PFSsim Tutorial

Yonggang Liu

yonggang@acis.ufl.edu

(Based on the Code Version 0.20.0)

Introduction

PFSsim is a modular simulator for simulations on Parallel File Systems (PFS), to provide a platform for storage system developers and administrators to test and evaluate their ideas about new designs in systems and I/O management schemes.

Environment Setup

1. Please refer to the website http://www.omnetpp.org/ for the installation and setups of OMNeT++.
2. Python should be installed.

Download and Run PFSsim

1. Download the latest version of PFSsim from https://github.com/myidpt/PFSsim and extract it to your destination directory.
2. To generate the sample input and configuration files (16 clients, 2 traces per client, each generating read I/O to a 512MB file, each I/O request is 1MB, and the data are evenly striped onto 8 data servers with the stripe size of 64KB), run (option --help for help information):

   $ ./gen_input.sh
3. To generate the binary, run (option --help for help information):

   $ ./gen_exe.sh
4. The binary and input should be generated if you have done above procedures without error. Then you can run the binary (by default it is PFSsim):

   $ ./PFSsim

Develop PFSsim in OMNeT++ SDE

1. Create a new OMNeT++ project by doing:

   Start OMNeT++ from command line.

   Select "File → New → project...", extend the "OMNeT++" folder and select "OMNeT++ Project…". Click "Next".

   Give the project a name. Click "Next". Select "Empty project" and click "Finish".
2. Import the PFSsim code to the project by doing:

   Right-click the project name in the navigation panel, and select "Import...".

Extend the “General” folder and select “File System”, click “Next”.

Browse to /github_path/PFSsim/omnetpp/, and click “OK”.

Check the “omnetpp” folder in the left window, and click “Finish”.

3. Build the OMNeT++ component by doing:

   Right click the project name in the navigation panel, and select “Build Project”.

4. To generate the sample input and configuration files (16 clients, 2 traces per client, each generating read I/O to a 512MB file, each I/O request is 1MB, and the data are evenly striped onto 8 data servers with the stripe size of 64KB), run (option --help for help information):

   $ ./gen_input.sh

5. To run the project, click the “Run” button on the panel.

Modules and Classes in OMNeT++

Folder: client/

Client Module

The *Client* module is defined in the file *Client.ned*. It simulates a client entity in the distributed file systems. The *Client* module has an inout gate that simulates its interface to the network. The initial messages are sent from this module.

Configuration Table

Name	Description
trc_path_prefix	The prefix of the trace file’s path. It is concatenated with an ID (refer to the *Input and Output Files::Trace File* subsection for the format) to form a full path.
rslt_path_prefix	The prefix of the result file’s path. It is concatenated with an ID (the same as the case for *trc_path_prefix*) to form a full path.
trc_proc_time	The simulated processing time of each trace. This time span is simulated after reading the trace from file, before the processing of the trace.
pkt_proc_time	The simulated processing time of each packet. This time span is simulated before the issuing of the packet.
pkt_size_limit	The maximum size of the data payload in the packets sent by the client.
small_io_size_threshold	This threshold is specific to some PFSs that have small IO requests. If the request size is smaller than this limit, it is a small IO request.
trcs_per_client	The number of trace files each client reads.

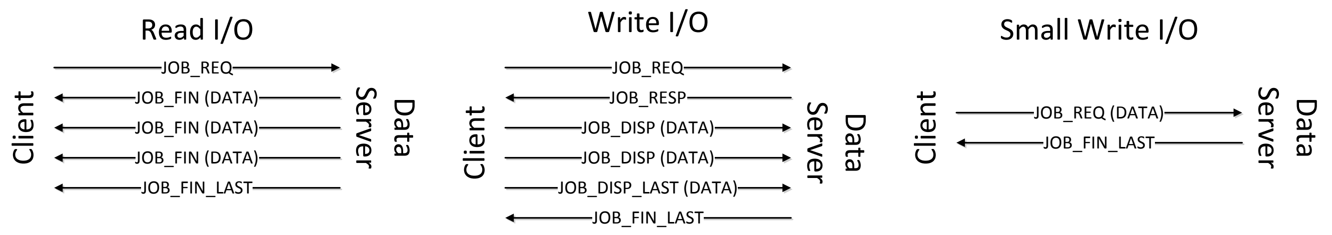

Figure 1. The packets triggered in the processing of different client I/O requests

Client Class

The *Client* class implements the *Client* module functionalities. The *Client* maintains a *layoutlist*, which is a list of the *Layout* objects; each *Layout* object stores the layout information for one file in the distributed file system.

