IOInsight User Manual

1 Overview

IOInsight is a virtual appliance that provides a framework to monitor VM I/Os and run various analysis on them. The IOinsight appliance comes with two built-in analysis plugins, (i) to characterize the VM I/O workload with respect to basic metrics like read-write ratio, sequentiality, etc., and (ii) to estimate the cache friendliness of the workload. In addition to this, if the user wants to develop new analyzer plugins, the IOInsight SDK would be useful. In this manual, we cover various information on how to implement new analyzer plugins.

The following figure shows the basic architecture of IOinsight. The IOinsight has a monitor component that captures I/O metrics like time of issue, IO type, IO size, IO offset, etc., for every I/O that a VMDK issues and these are fed to an ephemeral database. These I/O metrics are then consumed by the various analysis plugins for appropriate characterization. The analysis plugins then store their results in a database that is served to the user by means of graphical visualization and raw data. Even though we capture information for every I/O, it is consumed by the analyzer plugins in a live manner and then discarded. Because of this, there is only minimal overhead in terms of storage space, enabling long durations (like multiple days) of monitoring.

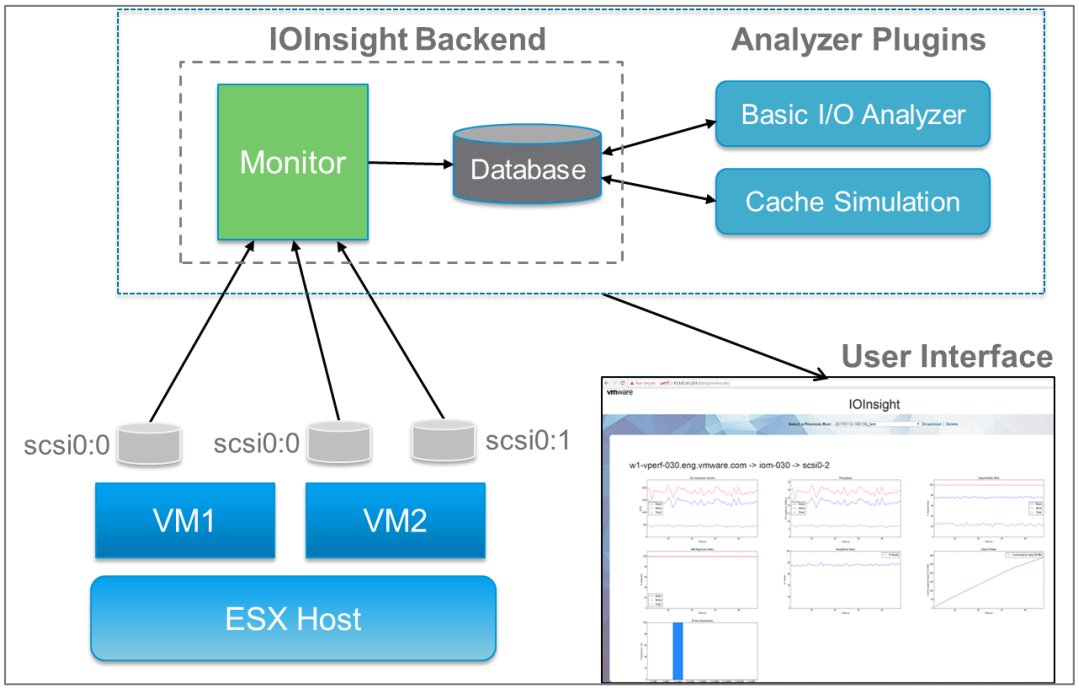


Figure 1 IOInsight Framework Design

IOInsight captures 3 metrics for every I/O:

1. TYPE : Type of the operation. Value is either IOTYPE_READ or IOTYPE_WRITE
2. SIZE : Size of the IO in bytes
3. LBA : Logical Block Address of the IO

An analyzer can use either all or a subset of these metrics for analysis. This can be defined in the analyzer configuration as *'inputMetrics'* field.

