DITIS: A Distributed Tiered Storage Simulator

Edson Ramiro Lucas Filho[1], Lambros Odysseos[1], Yang Lun[2], Fu Kebo[2], and Herodotos Herodotou, *IEEE* [1,*]

***Abstract*—This paper presents DITIS, a simulator for distributed and tiered file-based storage systems. In particular, DITIS can model a distributed storage system with up to three levels of storage tiers and up to three additional levels of caches. Each tier and cache can be configured with different number and type of storage media devices (e.g., HDD, SSD, NVRAM, DRAM), each with their own performance characteristics. The simulator utilizes the provided characteristics in fine-grained performance cost models (which are distinct for each device type) in order to compute the duration time of each I/O request processed on each tier. At the same time, DITIS simulates the overall flow of requests through the different layers and storage nodes of the system using numerous pluggable policies that control every aspect of execution, ranging from request routing and data redundancy to cache and tiering strategies. For performing the simulation, DITIS adapts an extended version of the Actor Model, during which key components of the system exchange asynchronous messages with each other, much like a real distributed multi-threaded system. The ability to simulate the execution of a workload in such an accurate and realistic way brings multiple benefits for its users, since DITIS can be used to better understand the behavior of the underlying file system as well as evaluate different storage setups and policies.**

***Index Terms*—Storage Simulator, Distributed Data Storage, Tiered Storage, Performance Cost Models**

I. INTRODUCTION

THE inclusion of multiple storage and caching tiers consisting of multiple HDD, SSD, NVRAM, and DRAM devices (among others) are common in modern data storage systems, but require the development of new data management policies for controlling the flow, placement, and migration of data across the tiers. At the same time, it is hard to evaluate the impact of the tiers and their policies across different workloads as that would require constantly modifying and redeploying the storage system. Hence, the development and testing of such policies can quickly become a very cumbersome and time-consuming process. From the end-users' perspective, it becomes exceedingly difficult to (i) identify whether their workloads will execute efficiently on a particular multi-tiered system configuration, or (ii) select the best system configuration that will satisfy their requirements.

DITIS is a new distributed tiered storage simulator that can be used to address the aforementioned challenges by enabling its users to accurately simulate I/O flows and data storage operations for given workloads and system configurations. In particular, DITIS is able to represent a set of distributed nodes containing multiple storage tiers with different storage media and performance characteristics, as well as multiple levels of caches. DITIS processes a workload trace and simulates the execution of file system operations, which are guided by numerous data flow, caching, and tiering policies, while maintaining all metadata information and several statistics. As a result, developers can use DITIS to narrow down the design spaces, evaluate design trade-offs, test different setups and policies, and reduce prototyping efforts, while end users can use it to better understand the system's behavior and identify the system configuration that best satisfies their requirements.

Even though DITIS is a discrete-event simulator (i.e., it models operations as a discrete sequence of events), it does not follow the typical event-oriented or process-oriented models. Instead, DITIS adapts the **actor model** as a basic design principle [1]. As such, each key component is an actor that maintains its own private state, processes messages received from other actors, and sends messages to other actors. This enables the seemingly concurrent computation of actors that interact only through direct asynchronous message passing. In DITIS adaptation, all outgoing messages are associated with a simulated (virtual) time of submission, based on which DITIS schedules message delivery. The use of the actor model and other crucial design decisions resulted in a simulator that is:

- **Configurable:** With over 100 configuration parameters, DITIS can simulate a large variety of different system setups and scenarios. For example, a user can configure a system with multiple storage nodes, with up to 3 different persistent storage tiers, and up to 3 additional levels of caches, along with the performance characteristics of the storage media.
- **Extensible:** All key decisions made by a storage system are modelled as policies that can be replaced for changing the behavior of the system and the simulation. Currently, there are 39 policies that control every aspect of execution, including the routing of requests, data flow management, caching, tiering, and performance modeling.
- **Accurate:** DITIS utilizes fine-grained performance cost models at the level of individual storage devices and network data transfers while modeling (and costing) the flow of messages between the different system components.

Section II presents the design of DITIS. Section III presents the flow management of I/O requests. Section IV presents the device-specific performance cost models. Section V presents the experimental evaluation of DITIS. Section VI presents the related work. Finally, Section VII concludes the paper.

II. DESIGN AND ARCHITECTURE

This section presents the design and architecture of DITIS.

[1] E. R. Lucas Filho, L. Odysseos, and H. Herodotou are with Cyprus University of Technology, Cyprus (e-mail: edson.lucas@cut.ac.cy, lambros.odysseos@cut.ac.cy, herodotos.herodotou@cut.ac.cy)

[2] Y. Lun and F. Kebo are with Huawei Technologies Co., Ltd., China (e- mail: yanglun12@huawei.com, fukebo@huawei.com).

* Corresponding author.

DOI: 10.36244/ICJ.2022.4.3

A. Simulation Input

DITIS requires two input files for simulating a workload execution on a storage system. The first one is an *input workload trace* (in CSV) with each line corresponding to one file request to be processed by the storage system. A file request consists of (1) the process id of the application that submitted the request, (2) the submission epoch timestamp (in microseconds), (3) the file operation (open, close, read, write, or delete), (4) the name for the file to be accessed, (5) the offset of the file (in bytes) when reading from or writing to a file, (6) the length containing the amount of data to be processed (in bytes), (7) the current file size, and (8) the original duration of the operation (in microseconds). The original duration is ignored by the simulator but having it facilitates its comparison with the simulated duration time.

