Joint Optimization of Service Caching Placement and Computation Offloading in Mobile Edge Computing System

Suzhi Bi, Liang Huang, and Ying-Jun Angela Zhang

Abstract

In mobile edge computing (MEC) systems, edge service caching refers to pre-storing the necessary programs for executing certain computation tasks at MEC servers, which is effective to reduce the real-time delay/bandwidth cost on acquiring and installing the programs. Due to the limited caching space at resource-constrained edge servers, it calls for careful design of caching placement to determine which programs to cache over time. This is in general a complicated problem that highly correlates to the computation offloading decisions of computation tasks, i.e., whether or not to offload a task for edge execution. In this paper, we consider an edge server assisting a mobile user (MU) in executing a sequence of computation tasks. In particular, the MU can upload and run its customized programs at the edge server, while the server can selectively cache the previously generated programs for future service reuse. To minimize the computation delay and energy consumption of the MU, we formulate a mixed integer non-linear programming (MINLP) that jointly optimizes the service caching placement, computation offloading decisions, and system resource allocation (e.g., CPU processing frequency and transmit power of MU). To tackle the problem, we first derive the closed-form expressions of the optimal resource allocation solutions, and subsequently transform the MINLP into an equivalent pure 0-1 integer linear programming (ILP) that optimizes only the binary caching placement and offloading decisions. To further reduce the complexity in solving a large-size ILP, we exploit the underlying graphical structures in caching causality and task dependency models, and accordingly devise a reduced-complexity alternating minimization technique to iteratively update either the caching placement or offloading decision by fixing the other. Extensive simulations show that the proposed joint optimization techniques achieve substantial resource savings of the MU compared to other representative benchmark methods considered.

Index Terms

Mobile edge computing, service caching, computation offloading, resource allocation.

S. Bi is with the College of Electronic and Information Engineering, Shenzhen University, China (bsz@szu.edu.cn). L. Huang is with the College of Information Engineering, Zhejiang University of Technology, China (lianghuang@zjut.edu.cn). Y-J. A. Zhang is with the Department of Information Engineering, The Chinese University of Hong Kong, HK (yjzhang@ie.cuhk.edu.hk).

I. Introduction

A. Motivations and Summary of Contributions

The proliferation of modern wireless applications, such as mobile gaming and augmented reality, demands persistent high-performance computations at average commercial wireless devices to execute complex tasks with ultra-low latency. Over the last decade, large-scale *cloud computing* platforms have been extensively deployed, which allow the wireless devices to offload intensive computations to remote cloud servers with abundant computing resource [1]. To reduce the long backhaul transmission delay in the cloud, *mobile edge computing* (MEC) has recently emerged to support ubiquitous high-performance computing, especially for delay-sensitive applications [2]. Specifically, MEC pushes publicly accessible computing resource to the edge of radio access network, e.g., cellular base stations and WiFi access points, such that mobile users (MUs) can quickly offload computing tasks to their nearby edge servers.

Computing a task requires both the user task data as the input and the corresponding program that processes it. The use of *caching* to dynamically store the program and/or task data at the MEC system has been recently recognized as a cost-effective method to reduce computation delay, energy consumption, and bandwidth cost [3], [10]. Here, we refer to the techniques to cache the input and/or output of computation tasks at the server/user side as *computation content caching* (such as in [3]–[7]). On one hand, content caching reduces the data exchange between the edge servers and MUs if the required input data can be found in the cache. On the other hand, if the desired computation output is already cached from previous execution of an identical task, the entire computation process can be saved. Compared to resource-abundant cloud servers, edge servers are often limited in the caching space. Therefore, a major design issue in MEC system is to selectively cache the task contents over space (e.g., at multiple edge servers) and time for achieving optimum computing performance, e.g., minimum computation delay.

Notice that the effectiveness of computation content caching relies on a strong assumption that the cached input/output of a computation task are frequently reused by future executions. In practice, however, although an application may be repeatedly executed, the input data and the corresponding computation output are rather dissimilar and hardly reusable for separate executions, e.g., human face recognition and interactive online gaming. In comparison, program data (and/or library data) in the cache is evidently reusable by future executions of the same application, e.g., the program and library for human face recognition. Its deployment effectively

reduces the delay caused by real-time program dowload/installation[1] or remote computation migration due to the absence of necessary program [9]. To distinguish from content caching, we refer to the techniques to cache the program data as *computation service caching*.

