MRC: Parallel Cache Simulation for Miss-Ratio Curves

Parallel Functional Programming | Fall 2021 | Final Project Report

Jeffrey Tao
jat2164

Kaylee Treviño
kt2846

Abstract

In this project, we implemented a cache simulator, a variety of storage workloads, and four cache eviction policies. We parallelize the execution of these cache simulations at different cache sizes to generate points along miss-ratio curves.

Background

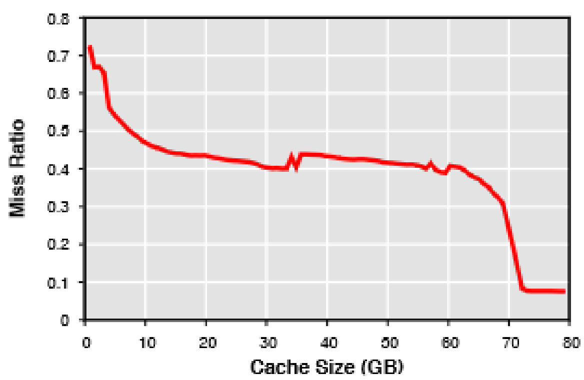

Figure 1: A Miss Ratio Curve [1].

Miss ratio may fluctuate in unexpected ways for a given cache algorithm and workload across cache sizes. Cache misses invariably translate to higher latency for data-dependent operations. To this end, Miss-Ratio Curves (MRCs) are a useful tool for profiling how a given cache eviction policy performs on a given workload, allowing a system designer to pick an appropriate cache eviction policy and cache size. The miss ratio is expressed as:

$$\frac{\#\ of\ misses}{\#\ of\ total\ operations}$$

A MRC can be produced by running the workload on a simulated cache. As the simulation runs, it simply tracks whether each cache operation is a hit or a miss, and computes the miss ratio after it finishes simulating the workload. This produces one point on the MRC. The full MRC is produced by doing this simulation at every cache size between a lower and upper bound, usually from 0 (100% miss rate) to the total size of the data touched by the workload (0% miss rate).

Implementation

CLI

We implement a rudimentary command-line interface to make testing of different workload configurations easier.

```
stack exec – mrc-exe <serial | parallel> <workloads> <cache sizes>
```

The first positional parameter changes whether the graphs are generated serially or in parallel. The second positional parameter is a comma-separated list of workloads to run (uniform, skewed, arc). The third positional parameter is a comma-separated list of cache sizes, interpreted as 2^n (e.g. "3,4,5" will run at cache sizes 8, 16, 32).

Plots are produced by running the python script on the two lines of output produced by mrc-exe.

```
python plot_mrc outFile
```

Workloads

To simplify, we opted to treat accesses as operating on *cache lines* instead of variable-sized objects. As such, we model workloads as simple lists of Ints, where each unique number represents the ID of a particular object being accessed (i.e. the address). All of our workload-producing functions thus produce [Int].

We first implemented three synthetic workloads:

- Cycling: Parameterized by cycle width. Performs a linear cycle over a range of values (e.g. 1, 2, ... 100, 1, 2, ...)

- Uniform Random: Parameterized by key space. Generates random values in the key space at random probability.
- Skewed Random: Parameterized by Zipfian constant and key space. Generates random values in the key space according to the Zipfian distribution. Higher Zipfian constant means

We also use the production traces originally used in the ARC [5] paper. These traces were collected from real production workloads to evaluate the performance of the ARC cache eviction algorithm and include traces from desktop workstations, a production OLTP database server, and a production search engine server. We sourced these trace files from the Dgraph project, which uses them as benchmarks for its in-memory cache Ristretto. These traces come compressed with GZip. When uncompressed, they are in a plain text format with 4 numbers per line, with the first two numbers corresponding to a base address and a read width (e.g. 1000 8 0 0 -> read 1000, 1001, 1002, 1003, 1004, 1005, 1006, 1007). At runtime, we read the compressed trace file, uncompress it, and flatten the read sequence into our standard read ID list format.

Due to the size of the ARC traces (millions of accesses), we limit our analysis to simulations on fixed-length windows from the beginning of each trace (50,000 ops).

