Simulation and Modeling Coursework

CID: yhc615 Yoon Hyuk Chang

CID: thm15 Tom Miller

November 13, 2017

1 Q1

Our simulator takes 4 arguments:

- Item Size (n)
- Cache Size (m)
- Time Steps (t)
- Replacement Policy (FIFO:0, RAND:1)

To calculate point/interval estimates for:

- Item Size=1000
- Cache Size=100/50/10
- Replacement Policy=FIFO/RAND
- Replacement Policy (FIFO:0, RAND:1)

We set Time Steps=1000, Number of Simulation Runs=10 to achieve the appropriate results. These arguments resulted in the results below in which we set a 95% confidence level for generating our Interval Estimates.

		Run-1	Run-2	Run-3	Run-4	Run-5	Run-6	Run-7	Run-8	Run-9	Run-10	Point Estimate	Interval Estimate	
													-	+
FIFO m=100	Hit Rate	3.63801	4.23947	3.88792	4.01217	3.86513	4.20008	4.12014	4.23739	3.85231	3.67658	3.97292	3.81312	4.13272
	Miss Rate	3.48138	3.51094	3.72053	3.48721	3.78860	3.27337	3.34388	3.17063	3.71608	3.50424	3.49969	3.35593	3.64344
FIFO m=50	Hit Rate	3.12604	2.74437	3.25388	3.23054	3.10193	2.85238	3.16791	3.21668	2.85397	2.76070	3.03084	2.88478	3.17690
	Miss Rate	4.12695	4.73348	4.34864	4.53518	4.50084	4.65388	4.71246	4.36983	4.44519	4.52347	4.49499	4.36290	4.62708
FIFO m=10	Hit Rate	1.51475	1.42562	1.18720	1.35498	1.50600	1.13227	1.32463	1.34450	1.47846	1.36324	1.36317	1.27183	1.45450
	Miss Rate	6.09707	6.19802	6.27945	5.85237	6.21707	6.07965	6.20168	5.92308	5.62951	6.29543	6.07733	5.92404	6.23062
RAND m=100	Hit Rate	4.33805	4.56947	4.50855	3.82613	4.27887	3.62192	3.94485	3.73940	4.18649	4.10070	4.11144	3.88029	4.34259
	Miss Rate	3.38090	3.34989	3.29171	3.58885	3.50089	3.45214	3.54063	3.56411	3.56627	3.42352	3.46589	3.39253	3.53925
RAND m=50	Hit Rate	3.01170	2.96687	2.84776	3.23609	2.78829	3.12969	2.64659	3.10641	3.05377	3.14331	2.99305	2.86224	3.12385
	Miss Rate	4.63221	4.60168	4.54903	4.45058	4.39803	4.35761	4.28165	4.11780	4.46783	4.16671	4.40231	4.27755	4.52707
RAND m=10	Hit Rate	1.42398	1.36850	1.54111	1.30849	1.35449	1.41404	1.65079	1.25180	1.24493	1.44596	1.40041	1.31020	1.49062
	Miss Rate	5.91612	5.91077	6.01336	5.72642	6.04708	6.10744	6.44131	6.62116	6.20973	6.20461	6.11980	5.93104	6.30856

Analysing the results side by side, and comparing those from the FIFO and RAND policy, we found that they matched very well for each cache size. From these results alone it demonstrates that the choice of FIFO/RAND have little effect on the resulting Hit/Miss Rates.

Cache Size(m)	FIFO	RAND
100	3.97(3.81-4.13)	4.11(3.88-4.34)
	3.50(3.36-3.64)	3.47(3.39-3.54)
50	3.03(2.88-3.18)	2.99(2.86-3.12)
	4.49(4.36-4.63)	4.40(4.28-4.53)
10	1.36(1.27-1.45)	1.40(1.31-1.49)
	6.08(5.92-6.23)	6.12(5.93-6.31)

1.1 Code explanation:

Our simulator operates by making use of two main functions:

- `timeStep(globalTime,eventTimeline,cache)`
- `nextRequest(`λ`)`

We first initialize our cache by filling it with Items $(1, 2, 3, \ldots, m)$, before then populating the timeline with the initial request of those items. To generate these initial and future arrival times we modelled the Poisson processes by randomly sampling an exponential distribution for each Item using $-\log(\texttt{randomNum}) * \lambda$, in which $\lambda = 1/k$ and k denotes the Item ID.