The second input is a *configuration file*, which defines the storage system and its behavior. DITIS is a very flexible and extensible simulator, and its configuration allows users to adapt the simulated storage system in many ways. For instance, users can resize internal components of the storage system (e.g., set 3 nodes, 10 SSDs per node), specify their performance characteristics (e.g., disk IOPS, RPM), as well as determine which combination of policies to use during the simulation.

B. Components

Figure 1 depicts the overall architecture and key components of DITIS, inspired by modern hybrid storage systems such as Huawei OceanStor series. Next, we present the description of each component.

Workload Level: The *Trace Parser* is responsible for parsing requests from the input trace, validating them, and preparing them to be processed by DITIS as trace events. The Trace Parser is used by the Workload Initializer and the Workload Replay to process the input trace. The *Workload Initializer* is responsible for creating an initial state for the storage system before the trace is executed. For example, it can create files that are read by the trace but not created by the trace, place files in specific layers, populate caches, or execute any other action required. The *Workload Replay* is a policy that dictates the order and timing for replaying the trace of file requests based on the submission timestamps. The Workload Replay creates a new Application for each distinct process id it encounters and a message for every file request. This message is then sent to its corresponding Application for processing. Each *Application* represents a different client application outside the storage system that submits file requests to the storage system.

Access Layer: The Access Layer defines the interface of the storage system for client Applications. It holds a set of *Access Modules*, and an *Application Connector*, which is responsible for balancing incoming Application connections to the available Access Modules. An Access Module represents either the storage system's client running on the Application node or an access component of the storage system running on a storage node. The Access Module receives and processes file requests from Applications, and has three main components: the *Dataflow Manager*, which determines when and how to process or forward a file request based on the *Dataflow Policies*; the *Metadata Manager Cache* that manages the information about the files stored in the system and accessed by this Access module; and the *Cache Manager* that manages the data stored in the *L0 DRAM Data Cache* using the *L0 Cache Policies*, which are responsible for admitting, evicting, or prefetching data to/from the L0 Data Cache.

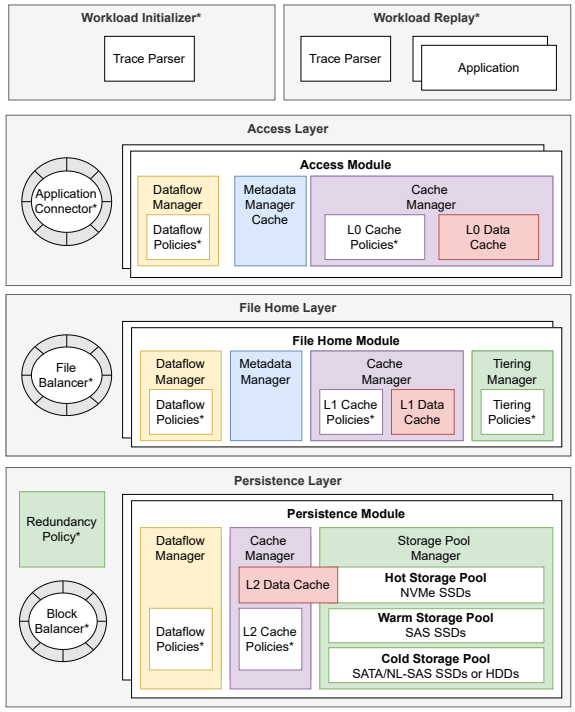

Fig. 1: The architecture of DITIS. Components marked with a * are pluggable policies.

File Home Layer: The File Home Layer holds the *File Balancer* and a list of *File Home Modules*, one for each storage node of the system. The File Balancer distributes files across the File Home Modules based on full file paths, and the File Home Modules maintain the file system namespace and process the file requests forwarded by the Access Layer. The File Home Module consists of the *Dataflow Manager* and its *Dataflow Policies*; the *Metadata Manager* that manages the metadata information about the files stored in the system; the *Cache Manager* that hosts the *L1 NVRAM Data Cache* and the *L1 Data Policies*; and the *Tiering Manager* that manages the *Tiering Policies*, which decide when to place, migrate, and delete files from the storage tiers in the Persistence Layer.

Persistence Layer: The Persistence Layer models the underlying storage capabilities of the storage system. It holds the *Redundancy Policy* that determines how to create and distribute redundancy blocks (e.g., using Erasure Coding, replication, or per-node RAID); the *Block Balancer* that distributes blocks across the Persistence Modules; and the *Persistence Modules* (one per node) that process block requests forwarded by the Access and File Home Layers as well as store data blocks on the different storage pools that form the storage tiers. The Persistence Module has three main components: the *Dataflow Manager* with its *Dataflow Policies*; the *Cache Manager* that

```
// initialize storage
storage.initialize()
// Process events scheduled in the queue
lastTraceTime ← -1;
while lastTraceTime != INF or queue.hasPendingMessages() do
  if !queue.hasPendingMessages() or lastTraceTime <
  ↪  queue.getNextMessageTime() then
    // Get a new trace item and generate a new message
    lastTraceTime ← workloadReplay.processNextTraceItem()
  else
    // Process the next available message
    queue.processNextMessage()
  end
end
```

Algorithm 1: Main simulation control loop.

hosts the *L2 Data Cache* in the hot tier and manages the *L2 Data Policies*; and the *Storage Pool Manager*, which manages the storage pools consisting for storage medias (e.g., HDDs, SSDs) organized in up to three tiers (Hot, Warm, and/or Cold).

