Proactive Computation Caching Policies For 5G-and-Beyond Mobile Edge Cloud Networks

Nicola di Pietro and Emilio Calvanese Strinati
CEA, LETI, MINATEC, F-38054, Grenoble, France
E-mail: {nicola.dipietro, emilio.calvanese-strinati}@cea.fr

***Abstract*—Computation caching is a novel strategy to improve the performance and the quality of service of mobile edge cloud networks. It consists in storing in local memories situated at the edge of the network, here mobile access points, the already processed results of computations that users offload to the mobile edge cloud. The goal of this technique is to avoid redundant and repetitive processing of the same tasks, thus streamlining the offloading process and improving the exploitation of both users' and network's resources. In this paper, three different computation caching policies are proposed and evaluated. They are based on three main parameters: the popularity of offloadable tasks, the size of their inputs, and the size of their results. Numerical simulations show that good policies need to take into account these three parameters altogether.**

I. Introduction

The future of mobile communications will be characterized by ubiquitous connection availability, very dense networks, ultra-low latency, and energy efficiency. The exchange of data and information will be extremely fast, copious, and secure. The 5G network revolution will be enabled by cutting-edge technological innovations, concerning millimeter-wave radio communications, baseband and RF architecture, resources virtualization, and much more. A game-changing idea consists in empowering network mobile terminals with intensive data elaboration and storage capabilities, thus bringing cloud support the closest possible to users. This paradigm is called Mobile Edge Cloud (MEC) [10], [11], [21]. A rich research current focuses on techniques for optimizing allocation and exploitation of MEC resources, usually divided into three main categories: communication, computing, and caching resources [2], [3], [13], [14], [16], [19], [21].

In MEC networks, Serving Small Cells (SSCs) endowed with radio access technology, computing units, and local cache memories can be charged by User Equipments (UEs) to run computing tasks on their behalf. The procedure of entrusting these computational assignments to small cells is called *task* or *computation offloading*. It allows UEs to save both time and energy and revolutionizes the classical interaction between UEs and mobile terminals.

The role of content caching in MEC networks is critically important and deeply investigated [1], [4], [7], [21]. In the context of task offloading, a new form of caching was recently introduced, after noticing the pointlessness of repeating many times the same computation for the same reiterated offloading request. This paradigm is called *computation caching* [15], [17], [18] and suggests to exploit small cells' memory to store the results of offloadable computations [6], so that they can be simply retrieved from the cache instead of being recomputed each time they are requested. The goal is to decimate redundant and repetitive processing and has several advantages, e.g., drastically reducing computation time and saving energy for both UEs and SSCs, preventing uplink bottlenecks, freeing network resources and decreasing the SSCs' workload, diminishing the number of virtual machine instantiations. The consequent resource gains can be reinvested to optimize the network performance and increase users' satisfaction. Although computation caching can be applied to several aspects of 5G-and-beyond networking, in this paper we focus on the interaction between a single UE and its SSC. This is the first fundamental stage on which to build scenarios with more users, more small cells, and complex interactions among them [14], [16].

The goal of this paper is to introduce, evaluate, and compare three different computation caching policies that depend on task popularity. These policies are enablers for *proactive* computation caching [6], intended as the strategy of dynamically adapting the content of cache memories, based on the continuous learning of task popularity and other statistics. The leading concept is that future offloading traffic can be predicted and computation caching can be proactively adjusted to smoothly react to traffic fluctuations. The goal of this paper is not to describe how to learn and predict task popularity, but rather to show how it can be exploited to design efficient and effective computation caching policies.

An important novelty of this work is the introduction of a policy that simultaneously takes into account three different characterizing parameters of an offloadable task: its popularity, the size of its input, and the size of its result. In particular, this last parameter plays a crucial role: in computation caching, the size of the data to cache and to download (the task result) can be significantly different from the size of the data to upload from the UE to the SSC (the task input). This marks a sharp difference with classical content caching, in which cached data essentially have the same size of the corresponding data travelling through the network. Our main contribution is to show that this difference can be exploited to significantly improve the effectiveness of computation caching. With Lemma 1, we also mathematically characterize the optimal caching policy when the costs of uploading and elaborating the input data are proportional to their size and all the task results have the same standard size.