This sampling occurs each time `nextRequest(`λ`)` is called, which for our simulator occurs every time an Item is processed and in turn needs to be re-added to the `eventTimeline`, at `globalTime+nextRequest(`λ`)`.

Essentially this means that every time we call `timeStep(ARGS)` we are simply requesting the next item on the `eventTimeline`, which due to it being a sorted map, will always be the first element.

2 Q2, CTMC model of Cache with Random Eviction Policy

Recognizing that the order in which the cache contents are placed in is independent of whether a cache request hits or misses, the cache can be modeled as CTMC with the following states:

1. Cache contains 1 AND 2 : $S = (1, 2) \mid (2, 1)$
2. Cache contains 2 AND 3 : $S = (2, 3) \mid (3, 2)$
3. Cache contains 1 AND 3 : $S = (1, 3) \mid (3, 1)$

The transition rate, $q_{i,j}$, from state i to j can be obtained by examining the cache contents of the current state and evaluating the rate at which request for item k that would cause a cache miss arrives, and multiplying that by the probability that item z, which is the cache item that needs to be evicted in order to move from state i to j, is evicted:

$$q_{i,j} = \lambda_k * P(\text{item } z \text{ is evicted}),$$

Where $P(\text{item } z \text{ is evicted}) = 1/2$, as items are randomly evicted from the cache. (Either s_1 or s_2 is evicted)

The state space size, $|S|$ is 3. Therefore, *the generator matrix* $\mathbf{Q}$ is calculated from the transition rates as a 3 x 3 matrix:

$$\mathbf{Q} = \begin{pmatrix} -\lambda_3 & \lambda_3/2 & \lambda_3/2 \\ \lambda_1/2 & -\lambda_1 & \lambda_1/2 \\ \lambda_2/2 & \lambda_2/2 & -\lambda_2 \end{pmatrix}$$

3 Q3

Due to the replacement policy being RAND, all transitions out of a state will be equally as likely to occur, for example with $m = 5$ the number of possible transitions per state is $2(m-2) = 6$. This means that each transition has a rate of $\frac{1}{6}\lambda_k$ or in total, $6 * \frac{1}{6}\lambda_k = \lambda_k$ where λ_k is the probability of being in state k.

This means that the total rate out of a state for arbitrary m is equal to λ_k, the probability of being in that state.

4 Q4

Substituting in $\lambda_k = 1/k$, and filling column 4 with 1s gives us the balance equation:

$$p\mathbf{Q} = 0 \implies \begin{pmatrix} p_1 \\ p_2 \\ p_3 \end{pmatrix} * \begin{pmatrix} -1/3 & 1/6 & 1/6 & 1 \\ 1/2 & -1 & 1/2 & 1 \\ 1/4 & 1/4 & -1/2 & 1 \end{pmatrix} = \begin{pmatrix} 0 & 0 & 0 & 1 \end{pmatrix}$$

Solving using `matlab` gives us $p = \begin{pmatrix} 1/2 & 1/6 & 1/3 \end{pmatrix}$.

To calculate the hit ratio p_{hit}, solve for the probability that a requested item i, is in the cache:

$$\begin{aligned} p_{hit} &= P(\text{requested item } i \text{ is in the cache}) \\ &= P(i=1) * (p_1 + p3) + P(i=2) * (p_1 + p2) + P(i=3) * (p_2 + p3) \\ &= 8/11 \approx 0.727 \end{aligned}$$

Running the `matlab` implementation (see appendix) gives us the console output:

```
>> ctmc
-------------Results for RAND eviction -------------
Hit Ratio from CTMC modelling: 0.727273
Average hit ratio from simulation (10 runs) : 0.720000
```

Giving us a pretty close result to the model.

5 Q5

For a FIFO replacement policy, we trace the possible state transitions of the cache, given $n = 3$ and $m = 2$, and that the cache is initially loaded as $(1, 2)$. After tracing (See Appendix C), it can be observed that the cache transitions through the following states: $(1, 2) \to (3, 1) \to (2, 3) \to (1, 3) \to (2, 1) \to (3, 2)$. Note that the CTMC is not irreducible. Mapping each configuration of the cache to one state(eg. $(1, 2)$ maps to state 1, $(1, 3)$ to state 2, $(2, 1)$ to state 3, and so forth), it can be seen that the cache goes through the following state transitions: $1 \to 5 \to 4 \to 2 \to 3 \to 6$, before going back to state 1.