A common service caching model in MEC system is that an MU offloads its computation task to an edge server if the server has cached the required program. Otherwise, if the required program is not available at any accessible edge server, the MU resorts to a remote cloud server that can always compute the task but at the cost of longer backhaul delay and larger bandwidth usage (e.g., see [10]–[15]). Several works have studied the optimal offline and online service caching placement problems (i.e., what, when and where to cache) to minimize the computation workload forwarded to the cloud. Nonetheless, offloading all computation tasks for edge/cloud execution can be costly, if not impossible, when the channel is under deep shadowing or the required program is currently unavailable at the destined server. Alternatively, computing some tasks locally at the MUs could be better off. Notice that the task offloading decisions (i.e., whether offloading a task or computing locally) are closely related to the service caching placement. On one hand, we tend to offload a task if the required program is in the edge cache. Meanwhile, caching a program is cost-saving only if the related tasks are frequently offloaded for edge execution. Therefore, it necessitates a joint optimization of both service caching placement and offloading decisions in an MEC system, which, however, is currently lacking of concrete studies.

Meanwhile, most of existing works implicitly assume that a central entity, e.g., the owner of the edge/cloud servers, is responsible for provisioning the program data in the cache, and all the required computation programs can be retrieved from a program pool in the backhaul network (e.g., in [10]–[15]). However, as the mobile computing scenarios become increasingly heterogeneous, it is common to allow the MU themselves to run custom-made or user-generated programs at the edge/cloud platforms. In fact, this is consistent with the concept of virtualization and Infrastructure-as-a-service (IaaS) in edge/cloud computing paradigms, where the infrastructure owner only provides the physical resources of computing, storage, and networking to meet individual computation demands through resource slicing, while the MUs are entitled to execute their own programs [16]. For instance, an MU can upload its own program code (e.g., C code in less than several Megabytes) to the edge server, which then runs the code to generate executable files (e.g., .EXE file in tens of Megabytes) for processing the task data later offloaded. Noticeably,

[1]The generation and loading time of a program can take several even tens of seconds as compared to millisecond-level task computation time for some common IoT applications [8].

in this case, the overhead on uploading and installing the program could have significant impact to the service caching placement and offloading decisions.

In this paper, we consider an MEC system, where an edge server assists an MU in executing a sequence M *dependent* tasks, i.e., the output of one task is the input of the next one. Each task belongs to one of the N applications and is either computed locally or offloaded for edge execution. In particular, the MU provides the program data for computing the tasks in the edge, while the edge server can selectively cache the previously generated programs and reuse them for processing future tasks. The detailed contributions of this paper are as follows.

- We formulate a mixed integer non-linear programming (MINLP) problem to minimize the overall computation delay and energy consumption of the MU. Specifically, the problem jointly determines the optimal offloading decision of each task (M binary variables), the service caching placement at the edge server throughout the task execution time (MN binary variables), and system resource allocation (continuous variables representing the CPU processing frequency and transmit power of MU). The MINLP problem is in general lacking of efficient optimal algorithm in its original form.
- To tackle the problem, we first show that the system resource allocation can be separately optimized and derive the closed-form expressions of the optimal solutions. Based on the results, we then introduce auxiliary variables to transform the MINLP into a pure 0-1 integer linear programming (ILP) problem that optimizes only the binary offloading decisions and service caching placements. The ILP problem can be handled by some standard integer optimization algorithms, e.g., branch and bound method [18]. However, the exponential worst-case complexity can be high when either M or N is large.
- To gain more insight on the optimal solution structure and reduce the complexity of solving a large-size ILP problem, we first study the problem to optimize the MN caching placement variables given the offloading decisions. By exploiting the structure of caching causality condition, we transform the original problem into a standard multidimensional knapsack problem (MKP), which has no more than M binary variables and can be efficiently handled by some off-the-shelf algorithms even if M is relatively large, e.g., $M = 600$ [19].
- We then consider optimizing the M offloading decisions given the caching placement. Interestingly, we find that the only difficulty lies in optimizing the offloading decisions of those "uncached" tasks, whose required programs are not in the cache, while the optimal offloading decisions of the other cached tasks can be easily retrieved. Together with the result

on caching placement optimization, this property leads to a reduced-complexity alternating minimization that iteratively updates the caching placements and offloading decisions.

Our simulations show that the joint optimization significantly reduces the computation delay and energy consumption of the MU compared to other benchmark methods. Meanwhile, the sub-optimal alternating minimization is suitable for real-time implementation. It is worth mentioning that this paper considers an offline model that assumes non-causal knowledge of future computation task parameters. The assumption is made to characterize the optimal structures of caching placement and offloading decisions. The obtained results can serve as an offline benchmark and may inspire future online algorithm designs that assume more practical prior knowledge.