The *Client* reads the traces from the trace files (It is possible that a *Client* reads multiple trace files), and creates one *Trace* object per trace in the trace file. For each *Trace* object, it first tries to find if the layout information is locally stored in the *layoutlist*. If not, a *qPacket* is created and sent to the metadata server (*Mserver*) to query the layout of the requested data. A new *Layout* object is inserted into *layoutlist* when the *qPacket* carries the layout information back. And then, a set of *gPackets* carrying data access information to each related data server are generated by the *Trace* and sent by the *Client* according to the data layout information. Every time the *Client* receives a reply from the server side, it feeds the *gPacket* to the *Trace* object, and the *Trace* checks if more data need to be accessed. Note that a trace may have multiple transmission rounds, each is one / a set of concurrent data requests to one / multiple data servers. When all the data I/O from the current trace are successfully done, the *Trace* object is destroyed by *Client*.

The *Client* reads the next trace when the previous one is done. For each trace, if it is synchronous (which means the start of this trace will be based on the finish of the previous one), the issue time of the next trace is the time provided in the trace plus the finish time of the previous time. Otherwise, the trace will be issued at the timestamp given in the trace file (but no earlier than the finish time of the previous request). Due to the protocols of parallel file systems and the limitations in packet size, a data request to a *Dserver* can consist of multiple packets. The I/O patterns for different I/O requests can be plotted in Fig.1.

Folder: trace/

Trace Class

The *Trace* class is created by the *Client* upon reading a new trace from the trace file, and destroyed by the *Client* when all the data I/O in that trace have finished. The *Trace* object contains a pointer pointing to a *Layout* object which has the layout of the target file. The data layout query process is omitted here as it is described in the *Client* class. According to the layout pattern, the requested data may be striped to multiple chunks, which are stored on a certain number of data servers. In each transmission round, the *Trace* issues all the data requests in a window (which is a range of data that can be concurrently accessed on multiple data servers). Once all the data requests from that window are finished, the *Trace* will move the window forward to issue new requests. In the default setup, each transmission round allows 1 I/O request to be sent to each involved data server.

Folder: layout/

Layout Class

Each *Client* and *Mserver* (class for metadata server) object maintains a list of *Layout* objects, each *Layout* object records the layout information of one file. The *Mserver* reads the layout information from the PFS file layout configuration file at the system start up. When a *qPacket* from *Client* comes, the *Mserver* will look up the file ID in the *Layout* list, copy the layout information to the *qPacket* and send it back. On the *Client*, when the *qPacket* reply comes back, the information is copied to the *Layout* list, which is used by the *Trace* objects. By default, the *Layout* class simulates a data layout as follows: the data are stored onto a list of designated data servers (can be a subset of all data servers) in a round-robin manner, and each data server is specified with a stripe size, meaning in each round, the data server stores the data of the stripe size.

Folder: mserver/

Mserver Module

The metadata server (*Mserver*) module is defined in the file *Mserver.ned*. An *Mserver* module simulates a metadata server in the parallel file system. The *Mserver* module has an inout gate that simulates its interface to the network. The *Mserver* Class implements the metadata server module.

Configuration Table

Name	Description
layout_input_path	The path of layout configuration file.
ms_proc_time	The simulated processing time of the *qPackets* on the metadata server.

Mserver Class

The *Mserver* class implements the metadata server module functionalities. The *Mserver* reads the data layout information from the configuration file at the system initialization step, and creates a *layoutlist* that each file is represented by a *Layout* object in the list. When the *Mserver* receives the *qPacket* querying the layout information, it looks up the corresponding file ID in the *layoutlist*, and returns the layout information by writing it to the *qPacket* and sending it back.

In default topology, only one *Metadata* server is simulated, but the users have the flexibility to extend that by introducing *Mserver* IDs and adding routing rules to the Router/Switch.