C. Simulation Model

DITIS employs a modified version of the *Actor Model* for simulating a distributed data storage system. In DITIS, the Workload Replay, the Applications, the Access Modules, the File Home Modules, and the Persistence Modules are modeled as *actors*. Actors are only responsible for maintaining their own private state, making local decisions, and exchanging messages to communicate with each other. In the original Actor Model, every actor can concurrently send messages to other actors, create new actors, and react on a message basis asynchronously. There is no ordered sequence that needs to be followed, and these actions can be executed in parallel. In DITIS, however, instead of exchanging messages directly to each other, DITIS implements a *global simulation message queue*. This is a priority queue, where the timestamp of messages is the priority token. Thus, actors exchange messages by writing and reading to and from the simulation queue. The messages are then delivered based on the timestamp of each message to simulate the passing of time in an orderly fashion.

Another difference from the original actor model is that, in DITIS, actors are allowed to perform concurrent actions respecting the simulation time. DITIS implements a *Simulation Clock* that maintains the simulation time. The timestamps of exchanged messages are set based on the Simulation Clock and the duration time of the request processing. The requests hold the duration time, which accounts for the simulated time taken to process the request. Each time some processing is taking place, the processing is calculated based on some performance cost model and added to the duration time.

This enables a fine grained modeling of the various actions that take place during processing, such as exchanging data over the network, accessing a cache, accessing one or more disks in parallel, etc. The simulated duration at various points is added to the Simulation Clock to specify the time the next message needs to be delivered. The scheduling of messages from the simulation queue then respects the execution flow of the simulated storage system and accounts for all processing taken at different parts of the system.

Algorithm 1 presents the main simulation control loop of DITIS. First, the storage is initialized by the Workload Initializer policy. Then, if the simulation queue does not have any pending message to be processed, it will schedule a new trace event in the queue. Processing the next trace event means that the Workload Replay will create a new message based on the next trace event and queue it to be processed by its application. A new trace event is also scheduled if there is a gap between the last trace event and the next message in the queue to ensure that messages in the trace are scheduled correctly before other pending messages in the queue. Otherwise, if the system has messages in the queue, DITIS will process the next message in the queue, which also sets the current simulation time.

D. Simulation Output

During the simulation, DITIS will generate an *output trace*. This is a trace with the same sequence of requests given in the input trace but contain the simulated duration time instead. The output trace is written as the file requests are finished but following the correct order of submission. DITIS also generates a *report* containing a wealth of information regarding the simulated execution of the workload trace. In particular, it contains information and statistics about the input trace, for the storage initialization, for each storage layer, and for each application, including the number of bytes read and written by the file requests, the number of files opened, closed, written, and read, the number of requests processed, the cache hit ratio, the throughput, and much more.

III. Request Flow Management and Routing

During a simulation, the I/O requests originate from applications, traverse the appropriate layers of the storage system in order to be served, and then are returned to the applications. This section presents the request routing model and the key flow of requests in DITIS.

A. Three-level Request Routing

DITIS has three levels of routing used to distribute requests across storage layers and nodes due to semantic differences. The first level routes requests from Application to Access modules and is implemented by the *Application Connector*. The first time an Application is ready to submit a request to the file system, the Application Connector is invoked to connect a specific Access Module with the Application. The default policy connects an Application to an Access Module in a Round Robin fashion, simulating the presence of a basic load balancer at the top of the file system. DITIS also supports a Client Mode policy that connects each Application to its own private Access Module, which runs on the same (compute) node as the Application. This enables DITIS to simulate a scenario where each application has its own local data cache and showcases another aspect of DITIS's flexibility.

The second level routes requests to the appropriate File Home Modules based on file semantics and is implemented by the *File Balancer*. When an Access Module submits a file

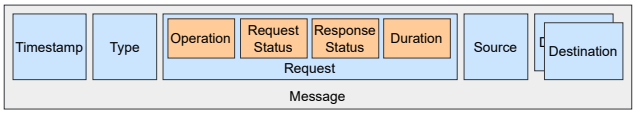

Fig. 2: Simulator message attributes.

request to the File Home Layer, the File Balancer is used to find the File Home Module hosting the required file so that the request is correctly routed there. The current policy uses hashing based on file path to determine the appropriate File Home Module but more complex approaches such as consistent hashing or distributed hash table are easily supported.

The last level routes requests to the appropriate Persistence Modules based on block semantics and is implemented by the *Block Balancer*. When an Access or File Home Module sends a block request to a Persistence Module, the Block Balancer is used for determining the Persistence Module that is responsible for managing that particular block. The default policy also employs hashing based on the block id but more advanced routing strategies are also easy to support.

B. General Request Flow

During a simulation, the trace events in the input trace are converted into I/O *requests*. Currently, a request can be a *file*, a *standard stream* (e.g., stdout), or a special *device* (e.g., to CD-ROM) request. Only file requests are sent and served by the simulated storage system. Every request holds the *type of operation* (open, close, read, write, or delete), the *request status* (pending, in-progress, or completed), the *response status* after request completion (success or fail); and the *duration time* taken to process the request. Requests are encapsulated in *messages*, in order to be sent from one component to another based on the actor model. Figure 2 depicts the message attributes, including the request. A message consists of a *timestamp*, referring to the simulated submission time; a *type* that distinguishes whether the message should be sent forward to the next destination or backward to the previous source; the *source* component that is sending the message; and the list of *destination* components that received the messages in order. The list of destination components forms the lineage of the request and is used for returning the message back through the components that initiated or forwarded the request.