In Section II, we describe the system model and fix some notation. Then, three computation caching policies are formally defined in Section III and numerically evaluated in Section IV. Some concluding remarks are outlined in Section V.

II. System Model

In our setting, a UE offloads computational tasks to the MEC via its SSC. The communication rates are denoted R_{UL} in uplink and R_{DL} in downlink and are measured in bits per second. We suppose that the computational capacity of the SSC is f CPU cycles per second and that the SSC can store up to m bits to perform computation caching on a local memory.

Offloadable tasks belong to a finite set $\mathcal{C} = \{c_1, \ldots, c_K\}$, that we call the computation *catalogue*. A task $c_k \in \mathcal{C}$ is represented by a triplet: $c_k = (W_k, e_k, W'_k)$, where W_k is the input data (a sequence of bits) to be processed, e_k is the number of CPU cycles per bit needed to elaborate the data, and W'_k is the computation result (another sequence of bits). We denote $|W_k|$ and $|W'_k|$ the sizes in bits of W_k and W'_k.

Definition 1 (Cache Indicators): We call *cache indicator* the vector $\sigma = (\sigma_1, \ldots, \sigma_K) \in \{0,1\}^K$ such that $\sigma_k = 1$ if and only if the result W'_k of $c_k \in \mathcal{C}$ is stored in the SSC's cache. Thus, a cache indicator fully identifies the cache content.

Definition 2 (Feasible Cache Indicators): Since the cache size is limited to m bits, in general not all vectors in $\{0,1\}^K$ correspond to a feasible cache configuration. Therefore, we define the set of *feasible cache indicators* as follows:

$$\mathcal{F} = \left\{ \sigma \in \{0,1\}^K : \sum_{k=1}^{K} \sigma_k W'_k \leq m \right\}.$$

Task offloading starts with a request from the UE to the SSC specifying the task to run and a time delay within which the UE needs to retrieve its result. Such an offloading request is denoted $r = (k, t)$, meaning that the UE asks for the execution of the k-th task and to receive its result within t seconds. If the SSC has enough available resources to elaborate the task, the request is accepted, otherwise it is denied.

Our goal is to describe strategies to reduce the costs of tasks offloading. The total cost of the offloading procedure is made of several independent contributions, among which we identify two main components: i) the cost of uploading the computation inputs from the UE to the SSC; ii) the cost of running the computation at the SSC. Depending on the application, the word "cost" can indicate energy consumptions, time delays, or any other metric that measures an expense or the quality of service. Nonetheless, in all scenarios, there are evident benefits in keeping available in the cache memory a computation result before it is requested to the SSC: indeed, whenever a result W'_k is stored, the task c_k does not need to be run, its input data do not need to be uploaded, and W'_k can be straightforwardly sent to the UE. The most important consequence is that the two cost components mentioned above are zeroed. Hence, the total cost of offloading a cacheable task $c_k \in \mathcal{C}$ is:

$$\begin{aligned}\Gamma_{\mathrm{tot}}(c_k) = \Gamma_{\mathrm{req}}(c_k) + (1-\sigma_k)\left(\Gamma_{\mathrm{UL}}(c_k) + \Gamma_{\mathrm{comp}}(c_k)\right) \\ + \Gamma_{\mathrm{DL}}(c_k) + \gamma(c_k),\end{aligned} \tag{1}$$

where $\Gamma_{\mathrm{req}}(c_k)$ is the cost of sending $r = (k, t)$, the offloading request; σ_k is the k-th entry of the cache indicator; $\Gamma_{\mathrm{UL}}(c_k)$ is the cost of uploading the input data; $\Gamma_{\mathrm{comp}}(c_k)$ is the cost of computing the task result (assuming, for simplicity, that the CPU state does not vary in time and the computation cost only depends on the task parameters); $\Gamma_{\mathrm{DL}}(c_k)$ is the cost of sending the result back to the UE; and $\gamma(c)$ includes any other fixed cost that does not directly depend on c, e.g., any fixed processing cost at the MEC level or the cost of maintaining active the SSC's hardware, including the cache memory. The cost of reading a task result from the cache is considered negligible.

