or work around these limitations. In Section 3 we discuss the I/O request dependencies and how we construct a dependency graph for a given workload by speculating I/O request dependencies. We also describe our approach of replaying the I/O workload considering the influences of the target storage in a scaled environment using the scalable replay engine of *hfplayer*. Then we evaluate the replay accuracy of *hfplayer* on both unscaled and scaled storages and compare the results with existing tools in Section 5. We refer and describe the relevant literature of this paper in Section 6. Finally, we summarize and offer some conclusions in Section 7.

2. Sources of Workload Replay Uncertainty

In this section, we introduce several key limitations to faithfully replaying a block I/O workload in the Linux I/O stack. One solution to potentially limiting OS impacts the workload replay accuracy is to embed the I/O engine of the workload replay tool in the kernel space [5]. This would reduce forced preemption of replay engine threads and eliminates the cost of user to kernel space context switch in replaying an I/O request. However, this approach limits the portability of the developed engine to a single OS platform and even to a specific kernel version. Therefore, we focus on developing a replay tool with a user space engine which is capable of working with the standard system calls on most OS platforms. In this work, we focus on Linux and propose a method to significantly reduce the context switch overhead. We will introduce the integration with IBM's AIX OS in the future work. Moreover, we believe user space I/O engine development is aligned with the emerging efforts in the industry to develop user space APIs like Intel SPDK for ultra high-speed storage devices [2].

User-space replay threads can submit I/O requests using synchronous or asynchronous system calls. Replay tools that are implemented with the synchronous method like *Buttress* [6] and *blkreplay* [15] need a large number of worker threads running in parallel to achieve a high replay throughput. Therefore, they may expose more inter-thread coordination overheads compared with the asynchronous method. These overheads are known as a major source of uncertainty in the workload replay [13]. As we show in Section 5, these limitations severely impact the accuracy of replaying I/O intensive workloads. Therefore, *hfplayer* replay engine is based on the asynchronous method and exclusively uses *libaio* which is Linux kernel support for asynchronous I/O operations as well as *NO-OP* scheduler without its *merge* functionality. We have identified that Linux kernel is not able to fully disable the I/O merge functionality in certain situations. We have proposed a patch to fix this bug and the patch has been merged into the mainstream branch since the kernel v3.13 [3].

Accurate I/O replay timing is another challenging issue which heavily depends on the timing predictability of the I/O stack. In an ideal situation, it should cost each I/O request a fixed amount of time to travel from the replay engine to the storage device. Therefore, if there is an inter-arrival gap of *n* milliseconds between two consecutive I/O requests in the given workload trace, the replay thread just needs to wait *n* milliseconds after issuing the first I/O request to issue the second request and expects both requests to arrive at the device with an accurate time interval of *n* milliseconds. However, our experiments show that the I/O stack travel time is quite unpredictable and thus such a timing accuracy is impossible if we do not carefully tune and watch the I/O queuing mechanism in the kernel. In order to work around these limitations, we have practiced the following four techniques to improve the workload replay accuracy.

2.1. I/O Stack Queue Sizing

In general there are two queuing layers in the asynchronous I/O submission path of Linux kernel. The *io_submit* system call first pushes a block I/O request (or *bio*) into the tail of the block layer queue, then it pulls another *bio* from the head of the queue, transforms it into a SCSI command and pushes it into the SCSI layer queue. Finally, it returns the success code to the user space, while the SCSI Initiator processes the commands in the SCSI layer queue using a kernel worker thread. These two queues usually do not have enough room for all new requests when the I/O stack is overloaded. In this case, the execution time of the system call becomes unpredictable.

While the block layer request queue size is tunable, the SCSI layer queue size is limited to the maximum supported TCQ tags by the SCSI layer. Usually a system call puts the *bio* request into a waiting list if it cannot find an empty queue tag and schedules a kernel worker thread to push it later. This schedule is expensive and significantly increases I/O submission unpredictability. Moreover, it changes the delivery order of I/O requests in the kernel space even if the replay tool thread submits these requests in order. Therefore, internal I/O queues in the kernel should be sized according to the HBA capability to reduce queue insertion failures. In our test environment, while our HBA can handle 2048 in-flight requests, the default block layer queue size is set to 128 and the default SCSI layer queue size is 32.

Obviously, adjusting the I/O stack queue sizes below the HBA capability reduces the potential I/O throughput and impacts the replay accuracy to reproduce high-throughput workload. On the other hand, enlarging the queue sizes more than HBA capability without any control on the number of in-flight requests overloads the HBA and results in multiple retries to sink I/O requests from the block layer queue into the SCSI command queue. Therefore, we set the internal queue size to the maximum size supported by HBA hardware and dynamically monitor in-flight requests in the replay engine to make sure it does not overload the internal queues.

2.2. System-Call Isolation

We have identified that forced preemption is another factor making the execution time of I/O submission system call unpredictable. The system call execution thread can be preempted by other kernel processes with higher priorities like interrupt service routines. Consequently, it will impact the timing and ordering accuracy of a typical multi-threaded I/O workload replay tool. While it is possible to isolate regular interrupt handlers from system call threads using IRQ and CPU affinities, it is hard to avoid collisions of the system call threads with non-maskable timer interrupts. Moreover, scheduling replay tool threads with real-time scheduling priority is not a viable option since it has been discovered as another source of uncertainty for the workload replay [13]. Therefore, our best effort is to pin the replay tool threads to a CPU set and exclude all maskable interrupts from execution on that CPU set.

2.3. In-Flight I/O Control

After making an I/O submission system call more predictable with the aforementioned techniques, we need to work around potential scenarios that the I/O path unexpectedly takes more time to submit asynchronous I/O requests. When the submissions of a few I/O requests get delayed in a worker thread, it prevents the following I/O requests from being submitted on time. For example, assuming three consecutive I/O requests are scheduled to issue at time t, $t+10\mu s$ and $t+20\mu s$ by a worker thread. If the submission of the first request unexpectedly

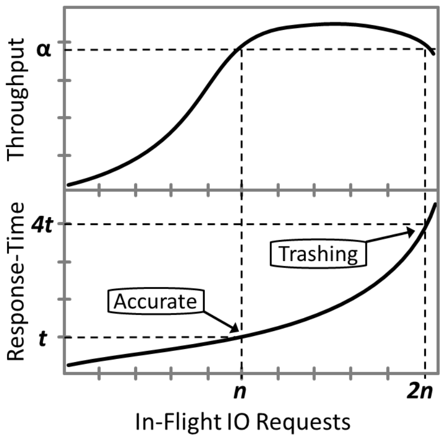


Figure 1: Response-Time Trashing

takes $30\mu s$, there is no need to wait any longer to issue the following two I/O requests since the scheduled issue times of these two requests are passed. In such a scenario, the replay tool issues several I/O requests in a very short amount of time and this burst of I/O requests forces the I/O stack to operate in a state that we call *Response-Time Trashing* state. In this state, an artificially large number of I/O requests are pushed into the I/O stack queues which severely impacts I/O response-time accuracy.