Figure 3 depicts the general request flow starting from the Workload Replay, which creates and passes requests to the appropriate Applications (based on the request's PID in the input trace), and continuing through the components of the simulator. Access Modules receive messages from Applications containing file requests. Each Access Module extracts the request and forwards it to the Data Flow Manager. The Data Flow Manager processes the request according to its Data Flow Policies, which determine how the processing interacts with the Cache Manager and the Metadata Manager, and decide if the request can be completed or not. In the former case, a response message is send back to the Application. Otherwise, either file requests are sent to appropriate File Home Modules or block requests to appropriate Persistence

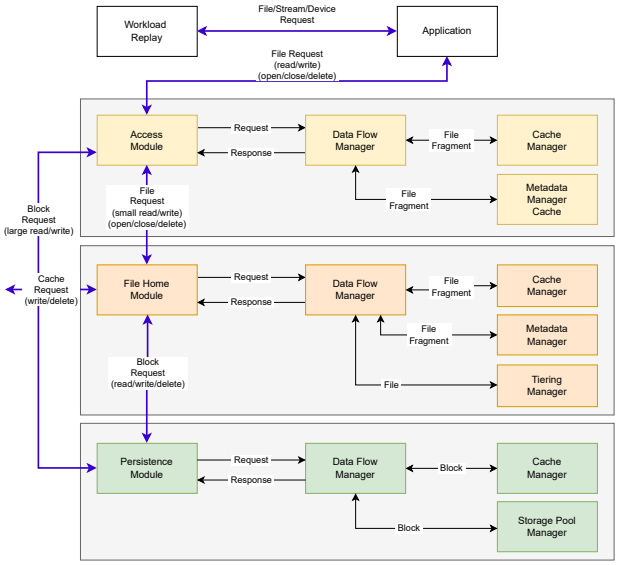

Fig. 3: Request Exchange.

Modules. The request flow in the File Home is analogous to the Access Module. The only additional component is the Tiering Manager, that invokes its Tiering Policies to decide how to place, migrate, or delete files among the storage tiers. Request processing in a File Home Module may result in cache mirroring requests to other File Home Modules or block requests to Persistence Modules. Finally, the Persistence Modules receive block requests from the Access and File Home Modules and processes them in a similar manner.

Messages exchanged among the modules can either be *synchronous* or *asynchronous*, depending on the simulated operation. Synchronous messages create chains of requests for which the successor requests need to be completed before the predecessor requests are completed. For example, if a file read request cannot be completed by the cache of the File Home Module, then synchronous block read requests are sent to appropriate Persistence Modules. Asynchronous messages contain requests that can execute independently from other requests. For example, when a fixed size of data is accumulated on a File Home cache, asynchronous block write requests are sent to Persistent modules for persisting that data.

C. Read/Write Request Flow

We present the key read and write operations simulated in the File Home Module by the default read and write policies. The data flow policies in the other modules are similar.

File Home Module read: The read policy receives a request containing a file name, an offset, and a data length. It checks with the Metadata Manager if the file exists and is open. If not, the request is returned with fail status. Then, the Cache Manager is invoked to check if the fragment is in the cache. If the cache contains the entire fragment, the fragment is read from the cache. The Data Flow Manager adds the read time to the request duration, marks the request as completed, and returns it to the source Access module. If the cache contains

only some parts of the file fragments, those parts are read and the missing file fragments are computed. If the cache does not contain the fragment, then the entire fragment is considered missing. The read policy asks the Metadata Manager for the location of the blocks storing the missing fragments. Then, it creates a child block request for each block, and sends them to the Persistence layer in a synchronous manner. When the child requests are received back, their fragments are offered to the cache. The Cache Manager decides whether to cache the fragments or not based on an admission policy. After receiving all child requests, the Data Flow Manager adds the time of the slowest request to the read duration time of the original request (since child requests were executed in parallel and any potential interactions are accounted for in the lower layers), sets it as completed, and returns to the appropriate Access module.

File Home Module write: The received write request contains a file name, an offset, and a data length. First, it checks with the Metadata Manager whether the file exists and is open. If not, the request is returned with fail status. Note that a new file is created and opened during an open request. The fragment is written directly to the File Home Cache and then mirrored to other File Home Modules (to other storage nodes) according to a data mirroring policy. The times to write to the cache and to mirror to other nodes are both accounted for in the duration of the request. If the insertion of new file fragments into the cache caused other fragments to be evicted, and if those fragments were dirty (i.e., they are not stored in the Persistence layer), then the dirty evicted fragments are sent to the Persistence layer as block requests in a synchronous manner. New file fragments that are added into the cache are aggregated into larger data blocks before they are flushed to the Persistence layer. If enough data has been aggregated for flushing, then that data is sent to the Persistence layer. These operations are managed by a flushing policy that decides when and which data to flush. For example, if erasure coding is used, the flushing policy will wait until a full stripe of data is formed before flushing it. The original request is considered completed when all synchronous child requests return from the Persistence layer (if any). The Data Flow Manager adds to the original duration the write time of the slowest request and returns to the originating Access module.

IV. PERFORMANCE COST MODELING

This section presents the current cost models used to simulate the performance of storage media types and network. The cost models are pluggable, and hence, can be easily replaced.

A. HDD Modeling

A Hard Disk Drive (HDD) is a non-volatile data storage device that consists of an *arm*, a *platter*, a *spindle*, a *read and write head*, and an *interface*. In order to serve I/O requests, in a simplified description, the *arm* moves to find the right track and the *spindle* spins the *platter* to set it to the right sector. Then, the *read and write head* transfers data to or from the *platter*. The data is received from or sent to the interface, and the I/O request is completed [2].

