PyMimircache Documentation

Release 0.0.2.103

Juncheng Yang

Nov 25, 2019

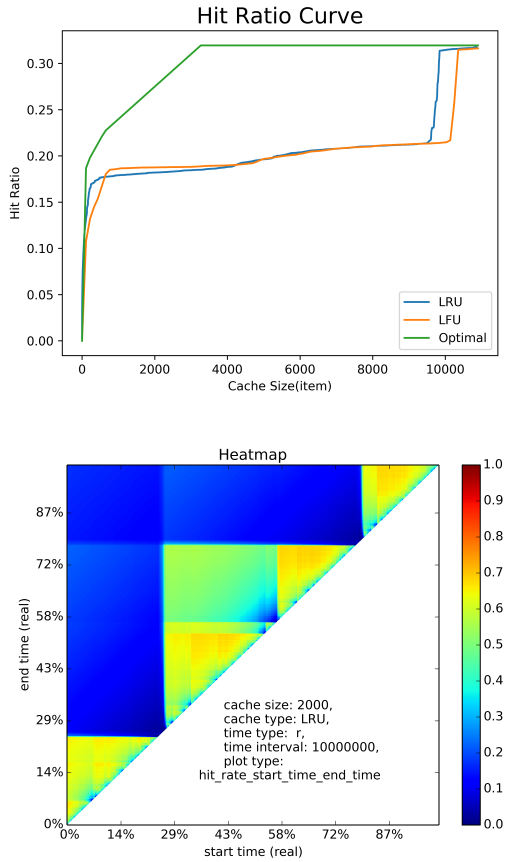
Contents

Release v0.0.2.103.

Welcome to the documentation of PyMimircache, a Python3 cache analysis platform. The target users of PyMimircache are **researchers** and **system administrators**. The goal behind PyMimircache is to provide a platform that

- allows **researchers** to study and design cache replacement algorithms easily and efficiently.
- allows **system administrators** to analyze and visualize their cache performance easily and efficiently.

The power of PyMimircache:



An example of hit ratio curve plot and hit ratio heatmap.

```
>>> from PyMimircache import Cachecow
>>> c = Cachecow()
>>> c.vscsi("trace.vscsi")       # this file is in the data folder on GitHub, other␣
↪data types also supported
>>> print(c.stat())
>>> print(c.get_reuse_distance())
[-1 -1 -1 -1 -1 -1 11 7 11 8 8 8 -1 8]
```

```
>>> print(c.get_hit_ratio_dict("LRU", cache_size=20))
{0: 0.0, 1: 0.025256428270338627, 2: 0.031684698608964453, ... 20: 0.
↪07794716875087819}
```

```
>>> c.plotHRCs(["LRU", "LFU", "Optimal"])
```

```
>>> c.heatmap('r', "hit_ratio_start_time_end_time", time_interval=10000000)
```

CHAPTER 1

The User Guide

1.1 Introduction

The study of cache has a long history; however, there is no single open-source platform for easy and efficient analysis of cache traces. That's why we are building PyMimircache, a Python3 platform for analyzing cache traces. Note that PyMimircache only works on Python3, not Python2.

The target users of PyMimircache are **researchers** and **system administrators**. The goal behind PyMimircache is to provide a platform that

- allows **researchers** to study and design cache easily and efficiently.
- allows **system administrators** to analyze and visualize their cache performance easily and efficiently.

The KEY philosophy is that we would like to design a cache analysis platform that is **efficient**, **flexible** and **easy to use**. With these in mind, we designed PyMimircache in Python3 for easy usage, and we implemented state-of-the-art algorithms in C as the backend for efficiency. However, PyMimircache can also be used without the C backend. In other words, PyMimircache depends on CMimircache (backend), but you can use either of them independently. Besides, PyMimircache allows the user to plug in an external reader for reading special data and allows the user to write their own cache replacement algorithm easily.

1.1.1 Evaluate and Design Algorithm

PyMimircache supports **comparison of different cache replacement algorithms**, including Least Recent Used (LRU), Least Frequent Used (LFU), Most Recent Used (MRU), First In First Out (FIFO), Clock, Random, Segmented Least Recent Used (SLRU), Optimal, Adaptive Replacement Cache (ARC). We are actively adding more cache replacement algorithms. For an extensive list of supported cache replacement algorithms, see here.

Best of all is that you can easily and quickly **implement your own cache replacement algorithm**.

For all cache replacement algorithms, including the ones built-in and the ones you implement yourself, PyMimircache supports all kinds of comparison, **there is nothing you can't do, there is only things that you can't imagine**.

To help you better evaluate different cache replacement algorithms, we also include a variety of visualization tools inside PyMimircache. For example, you can plot the hit ratio curve (HRC), the miss ratio curve (MRC), different

variants of heatmaps and differential heatmaps. For LRU, it also supports efficient reuse distance calculation, reuse distance distribution plotting, etc.

1.1.2 Visualize and Analyze Workload

Another great usage of PyMimircache is **understanding workloads** so that you can be the tailor of your cache, **design better strategy** to cache your data or know why your cache has certain behavior and how your cache behaves with time.

In this part, we have figures that show you the hit ratio over time, request rate over time, object popularity distribution, reuse distance distribution, and different types of heatmaps. These show the opportunity of getting better cache performance.

1.1.3 Performance, Flexibility and Easy-to-Use

Three features provided by PyMimircache are **high performance**, **flexibility**, and **easy usage**.

- **Performance**: PyMimircache uses CMimircache with state-of-the-art algorithm as backend for best performance.
- **Flexibility**: PyMimircache can also be used without CMimircache, thus using all Python-based modules. However,

both usages have the same interface, so no need to learn different tools. Besides, PyMimircache supports three types of readers: PlainReader for reading plain text data, CsvReader for reading csv data, and BinaryReader for reading arbitrary binary data. We also supports VscsiReader for reading vscsi data, which is a special type of binary data. If your data is in a special format, don't worry! You can easily implement your own reader within a few lines, and you are good to go! - **Easy Usage**: We provide Cachecow as the top-level interface, which provides most of the common usages. Alternatively, you can easily plug in a new algorithm to see whether it can provide better performance than existing algorithms.

1.1.4 Work in Progress

- More algorithms.
- Connection with Memcached and Redis.
- Windows support.
- GPU support.

1.2 Installation

This part of the documentation covers the installation of PyMimircache. PyMimircache currently has the following dependencies:

pkg-config , glib , scipy , numpy , matplotlib,

PyMimircache has been tested on Python3.4, Python3.5, Python3.6. **Mac Users**: if you don't know how to install these packages, try macports or homebrew; google will help you.

1.2.1 General Installation (pip)

Using pip3 is the preferred way for installing PyMimiracache, notice that PyMimircache does not support python2.

First Step: Install C Library

First: use any package management software to install pkg-config and glib.

Second Step: Install Python Dependencies

Use pip3 to install python dependencies:

```
$ sudo pip3 install matplotlib heapdict
```

Third Step: pip Install PyMimircache

To install PyMimircache, simply run this command in your terminal of choice:

```
$ sudo pip3 install PyMimircache
```

1.2.2 Install From Source

This is an alternative method, only use this one when you can't install using the methods above, or you want to try the newest feature of PyMimircache. Beware that it might have bugs in the newest version. We highly recommend that you to use the stable version from pip3.

Install All Dependencies

Installing all dependencies is the same as in General Installation .

Get the Source Code

PyMimircache is actively developed on GitHub, where the code is always available here.

You can clone the public repository:

```
$ git clone -b master --recurse-submodules git://github.com/1a1a11a/PyMimircache.git
```

Once you have a copy of the source, you can install it into your site-packages easily:

```
$ sudo python3 setup.py install
```

1.2.3 Install Using Docker Container

As an alternative, you can use PyMimircache in a docker container,

Use interactive shell

To enter an interactive shell and do plotting, you can run:

```
sudo docker run -it --rm -v $(pwd):/PyMimircache/scripts -v PATH/TO/DATA:/
↪PyMimircache/data 1a1a11a/PyMimircache /bin/bash
```

After you run this command, you will be in a shell with everything ready, your current directory is mapped to */PyMimircache/scripts/* and your data directory is mapped to */PyMimircache/data*. In addition, we have prepared a test dataset for you at */PyMimircache/testData*.

Run scripts directly

If you don't want to use an interactive shell and you have your script ready, then you can do:

```
docker run --rm -v $(pwd):/PyMimircache/scripts -v PATH/TO/DATA:/PyMimircache/data␣
↪1a1a11a/PyMimircache python3 /PyMimircache/scripts/YOUR_PYTHON_SCRIPT.py
```

However, if you are new here or you have trouble using docker to run scripts directly, we suggest using an interactive shell which can help you debug.

1.2.4 Special Instructions for Installing on Ubuntu

First Step: Install Python3, pip and all dependencies

** If you are using Ubuntu 12**, you need to do the first step. Since Ubuntu 12 does not come with Python3 or above and some related components like pip, you can either compile Python3 from source or add the repository from Ubuntu 14 into your source list. Below are the instructions for using Ubuntu 14 repository to install Python3 and pip.

Add the following two lines to the top of /etc/apt/source.list

```
deb http://us.archive.ubuntu.com/ubuntu/ trusty main restricted universe multiverse
deb-src http://us.archive.ubuntu.com/ubuntu/ trusty main restricted universe␣
↪multiverse
```

Then update your repository by:

```
$ sudo apt-get update
```

If you are using Ubuntu 14 and above, start here, Ubuntu 12 continues here:

Now install python3, python3-pip and all dependencies:

```
$ sudo apt-get install python3 python3-pip python3-matplotlib pkg-config libglib2.0-
↪dev
```

Second Step: Install PyMimircache

```
$ sudo pip3 install PyMimircache
```

Congratulations! You have finished installation, welcome to the world of PyMimircache

1.3 Quick Start

1.3.1 Get Prepared

With PyMimircache, testing/profiling cache replacement algorithms is very easy. Let's begin by getting a cachecow object from PyMimircache:

```
>>> from PyMimircache import Cachecow
>>> c = Cachecow()
```

1.3.2 Open Trace File

Now let's open a trace file. You have three choices for opening different types of trace files. Choose the one that suits your needs.

```
>>> c.open("trace/file/location")
>>> c.csv("trace/file/location", init_params={'label':x})  # specify which column␣
↪contains the request key(label)
>>> c.vscsi("trace/file/location")           # for vscsi format data
>>> c.binary("trace/file/location", init_params={"label": x, "fmt": xxx})    # use␣
↪same format as python struct
```

see *here* for details.