Eviction Algorithms

We implemented 4 eviction algorithms: Least Recently Used (LRU), First In First Out (FIFO), Least Frequently Used (LFU), and Decay Least Frequently Used (DLFU). Many algorithms require some state for bookkeeping. We can conceptualize the cache eviction algorithm as a pure function:

```
f(state, cache contents, next access) → (state, eviction choice)
```

State is opaque to the simulation runner and is algorithm-specific. As such, it is simply stored after an invocation of the eviction algorithm and passed back into the next invocation. An example of state is the priority queue for the Least Recently Used algorithm.

Cache Simulation

The cache simulator is initialized with the cache size, eviction algorithm, and workload. It creates a representation of the (initially empty) cache contents and begins simulating the workload. For each access in the workload, it tracks if the access is a hit or miss. If it is a miss and the cache is full, it invokes the eviction algorithm and applies the eviction choice to the cache contents. The simulator continues this process until the workload trace is completely consumed. At the end, the simulator returns the miss ratio. Formally:

```
simulate(algorithm, size, workload) → double
```

As final output, once all of the simulations across cache sizes for a given workload and eviction algorithm are complete, the program outputs all of the data points as a complete MRC. We can then visualize the data points separately as MRC plots.

Plotting

For expedience of implementation, we emit results from the parallel cache simulation as plain text. We implemented a python script with matplotlib to parse the results and create graphs.

Parallelization

Simulation of a particular eviction algorithm over a given workload trace is necessarily serial. However, the generation of MRCs requires repeating simulations at different cache sizes, which can be parallelized. Each individual simulation takes as input the intersection of three parameters selected from the sets:

- Eviction Algorithm (A)
- Cache Size (S)
- Workload Trace (W)

Hence, we have $|A| * |S| * |W|$ total simulations which can be performed in parallel.

Results

Miss-Ratio Curves

Below are some of the MRC plots that we have managed to generate using our simulator.