Folder: dserver/

Dserver Compound Module

The data server (*Dserver*) compound module is defined in the file *Dserver.ned*. A *Dserver* compound module simulates a data server (a.k.a. Object Storage Device) entity in the PFS. A *Dserver* compound module contains the following modules: data server daemon (*DSD*), virtual file system (*VFS*), disk cache (*DiskCache*), local file system (*LFS*) and physical disk (*Disk*). The *Dserver* has an inout gate that simulates the network interface; this inout gate is connected with the *DSD* module inside *Dserver*. Connections are also maintained among modules in the *Dserver* compound module. The connections are plotted in *Dserver.ned*.

Folder: dserver/dsd/

DSD Module

The *DSD* module is defined in the file *DSD_M.ned*. It simulates a PFS data server daemon component on the data server. In the default setup (for PFSs such as PVFS2 and Ceph) the *DSD* module has two inout gates, one is connected with the network interface on the *Dserver* compound module, and the other one is connected with the *VFS* module. In an alternative setup (for PFSs such as Lustre and PanFS), the *DSD* module has three inout gates, connecting the network interface, the disk cache and the disk system (users need to tune the system for this support). The *DSD_M* Class implements the *DSD* module.

Configuration Table

Name	Description
write_data_proc_time	The simulated processing time of a data packet (write) on the data server daemon.
write_metadata_proc_time	The simulated processing time of a write request (this is a metadata packet).
read_metadata_proc_time	The simulated processing time of a read request (this is a metadata packet).
small_io_size_threshold	This threshold is specific to some PFSs that have small IO requests. If the request size is smaller than this limit, it is a small IO request.
object_size	The size of the object inside the data server. This is the maximum unit of a data access locally on the data server.
max_subreq_size	The maximum size of the sub-requests.
degree	The degree of the data server daemon.
pfsname	The name of the PFS. Currently we only support PVFS2.
parallel_job_proc_time	The simulated processing time of a finished request on the data server daemon.
O_DIRECT	The flag for opened files. If set, the I/O will not go through *DiskCache*.

DSD_M Class

The *DSD_M* class is the core class for the data server daemon module. It provides a plugin board for the PFS-specific data server daemon functionalities. Those data server daemon functionalities are implemented by classes inherited from the base class *IDSD* (e.g., *PVFS2DSD*). As seen in Fig. 1, *DSD_M* processes different packets in different ways. If the read request (*JOB_REQ*) comes, *DSD* retrieves the data from the lower levels, and reply with the data packets (*JOB_FIN*). If a regular write request comes, a write response (*JOB_RESP*) is sent back to the *Client* to notify if the I/O can be started. If a write I/O data request (*JOB_DISP*) comes, the data are sent to the lower levels. When all the data are written, *DSD* replies a *JOB_FIN_LAST* packet. If a small-write (in some specific PFSs small I/Os are treated differently) request comes, the data are already in the request. So the data are directly written to the lower levels.

IDSD Class

The *IDSD* class is the base class for the classes implementing the data server daemon functionalities. It defines the interfaces for the *DSD_M* class to interact with the classes implementing PFS-specific functionalities.

PVFS2DSD Class

The *PVFS2DSD* class realizes the functionalities of the PVFS2 data server daemon. PFS-specific I/O patterns may be realized by this class. It may divide the data requests into smaller I/O requests according to the object size. Also, it may achieve concurrency when forwarding data I/O by starting to send part of the requested data before the entire data are received.

Folder: dserver/vfs/

VFS Module

The *VFS* module is defined in the file *VFS.ned*. It simulates a virtual file system. The *VFS* module has three inout gates, one is connected with the *DSD* module, and the other two are connected with the *DiskCache* module and the *LFS* module.

Configuration Table

Name	Description
page_size	The page size of the disk cache.
degree	The degree of the virtual file system.

VFS Class

The *VFS* class implements the *VFS* module. It realizes functionalities of Linux kernel v2.6. It maps between the I/O requests in file offsets and the page indexes. If the *O_DIRECT* flag is set in the request, the request is sent to the *LFS* directly, otherwise, it is sent to the *DiskCache*. Specifically, when the write I/O start/end bit is not aligned with any page's edge (some page is only partially updated), the partially-updated pages are read first, and then updated.

Folder: dserver/diskcache/

DiskCache Module

The *DiskCache* module is defined in the file *DiskCache.ned*. It simulates a disk cache system. The *DiskCache* module has two inout gates, one is connected with the *VFS* module, and the other one is connected with the *LFS* module.