III. Policies for Computation Caching

The previous considerations lead to the main question of this work: given a cache of finite size, how to choose which W'_k's to store, with the goal of minimizing the overall costs? To answer, let us consider R offloading requests $r_1, \ldots, r_R$, sent from the UE to the SSC during its service period. By definition, every request uniquely corresponds to a task: for every $i = 1, \ldots, R$, we have $r_i = (k, t_i)$, for some $k \in \{1, \ldots, K\}$ identifying a task in $\mathcal{C}$ and some latency constraint t_i. Thus, $\Gamma_{\mathrm{tot}}(r_i) = \Gamma_{\mathrm{tot}}(c_k)$ for some k and we define the *cost over the whole serving period* as:

$$\Gamma(\sigma) = \sum_{i=1}^{R} \Gamma_{\mathrm{tot}}(r_i). \tag{2}$$

Our goal is to find the cache indicator that minimizes $\Gamma(\sigma)$:

$$\sigma_{\mathrm{opt}} = \arg\min_{\sigma \in \mathcal{F}} \Gamma(\sigma) = \arg\min_{\sigma \in \mathcal{F}} \left(\sum_{i=1}^{R} \Gamma_{\mathrm{tot}}(r_i) \right). \tag{3}$$

Ideally, if σ_{opt} is known, the SSC guarantees an optimal cost minimization by storing the W'_k's for which $(\sigma_{\mathrm{opt}})_k = 1$.

Since the number of cache indicators grows exponentially with K, it is not always algorithmically possible to run through all of them to exhaustively determine σ_{opt}. The scope of this paper is to propose and evaluate strategies to choose cache indicators with close-to-optimal associated performance. A very natural choice is to assign a hierarchy among tasks and to fill the cache with the results of the highest-priority ones.

Definition 3 (Caching Metrics and Policies): A *caching metric* $\lambda : \mathcal{C} \to \mathbb{R}^+$ assigns to each task a "cacheability value". The *caching policy* based on λ is the application of the following cache filling algorithm, that prioritizes the tasks with the highest cacheability value:

1: Let $\pi : \{1, \ldots, K\} \to \{1, \ldots, K\}$ be a permutation such that $\lambda(c_{\pi(1)}) \geq \ldots \geq \lambda(c_{\pi(K)})$.
2: Set $\sigma \leftarrow (0, 0, \ldots, 0)$ and $s \leftarrow 0$.
3: **for** $k = 1, \ldots, K$ **do**
4: **if** $s + |W'_{\pi(k)}| \leq m$, **then**
5: set $\sigma_{\pi(k)} \leftarrow 1$ and $s \leftarrow s + |W'_{\pi(k)}|$.
6: **end if**
7: **end for**
8: Fill the SSC's cache according to σ.

We call $\sigma(\lambda)$ the indicator yielded by the previous algorithm. Clearly, a caching policy is based on a well-designed metric if $\Gamma(\sigma(\lambda))$ is close to $\Gamma(\sigma_{\text{opt}})$. A first observation, very spontaneous and common to the context of content caching [7]–[9], [12], [20], is that a good caching policy needs to depend on the *popularity* of tasks. Indeed, to reduce costs, we want to avoid to repeatedly process frequently requested tasks. In this perspective, we define the popularity p_k of task c_k to be the probability that c_k is offloaded to the SSC. In our setting (and, in general, whenever the offloading requests are pairwise independent and if their total number R is big enough to be statistically representative), we can write:

$$p_k = |\{i : r_i = (k, t_i), \exists\ t_i\}| \cdot R^{-1}. \tag{4}$$

First policy: simply based on task popularity, we define:

$$\lambda_1(c_k) = p_k,\ \forall k \in \{1, \dots, K\}.$$

λ_1 is essentically the metric used in [6]. A better choice comes from the observation that caching the result of a very popular task with low input uploading and computation costs, can be less advantageous than caching the result of a less popular task with higher costs. The latter directly depend on the size (in bits) of W_k, denoted $|W_k|$, which justifies the next policy.