Figure 1 illustrates the *Response-Time Trashing* which occurs when the number of in-flight I/O requests is artificially high. For example, assuming we are planning to replay an I/O workload with a throughput of α and response-time of t. An accurate replay should reproduce the workload with the same throughput and response-time by keeping n I/O requests in-flight. However, unpredictable delays in the path cause a burst submission of an unexpectedly large number of I/O requests since their scheduled issue times have already passed. Therefore, in this example $2n$ requests are being queued in the stack instead of n requests. The throughput of I/O becomes saturated after a threshold number of queued requests is reached. Pushing more I/O requests to the internal queues only exponentially increases the response-time without any throughput improvement. Without an in-flight I/O control mechanism, the workload is being replayed with response-time of $4t$ instead of t since an artificially high number of requests are queued in the stack. Moreover, there is no reason to slow down or flush the queues since the workload is being replayed with the expected throughput of α. Note that some I/O workloads do not experience the dropping tail of I/O throughput by increasing the number of in-flight requests. These workloads usually saturate physical hardware resources of the SAN controller.

Limiting internal queue sizes is a conservative default approach to prevent I/O stack operating in the trashing state. This conservative approach bounds the maximum throughput intentionally to prevent typical applications without in-flight I/O control from creating a burst submissions of I/O request and forcing the I/O stack to operate in the trashing state. Therefore, we need to keep the internal kernel queues open and meanwhile dynamically monitor and control the number of in-flight I/Os in the replay engine.

Depending on the information included in the workload trace, it is possible to estimate the number of in-flight I/O requests required to replay the workload. The number of in-flight I/O requests at time t can be calculated by counting the number of I/O requests with issue times smaller than t and completion times larger than t. Assume that n requests are counted as the number of in-flight requests at time t for a given trace file. During the replay process, we can throttle the I/O injection rate if more than n requests have been issued but not yet completed. This throttling creates a small pause and helps the I/O stack to reduce the artificially high number of in-flight requests. This approach dynamically controls the number of the in-flight I/O requests and protects them from going beyond a predetermined limit before replaying each I/O request.

Note that we round up the dynamic limit of in-flight requests to the next power of two number. If we limit the worker thread to keep at most n requests in-flight, it can replay I/O requests at the same pace as they were captured, but cannot speed up even when some I/O requests fall behind the replay schedule due to unexpected latency of system calls. Therefore, it will negatively impact the timing accuracy of the replay process. On the other hand, if we do not put any limitation on the in-flight I/O count, the replay thread tries to catch up as fast as possible and issues too many I/O requests in a short period of time which resulting in *Response-Time Trashing*. Therefore, we round up the predetermined limit of in-flight requests to the next power of two number and let the replay threads speed up slowly to catch up with the schedule if needed.

2.4. I/O Request Bundling

If the inter-arrival time between two consecutive I/O requests is shorter than the average latency of an I/O submission system call, it is impossible for a single worker thread to submit both requests one after another on time with separate I/O submission system calls. On the other hand, if we deploy two worker threads, it is possible to submit both I/O requests on time, but they may arrive out-of-order to the SAN controller due to the variations of system call latency and device driver queue processing. Therefore, the high-fidelity replay engine of *hfplayer* bundles these consecutive requests into one group and lets the *libaio* unroll it on a single worker thread context within the kernel space. As a result, not only the I/O requests arrive at the SAN controller in order but there is less overhead of kernel to user-space context switch involved in the I/O submissions of multiple I/O requests. We recommend bundling multiple I/O requests before the submission if the inter-arrival time between consecutive I/O requests is less than a certain threshold. This inter-arrival time threshold is proportional to the system performance to execute I/O submission system calls. We will discuss this threshold value in Section 5.1.5.

3. Dependency and Scalable Replay

In most applications, there are dependencies that exist between I/O operations. A simple case would be an application that reads an input file, manipulates the data in some manner and writes the result to an output file. In this case, there is a dependency that exists between the read and the write since the write cannot occur until the read and the data manipulation have completed. Therefore, the actual issue time of an I/O request in the application is relative to the latency of its dependent I/Os as well as the data manipulation time.

The I/O latency is determined by the storage device performance and I/O queue depth. On the other hand, the data manipulation time is determined by the computing power of the server. If we replay a captured I/O trace in an environment similar to the one where the application was run initially (i.e., an unscaled environment), both I/O latency and data manipulation time would be the same as the original environment (with a reasonable pre-conditioning step). Therefore, I/O dependencies are maintained by simply replaying the I/O requests in the sequence that they were originally captured.

However, if we want to scale the workload replay to emulate running the application on a faster storage system or host (i.e., a scaled environment), the I/O latency or data manipulation time would be different from those in the original environment. Therefore, simply replaying the I/O requests in the captured order and time does not necessarily maintain real application I/O behavior. The application might issue I/O requests with a different order and perhaps shorter inter-arrival time in a scaled environment. The sequence between dependent I/O requests is actually maintained by the application and should be respected during replay. Therefore, an accurate scalable workload replay tool should be able to estimate these I/O dependencies from a captured trace and issue an I/O request after the completion time of its dependent I/O requests plus a data manipulation time.

Unlike traditional data dependency analysis in the compiler theory, it is not possible to precisely declare dependencies based on block I/O trace information without accessing application source code. Moreover, the file system semantics are not available in a block I/O request and it is not practical to derive dependencies similar to ARTC and TBBT replay approaches [24, 25] based on the file system level operations. Moreover, a set of I/O requests considered as independent from the point of view of either application or file system might become dependent with each other in the block layer since they are pushed into the same I/O queues and processed in order by the same I/O logic. Therefore, our best effort is to take a heuristic method and speculate potential I/O dependencies from a given trace and propagating I/O-related performance gains along the dependency chains during a workload replay on a faster (scaled) storage device. Finally in Section 5.2 we demonstrate the accuracy of our heuristic method by comparing the performance characteristics of the replayed workloads to real workloads generated by the applications on the scaled environment.

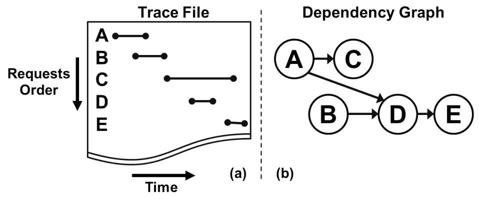
3.1. Inferring Dependency from Block I/O Trace

A block I/O trace usually includes information about the read and write request types, logical block address, transfer size and timestamps of issuing and completing I/O requests. Unfortunately, this information of the I/O operations itself does not help inferring the dependencies. We try to infer request dependencies using the timing relationships between these operations in the workload trace.