Modeling the performance of HDD devices require accounting for the delays of every internal action. First, moving the arm accounts for the *seek* time. Manufacturers generally report the average seek time $\bar{t}_{seek} = s$ as a constant. Yet, if this parameter is not given, estimations present that this value is approximately one-third of the full seek time [2]. In short, $\bar{t}_{seek} = n^3/3$, where n is the number of tracks.

Rotating the platters to position the head to the right sector accounts for the *rotation* time. The average rotational time $\bar{t}_{rotation}$ is derived directly from the disk rotation speed, which is given by manufacturers as Rotations Per Minute (RPM). Thus, the rotation time for a disk with r RPM is given by:

$$\bar{t}_{rotation} = \frac{60}{r} \cdot \frac{1}{2} \cdot 10^6 \tag{1}$$

where $60/r$ is the time (in seconds) to execute one single rotation, the $1/2$ is included because, on average, a request will require a half rotation [2], and we multiply by 10^6 to convert to microseconds. The data being accessed might be contiguous to the previous data, and consequently, the seek and rotation times would be lower. Our model can differentiate random from sequential operations.

Finally, transferring data accounts for the *transfer* time. The average transfer time $\bar{t}_{transfer}$ depends on the amount of data transferred over the peak transfer rate. In particular, $\bar{t}_{transfer}$ in microseconds is calculated as:

$$\bar{t}_{transfer} = \frac{\lceil \frac{k}{p} \rceil \cdot p}{m} \cdot 10^6 \tag{2}$$

where, m is the maximum transfer rate of the disk in MB/sec, p is the minimum amount of data in a single transfer (and equals the disk page size), and k is the amount of data requested.

Thus, the total duration time for a single **random request** is the sum of the seek, rotation, and transfer times. For sequential I/O requests, there will be no seek and rotation costs.

Disks also maintain a queue of outstanding requests that need to wait for some time while the disk is serving other requests. If the disk is idle when the request arrives, the wait time is zero. Otherwise, the wait time equals the time left for processing the currently active request plus the duration times of all outstanding requests in the disk queue. DITIS computes the wait time by (i) maintaining the virtual completion time of the last request that arrived at the disk, and (ii) subtracting the virtual completion time from the current virtual time if the current request arrives before the last completion time.

B. SSD Modeling

A solid-state drive (SDD) device, in a simplified manner, consists of an *I/O controller*, a *flash array*, a *data register*, and a *cache register*. The I/O controller receives requests for reading or writing data, which is stored in the flash array. The data register acts as data buffer for the flash array, and the cache register acts as a buffer for the I/O requests. A read request involves decoding the I/O request, reading data from the flash array to the data register, then transferring data from the data register to the I/O bus. When reading multiple pages, it will first transfer data to the data register, then to the cache register, which will finally transfer the data to the I/O bus [3].

Consider t_{rr} as the total duration time for reading a single random page. Then:

$$t_{rr} = t_{cmd} + t_{read} + t_{trans} \tag{3}$$

where t_{cmd} is the time to decode the I/O request, t_{read} is the time to read a page from the flash array, and t_{trans} is the time to transfer a page from the data register or cache register to the I/O bus. For sequential read requests, there will be only one single I/O request, but multiple transfers from the data register to the data cache and I/O bus. The following equation generalizes the average time to read n bytes of data:

$$\bar{t}_n = \bar{t}_{acc} + \frac{\lceil \frac{n}{p_{size}} \rceil \cdot p_{size}}{m} \tag{4}$$

where t_{acc} is the average access time (accounting for decoding and reading the first page), p_{size} is the page size, and m is the maximum data transfer rate.

Similar to HDDs, SSDs also maintain a queue of outstanding requests. The simulator computes the wait time of each arriving request in the same manner as for HDDs.

C. DRAM/NVRAM Modeling

A memory address consists of a *bank*, a *row*, and a *column*. Multiple DRAM commands are required to access a particular location. The duration of internal steps required to serve a request is counted in clock cycles. Accessing a specific location requires that an entire row from a specific bank to be *activated*. After the activation, any location within the row can be read or written [4], [5]. DRAM manufacturers provide the number of clock cycles required to perform these internal actions. The constant t_{cl} is the number of clock cycles between receiving a request and having the data ready; t_{rp} is a minimum number of cycles to open a new row; and t_{ras} is the amount of cycles that a row must be open to write data.

Considering f to be the DRAM frequency, n to be the number of bytes to be accessed, r the size of a row in bytes, m_r the maximum read transfer rate, and t_r the time to execute a read request, then:

$$t_r(n) = \frac{\lceil \frac{n}{r} \rceil \cdot r}{m_r}, \text{ where } m_r = \frac{f \cdot r}{t_{cl}} \tag{5}$$

Similarly, the time to execute a write request t_w for n bytes can also be expressed using:

$$t_w(n) = \frac{\lceil \frac{n}{r} \rceil \cdot r}{m_w}, \text{ where } m_w = \frac{f \cdot r}{t_{cl} + t_{rp} + t_{ras}} \tag{6}$$