Second policy: based on popularity and input data size, let

$$\lambda_2(c_k) = p_k|W_k|,\ \forall k \in \{1, \dots, K\}.$$

In some scenarios, the previous policy is optimal:

Lemma 1: Let us suppose that, for every $k \in \{1, \dots, K\}$,

$$\Gamma_{\text{UL}}(c_k) = a|W_k| \quad \text{and} \quad \Gamma_{\text{comp}}(c_k) = b|W_k|, \tag{5}$$

for some constants $a, b > 0$. Let us also suppose that the computation results have all a standard output size, i.e., $|W'_k| = w$, for every $k \in \{1, \dots, K\}$. Then, $\sigma(\lambda_2) = \sigma_{\text{opt}}$.

Proof: Let us call $\rho_k = |\{i : r_i = (k, t_i), \exists\ t_i\}|$. Hence, by (2) and (4), we have

$$\Gamma(\sigma) = \sum_{i=1}^{R} \Gamma_{\text{tot}}(r_i) = \sum_{k=1}^{K} \rho_k \Gamma_{\text{tot}}(c_k) = R \sum_{k=1}^{K} p_k \Gamma(c_k).$$

Now, all the addends in (1) that do not depend on σ_k do not influence the minimization that defines σ_{opt} in (3). Therefore,

$$\sigma_{\text{opt}} = \arg\min_{\sigma \in \mathcal{F}} \left(-R \sum_{k=1}^{K} p_k \sigma_k (\Gamma_{\text{UL}}(c_k) + \Gamma_{\text{comp}}(c_k)) \right).$$

Using (5), we obtain:

$$\begin{aligned} \sigma_{\text{opt}} &= \arg\min_{\sigma \in \mathcal{F}} \left(-R(a+b) \sum_{k=1}^{K} p_k \sigma_k |W_k| \right) \\ &= \arg\max_{\sigma \in \mathcal{F}} \left(\sum_{k=1}^{K} p_k \sigma_k |W_k| \right). \end{aligned} \tag{6}$$

When $|W'_k| = w$ is constant, the cache can store at most $\lfloor m/w \rfloor$ computation results, independently from k. So,

$$\mathcal{F} = \left\{ \sigma \in \{0,1\}^K : \sum_{k=1}^{K} \sigma_k \leq \left\lfloor \frac{m}{w} \right\rfloor \right\}.$$

TABLE I
PARAMETERS FOR NUMERICAL SIMULATIONS

Parameter	Value	Parameter	Value
K in Fig. 1	25	K in Fig. 2, 3, 4	50000
R_{UL}	125 Mbit/s	$\lvert W_k \rvert$ in Fig. 1	[1e6 : 1e9[bits
R_{DL}	500 Mbit/s	$\lvert W'_k \rvert$ in Fig. 1	[1e6 : 1e7[bits
α	0.6	$\lvert W_k \rvert$ in Fig. 2, 3, 4	[1e6 : 1e9[bits
e_k/f	10^{-8} s/bit	$\lvert W'_k \rvert$ in Fig. 2, 3, 4	[1e3 : 1e9[bits

Consequently, the summation in (6) is maximized when $\sigma_k = 1$ for the $\lfloor m/w \rfloor$ tasks with the highest $p_k|W_k|$. This is equivalent to fill the cache based on the second policy. Therefore, $\sigma(\lambda_2) = \sigma_{\text{opt}}$. ■

In this context, λ_2 corresponds to the metric used in [18], where it was already highlighted the need for a policy that mixes popularity and input data size.