Inferring potential I/O dependency might create both false positives and false negatives. A false positive is to mistakenly identify two requests as dependent with each other, whereas they are independent. Therefore, they are replayed from a single dependency chain while they originated from separate chains in the application. A false negative is the prediction that mistakenly considers two dependent requests as independent. Therefore, they are replayed from separate dependency chains, while their relative order and timing are ignored. Consequently, the performance gains originated from the scaled system are not properly distributed among these requests.

False negative dependencies are more destructive for the workload replay accuracy on a scaled system because falsely identified independent requests can take advantage of any performance gains and replay as fast as possible with an arbitrary order that is not reproducible by the original application. On the other hand, false positive dependencies are more conservative since they are only allowed to take advantage of performance gain in a single dependency chain, while they could issue independently and perhaps even faster by the original application.

Moreover, some of these false positive dependencies may actually help improve the workload replay accuracy. For example, consecutive dependent I/O requests with a very long inter-arrival time are probably independent from the application point of view. They may be generated by independent threads at different times or the same thread with a long idle time. However, *hfplayer* still considers these consecutive requests as dependent to preserve the long idle time during the workload replay task. Otherwise, such a long think time will be dismissed if these requests are being considered as independent and will be replayed around the same time from separate dependency chains.



$$\begin{tikzcd} A \arrow[r] \arrow[dr] & C & \\ B \arrow[r] & D \arrow[r] & E \end{tikzcd}$$

Figure 2: Example of Block I/O Dependency Graph

Therefore, it can be beneficial for a workload replay to maintain some of these false positive dependencies.

hfplayer takes the following conservative approach to infer I/O dependencies with a minimum false negative prediction rate. First it assumes all consecutive pairs of requests are dependent, meaning that the second request depends on the outcome of the first request. Since no independent requests are detected initially, the false negative prediction rate would be zero. However, the false positive rate is at a maximum value. Second, it scans the I/O trace and tries to reduce the false positive rate by excluding those requests that are impossible to be dependent with each other. Finally, it searches the remaining list of dependent I/O requests and removes redundant dependencies to simplify the dependency graph.

Requests that are impossible to be dependent on each other are identified solely by their timing information in the trace file. Each block I/O operation in the request trace has a start time and a completion time. Start time is the time-stamp indicating when a request arrives at the trace capture point and the completion time is the time-stamp indicating when a request completion signal is returned back to the trace capture point in the I/O stack. Figure 2-a illustrates a sample request trace where the start and completion times of a request create a time interval represented as a line segment capped by two dots. Figure 2-b illustrates the dependency graph that is constructed based on these time intervals. We use traditional directed acyclic graph to visualize I/O dependencies in the trace file. During workload replay, the graph dependency edges implies that child request will be issued after the completion of the parent request plus a manipulation time associated with the parent. *hfplayer* distinguishes independent requests without introducing false negative in the following scenarios:

Overlapping Requests: A pair of I/O operations are identified as independent if they have overlapping time intervals since the first request finishes after the start time of the second request. Therefore, it is impossible for the second request to depend on the outcome of the first request. For example, Requests A and B in Figure 2 are independent since they are identified as overlapping requests.

Short Think Time: Think time is defined as the time duration starting from the completion of the first operation and ending at the start of the second operation. A pair of I/O operations with a very short think time are also identified as an independent pair. The threshold value of this short think time varies depending on the processing power of the application server and represents the time it takes to push up the completion signal of the first request from the capture point to the file system layer plus the time to issue the second operation from the file system. Therefore, it is impossible for the second request to depend on the outcome of the first request with a very short think time. Although an artificially large threshold value for the think time helps reduce the false positive rate, it may also introduce a false negative since a pair of dependent I/O requests is falsely marked as independent. Therefore, we take a conservative approach and just assign the smallest possible threshold of think time for a consecutive pair of I/O requests. This would help to keep the minimum false negative rate. This threshold time is around $10\mu s$ in our test environment where the trace capture point is located in the SAN controller. For example, the think time between Requests B and C in Figure 2 is too short such that they are considered as independent requests.

After excluding independent requests based on their timing positions, *hfplayer* scans the remaining list of dependent I/O requests, constructs a dependency graph and removes redundant dependencies. A pair of I/O operations has a redundant dependency in the graph if there is an indirect path from the first to the second operation in the graph. Therefore, removing redundant dependencies does not affect semantic information of the graph. For example, the dependency between Requests A and E in Figure 2 is maintained implicitly since Request E waits for the completion of Request D and Request D starts after completion of Request A. Therefore, there is no need to add another dependency edge from Request A to Request E.

3.2. Scalable Replay According to I/O Dependency

A scaled server may be capable of running the application at a faster pace and pushing more I/O requests into the I/O stack with less think time. On the other hand, a scaled storage device may pull more I/O requests out of the stack and process the requests faster with shorter response-time (I/O latency). An accurate I/O replay on a scaled system not only requires I/O dependency prediction, but also requires differentiation between server and storage scale factors.

Figure 3 illustrates the way that block I/O replay tools issue Requests A and B on a scaled storage device where Request B is dependent to the outcome of Request A. We assume that the workload is replayed from a server with the performance characteristics similar to that of the original application server. Therefore, it is expected to see the same think time and reduced I/O latency for both

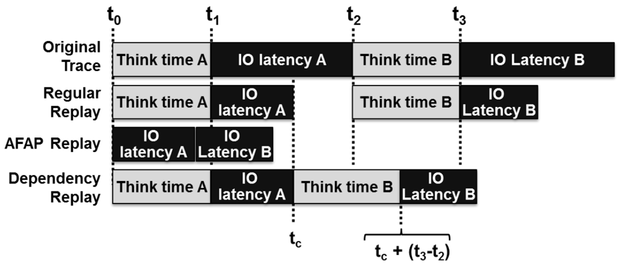


Figure 3: Replay methods on the Scaled Storage

Requests A and B. Typical workload replay tools either ignore the I/O latency reduction (i.e., issue I/O requests based on their original scheduled time: regular replay), or ignore the scheduled time (i.e., issue I/O requests As Fast As Possible: AFAP replay). We believe neither one of these approaches is accurate. The regular replay approach issues the requests at the same speed as they were captured on the unscaled storage device. The AFAP replay approach issues I/O requests too fast which can violate the I/O request dependencies and accumulates a large number of in-flight I/O requests that severely impacts the response-time and throughput accuracy of the reproduced workload.

hfplayer emulates the execution of the original application on a scaled storage device by preserving the original think times and dynamically adjusting the issue times of I/O operations according to their dependent parent's completion time. If the completion signal of Request A (parent) arrives at t_c, the issue time of Request B (child) will be set to $t_c + (t_3 - T_2)$ during the replay process.

Note that only a few I/O requests in the beginning of the workload trace that are in-flight at the same time do not have parents. Therefore, almost all I/O requests in the workload trace have at least one dependent parent that form multiple dependency chains. The total number of parallel dependency chains at any point of the time obviously cannot exceed the maximum number of in-flight I/O requests recorded in the workload trace file.