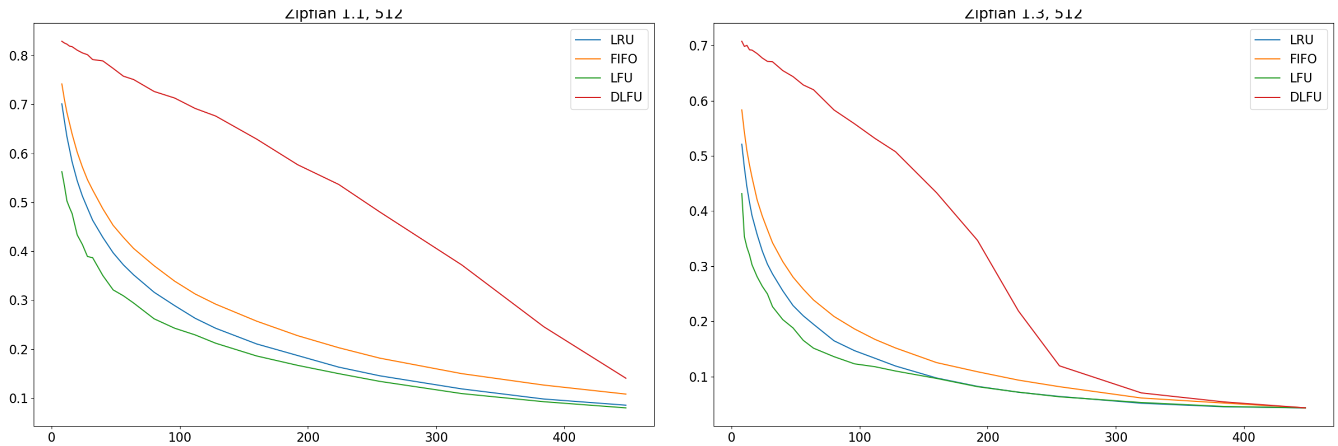

Figure 2: MRCs for a skewed workload at Zipfian constants 1.1 and 1.3

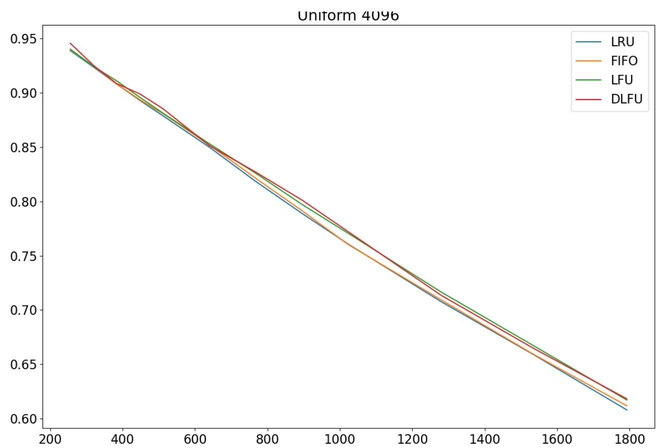

Figure 3: MRC for a uniform random workload

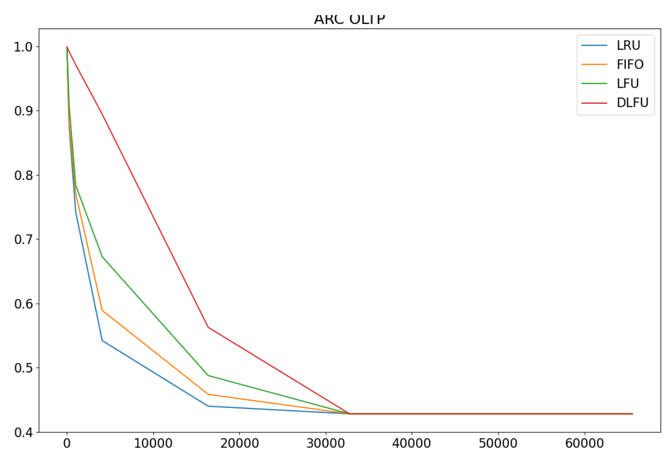

Figure 4: MRC for ARC OLTP trace

Parallelization Speedup

Workload	Serial Time	Parallel Time	Speedup
Uniform Random, 10,000 ops, cache size 2^{3-10}	24.6s	9.66s	2.54x
Skewed Random, 10,000 ops, cache size 2^{3-10}	149s	46.4s	3.2x
ARC, 50,000 ops, cache size 2^{4,6,8,10}	116s	49s	2.3x

Our parallelization strategy improves simulation time by a factor of 2 - 4. Given that we have 8 cores available, we expected the serial to parallel speedup factor to be closer to 8. However, examining the event log trace in Threadscope (shown in Figure 5) reveals that significant time is spent in garbage collection, possibly after a simulation completes.

