POLITECNICO DI TORINO

Facoltà di Ingegneria

Corso di Laurea in Master of Science in Computer and Communication Networks Engineering

Tesi di Laurea Magistrale

Caching Systems: K-LRU Optimization and Performance Evaluation

Relatore:
prof. Emilio Leonardi

Candidato:
Valerio Casella

Anno accademico 2017-2018

Summary

This work has the aim of performance evaluation of the optimized K-LRU caching system: in particul we will refer to 2-LRU and 3-LRU policies. Such improvement is usuful to Content Deliver Networks (CDN) provider, in order to increase the performance satisfying large users request. We will show, furthermore, that our optimization works under different traffic model, e.., IRM and SNM, different network scenarios, e.g., single cache and cache networks, and with different content popularity distributions.

Acknowledgements

A special thanks to my supervisor Prof. Emilio Leonardi and to Dott. Michele Tortelli, who helped me to develop and implement such optimization in caching systems scenario.
A special thanks to my family and to Silvia who supported me during the development of this work.

Contents

Chapter 1

Background

1.1 Introduction

In the past few years the performance of caching systems, one of the most traditional and widely investigated topic in computer science, has received a wide interest by the networking research community. Because the crucial role played by caching in new content distribution systems emerging within the impact of Internet. Content Delivery Networks (CDN) represent today the standard solution adopted by content providers to serve large populations geographically distributed in order to satisfy spread users[1]. Thanks to caching contents close to the users, network traffic is reduced by improving user-perceived experience.
The fundamental role played by caching systems in the Internet exceed the existing content delivery networks, by changing the communication model in host-to-content paradigm. Indeed, the Information-Centric Networking (ICN) architecture has been proposed for the future Internet to better answer to the today and future traffic request[2]. In this architecture, caching becomes functionality available at each router thanks to its characteristics.
Evaluating the performance of cache networks is hard, considering that the computational cost needed to analyse just a single LRU (Least Recently Used) cache, grows exponentially with both the cache size and the number of contents.[3], [4] Nevertheless, several approximations have been proposed over the years [5], [6], [7], [8], [9] which can accurately predict cache performance at a reasonable computational cost.
The main drawback of existing analytical techniques consist in the approximation due the simplified traffic conditions for several caching policies (mainly LRU)[10]. This work refer to the applicability of the main approximation (the so called Che's approximation) specifically used for a different caching policies under IRM traffic and a particular type of ON-OFF traffic, called SNM. The aim consists to provide performance results of a variety of caching systems through simulations done with a scalable chunk-level simulator for CCN environments, called ccnSim[27].

In particular, this work is based to the evaluation of "optimized" multi-stage LRU scheme (essentially 2-LRU and k-LRU) under traditional traffic model IRM, and a type of ON-OFF traffic called SNM, cosidering the effects of temporal locality in the requests arrival process (in particular, we refer to a standard Poisson traffic model for all the above-mentioned caching policies). Such cache decision policies have been investigated with different networks scenarios: single cache and network of LRU caches having a binary-tree topology of interconnected caches with four layers (15 caches).
In the case of IRM traffic, Zipf-like law is used to describe content popularity, whereas in the case of SNM traffic in order to garantee a major stabilization the average number of requests are characterized by both Zipf's law and Pareto probability distribution, (at each ingress cache in the case of binary-tree network topology).
Hence, the aim of this work, is providing results of performance analysis of caching systems under the Che's approximation, however all plots are obtained through simulations enabling to evaluate the *hit probability* of several caching policies and "optimized" ones, at variation of caching size.

1.2 System Assumption

1.2.1 Traffic Model

The traffic model adopted in the literature to characterize the pattern of object requests arriving at a cache is the so-called Independent Reference Model (IRM)[10]. The IRM is based on the following fundamental assumptions: i) users request items from a fixed catalogue of M object; ii) the probability p_m that a request is for object m, $1 \leq m \leq M$ is constant (i.e., the object popularity does not vary over time) and *independent* of all previous requests, generating an i.i.d. sequence of requests[10].

By construction, the IRM does not take into account temporal correlations in the sequence of requests. In particular, makes negligible the principle of temporal locality: requests for a given content become denser over short periods of time. However in the real contest request is often observed, indeed temporal locality play an important role on cache performance, is well known even in the context of computer memory architecture[10] and web traffic[10]. Several approaches referring to IRM have been proposed to reproduce content temporal locality[12],[13],[14]. The majority of proposed approaches, based on IRM, have the following two assumptions: i) the content catalog consists of a fixed number of objects, which does not change over time; ii) the request process for each content is stationary (typically it is assumed to be either a renewal process or a semi-Markov-modulated Poisson process with average rate equal to λ)[13],[15].

Recently a new traffic model, named Shot Noise Model (SNM), has been proposed as a adaptive alternative to traditional traffic models able to capture the effects of content popularity dynamics[14], it means that allows to simulate scenarios where content popularities changes and evolves over time, where the temporal locality of requests is modeled through an ON-OFF process that models the request patterns of different classes of contents.[27]

The basic idea of the SNM is to represent the overall request process as overlap of a potentially infinite population of independent inhomogeneous Poisson processes (shots), whose instantaneous rate at time t is given by $V_m\lambda_m(t - \tau_m)$[14] and each request referring to an individual content m.[11]
We represents with τ_m the time instant at which the content enters in the system (i.e., when it can be requested by the users and becomes available to the users), V_m denotes the average number of requests generated by the content and $\lambda_m(t)$ is the popularity profile, describing how the request rate for content m evolves over time.[14][43]

Computing its analytical models for the evaluation of cache performance under the SNM[14],[16], however, is significantly challenging[15], especially when non-LRU caches and networks of caches are analyzed.
For the sake of semplicity we will refer to SNM traffic model where inter-arrival request times are independently, exponentially distributed, so that requests for contents m are generated according to a homogeneous Poisson process of constant rate λ_m[11] .

1.2.2 Content popularity: Zipf's law

Traffic models like the IRM (and its generalizations) are commonly used in combination with a Zipf-like distribution of content popularity, which is often observed in traffic measurements and widely adopted in performance evaluation studies[17],[18]. In its simplest form, Zipf's law states that the probability to request the i-th most popular item is proportional to $1/i^{\alpha}$ where the exponent α depends on the considered system (especially on the type of objects), and plays a crucial role on the resulting cache performance[6].
In particular, Zipf's distributed random numbers are generated without allocating memory space to store the Cumulative Distribution Function (CDF) vector, which for large scenarios, represents a main cause for memory request[27]. Estimates of α reported in the literature for various kinds of systems range between 0.65 and 1[19]. This work will consider a simple Zipf's distribution as the

object popularity law, although results obtained hold for any given distribution of object request probabilities.

1.3 Replacement Policies

There exists a huge number of different policies to manage the insertion/eviction cache objects, which differ either for the insertion or for the eviction rule. We will consider the following algorithms, as a set of most used existing policies taken from [11]:

- **LFU**: the *Least Frequently Used* policy statically stores in the cache the C most popular contents (assuming their popularity is known a-priori); LFU is known to provide optimal performance under IRM.
- **LRU**: upon arrival of a request, an object not already stored in the cache is inserted into it. If the cache is full, to make room for a new object the *Least Recently Used* item is evicted, i.e., the object which has not been requested for the longest time.
- **q-LRU**: it differs from LRU for the insertion policy: upon arrival of a request, an object not already stored in the cache is inserted into it with probability q. The eviction policy is the same as LRU.
- **FIFO**: it differs from LRU for the eviction policy: to make room for a new object, the item inserted the longest time ago is evicted. Notice that this scheme differs from LRU in this respect: requests finding an object in the cache do not "refresh" the arrival time associated to it.
- **RANDOM**: it differs from LRU for the eviction policy: to make room for a new object, a random item stored in the cache is evicted.
- **k-LRU**: this strategy provides a clever insertion policy by exploiting the following idea: before arriving at the (physical) cache which is storing actual objects, indexed by k, requests have to advance through a chain of $k-1$ (virtual) caches put in front of it, acting as filters, which store only object pointers performing caching operations on them.
  Specifically, upon arrival of a request, a content/pointer can be stored in cache $i > 1$ only if its pointer is already stored in cache $i-1$ (i.e. the arrival request has produced a hit in cache $i-1$)[11]. The eviction policy at all caches is LRU. We remark that this policy can be seen as a generalization of the two-stages policy proposed in[20], called there LRU-2Q.[11]
  Let T_C^i be the eviction time of cache i, let the hit probability of object m in cache i, p_{hit} (i, m), to the hit probability p_{hit} (i - 1, m) of object m in the previous cache, under the independence assamption between caches and IRM traffic, we obtain:

$$p_{hit}(m) = p_{in}(m) \approx (1 - e^{-\lambda_m T_C^i})[p_{hit}(i,m)(p_{hit}(i-1,m))(1 - p_{hit}(m))][11] \tag{1.1}$$

- **k-RANDOM**: it works exactly like k-LRU, with the only difference that the eviction policy at each cache is RANDOM.

1.4 2-LRU

As I introduced in the previous section (i.e., Sec. 1.3), in this work is usuful to describe the implementation of a particular case of k-LRU cache decision policy and proposed in [11], namely 2-LRU. For this system, by starting from a single LRU cache, an additional LRU cache (known as Name Cache) where names of requested contents are stored is put in front of the main one.[27] When a request is received, a first lookup is performed on the Name Cache: in case of a positive outcome, the correspondent fetched content will be later stored inside the main cache (the second stage cache), otherwise only the ID content name of the

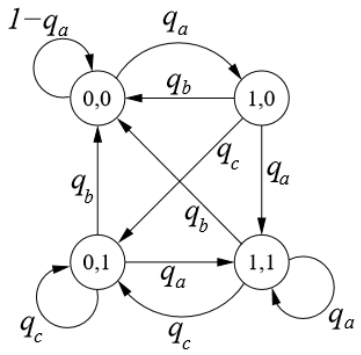


Figure 1.1. DTMC describing the dynamics of an object in 2-LRU, sampled at request arrival times.[11]

requested content will be inserted inside the Name Cache (the first stage cache) without effectively caching the retrieved object.[27]
The aim of the 2-LRU cache decision policy is to reduce unpopular contents (i.e., contents that are requested very rarely) in order to increase the hit probability. The 2-LRU policy provides, also, good performance in the presence of temporal locality, meaning that it can locally adapt to content popularity changes over short-term time caused by temporal locality [27].
Let T_C^i be the eviction time of cache i. Under IRM, $p_{in}(m)$ and $p_{hit}(m)$ (which are identical by PASTA) can be approximately derived by the following constraint: object m is found in cache 2 at time t if and only if the last request arrived in τ within interval $[t - T_C^2, t]$ and either object m was already stored in cache 2 at time τ^- or it was not in cache 2 at time τ^- but its ID was already stored in meta-cache 1[11],[14],[43].
Under the additional approximation that the states of meta-cache 1 and cache 2 are independent at time τ^-, from the extended eq 1.1, we obtain:

$$p_{hit}(m) = p_{in}(m) \approx (1 - e^{-\lambda_m T_C^2})[p_{hit}(m)(1 - e^{-\lambda_m T_C^1})(1 - p_{hit}(m))][11] \tag{1.2}$$

Dynamics of object m in the system, sampled at request arrivals, can be described by the four states Discrete Time Markov Chain (DTMC) (rappresented in Fig. 1.1) where each state is denoted by two binary variables indicating the presence of object m in cache 1 or cache 2, respectively[11].

We remark that LRU has been widely adopted, since it provides good performance while being reasonably simple to implement. RANDOM and FIFO have been considered as viable alternative to LRU in the context of ICN, as their hardware implementation in high-speed routers is even simpler[11]. The q-LRU policy and multi-stage caching systems similar to k-LRU have been proposed in the past to improve the performance of LRU by means of a better insertion policy. Regarding q-LRU, in its simplicity, gives an immediate interpretation in terms of probabilistic content replication for cache networks. The main power of k-LRU, instead, consists in the fact that it requires just one traffic-independent parameter (the number of caches k), providing significant improvements over LRU even for very small k (much of the possible gain is already achieved by $k = 2$)[11].

1.5 Decision Strategies for cache networks

In a system of interconnected caches, requests producing a miss at one cache are typically forwarded along one or more cache chain-connected toward repositories storing all objects. After the request eventually hits the target object, we need to specify how such object gets replicated back in the network, in particular along the path traversed by the request[11]. There are a series of mechanisms (or meta-caching algorithm) that decide if the incoming Data should be stored or not in the local cache. We will consider the following strategies taken from[22][27]:

- **leave-copy-everywhere (LCE)**: the object is sent to all caches of the backward path, meaning that each incoming contents is always stored within the cache.
- **leave-copy-probabilistically (LCP)**: the object is sent with probability q to each cache of the backward path.
- **leave-copy-down (LCD**: the object is sent only to the cache preceding the one in which the object is found (unless the object is found in the first visited cache), meaning that the incoming content is stored in the local cache only if it has been originally fetched from a node one hop away from the current node.

Notice that LCP, combined with standard LRU at all caches, is the same as LCE combined with q-LRU at all caches.[11]

1.6 The Che Approximation

We briefly recall Che's approximation for LRU under the classical IRM[5]. Consider a cache capable of storing C objects. Let $T_C(m)$ be the time needed before C *distinct* objects (not including m) are requested by users. Therefore, $T_C(m)$ is the *cache eviction time* or content m, i.e., the time since the last request after which object m will be evicted from the cache (if the object is not again requested in the meantime), in other words the interval of time after which a new inserted content m will be evicted from the cache.[11] Che's approximation assumes $T_C(m)$ to be a *deterministic constant, independent* of the selected content m. This assumption has been taken from [6] as a theoritical assumption, where it is shown that, under a Zipf-like popularity distribution, the coefficient of variation of the random variable representing $T_C(m)$ tends to converge as the cache size grows. Furthermore, the dependence of the eviction time on m becomes negligible when the catalogue size is sufficiently large. Che's approximation is asymptotic validate for $\alpha > 1$. as provided in [23] More in detail, thanks to Che's approximation, we can claim that an object m is in the cache at time t, if and only if a time smaller than $T_C(m)$, has elapsed since the last request for object m, i.e., if at least one request for m has arrived in the interval $[t - T_C, t]$[11]. Under the assumption that requests for object m arrive according to a Poisson process of rate λ_m, the time-average probability $p_{in}(m)$ that object m is in the cache is then given by:

$$p_{in}(m) = 1 - e^{-\lambda_m T_C}[11]$$

As immediate consequence of PASTA property for Poisson arrivals, observe that $p_{in}(m)$ represents, by construction, also the hit probability $p_{hit}(m)$, i.e., the probability that a request for object m finds object m in the cache. The only unknown quantity in the above equality is T_C, which can be obtained with arbitrary precision by a fixed point procedure. The average hit probability of the cache is:

$$p_{hit} = \sum_m p_m p_{hit}(m)[11]$$

Both equations for $p_{in}(m)$ and p_{hit} are taken from [11] which are usuful just for the purpose of better understanding the behaviour of the average probability for object m in function of λ_m and cache eviction time T_C under the Che's approximation.

1.7 ccnSim: Simulation Environment

1.7.1 ccnsim Overview

CcnSim is a simulator for Content Centric Networks (CCN) [27], whose development started in the context of the Connect ANR Project. Written in C++, it is developed upon the Omnet++ framework, which provides all the APIs used to simulate Key Performance Indicators (KPI) of CCN networks, i.e., forwarding and caching replicement policies, cache decision strategies, content request (R), Catalog cardinality (M), Cache size(C) and so on[27]. CcnSim allows to perform classic *event-driven* (ED) simulations of large-scale CCN networks, i.e., up to $M = 10^9$ contents, with moderate memory occupancy and CPU time[27]. The last v0.4 version (distributed as free and open source software at [27]) provides, also, a new downscaling technique based on TTL caches *Modelgraft*[10],[29], which provide better performance in terms of memory occupancy and CPU time, thus enabling the simulation of growing CCN networks (i.e., up to $M = 10^{12}$ contents).

1.7.2 Omnet++ Engine and ccnSim workflow

Omnet++ is a C++ based event-driven framework used in networking simulation. It is characterized by: i) a set of core C++ classes, which can be extended in order to customize the simulated environment; ii) a simple network description language (ned) used to describe the interactions between modules; iii) a msg language defining messages exchanged between network nodes.
CcnSim includes a set of custom modules and classes that extend the Omnet++ core in order to simulate a CCN network[27]. A classic workflow for ccnSim is depicted in Fig. 1.2 and consists in:

- Compiling ccnSim source files, and linking them with the Omnet++ core.
- Writing *.ned* files which describe network topologies (it comprisis in creating connections between CCN nodes).
- Initializing the parameters of each module. This can be done either directly from the *.ned* files, or from the *.omnetpp.ini* initialization file.
  By varying the values of parameters,(i.e., cache size, user request, catalog cardinality, Zipf's α, average request rete λ, cache policies and decision strategies) it is possible to create several different CCN network topologies[27].
- Launching the simulation[27].

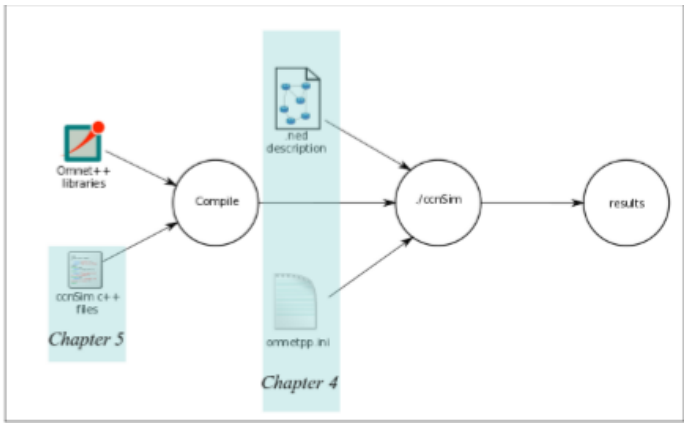
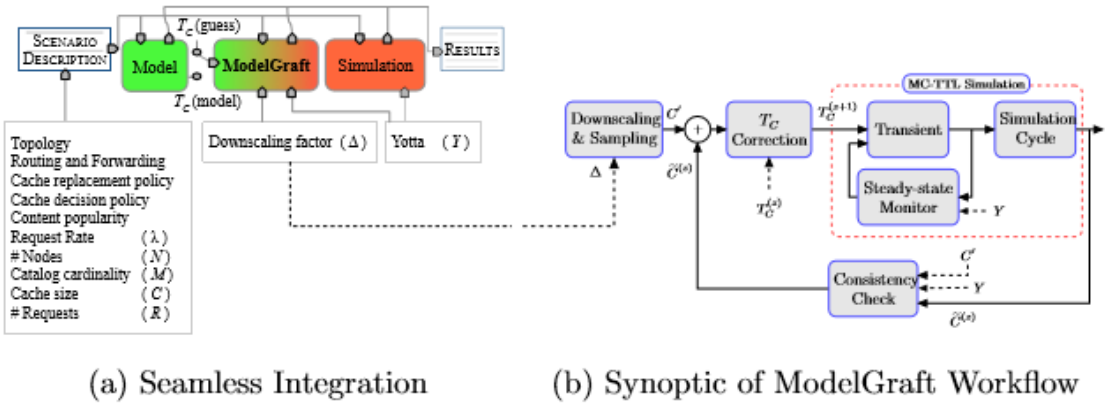


Figure 1.2. ccnSim workflow [27].



(a) Seamless Integration (b) Synoptic of ModelGraft Workflow

Figure 1.3. ccnSim-v.04 overview [27].

1.7.3 ccnSim Structure and Simuation techniques

CcnSim-v0.4 (the last version) provides users with the possibility of selecting the most suitable simulation technique, as reported in Fig. 1.3(a). Starting from a unique scenario description, users can analyze the performance of cache networks via either an analytical model (when available, left), a classic *event-driven* simulation engine (right), or via the *ModelGraft* [22], [23] engine (middle), where MonteCarlo simulations after being opportunely downscaled, systems are performed by replacing LRU caches by their Che's approximated version, implemented in practice as TTL caches.

CcnSim-0.4 now offers users a simulation framework through which they can select the simulation technique that according with their need.
In particular, users can select between:

- *Event-driven* (ED)
- *ModelGraft*

This work refers to *Event-driven* simulations, which is suitble for our purpose.

1.7.4 Downloading, installing and compiling ccnSim

ccnSim-v0.4 can be freely downloaded from [27]

Prerequisites

- **Omnet++**: (version $\geq$ 4.1) downloaded from [30].
  For this work has been used Omnet++ v5.0.
- **Boost libraries**: installed either by using the standard packet manager of your system (e.g., apt-get install, or downloading them from [44])

Installing

In order to install ccnSim-v0.4, it is necessary to patch Omnet++ before. CcnSim-v4.0 comes with two different set of patches, each correspondent to the installed version of Omnet++ (e.g., v4.x or v5.x).

The opeating system should be a Linux distribution; for compatibility of libraries has been choosed Linux Fedora 64 bit.

Compiling

The respective ccnSim compilation commands are:

```
pint:~$ cd $CCNSIM_DIR
pint:CCNSIM_DIR$ cp ./patch/omnet-5.0/ctopology.h $OMNET_DIR/include/omnetpp
pint:CCNSIM_DIR$ cp ./patch/omnet-5.0/ctopology.cc $OMNET_DIR/src/sim
pint:CCNSIM_DIR$ cd $OMNET_DIR && make && cd $CCNSIM_DIR
pint:CCNSIM_DIR$ ./scripts/makemake.sh
pint:CCNSIM_DIR$ make
```

where $\$OMNET\_DIR$ and $\$CCNSIM\_DIR$ are respectively Omnet-5.0 path and ccnSim path.

1.7.5 Statistics and Resuts

The key point of a simulation is that of collecting statistics and results. The dimensioning of the warm-up phase plays a relevant role when the aim is to collect statistics which are actually computed at steady-state during the transient period of the first MonteCarlo-TTL simulation cycle (with initial guesses for T_C that can not be necessarily correct because the feedback-loop control automatically converge to a correct T_C) in the case of TTL Modelgraft based on MonteCarlo simulative approach[22]; in real cases, the length of the transient can be affected by several parameters, like forwarding strategy (e.g., shorter paths under ideal NRR can reduce the transient with respect to shortest path [31]), or cache decision policy (e.g., Leave Copy Probabilistically (LCP)[31], where the content acceptance ratio is reduced with respect to Leave Copy Everywhere (LCE), is expected to achieve longer transient durations). In particular, the convergence of a single node i is effectively monitored using the Coefficient of Variation (CV) of the *measured hit probability ratio*, $p_{hit}(i)$, computed via a batch means approach[22][27]. The node i is considered to enter a steady-state regime when:

$$CV_i = \frac{\sqrt{\frac{1}{W-1}\sum_{j=1}^{W}(p_{hit}(j,i) - p_{hit}(i))^2}}{\frac{1}{W}\sum p_{hit}(j,i)} \leq \epsilon_{CV}[27] \tag{1.3}$$

where W is the size of the sample window, and ϵ_{CV} is a user-defined convergence threshold and denoting $p_{hit}(j,i)$ with the j-th sample and $p_{hit}(i) = \frac{\#_{hit}(i)+\#_{miss}(i)}{\#_{hit}(i)}$[27].
The eq. 1.1 is taken from [27] with the only purpose of better understand how is stringent for a node i enter to a self-stabilitation phase by respecting convergence constraints. The drowbacks for batch means are bias. To avoid biases, new samples are collected only if the cache has received a positive number of requests since the last request, and its state has changed, i.e., at least a new content had been inserted in the cache since the last insertion[27]. Eq. 1.1 must be extended ror all node in the cache to avoid unnecessarily slow down convergence of the whole network, often due to particular routing protocols and/or topologies. Therefore whole system to enter steady-state when[27]

$$CV_i \leq \epsilon_{CV}[27] \qquad \forall i \in Y \tag{1.4}$$

where Y is the *set of the first YN*

Chapter 2

Very-Large Scale Scenario for different policies: IRM vs SNM

2.1 Introduction

In this chapter I considered the largest scenario and in such case I investigated via *event-driven* simulation gathered via ccnSim[27]. The aim consists to evaluate the same network topology, first by injecting the traffic model in the simulation engine in the classical IRM when inter-arrival request times are independently, exponentially distributed, so that requests for object m are generated according to a homogeneous Poisson process[15] of rate λ_m, then, the same traffic modeled under SNM tecnique, since the IRM completely ignores all temporal correlations in the sequence of requests and does not take into account the characteristics of real traffic referred to as temporal locality, which means that if a content is requested at a given time, then it is more probably that the same content will be requested again in the near time after[45][49]. It is well-known that the temporal locality has a beneficial effects on the cache performance, since it increases the hit probability. [49]

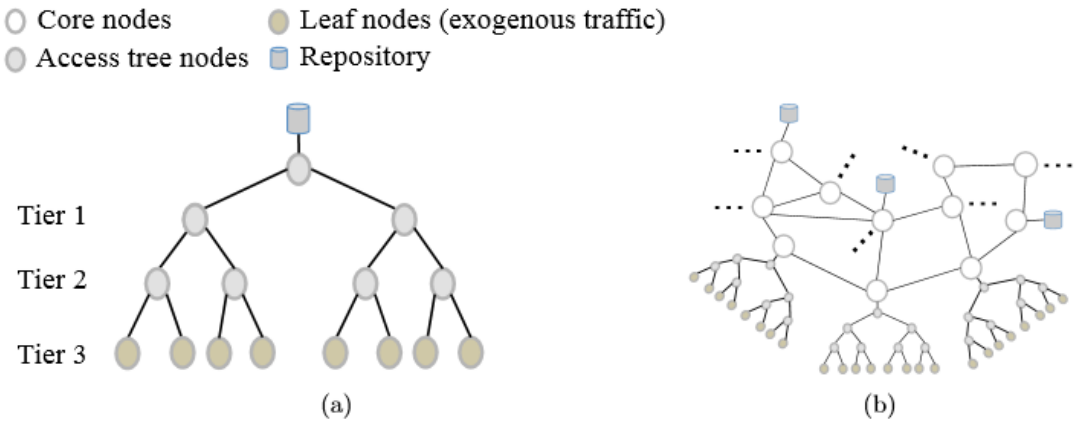


Figure 2.1. Network Topology. (a) 4-level binary tree, (b) CDN-like.[29]

2.2 IRM Validation

The considered scenario represents an ICN-access tree network[29], where the topology is a $N = 15$ nodes 4-level binary tree depicted in Fig. 2.1. A single repository, connected to the root node, stores a $M = 10^9$ objects catalog, where objects follow a Zipf popularity distribution with exponent α. In particular we vary the Zipf popularity distribution by changing the exponent $\alpha \in \{0.8, 1\}$, and the cache size of each node $C = \{100, 1000, 10000, 100000\}$. An overall $R = 10^9$ requests are injected at each leaf nodes with aggregate request rate for each client of $\lambda = 20$ req/s per leaf[29].