B. Related works

The computing performance of an MEC system often requires joint optimization of the task offloading decision (i.e., whether or how much data to offload) and system-level resource allocation (e.g., spectrum and computing power) [20]–[25]. Depending on the nature of computation tasks, computation offloading is performed either following a *partial offloading* policy [20], i.e., an arbitrary part of the task data can be offloaded for edge execution, or a *binary offloading* policy that an entire task is either offloaded or computed locally [21]. For instance, [24]–[26] optimize the computing performance of MEC systems powered by wireless energy transfer following either the partial or binary offloading policy. When the computing tasks at different MUs have input-output dependency, [23] studies the optimal binary offloading strategy and resource allocation that minimizes the computation delay and energy consumption. In this paper, we consider a sequence of dependant tasks that follow the binary offloading policy.

Integrating content caching into MEC system design can effectively reduce computation delay, energy consumption, and bandwidth cost. In particular, an edge server can cache task output data [3], task input data [4], and intermediate task computation results that are potentially useful for future task executions [5]. Meanwhile, content caching can also be implemented at the MU side to minimize the offloading (downloading) traffic to (from) the edge server [6]. To address the uncertainty of future task parameters, [5] and [7] propose online caching placement and prediction-based data prefetch methods. Despite their respective contributions, the fundamental assumption on reusing task input/output data may not hold for many mobile applications.

Computation service caching, on the other hand, considers caching the program data for processing a specific type of application. For instance, [10] considers caching program data of

multiple applications in a set of collaborative BSs, and optimizing the caching placement and user-BS associations to minimize the data traffic forwarded to the remote cloud. A similar service caching placement problem is considered in [11] under communication, computation, and caching capacity constraints. Under the uncertainty of user service requests, e.g., application type and computation workload, [12] proposes a prediction-based online edge service caching algorithm to reduce the traffic load forwarded to the cloud. For a single edge server, [13] assumes zero knowledge of future task arrivals and proposes an online service caching algorithm that achieves the best worst-case competitive ratio under homogeneous task arrivals. [14] also proposes online caching algorithm for collaborative edge servers to minimize the overall computation delay. Unlike [10]–[14] that assume an entire task is computed either at an edge server or the cloud, [15] considers that a task can be partitioned and executed in parallel at both the cloud and edge servers that have cached the necessary program, and designs an online service caching method.

All the above works neglect an important scenario that a task may be computed locally at the MU when edge execution is costly. Besides, they implicitly assume that a service program pool can provide all the programs required by the MUs. In this paper, we include local computation as an option for the MU, and allow the MU to upload its own programs to run at the edge server. In this case, the optimal caching placement is closely related to the offloading decisions, and vice versa, such that a joint optimization is required for maximum computation performance.

II. System Model

In Fig. 1, we consider an MU that has a sequence of M computation tasks to execute, where each task is processed by one of the N programs considered. We refer to a task as a type-j task if it is processed by the j-th program. Accordingly, we use a binary indicator $u_{i,j} = 1$ to denote that the i-th task is a type-j task, and 0 otherwise. The M tasks are dependent such that the input of the $(i+1)$-th task requires the output of the i-th task, $i = 1, \cdots, M-1$. The size of the input and output data of the i-th task is denoted by I_i and O_i, respectively. Besides, L_i denotes the computing workload to process task i. For simplicity of illustration, we introduce two pseudo tasks indexed as 0 and $M+1$, and set $L_0 = L_{M+1} = 0$, $O_0 = I_1$ and $O_M = I_{M+1}$. Overall, the input and output task data sizes are related by $I_i = O_{i-1},\ i = 1, \cdots, M+1$. The MU follows the binary offloading policy so that each task can be computed either locally at the MU or offloaded to the edge server for remote execution. We use $a_i \in \{0, 1\}$ to denote that the