4. hfplayer Architecture

The main components of *hfplayer* are illustrated in Figure 4. The dependency analyzer module is used only during the scaled replay mode. It takes the trace file with a short inter-arrival time threshold from the inputs, identifies I/O request dependencies and creates a dependency graph based on the method as we have described in Section 3. Finally, this module annotates the original trace file and creates a new intermediate trace file with a list of dependent parents for each I/O request.

We have deployed four types of threads for a replay session, the worker threads, a harvest thread, a timer thread and the main application thread. The main application thread is responsible for initializing and preparing the required data structures for other threads. These data structures include individual I/O request data structure with optional request bundling and dependency information as well as internal I/O queues dedicated to each worker thread. We have carefully aligned these data structures to the processor cache line size to avoid multiple cache misses when *hfplayer* touches these data structures.

In the unscaled replay mode, each worker thread picks an I/O request from its request queue and issues the request according to the scheduled issue time recorded in the I/O data structure. It also makes sure that the I/O stack is not operating in the response time trashing mode by keeping track of the total number of I/O requests issued and completed as we discussed in Section 2.3.

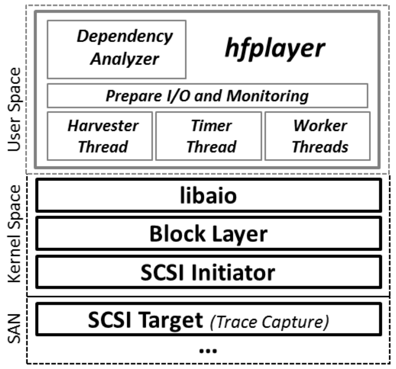


Figure 4: hfplayer *Components on Linux I/O Stack*

In the scaled replay mode, a worker thread issue an individual I/O request after it receives the completion signal of all of its parents plus their corresponding think times. After a worker thread successfully issues an I/O request to the storage controller, its I/O completion signal will be picked up by the harvester thread. Then, the harvester thread decrements the number of in-flight parents counter in its children's data structure. This process eventually reduces individual children's counter to zero and then let other worker threads to issue its children requests in a dependency chain.

Finally, a timer thread is used to share a read-only timer variable to all worker threads with nanoseconds accuracy. Worker threads in the unscaled replay mode compare this shared timer value with the scheduled issue time of the current request to make sure I/O requests are submitted on time. We add a fixed offset to the timer value to make sure all worker threads have enough time to start and wait for the submission of the very first I/O request.

5. Evaluation

We evaluate the replay accuracy of the *hfplayer* in the following two scenarios. First, we evaluate its **high-fidelity unscaled replay engine** in a unscaled environment and compare it with existing replay tools in Section 5.1. Then, we evaluate its **scalable replay engine** on scaled storage devices in Section 5.2. These experiments are done with Linux 3.13 kernel running on a 2x6-Core Xeon X5675 server with 64GB memory and 8Gbps Fiber Channel connection to a NetApp E-Series SAN controller.

We have created three SAN volumes for these experiments with RAID-0 configuration. The first volume is composed of four enterprise SAS solid-state drives (4xSSD volume). The second volume is composed of nine enterprise SSDs (9xSSD volume) and the last SAN volume is composed of two 15K RPM SAS hard disk drives (2xHDD volume). We have disabled the read and write cache since warming up the SAN controller caches is out of the scope of this work. Note that the performance numbers presented in this section do not represent the actual capability of the products used in these experiments.

5.1. Workload Replay on the Unscaled Storage

In this experiment, we collect 12 I/O workload traces, then replay them with multiple replay tools and quantify the unscaled workload replay accuracy. At first, we compare the *hfplayer* replay accuracy with other existing block I/O replay tools. Next, we evaluate its multi-threading and bundling features.

5.1.1. Workload Types: An accurate replay tool should be able to faithfully replay any workloads in a wide intensity spectrum from low to high throughputs. However, as we mentioned earlier it is a challenge to maintain the replay accuracy for intensive and high throughput workloads. Therefore, we need a tunable synthetic I/O workload to demonstrate when the accuracy of existing replay tools drops as we increase the I/O throughput in this experiment. Although it is more appealing to replay the workload trace of a realistic application, we use synthetic workloads in this experiment since most of the open-source enterprise and HPC applications does not use a raw volume without a file system. Therefore it require a large cluster of servers and huge parallelism to overcome the application and file system layer overheads and stress the storage controller with an intensive I/O workload. For example, a recent study reported that 168 Lustre nodes (OSS and OST) are required to create a storage space with 100K IOPS [7]. Obviously, producing such a workload is a challenge not only for us with limited hardware/software resources but also for the research community to reproduce and validate our research results. Therefore, we have selected synthetic workloads that can stress the SAN controller up to 200K IOPS and then demonstrate how *hfplayer* can replay these workloads with high-fidelity on a similar storage device.

We have used *FIO* and its *libaio* engine to generate 11 synthetic I/O workloads that cover low to high throughput intensity spectrum on the raw 4xSSD volume. We have also selected *Sequential Write* and *Random Read/Write* with a default 50/50 split for the I/O patterns. These two patterns then are used to generate 10 workloads with the incremental throughput tuned by *iodepth* values of 16, 32, 64, 128 and 512. Finally, we generated another workload with the I/O pattern of *Sequential Read* and *iodepth* of 512 to create a very high-throughput workload for evaluation of multi-threading and request bundling features of *hfplayer*. All of these workloads are generated for the duration of 60 seconds. The traces of block I/O requests that have been generated by these workloads are collected from the SAN controller SCSI Target layer. These trace files are then used in our replay experiments. *hfplayer* input trace file format is compatible with SNIA draft of block-trace common semantics [17] and does not include proprietary fields.

5.1.2. Replay Tools: We compare the workload replay accuracy of *hfplayer* with two publicly available block I/O replay tools. First, we use *blkreplay* [15] which is developed by a large web hosting company for the performance validation of block storage devices used in its data centers. *blkreplay* is built with a synchronous multi-threaded I/O engine as we mentioned in Section 2 and suffers from inter-thread coordination overhead. We configure it with *no-overhead* and *with-conflicts* to disable its write-verify and write-protection mode in order to make an apples to apples comparison with other tools. Moreover, we set the maximum thread count to 512 since some of our workloads carry-on such a high number of in-flight I/O requests.

Second, we use *btreplay* [8] which is developed originally to replay workloads that are captured with the *blktrace* tool on Linux. It uses *libaio* with an asynchronous I/O engine like *hfplayer* and has a similar feature to combine the I/O requests into a bundle if they fit in a fixed time window with a fixed request count per bundle. The bundle time window and the maximum request count are statically set in the command line and are not dynamically set according to the workload intensity. However, as we mentioned in Section 2.4, *hfplayer* can dynamically adjust the bundle size since it bundles requests together if their issue time is very close to each other. Moreover, we just use a single worker thread in the *btreplay* since it cannot replay a single trace file using multiple worker threads. We configure *hfplayer* to work with a single worker thread for a fair comparison as well and evaluate its multi-threading and bundling features separately in Section 5.1.5. We do not use *FIO* replay engine since it is similar to *btreplay* and uses asynchronous I/O engine. Moreover, it has been reported in the open-source community that *FIO* is not as accurate as *btreplay* [1].

