Caching
TBD

MT: Write a small introduction that explains the content of the TP.

PART 1 - Caching: What, Why, and How

MT: Explain: (i) what does it mean to "cache a content"; (ii) why (how the hit ratio influences the user experience) and where it is used; (iii) which factors contribute to caching performance (i.e., replacement and admission policies, catalog properties like content popularity, request model, and so on.)

...

The *hit ratio* h, and so the resulting user experience connected to the performance of a caching system, is the result of multiple factors interplaying with each other, as shown in Fig. MT: Add a figure with a sum of all factors giving the final hit ratio. Well known classes of each factor are listed in the following.

Content Popularity. The context a caching system is used in, and especially *how* requests are issued for contents identified by a particular *popularity distribution*, represents one of the influencing factors for the final hit ratio h.

- **Zipf's law**

  It is considered as the de facto popularity distribution (along with Weibull and Mandelbrot-Zipf ones) that best approximates the dynamics of common Internet services, from popularity of requested contents in a P2P network or in a CDN, to distribution of requests for websites, or popularity of videos in a streaming platform.

  In particular, denoting with M the *catalog cardinality*, and with $1 \leq i \leq M$ the rank of the i-th most popular content, the probability of requesting the content with rank i is expressed as:

  $$P(X = i) = \frac{i^{-\alpha}}{B}. \tag{1}$$

  where $B = 1/\sum_{j=1}^{M} j^{-\alpha}$, and $\alpha > 0$.

  Clearly, the value of the Zipf exponent α plays a paramount role in determining the caching performance; indeed, the percentage of requests directed to the C most popular contents, i.e., $P(C) = \sum_{i=1}^{C} P(X = i)$, heavily varies with α. For large values of α, e.g., $\alpha = 2$, and supposing a catalog with a cardinality of $M = 10^8$ contents, $C = 12$ represents the content correspondent to the 95-th percentile of requests, meaning that 95% of requests are directed towards only the $C = 12$ most popular contents (as $\sum_{i=1}^{12} P(X = i) > 0.95 \quad for \quad \alpha = 2$). At the same time, with $\alpha = 0.8$, for example, the head of the catalog for which the 95% of requests is directed becomes much wider, being $C =$ MT: complete. It can be concluded that the higher α, the smaller the size of the cache needed to guarantee a certain hit ratio h.

  It is worth noticing, also, that the whole Internet traffic is made of a mixture of different "types" of traffic (Web, P2P, Video, etc.), each one with a specific set of parameters, content popularity (expressed in terms of request frequency) being one of them. This is the reason why there is neither a single α value that characterizes all the scenarios above, nor a broad consensus on the particular α that needs to be use to model each type of traffic. Empirical studies have tried to provide intervals for each application: for example, for Web and File Sharing traffic $\alpha \in [0.6, 0.85]$, for VoD $\alpha \in [0.65, 1]$, and for UGC $\alpha \geq 2$ [7].

Temporal Locality. The temporal dynamics of popularity, i.e., how requests are distributed in time, complement the information provided by popularity distribution, which expresses only the number of requests submitted for each content, without describing how these requests are distributed in time. This aspect is of primary importance when considering the performance of

caching schemes since the ordering of the requests obviously affects the contents that will be stored inside the cache. There exists several models used to characterize the pattern of requests arriving at a cache:

- **Independent Reference Model (IRM)**

  This model, widely adopted in the literature due to its simplicity, is based on the following fundamental assumptions: i) users request items from a fixed catalog of M objects; ii) the probability $P(X = m)$ 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 past requests, generating an i.i.d. sequence of requests. It results that the IRM completely ignores all temporal correlations in the sequence of requests (aka temporal locality), i.e., the fact that, if an object is requested at a given point in time, then it is more likely that the same object will be requested again in the near future (a characteristic that increase the hit ratio h of a cache).