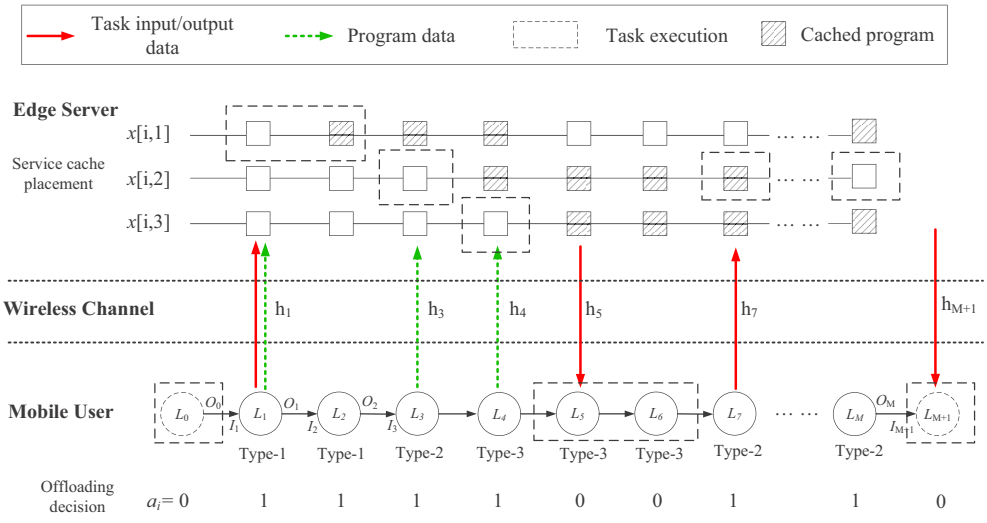

Fig. 1: Schematics of the considered service cache-assisted MEC system.

i-th task is executed locally ($a_i = 0$) or at the edge server ($a_i = 1$). In particular, we set $a_0 = 0$ and $a_{M+1} = 0$, indicating that the series of computations initiate and terminate both at the MU.

Suppose that the MU runs its customized programs at the MEC platform by uploading its own program data (e.g., C/C++ code to generate a program). We denote the size of data to generate the j-th program as s_j, $j = 1, \cdots, N$. After receiving the program data, the edge server generates the corresponding program (e.g., the binary executable .EXE file) for processing the task data later offloaded. We denote the size of the j-th generated program as c_j, where c_j is in general much larger than s_j. Meanwhile, the edge server has a service cache that can cache the previously generated programs for future service reuse. We denote $x\,[i, j] = 1$ (or 0) if the j-th program is in the edge service cache (or not) before the execution of the i-th task, either locally or at the edge, where $i = 1, \cdots, M$. The edge server can decide to add (or remove) a program to (from) the cache during each task execution time, if the action is feasible under a finite caching space. For simplicity of illustration, we neglect the cost of adding or removing a program at the service cache and assume that the cache is empty initially, i.e., $x[1, j] = 0$ for all j.

Notice that the program data and task data can be offloaded separately. As an illustrative example in Fig. 1, at the server side, a shaded (an empty) square in the j-th row and i-th column denotes $x[i, j] = 1$ ($x[i, j] = 0$). Besides, the dashed square denotes the location of each task execution. For the first task that is executed at the edge, we need to upload both program and task data, as they are both absent at the server before the execution. However, we only need to offload the program data of the 3-rd and 4-th tasks, because their task input data is already

at the edge as the output of previous edge computations. In addition, the 7-th task only uploads the task data, because the corresponding program is already in the edge service cache.

The detailed computation, caching, and communication models are described as follows:

1) Computation Model: We assume that the MU has all the programs needed to process its tasks, e.g., pre-installed in the on-chip disk, such that the time consumed on processing a task i locally only consists of the computation time.[2] Specifically, the time and energy consumed on computing the i-th task locally are [27]

$$\tau_i^l = \frac{L_i}{f_i}, \quad e_i^l = \kappa f_i^3 \tau_i^l = \kappa \frac{(L_i)^3}{(\tau_i^l)^2}, \tag{1}$$

respectively, where f_i denotes the local CPU frequency and is constrained by a maximum frequency $f_i \leq f_{max}$, and $\kappa > 0$ denotes the computing energy efficiency parameter.

On the other hand, when a task i is executed on the edge, the computation time includes two parts. First, the task processing time $\tau_i^c = \frac{L_i}{f_0}$, where f_0 denotes the fixed CPU frequency of the edge server and is assumed $f_0 > f_{max}$, i.e., the server has stronger computing power than the MU. Second, the server may need to install a new program if the program is not in the cache. The program installation time of the i-th task, if necessary, is $W_i \triangleq \sum_{j=1}^{N} u_{i,j} D_j$, where D_j denotes the installation time of the j-th program.

2) Service Caching Model: We assume that the MU can only upload the j-th program data to the edge server when it is needed for task execution. That is, when $u_{i,j} = 1$, the MU can only upload the j-th program data when executing the i-th task at the edge. Accordingly, $x[i,j] = 1$ is attainable only if at least one of the following two conditions hold:

1) the j-th program was in the cache before the execution of the last task ($x[i-1,j] = 1$);
2) the j-th program data was uploaded to the edge server in the last task execution time. This requires $u_{i-1,j} = 1$ and $a_{i-1} = 1$, or equivalently $a_{i-1} u_{i-1,j} = 1$.