2 Built-In Analyzers

IOInsight virtual appliance comes with two built-in analyzers namely “Basic I/O Analyzer” and “Cache Simulation”. While running IOInsight, users can choose either one or both these analyzers.

2.1 Basic I/O Analyzer:

The basic I/O analyzer characterizes the I/O workload with respect to the following metrics:

1. Number of I/Os issued per second (Read, Write)
2. Amount of data in MBps issued per second (Read, Write)
3. Number of I/Os that are sequential vs. non-sequential (Read, Write)
4. Number of I/Os whose LBAs (offsets) are aligned to 4KB-boundaries
5. Percentage of read I/Os among total I/Os
6. Cumulative data fill rate: This shows the cumulative amount of data that is written to disk from the start of the monitoring duration. Here we ignore write I/Os that repeatedly go to a particular location. Only the first writes to a particular location is counted. This gives an approximate view of the rate of growth of a thinly-provisioned disk.
7. IO Size histogram (total IO)

Input: LBA, TYPE, SIZE for each I/O

Output: Graphs showing metrics 1-7 mentioned above.

2.2 Cache Simulation:

The cache simulation analyzer is designed to provide an accurate estimate of cache hit rate achievable for the workload for any given cache size. This can be useful in optimal sizing storage-level caches in storage arrays or vSAN. For this purpose, this analyzer runs a full simulation of Adaptive Replacement Cache (ARC) and the I/O workload captured by the monitor is fed to this simulation to provide accurate cache hit rate.

Cache size and cache block size are the two tunable parameters for this analyzer. Multiple cache sizes and cache block sizes can be specified here and parallel simulations are run for each of these configurations giving individual cache hit rates.

Input: ‘n’ cache configurations and LBA, SIZE, TYPE for each I/O.

Output: For each cache configuration, graphs showing (i) Read I/Os per second and (ii) Cache Hit Ratio per second.

The reason for displaying both read I/Os per second and the cache hit rate is that, the cache hit rate is calculated as the following:

$$\text{Cache Hit Rate} = \frac{\text{Number of read I/Os hit in cache}}{\text{Total number of read I/Os}}$$

Therefore, when the number of read I/Os is less, the impact of having a low cache hit rate is less and vice versa.

3 Advanced Mode: Developing new analyzer plugins:

In addition to the analyzers that comes built-in with the appliance, users can come up with their own analysis methods for the monitored metrics by developing a new analyzer. IOInsight framework is designed in such a way that new plugins can be developed and easily plugged in.

3.1 High-level Steps:

- The '`example`' analyzer source code, which is distributed along with ova, can be used as a reference for developing new analyzers.
- Start with a new directory for your analyzer files. Minimally it would contain a separate binary for your analyzer and a configuration file.
- The configuration file should be named '`analyzerCfg.json`'. A template for the configuration file is also provided in the IOInsight SDK. Refer section:3.2 for description of each fields in the configuration.
- Implement the analyzer in C using IOInsight APIs defined in `ioinsight.h`. APIs are explained in section:3.4
- Analyzer binary should be built by linking the `libioinsight.a` library, which ships as part of IOInsight SDK.
- New analyzer can be added or removed from IOInsight using commandline options `register` and `unregister` respectively, detailed in section:3.3

3.2 Analyzer Configuration

Analyzer configuration should be specified in a file named `analyzerCfg.json` as per the given template file `analyzerCfg_template.json`. This configuration file should be present in the same directory as the analyzer binary at the time of registering new analyzer.