5.1.3. Accuracy Metrics: An accurate unscaled workload replay should maintain both temporal and spatial characteristics of the original trace workload. Temporal characteristics of an I/O workload are usually quantified by I/O throughput and response-time. We use relative error percent of average response-time to demonstrate the response-time accuracy of the workload replay. Moreover,

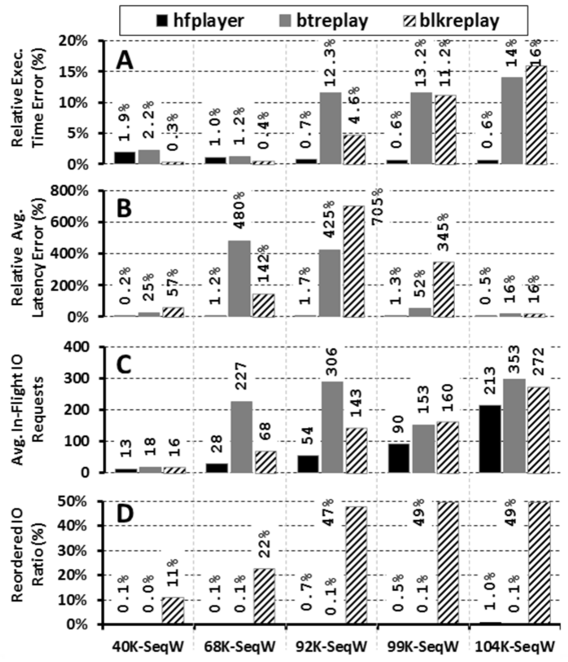


Figure 5: Replay Accuracy for the Sequential Write Workloads

determining factor of I/O throughput is the total execution time since the number of replayed requests is the same in each replay experiment. Therefore, we use relative error percentage of execution time to demonstrate the I/O throughput accuracy of the workload replay. The spatial characteristics of the workload replay remain the same if all I/O requests arrive in the storage device(s) with the same order that is presented in the original trace. Therefore, we have used the Type-P-Reordered metric [12] to measure the number of requests that are reordered in the replayed workload.

5.1.4. Workload Replay Accuracy Comparisons: Figure 5 demonstrates the accuracy of the three replay tools that we have tested using five sequential write workloads. The shared horizontal axis represents the demanding throughput of the captured workload in IOPS. The vertical axes A, B and D represent three accuracy metrics discussed earlier. First, axis A shows the relative execution error of the replayed workload. While all three replay tools can reproduce low throughput workloads with negligible timing errors, *btreplay* and *blkreplay* cannot keep up when throughput demand increases from 92K to 104K IOPS and cannot finish the replay task on time.

Second, axis B shows the relative error of average I/O response-time during the workload replay compared with the original workload. Once again all three tools have a negligible response-time error to replay the low throughput workload (40K IOPS). However, both *btreplay* and *blkreplay* fail to replay with accurate response-time for higher throughput demanding I/O workloads. Starting with 68K IOPS workload, these tools build an artificially high number of in-flight I/Os in the I/O path as we show in axis C which results in *Response-Time Trashing*. The I/O response-time error of the replay task for the most demanding I/O workload (104K) is negligible because the I/O path is already overloaded and achieving its maximum response-time at this throughput rate. Therefore, the internal overhead of *btreplay* and *blkreplay* prevents them from achieving the target throughput and causes them to finish the replay task with 14% and 16% relative execution time error respectively.

Finally, axis D shows the ordering accuracy of the replayed workload. *blkreplay* is based on a multi-threaded synchronous I/O engine. Therefore, it is expected to see it reorders most of I/O requests in the workload. We believe reordering can significantly change the workload characteristics. In this case, it converts a sequential write workload into a half sequential, half random workload. *btreplay* and *hfplayer* both use the single threaded async I/O engine and can submit all I/O requests in order but a negligible number of these requests are reordered in the I/O stack by the SCSI Initiator since it operates in the *Simple* TCQ mode which is inevitable.

Figure 6 shows the replay accuracy for mixed random read/write workloads. First, axis A shows all three replay tools are capable of replaying mixed workloads with a negligible error. That is because the overhead of the I/O stack is lower for read/write workload compared with previous write-only workload. Typically write requests are more resource consuming since writes carry a payload that requires extra protection for data consistency and durability both in the Kernel and SAN controller I/O stacks. Second, axis D shows both *blkreplay* and *btreplay* fail to replay the workload with accurate response-time which is directly impacted by the number of in-flight I/O requests shown by axis C. Note that the only workload that *hfplayer* does not provide the best response-time accuracy is on the 99K IOPS workload, where its relative error rate is 52%, whereas *blkreplay* error rate is 16%. That is because *blkreplay* managed to replay the workload with a lower number of in-flight I/O requests. However, *blkreplay* cannot maintain a low request ordering accuracy at the same time. As we show in axis D, about 34% of requests that are issued with this tool at 99K IOPS workload arrived in the controller out of order.

Note that the number of reordered requests in the mixed read/write workload is significantly more than sequential write workload for *btreplay* and *hfplayer* (comparing axis D in Figures 6 and 5). These requests are submitted in order with a single worker thread in both replay tools. However, they are reordered in the SCSI initiator layer since read and write requests have different service time. Usually read requests are serviced faster since the read I/O path is faster on a SAN controller without a caching layer. Therefore, the queue tags that are allocated for the

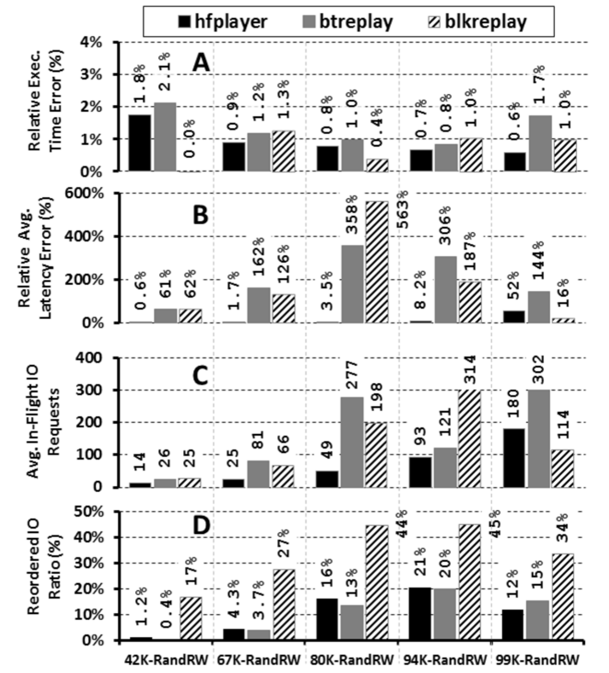


Figure 6: Replay Accuracy for the Random R/W Workload

read requests in the SCSI initiator layer are deallocated faster than the tags allocated for writes requests. However, in the write-only workload, all the tags are allocated and deallocated with the same rate and the SCSI initiator queue will operate similar to a FIFO queue.