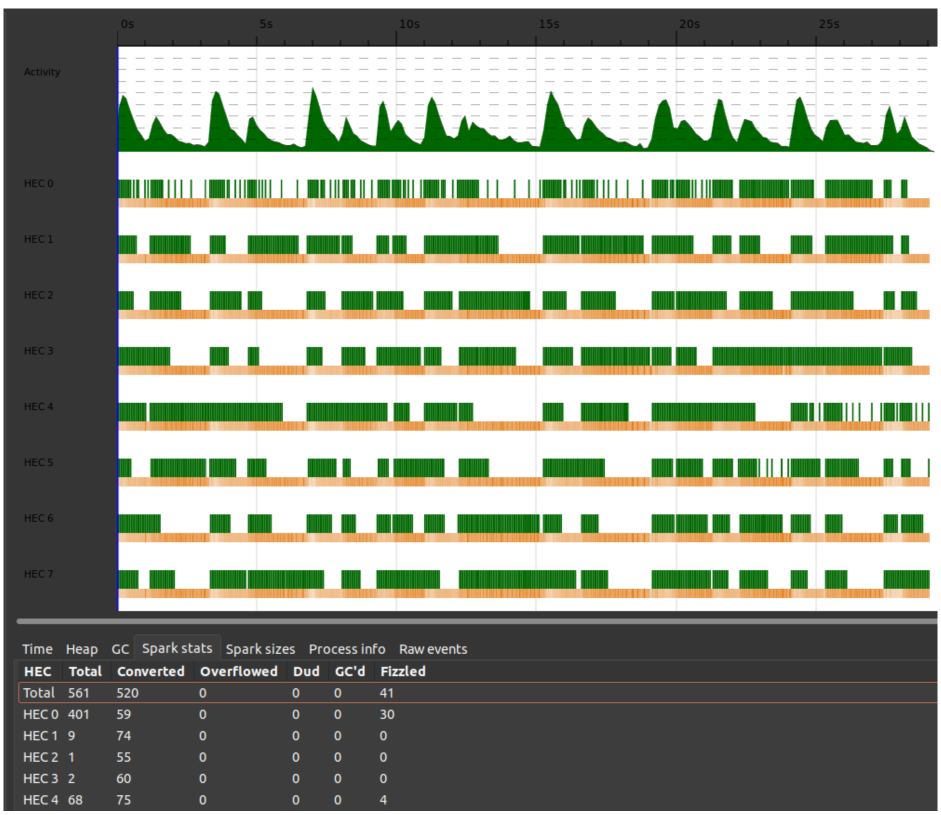

Figure 5: Threadscope visualization of uniform & skewed workloads on an 8-core machine.

Future Work

- Implement Belady's Algorithm to provide a lower bound on cache miss ratio to MRC plots.
- Change data structures used for cache eviction policies to lower garbage collection overhead and computation cost per-eviction.
- Implement sampling to better approximate MRCs for trace-based workloads (e.g. ARC traces) instead of using time-based windowing.

References

[1] Waldspurger, Carl A. et al. "Cache Modeling and Optimization using Miniature Simulations." *USENIX Annual Technical Conference* (2017).

[2] twitter/cache-trace: A collection of Twitter's anonymized production cache traces.

[3] cache2k/cache2k-benchmark: Benchmarks for cache2k and third-party Java caching products

[4] sunnyszy/lrb: A C++11 simulator for a variety of CDN caching policies.

[5] Megiddo et al. "ARC: A Self-Tuning, Low Overhead Replacement Cache." USENIX Conference on File and Storage Technologies (2003).

[6] dgraph-io/benchmarks: Run benchmarks with RDF data.

[7] Minkirri ZWiki DecayingLFUCacheExpiry

Code Listings

app/Main.hs

```haskell
module Main where

import qualified Data.Heap                    as Heap
import qualified Data.List                    as List
import           Data.List.Split              ( splitOn )
import           Lib                          ( DLFU(..)
                                              , FIFO(..)
                                              , LFU(..)
                                              , LRU(..)
                                              , Wrapper(..)
                                              , arcTraceWorkload
                                              , arcTraceWorkload'
                                              , arcTraces
                                              , simulateGraphs
                                              , simulateGraphsSerially
                                              , uniformWorkload
                                              , zipfWorkload
                                              )
import           System.Environment           ( getArgs
                                              , getProgName
                                              )
import           System.Exit                  ( die )
import           System.Random.SplitMix       ( initSMGen )

main :: IO ()
main = do
  args <- getArgs
  usageOrRun args
 where
  usageOrRun [serialOrParallel, workloadSpec, cacheSizesSpec] = do
      let executionStrategy = resolveExecutionStrategy serialOrParallel
      let sizes = interpolateLogSteps $ map read $ splitOn "," cacheSizesSpec

      -- Workloads
      ds1  <- arcTraceWorkload "ds1.arc.gz"
      oltp <- arcTraceWorkload "oltp.arc.gz"
      p3   <- arcTraceWorkload "p3.arc.gz"
      p8   <- arcTraceWorkload "p8.arc.gz"
      s3   <- arcTraceWorkload "s3.arc.gz"

      let arc =
            [ ("ARC DS1" , take 50000 ds1)
            , ("ARC OLTP", take 50000 oltp)
            , ("ARC P3"  , take 50000 p3)
```

```
          , ("ARC P8"  , take 50000 p8)
          , ("ARC S3"  , take 50000 s3)
          ]

    gen <- initSMGen
    let skewed =
          [ ( "Zipfian " ++ show alpha ++ ", " ++ show keyRange
          , zipfWorkload alpha keyRange 10000 gen
          )
          | alpha    <- [1.1, 1.3, 1.5]
          , keyRange <- [2 ^ 8, 2 ^ 9, 2 ^ 10]
          ]
    let uniform = [("Uniform 1024", uniformWorkload (2 ^ 10) 10000 gen)]

    let workload =
          buildWorkload [("skewed", skewed), ("uniform", uniform), ("arc",
arc)]
          $ splitOn "," workloadSpec

    -- Run simulation
    print sizes
    print $ executionStrategy
    workload
    [ LRU $ LRUList []
    , FIFO $ FIFOList []
    , LFU $ LFUHeap Heap.empty
    , DLFU $ DLFUHeap Heap.empty
    ]
    sizes
  usageOrRun _ = do
    pname <- getProgName
    die $ "Usage: " ++ pname ++ " serial|parallel 'uniform,skewed,arc'
'6,7,8'"
  interpolateLogSteps sizes = concatMap
    (\size -> [ 2 ^ size + (2 ^ (size - 2)) * step | step <- [0 .. 3] ])
    sizes
  buildWorkload
    :: [(String, [(String, [Int])])] -> [String] -> [(String, [Int])]
  buildWorkload _ [] = []
  buildWorkload workloads (w : ws) =
    case List.find (\w' -> fst w' == w) workloads of
    Just (_, workload) -> workload ++ buildWorkload workloads ws
    Nothing           -> buildWorkload workloads ws
  resolveExecutionStrategy "serial"   = simulateGraphsSerially
  resolveExecutionStrategy "parallel" = simulateGraphs
  resolveExecutionStrategy _ =
    error "Valid execution strategies: serial, parallel"
```