- **Shot Noise Model (SNM) [4]**

  The basic idea is to represent the overall request process as the superposition of many independent processes (shots), each referring to an individual content. Specifically, the arrival process of requests for a given content m at a cache is described by an inhomogeneous Poisson process of intensity $V_m h(t - t_m)$, where V_m denotes the average number of requests attracted by the content, t_m is the time instant at which the content enters the system (i.e., it becomes available to the users), and $h()$ is the (normalized) "popularity profile" of content m. The SNM models, in this way, the evolution of content popularity over time.

Spatial Locality. It refers to the way requests are geographically distributed throughout the network. Being able to characterize request patterns in different areas of the network might help avoiding redundant traffic in the network itself; indeed, if requests are highly localized in some areas of the entire network, then similar requests can be more efficiently served with mechanisms such as caching.

Replacement Policy. Since caches have a limited amount of space to store contents (which we suppose being equal to C contents), we need a *replacement policy* to decide which content needs to be evicted when the cache is full and a new content needs to be stored. Among several techniques, the most adopted and studied ones are:

- **LFU**: in its classic implementation, a *Least Frequently Used* cache keeps track of the access frequency by assigning a counter to each stored content, and evicting the one with the lowest counter when the cache is full. Furthermore, if the content popularity is known a priori, a LFU cache can be implemented by statically storing the C most popular contents, thus providing optimal performance under IRM. However, a pure LFU system is discouraged in real contexts with evolving popularity (e.g., newly cached items with low counters might be soon evicted even though they might be required frequently thereafter).

- **LRU**: a *Least Recently Used* cache will evict the content that has not been requested for the longest time, in case a new content needs to be inserted in the cache, and there are already C contents stored in it. This means that each time a cached content is requested, its timer is updated. A LRU cache provides the best compromise between performance and easy implementation.

- **FIFO**: in this case, the timer of cached contents is not updated in case of a cache hit; as a consequence, the content that has been inserted since the longest time is evicted if the cache is full.

- **RANDOM**: the content to be evicted in case of a full cache is selected randomly.

Admission Policy. The effective storage of a newly arrived content inside the cache can be regulated by an *admission policy*, which might be related or not to the "position" of the cache itself inside the whole network topology. The most diffused techniques are:

- **LCE**: with *Leave Copy Everywhere*, a newly arrived content is aways cached, if not already stored inside the cache.
- **LCP**: *Leave Copy Probabilistically* adds a filter to LCE, in the sense that a newly arrived content will be cached only with probability p, a parameter that can be set in order to change the caching behavior and reduce the so called *cache pollution* (i.e., the worthless caching of unpopular contents).
- **2-LRU**: a more affective way than LCP to reduce cache pollution is provided by this strategy [6]; before the physical cache, where actual objects are stored, there is a second cache where only content IDs of incoming requests are stored following a LRU policy. Therefore, only contents whose ID is found in the first cache are then cached in the physical one when fetched from upstream nodes. This provides a self-tuning filter that reduces cache pollution and which can cope with the temporal evolution of content popularity.
- **LCD**: in a hierarchical network of caches, the *Leave Copy Down* strategy leaves a copy of the requested content only inside the cache which is one hop below the cache the content has been found in.

PART 2 - Cost and errors of computing Cache Hit Ratio h

With notions of PART 1, we are now ready to go one step further and have a quick overview of the available analytical tools used to compute the *cache hit ratio* h. In particular, we are interested in their computational complexity and in assessing potential errors that can be introduced when relying on simplifications. Due to the numerous and diverse scenarios that might result from the combination of the different factors seen above, in this TP we will make the following assumptions:
(i) we consider a system composed by a *single cache* under
(ii) an IRM request model, and
(iii) we focus on the specific aspect of *replacement policies*, considering LFU and LRU as case studies.

LFU. As seen in PART 1, the quantile of the Zipf distribution for an object of rank C of a catalog with cardinality M can be written as:

$$P(C) = \frac{\sum_{k=1}^{C} 1/k^{\alpha}}{\sum_{k=1}^{M} 1/k^{\alpha}}, \tag{2}$$

which also represents the aggregate rate for the C most popular contents. Considering that a single LFU cache with size C statically stores the C most popular contents, it results that (2) also corresponds to the *cache hit ratio* h of that cache:

$$h_{LFU} = P(C) \tag{3}$$

For big values of M, (3) becomes computationally complex. However, except for $\alpha > 1$, it is possible to find a closed form expression for h_{LFU} by using some simplification.