1.3.3 Get Basic Statistics

You can get some statistics about the trace, for example how many request, how may unique requests.

Functions	Parameters	Description
num_of_req	None	return the number of requests in the trace
num_of_uniq_req	None	return the number of unique requests in the trace
stat	None	get a list of statistical information about the trace
characterize	type (short/medium/long/all)	plot a series of fig, type indicates run time
len	None	return the number of requests in the trace

If you want to read your data from cachecow, you can simply use cachecow as an iterator, for example, doing the following:

```
>>> for request in c:
>>>     print(c)
```

1.3.4 Profiler and Profiling

Now cachecow supports basic profiling, to conduct complex profiling, you still need to get a profiler. With a profiler, you can obtain the reuse distance of a request, the hit count and hit ratio of at a certain size, you can even directly plot the hit ratio curve (HRC). See *here* for details.

cachecow supports two type of profiling right now, calculate reuse distance and calculate hit ratio. The syntax is listed below.

```
>>> # get an array of reuse distance
>>> c.get_reuse_distance()
>>> # get a dictionary of cache size -> hit ratio
>>> c.get_hit_ratio_dict(algorithm, cache_size=-1, cache_params=None, bin_size=-1)
```

See *API-cachecow* section for details.

1.3.5 Two Dimensional Plotting

cachecow supports the following two dimensional figures,

plot type	required parameters	Description
cold_miss_count	time_mode, time_interval	cold miss count VS time
cold_miss_ratio	time_mode, time_interval	coid miss ratio VS time
request_rate	time_mode, time_interval	num of requests VS time
popularity	NA	Percentage of obj VS frequency
rd_popularity	NA	Num of req VS reuse distance
rt_popularity	NA	Num of req VS reuse time
mapping	NA	mapping from original objID to sequential number
interval_hit_ratio	cache_size	hit ratio of interval VS time

The basic syntax for plotting the two dimensional figures is here

```
>>> # see table for plot_type names
>>> c.twoDPlot(plot_type, **kwargs)
```

See *API-twoDPlots* section and *basic plotting* for details.

1.3.6 Hit Ratio Curve Plotting

cachecow supports plotting against a list of cache replacement algorithms, using the following syntax:

```
>>> plotHRCs(algorithm_list, cache_params=(), cache_size=-1, bin_size=-1, auto_
↪resize=True, figname="HRC.png", **kwargs)
```

See *API-LRUProfiler and API-cGeneralProfiler* section and *basic plotting* for details.

1.3.7 Heatmap Plotting

cachecow supports basic heatmap plotting, and supported plot type is listed below.

```
>>> # plot heatmaps
>>> heatmap(time_mode, plot_type, time_interval=-1, num_of_pixels=-1, algorithm="LRU",
↪ cache_params=None, cache_size=-1, **kwargs)
>>> # plot differential heatmaps
>>> diff_heatmap(time_mode, plot_type, algorithm1, time_interval=-1, num_of_pixels=-1,
↪ algorithm2="Optimal", cache_params1=None, cache_params2=None, cache_size=-1,␣
↪**kwargs)
```

plot type	Description
• hit_ratio_start_time_end_time	Hit ratio heatmap of given start time and end time
• hit_ratio_start_time_cache_size (python only)	Hit ratio heatmap of given start time and cache size
• avg_rd_start_time_end_time (python only)	Average reuse distance of start time and end time
• cold_miss_count_start_time_end_time (python only)	deprecated
• rd_distribution	Heatmap of reuse distance distribution over time
• rd_distribution_CDF	Heatmap (CDF) of reuse distance distribution over time
• future_rd_distribution	Heatmap of future reuse distribution over time
• dist_distribution	Heatmap of distance distribution over time
• reuse_time_distribution	Heatmap of reuse time distribution over time

Heatmap plotting section describes how to use PyMimircache to plot heatmaps. See *API-cHeatmap* section and *here* for details.

Congratulations! You have finished the basic tutorial! Check *Advanced Usage* part if you need.

1.4 Open Different Traces

1.4.1 Supported Trace File Type

- plain Text
- csv file
- binary file
- vscsi trace

1.4.2 How to Open a Trace File

Now let's open a trace file. You have three choices for opening different types of trace files. Choose the one that suits your needs.

```
>>> import PyMimircache as m
>>> c = m.cachecow()
>>> c.open("path/to/trace")
```

(continues on next page)

(continued from previous page)

```
>>> c.csv("path/to/trace", init_params={'label':x})  # specify which column contains␣
↪the request key(label)
>>> c.binary("path/to/trace", init_params={"label": x, "fmt": xxx})   # use same␣
↪format as python struct
>>> c.vscsi("path/to/trace")                          # for vscsi format data
```

Note: for csv and binary data, the column/field number begins from 1, so the first column(field) is 1, the second is 2, etc. In the init_params, other possible parameters are listed in the table below

Keyword Argument	relavant file type	Possible Value	Default Value	Description
label	csv/binary	int	this is required	the column of label of the request
fmt	binary	string	this is required	fmt string of binary data, same as python struct
header	csv	True/False	False	whether csv data has header
delimiter	csv	char	“,”	the delimiter separating fields in the csv file
real_time	csv/binary	int	NA	the column of real time
op	csv/binary	int	NA	the column of operation (read/write)
size	csv/binary	int	NA	the column of block/request size

OK, data is ready, now let’s play!

If you want to read your data from cachecow, you can simply use cachecow as an iterator. For example, do the following:

```
>>> for request in c:
>>>     print(c)
```

Note: If you have a special data format, you can write your own reader in a few lines, see *here* about how to write your own cache reader.

1.5 Get Profiler/Do Profiling

Profiler is one component of PyMimircache, which can be used for profiling and plotting, including getting reuse distance and hit ratio curve plotting.

1.5.1 Profiling with LRU

First, let’s try LRU (least recently used), here is how to get a LRUprofiler:

```
>>> profiler_LRU = c.profiler('LRU')
```

- To get reuse distance for each request, simply call the functions below. The returned result is a numpy array:

```
>>> reuse_dist = profiler_LRU.get_reuse_distance()
array([-1, -1, -1, ..., -1, -1, -1], dtype=int64)
```

- To get hit count, hit ratio or miss ratio, we can do the following:

```
>>> profiler_LRU.get_hit_count()
array([0,  2685,  662, ...,   0,   0,  48974], dtype=int64)
```

Hit count is a numpy array. The nth element of the array means that among all requests, that many requests will be able to fit in the cache if we increase cache size from n-1 to n. The last two elements of the array are different from all others, the second to the last element indicates the number of requests that needs larger cache size (if you didn't specify the cache_size parameter, then it is 0). The last element says the number of cold misses, meaning the number of unique requests.

```
>>> profiler_LRU.get_hit_ratio()
array([ 0,  0.02357911,  0.02939265, ...,  0.56992061,   0,  0.43007939])
```

Hit ratio is a numpy array. The nth element of the array means the hit ratio we can achieve given cache size of n. Similar to hit count, the last two elements give the ratio of requests that needs larger cache size and the ratio of requests that are unique.

```
>>> profiler_LRU.get_miss_ratio()
array([ 1,  0.97642089,  0.97060735, ...,  0.43007939,   0,  0.43007939])
```

Miss ratio is a numpy array. The nth element of the array means the miss ratio we will have given cache size of n. The last two elements of the array are the same as that of the hit ratio array.

Note: for reuse distance, hit count, hit ratio, miss ratio, if you don't specify a cache_size parameter or specify cache_size=-1, it will use the largest possible size automatically.

- With the data calculated from profiler, you can do plotting yourself, or any other calculation. But for your convenience, we have also provided several plotting functions for you to use. For plotting hit ratio curve (HRC):

```
>>> profiler_LRU.plotHRC()
```

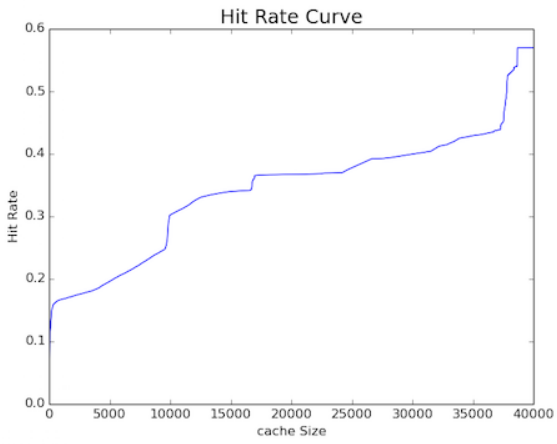


Fig. 1: Hit ratio curve(HRC) of the trace

- Similarly, we can plot miss ratio curve (MRC):

```
>>> profiler_LRU.plotMRC()
```

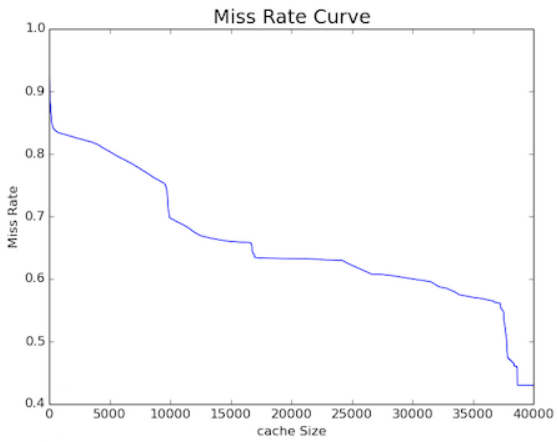


Fig. 2: Miss ratio curve(MRC) of the trace

Warning: Upon testing, using keyword arguments might cause error in some of 32-bit platform, if you get an error, please try not using keyword arguments.