Third policy: the hypothesis that computation outputs have standard constant size cannot be realistic for all applications. The third policy is based on the observation that caching task results whose size is small allows to store more of them. Hence, it may be more convenient to cache a high number of small-size results, even if their popularity and input size do not maximize λ_2. To increase the caching priority of tasks with small $|W'_k|$, we define:

$$\lambda_3(c_k) = p_k|W_k||W'_k|^{-1},\ \forall k \in \{1, \dots, K\}.$$

The introduction of λ_3 is one of the novelties of this work. We will show in the next section that it is the most advantageous metric of the three, from several points of view.

IV. NUMERICAL RESULTS

In our numerical simulations, $|W_k|$ and $|W'_k|$ are chosen independently at random for every k as follows: let $y, Y \in \mathbb{N}$ satisfy $y \leq Y$ and let $x, X \in \mathbb{R}$ be two real numbers in $[1, 10]$ (with $x \leq X$ if $y = Y$). When we say that $|W_k|$ belongs to $[xey : XeY[$, we mean that $x \cdot 10^y \leq |W_k| < X \cdot 10^Y$ and $|W_k| = u \cdot 10^v$, with u and v randomly fixed as follows: first, v is chosen uniformly in $\{y, y+1, \dots, Y\}$; then, u is chosen uniformly either in $[x, 10[$ (if $v = y$) or in $[1, 10[$ (if $y < v < Y$) or in $[1, X[$ (if $v = Y$). The same rule is used for $|W'_k|$, independently from the corresponding $|W_k|$. With this strategy, there is no privileged order of magnitude among the values taken by $|W_k|$ and $|W'_k|$, even when the maximum possible value is much bigger than the minimum.

In all figures, the abscissae represent the SSC's cache size. 0 % means that $m = 0$ and 100 % that $m = \sum_{k=1}^{K} |W'_k|$. Fig. 1, 2, 3, and 4 were obtained with the simulation parameters specified in Table I. In particular, we considered stable radio channel conditions and constant uplink and downlink communication rates. First, the cache was filled applying one of the policies defined in Section III, then a high number of offloading requests were simulated. We supposed the popularity of offloading requests to obey the Zipf law [4], [5]:

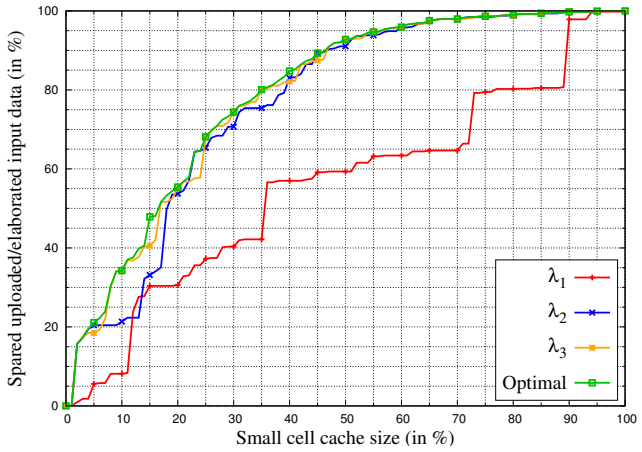

Fig. 1. Spared input data for $K = 25$.

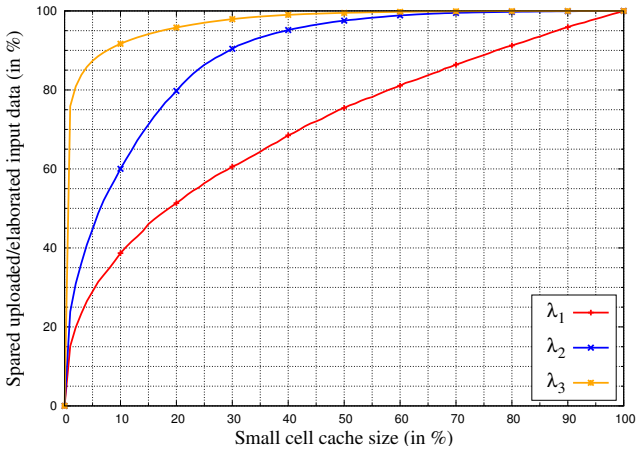

Fig. 2. Spared input data for $K = 50000$.