If neither condition is satisfied, we have $x[i,j] = 0$. Equivalently, the above *caching causality constraint* is expressed as $x[i,j] \leq a_{i-1} u_{i-1,j} + x[i-1,j]$, for $i = 1, \cdots, M,\ j = 1, \cdots, N$. We denote the caching space allocated by the MEC platform to serve the MU as C, such that the following cache capacity constraint needs to be satisfied throughout processing the M tasks,

$$\sum_{j=1}^{N} c_j \cdot x[i,j] \leq C,\ i = 1, \cdots, M. \tag{2}$$

[2]For fast service data access and removal, the edge server caches the programs in high-speed memory, e.g., SRAM or RAM. In comparison, the MU pre-installs the programs at its disk memory, which is slower but much less expensive (e.g., several Gigabytes disk memory available at the MU v.s. several hundred of Megabytes RAM allocated by the server to serve a MU.)

3) Communication Model: Data transmissions between the edge server and the MU include uploading the program and/or task data, and downloading the computation result. For simplicity, we assume uplink/downlink channel reciprocity and use h_i to denote the channel gain when transmitting the data of the i-th task. Then, the uploading data rate for the i-th task is $R_i^u = B\log_2\left(1+\frac{p_i h_i}{\sigma^2}\right)$, where B denotes the communication bandwidth, p_i denotes the transmit power, and σ^2 denotes the noise power. Suppose that the i-th task is a type-j task. Then, the time consumed on offloading the program data of the i-th task is

$$\tau_i^s = \frac{\sum_{j=1}^N u_{i,j}s_j + PH}{R_i^u} \triangleq \frac{V_i + PH}{R_i^u}, \tag{3}$$

where $V_i \triangleq \sum_{j=1}^N u_{i,j}s_j$ denotes the program data size of the i-th task. PH is the fixed packet header length, which is assumed to be 0 without loss of generality in the following analysis. Define function $f(x) = \sigma^2\left(2^{\frac{x}{B}}-1\right)$. Then, it follows from (3) that the transmit power p_i^s and the energy consumption e_i^s are

$$p_i^s = \frac{1}{h_i} f\left(\frac{V_i}{\tau_i^s}\right), \quad e_i^s = p_i^s\tau_i^s = \frac{\tau_i^s}{h_i} f\left(\frac{V_i}{\tau_i^s}\right), \tag{4}$$

respectively. Notice that the above e_i^s is convex with respect to τ_i^s. Similarly, the time, power and energy spent on offloading the task data for the i-th task are denoted as

$$\tau_i^u = \frac{O_{i-1}}{R_i^u}, \quad p_i^u = \frac{1}{h_i} f\left(\frac{O_{i-1}}{\tau_i^u}\right), \quad e_i^u = p_i^u\tau_i^u = \frac{\tau_i^u}{h_i} f\left(\frac{O_{i-1}}{\tau_i^u}\right), \tag{5}$$

respectively. When both the task data and program data are offloaded to the edge, we assume that they are jointly encoded in one packet to reduce the packet header overhead. Accordingly, the edge server only starts installing the program after receiving and decoding the whole packet. It can be easily verified that the time and energy consumed on transmitting both the program and task data of length $(V_i + O_{i-1})$ are merely the sum of the corresponding two parts in (3)-(5).

Furthermore, the time consumed on downloading the input data of the i-th task for local computation is $\tau_i^d = \frac{O_{i-1}}{R_i^d}$, where $R_i^d = B\log_2\left(1+\frac{P_0 h_i}{\sigma^2}\right)$ denotes a given downlink data rate for the i-th task when the server transmits with fixed power P_0.

III. Problem Formulation

A. Performance Metric

In this section, we formulate the joint caching placement and computation offloading optimization problem. We first introduce the key performance metric considered in this paper: *computation time and energy cost* (TEC) of the MU. Firstly, the total computation time consists

of two parts. One is the task execution time of the M tasks, which can be expressed as

$$T^{exe} = \sum_{i=1}^{M+1} \left[(1 - a_i) \tau_i^l + a_i \tau_i^c \right]. \quad (6)$$

The two terms correspond to the processing delay that a task is executed locally and at edge server, respectively. The other part, denoted as T^{pre}, is the time spent on preparing for the program and task data before task execution, i.e., data transmission and program installation. Consider a tagged task i, we discuss the preparation time for the task in the following cases.