src/Lib.hs

```haskell
{-# LANGUAGE GADTs #-}

module Lib
  ( module Simulate
  , module Policies
  , module Workloads
  ) where

import            Policies
import            Simulate
import            Workloads
```

src/Policies.hs

```haskell
{-# LANGUAGE GADTs #-}

module Policies
  ( LRU(..)
  , FIFO(..)
  , LFU(..)
  , DLFU(..)
  , Wrapper(..)
  , Policy
  , update
  , evict
  ) where

import qualified Data.Heap                      as Heap
import qualified Data.List                      as List

class Policy s where
      update :: s -> Int -> s
      evict :: s -> Int -> (Int, s)

newtype LRU = LRUList [Int]
newtype FIFO = FIFOList [Int]
newtype LFU = LFUHeap (Heap.MinPrioHeap Int Int)
newtype DLFU = DLFUHeap (Heap.MinPrioHeap Float (Int, Int, Int)) -- Format:
(count (lastTime, totalTime, id))
data Wrapper = LRU LRU | FIFO FIFO | LFU LFU | DLFU DLFU

instance Policy Wrapper where
  update (LRU  l) wid = LRU $ update l wid
  update (FIFO l) wid = FIFO $ update l wid
  update (LFU  l) wid = LFU $ update l wid
  update (DLFU l) wid = DLFU $ update l wid
```

```
  evict (LRU  l) wid = (e, LRU p) where (e, p) = evict l wid
  evict (FIFO l) wid = (e, FIFO p) where (e, p) = evict l wid
  evict (LFU  l) wid = (e, LFU p) where (e, p) = evict l wid
  evict (DLFU l) wid = (e, DLFU p) where (e, p) = evict l wid

instance Show Wrapper where
  show (LRU _) = "LRU"
  show (FIFO _) = "FIFO"
  show (LFU _) = "LFU"
  show (DLFU _) = "DLFU"

instance Policy LRU where
  update (LRUList lruList) wid = LRUList $ wid : List.delete wid lruList
  evict (LRUList lruList) wid = (last lruList, LRUList $ wid : init lruList)

instance Policy FIFO where
  update (FIFOList fifoList) wid | wid `elem` fifoList = FIFOList fifoList
                                 | otherwise = FIFOList $ wid : fifoList
  evict (FIFOList fifoList) wid =
      (last fifoList, FIFOList $ wid : init fifoList)

instance Policy LFU where
  update (LFUHeap lfuHeap) wid
      | length idList == 1 = LFUHeap $ Heap.insert (p + 1, wid) updatedHeap
      | otherwise          = LFUHeap $ Heap.insert (1, wid) updatedHeap
   where
      heapList           = Heap.toList lfuHeap
      (idList, otherList) = List.partition (\(_, val) -> val == wid) heapList
      [(p, _)]           = idList
      updatedHeap        = Heap.fromList otherList

  evict (LFUHeap lfuHeap) wid =
      (evicted, LFUHeap $ Heap.insert (1, wid) $ Heap.drop 1 lfuHeap)
      where [(_, evicted)] = Heap.take 1 lfuHeap

instance Policy DLFU where
  update (DLFUHeap dlfuHeap) wid
      | length idList == 1 = DLFUHeap
      $ Heap.insert (count, (totalT, totalT, wid)) updatedHeap
      | otherwise = DLFUHeap $ Heap.insert (1.0, (1, 1, wid)) dlfuHeap
   where
      decay = 1.0 / (0.0002 * log 2)
      hList = Heap.toList dlfuHeap
      heapList = [ (c, (l, t + 1, v)) | (c, (l, t, v)) <- hList ]
      (idList, otherList) =
      List.partition (\(_, (_, _, val)) -> val == wid) heapList
      [(p, (lastT, totalT, _))] = idList
      count = p * decay / (decay + fromIntegral (totalT - lastT))
      updatedHeap = Heap.fromList otherList
```

```
  evict (DLFUHeap dlfuHeap) wid =
      (evicted, DLFUHeap $ Heap.insert (1.0, (1, 1, wid)) $ Heap.drop 1
dlfuHeap)
      where [(_, (_, _, evicted))] = Heap.take 1 dlfuHeap
```