The goal of these simulations is showing the hit probability achieved by four different cache decision strategies, under IRM traffic, considered for the comparison: (i)LRU, (ii) LCE, where fetched contents are always cached in every traversed node; (iii) LCD, where the contents are cached only in the cache preceding the one when the fetching contents occur; (vi) 2-LRU, where pollution cache generated by unused contents is reduced by using an additional cache in front of the main one, with the purpose of caching only the requested contents: the fetched contents will be stored in the main cache only in case of a hit event in the Name cache, however the cache preceding the main one [11]. (v) (a) q-LRU, $q = (1/10)$, that probabilistically admits contents in the cache (configured so that one over ten fetched contents are cached on average); and (b) q-LRU, $q = (1/100)$ (where one over hundred fetched content are cached on average);

We note that LCD significantly outperforms LCE, thanks to an improved filtering effect since LCD can be view as the dual of k-LRU for cache networks. However LCE replication policy does not exploit, as it better, the comprehensive storage capacity in the network trying to avoid the concurrent placement of the content within all cache in case of tree-like network topology[11]. This is the reason why is preferible engage LCP policy fed by incoming requests uniformly distributed, (e.g., with either $q = 0.5$ or $q = 0.25$).
A single meta-cache (2-LRU policy) achieves very high benefits, providing performance very close to optimal. It significantly outperforms LRU and its probabilistic version (q-LRU) thanks to the fact that the meta-cache behaves as a filter, since the insertion policy plays a crucial role in cache performance, especially when we deal with small size cache networks[11]. We observe, also, that q-LRU with $q = 0.01$ performs better then q-LRU with $q = 0.1$, this beacause when $q = 0.01$, there is more probability to find an object present in the cache node, since the insertion policy of new objects is more stringent, this means that an objects already stored in the cache stays longer time, however the probability to find it (i.e., hit event) bacomes higher with respect to $q = 0.1$.

In small-cache regime, defferences among different caching of policies become significants, in this case, we observe that insertion policies providing some protection against unpopular objects largely outperform policies which do not filter any request [11]. Performance comparisons among caching strategies for different α, under IRM traffic model, are shown in Fig. 2.2 and Fig. 2.3.

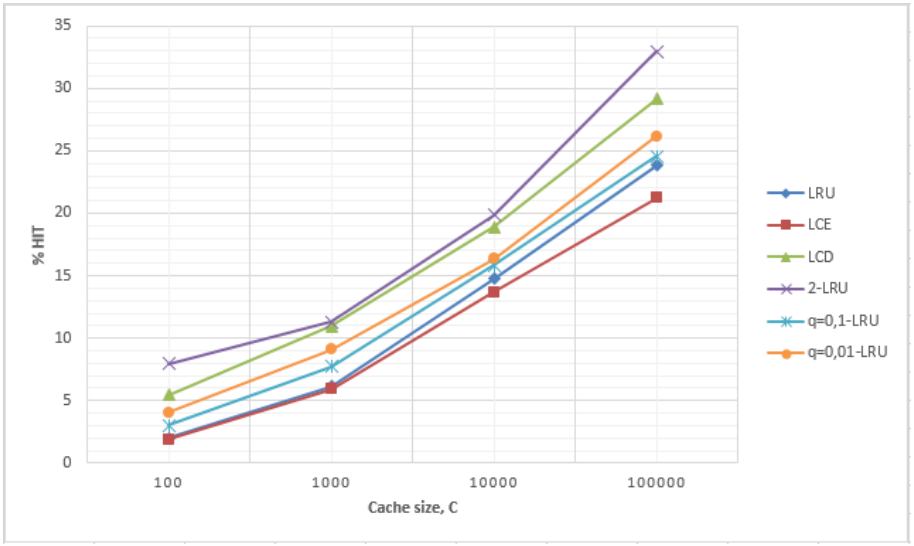


Figure 2.2. Hit probability vs cache size, for various poicies, under IRM, in the case of $\alpha = 0.8$.

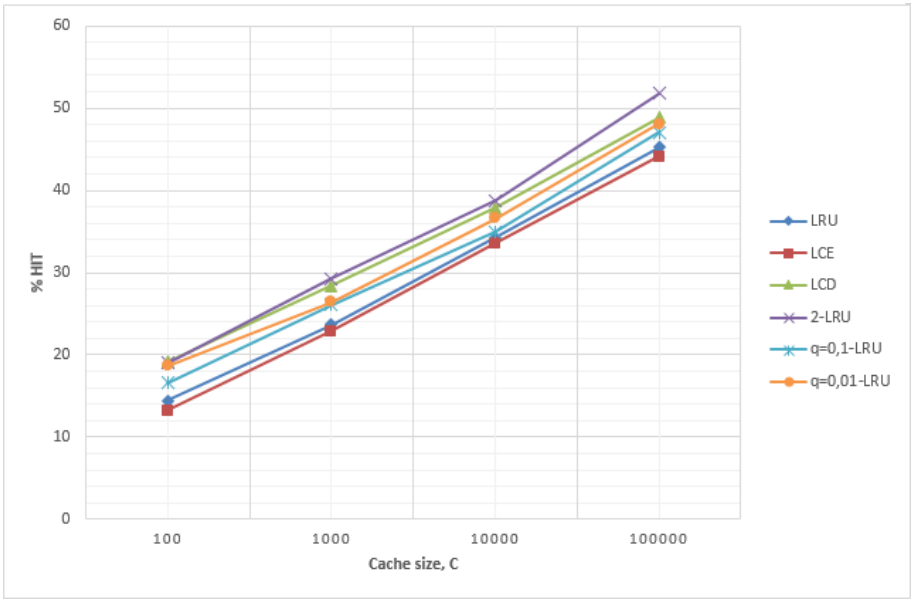


Figure 2.3. Hit probability vs cache size, for various poicies, under IRM, in the case of $\alpha = 1$.

2.3 SNM Validation

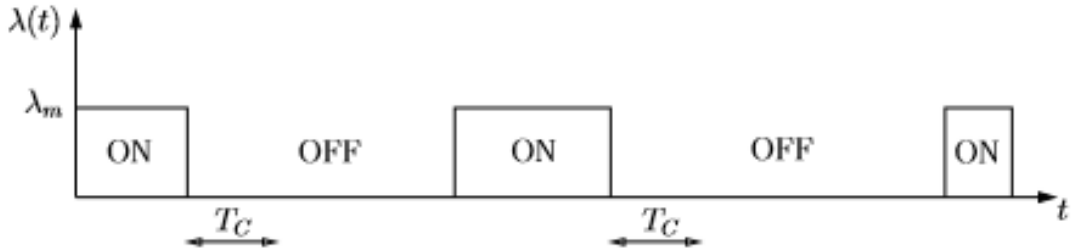


Figure 2.4. SNM modulated Poisson process of requests for a given content m. [15].

The basic idea is to capture the impact of dynamic contents (i.e., contents start to be available in the system at a given time, and their popularity evolves according to a certain profile), by using a stationary, ON-OFF traffic model associated to a fixed catalogue of M contents.[15] For the sake of simplicity, we assume that new contents become available in the system according to a homogeneous Poisson process of rate λ_m. We refer to this model as Shot Noise Model (SNM), since the overall process of requests arrival is known as a Poisson shot noise process[14] as depicted in Fig. 2.4 which shows the SNM modulated homogeneous Poisson process describing the arrival process of requests for a given content m of a chosen fixed catalogue[15].
Contents are divided into classes characterized by a set of parameters.
First of all, a life-span (T_{on}) is associated to each class, indicating the time during which the respective objects are active and can be retrieved. At the same time, each class has its own period of inactivity (T_{off}), during which the respective objects cannot be fetched.[27] We assume that both ON and OFF are exponentially distributed with *mean* duration of T_{on} and T_{off}. During an ON period, requests arrive with constant intensisty λ_m and each class is, then, characterized by an average total request rate ($E[V]$), which expresses the total number of requests per seconds generated for that class during the ON period, and is given by $E[V] = \lambda_m T_{on}$[15][14]. The contents placed inside each class are also characterized by a popularity distribution that influences the generation of content requests λ. Another crucial parameter is k, intrinsecally connected to the ON-OFF process and the actitivity/inactivity period, indeed $T_{off} = k \times T_{on}$.
The investigated network scenario is the same used for IRM model: 4-level binary tree (see Sec. 2.2). I simulated 10^9 content requests, average total request rate per second $E[V] = 20$, and probability distribution with Zipf's exponent $\alpha = 1$ to generate content requests. For the sake of semplicity I evaluated all policies performance with just one class, by varying T_{on} and T_{off} periods with fixed number of contents; this means that for each period T_{on}, I varied T_{off} period by acting on the moltiplicator factor k, within the equation $T_{off} = k \times T_{on}$.
Values of T_{off} and T_{on} with the corresponding number of contents are provided in Tab. 2.5 and Tab 2.6.

Ton1(s)	Toff=k*Ton1(s)	Content
43200	43200 (k=1)	62500
43200	345600 (k=8)	62500

Figure 2.5. Settings for SNM scenario with T_{on1}.

Ton2=50*Ton1(s)	Toff=k*Ton2(s)	Content
2160000	2160000 (k=1)	62500
2160000	17280000 (k=8)	62500

Figure 2.6. Settings for SNM scenario with T_{on2}.

Under SNM traffic, as IRM model, 2-LRU strategy performs very good, also in the presence of strong temporal locality. This because, 2-LRU, filter out unpopolar content thus its insertion policy is able to adapt fast to popularity variations introduced by short-term temporal locality[11]. In this case, the eviction policy, (in contrast to insertion policy), is not significant. When the caches are too small with respect to catolog size, the eviction policy becames pratically negligible[15]. Hence, 2-LRU performs, dramatically better than q-LRU, since its filtering action is more effective and selective, especially against small q (i.e. $q = 0.01$); this fact goes in contrast to what we have seen under IRM, where small q performed better then larger q (i.e. $q = 0.1$): q-LRU with very small q tends to behave like LFU, hence does not get advantage from the temporal locality when a process is requested[11]. 2-LRU and q-LRU outperforms LRU, in presence of dynamic variation of content, for small cache sizes, because their filter action cut unpopular contents, by exploiting the limited portion of cache. Whereas, for increasing values of sizes, the presence of filter limits the objects insertion, by leading to a worse hit probability performance with respect to a single LRU.[15]
Furthermore we observe that the gain achieved by q-LRU with respect to LRU is significantly higher, this beacause q-LRU policy at each cache is equivalent to use the LCP (Leave Copy Probabilistic) strategy in an network of LRU caches, hence, probabilistic insertion policy allows to better exploit the aggregate storage capacity of the system, by avoiding the simultaneous placement of a given content in all caches along the network route[15][19].
At last we can even observe that by incresing T_{off} period (through the k moltiplicator factor), decrease the number of concurrent number of active content at the same time, thus the hit performance results better, as showed in Fig. 2.7 and Fig. 2.8 reporting the hit probabilities for various policies in case of T_{on1} and different T_{off} periods, under SNM traffic model.

Then, we performed the same analysis with T_{on2} grater than T_{on1}, computed as $T_{on2} = 50 \times T_{on1}$. We note that for T_{on2} the performance are worse than those one simulated with T_{on1} because the activity period T_{on}, where simultaneous contents can be requested at same time, become larger. Performance for various policies, in case of T_{on2}, and different T_{off} periods, under SNM traffic model, are shown in Fig. 2.9 and Fig. 2.10.

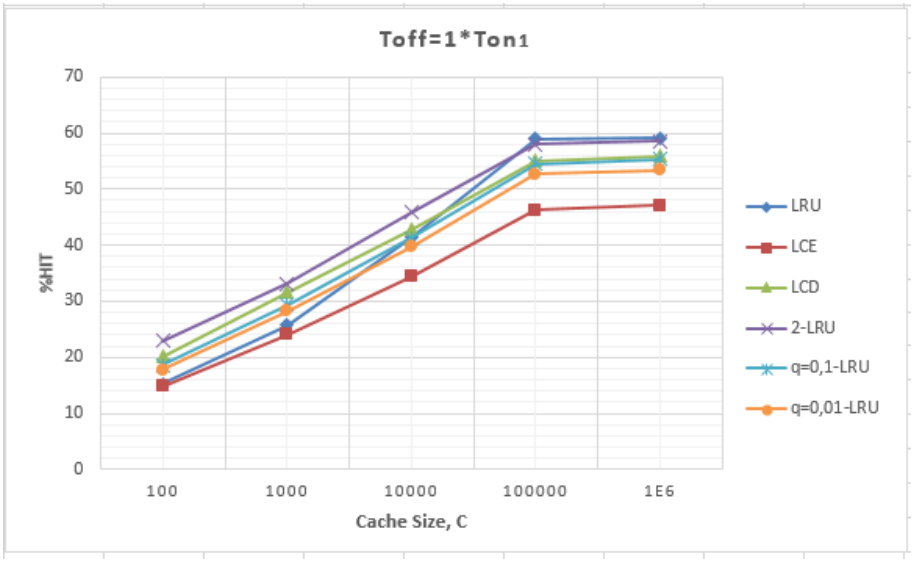


Figure 2.7. Hit probability vs cache size, for various policies, under SNM, for $T_{off} = T_{on1}$.

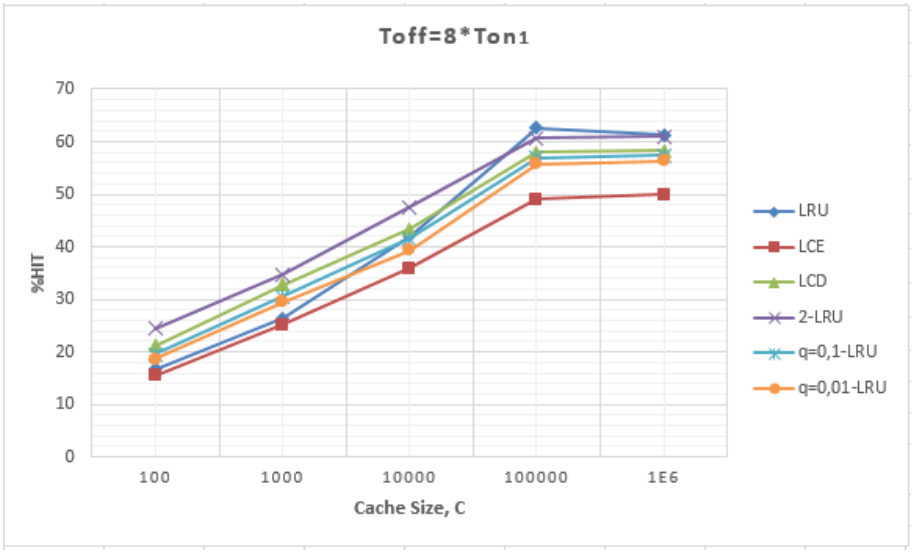


Figure 2.8. Hit probability vs cache size, for various policies, under SNM, for $T_{off} = 8 \times T_{on1}$.

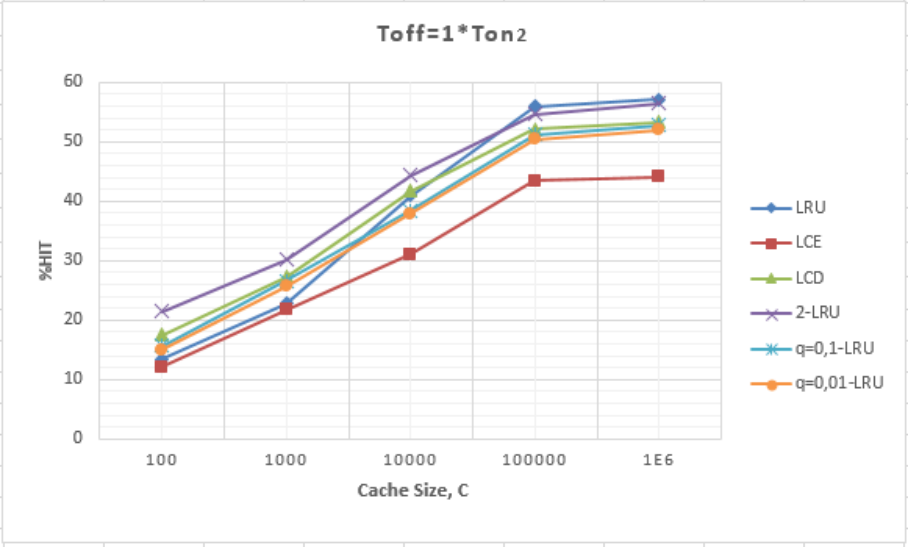


Figure 2.9. Hit probability vs cache size, for various policies, under SNM, for different $T_{off} = T_{on2}$.

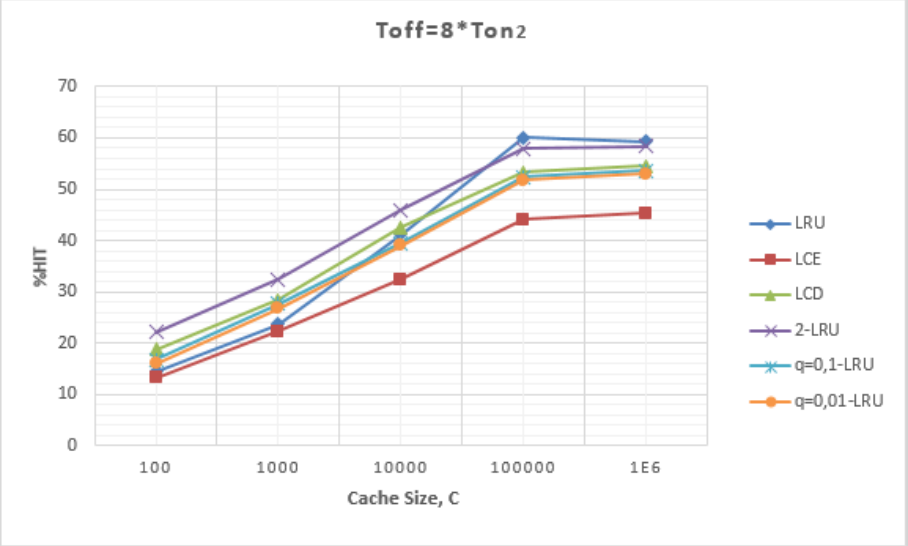


Figure 2.10. Hit probability vs cache size, for various policies, under SNM, for different $T_{off} = 8 \times T_{on2}$

.

2.4 IRM vs SNM Validation

Despite in SNM dynamic content scenario, is hard to estimate cache hit probability of policies, especially for policies different from LRU and, in particular, for cache network is very challanging to predict results close to real scenarios, IRM models which do not take into account the temporal locality of contents, performance are more pessimistic (i.e hit ratio). About SNM traffic model we achieve very good performance with $T_{off} \gg \mathrm{T}_{on}$, indeed if we set too large T_{on} life-span we achieve performance very close to IRM traffic model.
Performance comparison for various policies, under IRM vs SNM, in case of large T_{off}[1] and $\alpha = 1$, are plotted in Fig. 2.11 for T_{on1} and in Fig. 2.12 for T_{on2}.

[1]The choice of large T_{off} has the aim of better exploit the benefits of SNM traffic against IRM traffic.

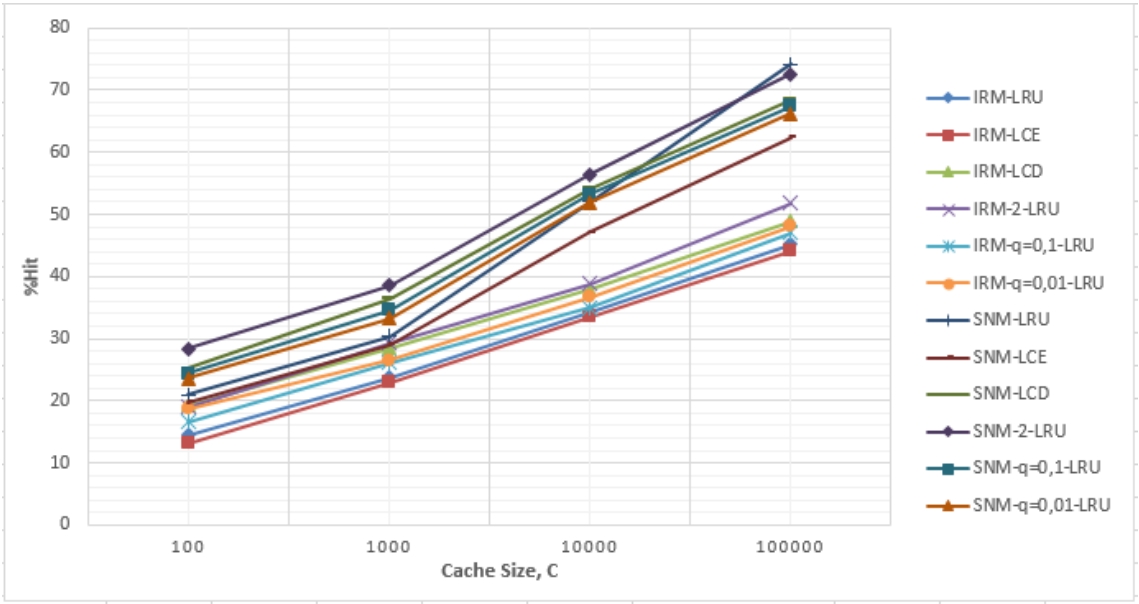


Figure 2.11. Hit probability vs cache size, for various poicies, under IRM vs SNM, in the case of $T_{off} = 8 \times T_{on1}$.

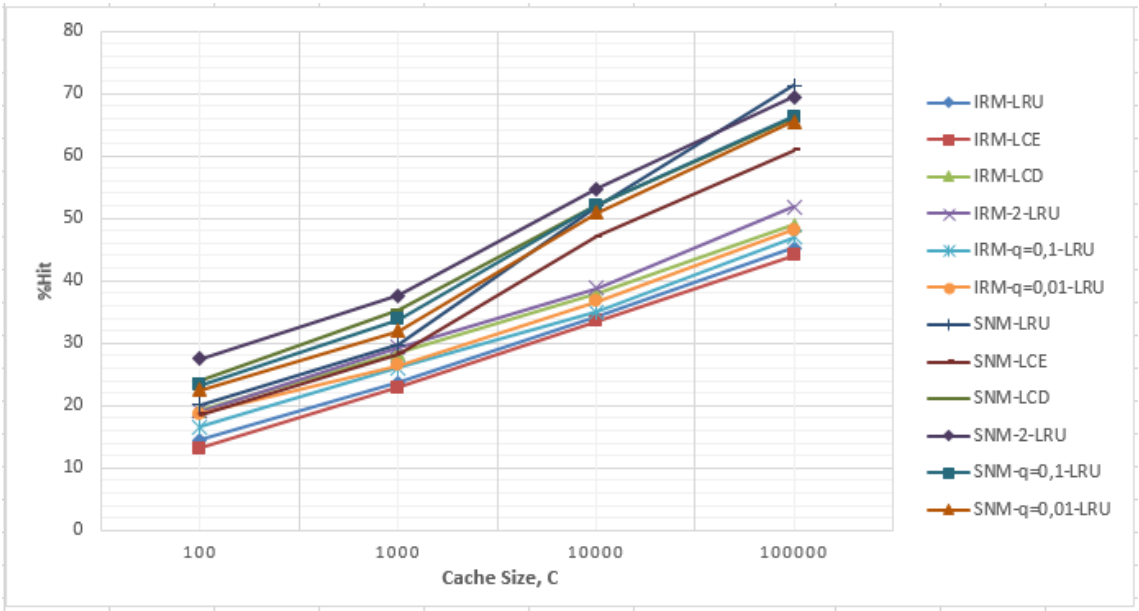


Figure 2.12. Hit probability vs cache size, for various poicies, under IRM vs SNM, in the case of $T_{off} = 8 \times T_{on2}$.

Chapter 3

Performance evaluation of "cache-decoupled" 2-LRU vs k-LRU, under IRM

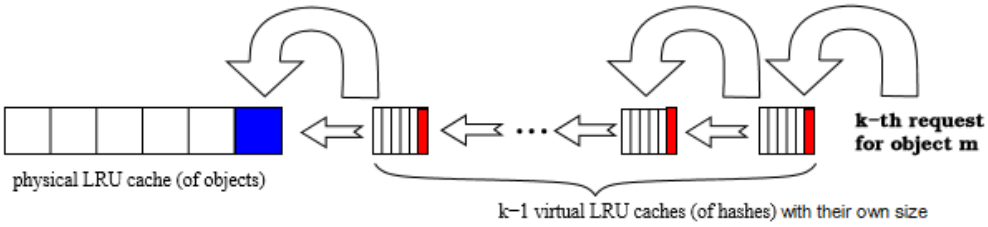


Figure 3.1. An illustration of k-LRU "cache-decoupled" policy.[11]

3.1 Introduction

So far we have worked with 2-LRU having both second stage (meta-cache) and first stage (Name Cache) the same cache size, this is a limit within ccnSim[27] implementation of 2-LRU policy. In this chapter I modified the ccnSim C++ program in order to simulate 2-LRU strategy with main cache and Name Cache decoupled, meaning that the two cache can have different size values. This is an optimization that leads to obtain better performance without placing other Name Cache in tail. A list of simulations reported in section 3.3 highlight this fact. Later I implemented k-LRU, not present among the ccnSim policies, directly in a way you can assign different sizes to all k-1 caches (see Fig. 3.1).

3.2 2-LRU "cache-decoupled"

First of all we observe that the *cache-decoupling* breaks the independence assumption between cache 2 and meta-cache 1 considered in Sec 1.4, where in the case when the two caches have the same size the eviction time of the second cache (T_C^2) is significantly larger than the eviction time of meta-cache (T_C^1), since the ID is discarded by the meta-cache 1 before the correspondent content is evicted by the cache 2, making it possible to find the content in cache 2, and not in cache 1[11]. Therefore we should guarantee that $T_C^1 \geq T_C^2$ does not occur,

I provided the main C++ parts of ccnSim program, added in order to develop such optimization of 2-LRU, in particular all functions, variables and pointers acting on the first stage Name Cache, furthermore I provided the part of program realizing 2-LRU class called "Two_Lru" inside the "two_lru_policy.h" header, necessary to create a new LRU acting as a Name Cache able to set its own size; 2-LRU indeed requires node to allocate a second cache, Name Cache, (always with LRU replacement), in order to keep track of the IDs of the received interest packets. In case of HIT inside the Name Cache, the retrieved Data packet will be cached in the normal cache (i.e. the one that contains real contents); otherwise, it will be just forwarded back.

3.2.1 2-LRU "cache-decoupled" implementation

In "base_cache.cc" source file:
inside "base_cache::initialize()" I add, for 2-LRU, an else-if branch to set the size of the Name Cache (NC):

```
else if (decision_policy.compare("two_lru")==0)
{
              name_cache_size = par("NC");
              decisor = new Two_Lru(name_cache_size);
}
```

Functions

I created two functions for storing and lookup content id for 2-LRU Name Cache.