1) Case 1 ($a_{i-1} = 0$ and $a_i = 0$): In this case, the two consecutive tasks are computed locally, which incurs no delay on either program or task data transmission.
2) Case 2 ($a_{i-1} = 0$ and $a_i = 1$): In this case, it takes τ_i^u amount of time to offload the task data to the edge. Meanwhile, program data uploading and program installation is needed if the program for computing the i-th task is not in the cache. Mathematically, the delay overhead in offloading and installing the program is
$$\tau_i^o \triangleq (W_i + \tau_i^s) \sum_{j=1}^{N} (1 - x[i,j]) u_{i,j}. \quad (7)$$
Overall, the preparation time is $\tau_i^u + \tau_i^o$.
3) Case 3 ($a_{i-1} = 1$ and $a_i = 0$): Only the computation output of the previous task needs to be downloaded to the MU. Accordingly, the consumed time is τ_i^d.
4) Case 4 ($a_{i-1} = 1$ and $a_i = 1$): The input task data of the i-th task is already available after the computation of the previous task. Thus, the preparation time is the time needed for program data transmission and installation, if the program data is not in the service cache. In other words, the time consumed is τ_i^o.

From the above analysis, we have

$$T^{pre} = \sum_{i=1}^{M+1} \left[(1 - a_{i-1}) a_i \tau_i^u + a_{i-1} (1 - a_i) \tau_i^d + a_i \tau_i^o \right], \quad (8)$$

where $a_0 = a_{M+1} = 0$ by definition. Therefore, the total computation delay of the M tasks is

$$T = T^{exe} + T^{pre} = \sum_{i=1}^{M+1} \left[(1 - a_{i-1}) a_i \tau_i^u + a_{i-1} (1 - a_i) \tau_i^d + (1 - a_i) \tau_i^l + a_i \tau_i^o + a_i \tau_i^c \right]. \quad (9)$$

Meanwhile, the energy consumption of the MU is

$$E = \sum_{i=1}^{M+1} \left[(1 - a_i) e_i^l + (1 - a_{i-1}) a_i e_i^u + a_i e_i^o \right], \quad (10)$$

where $e_i^o = e_i^s \sum_{j=1}^{N} (1 - x[i,j]) u_{i,j}$ denotes the energy consumed on uploading the program data for the i-th task. The other two terms correspond to the energy consumed on local computation and task data offloading, respectively. The performance metric TEC is the weighted sum of the two objectives, i.e., $TEC = \beta T + (1 - \beta) E$, where $\beta \in [0, 1]$ is a weighting parameter.

B. Problem Formulation

Overall, we are interested in minimizing the TEC of the MU by jointly optimizing the task offloading decision $\mathbf{a}$, the computational caching decision $\mathbf{X}$, and the system resource allocation $\{\mathbf{f}, \boldsymbol{\tau}, \mathbf{p}\}$. Here, $\mathbf{f} = \{f_i\}$, $\boldsymbol{\tau} = \left\{\tau_i^l, \tau_i^u, \tau_i^s\right\}$, $\boldsymbol{p} = \{p_i^u, p_i^s\}$. That is, we solve

$$
\begin{aligned}
(P1): \quad \underset{\mathbf{a},\mathbf{X},\mathbf{f},\boldsymbol{\tau},\mathbf{p}}{\text{minimize}} \quad & \beta T + (1-\beta)E && \text{(11a)}\\
\text{subject to} \quad & \textstyle\sum_{j=1}^{N} c_j \cdot x[i,j] \le C,\ i = 1, \cdots, M, && \text{(11b)}\\
& x[i,j] \le a_{i-1} u_{i-1,j} + x[i-1,j],\ \forall i,j, && \text{(11c)}\\
& 0 \le p_i^u, p_i^s \le P_{max},\ 0 \le f_i \le f_{max},\ \forall i, && \text{(11d)}\\
& \tau_i^l, \tau_i^u, \tau_i^s \ge 0, \forall i, \quad a_i, x[i,j] \in \{0,1\},\ \forall i,j. && \text{(11e)}
\end{aligned}
$$

Here, (11b) and (11c) correspond to the caching capacity and causality constraints, respectively. (11d) indicates the maximum transmit power and CPU frequency of the MU. From (1), there is a one-to-one mapping between τ_i^l and f_i. Besides, p_i^u is uniquely determined by τ_i^u in (5), and p_i^s is uniquely determined by τ_i^s in (4). By substituting $\{\mathbf{f}, \mathbf{p}\}$ with $\boldsymbol{\tau}$, we can equivalently express (P1) as

$$
\begin{aligned}
(P2): \quad \underset{\mathbf{a},\mathbf{X},\boldsymbol{\tau}}{\text{minimize}} \quad & \beta T + (1-\beta)E\\
\text{subject to} \quad & (11b), (11c)\\
& \tau_i^l \ge \frac{L_i}{f_{max}}, \tau_i^u \ge \frac{O_{i-1}}{R_i^{max}}, \tau_i^s \ge \frac{V_i}{R_i^{max}},\ i = 1, \cdots, M,\\
& a_i, x[i,j] \in \{0,1\},\ \forall i,j,
\end{aligned}
\tag{12}
$$

where $R_i^{max} = B \log_2\left(1 + \frac{h_i P_{max}}{\sigma^2}\right)$ is a parameter. The above problem (P2) is a mixed integer non-linear programming (MINLP), which is lacking of efficient algorithm in its current form. In the following sections, we first show that the problem can be equivalently transformed into a pure 0-1 integer linear programming (ILP), and then propose reduced-complexity algorithms by exploiting the structures of optimal caching placements and offloading decisions.