D. Network Modeling

A model developed by Mathis et al. [6] predicts network bandwidth for a sustained TCP connection subjected to moderate packet losses, including losses caused by network congestion. According to this model, the maximum network bandwidth bw is measure by:

$$bw = \frac{MSS}{RTT} \cdot \frac{C}{\sqrt{p}} \tag{7}$$

where MSS is the Maximum Segment Size, i.e., the amount of data in bytes that a computer can receive in a single TCP segment; RTT is the Round Trip Time, i.e., the time a packet takes to go to a destination and return; C is a constant of proportionality; and p is a random packet loss at constant probability. The bw value represents the maximum throughput in a channel. Thus, the time to transfer n bytes end-to-end is:

$$t_n = \frac{n}{bw} \tag{8}$$

When multiple clients are actively using the network concurrently, the network bandwidth is split evenly between the clients. The simulator keeps track of the active network connections a and adjusts the model to compute the transfer of n bytes accordingly:

$$t_n = \frac{n \cdot a}{bw} \tag{9}$$

V. EXPERIMENTAL EVALUATION

In this section, we present the evaluation of DITIS. All experiments were executed on a machine with i7-7500U CPU @ 2.70GHz, 12 GB of RAM, and OpenJDK 17.0.3. The simulator was developed with Java v17, and Maven v3.8 is used for automating the process of building the project. At the moment, the simulator consists of 25 packages, 12 enumeration types, 31 interfaces, 9 abstract classes, and 140 classes, totaling over 10700 lines of code.

We evaluate DITIS by simulating eight traces derived from production workloads provided by Huawei Technologies Inc. To evaluate the simulation accuracy, we compare the real and simulated times of each request present in the traces, and report the *Mean Absolute Percentage Error* (*MAPE*). The details of each trace are presented in Table I, including the number of file requests per trace, the ratio of read and write operations, the number of random and sequential (read/write) operations, as well as the corresponding MAPE. As it can be seen, the traces exhibit a wide-spectrum of behavior ranging from read-heavy to write-heavy, with different mixes of sequential versus random read/write requests. Note that we consider sequential requests those that operate over consecutive file fragments through a sequential time frame. In our evaluation, DITIS simulates each input trace with a different storage configuration (which we cannot reveal due to privacy considerations), as each trace was originally executed with a different setup. Since open and close requests are straightforward operations, we focus our evaluation on read and write requests.

DITIS is able to accurately simulate write and read requests in most cases. The MAPE metric ranges from 0.06 up to 0.26 in 7 out of 8 traces for write requests, with an outlier of 0.96 for trace 5. Note that up to 70% of trace 5 amounts to read requests, which were simulated with a MAPE of 0.27. Read request simulations exhibited a slightly worse MAPE ranging from 0.07 to 0.93. Figure 5 presents the real and simulated execution times for the write requests of trace 4. DITIS is able to correctly follow the execution trends of the real workload along with all the spikes, albeit with lower magnitude for

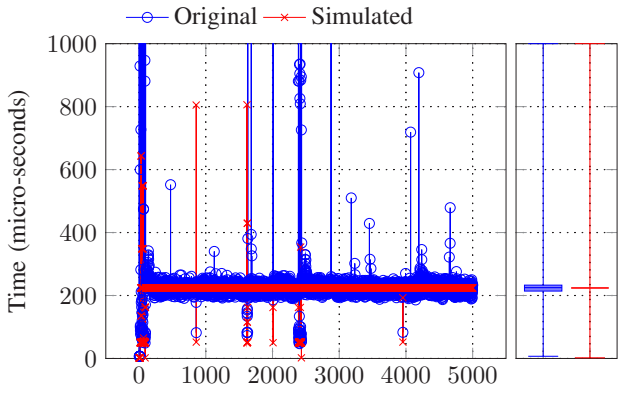

Fig. 4: Real and simulated execution times (raw values and distributions) for read requests for trace 2.

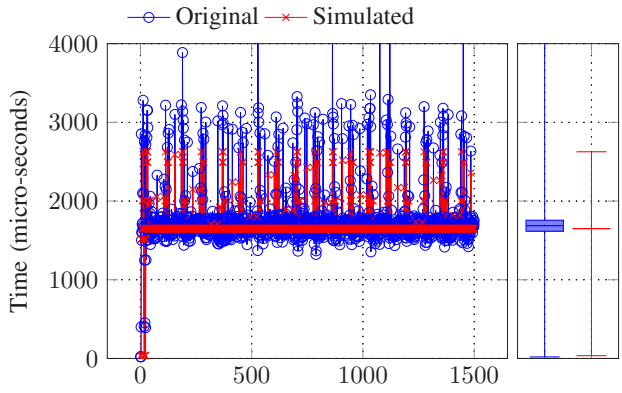

Fig. 5: Real and simulated execution times (raw values and distributions) for write requests for trace 4.

TABLE I
REQUEST DISTRIBUTION AND SIMULATION ERROR PER TRACE.

		Ratio (%)		Sequential		Random		MAPE	
	Requests	Read	Write	Read	Write	Read	Write	Read	Write
1	3106	100.0	0.0	3022	0	84	0	0.34	-
2	5528	99.7	0.3	5342	9	171	6	0.07	0.31
3	49883	0.2	99.8	101	49770	12	0	0.75	0.21
4	16118	89.6	10.4	3388	1544	11047	139	0.70	0.06
5	71496	68.3	31.7	48767	22630	92	7	0.27	0.96
6	47823	85.4	14.6	10562	0	30263	6998	0.92	0.13
7	54473	96.1	3.9	8633	1573	43713	554	0.92	0.13
8	1483669	99.3	0.7	1285	10053	1472181	150	0.93	0.26

TABLE II
DITIS RUN TIMES (IN SECONDS) AND INTERESTING STATISTICS DURING INITIALIZATION AND SIMULATION.