- Storage handling of the recivied content ID inside the name cache for 2-LRU meta-caching:

```
void base_cache::store_name(chunk_t elem)
{
       if (cache_size == 0)
                     std:: stringstream ermsg; // The size of name cache
                         is set to 0!
                     severe_error(_FILE_,_LINE_,ermsg.str().c_str() );
}
data_store(elem); // Store the content ID inside Name Cache.
```

- Lookup function to lookup content ID inside the Name Cache.

```
bool base_cache::lookup_name(chunk_t chunk)
{
       bool found = false;
       if (data_lookup(chunk)) // The content ID is present inside the
           Name Cache.
              found = true;
```

```
            else
                    found = false;
            return found;
    }
```

In "core_layer.cc" source file:
inside "base_cache::initialize()" function I added a pointer for 2-LRU class and a *flag* for cacheble if the ID content is present inside Name Cache:

```
if (decision_policy.compare("two_lru")==0) // 2-LRU
{
        Two_Lru* tLruPointer = dynamic_cast<Two_Lru *>
            (ContentStore->get_decisor());
        if(!(tLruPointer->name_to_cache(chunk))) // The id is not present
            inside name cache

               cacheble = false; // cacheble flag will be set to '0'.
}
```

In "base_cache.h" source file:
I defined *storing* and *lookup* functions for Name Cache and an integer variable for its size:

functions:

```
void store_name(chunk_t);
bool lookup_name(chunk_t);
```

variable:

```
int name_cache_size;
```

In "two_lru_policy.h" source file:
I add in "Two_Lru" class a pointer to Name Cache allowing to set its own size since "Two_Lru" class is in charge to take cache decision of the 2-LRU. The "data_to_cache" function checks the presence of the content ID inside the Name Cache, and eventually stores it. Since the *flag* that indicates the decision (cache or not) is present inside the entry, already been set by the core_layer, such function returns always true.
The rest of "Two_Lru" class makes the hit/miss operations for Name Cache, by inserting/ereasing elements inside map.
The complete program implementing "Two_Lru" class can be consulted in Appendix A.

Below is provided the part of "Two_Lru" class allowing the Name Cache pointer to set its own size:

```
public:
        Two_Lru(uint32_t cSize):ncSize(cSize){
                base_cache* bcPointer = new lru_cache(); // Create a new LRU
                    cache that will act as a Name Cache.
                name_cache = dynamic_cast<lru_cache *> (bcPointer);
                name_cache->set_size(ncSize);} // Set the size of the Name
                    Cache.
```

```
virtual bool data_to_cache(ccn_data *)
{return true;}
```

In order to add the Name Cache size within ".ini" file, inside "core_layer.ned" file I inserted an NC variable as integer:

```
moduleinterface cache
{
       parameters:
       int C;
       int NC;
       gates:
       inout cache_port;
}
```

3.3 Validation of 2-LRU "cache-decoupled", under IRM

In this section I validated, according *event-driven* technique, the 2-LRU "cache-decoupled" through a list of simulations, under IRM traffic. The validation scenario is an ICN-access tree network where the topolgy is a $N = 15$ nodes 4-level binary tree depicted in Fig. 2.1 [1]. This version is an optimization of 2-LRU policy where both second stage (meta-cache) and first stage (Name Cache) can have its own size. In order to validate that version of 2-LRU strategy, I fixed the size of second stage, by varying the size of the first stage from a minimum of 10% to a maximum of 500% of the second stage meta-cache, for two values of Zipf's exponent: $\alpha = \{0.8, 1\}$.

Small cache sizes

In the case of very small cache sizes ,i.e., 10^3, we achieved a poor *gain*, as we expected, since the policy didn't obtain good perfermance in terms of *hit*, indeed by tuning the Name Cache to larger sizes we exceded 11.4% with $\alpha = 0.8$ as showed in Fig. 3.2, and we didn't reach 30% with $\alpha = 1$ as showed in Fig. 3.3. With respect to the "original" 2-LRU[2]policy we obtained a pratically negligible *gain*, (i.e, less than 1%) with $\alpha = 0.8$, and slightly better with $\alpha = 1$, (i.e, $> 1.5\%$); so the *cache-decoupling* didn't lead significant benefits.
In the case of small cache sizes, i.e., 10^4, we obtained results similar to the case wh second stage had size equal to 10^3, by confirming that working with small sizes 2-LRU performs not at its better, although compared with other policies, performs very well. In terms of *hit*, we achieved maximum values over 20% with $\alpha = 0.8$ as plotted in Fig. 3.4 and just a little over 39% with $\alpha = 1$ as plotted in Fig. 3.5. In terms of *gain* we obtained *overall* only 2%, because Name Cache sizes too small didn't influence significantly the full performance of such policy.

[1] see Chapter 2, Sec. 2.2, for all parameters specification
[2] ...it means that the 2 caches have the same size

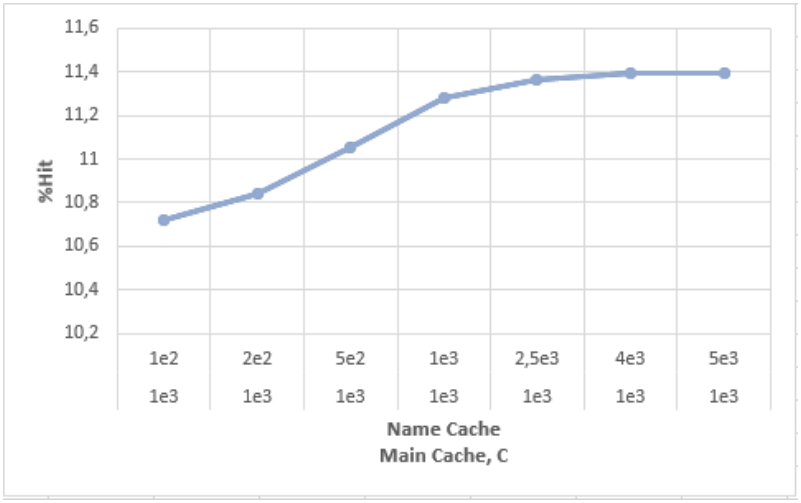


Figure 3.2. Hit probability vs very small cache size for 2-LRU "cache-decoupled", under IRM and $\alpha = 0.8$.

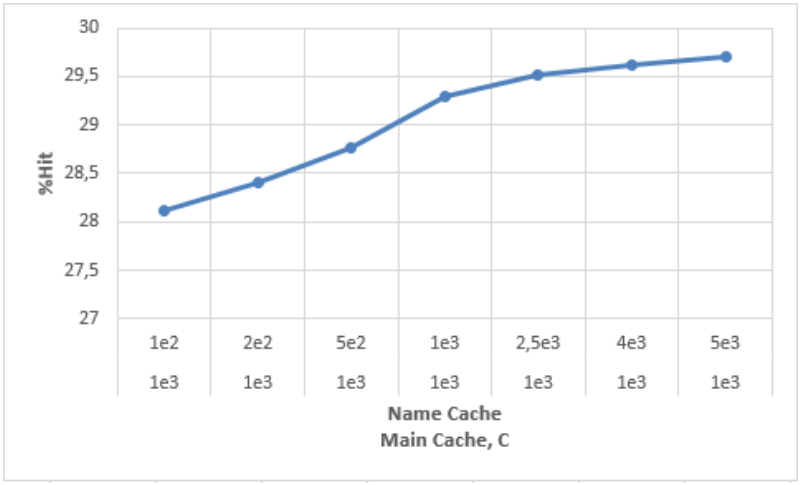


Figure 3.3. Hit probability vs very small cache size for 2-LRU "cache-decoupled", under IRM and $\alpha = 1$.

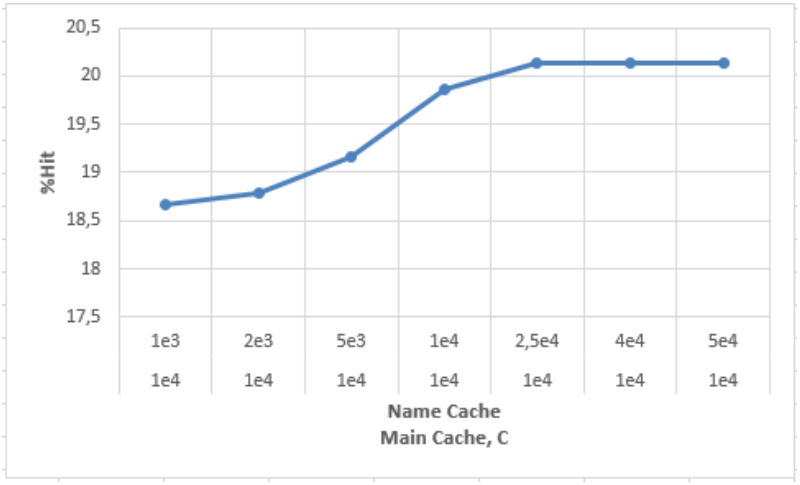


Figure 3.4. Hit probability vs small cache size for 2-LRU "cache-decoupled", under IRM and $\alpha = 0.8$.

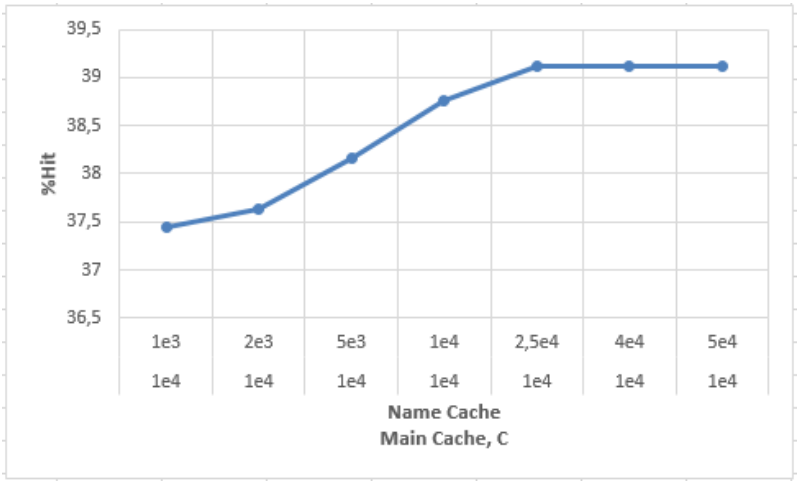


Figure 3.5. Hit probability vs small cache size for 2-LRU "cache-decoupled", under IRM and $\alpha = 1$.

Large cache sizes

Now we analyze the case of large size values of second stage known even as meta-cache or main cache, i.e., 10^5 and we evaluated our *optimizited* 2-LRU in terms of *hit* and *gain*. Here we obtained good performance due to an increased size of the main cache, which is able to store more objects, in case of miss in the Name Cache.
In this case such policy achieved a *total gain* of about 10% for $\alpha = 0.8$ and lesser for $\alpha = 1$, hence, our optimization worked very well beacuse for large cache size the presence of the Name Cache influenced significantly on the overall performance impact. In terms of *hit*, we got acceptable results in phase of simulation, by verify that for values of $\alpha < 1$ (i.e., $\alpha = 0.8$ in our case of study), the maximum hit achieved is between 30% and 40%, whereas with $\alpha \geq 1$, the maximum hit is $\approx$ 50%, thus 1/2 of the ideal hit. Fig. 3.6 and Fig. 3.7 show the performance got for large cache size with different Zipf's exponent α.
By increasing the size of the main cache to a power of 10, i.e, 10^6, we obtained optimal performance, however our optimization introduced dramatic benefits on the *overall gain*, but for cache sizes too large, the *cache-decoupling* optimization didn't provide significant *gain* from the "original" 2-LRU, since such policy is already designed to achieve optimal performance in a very large cache size scenarios.
In this case the content popularity profile plays a crucial role, indeed high values of α, (e.g., $\alpha = 1$), allow to reach the maximum performance that 2-LRU strategy can achieve as showed in Fig 3.9, (the curve stabilized to 51.77%). Although large values of α leaded to better performance in terms of *hit*, they didn't produce huge advantages in terms of *gain*, since in our examinated case we perceived a total *hit gain* of about 3% with $\alpha = 1$, with respect to about 10% in the case of smaller value of α, as depicted in Fig. 3.8, in which we simulated the behavior with $alpha = 0.8$.
This result tell us that the our improvement of the *cache-decoupling* works better with smaller values of α, exactly $\alpha < 1$, whereas higher α values lead the better performance, thus requiring a trade-off, between cache size and α in order to exploit, at its best, such optimization, trying to get optimal performance at the same time.

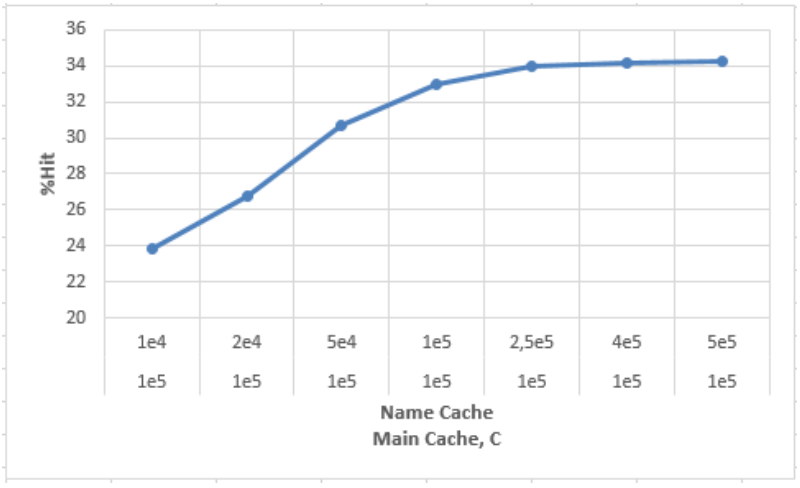


Figure 3.6. Hit probability vs large cache size for 2-LRU "cache-decoupled", under IRM, and $\alpha = 0.8$.

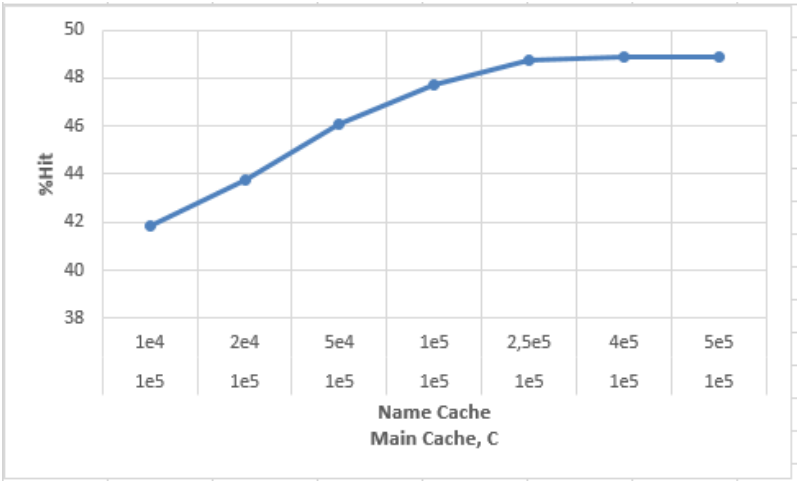


Figure 3.7. Hit probability vs large cache size for 2-LRU "cache-decoupled", under IRM, and $\alpha = 1$.

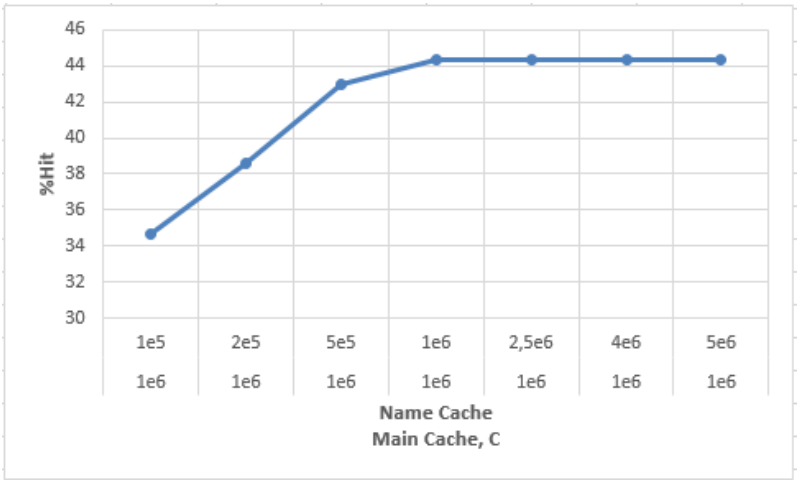


Figure 3.8. Hit probability vs very large cache size for 2-LRU "cache-decoupled", under IRM, and $\alpha = 0.8$.

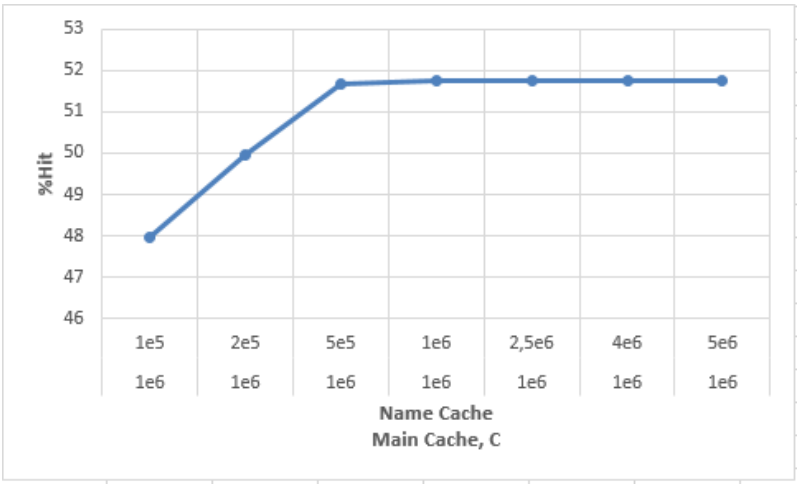


Figure 3.9. Hit probability vs very large cache size for 2-LRU "cache-decoupled", under IRM, and $\alpha = 1$.

3.4 K-LRU "cache-decoupled"

As for 2-LRU "cache-decoupled" case, we observe that the *cache-decoupling* breaks the independence assumption between cache k and k-1 considered in Sec 1.3 [3], where in the case when the k caches have the same size the eviction time of the k-th cache (T_C^k) is significantly larger than the eviction time of k-1 caches (T_C^{k-1}), since the ID is discarded by the k-1 cache before the correspondent content is evicted by the k-th cache, making it possible to find the content in the k-th cache, and not in k-1 cache [11]. Therefore we should guarantee that $T_C^{k-1} \geq T_C^k$ does not occur.
Regarding k-LRU, as 2-LRU, I provided the main C++ parts of ccnSim program, implementing the policy as an extention of 2-LRU "cache-decoupled". Here the basic idea consists to create a unque cass called "k_LRU", inside "k_lru_policy.h" header, that defines, according to k levels, the numbers of new LRU cache to instantiate (k-LRU differs to 2-LRU just for the number of Name Cache placed before the main cache, where contents are stored). The eviction policy is therefore the same of 2-LRU.

3.4.1 K-LRU "cache-decoupled" implementation

In "base_cache.cc" source file:
inside "base_cache::initialize() I add for k-LRU an else-if branch to set the size of the k-1 Name Caches (NC,NC2,...,NCk):

```
else if (decision_policy.compare("k_lru")==0)
{
              name_cache_size = par("NC");
              decisor = new k_Lru(name_cache_size);
}
```

In "core_layer.cc" source file:
inside "base_cache::initialize()" function I implemented a new interest lookup according to k levels, I set a *flag* that allows to abilitate/disabilitate the caching of retrieved content:

```
if (decision_policy.compare("k_lru")==0) // k-LRU
{
       K_Lru* tLruPointer = dynamic_cast<K_Lru *>
           (ContentStore->get_decisor());
       if(!(tLruPointer->name_to_cache(chunk))) // The id is not present
           inside Name Cache
       // It has to be implemented for the k-1 levels
              cacheble = false; // cacheble flag will be set to '0'.
}
```

In "k_lru_policy.h" source file: I created a new "K_Lru" class where cache decision of the k-LRU are taken. For this purpose k-1 pointers to k-1 Name Caches have to be implemented in added to the pointer reserved to 2-LRU.
Below is provided the part of "K_Lru" class where the k Name Cache pointers (inclused 2-LRU) set their own size. The rest of K_Lru class makes the hit/miss operations for Name Cache, by inserting/ereasing elements inside map.
The complete program of K_Lru class can be consulted in Appendix B.

[3] See k-LRU as replacement policy

```
public:
       K_Lru(uint32_t cSize):ncSize(cSize){
              base_cache* bcPointer = new lru_cache(); // Create a new LRU
                  cache that will act as a Name Cache.
              name_cache = dynamic_cast<lru_cache *> (bcPointer);
              name_cache->set_size(ncSize);} // Set the size of the Name
                  Cache.
              // This have to be implemented for k-1 levels of Name Cache
              base_cache* bckPointer = new lru_cache(); // Create k-1 new LRU
                  cache that will act as a k-1 Name Cache.
              namek_cache = dynamic_cast<lru_cache *> (bcPointer);
              namek_cache->set_size(ncSize);} // Set the size of the k-th
                  Name Cache.

              virtual bool data_to_cache(ccn_data *)
              {return true;}
```

As a conseguence even the part in charge to check the presence of content ID inside the k Name Caches, by storing it eventually, changes according to "namek_cache" pointer.

All the functions and variables created for 2-LRU, are still valid for k-LRU, and the new policy does not change the the access way to single nodes.
In order to add the k Name Cache sizes within ".ini" file, inside "core_layer.ned" file I added a new kNC integer variable able to set the value of k NC caches:

```
moduleinterface cache
{
       parameters:
       int C;
    int kNC; // It will contain k Name Cache (included 2-LRU)
       gates:
       inout cache_port;
}
```

3.5 Validation of k-LRU "cache-decoupled", under IRM

3.5.1 3-LRU vs 2-LRU

In this section I validated the k-LRU policy implemented in ccnSim, according *event-driven* tecnique, through a list of simulations, under IRM traffic. The validation scenario is an ICN-access tree network where the topolgy is a $N = 15$ nodes 4-level binary tree depicted in Fig. 2.1. [4] For the sake of simplicity I evaluated only the performance of 3-LRU (i.e. k-LRU, with $k = 3$), against 2-LRU "cache-decoupled" optimized, described in Sec. 3.2, with the same criterion of 2-LRU, thus by varying the size of each Name Cache of both policies, from a minimum of 10% to a maximum of 500% of the main cache and for $\alpha = \{0.8, 1\}$. We note that in this case main cache is no longer the *second* stage, but becomes the *third* stage, since a new LRU acting as Name Cache has been added in tail.

3.5.2 Small cache sizes

In the case of very small cache size our improvement of 3-LRU provided a significant increase of *gain*, with respect to optimized 2-LRU, especially with small values of α, but it didn't achieve better performance than the maximum *hit* already obtained by 2-LRU, because the impact of the added LRU Name Cache is probabilistically low as showed in Fig. 3.10. Hence, in terms of *hit*, better performance was achieved with $\alpha = 1$, and the maximum *gain* obtained by 3-LRU against 2-LRU, is verified when both Name Caches were evaluated between 10% and 50% of main cache, by reaching a gain of 0.7% in case when Name Cache 1 and Name Cache 2 were 50% of the main cache size as showed in Fig. 3.11.
When the power size of main cache was increased by a factor of 10, i.e., 10^4, we can see how the performance of 3-LRU were improved in terms of *hits*, by showing an *hit gain* similar to the case of smaller size. By analyzing the case of $\alpha < 1$ (Fig. 3.12), we can appreciate that the *gain* obtained by 3-LRU vs 2-LRU is larger than the one obtained by $\alpha = 1$ (Fig 3.13). This results are compliant to what we have seen under 2-LRU.
Furthermore I would *emphasize* that small cache sizes got a good, but not optimal performance in terms of *hit* probability, stabilizing under 40% when $\alpha = 1$, however the percentege *gain* between 3-LRU and 2-LRU results more evident, thus 3-LRU performs significantly better than 2-LRU in small cache size scenarios. The major advantage of adding a new LRU when we work with small cache sizes is that 3-LRU policy, with respect to 2-LRU, obtains the maximum *hit* percentage at least when both Name Caches have the same size, referred in the case of "orginal" policy.

[4] see Chapter 2, Sec. 2.2, for all parameters specification

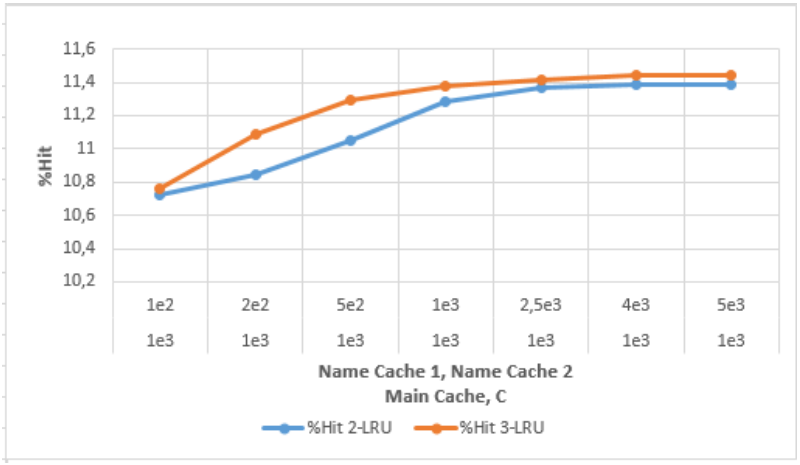


Figure 3.10. Hit probability vs very small cache size for 3-LRU vs 2-LRU "cache-decoupled", under IRM and $\alpha = 0.8$.

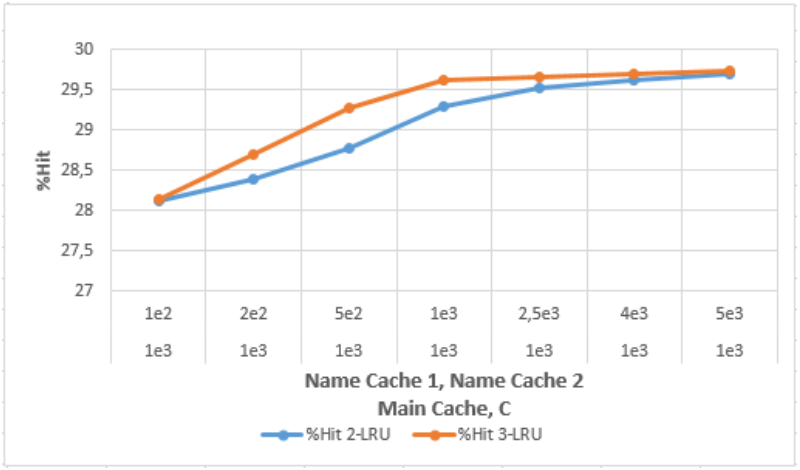


Figure 3.11. Hit probability vs very small cache size for 3-LRU vs 2-LRU "cache-decoupled", under IRM and $\alpha = 1$.