Configuration Table

Name	Description
total_pages	The total number of pages in the cache.
usable_pages	The total number of available pages in the cache.
cache_r_speed	The simulated time span of reading one page in disk cache.
cache_w_speed	The simulated time span of writing one page in disk cache.
dirty_ratio	The *dirty_ratio* threshold of disk cache. The data server daemon I/O will be blocked if it is exceeded.
dirty_background_ratio	The *dirty_background_ratio* threshold of disk cache. The disk cache will start to write out dirty pages if it is exceeded.
dirty_expire_centisecs	The *dirty_expire_centisecs* threshold of disk cache pages. The dirty page will expire when it is exceeded for the page.
writeout_batch	The maximum number of pages to be written out every time when the system forces pages to be written out.
diskread_batch	The maximum number of pages to be read from the disk every time.
disable_ra	This flag can be 1 or 0. 1 means the read ahead functionality is disabled. Otherwise, it is enabled.

DiskCache Class

The *DiskCache* class implements the functionalities of the *DiskCache* module. The page table structure is realized by a doubly-linked list - we call it page table list, with each node storing a chunk of adjacent pages that have identical page flag settings (e.g., *PG_dirty*). For page reclaiming, each node in the page table list is also in a page reclaiming list. The LRU algorithm is implemented to manage the list. Fundamental page pre-fetch and page write back functionalities are implemented based on the Linux kernel 2.6. Similar to a real world system, users are able to tune the write back thresholds by setting up system parameters such as *dirty_ratio*, *dirty_background_ratio* and *dirty_expire_centisecs*.

Folder: dserver/lfs/

LFS Module

The *LFS* module is defined in the file *LFS_M.ned*. It simulates a local disk file system (LFS) component on the data server (if the local disk file system exists in the PFS). The *LFS* module has three inout gates, one is connected with the *DiskCache* module, one is connected with the *VFS* module, and the other one is connected with the *Disk* module. The *LFS_M* class implements the *LFS* module.

Configuration Table

Name	Description
page_size	The page size of the disk cache.
blk_size	The block size of the local disk system.
new_ext_size	The default size of the newly created extent in the local disk system.
new_ext_gap	The default size of the gaps between the locations of the newly created extents in the local disk system.
degree	The number of concurrent requests can be dispatched to the local disk system.
fsname	The name of the local disk file system.
disk_size	The size of the local disk
ext_in_path_prefix	The prefix of the path for the input extent information.
ext_out_path_prefix	The prefix of the path for the output extent information.

LFS_M Class

The *LFS_M* class implements the *LFS* module. It provides a plugin board for the LFS-specific data layout mechanisms. Those data layout mechanisms are implemented by classes inherited from the base class *ILFS* (e.g., *EXT3*). The *LFS_M* class receives the page-based requests from the *VFS* or *DiskCache*, generate the block-based requests according to the specific *LFS* functionalities, and send them to the *Disk* module. When a block-based I/O response comes back from the *Disk* module, the above process is reversed and the page-based response will be sent to the *VFS* or *DiskCache*.

ILFS Class

The *ILFS* class is the base class for the classes implementing the local disk file system data layout mechanisms. It defines the interfaces for the *LFS_M* class to interact with the classes implementing LFS-specific data layout mechanisms.

EXT3 Class

The *EXT3* class realizes the functionalities of the EXT3 local disk file system. *EXT3* sets up the data layout of local files from the input script, and reads/updates it through the simulation. At the end of the simulation, the updated data layout file is written to the output file. To allocate blocks for new file data, *EXT3* will jump the size *new_ext_gap* from current end of the file, and allocates a new extent of the size *new_ext_size*. Currently we assume the metadata for each extent is stored on the block before that extent. When an I/O accesses an extent never accessed before, the corresponding metadata will be read first.

Folder: dserver/disk/

Disk Module

The *Disk* module is defined in the file *Disk.ned*. It simulates the disk system on the data server. The *Disk* module has an inout gate, which is connected with the *LFS* module. The *Disk* class implements the *Disk* module.

Configuration Table

Name	Description
disk_size	The size of the local disk.
ra_size	The read-ahead size of I/O on the local disk.
degree	The maximum number of concurrent requests can be dispatched on the local disk.
disk_parm_path	The path of the disk parameter file.
return_zero	This flag has two possible values: 0 and 1. If it is 1, the disk head will return to zero offset on the disk on a period of *return_zero_period*.
return_zero_period	If *return_zero* is set, this is the period that the disk head returns to zero offset.