Here is an example configuration file:

```
{
   "name": "analyzerName",
   "displayName": "Display Name",
   "description": "A sample analyzer",
   "inputInterval": 1,
   "inputMetrics": [
      "LBA",
      "SIZE",
      "TYPE"
  ],
   "outputMetrics": [
      "readthpt",
      "writethpt",
      "totalthpt",
      "iosize4k",
      "iosize8k",
      "iosize16k"
   ],
   "userInput": [
      "blockSizeInKB",
      "displayString"
   ],
   "repeat": true,
   "graphCfg": {
      "timeseries": [
```

```
            {
               "title": "IO Throughput",
               "ylabel": "Throughput (MB/s)",
               "ylimit": [0, 500],
               "metrics": [
                  "readthpt",
                  "writethpt",
                  "totalthpt"
               ],
               "metriclabel": [
                  "Read",
                  "Write",
                  "Total"
               ]
            }
         ],
         "histogram": [
            {
               "title": "IO distribution",
               "xlabel": "IO Size",
               "ylabel": "Percentage",
               "ylimit": [0, 100],
               "metrics": [
                 "iosize4k",
                 "iosize8k",
                 "iosize16k"
               ],
               "metriclabel": [
                 "<=4KB",
                 "<=8KB",
                 "<=16KB"
               ]
            }
         ]
      },
      "outputInterval": 1
}
```

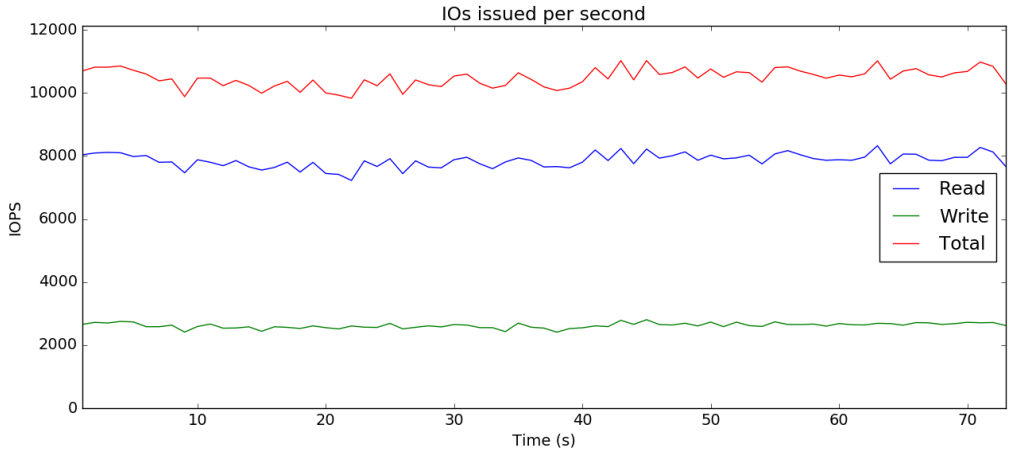
Field Name	Description
name	Name of analyzer binary. This should be a single word alphanumeric name with a maximum length of 31 characters
displayName	Name to be displayed in UI (multiple words are allowed). Max 31 characters.
description	Longer description of the analyzer. Maximum 255 characters.
inputInterval	Interval in which monitor metrics are consumed by the analyzer
inputMetrics	List of monitor metrics used by this analyzer
outputInterval	Interval in which output is generated by the analyzer
outputMetrics	List of results parameters generated by the analyzer

userInput	List of configuration parameters to be provided by the user at run time in UI. This array can be left as empty if no configuration value is needed from user.
repeat	To specify whether multiple instances of userInput is allowed or not. If value is false, only one set of userInput parameters can be entered from UI. If it's set to true, UI will show an option to add multiple instances of the configuration values.
graphCfg	Specification of graphs to be generated using values of outputMetrics fields. Graphs can be either 1) Time series : output metric values plotted against time or 2) Histograms. Graph can be defined with the following fields
timeseries	List of specifications for generating time series graphs
histogram	List of specifications for generating histograms
title	Graph title
xlabel	X-axis label (Needed only for histograms. It's 'Time' for time series graphs)
ylabel	Y-axis label
metrics	List of output metrics to be plotted in a given graph. This should be a subset of outputMetrics.
metriclabel	List of labels corresponding to each field in the above `metrics` list
ylimit	This is an optional parameter to specify range of values on Y-axis. This should be specified in the form [minValue, maxValue]

Consider the following graph specification:

```
"graphCfg": {
"timeseries": [
  {
    "title": "IOs issued per second",
    "ylabel": "IOPS",
    "metrics": [
      "riops",
      "wiops",
      "iops"
    ],
    "metriclabel": [
      "Read",
      "Write",
      "Total"
    ]
  },
  { <insert another time series specs here> }
],
"histogram": []
}
```

This would generate a time series graph like the following.