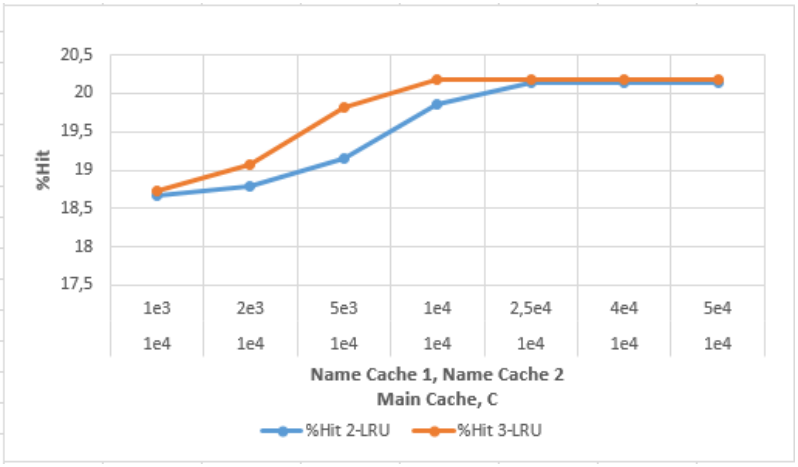


Figure 3.12. Hit probability vs small cache size for 3-LRU vs 2-LRU "cache-decoupled", under IRM and $\alpha = 0.8$.

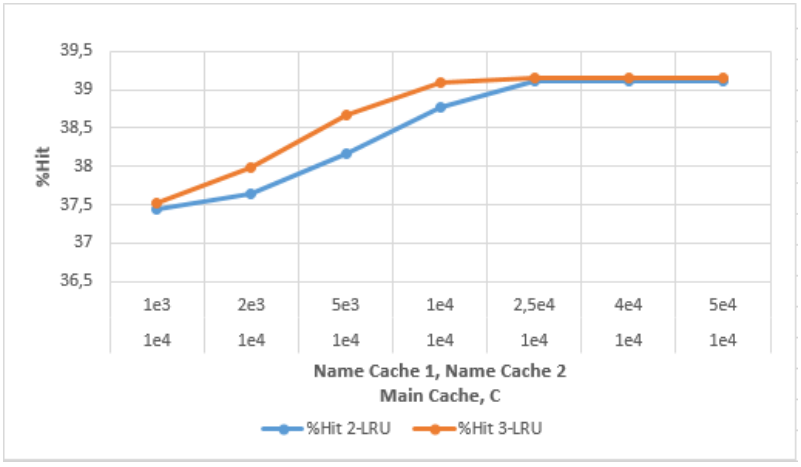


Figure 3.13. Hit probability vs small cache size for 3-LRU vs 2-LRU "cache-decoupled", under IRM and $\alpha = 1$.

3.5.3 Large cache sizes

When we work with large cache sizes, adding a new LRU Name Cache doesn't lead significant benefits, in terms of hit *gain*, than to 2-LRU policy, since the *cache-decoupling* optimization allows to increase directly the sizes of Name Caches until the same maximum hit is reached for both policies: this means that our improvement works very well on 2-LRU, but on 3-LRU policy doesn't provide huge advantages.
In this case α plays a crucial role in terms of *hit*, since 3-LRU evaluated with $\alpha = 1$ (Fig 3.15) performed dramatically better than to the case in which $\alpha = 0.8$ (Fig. 3.14).
Handling very high cache sizes, the addition of the second Name Cache, with respect to just one, is quite significative in terms of *hit*, whereas in terms of *gain* becames pratically negligible. We can ivestigate for which size ranges we achieved a longer null *gain*. As shown in Fig. 3.17, when the size of the third stage was 10^6 and $\alpha = 1$, 3-LRU got a gain equal to 0 against 2-LRU, when Name Cache 1 and Name Cache 2 were evaluated from 100% to 500% of the third stage size, thus the max *hit* achieved is the same obtained by 2-LRU: 51,77%. This result, is not too much encouraging, beacuse it's not convenient adding a new LRU with high cache size, with respect to the costs of placing it in chain, since much of the possible gain is already achieved by $k = 2$ (i.e., 2-LRU)[11]. This statement, as a consequence, can be extended to k-LRU policy.

3.6 Insights

At the end of an accurate invesigation we can claim that by increasing the cache sizes of 2-LRU or 3-LRU, and α, we obtains very optimal performance in terms of *hit* probability, since our optimization allows to reach very high performance by increasing directly the size of the Name Caches, independently by the main cache one, but in terms of *gain* between 3-LRU vs 2-LRU, we don't have huge benefits, especially by working with large cache size beacause 2-LRU is already designed to achieve the maximum possible gain: thus for 3-LRU, our improvement makes significant sense when we work with small cache sizes, especially for values of $\alpha \leq 1$.

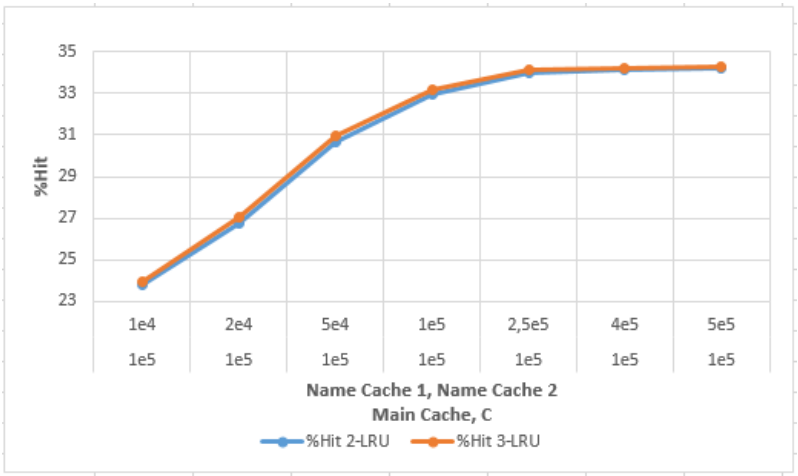


Figure 3.14. Hit probability vs large cache size for 3-LRU vs 2-LRU "cache-decoupled", under IRM and $\alpha = 0.8$.

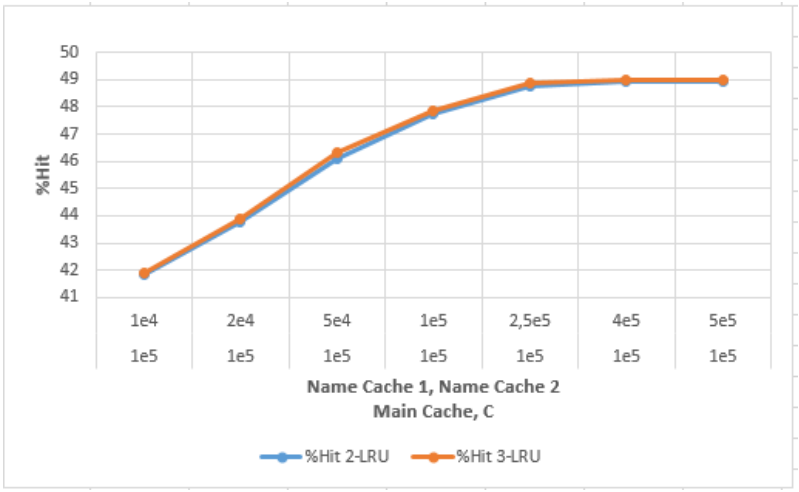


Figure 3.15. Hit probability vs large cache size for 3-LRU vs 2-LRU "cache-decoupled", under IRM and $\alpha = 1$.

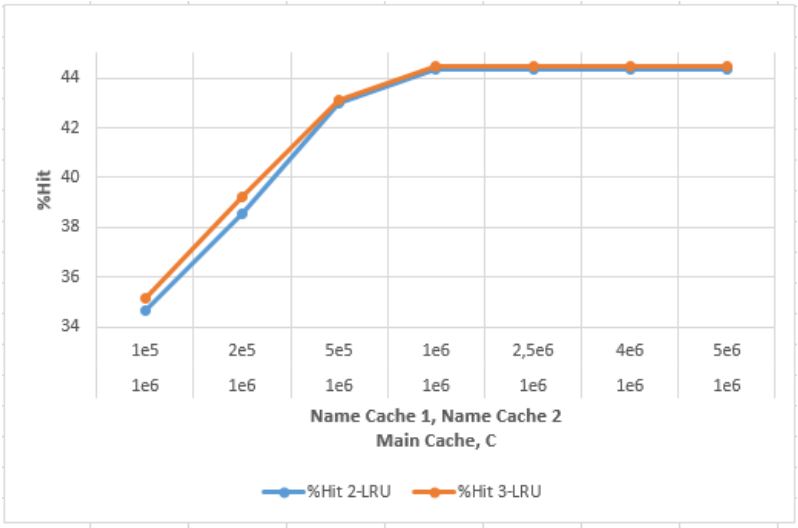


Figure 3.16. Hit probability vs very large cache size for 3-LRU vs 2-LRU "cache-decoupled", under IRM and $\alpha = 0.8$.

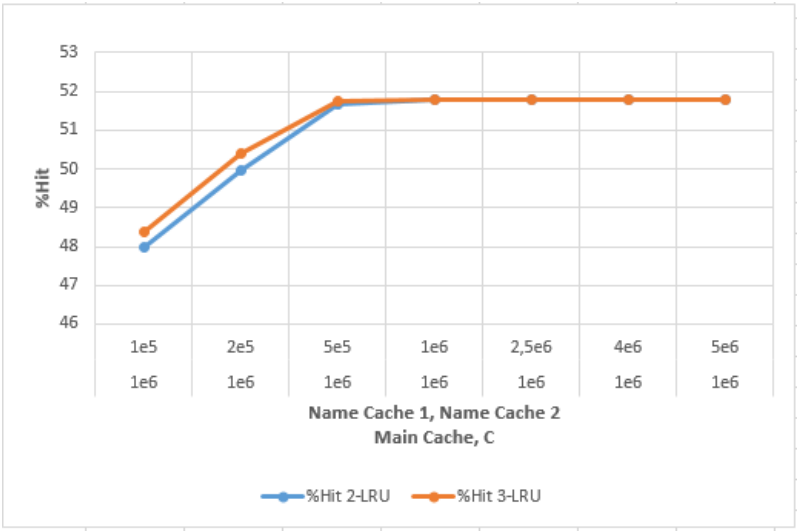


Figure 3.17. Hit probability vs very large cache size for 3-LRU vs 2-LRU "cache-decoupled", under IRM and $\alpha = 1$.

Chapter 4

Performance evaluation of "cache-decoupled" 2-LRU vs k-LRU, under SNM

4.1 Introduction

In this chapter I evaluated, according *event-driven* technique, the performance of "cache-decoupled" 2-LRU and k-LRU policies, under Shot Noise Model (SNM) traffic model, where the temporal locality of requests is modeled through an ON-OFF process. I remember that this optimized version of 2-LRU and k-LRU, allow to set their size to all Name Caches preceding the main one, separating, thus, the $k-1$ Name Cache stages to the k^{th} stage, i.e., the meta-cache.
The goal of this chapter is to better understand the impact on the cache performance of a mixture of heterogeneous contents characterized by different degrees of temporal locality, trying to emulate traffic observed in real networks.[15] In order to validate them, I investigated their performance in two scenarios:

- Single cache.
- Cache networks.

For both network scenarios, I evaluated the performance under different temporal locality, by varying the activity/inactivity period T_{on} and T_{off}. We assume that both ON and OFF are exponentially distributed with *mean* duration of T_{on} and T_{off}.[15] Even in this case requests arriving with constant intensisty λ which are generated during an ON period. Hence the average number of requests arriving during an ON period is given by $E[V] = \lambda_m T_{on}$ [15].
I simulated 10^9 content requests, average total request rate per second (generated during the ON period) $E[V] = 20$, and λ distribuited according Zipf's exponent $\alpha = 1$ to generate content requests. To reduce the complexity of networks, and to do not overload the CPU assigned to the Virtual Machine, I performed the simulations with just one class, and 3-LRU in place of k-LRU, by considering essentially k-LRU, with $k = 3$.
The periods of temporal locality T_{on} and T_{off} associated to the number of contents simulated are provided in Tab. 4.1 and Tab. 4.2, where is depicted that T_{on2} is computed as $T_{on2} = 50 \times T_{on1}$ and T_{off} is computed as $T_{off} = k \times T_{on}$ with $k = \{1, 8\}$ for both life-span T_{on1} and T_{on2}.
All the simulations are performed by keeping fixed the main cache (meta-cache) and by varyng the size of the Name Cache (case 2-LRU), or of the two Name Caches (case 3-LRU) from a minimum of 10% to a maximum of 500% of the main cache.

Ton1(s)	Toff=k*Ton1(s)	Content
43200	43200 (k=1)	62500
43200	345600 (k=8)	62500

Figure 4.1. Settings for the first SNM scenario.

Ton2=50*Ton1(s)	Toff=k*Ton2(s)	Content
2160000	2160000 (k=1)	62500
2160000	17280000 (k=8)	62500

Figure 4.2. Settings for the second SNM scenario.

4.2 Single cache

4.2.1 First SNM scenario: T_{on1}

Let's start to investigate the performance with the first SNM scenario, when the simulations was performed with life-span T_{on1} and different T_{off} periods.
We will note how the policies under SNM significant outperform IRM, since IRM does not consider the temporal locality. Then by increasing the period of inactivity T_{off} we achieved better performance, since decrease the number of simultaneous active contents at the same time.

- $T_{off} = T_{on1}$

Small cache sizes

The 2-LRU and 3-LRU, (more in general k-LRU policy), work well even in the case of very small cache size, under SNM traffic model. Our experiments are compliant to what we expected, indeed as showed in Fig. 4.3, by tuning up the sizes of the Name Cache 1 and Name Cache 2, we obtained a performance in terms of *hit* closer to 57%, we can note that thanks to the *cache-decoupling* optimization, both policies outperform the maximum *hit* obtained by themselves, in the case of very large cache size, under IRM (corresponding to 51.77%).
In terms of *gain*, our optimization leads to mare than 1.5% (of the total 3%) for both 2-LRU and 3-LRU, meaning that there is a good margin of improvement with respect to the "original"[1] policies.
We performed the same investigation by increasing the size of main cache power to an order of 10, i.e., 10^4. We can observe from Fig. 4.4, how the performance of both pocicies result dramatically better, by reaching *hit* over 77%. The *cache-decoupling* allows to *gain* about 1.5% from the case in which the Name Cache 1 and Name cache 2 have the same size to the case in which both ones are 500% of the main cache, i.e., 5×10^4, by providing a *total gain* of about 4%. This outcame tell us that our optimization allows k-LRU, in general, to improve its performance not only under IRM, but also in SNM traffic model, meaning that it mitigates even the temporal locality.

[1] ...it means that all the Name Caches's policy have the same size of the main cache one.

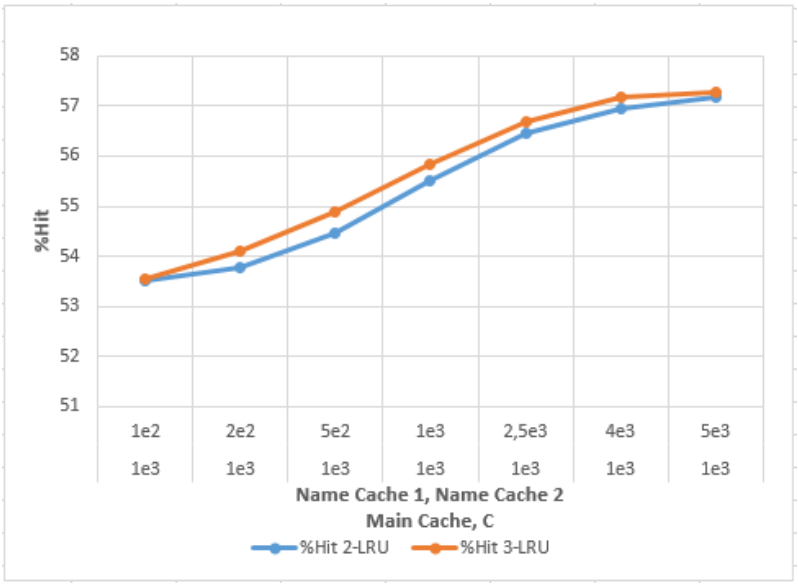


Figure 4.3. Hit probability vs very small cache size, 2-LRU vs 3-LRU, under SNM, single cache, $T_{off} = T_{on1}$

.

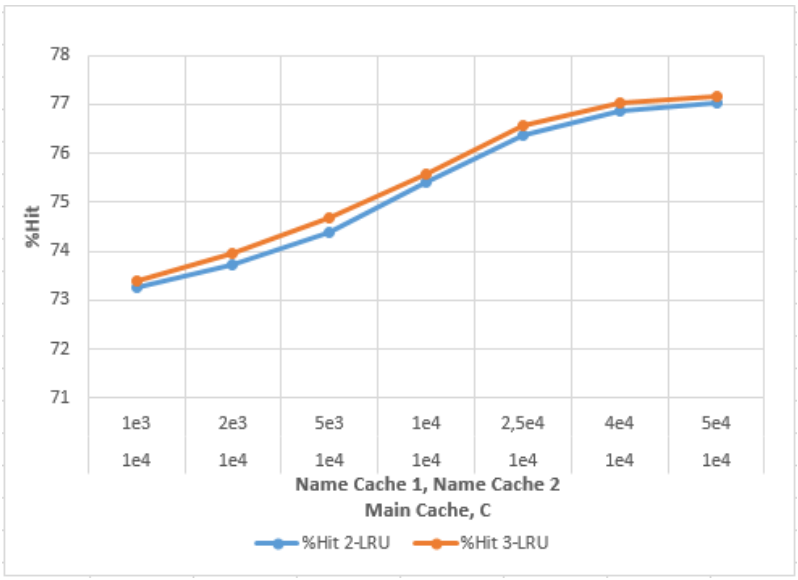


Figure 4.4. Hit probability vs small cache size, 2-LRU vs 3-LRU, under SNM, single cache, $T_{off} = T_{on1}$.

Large cache sizes

In the case of large sizes of cache, i.e., main cache, $C = 10^5$, with 2-LRU we achieved optimal performance, by confirming the strength of the temporal locality under SNM, indeed referring to Fig. 4.5, we can observe that such caching strategy, reached $\approx 95\%$ in terms of *hit*. Since 2-LRU gave much of possible gain yet, 3-LRU didn't lead significant benefits. This is the reason why the *cache-decouping* optimization didn't provide a real *gain* when we increased the Name Caches size over the fixed size of the main cache, whereas the impact of *cache-decouping* on the *overall gain* is very important, by producing high *hit* ranges.
When we examinated cache size larger than 10^5, i.e., 10^6, we obtained null *gain*: both cache strategies stabilized to the maximum *hit* of 95% (depicted in Fig. 4.6), hence for very large cache sizes our optimization becames negligible on caching performance. The employ of SNM model seems to be a powerful solution in caching analyzing, but it requires to know the entire popularity profile in the form of the function $\lambda_m(t)$, that user have to specify for each given content, otherwise is difficul estimating popularity profiles [14].

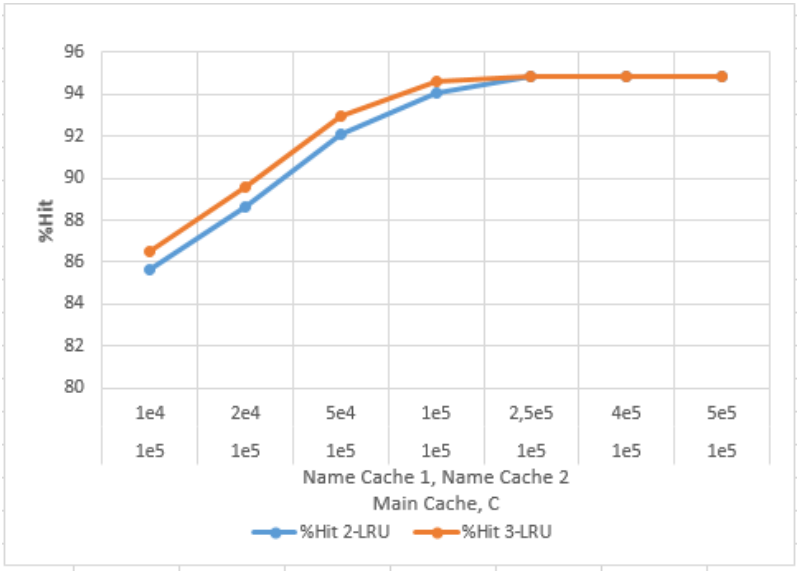


Figure 4.5. Hit probability vs large cache size, 2-LRU vs 3-LRU, under SNM, single cache, $T_{off} = T_{on1}$

.

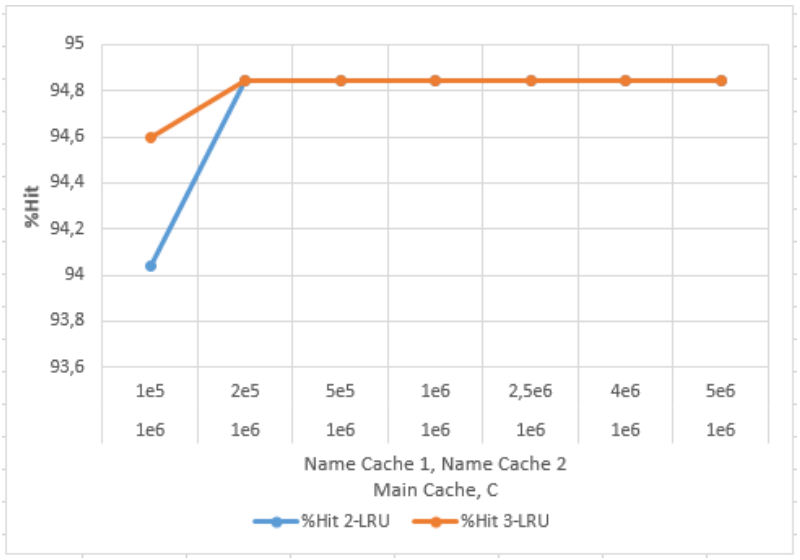


Figure 4.6. Hit probability vs very large cache size, 2-LRU vs 3-LRU, under SNM, single cache, $T_{off} = T_{on1}$.

- $T_{off} = 8 \times T_{on1}$

We performed a new analyses trying to enlarge the T_{off} period of 8 times the life-span T_{on1} in order to verify the performance of studied policies that should be better since decrease the simultaneous content active at the same time. Furthermore when T_{off} is significantly high and much larger than the cache eviction time T_C, at the end of the OFF period, the probability that the cache still contains a copy of a given object m become negligible[15].

Small cache sizes

In the case of larger T_{off}, as we expected, the performance of the both policies outperform these one obtained with weak temporal locality given by $T_{off} = T_{on1}$, by getting an *overall gain* of about 10%, on average, more (see Fig. 4.7).
From the point of view of our optimization, we can appreciate a lower *gain* than larger T_{off} (i.e., $\approx 1\%$ on 2-LRU, $\approx 0.5\%$ on 3-LRU) from sizes over the fixed main cache: meaning that 3-LRU outperforms 2-LRU for small sizes and the *cache-decoupling* works better with smaller T_{off}, because the temporal locality already enahances the policies's performance.
By increasing the size of the main cache to 10^4, we achieved even better performance in terms of *hit*, e.g., the curves reached until $\approx 85\%$ when the sizes of Name Cache 1 (2-LRU case) and Name Cache 2 (3-LRU case) were 500% of the fixed main cache size, C (see Fig 4.8). Even in this case both policies achieved lower *gain* than smaller T_{off} (about 0.5% lesser) for sizes larger than the fixed main cache, C. The positive result is that both poclicies kept unchanged the *overall gain* referred to the *cache-decoupling* optimization.

Large cache sizes

We examinated the behavior of the 2-LRU and 3-LRU in condition of large cache sizes, in this case we achieved very optimal performance. In Fig. 4.9 we can see that the maximum *hit* achieved corresponds to $\approx 96\%$; this means that it's closer to the ideal probability of 1, in other words, there is a probability of 100% to find a given content within the first Name Cache visited. Since this *hit* values are too high, they are not fully reliable, that confirms how is difficult estimating caching performance in a realistic scenario.
In terms of *gain*, 3-LRU outperformed 2-LRU when the size of the two Name Caches were evaluated between 10% to 100% of the main cache, by achieving about 8% as *total gain* thanks to our optimization, whereas it achieved null gain when the size of the two Name Caches were increased over the size equal to the main cache.
By increasing the main cache size, $C > 10^5$, (e.g., 10^6) we obtained constant *hit* greater to 96%, with a pratically null *gain*, as showed in Fig 4.10. Even this result is few reliable, but it makes us understand that 2-LRU, 3-LRU, more in general k-LRU, in a scenario with very large sizes and high degree of temporal locality performs very optimal, however our improvement loose its efficacy.

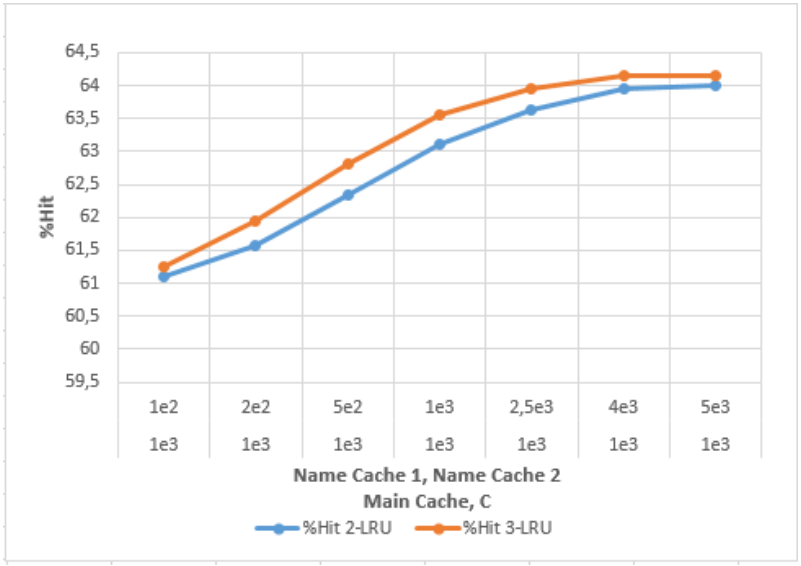


Figure 4.7. Hit probability vs very small cache size, 2-LRU vs 3-LRU, under SNM, single cache, $T_{off} = 8 \times T_{on1}$.

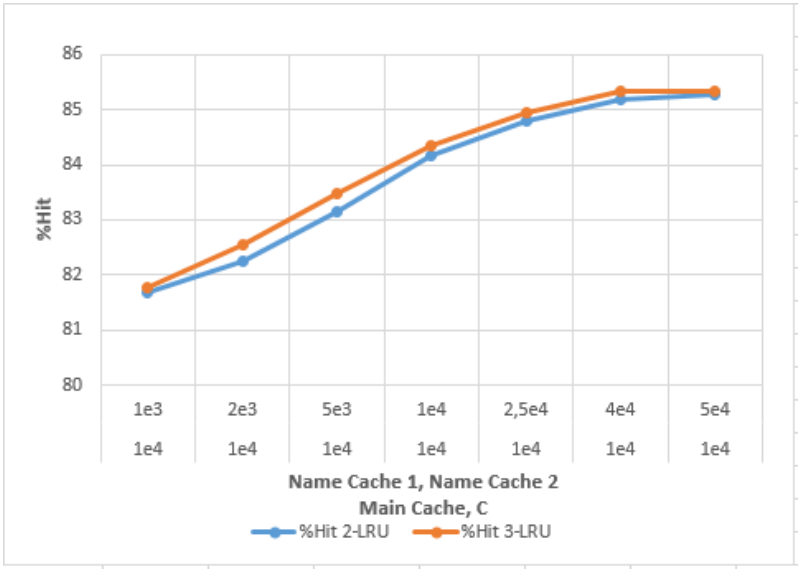


Figure 4.8. Hit probability vs small cache size, 2-LRU vs 3-LRU, under SNM, single cache, $T_{off} = 8 \times T_{on1}$.