$p_k = (Ak^{\alpha})^{-1}$, for constant α and $A = \sum_{k=1}^{K} k^{-\alpha}$. Notice that, without loss of generality, tasks can be assumed to be sorted in the catalogue by descending popularity.

The simulated offloading operation consisted of four main serial steps: offloading request, input data uploading, task computation, results downloading. If the results of the computation were found in the SSC's cache, data uploading and task computation were skipped and the results directly sent to the UE. In all simulations, we assumed that a new offloading request was sent instantaneously after the results of the previous one were downloaded.

Fig. 1 shows, as a function of the cache size, the percentage of task input data that did not need to be uploaded nor elaborated because the corresponding results were cached and available for downloading. For brevity, we call this the "spared input data". Measuring the spared input data is an effective approach to evaluate the goodness of the caching policies: the more it is, the higher the corresponding saving in energy, time, or any other metric, both for the UE and for the SSC. Differently from the other figures, in Fig. 1 we could compare the performance of the policies also with respect to the optimal cache configuration found by exhaustively looking for σ_{opt} among all feasible cache indicators. This was practicable because we kept the number of tasks in the catalogue small enough ($K = 25$). Fig. 1 suggests two main considerations: first, that the second and third policy clearly outperform the policy exclusively based on popularity; second, that the performance of the third policy is always very close or superimposed to the optimal and beats the policy based on λ_2, especially for cache sizes between 0 and 20 %. These are the sizes which interest us the most, because an effective caching strategy needs to achieve a good performance with a cache as small as possible. The curves in Fig. 1 are not extremely smooth. The "jumps" reflect the fact that a small increase in the cache size may suddenly allow to fit in the cache task results with good metric, but "relatively too big" to be cached before. This behaviour is more visible when K is small.

Fig. 2 shows the spared input data for $K = 50000$. In this case, the optimal performance could not be traced, but

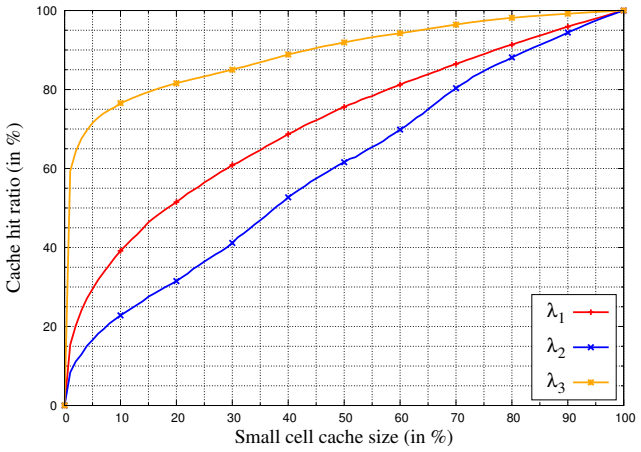

Fig. 3. Cache hit ratio for $K = 50000$.

the quality of the third policy is clearly confirmed by the separation among the curves. Remarkably, for a cache only as big as 2 % of the total size, the third policy allows to spare more than 80 % of the input data, whereas the first and second policy respectively achieve around 20 and 30 %.

Fig. 3 compares the policies in terms of cache hit ratio: by definition, this is the number of times that the results of the offloading requests were found in the cache, divided by the total number of requests. The figure suggests two observations: first, that weighing the task popularity by the cache input size to define λ_2 causes a loss in the cache hit ratio performance; this is quite natural, because λ_1 is by design a metric aimed at maximizing the cache hit ratio. Nonetheless and more importantly, this loss is completely recovered and even outdone by the third policy, which promotes for being cached the tasks with small-size results. We deduce that storing in this way more results, even if they do not correspond to the most popular tasks in absolute, yields a considerable gain: the other two policies are beaten and cache hit ratios of almost 60 % are obtained with the third policy when the cache size covers only 1 % of the total cacheable data (a performance from 4 to 7 times better with respect to the other two policies).