1.5.2 Profiling with non-LRU

Apart from LRU, we have also provided a variety of other cache replacement algorithms for you to play with, including Optimal, FIFO, LRU-2, LRU-K, MRU, LFU, LFU_Fast, Random, SLRU, S4LRU, clock, LinuxClock, TEAR, LightLRU adaptive SLRU.

Note: Check here for detailed information about each cache replacement algorithms.

To play with these cache replacement algorithms, you just substitue 'FIFO' in the examples below with the cache replacement algorithm you want, then give a cache_size and bin_size (how fine you want the profiling; the smaller, the slower). The reason why we need cache_size and bin_size is that for a general cache replacement algorithm, the profiling is done by sampling at certain points among all cache size, in other words, the nth element in numpy arrays returned represents the result at cache size of n*bin_size. Some examples are shown below:

- Obtaining a profiler:

```
>>> profiler_FIFO = c.profiler('FIFO', cache_size=2000, bin_size=100)
```

Several other parameters and their default values are listed below, use_general_profiler is only used when cache replacement algorithm is LRU. The reason why we want to use a general profiler for LRU is that profiling for LRU using a LRU profiler has time complexity of O(NlogN), while general profiler with sampling has time complexity of O(N). They will have a large time difference on big data.

Keyword Argument	Default Value	Required
cache_size	No default value	YES
bin_size	cache_size//100	No
cache_params	None	Depend on algorithm
num_of_threads	num of cores in the machine	No
use_general_profiler	False	No

- After obtaining the profiler, everything else is the same as above with LRUProfiler, you can obtain hit_count, hit_ratio, miss_ratio, you can plotHRC,

The only difference is that the returned hit_count array, hit_ratio array, miss_ratio array do not have the last two special elements as above.

Some examples are shown below: >>> profiler_FIFO.get_hit_count() >>> profiler_FIFO.get_hit_ratio() >>> profiler_FIFO.get_miss_ratio() >>> profiler_FIFO.plotHRC()

Note: Reuse distance related operation is only allowed on LRU, so don't call get_reuse_distance on non-LRU cache replacement algorithms.

Note: If you want to test your own cache replacement algorithms, check *here*.

1.6 Basic Plotting

1.6.1 Basic Plotting

PyMimircache allows you to plot a variety graphs, including basic graphs to help you visualize and understand your data. Before plotting, let's talk about the concept of time, there are two types of time concept in PyMimircache.

The first one is called **virtual time**, which basically is the order in the request sequence, so the first request has virtual time 1, the second request has virtual time 2, etc.

The other is **real time**, which is the wall clock time. It is used only when you have specified the field/column of real_time in the data.

Current supported plot types are listed in the table below

plot type	description
cold_miss	cold miss count VS time, how many cold misses occur every x seconds (real time and unit is the same as the unit in trace file) or every x requests (virtual time)
cold_miss_ratio	similar to cold miss plot, except it shows the ratio of cold miss instead of count of cold miss
request_num	the number of requests per x seconds (real time and unit is the same as the unit in trace file), virtual time does not make sense here
popularity	the popularity curve of the obj in the trace, it shows how many objects have how many hits
scan_vis	obj sorted by first access time and plot the requests against time, this is used to visualize "scan"
rd_distribution	the reuse distribution 2d plot, it shows how many requests have reuse distance of X

To plot these figures, you can invoke twoDPlot function on cachecow obj:

```
>>> c.twoDPlot(plot_type, **kwargs)
```

All optional kwargs all listed in the table below.

arguments	supported plot types	description
time_mode	cold_miss/request_num cold_miss_ratio	time mode used in plotting, "v" for virtual time, "r" for real time.
time_interval	cold_miss/request_num cold_miss_ratio	the time interval/span of each sample.
logX	popularity/rd_distribution	boolean, whether we use log scale X-axis
logY	popularity/rd_distribution	boolean, whether we use log scale Y-axis
cdf	popularity/rd_distribution	boolean, whether we want to plot CDF
partial_ratio	scan_vis	the zoom in ratio, if it is 1, then no zoom in, see below for further explanation.
figname	all plots	the name of the figure

Cold miss plot

- Cold miss plot: the count of cold misses in the given interval.

```
>>> c.twoDPlot('cold_miss', mode="v", time_interval=1000)    # the number of cold␣
↪misses every 1000 requests
```

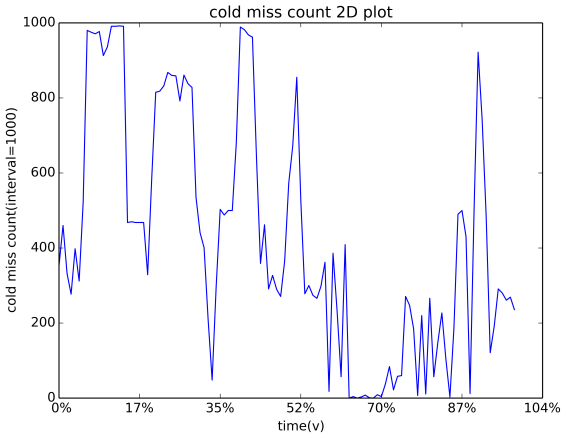


Fig. 3: Cold miss count in virtual time

Request number plot

- Request number plot: the number of requests in the given interval.

```
>>> c.twoDPlot('request_num', mode="r", time_interval=10000)    # the number of␣
↪requests every 10000 seconds
```

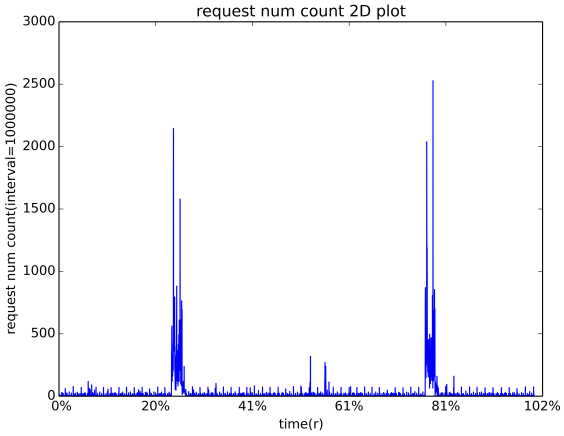


Fig. 4: Request number count in real time

Scan_vis plot

- Scan_vis plot: it renames each obj with sequence 1, 2, 3, . . . N based on first access time, then plot the time of each request thereafter.
- A pixel (X, Y) in the figure means obj Y is requested at virtual time X, a horizontal line y=Y plots the all the requests of a single obj Y.
- The default setting will plot two figures, the first figure plots all the requests with sampling, the second figure takes first partial_ratio (0.1) of all requests and do the same plotting, which can been thought as a zoom in for the first 10% of the trace.
- This plot is very useful when you are dealing with a block-level trace. We can see the scan very easily even if the scan is not scanning consecutive blocks.

```
>>> c.twoDPlot('scan_vis', partial_ratio=0.1)      # mapping plot
```

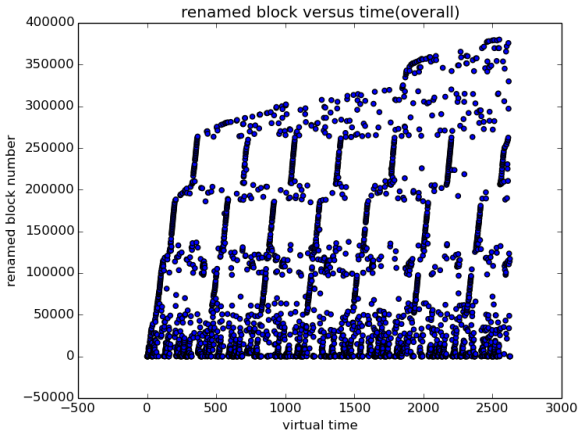


Fig. 5: Mapping plot

1.6.2 Hit Ratio Curve Plotting

To plot hit ratio curve (HRC), you can either get a profiler and plot like the ones shown in *profiling*, or you can use cachecow to directly plot hit ratio curves.

Using cachecow to plot HRC is easy, you just call plotHRC and pass in a list of algorithms you want to plot with:

```
>>> c.plotHRCs(algorithm_list, cache_params=None, cache_size=-1, bin_size=-1,
                auto_size=True, num_of_threads=4, use_general_profiler=False,
                save_gradually=False, figname="HRC.png", **kwargs)
```

A detailed explanation of each arguments can be found in the table below:

arguments	Value type or possile value	description
algorithm_list	a list of algorithm	All supported algorithms can be found here.
cache_params	a list of cache alg parameters	It is a list of the same size of algorithm_list, use *None* if an algorithm does not require parameters. If the list is all *None*, then cache_params is optional.
cache_size	int	The max size of cache, if -1, then cachecow will find the largest one for you automatically.
bin_size	int	cachecow will profiling at bin_size, bin_size*2 ... cache_size, if bin_size=-1, cachecow will use cache_size//100 as default.
auto_size	True/False	Whether cachecow should chnage cache_size to avoid plateau at the end of HRC curve.
num_of_threads	int	Control concurrency in the application, default is 4.
use_general_profiler	True/False	Only Used for LRU profiling, the default profiler is LRUProfiler, which gives high accuracy, but has time complexity of O(NlogN), which can be time consuming on big data, if use_general_profiler=True, then cachecow will use a generalProfiler for LRU as well, which has time complexity of O(N).
save_gradually	True/False	On big data, the hit ratio curve plotting can be very time consuming, save_gradually will save the plot every time when one algorithm is finished.
figname	string	The name of the figure, filename should contain suffix

Example

```
>>> c.plotHRCs(["LRU", "LFUFast", "ARC", "SLRU", "Optimal"],
               cache_params=[None, None, None, {"N":2}, None],
               save_gradually=True)
```

1.7 Heatmap Plotting

1.7.1 Plotting Heatmaps

Another great feature of PyMimircache is that it allows you to incorporate time component into consideration, taking cache analysis from static to dynamic. Currently six types of heatmaps are supported.