- Corresponding to each element in '`metrics`' array, there would be a line in the time series graph
- Corresponding to each element in '`metrics`' array, there would be a bar in the histogram
- Histogram specification is identical to time series spec, with an additional parameter '**`xlabel`**'.
- Each entry in the `timeseries` or `histogram` array corresponds to a separate graph.
- Optional graph parameter '`ylimit`' can be used to specify range of values on Y-axis. For example, with **`"ylimit": [0, 100],`** the Y-axis would be marked from 0 to 100.

3.3 Registering/Unregistering a new analyzer

Once a new analyzer is implemented, it has to be register to the IOInsight framework in order to use it. This is done by invoking the following command line in the IOInsight shell:

`python /usr/local/bin/startIoinsight.pyc register –a <analyzerPath>`

`analyzerPath` : `Directory containing binary and configuration file of the new analyzer`

In the cases of making changes to an already registered analyzer, one has to first unregister the analyzer, modify the analyzer binary/configuration and then re-register again with IOInsight. The following command line is used for un-registering an analyzer:

`python /usr/local/bin/startIoinsight.pyc unregister –a <analyzerName>`

`analyzerName` : `'name' field in its analyzerCfg.json at the time of its registration`

List of registered analyzers and monitors can be viewed by:

`python /usr/local/bin/startIoinsight.pyc list`

IOInsight installation can be reset to factory settings with the following command:

`python /usr/local/bin/startIoinsight.pyc reset`

3.4 IOInsight API Reference

3.4.1 Initializing / Terminating Analyzer

Function	**Analyzer_Init()**	
Description	Initializes the analyzer. Should be called before any other API invokation.	
Arguments	`analyzerName`	Registered name of the analyzer being initialized
Return	Void	

Function	**Analyzer_Destroy()**
Description	To be called before exiting the analyzer process.
Arguments	None
Return	Void

Each analyzer is started with its registered name as a command-line argument, which can be used to initialize the analyzer.

```
int main(int argc, char *argv[]) {
   ...
   analyzerName = argv[1];
   Analyzer_Init(analyzerName);
   ...
   Analyzer_Destroy();
}
```

3.4.2 Getting IOInsight Running State

Function	**getRunState()**
Description	Gets running state of IOInsight
Arguments	None
Return	Returns RUNSTATE_RUNNING : If IOInsight monitoring is running RUNSTATE_TERMINATE : If IOInsight monitoring is terminated already

3.4.3 Reading Analyzer Configurations

Function	**Analyzer_GetConfigValues()**
Description	To retrieve run-time analyzer configuration values given by user. This can be a single set of analyzer parameters or multiple instances of analyzer parameters
Arguments	None
Return	Pointer to `AnalyzerConfig` object which contains values of all run-time parameters for all the instances

Function	**Analyzer_FreeConfigValues()**
Description	Free memory allocated for `AnalyzerConfig` member fields.
Arguments	Pointer to the `AnalyzerConfig` object returned by `Analyzer_GetConfigValues()`
Return	Void

Analyzers can optionally take run time analyzer configurations from the user. These can be retrieved using `Analyzer_GetConfigValues()`, which returns a pointer to `AnalyzerConfig`.

The following snippet shows an example where the analyzer takes two runtime configurations (cacheSize and blockSize) from user. `numInstances` means the number of different cacheSize-blockSize pairs given by the user.

```
#define NUM_USER_INPUTS   2   /* Elements in 'userInput' array in analyzerCfg.json */

AnalyzerConfig *analyzerCfg;

analyzerCfg = Analyzer_GetConfigValues();
numInstances = analyzerCfg->numInstances;
assert(analzyerCfg->numConfigs == NUM_USER_INPUTS);

/* User can pass one or more instances of configuration parameters.
 * One iteration corresponds to parsing of an instance of config parameters */
for(i = 0; i < numInstances; i++) {
   /* All config values are stored as string. Convert to the desired type */
   cacheSize = atol(analyzerCfg->values[i][0]);   /* First runtime config parameter */
   blockSize = atol(analyzerCfg->values[i][1]);   /* Second runtime config parameter */

   /* Do whatever we have to do based on run-time config values */
}
Analyzer_FreeConfigValues(analyzerCfg);
```