Disk Class

The *Disk* class implements the Disk module. In the current implementation, we have the disk module implemented in OMNeT++, and the disk access time is simulated according to the parameters in the disk parameter file (refer to the *Disk Parameter File* subsection). In the current version, we don't have DiskSim provided for the disk simulation in the current version. But in the near future, we are going to provide DiskSim as an optional configuration for the users.

Folder: router/

Router/Switch Module

The *Router* module is defined in *Router.ned*. It is able to simulate the basic functionalities of a router/switch in the network. The inout gates can be configured to fit the network topology.

Routing Class

The *Routing* class simulates the routing/switching functionality inside a router/switch in the network. It forwards the packets to different interfaces on the router/switch according to the packet kinds and IDs.

Queue Class

The *Queue* class simulates the queues at the interfaces on the router/switch.

Folder: proxy/

Proxy Module

A *Proxy* module simulates the proxy entity in a real network. This *Proxy* class intercepts the data I/O requests as well as the data I/O responses. Scheduling algorithms can be implemented as independent classes by inheriting the *IQueue* class, and they are plugged into the *Proxy* class.

Configuration Table

Name	Description
algorithm	The scheduling algorithm used on the proxy (Refer to file *General.h*).
degree	The maximum number of concurrent requests on the local disk.
newjob_proc_time	The extra delay on the proxy to process a new request.
finjob_proc_time	The extra delay on the proxy to process a finished request.
numApps	The number of applications served by the proxy. This information may be used by the scheduler (e.g., SFQ).

IQueue Class

IQueue is the base class for any scheduling algorithms.

FIFO Class

This class is inherited from the *IQueue* class and it realizes the FIFO scheduling algorithm. This algorithm maintains a queue for the awaiting requests (*waitQ*), and a queue for the outstanding requests on the server (*osQ*). The user is able to track the number of outstanding requests in the system, and set up a degree to restrict the number of outstanding requests on the server.

SFQ Class

This class inherits from the IQueue class and it realizes the Start-time Fare Queuing scheduling algorithm. The reference is:

http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.107.1054&rep=rep1&type=pdf

Same as the *FIFO* algorithm, this algorithm maintains a queue for the awaiting requests (*waitQ*), and a queue for the outstanding requests on the server (*osQ*). The user is able to track the number of outstanding requests in the system, and set up a degree to restrict the number of outstanding requests on the server.

Folder: Packets/

Packets.msg

bPacket Class

The *bPacket* class is the base class for all requests / responses simulated in PFSsim. This class provides the basic information for the *IQueue* based classes: *ID* for the ID of packet. *SubID* is used when the *ID* is not enough, typically inside the data servers (a packet is split into multiple sub-requests). *App* stands for the ID of the application the request comes from. *Size* specifies the length of the packet. *ClientID* is the ID of the client associated with this packet. *SubReqNum* records the number of sub-requests this request triggers.

gPacket Class

The *gPacket* class is mainly used for data transfer between network entities, and it may also be used for event triggering (the *SELF* event kind) internally. It can be used for recording the detailed time stamps when processing the data. The offset is stored to highoffset and lowoffset in order to prevent overflow of values in the offset (*.msg* files do not support long long type). The *gPacket* class records plenty of timestamps through the I/O service process. But it may also be common that the users observe some of the timestamps are not updated (having the 0 value) after the simulation. This is because some *gPackets* do not participate in the entire I/O process between the *Client* and *Dserver*. The reasons can be the packet is generated on the fly on the *Dserver*, or it is a metadata request that does not trigger I/O processing on the lower levels of the data server.

qPacket Class

The *qPacket* class is used for querying and replying the data layout information between *Client* and *Mserver*. It has the following fields for representing a file's layout information: *FileID* is for the ID of the target file. *DsNum* is for the number of *Dservers* storing this file. *DsList[]* is the list of the IDs of the *Dservers* that stores the file. *DsStripeSizes[]* is a list of the sizes of stripes storing this file on the *Dservers.*

sPacket Class

The *sPacket* is the inter-schedule information packet, used for the communications among schedulers. Users can customize the fields in *sPacket* for specific scheduling algorithms.