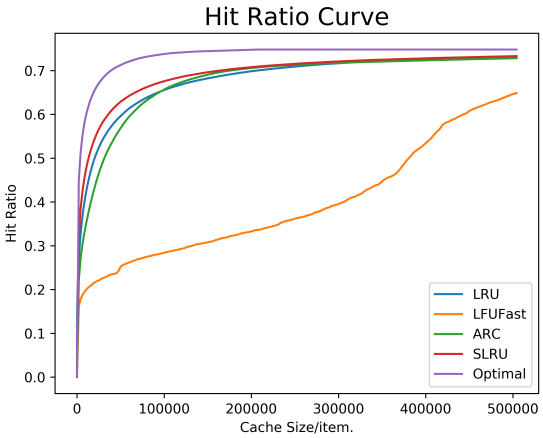


Fig. 6: Hit Ratio Curves

Plot Types

For all the plot types, the X-axis is the time in percent (either real or virtual).

plot type	Y-axis	plot detail
hit_ratio_start_time_end_time	end time (real or virtual) in percent	pixel (x, y) means the hit ratio from time x to time y.
rd_distribution	reuse distance	reuse distance distribution graph, pixel (x, y) represents at time x+time_interval, the number of requests have reuse distance of y (shown in color).
rd_distribution_CDF	reuse distance	similar to reuse distance distribution graph, but each points (x, y) represents the percent of requests have reuse distance less than or equal to y.
future_rd_distribution	future reuse distance	future reuse distance distribution graph, future reuse distance is defined as how far in the future, it will be accessed again.
hit_ratio_start_time_cache_size	cache size	each vertical line x=t is a hit ratio curve of trace starting at t
avg_rd_start_time_end_time	end time (real or virtual) in percent	pixel (x, y) means average reuse distance of requests from time x to time y

How to Plot

Plotting heatmaps is easy, just call the following function on cachecow,

```
>>> c.heatmap(time_mode, plot_type, time_interval=-1, num_of_pixels=-1,
              algorithm="LRU", cache_params=None, cache_size=-1, **kwargs)
```

The first two parameters are mode and plot_type, time mode is either r or v for real time or virtual time, the types of plot(see table above) The following keyword arguments are optional, however, you must provide one of time_interval and number_of_pixels, which controls how fine the graph will look and also determines the amount of computation. The time_interval variable implies the time span of a single pixel in the plot, sometimes it is not easy to estimate time_interval, so instead you can provide the number of pixels you want in one dimension.

Note: If you do not want the computation time to be very long, then specify a big time_interval or small num_of_pixels.

Keyword Arguments	Default Value	Possible Values	Necessary
time_interval	"-1"	a time interval	provide this value or num_of_pixels
num_of_pixels	"-1"	the number of pixels on one dimension	provide this value or time_interval
algorithm	"LRU"	All available cache replacement algorithms	No
cache_params	None	Depends on cache replacement algorithms	Depends on cache replacement algorithms, for example LRU_K
cache_size	-1	Positive integer	**Necessary for plot "hit_ratio_start_time_end_time"**
figname	heatmap.png	Any string, remember to include suffix	No
num_of_threads	4	Positive integer except 0	No

Attention: cache_size is necessary for hit_ratio_start_time_end_time graph.

Plotting Examples

```
>>> c.heatmap('r', "hit_ratio_start_time_end_time", num_of_pixels=200, cache_
↪size=2000, figname="heatmap1.png", num_of_threads=8)
```

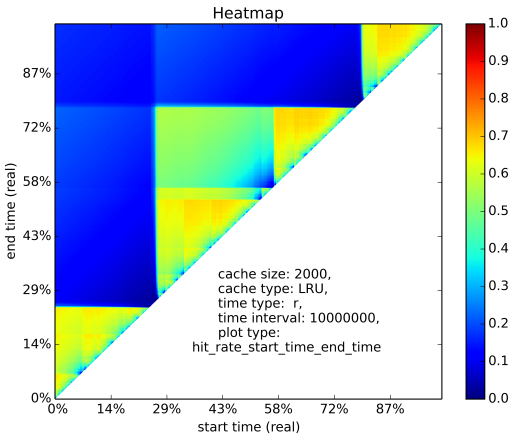


Fig. 7: Hit ratio of varying start time and end time

Another example

```
>>> c.heatmap('r', "rd_distribution", time_interval=10000000)
```

1.7.2 Plotting Differential Heatmaps

Want to know which algorithm is better? Not satisfied with hit ratio curve or miss ratio curve because they only show you the result over the whole trace? You are in the right place! Differential heatmaps allow you to compare cache replacement algorithms with respect to time.

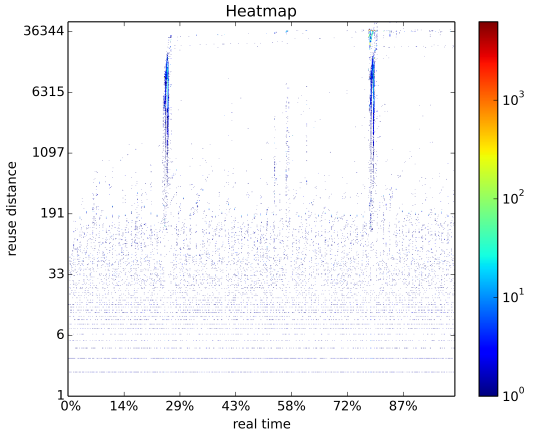


Fig. 8: Reuse distance distribution graph

Currently we only support differential heatmap of hit_ratio_start_time_end_time. Each pixel (x, y) in the plot represents the hit ratio difference between algorithm1 and algorithm2 divided by hit ratio of algorithm 1 from time x to y. The function to plot is shown below:

```
>>> c.diff_heatmap(time_mode, plot_type, algorithm1, time_interval=-1, num_of_pixels=-
↪1,
                   algorithm2="Optimal", cache_params1=None, cache_params2=None,␣
↪cache_size=-1, **kwargs)
```

The arguments here are similar to plotting heatmaps, the only difference is that we have one more algorithm, which is used for comparison,

Example:

```
>>> c.diff_heatmap('r', "hit_ratio_start_time_end_time", time_interval=1000000,␣
↪algorithm1="LRU", cache_size=2000)
```

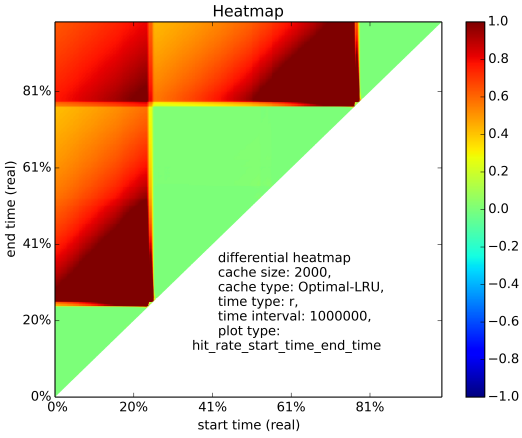


Fig. 9: Differential heatmap, the value of each pixel is (hit_ratio_of_algorithm2 - hit_ratio_of_algorithm1)/hit_ratio_of_algorithm1

Congratulations! You have finished the basic tutorial! Check *Advanced Usage* Part if you need.

1.8 Advanced Usages

1.8.1 PyMimircache and its components

Current version of PyMimircache is composed of three main components.

The first one is cache, which simulates corresponding cache replacement algorithm.

the second one is cacheReader, which provides all the necessary functions for reading and examing trace data file.

Most important of all, the third component is profilers, which extract data for profiling.

Currently, we have three kinds of profilers, the first one is LRU profiler, specially tailored for LRU; the second one is a general profiler for profiling all non-LRU cache replacement algorithms; the third profiler is a heatmap plot engine, which currently supports a variety of heatmaps. LRUProfiler is in C, so it is pretty fast. The other two profilers have corresponding C implementation (cGeneralProfiler and cHeatmap) used for caches available in C.

Each component has more functionality than described in tutorial, read the source code or raise a new issue in github if you want to know more or have questions.

1.8.2 Write your own cacheReader

Writing your own cacheReader is not difficult, just inherit abstractCacheReader.py. Here is an example:

```
from PyMimircache.cacheReader.abstractReader import AbstractReader

class PlainReader(AbstractReader):
    """
    PlainReader class

    """
    all = ["read_one_req", "copy", "get_params"]

    def __init__(self, file_loc, data_type='c', open_c_reader=True, **kwargs):
        """
        :param file_loc:            location of the file
        :param data_type:           type of data, can be "l" for int/long, "c" for␣
↪string
        :param open_c_reader:       bool for whether open reader in C backend
        :param kwargs:              not used now
        """

        super(PlainReader, self).__init__(file_loc, data_type, open_c_reader=open_c_
↪reader, lock=kwargs.get("lock"))
        self.trace_file = open(file_loc, 'rb')
        if ALLOW_C_MIMIRCACHE and open_c_reader:
            self.c_reader = c_cacheReader.setup_reader(file_loc, 'p', data_type=data_
↪type, block_unit_size=0)

    def read_one_req(self):
        """
        read one request
        :return: a request
        """
        super().read_one_req()

        line = self.trace_file.readline().decode()
```

(continues on next page)

(continued from previous page)

```
        while line and len(line.strip()) == 0:
            line = self.trace_file.readline().decode()

        if line and len(line.strip()):
            return line.strip()
        else:
            return None

    def read_complete_req(self):
        """
        read all information about one record, which is the same as read_one_req for␣
↪PlainReader
        """

        return self.read_one_req()

    def skip_n_req(self, n):
        """
        skip N requests from current position

        :param n: the number of requests to skip
        """

        for i in range(n):
            self.read_one_req()


    def copy(self, open_c_reader=False):
        """
        reader a deep copy of current reader with everything reset to initial state,
        the returned reader should not interfere with current reader

        :param open_c_reader: whether open_c_reader_or_not, default not open
        :return: a copied reader
        """

        return PlainReader(self.file_loc, data_type=self.data_type, open_c_
↪reader=open_c_reader, lock=self.lock)

    def get_params(self):
        """
        return all the parameters for this reader instance in a dictionary
        :return: a dictionary containing all parameters
        """

        return {
            "file_loc": self.file_loc,
            "data_type": self.data_type,
            "open_c_reader": self.open_c_reader
        }

    def __next__(self):  # Python 3
        super().__next__()
        element = self.trace_file.readline().strip()
        if element:
            return element
        else:
```