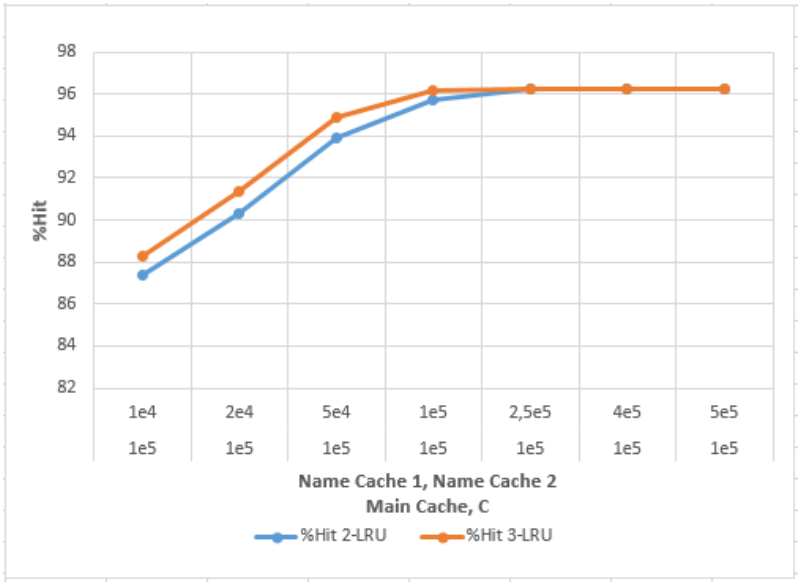


Figure 4.9. Hit probability vs large cache size, 2-LRU vs 3-LRU, under SNM, single cache, $T_{off} = 8 \times T_{on1}$

.

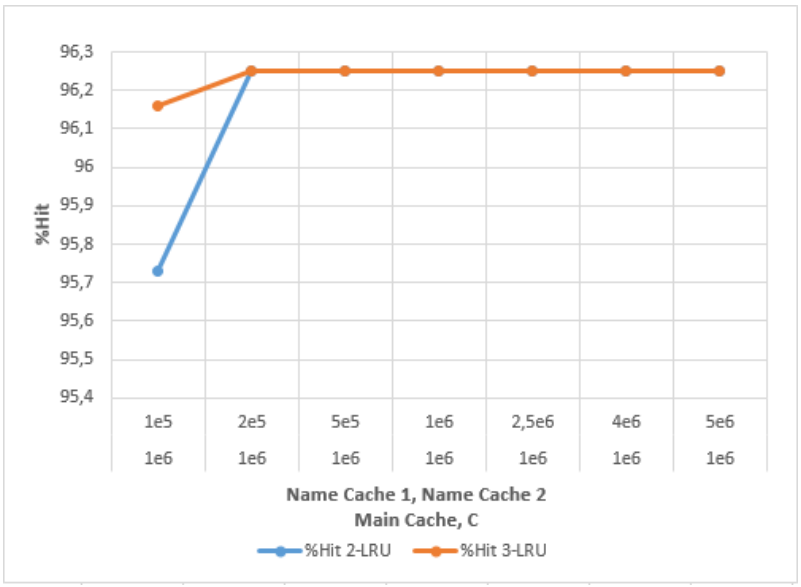


Figure 4.10. Hit probability vs very large cache size, 2-LRU vs 3-LRU, under SNM, single cache, $T_{off} = 8 \times T_{on1}$.

4.2.2 Second SNM scenario: T_{on2}

In this section I would *emphasize* that increasing the T_{on} period leads to worse performance, because the activity life-span T_{on}, where concurrent contents can be requested at same time, become larger. Those list of simulations highlight this theoretical forecast, through another Shot Noise Model Scenario, characterized by $T_{on2} > T_{on1}$, computed as $T_{on2} = 50 \times T_{on2}$, (see Tab. 4.2, sec. 4.1).
Therefore we analyzed the behavior of "cache-decoupled" 2-LRU and 3-LRU with changing T_{off}, in terms of *hit* and *gain*

- $T_{off} = T_{on2}$

Small cache sizes

First of all we evaluated the case of small cache sizes, as we expected, we achieved worse performance against T_{on1}. In such case the policy curves were in constant growth (see Fig. 4.11), differently from what happened with shorter T_{on1}, where, there was an interval of changing trend when the two Name Caches were evaluated between 50% and 100% of 10^3.
Then we increased the size of the main cache to 10^4, and even in this case we achieved worse performance with respect to the case of smaller T_{on1}, by confirming the theory.
For small cache sizes the *cache-decoupling* optimization works very well and slightly better than shorter life-span T_{on1}: it produced huge benefits, indeed as we can see from Figs. 4.11, and 4.12, we obtained an *overall gain* closer to 10% and until to 2% of *hit gain* from sizes over than fixed main cache. Those results are the prove that our improvement takes advantages when we handle small cache sizes and larger ON life-spans, hence when the policies perform worse at starting conditions.

Large cache sizes

In the case of large cache sizes, since k-LRU performs already very good, the advantages produced by our optimization was negligible. 3-LRU policy got a constant *hit* to about 67.5% (plotted in Fig. 4.13), thus about 30% less than T_{on1} case; 2-LRU performed as 3-LRU except in the case when Name Cache 1 was 10% of the main cache.
With size of main cache larger than 10^5, (e.g., 10^6), the performance of both 2-LRU and 3-LRU didn't change, because they reached their maximum stabilization *hit*, yet (depicted in Fig. 4.14); in this scenario, our optimization takes *gain* equal to 0 and doesn't influence their total performance.
We can note that in the case and *only* in this case of *big* sizes and *larger* T_{on2} life-spans, we obtained that our optimization of *cache-decoupling* worked bad, so it provided better benenifts, in terms of *gain*, with smaller T_{on1}.

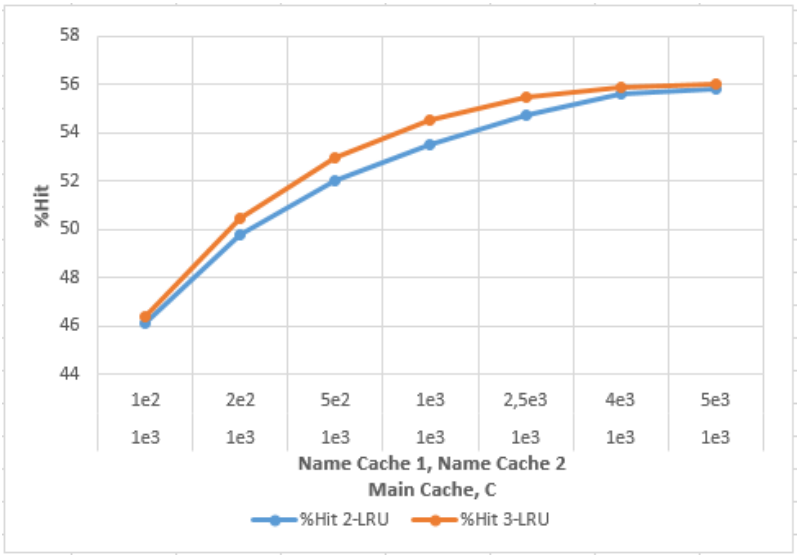


Figure 4.11. Hit probability vs very small cache size, 2-LRU vs 3-LRU, under SNM, single cache, $T_{off} = T_{on2}$.

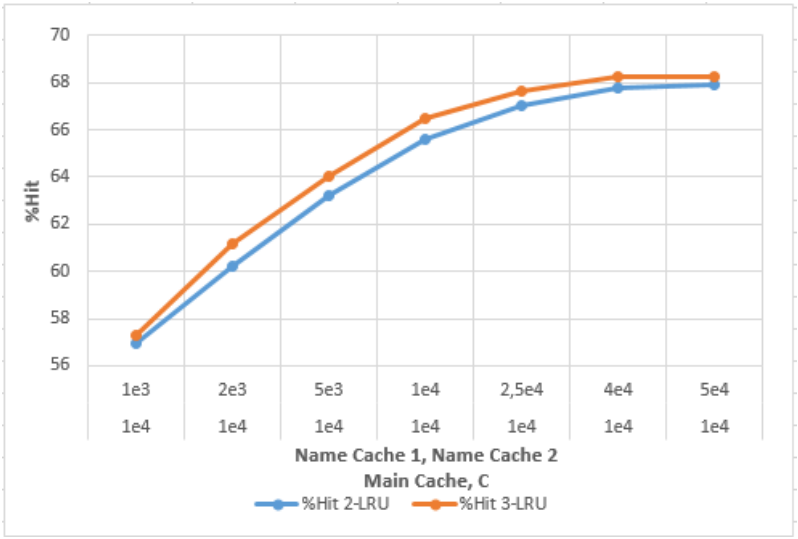


Figure 4.12. Hit probability vs small cache size, 2-LRU vs 3-LRU, under SNM, single cache, $T_{off} = T_{on2}$.

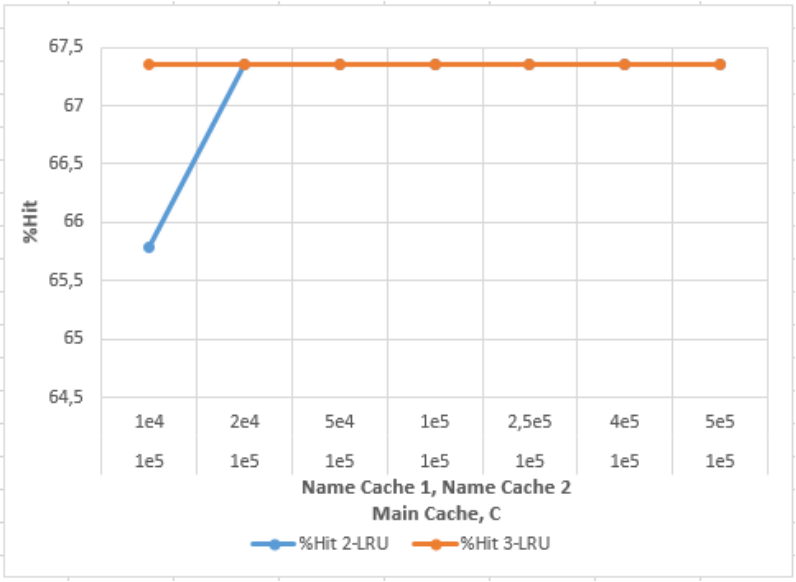


Figure 4.13. Hit probability vs large cache size, 2-LRU vs 3-LRU, under SNM, single cache, $T_{off} = T_{on2}$.

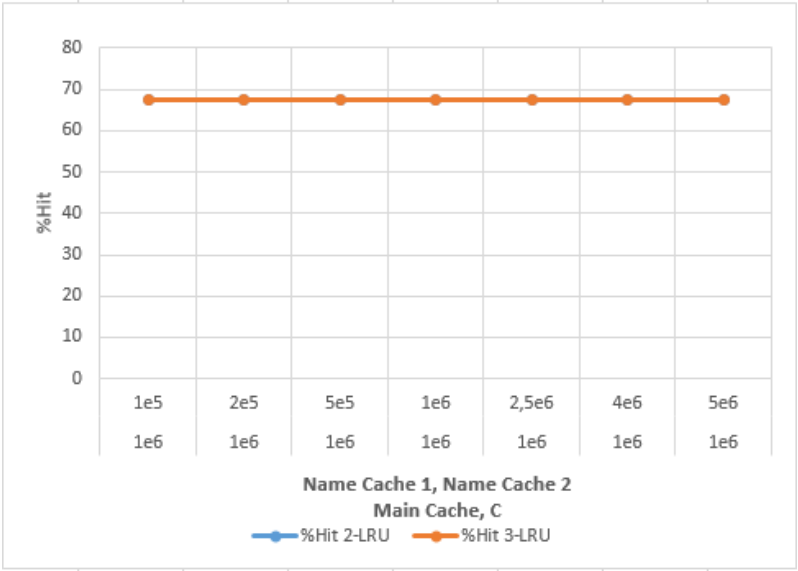


Figure 4.14. Hit probability vs very large cache size, 2-LRU vs 3-LRU, under SNM, single cache, $T_{off} = T_{on2}$.

- $T_{off} = 8 \times T_{on2}$

As the last analisys, we have evaluated the case of high T_{off} period, that correspond to 8 times T_{on2}, by keeping unchanged the number of contents distrbuted over the single class. In this case such high ratio T_{off}/T_{on2} leads to optimal performance, but however worse than shorter T_{on1}.

Small cache sizes

As we expected, even in this case of very small sizes, we achieved greater *gain* than T_{on1} scenario for k-LRU in general: around 10% *overall*, by perceiving a *gain* over 1.5% from the "original" [2] k-LRU policy, thus a little worse than the case of shorter T_{off}, since our optimization works better with weak temporal locality. In terms of *hit* we obtained until $\approx 64\%$ (showed in Fig 4.15), that under SNM, is however a good result.
Similar results were obtained by increasing the size of main cache to 10^4, by getting the same *overall gain* of 10^3, but with better performance in terms of *hit* (see Fig. 4.16). Our purpose was to mantain the same *gain*, even when the size of both the main cache and the Name Caches were increased.

Large cache sizes

By tuning up the size of the main cache to 10^5, we achieved reasonable values, with an increase of *hit* of $\approx 4\%$, by performing very well given by the high degree of temporal locality. In this case both 2-LRU and 3-LRU reached their maximum *hit* by stabilizing 76.76% (showed in Fig. 4.17). From the point of view of our optimization, the *cache-decoupling* didn't lead significant improvement, since the policies performed as their best yet.
The very advantage, for large sizes, is that our *cache-decoupling* optimization didn't adversely influence on the impact of the performance when it provided no benefits, although we were not in condition of weak temporal locality. Fig 4.18 shows the performance of 2-LRU and 3-LRU in case of very large sizes (i.e. $\geq 10^6$).

[2] ...it means that the k caches have the same size

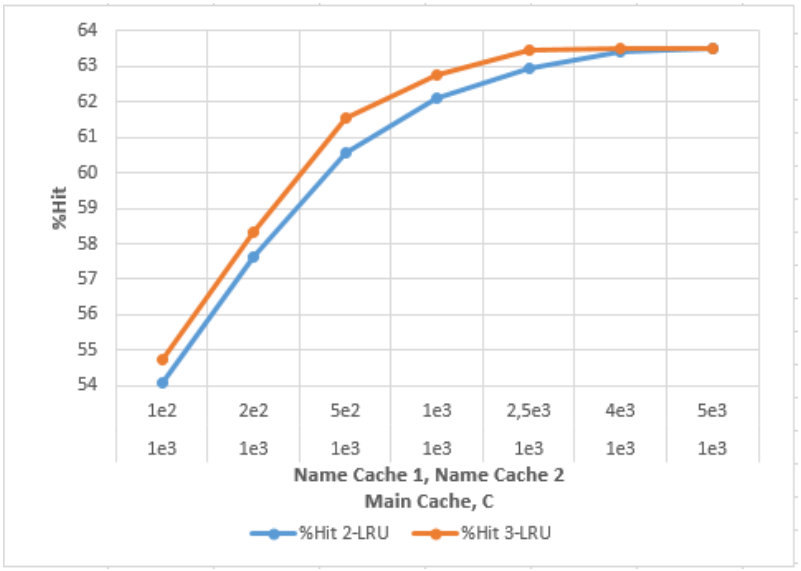


Figure 4.15. Hit probability vs very small cache size, 2-LRU vs 3-LRU, under SNM, single cache, $T_{off} = 8 \times T_{on2}$.

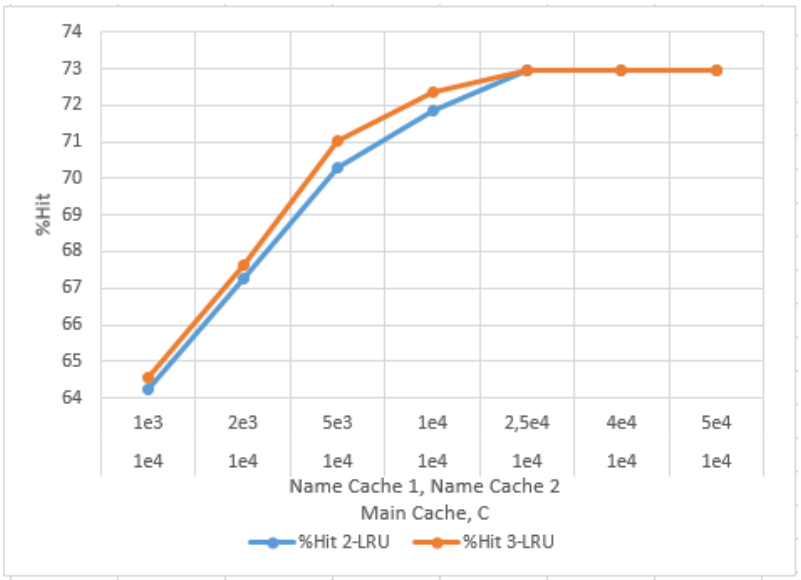


Figure 4.16. Hit probability vs small cache size, 2-LRU vs 3-LRU, under SNM, single cache, $T_{off} = 8 \times T_{on2}$.

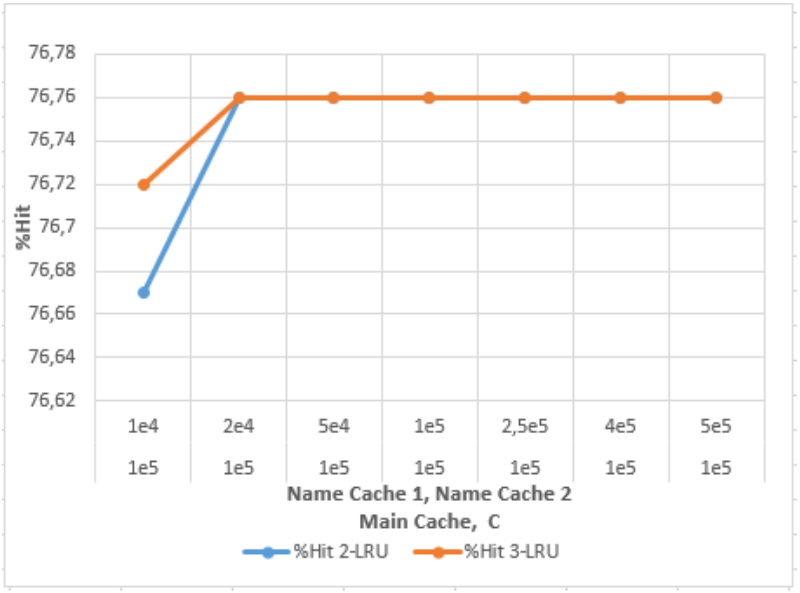


Figure 4.17. Hit probability vs large cache size, 2-LRU vs 3-LRU, under SNM, single cache, $T_{off} = 8 \times T_{on2}$

.

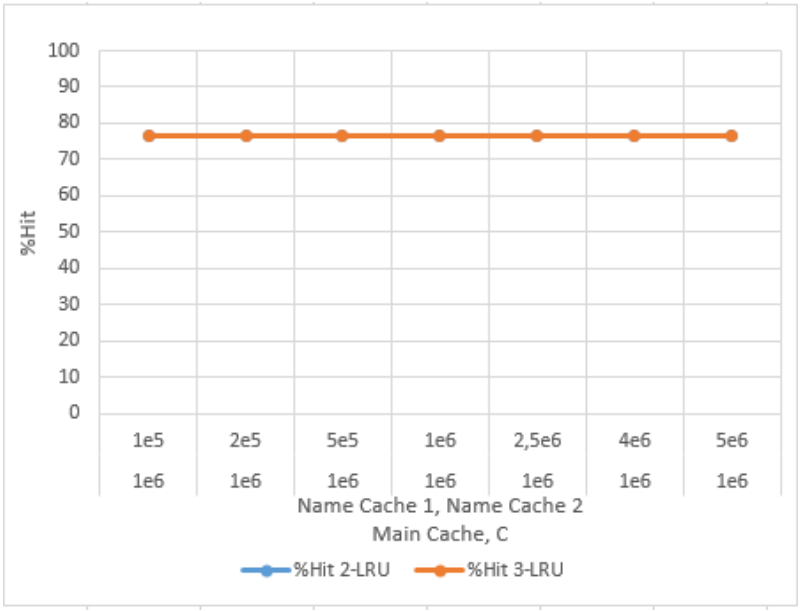


Figure 4.18. Hit probability vs very large cache size, 2-LRU vs 3-LRU, under SNM, single cache, $T_{off} = 8 \times T_{on2}$.

4.3 Cache networks

The considered scenario represents an ICN-access tree network[29], where the topology is a $N = 15$ nodes 4-level binary tree depicted in Fig. 2.1 of Chapter 2. A single repository, connected to the root node, stores a $M = 10^9$ objects catalog, where objects follow a Zipf popularity distribution with exponent $\alpha = 1$. An overall $R = 10^9$ requests are injected at each leaf nodes with aggregate request rate for each client of $\lambda = 20$ req/s per leaf.
Contents are replicated on all caches traversed by a request and in the case of a miss, requests are forwarded to the root.
The goal is two-fold: showing the performance of cache networks under SNM traffic at the variation of T_{on} and T_{off}; verifying that tree network performs worse than single cache scenario, since every tree's hit probability is computed through the *mean* of all cache nodes, thus the nodes placed along the end of the path adversely affect the total *mean* hit achieved.

4.3.1 Traffic Model for SNM cache networks

We now extend the basic SNM introduced in Chapter 1 (sec 1.2), to manage the case of multiple interconnected caches in a tree-like topology, in which edge caches receive the requests generated by users which different interests from one ingress point to another.
The general form associates to every content m and ingress point i, a tuple ($V_{m,i}$, $\lambda_{m,i}$, $\tau_{m,i}$), so that, at ingress point i, requests for content m arrive according to an inhomogeneous Poisson process, whose instantaneous rate at time t is given by $V_{m,i}\lambda_{m,i}(t - \tau_{m,i})$[43]. In our semplification we assume that that the instants at which content m starts to be available in the system at the different ingress points ($\tau_{m,i}$) are equal, thus $\tau_{m,i} = \tau_m$, and as a consequence the popularity evolution of a content is perfectly synchronized across different ingress points, by assuming that $\lambda_{m,i} = \lambda_m$[43][14].
Finally, we denote ($V_{m,i}$) as the volumes of requests that are generated by contents at the different ingress points. The our semplification is the following: for each content m, we assign a global volume V_m, representing the total number of requests that are generated in the whole system[43].

4.3.2 First SNM scenario: T_{on1}

As the case of single cache, we investigated the caching performance with the first SNM scenario, when the simulations was performed with T_{on1}, and we varied the T_{off} periods accordingly.

- $T_{off} = T_{on1}$

As we expected, from the discussion had in Sec 4.3, the performance achieved by the cache networks, on the following simulations outcame, are worse than single cache, but however better than IRM model. We examinated where our optimization got major benefits.

Small cache sizes

In the case of very small cache size (i.e., 10^3) we obtained *hits* not too performing, about 20%, on average, less than single cache, indeed by tuning up the size of the Name Cache 1 (2-LRU case) and Name Cache 2 (3-LRU case), the *hits* didn't exceed the 36%, as showed in Fig. 4.19, for both policies. The surprising result given by the *cache-decouping* optimization, was found in terms of *gain*: 2-LRU and 3-LRU gained overall 5% of hit, in particular, about 1.5% for 2-LRU and 1% for 3-LRU from the case in which the two Name Caches did the same main cache size, to 500% of that one.
Even in the case in which the size of the main cache was increased to 10^4, our improvement perceived about 5% of *overall gain* of which 1% for both 2-LRU and 3-LRU from sizes over the fixed main cache(see Fig. 4.20), by reducing $\approx$ 5%, on average, the gap against single cache. The challange was to keep the same *gain* also in the case of increased cache sizes. Although we succedeed, the most of the *gain* was achieved just when the two Name Caches sizes were evaluated between the 10% and 100% of the main cache, for both policies.
In other words, in cache networks scenario, the *cache-decouping* aims to reduce the differences of performance with respect the single cache, by increasing the *overall gain*, but however it didn't provide significative benefits with respect the "original" policies, in which both Name Caches have the same size of the main cache one.

Large cache sizes

For large cache sizes, k-LRU in general, works very well, even in the networks of cache scenario. Hence, when a request is forwarded to the upper level, in case of miss, the cache nodes near the root, affect less adversely, if the caches have large sizes, because they can contain more copies of possible content requested. In this case the *cache-decoupling* achieved huge benefits on the *overall gain*, however from Name Cache sizes greater than the main cache, the hit *gain* perceived was negligible. Fig. 4.21 shows the performance of 2-LRU and 3-LRU evaluated when the main cache sizes, C was 10^5.
For very large cache sizes (i.e., $\geq 10^6$), we achieved similar *hits* of large cache size scenarios, whereas since both strategies already performed very well, our improvement didn't provide benefits in terms of *gain*. Fig. 4.22 shows the performance of 2-LRU and 3-LRU evaluated when the main cache size, $C = 10^6$.

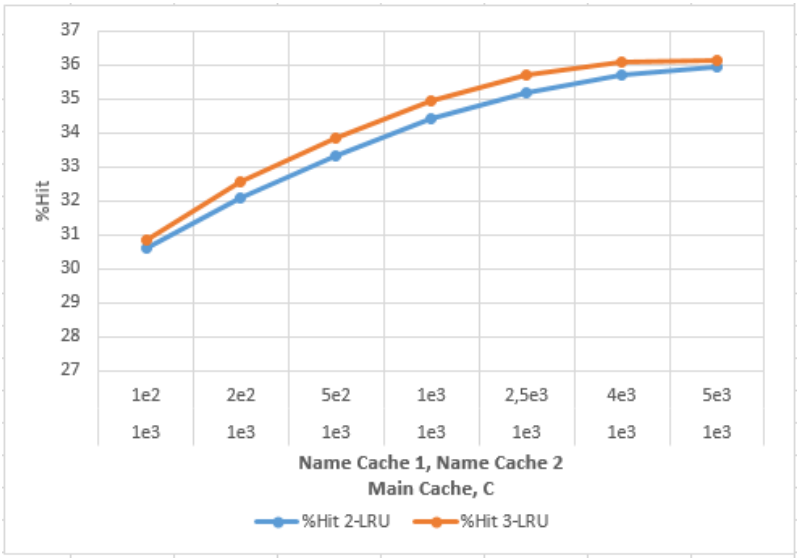


Figure 4.19. Hit probability vs very small cache size, 2-LRU vs 3-LRU, under SNM, cache networks, $T_{off} = T_{on1}$.