3.4.4 Retrieving Values of Monitored Metrics

Function	**Analyzer_GetNumMonitorData()**	
Description	Gets the number of samples of data collected by the monitor for the given datasource/VMDK in an interval	
Arguments	`s`	`DataSource` object referring to the source of data (VMDK)
	`tick`	Time interval
Return	Number of samples monitored in the given interval	

Function	**Analyzer_AllocateMonitorDataPayload()**	
Description	Allocate memory for storing	
Arguments	`mData`	Pointer to `MonitorData` object where retrieved monitored metric values should be stored. This should be invoked only if the value returned by `Analyzer_GetNumMonitorData()` is positive number and it should be assigned to `mData.numValues` before the invocation.
Return	Void	

Function	**Analyzer_GetMonitorData()**	
Description	Retrieve values of all metrics being monitored in a given interval	
Arguments	`tick`	Time interval. Invocation of the API should always start with value 0.
	`mData`	Pointer to `MonitorData` object to store retrieved monitor data. Valid `DataSource *` value should be assigned to `mData->s` before invoking this.
Return	Void	

Function	**Analyzer_FreeMonitorDataPayload()**	
Description	Free memory allocated for `MonitorData` member fields	
Arguments	`mData`	Pointer to `MonitorData` object
Return	Void	

Following code snippet shows an example for retrieving values of monitored metrics for all samples collected in an interval.

```
mData.s = dataSource;

while(getRunState() != RUNSTATE_TERMINATE) {
   count = Analyzer_GetNumMonitorData(dataSource, processedTick);

   if(count < 0 ) {
      /* In Progress */
      nanosleep(&sleepTime, NULL);
      continue;
   } else if (count == 0) {
      /* Idle second */
      ...
   } else  {
      mData.numValues = (uint32_t)count;
      Analyzer_AllocateMonitorDataPayload(&mData);
      Analyzer_GetMonitorData(processedTick, &mData);
      /* Insert code here to do analysis based on collected data/metrics */
      for(i = 0; i < count; i++) {
         /* Assumption: INDEX_IOTYPE refers to the
         char *ioType = mData.payload[INDEX_IOTYPE][i];
         if(strcmp(ioType, IOTYPE_READ) == 0) {
            readCount++;
         }
      }
      Analyzer_FreeMonitorDataPayload(&mData);
   }
   ...
}
```

3.4.5 Storing Analyzer Result Metrics

Function	**Analyzer_AllocateResultData()**	
Description	Creates `ResultData` object and allocate memory for storing values of result metrics of the analyzer in an interval. This should be passed to `Analyzer_StoreResultData()`	
Arguments	`numResultMetrics`	Number of registered output result metrics of the analyzer
	`resultFieldSize`	Maximum length of result metric value, if it's stored as a string (including null termination character)
Return	Pointer to `ResultData` object	

Function	**Analyzer_StoreResultData()**	
Description	Store results generated by the analyzer during a particular interval for a single analyzer configuration	
Arguments	`tick`	Time interval
	`rData`	Pointer to `ResultData` object. `rData->s` should be set with corresponding `DataSource` object before invoking this API.
	`cfgIndex`	Configuration number. By default, this value should be given as 0 and if there are multiple user configs, start with 0 and increment by 1.
Return	Void	

Function	**Analyzer_FreeResultData()**	
Description	Free memory allocated for `ResultData` object.	
Arguments	`rData`	Pointer to `ResultData` object.
Return	Void	