(continues on next page)

(continued from previous page)

```
            raise StopIteration

    def __repr__(self):
        return "PlainReader of trace {}".format(self.file_loc)
```

After writing your own cache reader, you can use it on generalProfiler and heatmap, for example:

```
>>> reader = vscsiCacheReader(PATH/TO/DATA)
>>> p = generalProfiler(reader, "FIFO", cache_size, bin_size=bin_size, num_of_
↪process=8)
```

the first parameter is the cacheReader object of your own, the second is the cache replacement algorithm, the third parameter is cache size, the fourth parameter is bin_size, and it can be omitted, in which case, the default bin_size if cache_size/100.

```
>>> hm = heatmap()
>>> hm.heatmap(reader, 'r', TIME_INTERVAL, "hit_rate_start_time_end_time", cache_
↪size=CACHE_SIZE)
```

1.8.3 Write your own cache replacement algorithm

Writing your own cache in Python is not difficult, just inherit Cache.py:

```
from PyMimircache.cache.abstractCache import Cache

class LRU(Cache):
    """
    LRU class for simulating a LRU cache

    """

    def __init__(self, cache_size, **kwargs):

        super().__init__(cache_size, **kwargs)
        self.cacheline_dict = OrderedDict()

    def has(self, req_id, **kwargs):
        """
        check whether the given id in the cache or not

        :return: whether the given element is in the cache
        """
        if req_id in self.cacheline_dict:
            return True
        else:
            return False

    def _update(self, req_item, **kwargs):
        """ the given element is in the cache,
        now update cache metadata and its content

        :param **kwargs:
        :param req_item:
        :return: None
        """
```

(continues on next page)

(continued from previous page)

```
        req_id = req_item
        if isinstance(req_item, Req):
            req_id = req_item.item_id

        self.cacheline_dict.move_to_end(req_id)

    def _insert(self, req_item, **kwargs):
        """
        the given element is not in the cache, now insert it into cache
        :param **kwargs:
        :param req_item:
        :return: evicted element or None
        """

        req_id = req_item
        if isinstance(req_item, Req):
            req_id = req_item.item_id

        self.cacheline_dict[req_id] = True

    def evict(self, **kwargs):
        """
        evict one cacheline from the cache

        :param **kwargs:
        :return: id of evicted cacheline
        """

        req_id = self.cacheline_dict.popitem(last=False)
        return req_id

    def access(self, req_item, **kwargs):
        """
        request access cache, it updates cache metadata,
        it is the underlying method for both get and put

        :param **kwargs:
        :param req_item: the request from the trace, it can be in the cache, or not
        :return: None
        """

        req_id = req_item
        if isinstance(req_item, Req):
            req_id = req_item.item_id

        if self.has(req_id):
            self._update(req_item)
            return True
        else:
            self._insert(req_item)
            if len(self.cacheline_dict) > self.cache_size:
                self.evict()
            return False

    def __len__(self):
        return len(self.cacheline_dict)
```

(continues on next page)

(continued from previous page)

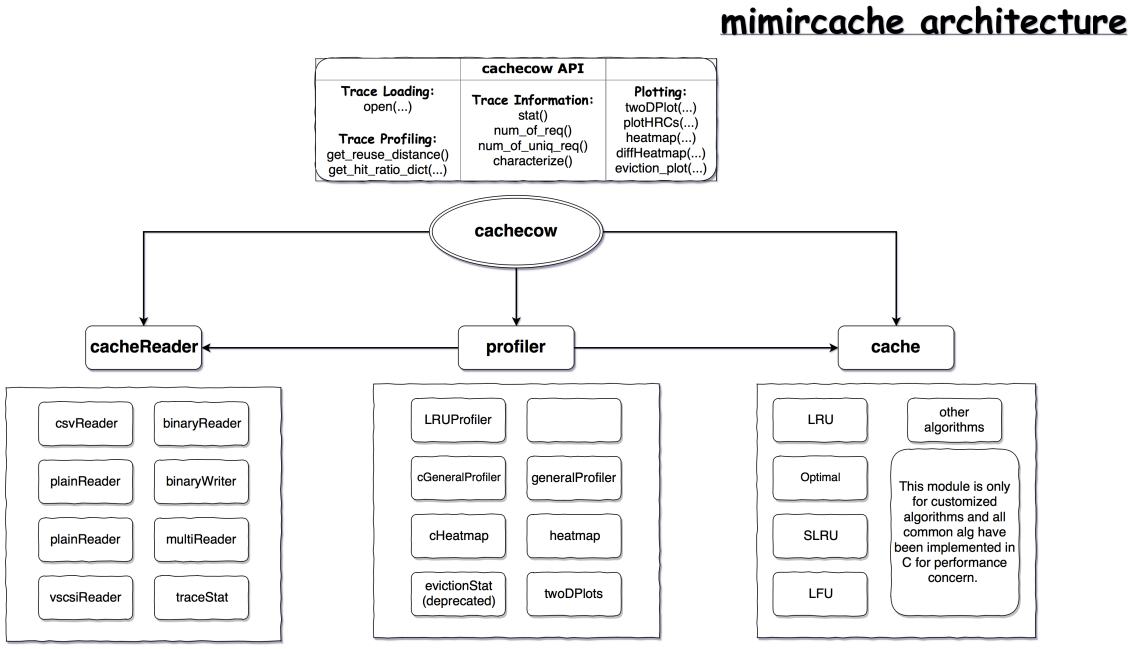
```
    def __repr__(self):
        return "LRU cache of size: {}, current size: {}, {}".\
            format(self.cache_size, len(self.cacheline_dict), super().__repr__())
```

The usage of new cache replacement algorithm is the same as the one in last section, just replace the algorithm string with your algorithm class.

Profiling in python is only applicable on small data set, so you can use it to verify your idea, when running on large dataset, we suggested implemented the algorithms in C, check the source code to find out how to implement in C.

1.9 API

The architecture of mimircache is shown in the diagram below, it contains three parts, profiler, cache and cacheReader, which provides the detailed functions. cachecow is an upper level API that provides most common basic operations. To perform detailed operations, you will need to deal with the three components, so check the API below.



1.9.1 API-cachecow

class PyMimircache.top.cachecow.**Cachecow**(***kwargs*)

cachecow class providing top level API

open(*file_path*, *trace_type='p'*, *data_type='c'*, ***kwargs*)

The default operation of this function opens a plain text trace, the format of a plain text trace is such a file that each line contains a label.

By changing trace type, it can be used for opening other types of trace, supported trace type includes

trace_type	file type	require init_params
"p"	plain text	No
"c"	csv	Yes
"b"	binary	Yes
"v"	vscsi	No

the effect of this is the save as calling corresponding functions (csv, binary, vscsi)

Parameters

- **file_path** – the path to the data
- **trace_type** – type of trace, "p" for plainText, "c" for csv, "v" for vscsi, "b" for binary
- **data_type** – the type of request label, can be either "c" for string or "l" for number (for example block IO LBA)
- **kwargs** – parameters for opening the trace

Returns reader object

csv (*file_path*, *init_params*, *data_type='c'*, *block_unit_size=0*, *disk_sector_size=0*, ***kwargs*)
open a csv trace, init_params is a dictionary specifying the specs of the csv file, the possible keys are listed in the table below. The column/field number begins from 1, so the first column(field) is 1, the second is 2, etc.

Parameters

- **file_path** – the path to the data
- **init_params** – params related to csv file, see above or csvReader for details
- **data_type** – the type of request label, can be either "c" for string or "l" for number (for example block IO LBA)
- **block_unit_size** – the block size for a cache, currently storage system only
- **disk_sector_size** – the disk sector size of input file, storage system only

Returns reader object

Keyword Argument	file type	Value Type	Default Value	Description
label	csv/ binary	int	this is required	the column of the label of the request
fmt	binary	string	this is required	fmt string of binary data, same as python struct
header	csv	True/False	False	whether csv data has header
delimiter	csv	char	","	the delimiter separating fields in the csv file
real_time	csv/ binary	int	NA	the column of real time
op	csv/ binary	int	NA	the column of operation (read/write)
size	csv/ binary	int	NA	the column of block/request size

binary (*file_path*, *init_params*, *data_type='l'*, *block_unit_size=0*, *disk_sector_size=0*, ***kwargs*)
open a binary trace file, init_params see function csv

Parameters

- **file_path** – the path to the data
- **init_params** – params related to the spec of data, see above csv for details
- **data_type** – the type of request label, can be either "c" for string or "l" for number (for example block IO LBA)
- **block_unit_size** – the block size for a cache, currently storage system only
- **disk_sector_size** – the disk sector size of input file, storage system only

Returns reader object

vscsi (*file_path*, *block_unit_size=0*, ***kwargs*)
open vscsi trace file

Parameters

- **file_path** – the path to the data
- **block_unit_size** – the block size for a cache, currently storage system only

Returns reader object

reset ()

reset cachecow to initial state, including reset reader to the beginning of the trace

close ()
close the reader opened in cachecow, and clean up in the future

stat (*time_period=[-1, 0]*)
obtain the statistical information about the trace, including

- number of requests
- number of uniq items
- cold miss ratio
- a list of top 10 popular in form of (obj, num of requests):
- number of obj/block accessed only once
- frequency mean
- time span

Returns a string of the information above

get_frequency_access_list (*time_period=[-1, 0]*)
obtain the statistical information about the trace, including

- number of requests
- number of uniq items
- cold miss ratio
- a list of top 10 popular in form of (obj, num of requests):
- number of obj/block accessed only once
- frequency mean
- time span

Returns a string of the information above

num_of_req()

Returns the number of requests in the trace

num_of_uniq_req()

Returns the number of unique requests in the trace

get_reuse_distance()

Returns an array of reuse distance

get_hit_count_dict(*algorithm*, *cache_size=-1*, *cache_params=None*, *bin_size=-1*, *use_general_profiler=False*, ***kwargs*)

get hit count of the given algorithm and return a dict of mapping from cache size -> hit count notice that hit count array is not CDF, meaning hit count of size 2 does not include hit count of size 1, you need to sum up to get a CDF.