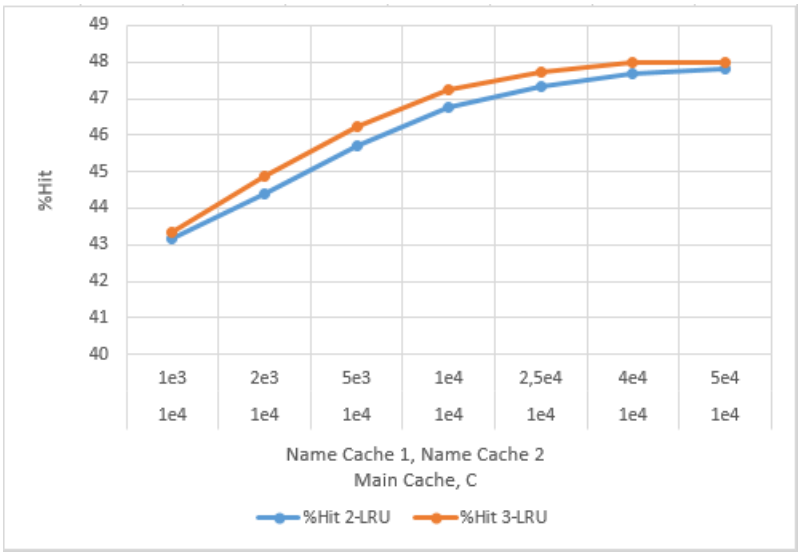


Figure 4.20. Hit probability vs small cache size, 2-LRU vs 3-LRU, under SNM, cache networks, $T_{off} = T_{on1}$.

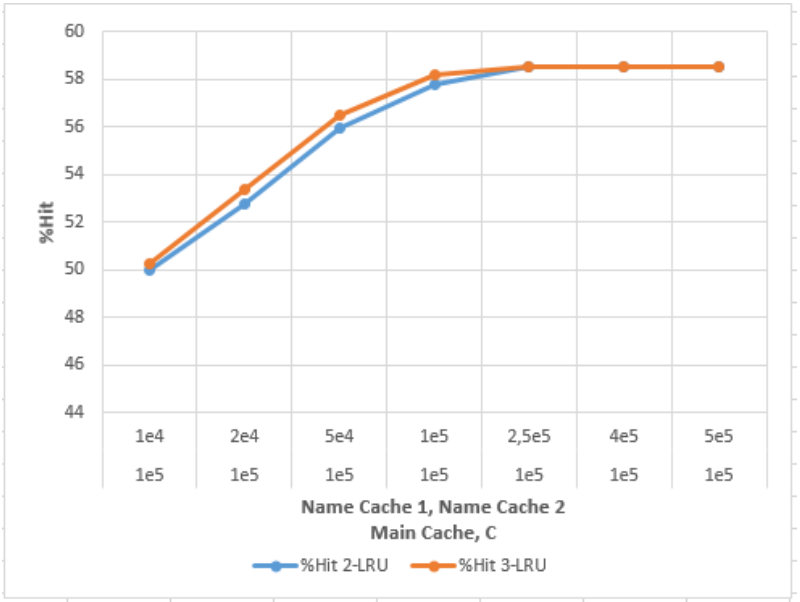


Figure 4.21. Hit probability vs large cache size, 2-LRU vs 3-LRU, under SNM, cache networks, $T_{off} = T_{on1}$.

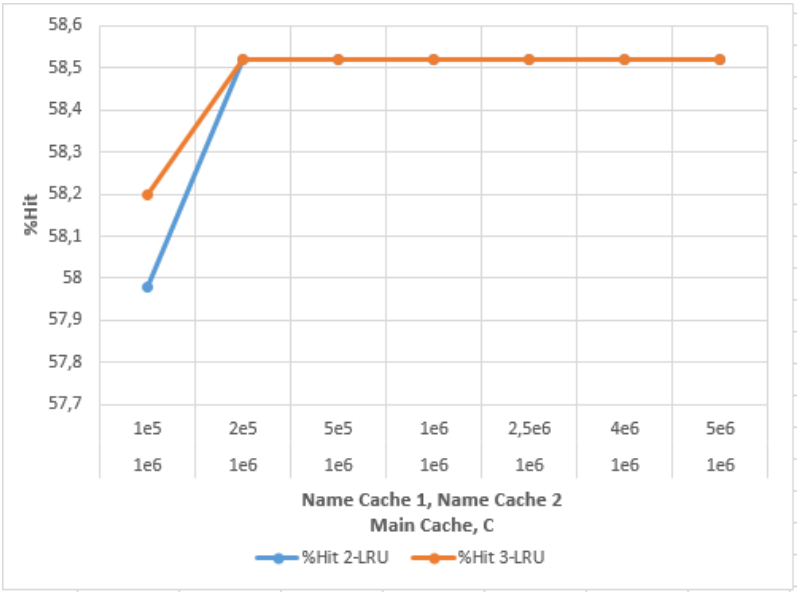


Figure 4.22. Hit probability vs very large cache size, 2-LRU vs 3-LRU, under SNM, cache networks, $T_{off} = T_{on1}$.

- $T_{off} = 8 \times T_{on1}$

In order to have better performance, we expanded the T_{off} period 8 times life-span T_{on1}.

Small cache sizes

With $T_{off} \gg T_{on1}$, we obtained improved performance, in terms of *hit*, but worse *gain* than the case given by smaller T_{off}, from the point of view of our optimization. This is due by the fact that with high degree of temporal locality, both 2-LRU and 3-LRU performed at their better, yet. Fig. 4.23, shows the performance with main cache size, $C = 10^3$, we observed, indeed, an achieved *gain* of about 3%, thus 2% less than the case of weak temporal locality given by $T_{off} = T_{on1}$.
In Fig. 4.24, are plotted the performance in which the size of main cache was icreased by a power of 10, in this case the *overall gain* was kept quite equal to the case of smaller cache sizes: 3%, by obtaining a good *gain* even for sizes larger than the main cache (i.e., 10^4). This is a good result, because, the *cache-decouping* optimization worked well even in a scenario already improved by the high degree of temporal locality, by achieving about 1% and 0.5%, respectively for 2-LRU and 3-LRU, with respect to the "original" policies.

Large cache sizes

As we expected, by investigating the behavior of the poclicies where we handle large cache sizes, we obtained an high *total gain* than single cache scenario, of about 8%, whereas we didn't get important advantages on the impact than the "original" policies, since both 2-LRU and 3-LRU stabilized to the maximum *hit* of about 72% (showed in Fig. 4.25), just when Name Cache 1 and Name Cache 2 were 50% of the main cache, C.
For size larger than 10^5, we can ascertain that both 2-LRU and 2-LRU reached a constant *hit*, because they achieved their maximum *hit*, by performing very good as depicted in Fig. 4.26, thus the *cache-decoupling* didn't lead any *gain* in such case.

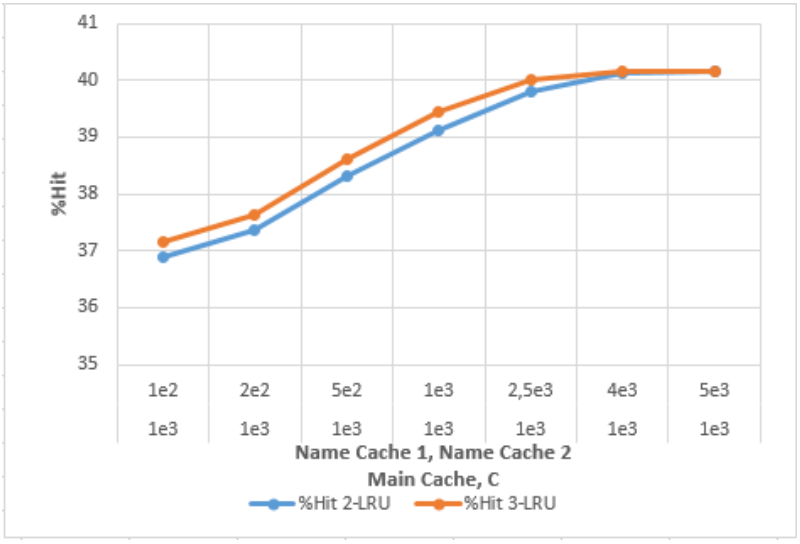


Figure 4.23. Hit probability vs very small cache size, 2-LRU vs 3-LRU, under SNM, cache networks, $T_{off} = 8 \times T_{on1}$.

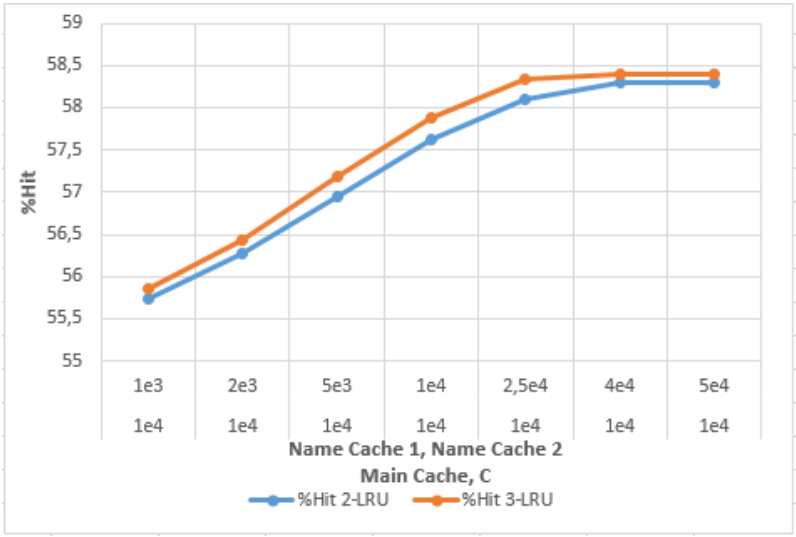


Figure 4.24. Hit probability vs small cache size, 2-LRU vs 3-LRU, under SNM, cache networks, $T_{off} = 8 \times T_{on1}$.

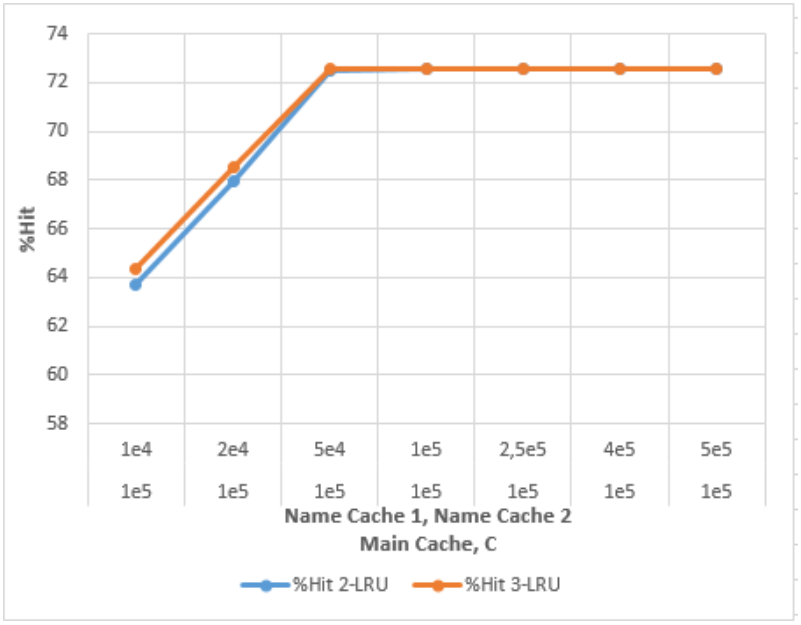


Figure 4.25. Hit probability vs large cache size, 2-LRU vs 3-LRU, under SNM, cache networks, $T_{off} = 8 \times T_{on1}$.

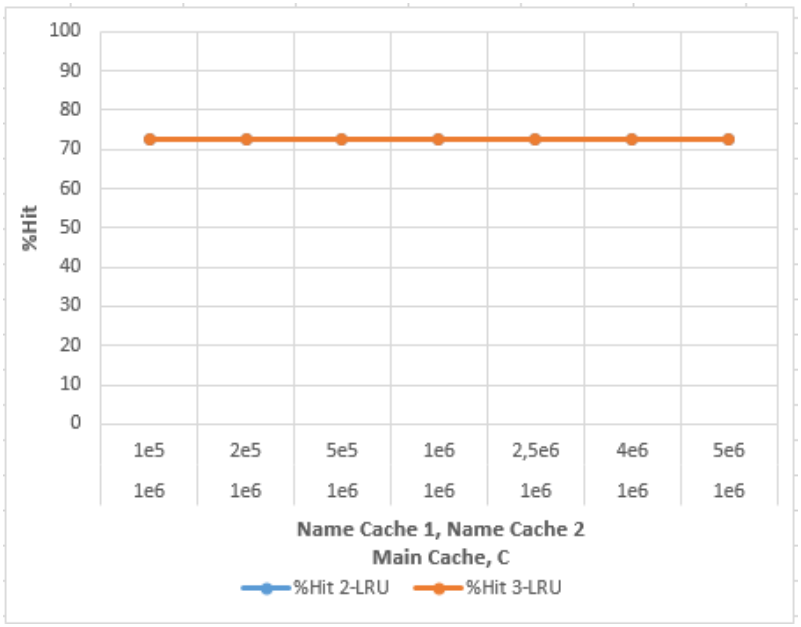


Figure 4.26. Hit probability vs very large cache size, 2-LRU vs 3-LRU, under SNM, cache networks, $T_{off} = 8 \times T_{on1}$.

4.3.3 Second SNM scenario: T_{on2}

In this section I would *emphasize* that even in a tree-like network, increasing the T_{on} life-span leads to achieve worse performance, remainding that our simplifying assumptions, according to which the popularity evolution of each content is "synchronized"[15] across different ingress points ($\tau_{m,i} = \tau_m$, $\lambda_{m,i} = \lambda_m$), we refer to the second Shot Noise Model Scenario, characterized by $T_{on2} > T_{on1}$ period [3] that leads to results similar to IRM traffic, knowing that the degree of synchronization increases with the content popularity referred to the life-span T_{on2}.
According to tree-like network, we analyzed the behavior of "cache-decoupled" 2-LRU and 3-LRU with changing T_{off}, in terms of *hit rate* and *gain*.

- $T_{off} = T_{on2}$

Small cache sizes

In the case of larger T_{on2} life-span period, and by evaluating small cache size of 2-LRU and $3-LRU$, in such scenario of cache networks, we achieved poor performance in terms of *hit*, around 20%, on average, lower than single cache, and just 2.5%, on average, less than T_{on1} thanks to the *cache-decoupling* optimization that in this case worked very well by leading to good percentage *hits gain* than the "original" caching strategies, since it obtained an *overall gain* of about 7%, of which more than 2% when the Name Caches sizes of both strategies were larger than the main cache one. The performance of both policies are showed in Fig. 4.27, where we observed that in this case of small sizes and larger T_{on2} 3-LRU outperforms 2-LRU.
By observing Fig 4.28, we can see that by incresing the size of the main cache to a power of 10, i.e., 10^4, we obtained less benefits than 10^3 by decreasing the *overall gain* of about 1%, whereas we kept the same *hit gain* of 2% with respect the "original" policies; furthermore we obtained better *hits* performance by expoloiting, thus, at its best the improvement of the *cache-decoupling*, by keeping fixed the gap, on average, with single cache and T_{on1}.

Large cache sizes

When we analyzed large cache sizes, 3-LRU performed slightly better than 2-LRU, but both policies stabilized to $\approx 56\%$ of *hit*, that we considered a good result as measurament of performance. This means that the *cache-decoupling* optimized significantly the "original" policies, since it worked better in condition of starting worse performance, as in such case of cache networks scenario and enlarged T_{on2}. Our improvement, hence, provided an *overall gain* of about 10%, and a good *gain* was achieved even when the two Name Caches size were larger than the main cache one ($\approx 1\%$) (see Fig. 4.29). The only case in which we obtained no *gain* from our optimization was when both Name Cache 1 and Name Cache 2 were evaluated between 250% and 500% of the main cache size, C.
With very large size of main cache, i.e. $C = 10^6$, we obtained a constant *hit* exceeding 58%, as showed in Fig. 4.30. In this case, for too large size, we got a negligible *gain* from the *cache-decoupling*.
For large and very large cache sizes, we obtained about 10% of *hit gain*, on average, lower than single cache, (half than small cache sizes) and still 2%, on average, less than T_{on1}.

[3] see Tab. 4.2, sec. 4.1

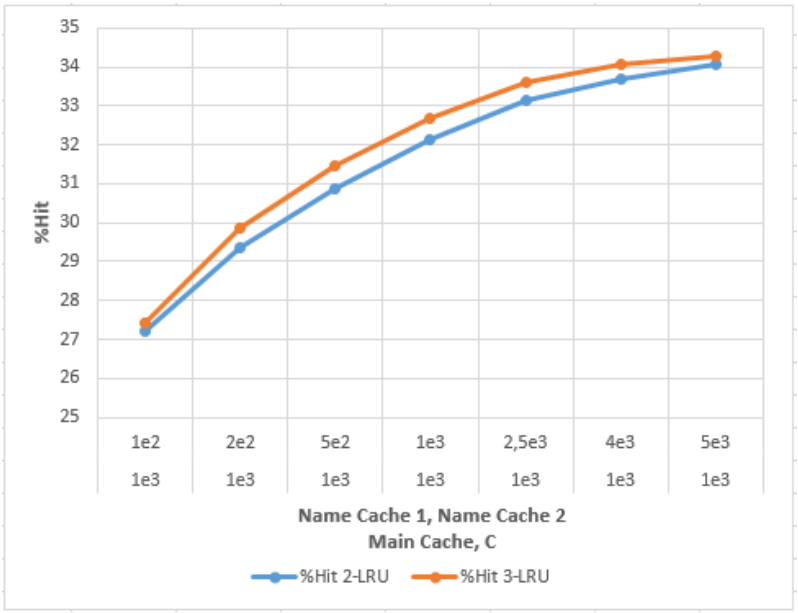


Figure 4.27. Hit probability vs very small cache size, 2-LRU vs 3-LRU, under SNM, cache networks, $T_{off} = T_{on2}$

.

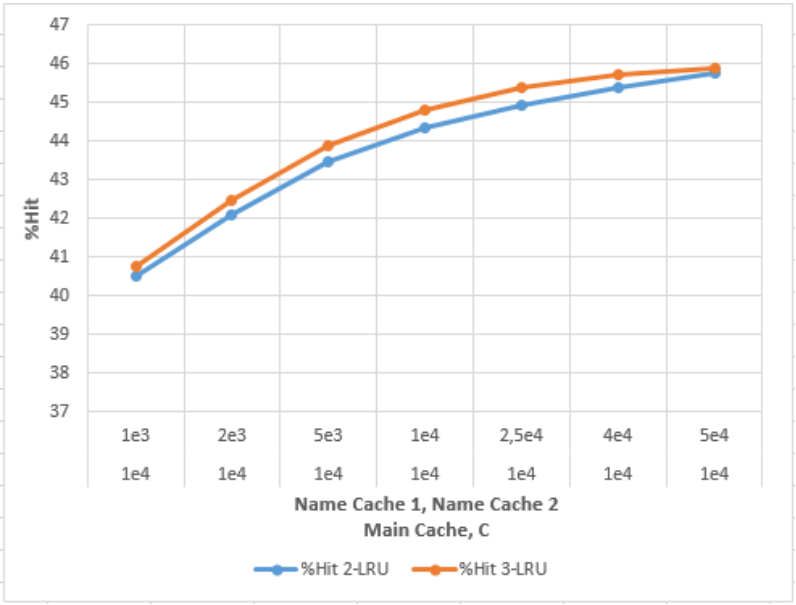


Figure 4.28. Hit probability vs small cache size, 2-LRU vs 3-LRU, under SNM, cache networks, $T_{off} = T_{on2}$.

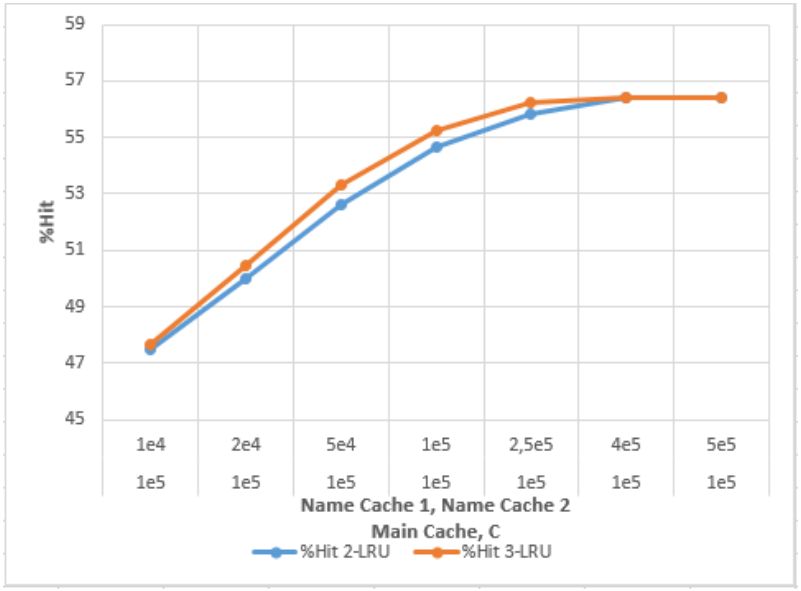


Figure 4.29. Hit probability vs large cache size, 2-LRU vs 3-LRU, under SNM, cache networks, $T_{off} = T_{on2}$.

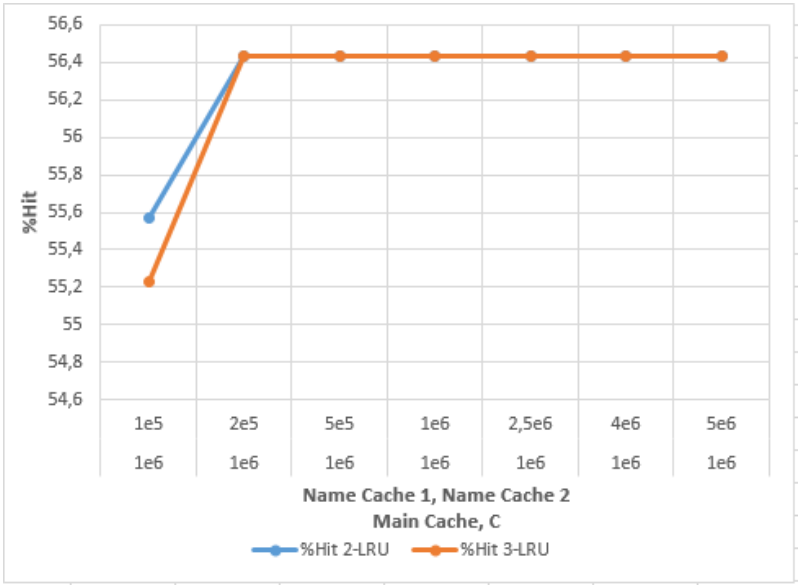


Figure 4.30. Hit probability vs very large cache size, 2-LRU vs 3-LRU, under SNM, cache networks, $T_{off} = T_{on2}$.

- $T_{off} = 8 \times T_{on2}$

We icreased the T_{off} period 8 times T_{on2}, in order to achieve better performance, by keeping unchanged the number of contents.

Small cache sizes

By considering the two policies when were evaluated with small cache sizes, i.e., 10^3, (see Fig. 4.31), we obtained better performance than the case of lower content temporal locality $T_{off} = T_{on2}$, however worse than shorter T_{on1} (i.e., $\approx 2\%$ lesser, on average), and even worse than single cache case (i.e., $\approx 25\%$ lesser, on average). The impact of the *cache-decoupling* brought good benefits on the *overall gain* achieved, by improving both 2-LRU and 3-LRU performance of about 5% of *hit*, of which more than 1.5% when the Name Cache 1 and Name Cache 2 size were larger than the fixed main cache one (i.e., 0.5% less than weak temporal locality $T_{off} = T_{on2}$).
Even in such case of small sizes and larger T_{on1} we can note that 3-LRU significantly outperforms 2-LRU.
We didn't keep the same *gain* of $T_{off} = T_{on2}$ when the two Name Cache sizes were larger than the main cache one, thus from the "original" caching strategies, (i.e., $\approx 1\%$ lesser), in the case in which we increased the size of the main cache to 10^4, (see Fig. 4.32), this is due to the strong degree of temporal locality that adversely influenced the *cache-decoupling* optimization, instead, with respect to shorter T_{on1} both policies achieved better *overall gain*.

Large cache sizes

In the case of large cache sizes, k-LRU more in general, performs however very well even in a network of caches scenario, i.e. 4-level binary tree, and larger T_{on2}. By showing Fig 4.33, when the cache size of main cache was increased to 10^5, we confirmed that with high degree of temporal locality the improvements of the *cache-decoupling* decreased, indeed, we obtained worse *gain*, exactly all the *overall gain* was achieved by 2-LRU and 3-LRU, when both Name Caches size were evauated between the 10 and 50% of the main cache, meaning that for cache sizes larger than the main cache we didn't achieve any *gain*, hence, in such range, our optimization, provided null *gain*.
With cache sizes over than 10^6, both policies reached their maximum *hit*, by stabilizing to about 68% (showed in Fig. 4.34), even in this case our optimization didn't produce benefits.

4.4 Insights

In this chapter we investigated the performance of 2-LRU and 3-LRU (i.e. k-LRU in general), under SNM, under different degree of temporal locality and different scenarios. We confirm, through such simulations, that by increasing T_{off} period, both caching strategies perform better, since decrease the number of simultaneous active contents, thus shorter T_{on} (i.e., T_{on1}) leads to better performance, especially in presence of single cache topology.
However high degree of temporal locality provides huge benefits on caching performance, but it limits the benefits of the *cache-decoupling* optimization on the cache impact in terms of *gain*: in practise such improvement helps to increase the policies's performance when we work with small cache size, and lower contents temporal locality, since our optimization influenced also the dynamic content changing, thus it mitigates the caching performance even under SNM traffic model.
By concluding, it needs to opportunely set the periods T_{on} and T_{off} in order to maximize the *hit gain* given by the *cache-decoupling* and, at the same time, ensuring high performance.

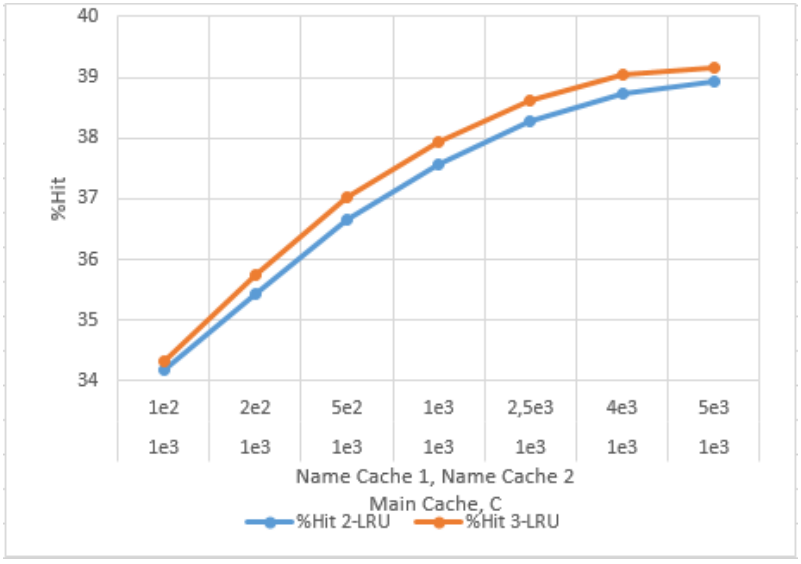


Figure 4.31. Hit probability vs very small cache size, 2-LRU vs 3-LRU, under SNM, cache networks, $T_{off} = 8 \times T_{on2}$.