5.1.5. Multi-threading and Request Bundling: We did not evaluate these two features of the *hfplayer* in the previous experiments to make an apples to apples comparison with other replay tools. However, these two features are useful during the replay of very high throughput workloads where the other replay tools are incapable of maintaining replay accuracy as we elaborated in the previous subsection. In this experiment, we go extreme and replay a very high throughput sequential read workload with 232K IOPS. This workload is captured and replayed on a 9xSSD volume, while the previous experiments were done on a 4xSSD volume.

Figure 7 shows the replay accuracy of *hfplayer* where multi-threaded and bundling features are used. The horizontal axis shows the maximum inter-arrival time between two requests to fit into a bundle. Obviously, *hfplayer* forms larger bundles with a larger maximum inter-arrival time value. We have replayed the workload with one to eight threads for each time to evaluate the multi-threading feature as well. On the vertical axes, we just show the relative execution time error (or throughput accuracy) and reordered I/O ratio. We do not show the average response-time accuracy since the I/O path in all of these experiments is fully loaded. Therefore, I/O response-times are all saturated and close to expectation. We have described a similar behavior in Figure 5-B that happens for all replay tools during replay of a high-throughput 104K IOPS workload. Figure 7 shows that *hfplayer* cannot finish replaying the 232K IOPS workload on time when the bundling feature is disabled. Enabling the multi-threaded feature helps to reduce the relative execution time error from 28% to 27% and 21% with two and four worker threads respectively. The improvement is minimal and the resulting error rate is still high. Moreover, using eight worker threads does not improve the replay accuracy further due to extra threading overhead to acquire block layer queue lock. On the other hand, adding more worker threads significantly increases the reordered I/O rate.

In contrast, when request bundling is enabled (for example, set bundling threshold to $20\mu s$), the ordering errors are reduced to around 30% in multiple worker threads mode. Moreover, multiple worker threads help *hfplayer* submit all I/O requests on time and reduce the execution time error rate. As we increase the bundling threshold, we see a negligible change in the accuracy metrics. That is because *hfplayer* can fit up to 256 requests in a bundle and increasing the bundle threshold cannot create a larger bundle when almost all bundles are full. The maximum bundle size limit is an implementation trade-off that forces the bundle data structure to fit in a CPU cache line. Therefore, the replay task does not suffer from multiple cache line misses when it sends a bundle to the kernel.

Finally, Figure 7 shows a significant amount of I/O requests are reordered in the single threaded mode when bundling is enabled. For example, 29.3% of requests arriving in the SAN controller are out of order when we set the bundling threshold to $30\mu s$. All of these requests are inserted into the SCSI initiator queue in order using a single thread that mostly lives in the kernel space. In other words, the *io_submit* system call unrolls a bundle and issues individual I/O requests from the kernel space and does not context switch to the user space frequently. Therefore, the tag allocation in the SCSI initiator queue takes place at a faster pace compared with deallocation. As a result, the SCSI driver cannot allocate in order tags when a burst of tag allocation requests arrive during the unrolling of a bundle in the kernel space.

5.2. Evaluate Scalable Workload Replay

In this subsection, we evaluate the scalability feature of the *hfplayer* and its scaled replay engine. The ultimate goal of this experiment is to evaluate how accurate *hfplayer* can replay a workload on a faster storage device, given a workload trace file that is captured from a slower storage device. First, we will describe the workloads that we have used for this experiment. Then we describe the evaluation methodology and finally we present the evaluation results.

5.2.1. Workload Types: As we described in Section 3, the dependency replay engine of *hfplayer* tries to emulate the application behavior by speculating the I/O dependencies from a given trace file and replay I/O requests according to their dependencies. Therefore, we could not use synthetic workloads for this experiment since they do not contain realistic dependency information that can be emulated. Therefore, we have tried to use realistic applications on top of a file system for this purpose. Instead of using a commercial I/O intensive application (like an Oracle Database) or making our own I/O intensive application (like copying large ISO files), we have used *Filebench* [21] and *mkfs* to generate the workloads for this experiment. These applications are available in the public domain and can be used by others to reproduce the results.

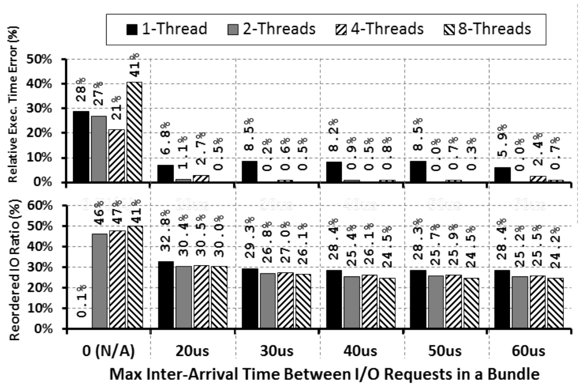


Figure 7: Multi-threading and Request Bundling Evaluation

The key features of the selected I/O intensive applications are a) they perform direct I/O instead of buffered I/O which eliminates the ext4 buffer cache impact on the I/O performance, b) they perform a meaningful task from the application layer and their durations depend only on the performance of the storage device, and c) the task execution time is fixed and reproducible on a single storage device.

According to these criteria, we have selected *Copyfiles* and *Createfiles* benchmarks from the *Filebench* suite. The first benchmark is configured to copy 100K files with 256 threads. The second benchmark is configured to create 50K files with mean directory width of 100 and mean file size of 16K using 16 threads and a 1MB request size. Both benchmarks are set to do direct I/O and the other configuration parameters are set to default. These two benchmarks run a meaningful I/O intensive task on the SAN volumes mounted with the *ext4* file system.

The *mkfs* application creates another I/O intensive workload during the file system initialization phase. We have created fixed size partitions on all SAN volumes and used *mkfs.ext4* utility without the lazy inode table and journal initialization to create a file system image. This forces *mkfs* to create all inode tree and journal data structures on the disk in the foreground.