- $\alpha = 1$

  In this case, numerator and denominator of 2 correspond to the Harmonic numbers $H_C = \sum_{k=1}^{C} 1/k$ (and H_M), which are known to be well approximated asymptotically by a logarithmic function, since

  $$\lim_{X \to \infty} H_X - ln(X) = \gamma, \tag{4}$$

  where $\gamma \approx 0.57$ is the Euler-Mascheroni constant. Hence, in the large C and M regime, we have:

  $$h_{LFU}^{app} \approx ln(C)/ln(M). \tag{5}$$

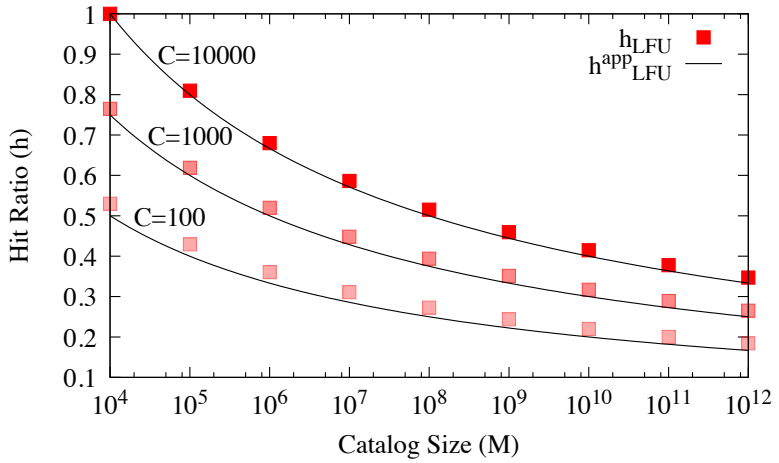

Figure 1: Single LFU cache with fixed size C and varying cardinality M. True hit ratio h_{LFU} vs its approximation h_{LFU}^{app} for $\alpha = 1$.

Fig. 1 compares the true hit ratio h (3) for a fixed size cache C and varying M against its approximation (5). It can be noticed that despite (i) this approximation works as better as C and M increase, (ii) a non-negligible error persists in every case. There is, indeed, a compromise to accept between computational ease and prediction error.

- $\alpha < 1$

  In this particular case, the harmonic number can be approximated as:

$$H_C \approx \int_1^C 1/x^\alpha dx = \frac{x^{(1-\alpha)}}{1-\alpha} = \frac{C^{(1-\alpha)}}{1-\alpha} \tag{6}$$

  It follows that:

$$h_{LFU}^{app} \approx \frac{C^{(1-\alpha)}}{M^{1-\alpha}} = \left(\frac{C}{M}\right)^{1-\alpha} \tag{7}$$

  It is worth noticing that the approximation (7) comes with a higher absolute error (i.e., $|h_{LFU} - h_{LFU}^{app}|$) with respect to the approximation (5) MT: Insert an explicative plot.

LRU. The computational complexity of exactly predicting the cache hit ratio h for a single cache with LRU replacement policy lies in the exponential growth with cache size C or with catalog cardinality M [5], which makes impossible to obtain the final results in a reasonable amount of time for real scenarios (i.e., with $M \geq 10^8$). Therefore, the only feasible way to obtain a reference point is by simulation. However, plenty of models and approximations have been proposed so far, among which Che's approximation [3] represents the most adopted and effective one. It was initially conceived for a single cache with LRU replacement policy and IRM traffic, and then successively extended for other traffic types, replacement and admission policies [6].