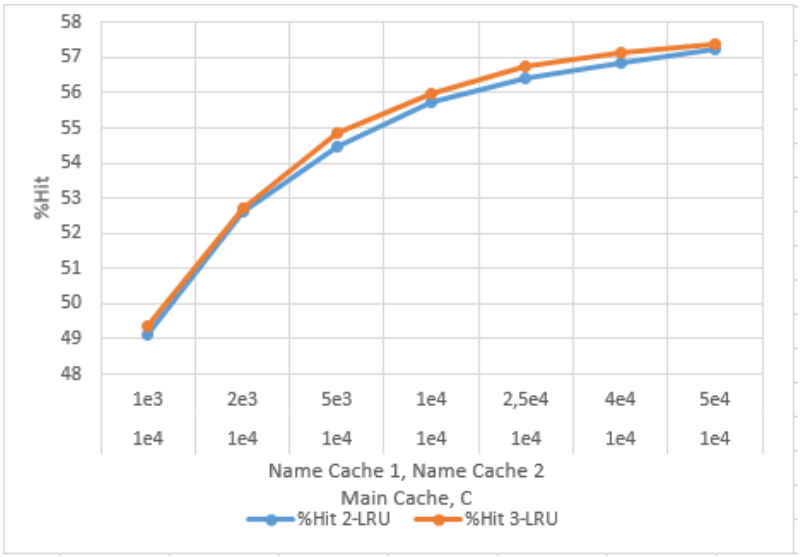


Figure 4.32. Hit probability vs small cache size, 2-LRU vs 3-LRU, under SNM, cache networks, $T_{off} = 8 \times T_{on2}$.

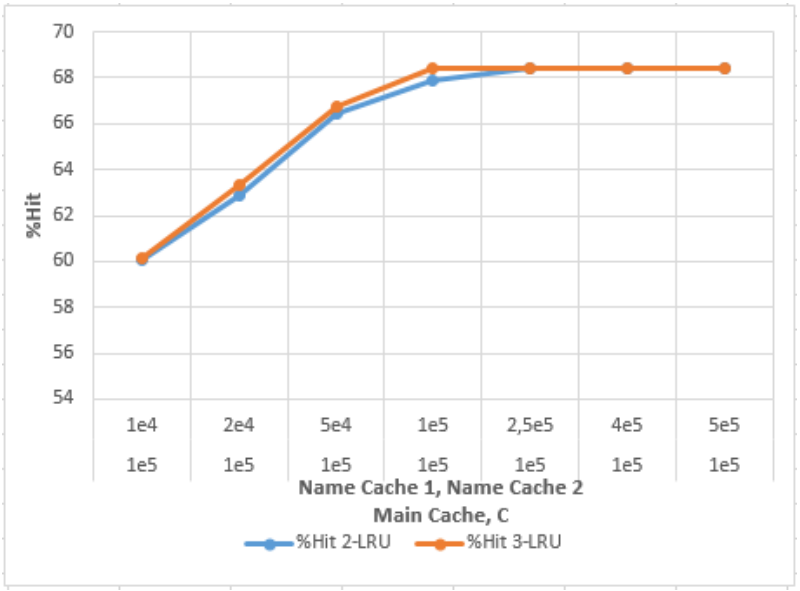


Figure 4.33. Hit probability vs large cache size, 2-LRU vs 3-LRU, under SNM, cache networks, $T_{off} = 8 \times T_{on2}$.

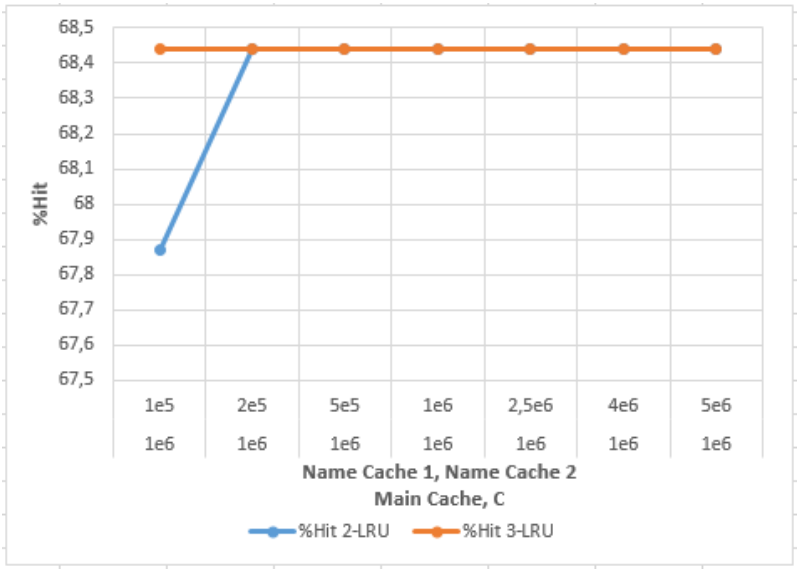


Figure 4.34. Hit probability vs very large cache size, 2-LRU vs 3-LRU, under SNM, cache networks, $T_{off} = 8 \times T_{on2}$.

Chapter 5

Performance evaluation of "cache-decoupled" 2-LRU vs k-LRU, under SNM and Pareto distribution

5.1 Introdution

In this chapter I investigated the performance of the optimized "cache-decoupled" 2-LRU vs k-LRU, under SNM model and under Pareto probability distribution, according *event-driven* technique.
In order to do it, I sustituded the Zipf's law in place of the Pareto Cumulative Distribution Function (CDF) within ccnSim program. The average number of request V attracted by contents follows the Pareto distribution with probability density: $f(v) = \beta {V^{\beta}}_{min}/v^{1+\beta}$[58] for $v \geq V_{min}$ (recall that the second moment of the Pareto distribution is finite for $\beta \geq 2$)[15]. The value of λ associated to contents of the fixed catalogue produced an average number of request during an ON period given by $E[V] = \lambda_m T_{on}$ which has the same distribution as the number of requests produced by SNM shots.
The choise of a Pareto distribution for V_m is justified by the fact that Zipf-like distribution is obtained when a large number of content requests are independently generated according to a Pareto distribution with exponent β[15].
As in chapter 3, I evaluated 2-LRU and for the sake of semplicity 3-LRU (i.e k-LRU, with $k = 3$). I investigated the basic case of single cache fed by a single-class SNM traffic model, by considering T_{on} and T_{off} periods associated to number of contents depicted in Fig. 5.1. Again the number of contents belonging to the catalogue size is large enough to consider the system ergodic, even if λ_m is the same for all the ON periods associated to content m[15]. For the experiments presented in this section, I investigated the performance of 2-LRU and 3-LRU with just two Pareto's exponents: $\beta = 2$, $\beta = 1.5$ and we fixed the average number of requests for each content to $E[V] = 20$.
All the simulations are performed by keeping fixed the main cache and by varyng the size of the Name Cache (case 2-LRU), or of the two Name Caches (case 3-LRU) for a minimum of 10% to a maximum of 500% of the main cache.

Ton2=50*Ton1(s)	Toff=k*Ton2(s)	Content
2160000	2160000 (k=1)	62500
2160000	17280000 (k=8)	62500

Figure 5.1. Settings for Pareto SNM scenario.

5.1.1 Implementation of Pareto distribution

I initialized the Pareto's CDF: $F(v) = 1 - V_{min}/v^{\beta}$[58], according to the specified catalog cardinality; i) the parameter x describes the content popuplarity v, and has the same access of Zipf's; ii) V_{min} is set equal to 1; iii) the parameter $Number0fElements$ describes the content cardinality.

```
void pareto_distribution::pareto_initialize(unsigned long long x, unsigned
    long long NumberOfElements)
{
 // Return if the cdf has been already initialized.
 if (cdfPareto.size() != 0)
       return;

       double c = 0;
       double num = 0;

       cout<<"Initializing Pareto distribution of Class # "" class_num
           <<""..."<< endl;

       // Initialization of the Pareto's CDF according to the specified
           catalog cardinality
       if (x <= 0 || x > NumberOfElements)
       return 0.0;
              {
              c = 1.0 - (1.0/ (pow(x, beta))); // Vmin has been set equal to 1
              }
 }
```

The parameter β is retrieved by the program through the function:

```
unsigned int pareto_distribution::get_beta()
{
       return beta;
}
```

We can note that in the program we have taken into account also the case of different content class, however for the sake of semplicity we have investigated just the case of single class of contents.

5.2 Validation of k-LRU "cache-decoupled", under SNM and Pareto distribution

Pareto distribition tryies to emulate the content profiles in a realistic scenario (e.g., Youtube videos), this is the reason why achieves *hit* further far from 1 than Zipf's distribution, since Zipf's law describes very well long-term number of requests attracted by each content[15]. Furthrmore in order to produce significant hit differences we have to tune the β exponent larger than Zipf's α one.

- $T_{off} = T_{on1}$

Even under Pareto distribution, cache performance are deeply impacted by the average life-span of contents, T_{on}, indeed for a given cache size, the *hit* probability is roughly inversely proportional to T_{on}[43].

Small cache sizes

To investigate 2-LRU and 3-LRU, fed by single cache under Pareto's law, we obtained low performance in terms of *hit*, especially when they were evaluated with small cache sizes. By comparing the case of $\beta = 2$ (Fig. 5.2) and $\beta = 1.5$ (Fig. 5.3), we observed that with $\beta = 1.5$, both policies performed better than the case of larger exponent $\beta = 2$, although they didn't achieve high *hit*. In this case the *cache-decoupling* optimization provided about 3% of *gain* in total, in both β cases. By increasing the the main cache size to 10^4, we achieved the same *overall gain* of 3% for both 2-LRU and 3-LRU, and for both values of β; this is an important resut, because we increased the performance's policies, by keeping the same hit *gain*, as showed in the case of of $\beta = 2$ (Fig. 5.4) and $\beta = 1.5$ (Fig. 5.5).

Large cache sizes

As expected, the distribution of the number of requests attracted by contents (V), plays a significant rule on the cache performance, thus large cache sizes required to achieve high hits probability. Even β plays an important rule on the cache performance, indeed, decreasing β leads to better *hit*. The comparison between the impact of β on 2-LRU and 3-LRU are showed in Fig. 5.6 and F.7 in the case of main cache size, $C = 10^5$, and in Fig. 5.8 and Fig. 5.9 in the case of main cache size, $C = 10^6$. Our optimization worked quite well, until cache sizes $\leq 10^5$, especially in the case of small β values, indeed they got $\approx 2.5\%$ with $\beta = 2$, whereas not too well with $\beta = 1.5$, i.e., $\approx 1\%$, as *total gain*. With cache sizes $\geq 10^6$, the *cache-decoupling* impact provided negligible benefits.

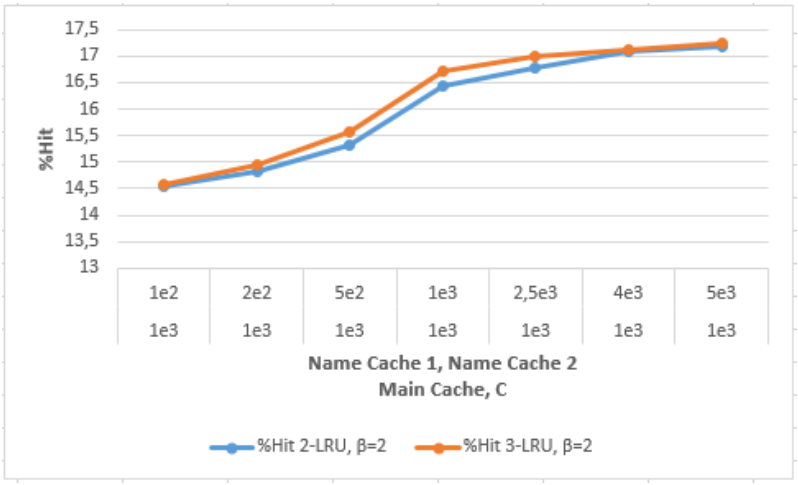


Figure 5.2. Hit probability vs very small cache size, 2-LRU vs 3-LRU, under SNM, $\beta = 2$, $T_{off} = T_{on1}$.

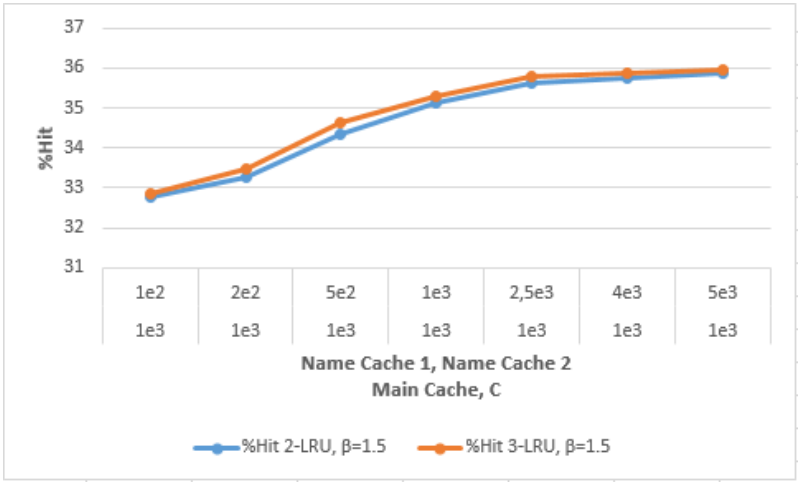


Figure 5.3. Hit probability vs very small cache size, 2-LRU vs 3-LRU, under SNM, $\beta = 1.5$, $T_{off} = T_{on1}$.

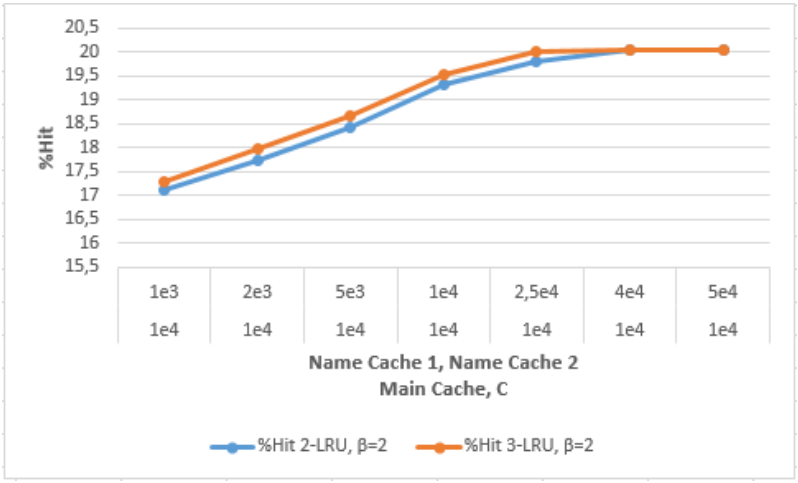


Figure 5.4. Hit probability vs small cache size, 2-LRU vs 3-LRU, under SNM, $\beta = 2$, $T_{off} = T_{on1}$.

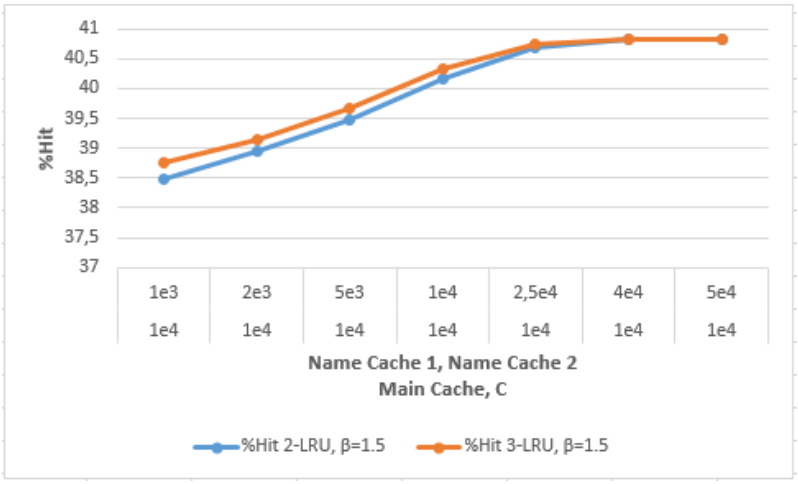


Figure 5.5. Hit probability vs small cache size, 2-LRU vs 3-LRU, under SNM, $\beta = 1.5$, $T_{off} = T_{on1}$.

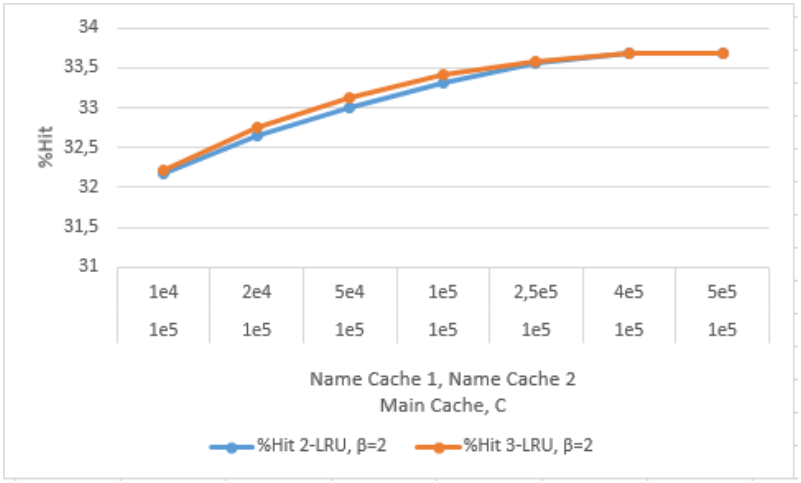


Figure 5.6. Hit probability vs large cache size, 2-LRU vs 3-LRU, under SNM, $\beta = 2$, $T_{off} = T_{on1}$.

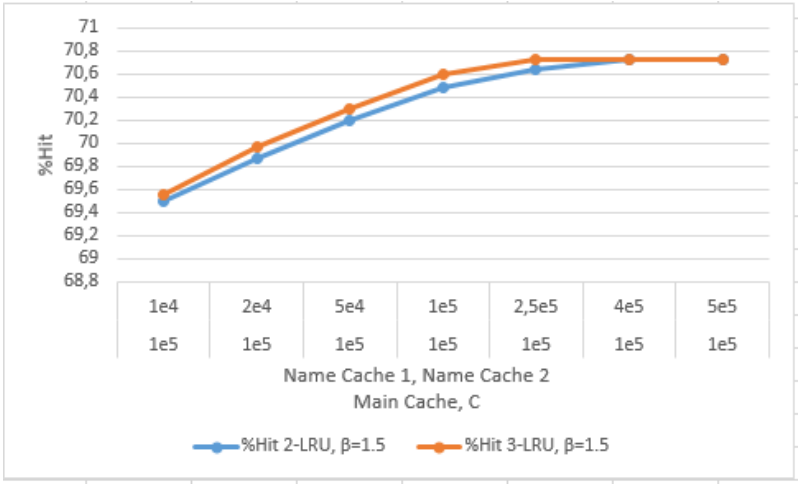


Figure 5.7. Hit probabilitiy vs large cache size, 2-LRU vs 3-LRU, under SNM, $\beta = 1.5$, $T_{off} = T_{on1}$.

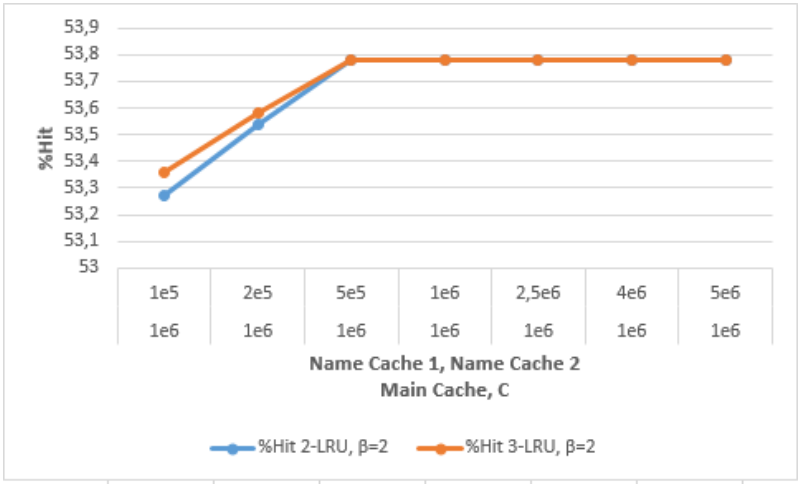


Figure 5.8. Hit probability vs very large cache size, 2-LRU vs 3-LRU, under SNM, $\beta = 2$, $T_{off} = T_{on1}$.

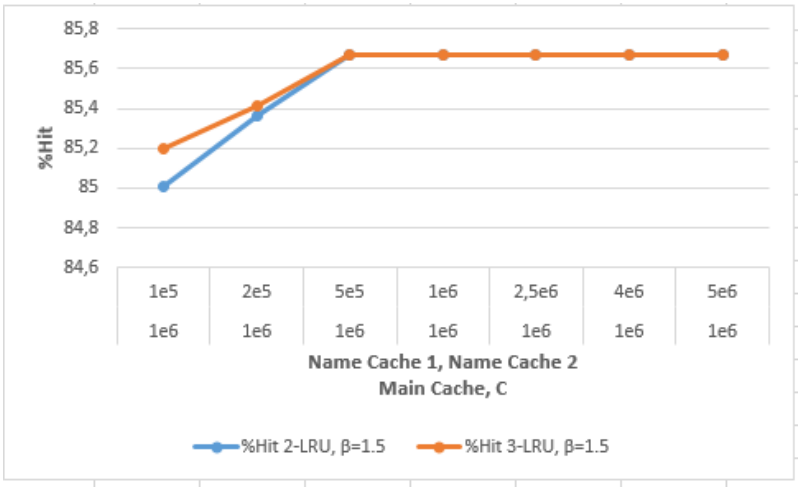


Figure 5.9. Hit probability vs very large cache size, 2-LRU vs 3-LRU, under SNM, $\beta = 1.5$, $T_{off} = T_{on1}$.

- $T_{off} = 8 \times T_{on1}$

Definitly better results are obtained when the T_{off} period was increased 8 times life-span T_{on1}, but however worse than Zipf's law content distribution.

Small cache sizes

In the case of small cache sizes, both 2-LRU and 3-LRU reached better *hit* than shorter T_{off}, by confirming how decreasing β exponent leads to better performance, as showed in Fig. 10 (i.e., $\beta = 2$) and Fig.11 (i.e., $\beta = 1.5$).
Our improvement allowed to preserve the same *gain* as $T_{off} = T_{on1}$, for both β exponents, with the advantage of perceiving better *hit*, that was our aim.
Furthermore we observed that the policies's curves, with $\beta = 1.5$, arised constantly, instead with $\beta = 2$, it didn't occur.
The same concept valids stronger in the case in which the main cache size was increased to a power of 10, i.e., 10^4: by looking Fig. 12, with $\beta = 2$, we can observe that 2-LRU and more evident 3-LRU achieved as maximum *hit*, about 30%, due to an incresing of sizes of both Name Cache 1 and Name Cache 2, thanks to the *cache-decoupling*, whereas with $\beta = 1.5$, the increase of the two Name Caches size over the main cache one, leaded to a maximum *hit* greater than 52%, showed in Fig. 13.

Large cache sizes

When we manage large cache sizes, the combination of wide cache size and large T_{off} allow to got very optimal performance. Pareto distribution against Zipf's law, guarantees more stable results, thus in the citical case of large cache sizes, the *hits* obtained through simulations should more reliable.
The performance obtained by 2-LRU and 3-LRU, in the case of main cache, $C = 10^5$, are showed in Fig. 5.14 and 5.15, respectively with $\beta = 2$, and $\beta = 1.5$. By making a comparison, we can *emphasize*, how the performance given by $\beta = 1.5$ outperform those one given by $\beta = 2$, with the an average difference *gain* of about 30%. We deduced that the our optimization, in case of very large cache sizes, worked bad, especially in scenarios of small β, as depicted in Fig. 5.16 and Fig. 5.17.
To conclude, decreasing popularity distribution β, guarantees significant benefits on the caching systems performance, especially for systems based on LRU strategies, but the *cache-decoupling* optimization works better with larger β values.

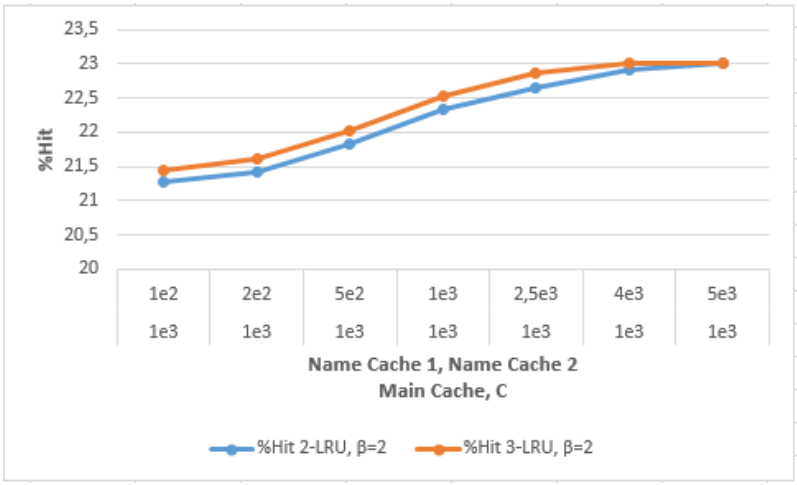


Figure 5.10. Hit probability vs very small cache size, 2-LRU vs 3-LRU, under SNM, $\beta = 2$, $T_{off} = 8 \times T_{on1}$

.

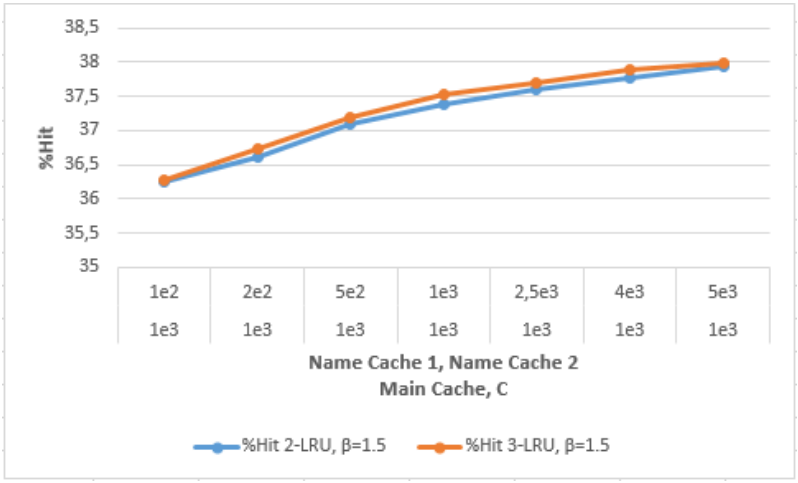


Figure 5.11. Hit probability vs very small cache size, 2-LRU vs 3-LRU, under SNM, $\beta = 1.5$, $T_{off} = 8 \times T_{on1}$.