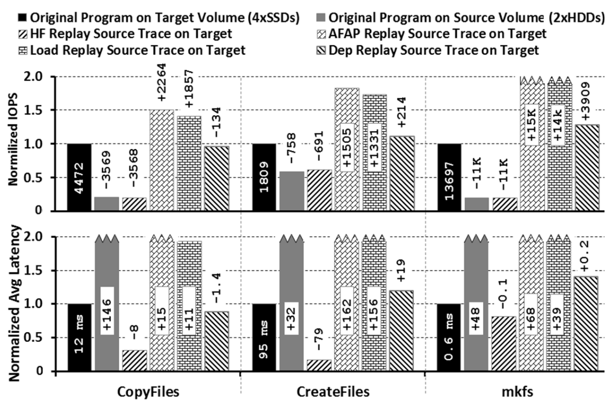


Figure 8: Replay Workloads Captured from 2xHDD on 4xSSD

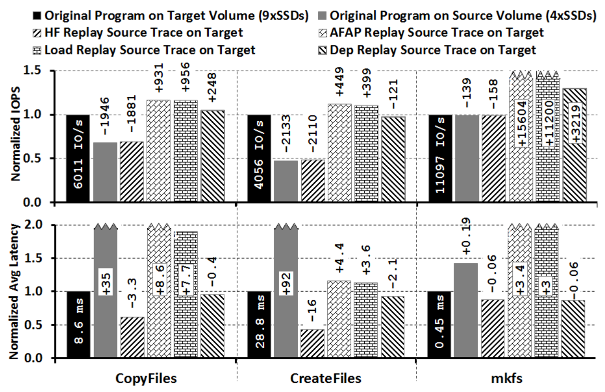


Figure 9: Replay Workloads Captured from 4xSSD on 9xSSD

5.2.2. Evaluation Methodology: We will describe our evaluation methodology with a simple example. Let us assume an I/O intensive application that executes on storage device A for 10 minutes. However, it takes just 5 minutes to run the same application on a faster storage device B. A scalable replay tool should be able to take the trace file of the I/O workload from storage device A (which lasts for 10 minutes) and replay it on device B in 5 minutes. In other words, storage device B takes almost the same workload during the replay as it received when the real application was running. In this example, storage device A is the *Source* and B is the *Target* storage device.

In practice, the source is the slower storage device that is installed in the production site and the target is a faster storage device that is under development or validation test. The workload is captured from the source (on the production site) and replayed on the target storage (in the lab). The expectation is that the replay tool will generate the same workload that the production application would have generated on the target storage and quantifies how much it can improve application performance.

Our methodology to quantify how *hfplayer* can meet such an expectation is as follows. First, we execute the I/O intensive applications on 2xHDD, 4xSSD and 9xSSD volumes for multiple times, capture their I/O workloads from the SAN controller and make sure the execution time of captured workloads are repeatable. We have validated that these applications are repeatable with less than 2.7% relative standard deviation. Then we consider the 2xHDD volume as a source and 4xSSD volume as the target for the replay. This means that the trace files captured on 2xHDD volume are replayed on 4xSSD volume. Finally, we compare the execution time and I/O response-time with the original application trace files that were captured from 4xSSD in the first step. We do the same steps to take 4xSSD volume as a source and 9xSSD as a target.

We use *hfplayer* to replay the captured workload from the source on the target storage in the following four replay modes to make our evaluation more comprehensive. First, we replay with the **HF** or high-fidelity replay engine. Note that this is the same replay mode that we have used in Section 5.1. Second, we replay with the **AFAP** replay mode which ignores the scheduled timing and tries to replay I/O requests on the target storage device as fast as possible. Note that other block I/O replay tools like blkreplay recommend this replay mode to replay a workload on a scaled storage [15]. Third, we replay with the **Load** replay mode which is similar to AFAP, but it only tries to dynamically match the number of in-flight I/O requests as described in a previous work [16]. Finally, we replay with the **Dep** replay mode which is based on the dependency replay engine of the *hfplayer*. It estimates I/O dependencies and replays requests according to those dependencies as we described in Section 3.

5.2.3. Result Discussion: Figure 8 shows the results of taking the captured workload from the 2xHDD volume (source) and replaying it on the 4xSSD volume (target). In this figure, the horizontal axis is the workload type. The top vertical axis is the workload IOPS normalized to original application IOPS running on the target storage device. The bottom vertical axis shows the average response-time again normalized with what is expected to see on the target storage device. An accurate replay should have a patterned bar that is very close to the solid black bar. The numbers on the patterned and gray bars show the error values or how far each bar is from the solid black bar.

This figure shows that the dependency replay mode can accurately replay captured workload from the 2xHDD volume on the 4xSSD volume in terms of both IOPS and average response-time. The IOPS generated with high-fidelity replay mode matches with the low throughput of the source volume (gray bar) and its its response-time is significantly lower than the original application response-time on the target storage (black bar). AFAP and Load replay mode both replay the workload on the target volume with a significantly higher throughput. As a result, they queue more requests than expected in the I/O path and inflate the response-time. Even the in-flight I/O rate control mechanism that is embedded in the Load replay mode does not help to slow down the workload replay and match with the original program I/O performance running on the target volume.

Finally, Figure 9 shows the results of taking a captured workload from the 4xSSD volume and replaying it on the 9xSSD volume. This figure shows that dependency replay mode of *hfplayer* can accurately replay a workload on a scaled storage even if the performance variation between the source and target is not significant. For example, *mkfs* is a single threaded application that typically does synchronous I/O with one in-flight request at a time. Therefore, as we see in this figure, both the source and target storage devices have the same IOPS value (solid black and solid gray bars). That is because only one SSD produces the throughput at any point in time with one in-flight I/O request. This is the worst case for *hfplayer* with dependency replay with an IOPS error rate less than 30%, compared to less than 10% IOPS error rates in all other cases.

6. Related Work

Various trace replay tools have been developed at file system level. For example, Joukov et al. developed Replayfs [9], a tool that can replay file system I/O traces that were captured at the VFS level. Zhu et al. proposed TBBT [25], an NFS trace replay tool that automatically detects and recovers missing file system I/O operations in the trace. Mesnier et al. proposed //TRACE [11] for replaying traces of parallel applications. It achieves a high accuracy in terms of inter-node data dependencies and inter-I/O compute times for individual nodes by utilizing a throttling technique. More recently, Weiss et al. designed ARTC [24], a new method of replaying system call traces of multi-threaded applications which can explore some non-deterministic properties of the target application. However, as we mentioned in Section 1 none of these replay tools are capable to reproduce high-throughput workload due to intrinsic file system overhead. More recently, Pereria et al. compares the replay accuracy of ARTC with TBBT [13].

There are several other block I/O replay tools. Liu et al. designed TRACER [10], a replay tool used for evaluating storage system energy efficiency. It can selectively replay a certain percentage of a real world block I/O trace to reach different levels of workload intensity by filtering trace entries uniformly. Anderson et al. proposed Buttress [6] as a toolkit to replay block traces with a loose timing accuracy of $100\mu s$ more than 10 years ago. This tool uses synchronous I/O and thus requires instantiating a great number of threads in order to achieve a high number of outstanding I/O requests on the target storage system. Therefore, its replay performance and scalability are limited by the threading overhead and cannot keep up with the capabilities of modern SAN storage systems. Sivathanu et al. proposed a load-aware trace replay [16] that aims to preserve the same I/O load pattern of the original application traces irrespective of the performance of the target storage system. However, we have evaluated this technique in Section 5.2 and demonstrated that it cannot replay a workload on the scaled target storage accurately.