IV. Joint Optimization via Integer Linear Programming

Notice that (P2) is convex in $\boldsymbol{\tau}$ if the offloading decision $\mathbf{a}$ and caching placement $\mathbf{X}$ are given. In this section, we first derive the closed-form expressions of the optimal $\boldsymbol{\tau}^*$ given the other variables. Then, based on the obtained results, we show that (P2) can be equivalently transformed into a pure binary ILP problem, which can be handled by off-the-shelf algorithms.

A. Optimal Resource Allocation

A close observation on the objective of (P2) indicates that $\boldsymbol{\tau}$ can be optimized separately from the binary variables $\{\mathbf{a}, \mathbf{X}\}$. After some simple manipulation, (P2) can be equivalently written as the following problem:

$$\begin{aligned} (P3): \quad & \underset{\mathbf{a},\mathbf{X}}{\text{minimize}} \quad \textstyle\sum_{i=1}^{M+1} \rho_i \\ & \text{subject to} \quad (11b), (11c), \quad a_i, x[i,j] \in \{0,1\}, \ \forall i,j, \end{aligned} \tag{13}$$

where

$$\rho_i \triangleq o_i^* \left(1-a_{i-1}\right) a_i + l_i^* \left(1-a_i\right) + s_i^* a_i \textstyle\sum_{j=1}^{N} \left(1-x[i,j]\right) u_{i,j} + \beta a_{i-1} \left(1-a_i\right) \tau_i^d + \beta a_i \tau_i^c, \tag{14}$$

and $\{o_i^*, l_i^*, s_i^*\}$'s are parameters obtained by optimizing the resource allocation variables $\boldsymbol{\tau}$. Specifically, u_i^* is obtained by optimizing τ_i^u as follows,

$$o_i^* = \underset{\tau_i^u}{\text{minimize}} \qquad \beta\tau_i^u + (1-\beta)\frac{\tau_i^u}{h_i} f\left(\frac{O_{i-1}}{\tau_i^u}\right) \tag{15a}$$

$$\text{subject to} \qquad \tau_i^u \geq \frac{O_{i-1}}{R_i^{max}}, \tag{15b}$$

for $i = 1, \cdots, M+1$. Likewise, l_i^* is obtained by optimizing τ_i^l as follows,

$$l_i^* = \underset{\tau_i^l}{\text{minimize}} \qquad \beta\tau_i^l + (1-\beta)\kappa\frac{(L_i)^3}{(\tau_i^l)^2} \tag{16a}$$

$$\text{subject to} \qquad \tau_i^l \geq \frac{L_i}{f_{max}}, \tag{16b}$$

for $i = 1, \cdots, M+1$. In addition, o_i^* is obtained by optimizing τ_i^s as follows,

$$s_i^* = \underset{\tau_i^s}{\text{minimize}} \qquad \beta W_i + \beta\tau_i^s + (1-\beta)\frac{\tau_i^s}{h_i} f\left(\frac{V_i}{\tau_i^s}\right) \tag{17a}$$

$$\text{subject to} \qquad \tau_i^s \geq \frac{V_i}{R_i^{max}}, \tag{17b}$$

for $i = 1, \cdots, M+1$. In other words, the resource allocation optimization of $\boldsymbol{\tau}$ can be transformed into individual scalar optimization problems. The following Lemma 1 derives the closed-form expression of the optimal solution $(\tau_i^u)^*$ to (15).

Lemma 1: The optimal solution τ_i^u is

$$(\tau_i^u)^* = \begin{cases} \frac{O_{i-1}}{R_i^{max}}, & \text{if } h_i \leq \frac{\sigma^2}{P_{max}}\left(\frac{A}{-\mathcal{W}(-A\exp(-A))} - 1\right), \\ \frac{\ln 2 \cdot O_{i-1}}{B \cdot \left[\mathcal{W}\left(e^{-1}\left[\frac{\beta h_i}{(1-\beta)\sigma^2} - 1\right]\right) + 1\right]}, & \text{otherwise}, \end{cases} \tag{18}$$

where $A \triangleq 1 + \frac{\beta}{(1-\beta)P_{max}}$ and $\mathcal{W}(x)$ denotes the Lambert-W function, which is the inverse function of $f(z) = z\exp(z) = x$, i.e., $z = \mathcal{W}(x)$.