- **Che's Approximation [3]**

  The core intuition is that of considering the eviction time $T_C(m)$ for content m, with $1 \leq m \leq M$, as if it was a constant value T_C, known as "characteristic time", independent of the content m. In particular, $T_C(m)$ can be considered for a cache of size C as the time needed before C distinct contents (excluding m) are requested, i.e., the time since the last request (for m) after which content m will be evicted (provided that it has not been requested in the meanwhile).

Che's assumption greatly simplifies the analysis of caching systems because it allows to decouple the dynamics of different contents: the interaction among contents is summarized by T_C, which acts as a single primitive quantity representing the response of the whole cache to an object request.

Formally speaking, Che's approximation states that an object m is in the cache at time t, if and only if a time smaller than T_C has elapsed since the last request for object m, i.e., if at least a request for m has arrived in the interval $(t - T_C, t]$. 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 given by:

$$p_{in}(m) = 1 - e^{-\lambda_m T_C} \tag{8}$$

It is worth noticing that $p_{in}(m)$ represents also the hit probability $p_{hit}(m)$ (thanks to PASTA's property), i.e., the probability that a request for content m finds content m inside the cache. Denoting with $\mathbb{1}_{\{\mathrm{A}\}}$ the indicator function for event A, we have that by construction, cache size C must satisfy:

$$C = \mathbb{E}\left[\sum_m \mathbb{1}_{\{m \text{ in cache at t}\}}\right] = \sum_m p_{in}(\lambda_m, T_C). \tag{9}$$

It follows that the characteristic time T_C can be computed by numerically inverting (9), which admits a single solution[3].

The *average hit probability* h of a LRU cache is then:

$$h = \sum_m p_m p_{hit(m)} \tag{10}$$

PART 3 - Use cases: how much h influences the User Experience

MT: Use cases can be file retrieval, DNS resolver, and so on.

Installation

In this section we will cover all the steps needed for the installation of ccn$\mathcal{S}$im-Parallel. We suggest the reader to check and execute them by following the presentation order.

Portability

ccn$\mathcal{S}$im-Parallel has been tested on the following platforms, and with the following softwares:

- Ubuntu Linux 14.04 (64-bit)
- Ubuntu Linux 16.04 Server (64-bit)
- Omnetpp-4.6
- Omnetpp-5.0
- Akaroa-2.7.13

Prerequisites

- **Boost libraries $\geq$ 1.54**: they can be installed either by using the standard packet manager of your system (e.g., apt-get install, yum install, port install, etc.), or by downloading them from `http://www.boost.org/users/download/`, and following instructions therein.
- **gcc > 4.8.1**: on Ubuntu platforms, gcc can be updated by adding the ubuntu-toolchain-r/test PPA. Sample commands for the latest 5.x version are:

```
sudo add-apt-repository ppa:ubuntu-toolchain-r/test
sudo apt update
sudo apt install gcc-5 g++-5
sudo update-alternatives --install /usr/bin/gcc gcc /usr/bin/gcc-5 60
        --slave /usr/bin/g++ g++ /usr/bin/g++-5
```

- **parallel-ssh**: in order to ease the simulation workflow, a passwordless SSH login needs to be present on all the available servers. *parallel-ssh* can than be used in order to send commands to slaves from the master node.

```
sudo apt-get install pssh
```

Akaroa2©

As mentioned above, Akaroa2© requires a specific license in order to be used. For instructions about software download and licenses, please visit the website `https://akaroa.canterbury.ac.nz/akaroa/`. The design of ccn*S*im-Parallel, concerning the Akaroa component, has started from the Akaroa-2.7.13 version. The whole set of modifications that have been made in order to implement the parallel ModelGraft strategy are grouped inside the *Akaroa_2.7.13_for_PMG.patch* patch, which is freely available at `http://perso.telecom-paristech.fr/~drossi/ccnSim`.

Once obtained the original licensed Akaroa-2.7.13, place it in the same directory with the aforementioned patch, and type:

```
patch -s -p0 < Akaroa_2.7.13_for_PMG.patch
```

Notice that the original Akaroa directory will keep the same name even after the application of the patch.