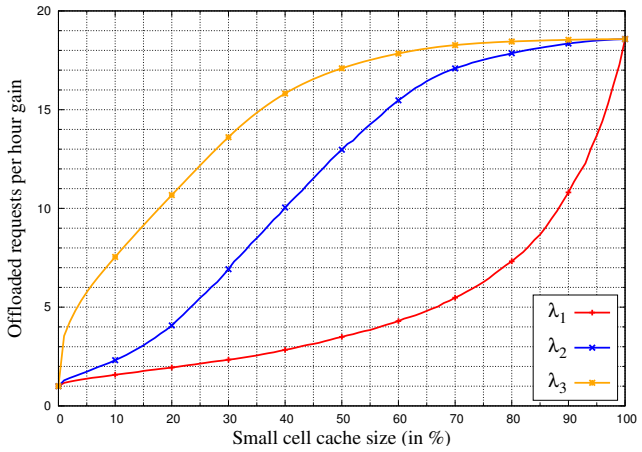

Fig. 4. Gain in served offloading requests per hour for $K = 50000$.

Fig. 4 shows the ratio between the average number of offloaded tasks per hour with and without computation caching. Measuring this gain involves the computation of the offloading time for every request. In the notation of Section II, where in this case $\Gamma_{\text{tot}}(c_k)$ indicates the total offloading time of $c_k = (W_k, e_k, W'_k)$, we have $\Gamma_{\text{req}}(c_k) = 128/R_{\text{UL}}$ (where we supposed that a request $r_i = (k, t_i)$ has a standard size of 16 bytes), $\Gamma_{\text{UL}}(c_k) = W_k/R_{\text{UL}}, \Gamma_{\text{comp}}(c_k) = e_k W_k/f$, and $\Gamma_{\text{DL}}(c_k) = W'_k/R_{\text{DL}}$. We also added to the previous terms a latency of 2 ms, corresponding to $\gamma(c_k)$. Fig. 4 reasserts the superiority of the third policy, which allows gains of up to a factor 10 for cache sizes of less than 20 %, whereas the gain with respect to the other policies does not go beyond a factor 4. These gains translate into reduced uplink transmissions and facilitate the prevention of uplink bottlenecks.

V. Conclusion

In this paper, we focused on the study, evaluation, and benchmarking of policies for proactive computation caching. We proposed a new caching metric and proved the importance of considering the size of offloadable task results for designing well-performing policies. We measured the effectiveness of our proposal via three different performance metrics: the "spared input data" percentage, the cache hit ratio, and the gain in the number of satisfied requests per hour. Our policy showed appreciable results for small cache sizes in all simulations, not only compared to the scenario in which computation caching is not performed, but also with respect to the other policies.

Future work will focus on the extension of our results to more complex scenarios, involving more UEs and more small cells. Building on the results of this paper, we will amplify the benefits of computation caching by combining it with the clustering gain obtained through federation of small cells with embedded intelligence. These techniques need a specific redesign of policies for cache filling and update.

Acknowledgment

The research leading to these results is jointly funded by the European Commission (EC) H2020 and the Ministry of Internal affairs and Communications (MIC) in Japan under grant agreements No 723171 5G MiEdge.

References

[1] A. Abouaomar, A. Filali, and A. Kobbane, "Caching, device-to-device and fog computing in 5th cellular networks generation: Survey," in *Proc. WINCOM*, Rabat, Morocco, 2017, pp. 1-6.

[2] S. Barbarossa, S. Sardellitti, and P. Di Lorenzo, "Communicating while computing: Distributed mobile cloud computing over 5G heterogeneous networks," *IEEE Signal Process. Mag.*, vol. 31, no. 6, pp. 45-55, Nov. 2014.

[3] S. Barbarossa, E. Ceci, M. Merluzzi, and E. Calvanese Strinati, "Enabling effective mobile edge computing using millimeter wave links," in *Proc. IEEE ICC Workshops*, Paris, France, 2017, pp. 367-372.

[4] E. Baştuğ, M. Bennis, and M. Debbah, "Living on the edge: The role of proactive caching in 5G wireless networks," *IEEE Commun. Mag.*, vol. 52, no. 8, pp. 82-89, Aug. 2014.