	Initialization			Simulation				
	Files Created	Bytes Written	Run Time	Write Requests	Bytes Written	Read Requests	Bytes Read	Run Time
1	56	35.2M	0.14	0	0.0	3106	11.99M	0.13
2	21	14.9G	1.50	15	109.9M	5513	6.7G	0.36
3	12	809.2K	0.16	49770	376.1M	113	817.2K	23.22
4	3136	526.7K	4.98	1683	7.0G	14435	3.8G	1.11
5	10	707.2M	0.37	22637	707.4M	48859	708.3M	55.30
6	3299	635.7M	5.38	6998	37.8G	40825	3.1G	3.87
7	3138	51.7M	5.00	2127	8.7G	52346	2.6G	1.92
8	857778	6.6G	13.84	10203	465.7K	1473466	11.3G	18.61

the bigger spikes. Similarly, Figure 4 presents the real and simulated times for the read requests for trace 2. While DITIS is able to correctly simulate most of the trace, there are a few outliers present in the trace that are missed by DITIS. These differences (observed mainly for read requests) are due to the different policies that move file fragments through the data storage with different approaches (e.g., policies related to cache, tiers, data placement during initialization), or delays that are not yet modeled by DITIS such as different levels of network contention within the distributed storage. For example, some read requests in DITIS were served from a cache, whereas they were probably served by the persistence storage in the real system (based on their duration). It is a complex task to precisely simulate and replay the various data management and caching decisions in the presence of several policies that works together and influences each other. Yet, DITIS is able to follow the overall trend of the real execution times as well as accurately match the average execution times.

Next, we evaluate the efficiency of DITIS during both the initialization phase and the trace simulation. The corresponding run times are shown in Table II along with statistics that explain DITIS' run times. The Workload Initializer (recall Section II-B) is responsible for creating an initial state for the storage system, such as creating files that existed prior to the beginning of the trace. The time needed for initialization is proportional to (i) the number of files created since DITIS maintains metadata for each file, and (ii) the file size since DITIS distributes file data into blocks that are stored across the storage media, and maintains metadata for each block, much like like a real storage system does. Even when hundreds of thousand of files are created and GBs of data are written, this part of the simulation executes within a few seconds.

Simulating a trace with DITIS is also very efficient and completes within seconds as shown in Table II. The simulation time depends heavily on both the number of write requests and bytes written in the trace for the same reasons explained above. The simulation time is also proportional to the number of read requests but is not affected much by the the number of bytes read. Finally, simulation time can also be affected by other, non-obvious factors, such as the order of requests (as it can impact cache policies), the size of requests (as it can impact data flow policies), as well as configuration parameters (such as the number of disks or disk block size). Nonetheless, DITIS is able to simulate large traces both efficiently and accurately.

VI. RELATED WORK

There are several efforts to simulate multi-tiered data storage systems. MDCSim [7] is a multi-tier data center simulation framework that supports a three-tier architecture, whereas OGSSim [8] enables users to explore the design space of storage systems by supporting various combinations of tiers and volumes. StorageSim [9] enable users to define up to three storage tiers with their performance profiles, while it provides pluggable data placement policies to analyze their impact in the storage's performance. All aforementioned simulators focus on single-node storage systems. EEffSim [10] supports pluggable data placement policies and aims to study the impact of data placement on energy efficiency for distributed (but single-tier) storage systems. Both PFSsim [11] and HPIS3 [12]

focus on simulating Parallel File Systems in High Performance Computing (HPC) centers, but HPIS3 also supports HDD/SSD hybrid systems. NCAR MSS [13] simulates storage drives and software components to explore the design space for cache on data storage systems. SANgo [14] employs reinforcement learning to explore the stability of data storage systems by adjusting the modeled hardware and introducing failures.

In contrast to the related work, DITIS is extremely versatile and extensible. DITIS implements a series of policies that govern all decisions related to cache, tiers, request processing flow, data redundancy, load balance, as well as other options and configurations, like storage device arrangement, number of nodes, threshold values, and enabling/disabling tiers.

VII. CONCLUSION

DITIS is a comprehensive storage simulator that is able to simulate the execution of file system requests on a distributed storage system with multiple levels of tiers and caches. Each tier and cache can be configured with different types of storage media devices, each with their own performance characteristics. The simulator will utilize the provided characteristics in fine-grained performance cost models (which are distinct for each device type) in order to compute the duration time of each request processed on each tier. At the same time, DITIS will accurately simulate the overall flow of requests through the different layers and storage nodes of the system using numerous pluggable policies that control every aspect of execution, ranging from request routing and data redundancy to cache and tiering strategies. The ability to simulate the execution of a workload in such an accurate and realistic way brings multiple benefits for its users, since DITIS can be used to better understand the behavior of the underlying file system as well as evaluate different storage setups and policies.

REFERENCES

[1] G. A. Agha, I. A. Mason, S. F. Smith, and C. L. Talcott, "A Foundation for Actor Computation," *J. Funct. Program.*, vol. 7, no. 1, p. 1–72, jan 1997. [Online]. Available: DOI: 10.1017/S095679689700261X

[2] R. H. Arpaci-Dusseau and A. C. Arpaci-Dusseau, *Operating Systems: Three Easy Pieces*. North Charleston, SC, USA: CreateSpace Independent Publishing Platform, 2018.