Once the patch is applied, type the following commands from the root directory of the patched Akaroa in order to install it:

```
./configure
make
sudo make install
```

Akaroa binaries, libraries, and include files will be installed in /usr/local/akaroa/bin, /usr/local/akaroa/lib, and /usr/local/akaroa/include, respectively. If you *do not have sudo privileges*, and you need to install Akaroa into another location, e.g., <install_path>, please add this location to your PATH variable, and use the following commands:

```
./configure --prefix=<install_path>
```

```
make
make install
```

Furthermore, always in the case of lacking sudo privileges, users might need to modify LIB and PYTHON paths in the several "Makefile.config_*" files and inside the "pyconfig.py" file according to the customized locations of related libraries, before compiling and installing Akaroa.

Omnet++

ccn$\mathcal{S}$im-Parallel has been tested with both *Omnetpp-4.6* and *Omnetpp-5.0*, which are available at `https://omnetpp.org/`. Before the installation, Omnetpp-x requires prerequisite packages to be installed. For the complete list, please read the official installation manual. Furthermore, since Omnetpp-x bin/ directory needs to be added to the PATH variable, the following line should be put at the end of the `~/.bashrc` file:

```
export PATH=$PATH:$HOME/omnetpp-x/bin
```

After having closed and saved the file, please close and re-open the terminal.
Regardless of the Omnetpp version, a couple of patch files need to be applied to it before compilation and installation. These patch files are provided with ccn$\mathcal{S}$im-Parallel, which we suppose being correctly downloaded (see next section) and decompressed in the $CCNSIM_DIR. Provided that Omnetpp is present in the $OMNET_DIR, specific instructions to patch, compile, and install it without the graphical interface (not needed for parallel ModelGraft simulations) are provided in the following, according to the selected version:

- Omnetpp-4.6

  ```
  pint:~$ cd $CCNSIM_DIR
  pint:CCNSIM_DIR$ cp ./patch/omnet4x/ctopology.h $OMNET_DIR/include/
  pint:CCNSIM_DIR$ cp ./patch/omnet4x/ctopology.cc $OMNET_DIR/src/sim/
  pint:CCNSIM_DIR$ cd $OMNET_DIR/
  pint:OMNET_DIR$ NO_TCL=1 ./configure
  pint:OMNET_DIR$ make
  ```

- Omnetpp-5.0

  ```
  pint:~$ cd $CCNSIM_DIR
  pint:CCNSIM_DIR$cp ./patch/omnet5x/ctopology.h $OMNET_DIR/include/omnetpp
  pint:CCNSIM_DIR$cp ./patch/omnet5x/ctopology.cc $OMNET_DIR/src/sim
  pint:CCNSIM_DIR$ cd $OMNET_DIR/
  pint:OMNET_DIR$ ./configure WITH_TKENV=no
  pint:OMNET_DIR$ make
  ```

Compilation flags can also be set through the file *configure.user* (e.g., to disable Qtenv).

ccn$\mathcal{S}$im-Parallel

As anticipated above, the .tgz file containing the ccn$\mathcal{S}$im-Parallel code can be obtained from `http://perso.telecom-paristech.fr/~drossi/ccnSim`. After having decompressed it in the $CCNSIM_DIR folder, installation requires the following steps:

```
./scripts/makemake.sh
make
```

If all the steps described in the previous sections have been successfully executed, simulations of general cache networks using the parallel ModelGraft technique can now be launched. The several steps needed to simulate sample scenarios will be described in the following.

Example-1: "Trillion" made possible!

In this section we will describe all the steps needed to simulate a Web-scale scenario, i.e., comprising a catalog with a trillion contents. It is an extreme scenario which highlights the capabilities of the new parallel ModelGraft technique. For an extended and thorough performance evaluation, which considers scenarios with different cardinalities, we refer the reader to our technical report [?citation?].