Following code snippet shows how result metrics generated for each interval can be stored. In this example, the analyzer generates two result metrics every interval and the maximum length of string representation of those values is RESULT_DATA_FIELD_SIZE.

```
ResultData *rData;
rData = Analyzer_AllocateResultData(2, RESULT_DATA_FIELD_SIZE);
rData->s = dataSource;
while(getRunState() != RUNSTATE_TERMINATE) {
   ...
   /* Get monitor data, analyze those and generate result metrics */
   ...
   // Assuming result metrics - result1 & result2 are evaluated in above code
segment
   sprintf(rData->payload[0], "%d", result1);
   sprintf(rData->payload[1], "%d", result2);
   Analyzer_StoreResultData(processedTick, rData, 0);
}
```

3.4.6 Logging

Log messages can be stored in analyzer's log file with these APIs. Analyzer log file will be located at:

`/IOInsight/logs/analyzer_<analyzerName>.log`

Function	**`LOG_ERROR() , LOG_INFO(), LOG_DEBUG()`**	
Description	To store messages in analyzer's log file.	
Arguments	`fmt`	Format string, similar to printf(). But unlike printf, newline characters are automatically added at the end of each log message.
Return	Void	

eg: `LOG_INFO("Initialized %s analyzer", analyzerName)`

Log message will be: **`Thu Dec 22 12:08:09 2016 [INFO] Initialized 'basic' analyzer`**

3.5 Guidelines

The following guidelines shall be followed while developing analyzers.

1. Time interval (tick) for retrieving monitor data should start from 0 and increment by 1
2. Monitored data using **`Analyzer_GetMonitorData()`** shall be consumed as soon as it's available. When the monitor metric values for an interval is consumed by all running analyzers, IOInsight deletes it from the database. Large delay in consuming data by the new analyzer can result in the in-memory database to grow and eventually exhausting memory.
3. Freeing memory: Every API which involves memory allocation must have a corresponding API call to free memory. These pairs of APIs shall be used together
   a. `Analyzer_Init() & Analyzer_Destroy()`
   b. `Analyzer_AllocateMonitorDataPayload() & Analyzer_FreeMonitorDataPayload()`
   c. `Analyzer_AllocateResultData() & Analyzer_FreeResultData()`
   d. `Analyzer_GetConfigValues() & Analyzer_FreeConfigValues()`
4. Prerequisites / Dependencies for invoking APIs
   a. `Analyzer_Init()` must be called before any other API invocation
   b. `Analyzer_AllocateMonitorDataPayload()` must be invoked only if the value returned by `Analyzer_GetNumMonitorData()` is positive number and this value must be assigned to `mData->numValues` before the invocation
   c. `Analyzer_AllocateMonitorDataPayload()` must be called before `Analyzer_GetMonitorData()`
   d. `mData->s` must be set before invoking `Analyzer_GetMonitorData()`
   e. `Analyzer_AllocateResultData()` must be called before `Analyzer_StoreResultData()`
   f. `rData->s` must be set before invoking `Analyzer_StoreResultData()`
   g. If `getRunState()` returns `RUNSTATE_TERMINATED`, analyzer must be terminated and no further calls to get monitor data should be made. Run state should be checked at least once every second

4 Best Practices

4.1 IOInsight VM Configuration

- vCPUs: Minimum 2, 4 if analyzers with heavy processing are added
- Provisioned memory : >= 2GB

4.2 Analysis/Run configuration

- Monitoring large number of VMDKs in parallel can result in high overhead if IOPS is very high for those VMDKs. We recommend selecting <8 VMDKs for a run.
- Cache Simulation analyzer is CPU-intensive. We recommend running cache simulation for 1 or 2 cache configurations at a time for better performance. Increase number of VCPUs for more parallel configurations.
- Recommended analysis duration is at least 10mins (default: 30min). The monitoring duration depends on applications. Typically, workloads follow similar patterns repeatedly. Some applications have a particular pattern for a few hours and then repeat the same pattern multiple times, while some other applications may have a pattern that repeats every day. Therefore, it is best to run monitoring for one such repeating cycle.
- IOInsight VM should be ideally deployed in the same/closer network as the hosts being monitored. Otherwise, larger RTTs may result in slower response times.

Supported Browsers: Google Chrome and Apple Safari