More recently, Tarihi et al. proposed DiskAccel [22], a sampling methodology to accelerate trace replay on conventional disk drives. DiskAccel uses a weighted variant of the K-Means clustering algorithm to select representative intervals of the I/O trace file. These I/O intervals instead of the whole trace are then replayed on the target disk drive. Therefore, a week long captured I/O trace file can be replayed in about an hour, while maintaining the same average I/O response-time. Moreover, Tarasov et al. proposed a flexible workload modeling technique that extract a mathematical model from the block trace [20]. This model is then used as an input for a benchmark tool to reproduce the workload. These trace reduction and modeling methodologies are complementary to our work and can be used to shrink the trace size and I/O workload duration with *hfplayer* as well. DiskAccel also implements a method to enforce I/O requests dependency during the replay job. However, due to the lack of block I/O dependency information, it assumes all reads requests are dependent and all write requests are independent. In contrast, *hfplayer* infers dependency information without such an unrealistic assumption.

Finally, Tarasov mentioned a few limitations of the workload replay on a scaled storage [18]. He described the dependency replay as a viable approach but claimed that approximation of I/O dependencies from the block layer can add extra dependencies that does not exist in the original workload. Therefore, the workload replay effort might not be as accurate as workload modeling effort. In this work, we have demonstrated a method to make an approximation of the I/O dependencies and found that the dependency workload replay can reproduce original application workload on a scaled storage device.

7. Conclusions and Future Work

In this paper, we have introduced new methods to replay intensive block I/O trace in a scaled or unscaled environments with more accuracy. First, we have proposed a detailed analysis of various points preventing an accurate replay in Linux I/O stack. Second, we have considered the notion of dependency between block I/O requests and then described how the *hfplayer* infers and replays events in a dependency aware fashion on a scaled system, efficiently propagating I/O-related performance gains along dependency chains. Finally, we have provided a careful evaluation of the *hfplayer* in both scaled and unscaled environments. In the future, we seek to port our replay tool to IBM's AIX operating system.

8. Acknowledgments

We thank our shepherd, Remzi Arpaci-Dusseau and Matias Bjørling, and the anonymous reviewers for their comments and suggestions. This work has been supported by NSF I/UCRC Center for Research in Intelligent Storage (CRIS) and the National Science Foundation (NSF) under awards 130523, 1439622, and 1525617 as well as the support from NetApp.

References

[1] fio mailing list, re: fio replay.

[2] Intel storage performance development kit (spdk) official web site.

[3] Linux bug fix patch: Enable sysfs nomerge control for i/o requests in the plug list.

[4] Spc-1 benchmak results fot top 10 block storage by price-performance.

[5] ANDERSON, E. Buttress, a cautionary tale. In *File and Storage Systems Benchmarking Workshop* (2008).

[6] ANDERSON, E., KALLAHALLA, M., UYSAL, M., AND SWAMINATHAN, R. Buttress: A toolkit for flexible and high fidelity i/o benchmarking. In *Proceedings of the 3rd USENIX Conference on File and Storage Technologies* (2004).

[7] BOURILKOV, D., AVERY, P. R., FU, Y., PRESCOTT, C., AND KIM, B. The lustre filesystem for petabyte storage at the florida hpc center. In *HEPiX Fall 2014 Workshop*.

[8] BRUNELLE, A. D. btrecord and btreplay user guide.

[9] JOUKOV, N., WONG, T., AND ZADOK, E. Accurate and efficient replaying of file system traces. In *FAST* (2005), vol. 5, pp. 25–25.

[10] LIU, Z., WU, F., QIN, X., XIE, C., ZHOU, J., AND WANG, J. Tracer: A trace replay tool to evaluate energy-efficiency of mass storage systems. In *Cluster Computing (CLUSTER), 2010 IEEE International Conference on* (2010), pp. 68–77.

[11] MESNIER, M. P., WACHS, M., SIMBASIVAN, R. R., LOPEZ, J., HENDRICKS, J., GANGER, G. R., AND O'HALLARON, D. R. //trace: parallel trace replay with approximate causal events. USENIX.

[12] MORTON, A., CIAVATTONE, L., RAMACHANDRAN, G., SHALUNOV, S., AND PERSER, J. Packet reordering metrics. *IETF internet-standard: RFC4737* (2006).

[13] PEREIRA, T. E., BRASILEIRO, F., AND SAMPAIO, L. File system trace replay methods through the lens of metrology. *32nd International Conference on Massive Storage Systems and Technology (MSST 2016).*

[14] SCHROEDER, B., WIERMAN, A., AND HARCHOL-BALTER, M. Open versus closed: A cautionary tale. In *Proceedings of the 3rd Conference on Networked Systems Design and Implementation, NSDI* (2006), vol. 6, p. 18.

[15] SCHBEL-THEUER, T. blkreplay and sonar diagrams.

[16] SIVATHANU, S., KIM, J., KULKARNI, D., AND LIU, L. Load-aware replay of i/o traces. In *The 9th USENIX Conference on File and Storage Technologies (FAST), Work in progress (WiP) session* (2011).

[17] (SNIA), S. N. I. A. Block i/o trace common semantics (working draft).

[18] TARASOV, V. *Multi-dimensional workload analysis and synthesis for modern storage systems*. PhD thesis, Stony Brook University, 2013.

[19] TARASOV, V., KUMAR, S., MA, J., HILDEBRAND, D., POVZNER, A., KUENNING, G., AND ZADOK, E. Extracting flexible, replayable models from large block traces. In *FAST* (2012), p. 22.

[20] TARASOV, V., KUMAR, S., MA, J., HILDEBRAND, D., POVZNER, A., KUENNING, G., AND ZADOK, E. Extracting flexible, replayable models from large block traces. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies* (2012), vol. 12, p. 22.

[21] TARASOV, V., ZADOK, E., AND SHEPLER, S. Filebench: A flexible framework for file system benchmarking.

[22] TARIHI, M., ASADI, H., AND SARBAZI-AZAD, H. Diskaccel: Accelerating disk-based experiments by representative sampling. *SIGMETRICS Perform. Eval. Rev. 43*, 1 (June 2015), 297–308.

[23] TRAEGER, A., ZADOK, E., JOUKOV, N., AND WRIGHT, C. P. A nine year study of file system and storage benchmarking. *ACM Transactions on Storage (TOS) 4*, 2 (2008), 5.

[24] WEISS, Z., HARTER, T., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Root: Replaying multithreaded traces with resource-oriented ordering. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2013), SOSP '13, ACM, pp. 373–387.

[25] ZHU, N., CHEN, J., CHIUEH, T.-C., AND ELLARD, D. Tbbt: scalable and accurate trace replay for file server evaluation. In *ACM SIGMETRICS Performance Evaluation Review* (2005), vol. 33, ACM, pp. 392–393.