Scenario, bare-metal, and scripts

Scenario. We consider a CDN-like topology composed of an Abilene core network, and several 4-level binary trees attached to it, thus representing access networks. By using the resulted topology, which comprises 67 nodes in total, we simulate an initial non-downscaled catalog of $M = 10^{12}$ contents, and through an Independent Reference Model (IRM), i.e., i.i.d. requests, we consider $R = 10^{12}$ total requests as a non-downscaled initial point. We consider nodes having non-downscaled Least Recently Used (LRU) caches of $C = 10^8$ contents. By following guidelines provided in [8], we set the *downscaling* factor of the ModelGraft technique at $\Delta = 10^7$. As a consequence, we effectively simulate a *downscaled* catalog of $M' = 10^5$ contents, with Time-to-Live (TTL) caches with a downscaled cache size of $C' = 10$, and we generate a downscaled number of $R' = 10^7$ requests.

Bare-metal. The dedicated cluster used for the experiment comprises $NS = 3$ Cisco UCS-B series servers, each one hosting 2 NUMA nodes, with a Xeon E5-2690 CPU, $NC = 12$ physical cores per NUMA node (i.e., 48 CPUs in total with hyperthreading) operating at 2.60GHz, and 378 GB of RAM memory. For ease of description we call our servers "modelgraftX", where $X \in \{1, 2, 3\}$, and we suppose that a *modelgraft* user is present in each of them. We also remind that it is important to have a passwordless SSH login on all the available servers. For this particular scenario, we report results related to simulations done with $NT = 64$ parallel threads and $NS = 2$ physical servers.

Scripts. The main file used to launch the simulation of the described scenario is:

```
(1) ./launch_Parallel_ModelGraft_1e12.sh
```

This *bash script* will appear inside the Akaroa folder after having applied the provided Akaroa_2.7.13_for_PMG.patch patch. In order to make it executable you need to type `chmod +x launch_Parallel_ModelGraft_1e12.sh`. It is important to notice that:

- The script is supposed to be executed from one of the servers used for the simulation, meaning that both Akaroa and ccn$\mathcal{S}$im-Parallel are installed on the same machine. If executed from

a different machine than the servers used for the simulations, the script will need some modifications.

- The script has been conceived considering the specified pool of 3 servers, i.e., modelgraft1, modelgraft2, and modelgraft3, with a common "modelgraft" user present on all of them;
- If using different servers and username, the script needs to be modified accordingly. Sometimes it is more practical to use server names instead of IP addresses; in order to do that, entries like "x.x.x.x servername" should be added in the "/etc/hosts" file of all the servers.

We strongly suggest to carefully read both the extensively documented script (1) for a detailed description of all its components and commands, and the User Manual of ccnSim-v0.4 [2] for insights related to its structure and to the ModelGraft technique.
In summary, script (1) will:

I. specify the number of *physical servers* and *parallel threads* that will be used to run the parallel ModelGraft;

II. set all the parameters needed to define the simulated scenario;

III. automatically probe and allocate resources on the available servers according to their load (i.e., the least loaded ones are considered);

IV. launch Akaroa daemons (both Master and Slaves) on the reserved resources;

```
${akaroaBinDir}/akmaster &
${akaroaBinDir}/akslave &
```

V. call the script "runsim_script_Parallel_ModelGraft.sh" dedicated to ccnSim-Parallel simulations.

```
(2) ./runsim_script_Parallel_ModelGraft.sh {parameters}
```

All the scenario parameters defined within script (1) are passed in the form of command line parameters to script (2), which is located in $CCNSIM_DIR. For a complete list of all the possible parameters please refer to the User Manual of ccnSim-v0.4 [2].