Parameters

- **algorithm** – cache replacement algorithms
- **cache_size** – size of cache
- **cache_params** – parameters passed to cache, some of the cache replacement algorithms require parameters, for example LRU-K, SLRU
- **bin_size** – if algorithm is not LRU, then the hit ratio will be calculated by simulating cache at cache size [0, bin_size, bin_size*2 ... cache_size], this is not required for LRU
- **use_general_profiler** – if algorithm is LRU and you don't want to use LRUProfiler, then set this to True, possible reason for not using a LRUProfiler: 1. LRUProfiler is too slow for your large trace because the algorithm is O(NlogN) and it uses single thread; 2. LRUProfiler has a bug (let me know if you found a bug).
- **kwargs** – other parameters including num_of_threads

Returns an dict of hit ratio of given algorithms, mapping from cache_size -> hit ratio

get_hit_ratio_dict(*algorithm*, *cache_size=-1*, *cache_params=None*, *bin_size=-1*, *use_general_profiler=False*, ***kwargs*)

get hit ratio of the given algorithm and return a dict of mapping from cache size -> hit ratio

Parameters

- **algorithm** – cache replacement algorithms
- **cache_size** – size of cache
- **cache_params** – parameters passed to cache, some of the cache replacement algorithms require parameters, for example LRU-K, SLRU
- **bin_size** – if algorithm is not LRU, then the hit ratio will be calculated by simulating cache at cache size [0, bin_size, bin_size*2 ... cache_size], this is not required for LRU
- **use_general_profiler** – if algorithm is LRU and you don't want to use LRUProfiler, then set this to True, possible reason for not using a LRUProfiler: 1. LRUProfiler is too slow for your large trace because the algorithm is O(NlogN) and it uses single thread; 2. LRUProfiler has a bug (let me know if you found a bug).
- **kwargs** – other parameters including num_of_threads

Returns an dict of hit ratio of given algorithms, mapping from cache_size -> hit ratio

profiler(*algorithm*, *cache_params=None*, *cache_size=-1*, *bin_size=-1*, *use_general_profiler=False*, ***kwargs*)

get a profiler instance, this should not be used by most users

Parameters

- **algorithm** – name of algorithm
- **cache_params** – parameters of given cache replacement algorithm
- **cache_size** – size of cache
- **bin_size** – bin_size for generalProfiler
- **use_general_profiler** – this option is for LRU only, if it is True, then return a cGeneralProfiler for LRU, otherwise, return a LRUProfiler for LRU.

  Note: LRUProfiler does not require cache_size/bin_size params, it does not sample thus provides a smooth curve, however, it is O(logN) at each step, in constrast, cGeneralProfiler samples the curve, but use O(1) at each step
- **kwargs** – num_of_threads

Returns a profiler instance

heatmap(*time_mode*, *plot_type*, *time_interval=-1*, *num_of_pixels=-1*, *algorithm='LRU'*, *cache_params=None*, *cache_size=-1*, ***kwargs*)

plot heatmaps, currently supports the following heatmaps

- hit_ratio_start_time_end_time
- hit_ratio_start_time_cache_size (python only)
- avg_rd_start_time_end_time (python only)
- cold_miss_count_start_time_end_time (python only)
- rd_distribution
- rd_distribution_CDF
- future_rd_distribution
- dist_distribution
- reuse_time_distribution

Parameters

- **time_mode** – the type of time, can be "v" for virtual time, or "r" for real time
- **plot_type** – the name of plot types, see above for plot types
- **time_interval** – the time interval of one pixel
- **num_of_pixels** – if you don't to use time_interval, you can also specify how many pixels you want in one dimension, note this feature is not well tested
- **algorithm** – what algorithm to use for plotting heatmap, this is not required for distance related heatmap like rd_distribution
- **cache_params** – parameters passed to cache, some of the cache replacement algorithms require parameters, for example LRU-K, SLRU
- **cache_size** – The size of cache, this is required only for *hit_ratio_start_time_end_time*
- **kwargs** – other parameters for computation and plotting such as num_of_threads, figname

diff_heatmap(*time_mode*, *plot_type*, *algorithm1='LRU'*, *time_interval=-1*, *num_of_pixels=-1*, *algorithm2='Optimal'*, *cache_params1=None*, *cache_params2=None*, *cache_size=-1*, ***kwargs*)

Plot the differential heatmap between two algorithms by alg2 - alg1

Parameters

- **cache_size** – size of cache
- **time_mode** – time time_mode "v" for virutal time, "r" for real time
- **plot_type** – same as the name in heatmap function
- **algorithm1** – name of the first alg
- **time_interval** – same as in heatmap
- **num_of_pixels** – same as in heatmap
- **algorithm2** – name of the second algorithm
- **cache_params1** – parameters of the first algorithm
- **cache_params2** – parameters of the second algorithm
- **kwargs** – include num_of_threads

twoDPlot(*plot_type*, ***kwargs*)

an aggregate function for all two dimenional plots printing except hit ratio curve

plot type	required parameters	Description
cold_miss_count	time_mode, time_interval	cold miss count VS time
cold_miss_ratio	time_mode, time_interval	cold miss ratio VS time
request_rate	time_mode, time_interval	num of requests VS time
popularity	NA	Percentage of obj VS frequency
rd_distribution	NA	Num of req VS reuse distance
rt_distribution	NA	Num of req VS reuse time
scan_vis_2d	NA	mapping from original objID to sequential number
interval_hit_ratio	cache_size	hit ratio of interval VS time
request_traffic_vol		
obj_size_distribution		

Parameters

- **plot_type** – type of the plot, see above
- **kwargs** – paramters related to plots, see twoDPlots module for detailed control over plots

plotHRCs(*algorithm_list*, *cache_params=()*, *cache_size=-1*, *bin_size=-1*, *auto_resize=True*, *figname='HRC.png'*, ***kwargs*)

this function provides hit ratio curve plotting

Parameters

- **algorithm_list** – a list of algorithm(s)
- **cache_params** – the corresponding cache params for the algorithms, use None for algorithms that don't require cache params, if none of the alg requires cache params, you don't need to set this

- **cache_size** – maximal size of cache, use -1 for max possible size
- **bin_size** – bin size for non-LRU profiling
- **auto_resize** – when using max possible size or specified cache size too large, you will get a huge plateau at the end of hit ratio curve, set auto_resize to True to cutoff most of the big plateau
- **figname** – name of figure
- **kwargs** – options: block_unit_size, num_of_threads, auto_resize_threshold, xlimit, ylimit, cache_unit_size

  save_gradually - save a figure everytime computation for one algorithm finishes,

  label - instead of using algorithm list as label, specify user-defined label

plotMRCs(*algorithm_list*, *cache_params=()*, *cache_size=-1*, *bin_size=-1*, *figname='MRC.png'*, ***kwargs*)

this function provides miss ratio curve plotting

Parameters

- **algorithm_list** – a list of algorithm(s)
- **cache_params** – the corresponding cache params for the algorithms, use None for algorithms that don't require cache params, if none of the alg requires cache params, you don't need to set this
- **cache_size** – maximal size of cache, use -1 for max possible size
- **bin_size** – bin size for non-LRU profiling
- **auto_resize** – when using max possible size or specified cache size too large, you will get a huge plateau at the end of hit ratio curve, set auto_resize to True to cutoff most of the big plateau
- **figname** – name of figure
- **kwargs** – options: block_unit_size, num_of_threads, auto_resize_threshold, xlimit, ylimit, cache_unit_size

  save_gradually - save a figure everytime computation for one algorithm finishes,

  label - instead of using algorithm list as label, specify user-defined label

characterize(*characterize_type*, *cache_size=-1*, ***kwargs*)

use this function to obtain a series of plots about your trace, the type includes

- short - short run time, fewer plots with less accuracy
- medium
- long
- all - most of the available plots with high accuracy, notice it can take **LONG** time on big trace

Parameters

- **characterize_type** – see above, options: short, medium, long, all
- **cache_size** – estimated cache size for the trace, if -1, PyMimircache will estimate the cache size
- **kwargs** – print_stat

Returns trace stat string

class PyMimircache.top.cachecow.**Cachecow**(***kwargs*)

cachecow class providing top level API

open(*file_path*, *trace_type='p'*, *data_type='c'*, ***kwargs*)

The default operation of this function opens a plain text trace, the format of a plain text trace is such a file that each line contains a label.

By changing trace type, it can be used for opening other types of trace, supported trace type includes

trace_type	file type	require init_params
"p"	plain text	No
"c"	csv	Yes
"b"	binary	Yes
"v"	vscsi	No

the effect of this is the save as calling corresponding functions (csv, binary, vscsi)

Parameters

- **file_path** – the path to the data
- **trace_type** – type of trace, "p" for plainText, "c" for csv, "v" for vscsi, "b" for binary
- **data_type** – the type of request label, can be either "c" for string or "l" for number (for example block IO LBA)
- **kwargs** – parameters for opening the trace

Returns reader object

csv(*file_path*, *init_params*, *data_type='c'*, *block_unit_size=0*, *disk_sector_size=0*, ***kwargs*)

open a csv trace, init_params is a dictionary specifying the specs of the csv file, the possible keys are listed in the table below. The column/field number begins from 1, so the first column(field) is 1, the second is 2, etc.