The transition rate here is simply λ_k, the rate that a kth item request that triggers the state transition from i to j. This gives us the transition matrix $\mathbf{Q}$:

$$\mathbf{Q} = \begin{pmatrix} -\lambda_3 & 0 & 0 & 0 & \lambda_3 & 0 \\ 0 & -\lambda_2 & \lambda_2 & 0 & 0 & 0 \\ 0 & 0 & -\lambda_3 & 0 & 0 & \lambda_3 \\ 0 & \lambda_1 & 0 & -\lambda_1 & 0 & 0 \\ 0 & 0 & 0 & \lambda_2 & -\lambda_2 & 0 \\ \lambda_1 & 0 & 0 & 0 & 0 & -\lambda_1 \end{pmatrix}$$

Setting up the transition matrix and solving the balance equations gives us $p = \begin{pmatrix} 1/4 & 1/6 & 1/4 & 1/12 & 1/6 & 1/12 \end{pmatrix}$.
To calculate the hit ratio p_{hit}, apply the methods in Q4 - ie. the sum of the probabilities of each state multiplied by probability that either of the contents in the cache in that state is requested, which results in a cache hit:

$$\begin{aligned} p_{hit} &= P(\text{requested item } i \text{ is in the cache}) \\ &= p_1 * (P(i=1) + P(i=2)) + p_2 * (P(i=1) + P(i=3)) + p_3 * (P(i=2) + P(i=1)) \\ &+ p_4 * (P(i=2) + P(i=3)) + p_5 * (P(i=3) + P(i=1)) + p_6 * (P(i=3) + P(i=2)) \\ &= 8/11 \approx 0.727 \end{aligned}$$

Running the `matlab` implementation (see appendix) gives us the console output:

```
-------------Results for FIFO eviction -------------
Hit Ratio from CTMC modelling: 0.727273
Average hit ratio from simulation (10 runs) : 0.702000>>
```

Again giving us a pretty close result to the model, and confirming that the hit ratio p_{hit} for random eviction is the same as that of the FIFO policy.

A Simulation Code

A.1 main.cpp implementation

```
#include <iostream>
#include <cmath>
#include <random>
#include <map>
#include <time.h>
#include "cache.hpp"

using namespace std;

double nextRequestTime(double lambda){
    double randNum = (double)rand()/RAND_MAX;
    return -log(randNum)*lambda;
}

int processQueue(double &time, map<double, int> &que, Cache &cache){
    int itemIndex = que.begin()->second;
    cache.request(itemIndex); // FIFO=0
    que.erase(que.begin());
    que.insert(pair<double,int>(time+nextRequestTime(itemIndex),itemIndex));
    return 0;
}

int timeStep(double &time, map<double, int> &que, Cache &cache){
    double eventTime = que.begin()->first;
    time = eventTime;
    processQueue(time, que, cache);
    return 0;
}


int main(int argc, char* argv[]) { // argv=>  [1] = item size ,
                                   //          [2] = cache size (m),
                                   //          [3] = time steps,
                                   //          [4] = mode (FIFO:0, RAND:1)
    srand(time(NULL)); // initialize random no. generator

    double gTime = 0;
    int missNum = 0;

    //convert params
    int n = atoi(argv[1]);
    int m = atoi(argv[2]);
```

```
    int tSteps = atoi(argv[3]);
    int mode = atoi(argv[4]);


    //Create and initialize cache
    Cache simCache(m, mode);
    simCache.fill();

    map<double, int> q;
    //Fill events(que) with exponentially distributed request times
    for(int i=1; i<=n; i++){
        q.insert(pair<double,int>(nextRequestTime(i),i));
    }

    simCache.print(); // initial cache contents
    for(int i=0; i<tSteps; i++){ // progress time in hops
        timeStep(gTime, q, simCache);
    }
    simCache.print(); // cache contents after simluation

    cout << "Final Time: " << gTime << endl;
    cout << "Hit Count: " << simCache.getHitCount() << endl;
    cout << "Miss Count: " << simCache.getMissCount() << endl;
    cout << "Hit Rate: " << simCache.getHitCount()/gTime << endl;
    cout << "Miss Rate: " << simCache.getMissCount()/gTime << endl;

    return 0;
}
```