We strongly suggest to carefully read the extensively documented script (2) for a detailed description of all its components and commands, and the User Manual of ccnSim-v0.4 [2] for insights related to its structure and to the ModelGraft technique.
In summary, script (2) will:

I. check if the required T_C file is already available; otherwise it will create a new one with *random* T_C values, which will be distributed to all the available servers;

II. create a new .ini file according to {parameters};

III. launch the parallel ModelGraft simulations through the Akaroa APIs;

IV. collect and elaborate results.

Point I. is worth to be discussed: since the ModelGraft technique [8] makes use of TTL caches with an eviction timer set according to the *characteristic time* T_C [3] of the respective LRU caches, files reporting the T_C value of each node in the simulated topology are needed as input. Since in most of the real scenario users do not know T_C values a priori, the ModelGraft technique is able to iteratively converge to a consistent state through a feedback loop, even when accepting random

T_C values as input. As a consequence, script (2) will firstly check if a user-provided T_C file is already present, and if not, it will randomly generate values for all the nodes, collect them in one file, and distribute it to all the available servers (since all the threads instantiated over multiple servers need to have a common T_C file as a starting point). This file will be automatically erased from all the servers if randomly generated (i.e., user-defined ones will not be erased).

As for the management of the output files produced by all the parallel threads, the policy adopted by the current version of script (2) is that of *removing them all from each server, thus saving only a SUMMARY file*, namely "ALL_MEASURES*", which will be kept in the *${CCNSIM_DIR}/${resultDir}/parallel* folder of the server where script (2) has been executed from (which, by default, is modelgraft1). If you want/need to change this policy, please follow instructions provided in the script (2) file.

Results

Results are presented in Fig. ?reference?, which reports both CPU time and memory occupancy on a logarithmic axis; in particular, results related to three different strategies are compared: classic event-driven (ED) simulation, single-thread ModelGraft, and the last parallel ModelGraft technique. As it can be noticed, simulating such a huge scenario would be prohibitive if relying on the classic event-driven approach, considering that we would need more than one year of CPU time and more the 7 TBytes of RAM. That is the reason why we report only projected results, computed from linearly interpolating results obtained for smaller scenarios. Since the introduction of the single-threaded ModelGraft technique [8], instead, memory occupancy is not considered as a bottleneck anymore (notice that only 27 MB are required); as a consequence, CPU time represented the only bottleneck (i.e., for this scenario, we would need more than 2 days of simulation). The new parallel ModelGraft technique, indeed, aims at overcoming the CPU bottleneck by parallelizing the simulation over multiple threads and physical servers. In this particular case, by instantiating $NT = 64$ parallel threads over $NS = 2$ physical servers, we notice from Fig. ?reference? a dramatic decrease in the CPU time, i.e., 9056× and 42× compared to classic ED simulation and single-threaded ModelGraft, respectively. As for the memory occupancy, instead, we can observe that while parallelizing the simulation by creating exact replicas of network and catalog increases the required memory, the total amount is still considerably slow and it can also be split over multiple servers. At the same time, the parallel ModelGraft technique still considerably reduces the memory occupancy w.r.t. ED simulation by up to 4065×.

Example-2: Parallel MG vs MG vs Classic ED Simulation

The Web-scale scenario seen before is treatable only by using the scalability and performance of the parallel ModelGraft technique, meaning that we actually miss other points of comparison. As a consequence, in this section we compare the new parallel ModelGraft technique against the previous single-threaded one, and against the classic event-driven (ED) simulation (both available with ccn$\mathcal{S}$im-v0.4 [2]), by considering a smaller scenario as a reference.

In particular we simulate a 4-level binary tree (i.e., 15 nodes) topology, with a non-downscaled catalog cardinality of $M = 10^9$, cache size $C = 10^6$, $R = 10^9$ total requests, and a downscaling factor of $\Delta = 10^5$. This scenario can be simulated by using the provided script

```
./launch_Parallel_ModelGraft_1e9.sh
```

Results

Results reported in Tab. ?reference? highlight three main considerations: (i) the parallelization of the ModelGraft technique *does not introduce further inaccuracies*, it (ii) allows to *drastically reduce the CPU time* of a further order of magnitude with respect to the single-threaded ModelGraft, while (iii) still *requiring a reasonable amount of memory* with respect to the number of instantiated

parallel threads, i.e., 32 in this case (it is worth considering that the total memory demand can be spread over multiple machines when multiple servers are available).