PageRequest Class

The *PageRequest* class is mainly used by the *DiskCache* class for the page I/O requests.

BlkRequest Class

The *BlkRequest* class is mainly used by the *LFS* and *Disk* classes for the block I/O requests.

Input and Output Files

Trace File

The trace files are in the folder traces/. A trace file is named with a common prefix with a 5-digit ID:

the first 3 digits start from 000 meaning the client ID, the last 2 digits start from 00 meaning the trace ID on the specific client. Because in the real world, each client can have multiple applications running, each trace file will be read by a *Trace* on the client with the corresponding client ID. Each line in the trace file represents a request submitted by the system client. The prefix of trace file paths is defined in *trc_path_prefix* in the *Client* module parameter table. The following is the format of the traces.

The Format of Traces

Name	Description
time	Waiting time since the finish of previous request (sync mode) or the start timestamp (async mode).
file ID	ID of the target file.
offset	Start offset of the I/O in the target file.
size	Size of the requested I/O.
R/W	Read/write flag of the I/O.
application ID	The application ID of the I/O request.
synchronous mode	If set (1), the start of this request is based on the finish of the previous one; otherwise (0) it is based on the simulated system time.

Result File

The result file contains the information of traces and the data packets in the simulation. The corresponding information is written when the trace or the data packet is finished. The result file format can be defined by the system user in the *trcStatistic()* and *pktStatistic()*functions in the *Client* class. The path prefix of the files is defined in *rslt_path_prefix* in the *Client* module parameter table. The ID naming rule is the same as that of the trace files.

Layout File

The layout files are read by the metadata servers to set up the data layout for each PFS file. It must cover all file IDs appearing in the trace files. The layout file is read by the metadata server at the beginning of the simulation. Each line in the layout file represents the data layout for an application.

The format of each line in the layout file is:

file_ID [ds_ID ds_stripesize] [ds_ID ds_ stripesize] ...

In each *[ds_ID ds_stripesize]*, the *ds_ID* means the ID of the *Dserver* that participates in the storage of the file data, the *ds_stripesize* means the size of the data stripe stored on this data server. In the default data layout strategy, the data of one application is stored in a round-robin manner on the corresponding data servers, and each data server only stores the data of *ds_stripesize* size in each round.

LFS Extent File

The LFS extent files are accessed by the *LFS* module for the layout information (about the data extents) for each file on the disk (specific to the extent-based local disk file systems). There are two LFS extent files: the input extent file and the output extent file. The prefixes of the two files are defined in *ext_in_path_prefix* and *ext_out_path_prefix* in the configurations of the *LFS* module. The path prefixes are concatenated with a 3-digit ID starting from 000 to form the complete paths. The input extent file is read at the simulation start to initialize the existing files in the local file system. And the output file is written at the simulation end to output the new file data layouts. The format of the LFS extent file is as follows:

File ID:

logical_off   physical_off   length

logical_off   physical_off   length

…

File ID is the ID of the target file. *Logical_off* is the logical offset (in the file) of the extent's first bit. *Physical_off* is the physical offset (on the disk) of the extent's first bit. *Length* is the length of the extent.

Disk Parameter File

The disk parameter file is used for providing the disk I/O time overheads for the OMNeT++ based *Disk* module. The prefix of the paths for the disk files is defined in *disk_parm_path* parameter in the *Disk* module. To form the complete path, PFSsim concatenates the prefix with an ID in the same way as the LFS extent file. In the disk parameter file there are a set of records. Each record takes two lines:

First line: **[*rw/rr/sw/sr   size*]**

This line specifies the I/O pattern and size. *rw* means random write, *rr* means random write, *sw* means sequential write, *sr* means sequential read. *size* means the size of the data I/O.

Second line: ***data   [data   data …]***

The data represent the time consumption of I/O on the disk, and they are in the unit of millisecond. If the I/O pattern is "random" (*rr* or *rw*), there will be multiple data values, representing the I/O time for different "jump sizes" (the space the disk head must move on the disk before the start of this I/O): 1 block, 2 blocks, 4 blocks, 8 blocks … The "jump sizes" are increased in a square manner. If the I/O pattern is "sequential" (*sr* or *sw*), there will be only one data value, representing the I/O time for the sequential access.

When a disk I/O comes to the *Disk* module, the access time is calculated by using linear approximation based on the existing disk parameters.