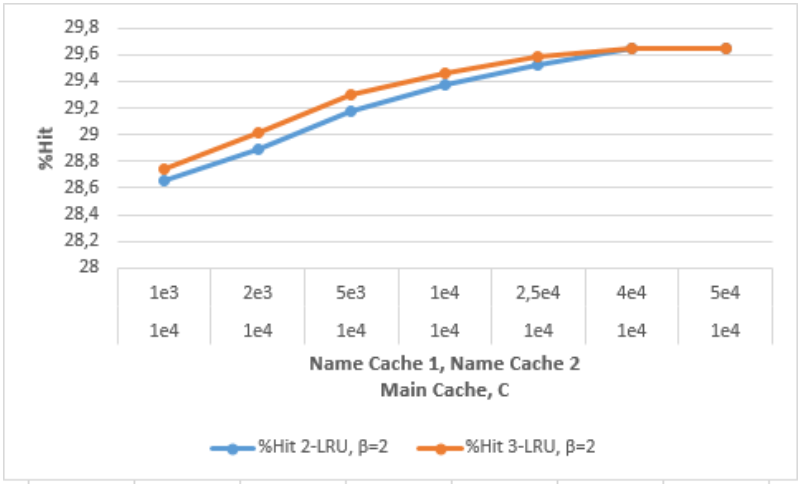


Figure 5.12. Hit probability vs small cache size, 2-LRU vs 3-LRU, under SNM, $\beta = 2$, $T_{off} = 8 \times T_{on1}$.

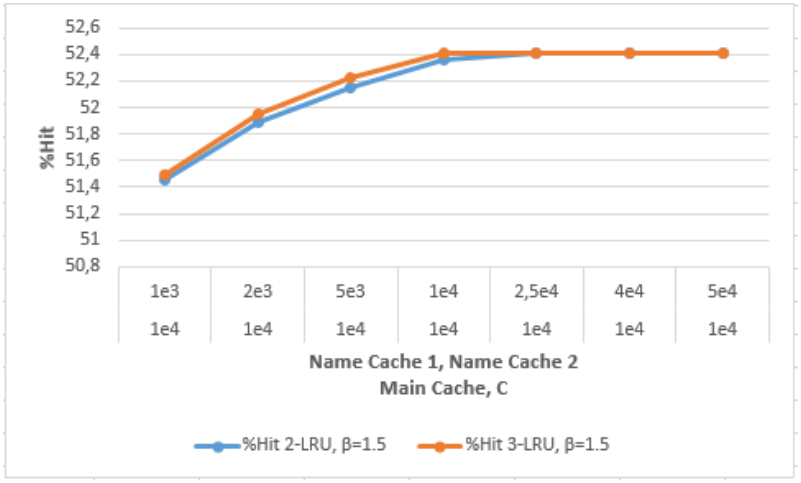


Figure 5.13. Hit probability vs small cache size, 2-LRU vs 3-LRU, under SNM, $\beta = 1.5$, $T_{off} = 8 \times T_{on1}$.

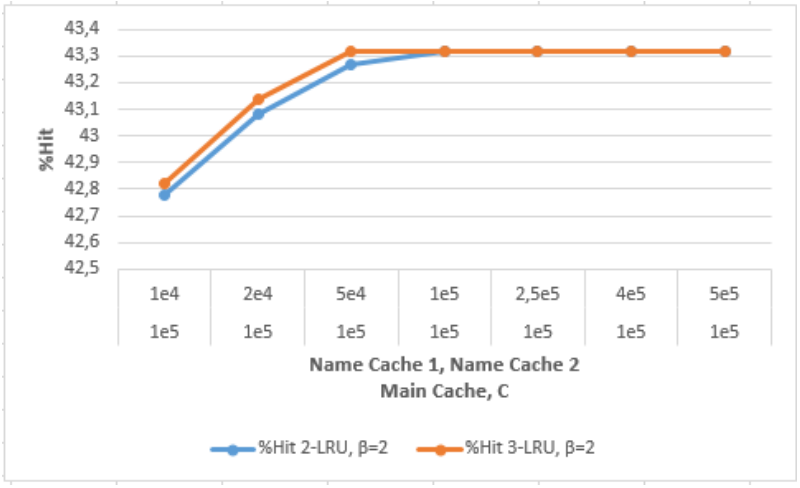


Figure 5.14. Hit probability vs large cache size, 2-LRU vs 3-LRU, under SNM, $\beta = 2$, $T_{off} = 8 \times T_{on1}$.

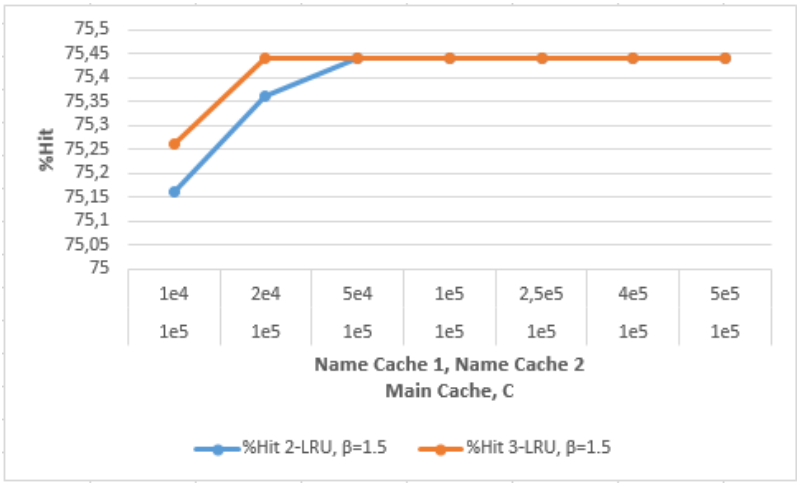


Figure 5.15. Hit probability vs large cache size, 2-LRU vs 3-LRU, under SNM, $\beta = 1.5$, $T_{off} = 8 \times T_{on1}$.

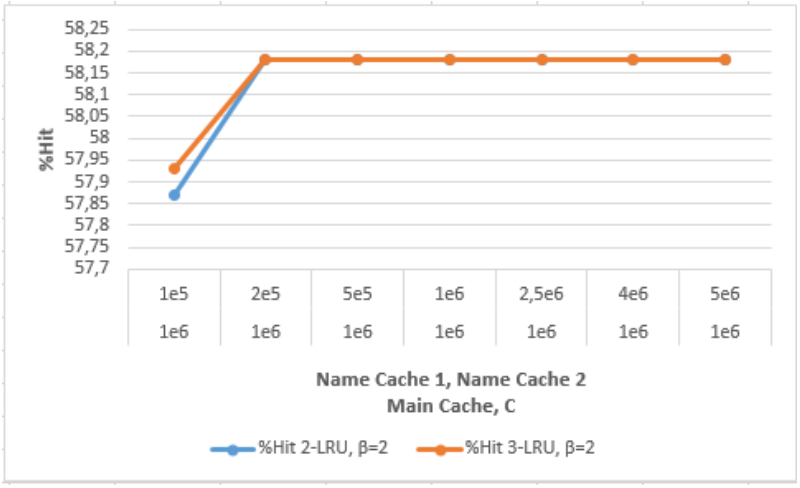


Figure 5.16. Hit probability vs very large cache size, 2-LRU vs 3-LRU, under SNM, $\beta = 2$, $T_{off} = 8 \times T_{on1}$

.

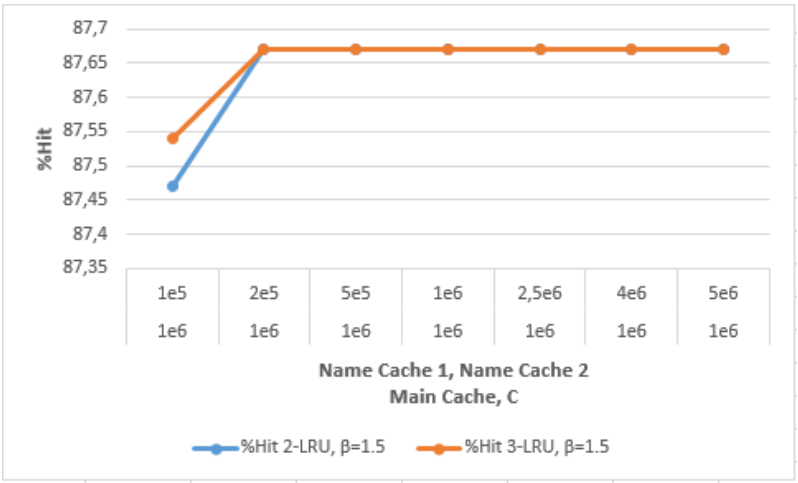


Figure 5.17. Hit probability vs very large cache size, 2-LRU vs 3-LRU, under SNM, $\beta = 1.5$, $T_{off} = 8 \times T_{on1}$.

5.3 Results Stabilization

We would garantee that results obtained through simulations are however stabile. A pratical approach, to ensure that at a given arbitrary time t, $P_t(\lambda^{'}_{mON} = \lambda_{mON})$, consists to extract a new value for $\lambda^{'}_{mON}$ at every new arriving content request everytime a given content enter in ON period (i.e. whenever λ_{mOFF} passes to λ_{mON}), since λ_{mOFF} is always 0.
We opportunely set T_{off} much larger than the cache eviction time T_C, (since is a free parameter) such that at the end of the every OFF period, during the next ON period, the performance in terms of hit probability produced by the object m, is exactly the same as if it was a completely new content made available in the caching system at the beginning of the following next ON period, by ensuring $P(\lambda^{'}_{mON} = \lambda_{mON})$[15][29].

5.4 Insights

We can claim that the distribution of content request volumes plays an important role on cache performance: the hit probability increases when the popularity distribution has a heavier tail, hence as we decrease β exponent of Pareto, by making the popularity distribution less and even less sloped[15]. However $\beta = 2$ behaves as threshold, because the variance of the content request volumes is finite, indeed the impact on cache performance when $\beta > 2$ is somewhat limited[43].
This fact highlight the significant difference with respect to the classical IRM traffic model, where the impact of the Zipf's law exponent of content popularity is always very large over its whole domain, this means that a small variation of α produces a high impact on cache performance[19].
The *cache-decoupled* optimization produces better benefits in scenarios of small cache sizes and larger β values.

Chapter 6

Conclusions

The main goal of this work is to show the performance of a variety of caching systems (both isolated and interconnected caches) operating under various insertion/eviction policies and traffic conditions, analysed within a Content Networks (CCN) simulator called ccnSim and based on Che's approximation[11].
We have also shown the performance of policies based on LRU decision strategy (i.e., k-LRU), in particular 2-LRU and 3-LRU, with the *cache-decoupling* optimization, under IRM traffic and SNM traffic model, by empolying different content popularity distributions (i.e. Zipf, Pareto).
Our study has revealed the superiority of the k-LRU policy, in terms of both simplicity and performance gains, although 2-LRU achieves much of the possible gain yet.

The most part of this work is referred to Shot Noise Model because provides a simple, flexible and accurate approach to describing the temporal and geographical locality found in Video-on-Demand traffic, allowing us also to develop accurate analytical models of cache performance[43].
Our main insights by investigating the k-LRU performance under SNM are: i) content locality plays also an important role in the distributed caching systems and should not be neglected; ii) larger T_{off} periods (or shorter life-span T_{on}) achieve better performance, since decrease the number of concurrent active contents at the same time; iii) the overall impact on cache performance of the distribution of the average number of requests attracted by the contents (and the corresponding distribution) is significantly affected by temporal locality[15] with respect to traditional traffic models (e.g., IRM) and at which life-span period λ_m is generated by the content distribution (e.g., λ_{mON}); iv) especially when caches are small, performance can be significantly improved by reserving access into the cache only to contents which are highly cacheable[19][43]: in order to obtain this information needs to exploit the contents popularity distribution which gives their profile.

From the point of view of our *cache-decoupling* optimization, the main insigths are: i) it works very well on 2-LRU, whereas only with small cache sizes on 3-LRU, since 2-LRU policy obtains much of the possible gain yet; ii) it works better with not too large cache sizes (tipically $\leq 10^4$) and either small content popularity distribution Zipf's α or large Pareto's β, especially for $k > 2$ (e.g., 3-LRU); iii) it works better under cache network when the starting performance are worse: when we have already high-performed scenarios, such optimization provides negligible gain; iv) it works well under both IRM and SNM traffic model, indeed, it mitigates also the content dynamic profiles (when content popularity change and evolve over time); v) strong temporal locality affects adversely on the benefits introduced by such optimization.

Appendix A

Implementation of optimized 2-LRU class

```
class Two_Lru: public DecisionPolicy
{
 public:
       Two_Lru(uint32_t cSize):ncSize(cSize){
              base_cache* bcPointer = new lru_cache(); // Create a new LRU
                  cache that will act as a Name Cache.
              name_cache = dynamic_cast<lru_cache *> (bcPointer);
              name_cache->set_size(ncSize);} // Set the size of the Name
                  Cache.

       virtual bool data_to_cache(ccn_data *)
       {return true;}

       bool name_to_cache(chunk))
       {
              if name_cache->lookup_name(chunk))
              {
              // The ID is already present inside the Name Cache, so update
                  its position and return True. As a consequence, the
                  'cacheable' flag will be set to 1.

              // HIT - Update the timestamp of the relative content
              if(nc_stable)
              {
                     map<chunk_t, double>::iterator itHit =
                         monitored_contents.find(chunk);
                     if (itHit == monitored_contents.end())
                     {
                     exit(1);
              }
              monitored_contents[chunk] = SIMTIME_DBL(simTime());
              }
              return true;
       }
       else
       {
```

```
            // The ID is NOT present inside the Name Cache, so insert it
                and return False.
            // As a consequence, the 'cacheable fag will be set to '0'.

            // MISS - Insert the new elememt
            if (nc_stable)
            {
                    if (current_size == ncSize) // The LRU element will be
                        discarded
                    {
                            chunk_t k = (name_cache->get_lru())->k;
                            map<chunk_t, double>::iterator it =
                                monitored_contents.find(k);

                            if (it == monitored_contents.end())
                            {
                            exit(1);
                            }
                            else
                            {
                            monitored_contents.erease(k);
                            }
                    }
                    //insert the new element inside the map
                    map<chunk_t, double>::iterator it =
                        monitored_contents.find(chunk);
                    if(it == monitored_contents.end())
                    {
                            monitored_contents[chunk]= SIMTIME_DBL(simTime());
                    }
                    else
                    {
                    cout << "The entry is present despite the previous miss
                        event!\n"
             exit(1);
            }
               }
               if (current_size < ncSize)
                       current_size++;

                       name_cache->store_name(chunk);
                       return false;
               }
       }

       lru_cache* name_cache;
       map <chunk_t, double> monitored_contents;
       bool nc_stable = false;

       private:
       uint32_t ncSize; // Size of the Name Cache in terms of number of
           content IDs.
       uint32_t current_size = 0;
};
```

Appendix B

Implementation of optimized k-LRU class

```
class K_Lru: public DecisionPolicy
{
public:
       K_Lru(uint32_t cSize):ncSize(cSize){
              base_cache* bcPointer = new lru_cache(); // Create a new LRU
                  cache that will act as a Name Cache.
              name_cache = dynamic_cast<lru_cache *> (bcPointer);
              name_cache->set_size(ncSize);} // Set the size of the Name
                  Cache.
              // This have to be implemented for k-1 levels of Name Cache
              base_cache* bckPointer = new lru_cache(); // Create k-1 new LRU
                  cache that will act as a k-1 Name Cache.
              namek_cache = dynamic_cast<lru_cache *> (bcPointer);
              namek_cache->set_size(ncSize);} // Set the size of the k-th
                  Name Cache.

       virtual bool data_to_cache(ccn_data *)
       {return true;}

       bool namek_to_cache(chunk))
       {
              if namek_cache->lookup_name(chunk))
              {
              // The ID is already present inside the Name Cache, so update
                  its position and return True. As a consequence, the
                  'cacheable' flag will be set to 1.

              // HIT - Update the timestamp of the relative content
              if(nc_stable)
              {
                     map<chunk_t, double>::iterator itHit =
                         monitored_contents.find(chunk);
                     if (itHit == monitored_contents.end())
                     {
                     exit(1);
              }
```

```
            monitored_contents[chunk] = SIMTIME_DBL(simTime());
            }
            return true;
    }
    else
    {
            // The ID is NOT present inside the Name Cache, so insert it
                and return False.
            // As a consequence, the 'cacheable fag will be set to '0'.

            // MISS - Insert the new elememt
            if (nc_stable)
            {
                    if (current_size == ncSize) // The LRU element will be
                        discarded
                    {
                            chunk_t k = (name2_cache->get_lru())->k;
                            map<chunk_t, double>::iterator it =
                                monitored_contents.find(k);

                            if (it == monitored_contents.end())
                            {
                            exit(1);
                            }
                            else
                            {
                            monitored_contents.erease(k);
                            }
                    }
                    //insert the new element inside the map
                    map<chunk_t, double>::iterator it =
                        monitored_contents.find(chunk);
                    if(it == monitored_contents.end())
                    {
                            monitored_contents[chunk]= SIMTIME_DBL(simTime());
                    }
                    else
                    {
                    cout << "The entry is present despite the previous miss
                        event!\n"
             exit(1);
            }
               }
               if (current_size < ncSize)
                       current_size++;

                       name2_cache->store_name(chunk);
                       return false;
               }
    }

    lru_cache* name_cache;
// Create many lru_cache* pointer as k-1 name cache
lru_cache* namek_cache;
    map <chunk_t, double> monitored_contents;
    bool nc_stable = false;
```

```
        private:
        uint32_t ncSize; // Size of the Name Cache in terms of number of
            content IDs.
        uint32_t current_size = 0;
};
```

Bibliography

[1] W. Jiang, S. Ioannidis, L. Massouli and F. Picconi, "Orchestrating massively distributed CDNs," in ACM CoNEXT, 2012.

[2] V. Jacobson, D. K. Smetters, J. D. Thornton, M. F. Plass, N. H. Briggs, and R. L. Braynard, "Networking named content," in ACM CoNEXT, 2009.

[3] W. King, Analysis of Paging Algorithms. IBM-Report, 1971.

[4] A. Dan and D. Towsley, "An approximate analysis of the LRU and FIFO buffer replacement schemes," SIGMETRICS Perform. Eval. Rev., vol. 18, Apr. 1990.

[5] H. Che, Y. Tung, and Z. Wang, "Hierarchical Web caching systems: modeling, design and experimental results," IEEE JSAC, vol. 20, no. 7,2002.

[6] C. Fricker, P. Robert, and J. Roberts, "A versatile and accurate approximation for lru cache performance," ITC, 2012.

[7] E. Rosensweig, J. Kurose, and D. Towsley, "Approximate Models for General Cache Networks," in INFOCOM, 2010.

[8] M. Gallo, B. Kauffmann, L. Muscariello, A. Simonian, and C. Tanguy, "Performance evaluation of the random replacement policy for networks of caches," SIGMETRICS Perf. Eval. Rev., vol. 40(1),2012.

[9] G. Bianchi, A. Detti, A. Caponi, and N. Blefari Melazzi, "Check before storing: what is the performance price of content integrity verification in lru caching?," SIGCOMM Comput. Comm. Rev., vol. 43,July 2013.

[10] M. Tortelli, D. Rossi, and E. Leonardi. "A hybrid methodology for the performance evaluation of internet-scale cache networks." Elsevier Computer Networks, Special Issue on Softwarization and Caching in NGN, 2017.

[11] V. Martina, M. Garetto, E. Leonardi. "A unified approach to the performance analysis of caching systems" in INFOCOM, 2014.

[12] V. Almeida, A. Bestavros, M. Crovella, and A. de Oliveira, "Characterizing reference locality in the www," in IEEE PDIS, 1996.

[13] S. Jin and A. Bestavros, "Sources and characteristics of web temporal locality," in IEEE MASCOTS, 2000.

[14] S. Traverso, M. Ahmed, M. Garetto, P. Giaccone, E. Leonardi, and S. Niccolini, "Temporal locality in today's content caching: Why it matters and how to model it," SIGCOMM Comput. Commun. Rev., vol. 43, Nov. 2013

[15] M. Garetto, E. Leonardi, and S. Traverso, "Efficient analysis of caching strategies under dynamic content popularity," in Infocom'15, 2015.

[16] F. Olmos, B. Kauffmann, A. Simonian, and Y. Carlinet, "Catalog dynamics: Impact of content publishing and perishing on the performance of a lru cache," in ITC, IEEE, 2014.

[17] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker, "Web caching and zipf-like distributions: evidence and implications," in INFOCOM, 1999.

[18] M. Cha, H. Kwak, P. Rodriguez, Y.-Y. Ahn, and S. Moon, "Analyzing the Video Popularity Characteristics of Large-Scale User Generated Content Systems," IEEE/ACM Trans. on Netw., vol. 17(5), 2009.

[19] C. Fricker, P. Robert, J. Roberts, and N. Sbihi, "Impact of traffic mix on caching performance in a content-centric network," in IEEE NOMEN Workshop, 2012.

[20] T. Johnson and D. Shasha, "2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm," in VLDB, 1994.

[21] P. R. Jelenkovi and A. Radovanovi , "The persistent-access-caching algorithm," Random Struct. Algorithms, vol. 33, Sept. 2008.
[22] G. Rossini and D. Rossi, "Coupling caching and forwarding: Benefits, analysis, and implementation," in ICN '14, (New York, NY, USA), ACM, 2014.
[23] P. R. Jelenkovi and and X. Kang, "Characterizing the miss sequence of the lru cache," SIGMETRICS Perform. Eval. Rev., vol. 36, Aug. 2008.
[24] E. Gelenbe, "A unified approach to the evaluation of a class of replacement algorithms," IEEE Trans. Comput., vol. 22, June 1973.
[25] A. Finamore, M. Mellia, M. Meo, M. M. Munafo' , and D. Rossi, "Experiences of Internet traffic monitoring with Tstat," IEEE Network, 2011.
[26] N. C. Fofack, P. Nain, G. Neglia, and D. Towsley, "Performance evaluation of hierarchical TTL-based cache networks," Computer Networks, vol. 65, 2014.
[27] ccnSim-v0-4 User Manual. http://perso.telecom-paristech.fr/~drossi/ccnSim.
[28] Van Jacobson, Diana K. Smetters, James D. Thornton, Michael F. Plass, Nicholas H. Briggs, and Rebecca L. Braynard. Networking named content. In ACM CoNEXT, page 1?12, 2009.
[29] M. Tortelli, D. Rossi, and E. Leonardi. ModelGraft: Accurate, Scalable, and Flexible Performance Evaluation of General Cache Networks. In Proc. of International Teletraffic Congress (ITC), Wurzburg, Germany, Sep. 2016.
[30] http://www.omnetpp.org.
[31] S. Arianfar and P. Nikander." Packet-level Caching for Information-centric Networking". In ACM SIGCOMM, ReArch Workshop. 2010.
[32] F. Baccelli, S. Machiraju, et al. "On optimal probing for delay and loss measurement". In Proc of the ACM SIGCOMM IMC. 2007.
[33] M. Branicky, V. Borkar, et al." A unified framework for hybrid control: model and optimal control theory". IEEE Transactions on Automatic Control, 1998
[34] M. Cha, H. Kwak, et al. "I tube, you tube, everybody tubes: analyzing the world's largest user generated content video system". In ACM IMC. 2007.
[35] H. Che, Y. Tung, et al. "Hierarchical web caching systems: Modeling, design and experimental results". IEEE JSAC), 2002
[36] H. Che, Z. Wang, et al. "Analysis and design of hierarchical web caching systems". In Proc of IEEE INFOCOM. 2001.
[37] G. Rossini and D. Rossi. ccnSim: "A highly scalable CCN simulator". In Proc. of IEEE ICC. 2013.
[38] N. Fofack, P. Nain, et al. "Performance evaluation of hierarchical TTL-based cache networks". Elsevier Computer Networks, 2014.
[39] C. Fricker, P. Robert, et al. "A versatile and accurate approximation for LRU cache performance". In Proc. of International Teletraffic Congress (ITC 24). 2012.
[40] N. Gast and B. V. Houdt. "Transient and steady-state regime of a family of list-based cache replacement algorithms"". In Proc of ACM SIGMETRICS Con- ference. 2015.
[41] e. a. H. Casanova. "Versatile, scalable, and accurate simulation of distributed applications and platforms". Journal of Parallel and Distributed Computing, 2014.
[42] M. Hefeeda and O. Saleh. "Traffic modeling and proportional partial caching for peer-to-peer systems". IEEE/ACM Transactions on Networking, 2008
[43] S. Traverso, M. Ahmed, P. Giaccone, E. Leonardi, M. Garetto, S. Niccolini. "Unravelling the Impact of Temporal and Geographical Locality in Content Caching Systems"
[44] http://www.boost.org/
[45] E. Leonardi, G. Torrisi. "Modeling LRU caches with Shot Noise request process"
[46] T. Johnson, D. Shasha, et al. 2q: "A low overhead high performance buffer management replacement algorithm". In 20th International Conference on Very Large Data Bases (VLDB). 1994.
[47] W. King. "Analysis of paging algorithms". In In IFIP Congress. 1971.
[48] N. Laoutaris, H. Che, et al. "The LCD interconnection of LRU caches and its analysis". Performance Evaluation, 2006.
[49] E. Leonardi and G. Torrisi." Least Recently Used caches under the Shot Noise Model". In Proc. of IEEE Infocom. 2015.

[50] P. Levis, N. Lee, et al. TOSSIM: “Accurate and Scalable Simulation of Entire” TinyOS Applications. In Proc. of ACM SenSys. 2003.
[51] Y. Liu, F. Presti, et al. “Scalable Fluid Models and Simulations for Large-scale IP Networks”. ACM Trans. Model. Comput. Simul., 2004.
[52] D. Berger et al. “Exact Analysis of TTL Cache Networks: The Case of Caching Policies Driven by Stopping Times”. In Proc. of ACM SIGMETRICS Confer- ence, 2014.
[53] S. Fayazbakhsh et al. “Less Pain, Most of the Gain: Incrementally Deployable” ICN. SIGCOMM Comput. Commun. Rev., 2013.
[54] N. Fofack et al. “On the performance of general cache networks”. In Proc. of VALUETOOLS Conference, 2014.
[55] K. Pentikousis et al. “Information-centric etworking: Evaluation methodology”. Internet Draft.
[56] M. Rosenblum et al. “Complete Computer System Simulation: The SimOS Approach”. IEEE Parallel Distrib. Technol., 3(4):34, 1995.
[57] G. Xylomenos et al. “A survey of information-centric networking research”. Com- munication Surveys and Tutorials, IEEE, 2014.
[58] M. Meo, M. Munafò: “Random-Variable Generators”
[59] R. Pan, B. Prabhakar, et al. SHRiNK: “A Method for Enabling Scaleable Performance Prediction and Efficient Network Simulation”. IEEE/ACM Trans. Netw., 2005.