A.2 cache.hpp header implementation

```
#ifndef CACHE_HPP
#define CACHE_HPP

#include <iostream>
#include <cmath>
#include <random>
#include <queue>

class Cache{
public:
        Cache(int m, int policy)
            : size(m), replacement_policy(policy), missCount(0),hitCount(0)
    {//std::cout<<"created Cache of capacity: "<<m<<std::endl;
         //std::cout<<"Replacement Policy: "<<replacement_policy<<std::endl;
    }
```

```
    //Methods
    void fill(){
        for(int i=0; i<size; i++){
            items.push_back(i+1);
        }
    }
    void print(){
        std::cout << "--------Printing Cache contents--------" << std::endl;
        for(int i=0; i<size; i++){
            std::cout << items[i] << "  |  ";
        }
        std::cout<<"\n";
        std::cout << "-------------------------------------------" << std::endl;
    }
    void request(int itemIndex){
        std::deque<int>::iterator it = find(items.begin(),items.end(),
                                            itemIndex);
        if(it == items.end()){
            if(replacement_policy==0){
                items.pop_front();
                items.push_back(itemIndex);
            }
            if(replacement_policy==1){
                int randNum = (double)rand()*size/RAND_MAX;
                items[randNum] = itemIndex;
            }
            //std::cout << "MISS: " << itemIndex << std::endl;
            missCount++;
        }
        else{
            //std::cout << "HIT: " << itemIndex << std::endl;
            hitCount++;
        }
    }

    //Getters
    int getMissCount(){
        return missCount;
    }
    int getHitCount(){
        return hitCount;
    }

        //Member data
private:
```

```
        int size;
        int replacement_policy; //FIFO: 0, RAND:1
        int hitCount;
        int missCount;
        std::deque<int> items; // each element is an int representing an item
    //from population
};

#endif
```

B Matlab code for solving CTMC equations & Result checking

```
lambda_k = [1 1/2 1/3];
p_k = []; %probability that item k is requested
for i=1:size(lambda_k,2)
   p_k(i) = lambda_k(i)/sum(lambda_k);
end

% RAND Modelling (CTMC) vs Simulation
Q_RAND = [-1/3 1/6 1/6 1; 1/2 -1 1/2 1; 1/4 1/4 -1/2 1]; % state transition
%matrix
B_RAND = [0 0 0 1];
P_RAND = B_RAND/Q_RAND; % PQ = B , B = 0 with c index to allow for unique
%solution to p

%calculate hit ratio
p_hit_RAND = p_k(1)*( P_RAND(1) + P_RAND(3) ) +
p_k(2)*( P_RAND(1) + P_RAND(2) ) + p_k(3)*( P_RAND(2) + P_RAND(3) );

RAND_sample = [ 68; 61; 67; 74; 79; 73; 73; 79; 69; 77 ]; %sample results

%Output results
output_txt = "---------------Results for RAND eviction --------------- \nHit
Ratio from CTMC modelling: %f \nAverage hit ratio from simulation (10 runs)
: %f \n";
fprintf(output_txt, p_hit_RAND, mean(RAND_sample)/100);

%FIFO
Q_FIFO = [-1/3 0 0 0 1/3 0 1; 0 -1/2 1/2 0 0 0 1; 0 0 -1/3 0 0 1/3 1;
          0 1 0 -1 0 0 1; 0 0 0 1/2 -1/2 0 1; 1 0 0 0 0 -1 1];
B_FIFO = [0 0 0 0 0 0 1];
P_FIFO = B_FIFO/Q_FIFO;
```

```
%calculate hit ratio
p_hit_FIFO = P_FIFO(1)*( p_k(1) + p_k(2) ) + P_FIFO(2)*( p_k(1) + p_k(3) )
+ P_FIFO(3)*( p_k(2) + p_k(1) ) + P_FIFO(4)*( p_k(2) + p_k(3) )
+ P_FIFO(5)*( p_k(3) + p_k(1) ) + P_FIFO(6)*( p_k(3) + p_k(2) );

FIFO_sample = [70; 73; 73; 62; 76; 69; 70; 70; 69; 70];
output_txt = "----------Results for FIFO eviction ----------
\nHit Ratio from CTMC modelling: %f \nAverage hit ratio from simulation
(10 runs) : %f";
fprintf(output_txt, p_hit_FIFO, mean(FIFO_sample)/100);
```

*note: entered some spacing between some lines in the code for visibility

C Tracing the FIFO cache