Customization

General Information

Users interested in experimenting and ameliorating the implementation of the parallel ModelGraft technique can refer to the CHANGELOG.txt file (present in both patched Akaroa folder and ccn$\mathcal{S}$im-Parallel folder) where a list enumerating all the modified files is reported. The goal is that of facilitating the orientation of newcomers into both Akaroa and ccn$\mathcal{S}$im-Parallel codebases. Nevertheless, for a deeper understanding of all the elements included in the parallel ModelGraft technique, we strongly encourage users to read (i) Akaroa documentation [1], ccn$\mathcal{S}$im-v0.4 documentation [2], ModelGraft paper [8], the parallel ModelGraft technical report [?citation?], and all the well documented scripts present in both patched Akaroa folder and ccn$\mathcal{S}$im-Parallel folder.

Applicability

For a complete list and discussion on all the possible scenarios the parallel ModelGraft technique can be used for, please refer to Section 3.1 of the ccn$\mathcal{S}$im-v0.4 manual [2].

Encoding of Messages Exchanged between Master and Slaves

The patched Akaroa-2.7.13 needs to define a MAX_MSG_LENGTH variable for the messages which are exchanged between Master and Slaves. In addition, one of the modifications introduced in order to let Akaroa and ccnSim-Parallel communicate and be compliant with each other, is that of including a *vector*, namely "HitMissVector", inside the exchanged messages. This vector should contain info about hit and miss events collected by the parallel and independent slaves, which are then sent to the Master node.

For the ease of implementation, both MAX_MSG_LENGTH and HitMissVect have been sized according to the biggest simulated topology. In practice, since each node in the network will send $2 \times engineWindow = 2 \times (W/NT)$ Hit and Miss samples, where W is the initially dimensioned window size, and NT is the number of allocated parallel threads, the size of the *aggregated* HitMissVect sent by each parallel thread will be equal to $vectSize = N \times 2 \times engineWindow$, where N is the total number of nodes. Since $W = 100$ by default, and since the largest topology that we simulated is the CDN-like one (i.e., with 67 nodes), the aforementioned variables are set at compile time as:

char HitMissVect[57000] (inside 'src/include/checkpoint.H' file of Akaroa)
#define MAX_MSG_LEN 57500 (inside src/ipc/connection.H' file of Akaroa)

The HitMissVect size (in Bytes) has been obtained by $N \times 2 \times engineWindow \times 4$, supposing $NT = 1$ (i.e., the case with the biggest vector). The exact value should be 53600 Bytes, so the used one is slightly increased for safety reasons. The size of the MSGs that should carry also the HitMissVect is, then, set accordingly to $MAX\_MSG\_LEN = 57500$ (i.e., 500 Bytes more to reserve space for other values transmitted within the same message).

It is IMPORTANT to notice that, if using other values then $N = 67$, $W = 100$, which bring to a bigger HitMissVect than 57000 Bytes (the equation is always $N \times 2 \times (W/NT)$), please modify the relative *checkpoint.H* and *connection.H* files, and recompile Akaroa (make; sudo make install). If smaller values are set w.r.t. the real ones, segmentation fault problems might happen.

References

[1] Akaroa Project Website. `https://akaroa.canterbury.ac.nz/akaroa/`.

[2] ccnsim website. `http://perso.telecom-paristech.fr/~drossi/ccnSim`.

[3] H. Che, Y. Tung, and Z. Wang. Hierarchical web caching systems: Modeling, design and experimental results. *IEEE JSAC*, 20(7):1305–1314, 2002.

[4] M. Garetto, E. Leonardi, and S. Traverso. Efficient analysis of caching strategies under dynamic content popularity. In *Proc. of IEEE INFOCOM*, Apr. 2015.

[5] W. F. King. Analysis of paging algorithms. In *Proc. of IFIP Congress*, Aug. 1971.

[6] V. Martina, M. Garetto, and E. Leonardi. A unified approach to the performance analysis of caching systems. In *Proc. of IEEE INFOCOM*, Apr. 2014.

[7] K. Pentikousis et al. Information-centric Networking: Evaluation and Security Considerations. RFC, Sep. 2016.

[8] 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.