[3] K. El Maghraoui, G. Kandiraju, J. Jann, and P. Pattnaik, "Modeling and Simulating Flash based Solid-state Disks for Operating Systems," in *WOSP/SIPEW '10*. ACM Press, 2010, p. 15. [Online]. Available: DOI: 10.1145/1712605.1712611

[4] J. H. Ahn, M. Erez, and W. J. Dally, "The Design Space of Data-parallel Memory Systems," in *Proc. of the 2006 ACM/IEEE Conference on Supercomputing (SC)*. ACM Press, 2006, p. 80. [Online]. Available: DOI: 10.1145/1188455.1188540

[5] A. Hansson, N. Agarwal, A. Kolli, T. Wenisch, and A. N. Udipi, "Simulating DRAM Controllers for Future System Architecture Exploration," *IEEE ISPASS*, pp. 201–210, 2014. [Online]. Available: DOI: 10.1109/ISPASS.2014.6844484

[6] M. Mathis, J. Semke, J. Mahdavi, and T. Ott, "The Macroscopic Behavior of the TCP Congestion Avoidance Algorithm," *Computer Communication Review*, vol. 27, no. 3, pp. 67–82, 1997. [Online]. Available: DOI: 10.1145/263932.264023

[7] S.-H. Lim, B. Sharma, G. Nam, E. K. Kim, and C. R. Das, "MDCSim: A Multi-tier Data Center Simulation Platform," in *IEEE Intl. Conf. on Cluster Computing and Workshops. IEEE*, 2009, pp. 1–9. [Online]. Available: DOI: 10.1109/CLUSTR.2009.5289159

[8] S. Gougeaud, S. Zertal, J. C. Lafoucriere, and P. Deniel, "A Generic and Open Simulation Tool for Large Multi-Tiered Hierarchical Storage Systems," *Simulation Series*, vol. 48, no. 8, pp. 91–98, 2016. [Online]. Available: DOI: 10.1109/SPECTS.2016.7570515

[9] C. San-Lucas and C. L. Abad, "Towards a Fast Multi-tier Storage System Simulator," *IEEE ETCM*, pp. 1–5, 2016. [Online]. Available: DOI: 10.1109/ETCM.2016.7750836

[10] R. Prabhakar, E. Kruus, G. Lu, and C. Ungureanu, "EEffSim: A Discrete Event Simulator for Energy Efficiency in Large-scale Storage Systems," *IEEE Intl. Conf. on Energy Aware Computing (ICEAC)*, 2011. [Online]. Available: DOI: 10.1109/ICEAC.2011.6136682

[11] Y. Liu, R. Figueiredo, Y. Xu, and M. Zhao, "On the Design and Implementation of a Simulator for Parallel File System Research," *IEEE Symposium on MSST*, pp. 0–4, 2013. [Online]. Available: DOI: 10.1109/MSST.2013.6558438

[12] B. Feng, N. Liu, S. He, and X. H. Sun, "HPIS3: Towards a High-performance Simulator for Hybrid Parallel I/O and Storage Systems," *Proc. of the 9th Parallel Data Storage Workshop*, pp. 37–42, 2014. [Online]. Available: DOI: 10.1109/PDSW.2014.12

[13] B. Anderson, "Mass Storage System Performance Prediction using a Trace-driven Simulator," *IEEE Symposium on MSST*, pp. 297–306, 2005. [Online]. Available: DOI: 10.1109/MSST.2005.19

[14] K. Arzymatov, A. Sapronov, V. Belavin, L. Gremyachikh, M. Karpov et al., "SANgo: A Storage Infrastructure Simulator with Reinforcement Learning Support," *PeerJ Computer Science*, vol. 2020, no. 5, pp. 1–16, 2020. [Online]. Available: DOI: 10.7717/peerj-cs.271

Edson Ramiro Lucas Filho is a post-doctoral researcher at the Data Intensive Computing Research Lab, Cyprus Univ. of Technology. He received his Ph.D. from the Federal University of Paraná, Brazil, in June 2020. He held positions as a post-doctoral researcher at the Scalable Database Systems group, Univ. of Passau, Germany, and as a Software Engineer R&D at the Interdisciplinary Centre for Security, Reliability and Trust, Univ. of Luxembourg.

Lambros Odysseos acquired his M.Sc. in Data Science and Engineering (2019) and his B.Sc. in Computer Engineering and Informatics (2017) from Cyprus Univ. of Technology (CUT) both with first student in class award. He worked as a research associate at CUT for 3 years and his research interests include data analytics and visualizations, smart data processing, Internet of Things, and machine learning. Currently, he works as an IT officer at CUT.

Yang Lun is an algorithm engineer in Huawei Data Storage Product Line. He received his Ph.D. in Mathematics and completed his undergraduate studies in Electrical Engineering from Beihang University in 2020 and 2014, respectively. He was a visiting student scholar in Energy Resource Engineering at Stanford University from 2018-2019. His research interests are in storage system algorithms.

Fu Kebo is an algorithm engineer in Huawei Data Storage Product Line. His research interests include data placement and intelligent storage algorithms.

Herodotos Herodotou is an Assistant Professor at the Cyprus Univ. of Technology leading the Data Intensive Computing Research Lab. He received his Ph.D. in Computer Science from Duke University in May 2012. His Ph.D. work received the ACM SIGMOD Jim Gray Doctoral Dissertation Award Honorable Mention. Prior, he held research positions at Microsoft Research, Yahoo! Labs, and Aster Data. His research interests are in large-scale data processing, storage, and database systems.