Parameters

- **file_path** – the path to the data
- **init_params** – params related to csv file, see above or csvReader for details
- **data_type** – the type of request label, can be either "c" for string or "l" for number (for example block IO LBA)
- **block_unit_size** – the block size for a cache, currently storage system only
- **disk_sector_size** – the disk sector size of input file, storage system only

Returns reader object

Keyword Argument	file type	Value Type	Default Value	Description
label	csv/ binary	int	this is required	the column of the label of the request
fmt	binary	string	this is required	fmt string of binary data, same as python struct
header	csv	True/False	False	whether csv data has header
delimiter	csv	char	","	the delimiter separating fields in the csv file
real_time	csv/ binary	int	NA	the column of real time
op	csv/ binary	int	NA	the column of operation (read/write)
size	csv/ binary	int	NA	the column of block/request size

binary (*file_path*, *init_params*, *data_type='l'*, *block_unit_size=0*, *disk_sector_size=0*, ***kwargs*)
open a binary trace file, init_params see function csv

Parameters

- **file_path** – the path to the data
- **init_params** – params related to the spec of data, see above csv for details
- **data_type** – the type of request label, can be either "c" for string or "l" for number (for example block IO LBA)
- **block_unit_size** – the block size for a cache, currently storage system only
- **disk_sector_size** – the disk sector size of input file, storage system only

Returns reader object

vscsi (*file_path*, *block_unit_size=0*, ***kwargs*)
open vscsi trace file

Parameters

- **file_path** – the path to the data
- **block_unit_size** – the block size for a cache, currently storage system only

Returns reader object

reset ()

reset cachecow to initial state, including reset reader to the beginning of the trace

close ()
close the reader opened in cachecow, and clean up in the future

stat (*time_period=[-1, 0]*)
obtain the statistical information about the trace, including

- number of requests
- number of uniq items
- cold miss ratio
- a list of top 10 popular in form of (obj, num of requests):

- number of obj/block accessed only once
- frequency mean
- time span

Returns a string of the information above

get_frequency_access_list(*time_period=[-1, 0]*)
obtain the statistical information about the trace, including

- number of requests
- number of uniq items
- cold miss ratio
- a list of top 10 popular in form of (obj, num of requests):
- number of obj/block accessed only once
- frequency mean
- time span

Returns a string of the information above

num_of_req()

Returns the number of requests in the trace

num_of_uniq_req()

Returns the number of unique requests in the trace

get_reuse_distance()

Returns an array of reuse distance

get_hit_count_dict(*algorithm*, *cache_size=-1*, *cache_params=None*, *bin_size=-1*, *use_general_profiler=False*, ***kwargs*)
get hit count of the given algorithm and return a dict of mapping from cache size -> hit count notice that hit count array is not CDF, meaning hit count of size 2 does not include hit count of size 1, you need to sum up to get a CDF.

Parameters

- **algorithm** – cache replacement algorithms
- **cache_size** – size of cache
- **cache_params** – parameters passed to cache, some of the cache replacement algorithms require parameters, for example LRU-K, SLRU
- **bin_size** – if algorithm is not LRU, then the hit ratio will be calculated by simulating cache at cache size [0, bin_size, bin_size*2 . . . cache_size], this is not required for LRU
- **use_general_profiler** – if algorithm is LRU and you don't want to use LRUProfiler, then set this to True, possible reason for not using a LRUProfiler: 1. LRUProfiler is too slow for your large trace because the algorithm is O(NlogN) and it uses single thread; 2. LRUProfiler has a bug (let me know if you found a bug).
- **kwargs** – other parameters including num_of_threads

Returns an dict of hit ratio of given algorithms, mapping from cache_size -> hit ratio

get_hit_ratio_dict(*algorithm*, *cache_size=-1*, *cache_params=None*, *bin_size=-1*, *use_general_profiler=False*, ***kwargs*)
get hit ratio of the given algorithm and return a dict of mapping from cache size -> hit ratio

Parameters

- **algorithm** – cache replacement algorithms
- **cache_size** – size of cache
- **cache_params** – parameters passed to cache, some of the cache replacement algorithms require parameters, for example LRU-K, SLRU
- **bin_size** – if algorithm is not LRU, then the hit ratio will be calculated by simulating cache at cache size [0, bin_size, bin_size*2 ... cache_size], this is not required for LRU
- **use_general_profiler** – if algorithm is LRU and you don't want to use LRUProfiler, then set this to True, possible reason for not using a LRUProfiler: 1. LRUProfiler is too slow for your large trace because the algorithm is O(NlogN) and it uses single thread; 2. LRUProfiler has a bug (let me know if you found a bug).
- **kwargs** – other parameters including num_of_threads

Returns an dict of hit ratio of given algorithms, mapping from cache_size -> hit ratio

profiler(*algorithm*, *cache_params=None*, *cache_size=-1*, *bin_size=-1*, *use_general_profiler=False*, ***kwargs*)
get a profiler instance, this should not be used by most users

Parameters

- **algorithm** – name of algorithm
- **cache_params** – parameters of given cache replacement algorithm
- **cache_size** – size of cache
- **bin_size** – bin_size for generalProfiler
- **use_general_profiler** – this option is for LRU only, if it is True, then return a cGeneralProfiler for LRU, otherwise, return a LRUProfiler for LRU.

  Note: LRUProfiler does not require cache_size/bin_size params, it does not sample thus provides a smooth curve, however, it is O(logN) at each step, in constrast, cGeneralProfiler samples the curve, but use O(1) at each step
- **kwargs** – num_of_threads

Returns a profiler instance

heatmap(*time_mode*, *plot_type*, *time_interval=-1*, *num_of_pixels=-1*, *algorithm='LRU'*, *cache_params=None*, *cache_size=-1*, ***kwargs*)
plot heatmaps, currently supports the following heatmaps

- hit_ratio_start_time_end_time
- hit_ratio_start_time_cache_size (python only)
- avg_rd_start_time_end_time (python only)
- cold_miss_count_start_time_end_time (python only)
- rd_distribution
- rd_distribution_CDF
- future_rd_distribution

- dist_distribution
- reuse_time_distribution

Parameters

- **time_mode** – the type of time, can be "v" for virtual time, or "r" for real time
- **plot_type** – the name of plot types, see above for plot types
- **time_interval** – the time interval of one pixel
- **num_of_pixels** – if you don't to use time_interval, you can also specify how many pixels you want in one dimension, note this feature is not well tested
- **algorithm** – what algorithm to use for plotting heatmap, this is not required for distance related heatmap like rd_distribution
- **cache_params** – parameters passed to cache, some of the cache replacement algorithms require parameters, for example LRU-K, SLRU
- **cache_size** – The size of cache, this is required only for *hit_ratio_start_time_end_time*
- **kwargs** – other parameters for computation and plotting such as num_of_threads, figname

diff_heatmap(*time_mode*, *plot_type*, *algorithm1='LRU'*, *time_interval=-1*, *num_of_pixels=-1*, *algorithm2='Optimal'*, *cache_params1=None*, *cache_params2=None*, *cache_size=-1*, ***kwargs*)

Plot the differential heatmap between two algorithms by alg2 - alg1

Parameters

- **cache_size** – size of cache
- **time_mode** – time time_mode "v" for virutal time, "r" for real time
- **plot_type** – same as the name in heatmap function
- **algorithm1** – name of the first alg
- **time_interval** – same as in heatmap
- **num_of_pixels** – same as in heatmap
- **algorithm2** – name of the second algorithm
- **cache_params1** – parameters of the first algorithm
- **cache_params2** – parameters of the second algorithm
- **kwargs** – include num_of_threads

twoDPlot(*plot_type*, ***kwargs*)

an aggregate function for all two dimenional plots printing except hit ratio curve

plot type	required parameters	Description
cold_miss_count	time_mode, time_interval	cold miss count VS time
cold_miss_ratio	time_mode, time_interval	cold miss ratio VS time
request_rate	time_mode, time_interval	num of requests VS time
popularity	NA	Percentage of obj VS frequency
rd_distribution	NA	Num of req VS reuse distance
rt_distribution	NA	Num of req VS reuse time
scan_vis_2d	NA	mapping from original objID to sequential number
interval_hit_ratio	cache_size	hit ratio of interval VS time
request_traffic_vol		
obj_size_distribution		

Parameters

- **plot_type** – type of the plot, see above
- **kwargs** – paramters related to plots, see twoDPlots module for detailed control over plots

plotHRCs(*algorithm_list*, *cache_params=()*, *cache_size=-1*, *bin_size=-1*, *auto_resize=True*, *figname='HRC.png'*, ***kwargs*)

this function provides hit ratio curve plotting

Parameters

- **algorithm_list** – a list of algorithm(s)
- **cache_params** – the corresponding cache params for the algorithms, use None for algorithms that don't require cache params, if none of the alg requires cache params, you don't need to set this
- **cache_size** – maximal size of cache, use -1 for max possible size
- **bin_size** – bin size for non-LRU profiling
- **auto_resize** – when using max possible size or specified cache size too large, you will get a huge plateau at the end of hit ratio curve, set auto_resize to True to cutoff most of the big plateau
- **figname** – name of figure
- **kwargs** – options: block_unit_size, num_of_threads, auto_resize_threshold, xlimit, ylimit, cache_unit_size

  save_gradually - save a figure everytime computation for one algorithm finishes,

  label - instead of using algorithm list as label, specify user-defined label

plotMRCs(*algorithm_list*, *cache_params=()*, *cache_size=-1*, *bin_size=-1*, *figname='MRC.png'*, ***kwargs*)

this function provides miss ratio curve plotting

Parameters

- **algorithm_list** – a list of algorithm(s)
- **cache_params** – the corresponding cache params for the algorithms, use None for algorithms that don't require cache params, if none of the alg requires cache params, you don't need to set this

- **cache_size** – maximal size of cache, use -1 for max possible size
- **bin_size** – bin size for non-LRU profiling
- **auto_resize** – when using max possible size or specified cache size too large, you will get a huge plateau at the end of hit ratio curve, set auto_resize to True to cutoff most of the big plateau
- **figname** – name of figure
- **kwargs** – options: block_unit_size, num_of_threads, auto_resize_threshold, xlimit, ylimit, cache_unit_size

  save_gradually - save a figure everytime computation for one algorithm finishes,

  label - instead of using algorithm list as label, specify user-defined label

characterize(*characterize_type*, *cache_size=-1*, ***kwargs*)

use this function to obtain a series of plots about your trace, the type includes