src/Simulate.hs

```
{-# LANGUAGE GADTs #-}

module Simulate
  ( simulate
  , simulateGraph
  , simulateGraphs
  , simulateGraphsSerially
  ) where

import qualified Control.Parallel.Strategies   as Strategies
import qualified Data.Set                       as Set

import            Policies                      ( Policy
                                                , Wrapper(..)
                                                , evict
                                                , update
                                                )

simulate :: Policy p => [Int] -> p -> Int -> Double
simulate workload policyStart size =
  fromIntegral (tickSimulate workload policyStart (Set.empty, size) 0 :: Int)
      / fromIntegral (length workload)
 where
  tickSimulate (nextTouch : restOfWorkload) policy cache@(cacheContents,
cacheSize) misses
      | Set.member nextTouch cacheContents
      = cacheHit
      | length cacheContents < cacheSize
      = cacheAdd
      | otherwise
      = cacheMiss
   where
      cacheHit =
      tickSimulate restOfWorkload (update policy nextTouch) cache misses
      cacheAdd = tickSimulate restOfWorkload
                              (update policy nextTouch)
                              (Set.insert nextTouch cacheContents, cacheSize)
                              (misses + 1)
      cacheMiss = tickSimulate restOfWorkload
                              policy'
                              (cacheContents', cacheSize)
```

```
                              (misses + 1)
      (evicted, policy') = evict policy nextTouch
      cacheContents'    = Set.insert nextTouch (Set.delete evicted
cacheContents)
  tickSimulate [] _ _ misses = misses

simulateGraph :: [Int] -> Wrapper -> [Int] -> [Double]
simulateGraph workload policy sizes = Strategies.parMap
  Strategies.rpar
  (\size -> simulate workload policy size)
  sizes
  --Strategies.withStrategy (Strategies.parList Strategies.rdeepseq)
  --  $ map (simulate workload policy) sizes

simulateGraphs
  :: [(String, [Int])] -> [Wrapper] -> [Int] -> [(String, String, [Double])]
simulateGraphs workloads policies sizes = Strategies.parMap
  Strategies.rpar
  (\((workloadName, workloadAccesses), policy) ->
      (workloadName, show policy, simulateGraph workloadAccesses policy
sizes)
  )
  [ (workload, policy) | workload <- workloads, policy <- policies ]

simulateGraphsSerially
  :: [(String, [Int])] -> [Wrapper] -> [Int] -> [(String, String, [Double])]
simulateGraphsSerially workloads policies sizes =
  [ (workloadName, show policy, simulateGraphSerially workloadAccesses
policy)
  | (workloadName, workloadAccesses) <- workloads
  , policy                          <- policies
  ]
 where
  simulateGraphSerially workload policy = map (simulate workload policy)
sizes
```