[5] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker, "Web caching and Zipf-like distributions: Evidence and implications," in *Proc. IEEE INFOCOM*, New York, USA, 1999, vol. 1, pp. 126-134.

[6] M. S. Elbamby, M. Bennis, and W. Saad, "Proactive edge computing in latency-constrained fog networks," in *Proc. EuCNC*, Oulu, Finland, 2017, pp. 1-6.

[7] A. Ioannou and S. Weber, "A survey of caching policies and forwarding mechanisms in information-centric networking," *IEEE Commun. Surveys Tut.*, vol. 18, no. 4, pp. 2847-2886, Fourthquarter 2016.

[8] Y. Jiang, M. Ma, M. Bennis, F. Zheng, and X. You, "A novel caching policy with content popularity prediction and user preference learning in fog-RAN," in *Proc. IEEE GC Workshops*, Singapore, 2017, pp. 1-6.

[9] S. Li, J. Xu, M. van der Schaar, and W. Li, "Popularity-driven content caching," in *Proc. IEEE INFOCOM*, San Francisco, USA, 2016, pp. 1-9.

[10] H. Liu, F. Eldarrat, H. Alqahtani, A. Reznik, X. de Foy, and Y. Zhang, "Mobile edge cloud system: Architectures, challenges, and approaches," *IEEE Syst. J.*, pp. 1-14, Feb. 2017.

[11] Y. Mao, C. You, J. Zhang, K. Huang, and K. B. Letaief, "A survey on mobile edge computing: The communication perspective," *IEEE Commun. Surveys Tut.*, vol. 19, no. 4, pp. 2322-2358, Fourthquarter 2017.

[12] N. M. Markovich and U. R. Krieger, "A caching policy driven by clusters of high popularity," in *Proc. IWCMC*, Paphos, Cyprus, 2016, pp. 363-368.

[13] J. Oueis, E. Calvanese Strinati, and S. Barbarossa, "The fog balancing: load distribution for small cell cloud computing," in *Proc. IEEE VTC Spring*, Glasgow, UK, 2015, pp. 1-6.

[14] J. Oueis, E. Calvanese Strinati, S. Sardellitti, and S. Barbarossa, "Small cell clustering for efficient distributed fog computing: A multi-user case," in *Proc. IEEE VTC Fall*, Boston, USA, 2015, pp. 1-5.

[15] J. Oueis and E. Calvanese Strinati, "Procédé et dispositifs associés de formation d'un nuage informatique stockant le résultat de l'exécution déportée d'une tâche informatique," French Patent FR1562546A, Dec. 16, 2015.

[16] J. Oueis, E. Calvanese Strinati, and S. Barbarossa, "Distributed mobile cloud computing: A multi-user clustering solution," in *Proc. IEEE ICC*, Kuala Lumpur, Malaysia, 2016, pp. 1-6.

[17] J. Oueis, "Joint communication and computation resources allocation for cloud-empowered future wireless networks," Ph.D. dissertation, Univ. Grenoble Alpes, Grenoble, France, 2016.

[18] J. Oueis and E. Calvanese Strinati, "Computation caching for local cloud computing," presented at WCNC, 5G TACNET Workshop, San Francisco, USA, 2017.

[19] S. Sardellitti, G. Scutari, and S. Barbarossa, "Joint optimization of radio and computational resources for multicell mobile-edge computing," *IEEE Trans. Signal Inf. Process. Netw.*, vol. 1, no. 2, pp. 89-103, July 2015.

[20] K. Suksomboon *et al.*, "PopCache: Cache more or less based on content popularity for information-centric networking," in *Proc. IEEE LCN*, Sidney, Australia, 2013, pp. 236-243.

[21] S. Wang, X. Zhang, Y. Zhang, L. Wang, J. Yang, and W. Wang "A survey on mobile edge networks: Convergence of computing, caching and communications," *IEEE Access*, vol. 5, pp. 6757-6779, 2017.