- short - short run time, fewer plots with less accuracy
- medium
- long
- all - most of the available plots with high accuracy, notice it can take **LONG** time on big trace

Parameters

- **characterize_type** – see above, options: short, medium, long, all
- **cache_size** – estimated cache size for the trace, if -1, PyMimircache will estimate the cache size
- **kwargs** – print_stat

Returns trace stat string

1.9.2 API-profiler

1.9.3 API-cLRUProfiler

1.9.4 API-profiler

1.9.5 API-cHeatmap

class PyMimircache.profiler.cHeatmap.**CHeatmap**(***kwargs*)

heatmap class for plotting heatmaps in C

static get_breakpoints(*reader*, *time_mode*, *time_interval=-1*, *num_of_pixel_of_time_dim=-1*, ***kwargs*)

retrieve the breakpoints given time_mode and time_interval or num_of_pixel_of_time_dim, break point breaks the trace into chunks of given time_interval

Parameters

- **reader** – reader for reading trace
- **time_mode** – either real time (r) or virtual time (v)
- **time_interval** – the intended time_interval of data chunk

- **num_of_pixel_of_time_dim** – the number of chunks, this is used when it is hard to estimate time_interval, you only need specify one, either num_of_pixel_of_time_dim or time_interval
- **kwargs** – not used now

Returns a numpy list of break points begin with 0, ends with total_num_requests

heatmap(*reader*, *time_mode*, *plot_type*, *algorithm='LRU'*, *time_interval=-1*, *num_of_pixel_of_time_dim=-1*, *cache_params=None*, ***kwargs*)

This functions provides different types of heatmap plotting

Parameters

- **reader** – the reader instance for data input
- **time_mode** – either real time (r) or virtual time (v), real time is wall clock time, it needs the reader containing real time info virtual time is the reference number, aka. the number of requests
- **plot_type** – different types of heatmap, supported heatmaps are listed in the table below
- **algorithm** – cache replacement algorithm (default: LRU)
- **time_interval** – the time interval of each pixel
- **num_of_pixel_of_time_dim** – if don't want to specify time_interval, you can also specify how many pixels you want
- **cache_params** – params used in cache
- **kwargs** – include num_of_threads, figname, enable_ihr, ema_coef(default: 0.8), info_on_fig

Returns

plot_type	required parameters	descriptions
"hr_st_et"	cache_size	hit ratio with regarding to start time (x) and end time (y)
"hr_st_size"	NOT IMPLEMENTED	hit ratio with reagarding to start time (x) and size (y)
"avg_rd_st_et"	NOT IMPLEMENTED	average reuse distance with regaarding to start time (x) and end time (y)
"rd_distribution"	N/A	reuse distance distribution (y) vs time (x)
"rd_distribution_CDF"	N/A	reuse distance distribution CDF (y) vs time (x)
"future_rd_distribution"	N/A	future reuse distance distribution (y) vs time (x)
"dist_distribution"	N/A	absolute distance distribution (y) vs time (x)
"rt_distribution"	N/A	reuse time distribution (y) vs time (x)

diff_heatmap(*reader*, *time_mode*, *plot_type*, *algorithm1*, *time_interval=-1*, *num_of_pixel_of_time_dim=-1*, *algorithm2='Optimal'*, *cache_params1=None*, *cache_params2=None*, ***kwargs*)

Parameters

- **time_interval** –
- **num_of_pixel_of_time_dim** –
- **algorithm2** –

- **cache_params1** –
- **cache_params2** –
- **algorithm1** –
- **plot_type** –
- **time_mode** –
- **reader** –
- **kwargs** – include num_of_process, figname

Returns

1.9.6 API-pyHeatmap

1.9.7 API-pyGeneralProfiler

1.9.8 API-twoDPlots

PyMimircache.profiler.twoDPlots.**request_rate_2d**(*reader*, *time_mode*, *time_interval*, *figname='request_rate.png'*, ***kwargs*)

plot the number of requests per time_interval vs time :param reader: :param time_mode: either 'r' or 'v' for real time(wall-clock time) or virtual time(reference time) :param time_interval: :param figname: :return: the list of data points

PyMimircache.profiler.twoDPlots.**request_traffic_vol_2d**(*reader*, *time_mode*, *time_interval*, *size_col*, *figname='request_traffic_vol.png'*, ***kwargs*)

plot the the request traffic volume (number of bytes) per time_interval vs time

Parameters

- **reader** –
- **time_mode** – either 'r' or 'v' for real time(wall-clock time) or virtual time(reference time)
- **time_interval** –
- **figname** –

Returns the list of data points

PyMimircache.profiler.twoDPlots.**cold_miss_count_2d**(*reader*, *time_mode*, *time_interval*, *figname='cold_miss_count2d.png'*, ***kwargs*)

plot the number of cold miss per time_interval :param reader: :param time_mode: either 'r' or 'v' for real time(wall-clock time) or virtual time(reference time) :param time_interval: :param figname: :return: the list of data points

PyMimircache.profiler.twoDPlots.**cold_miss_ratio_2d**(*reader*, *time_mode*, *time_interval*, *figname='cold_miss_ratio2d.png'*, ***kwargs*)

plot the percent of cold miss per time_interval :param reader: :param time_mode: either 'r' or 'v' for real time(wall-clock time) or virtual time(reference time) :param time_interval: :param figname: :return: the list of data points

PyMimircache.profiler.twoDPlots.**scan_vis_2d**(*reader*, *partial_ratio=0.1*, *figname=None*, ***kwargs*)

rename all the ojbID for items in the trace for visualization of trace so the first obj is renamed to 1, the second obj is renamed to 2, etc. Notice that it is not first request, second request...

Parameters

- **reader** –
- **partial_ratio** – take fitst partial_ratio of trace for zooming in
- **figname** –

Returns

PyMimircache.profiler.twoDPlots.**popularity_2d**(*reader*, *logX=True*, *logY=False*, *cdf=True*, *plot_type='all'*, *figname='freq_distribution_2d.png'*, ***kwargs*)

plot the popularity curve of the obj in the trace X axis is object frequency, Y axis is either obj percentage or request percentage depending on plot_type

Parameters

- **reader** –
- **logX** –
- **logY** –
- **cdf** –
- **plot_type** –
- **figname** –

Returns the list of data points

PyMimircache.profiler.twoDPlots.**rd_freq_popularity_2d**(*reader*, *logX=True*, *logY=True*, *cdf=False*, *figname='rdFreq_popularity_2d.png'*, ***kwargs*)

plot the reuse distance distribution in a two dimensional figure, X axis is reuse distance frequency Y axis is the number of requests in percentage I don't know why we need this plot

Parameters

- **reader** –
- **logX** –
- **logY** –
- **cdf** –
- **figname** –

Returns the list of data points

PyMimircache.profiler.twoDPlots.**rd_distribution_2d**(*reader*, *logX=True*, *logY=False*, *cdf=True*, *figname='rd_popularity_2d.png'*, ***kwargs*)

plot the reuse distance distribution in two dimension, cold miss is ignored X axis is reuse distance Y axis is number of requests (not in percentage) :param reader: :param logX: :param logY: :param cdf: :param figname: :return: the list of data points

PyMimircache.profiler.twoDPlots.**rt_distribution_2d**(*reader*, *granularity=10*, *logX=True*, *logY=False*, *cdf=True*, *figname='rt_popularity_2d.png'*, ***kwargs*)

plot the reuse time distribution in the trace X axis is reuse time, Y axis is number of requests (not in percentage)

Parameters

- **reader** –
- **granularity** –
- **logX** –
- **logY** –
- **cdf** –
- **figname** –
- **kwargs** – time_bin

Returns the list of data points

PyMimircache.profiler.twoDPlots.**obj_size_distribution_2d**(*reader*, *logX=True*, *logY=False*, *cdf=True*, *plot_type='all'*, *figname='size_distribution_2d.png'*, *size_col=-1*, *log_base=1.0002*, ***kwargs*)

plot the popularity curve of the obj in the trace X axis is object frequency, Y axis is either obj percentage or request percentage depending on plot_type

Parameters

- **reader** –
- **logX** –
- **logY** –
- **cdf** –
- **plot_type** –
- **figname** –

Returns the list of data points

PyMimircache.profiler.twoDPlots.**interval_hit_ratio_2d**(*reader*, *cache_size*, *decay_coef=0.8*, *time_mode='v'*, *time_interval=10000*, *figname='IHRC_2d.png'*, ***kwargs*)

The hit ratio curve over time interval, each pixel in the plot represents the exponential weight moving average (ewma) of hit ratio of the interval

Parameters

- **reader** –
- **cache_size** –

- **decay_coef** – used in ewma
- **time_mode** –
- **time_interval** –
- **figname** –

Returns the list of data points

PyMimircache.profiler.twoDPlots.**freq_distribution_2d**(*reader*, *logX=True*, *logY=False*, *cdf=True*, *plot_type='all'*, *figname='freq_distribution_2d.png'*, ***kwargs*)

plot the popularity curve of the obj in the trace X axis is object frequency, Y axis is either obj percentage or request percentage depending on plot_type

Parameters

- **reader** –
- **logX** –
- **logY** –
- **cdf** –
- **plot_type** –
- **figname** –

Returns the list of data points

1.9.9 API-cache

1.9.10 API-cacheReader

CHAPTER 2

Supported Features

- Cache replacement algorithms simulation.
- trace visualization.
- A variety of cache replacement algorithms, including LRU, LFU, MRU, FIFO, Clock, LinuxClock, TEAR, Random, ARC, SLRU, Optimal and etc.
- Hit/miss ratio curve (HRC/MRC) plotting.
- Efficient reuse distance calculation for LRU.
- Heatmap plotting for visualizing cache dynamics.
- Reuse distance distribution plotting.
- Cache replacement algorithm comparison.

CHAPTER 3

Customization

Now you can customize PyMimircache to fit your own need. You can write

- your own cache reader for reading your special cache trace files.
- your own cache replacement algorithms.
- a middleware for sampling your cache traces for analysis.

CHAPTER 4

Indices and tables

- search

Python Module Index

p

Index

R

S

T

V