src/Workloads.hs

```
module Workloads
  ( cycleWorkload
  , uniformWorkload
  , zipfWorkload
  , arcTraceWorkload
  , arcTraces
  , histogram
  , arcTraceWorkload'
  ) where
```

```haskell
import            Codec.Compression.GZip       ( decompress )
import qualified Data.ByteString.Lazy         as ByteString
import qualified Data.ByteString.Lazy.UTF8    as UTF8
import            Data.List                    ( group )
import            Data.Sort                    ( sort )
import            System.Random.SplitMix       ( SMGen
                                               , nextWord64
                                               )
import            System.Random.SplitMix.Distributions
                                               ( sample
                                               , samples
                                               , uniformR
                                               , zipf
                                               )

cycleWorkload :: (Num a, Enum a) => a -> Int -> [a]
cycleWorkload keyRange numOps = take numOps $ cycle [1 .. keyRange]

uniformWorkload :: Integral b => Int -> Int -> SMGen -> [b]
uniformWorkload keyRange numOps gen = map round
  $ samples numOps (fst $ nextWord64 gen) (uniformR 1 (fromIntegral
keyRange))

zipfWorkload :: (Eq t, Num t, Integral a) => Double -> a -> t -> SMGen -> [a]
zipfWorkload _    _     0     _   = []
zipfWorkload alpha keyRange numOps gen = fst nextValue
  : zipfWorkload alpha keyRange (numOps - 1) (snd nextValue)
 where
  nextValue = genNextValue gen
  genNextValue gen' = if fst candidate <= keyRange
      then candidate
      else genNextValue (snd nextSeed)
   where
      candidate = (sample (fst nextSeed) (zipf alpha), snd nextSeed)
      nextSeed  = nextWord64 gen'

histogram :: Ord a => [a] -> IO ()
histogram l = mapM_ (putStrLn . encoded) runs
 where
  runs = (group . sort) l
  encoded r = replicate (length r) '*'

arcTraces :: [[Char]]
arcTraces =
  ["ds1.arc.gz", "oltp.arc.gz", "s3.arc.gz", "p3.arc.gz", "p8.arc.gz"]

arcTraceWorkload :: (Num b, Read b, Enum b) => [Char] -> IO [b]
arcTraceWorkload traceFile = do
  traceContent <- fmap (UTF8.toString . decompress)
                        (ByteString.readFile ("traces/arc/" ++ traceFile))
```

```
  return (concatMap traceAccesses $ lines traceContent)
 where
  traceAccesses line = accessSequence $ words line
  accessSequence (base : count : _) = map (+ read base) [1 .. (read count)]
  accessSequence _                  = []


arcTraceWorkload' :: (Num b, Read b, Enum b) => [Char] -> IO [b]
arcTraceWorkload' traceFile = do
  contents <- readFile ("traces/arc/" ++ traceFile)
  return (concatMap traceAccesses $ lines contents)
 where
  traceAccesses line = accessSequence $ words line
  accessSequence (base : count : _) = map (+ read base) [1 .. (read count)]
  accessSequence _                  = []
```

plot_mrc.py

```
from os import listdir, makedirs
from os.path import isfile, join
import sys

from pprint import pprint
import matplotlib.pyplot as plt


def load_measurements(sizes, data):
  curves = {}
  for (workload, policy, points) in data:
      if not workload in curves:
      curves[workload] = {}
      curves[workload][policy] = points

  print(pprint(curves))
  return sizes, curves

# measurements: dict[workload]
def plot_measurements(out_dir, sizes, measurements):
  for workload in measurements:
      plt.rcParams["figure.figsize"] = (12, 8)
      plt.rcParams.update({'font.size': 16})

      for policy in measurements[workload]:
      plt.plot(sizes, measurements[workload][policy], label = policy)
```

```
    plt.tight_layout()  # otherwise the right y-label is slightly clipped
    plt.title(workload)
    plt.legend(loc="upper right")

    # plt.show()
    plt.savefig(join(out_dir, f"{workload}.png"))
    plt.clf()

if __name__ == "__main__":
    in_file = sys.argv[1]
    sizes = None
    data = None
    with open(in_file, "r") as f:
    sizes = eval(f.readline())
    data = eval(f.readline())
    out_dir = "mrcs"
    makedirs(out_dir, exist_ok=True)

    sizes, measurements = load_measurements(sizes, data)
    plot_measurements(out_dir, sizes, measurements)
```