Proof: Please see the detailed proof in Appendix A. ■

Similar to the proof in Lemma 1, the optimal $(\tau_i^s)^*$ to (17) is

$$(\tau_i^s)^* = \begin{cases} \frac{V_i}{R_i^{max}}, & \text{if } h_i \leq \frac{\sigma^2}{P_{max}}\left(\frac{A}{-\mathcal{W}(-A\exp(-A))}-1\right), \\ \frac{\ln 2 \cdot V_i}{B\cdot\left[\mathcal{W}\left(e^{-1}\left[\frac{\beta h_i}{(1-\beta)\sigma^2}-1\right]\right)+1\right]}, & \text{otherwise.} \end{cases} \tag{19}$$

Meanwhile, the optimal solution $\left(\tau_i^l\right)^*$ to (16) can be obtained by calculating the derivative of the objective and considering the boundary condition, as follows

$$\left(\tau_i^l\right)^* = \begin{cases} \frac{L_i}{f_{max}}, & \text{if } f_{max} \leq \left(\frac{\beta}{2\kappa(1-\beta)}\right)^{\frac{1}{3}}, \\ \left(\frac{2\kappa(1-\beta)}{\beta}\right)^{\frac{1}{3}} L_i, & \text{otherwise,} \end{cases} \tag{20}$$

When the optimal $\boldsymbol{\tau}^*$ is obtained, the optimal $\{\mathbf{f}^*, \mathbf{p}^*\}$ in (P1) can be retrieved accordingly from (1), (4) and (5).

Remark 1: For an offloaded task, because $\mathcal{W}(x) > -1$ when $x > -1/e$, the denominator in the second term of (18) is always positive. Besides, as $\mathcal{W}(x)$ is an increasing function when $x > -1/e$, the optimal offloading time $(\tau_i^u)^*$ becomes larger as h_i decreases. Meanwhile, when h_i is weaker than the fixed threshold in (18), the MU should transmit at maximum power $(p_i^u)^* = P_{max}$ (and thus the maximum data rate R_i^{max}) to minimize the offloading time. Similar results can also be obtained for $(\tau_i^s)^*$ and $(p_i^s)^*$ from (19). For the local computing tasks, the optimal solution $\left(\tau_i^l\right)^*$ in (20) shows that the MU should compute faster either when a larger weight β is assigned to the delay cost or when the local computation is more energy-efficient (small κ). When β is sufficiently large or κ is sufficiently small, the task should be computed locally at a maximum speed f_{max} to minimize the computation delay.

B. Equivalent ILP Formulation

Given the fixed parameters $\{o_i^*, l_i^*, s_i^*\}$ in (P3), the problem is a quadratic integer programming problem due to the multiplicative terms. To further simplify the problem, we introduce two sets of auxiliary variables $z_i \triangleq a_i x\left[i, t_i\right]$ and $b_i \triangleq a_i a_{i-1}$ for $i = 1, \cdots, M$. Here, t_i denotes the type of the i-th task, e.g., $t_1 = 1$ and $t_3 = 2$ in Fig. 1. Accordingly, we re-express (P3) as

$$\underset{\mathbf{a},\mathbf{b},\mathbf{z},\mathbf{X}}{\text{minimize}} \quad \sum_{i=1}^{M}\left(o_i^* + \beta\tau_i^c + \beta\tau_{i+1}^d + s_i^* - l_i^*\right)a_i - \sum_{i=2}^{M}\left(o_i^* + \beta\tau_i^d\right)b_i - \sum_{i=2}^{M} s_i^* z_i + \sum_{i=1}^{M} l_i^*, \tag{21a}$$

$$\text{subject to} \quad b_i \leq \frac{1}{2}\left(a_{i-1} + a_i\right), \ i = 1, \cdots, M, \tag{21b}$$

$$z_i \leq \frac{1}{2}\left(a_i + x\left[i, t_i\right]\right), \ i = 1, \cdots, M, \tag{21c}$$

$$(11b), (11c), \ a_i, b_i, z_i, x[i,j] \in \{0,1\}, \ \forall i, j. \tag{21d}$$

Constraint (21b) forces b_i to be zero if either a_{i-1} or a_i is zero. Otherwise, if $a_{i-1} = a_i = 1$, $b_i = 1$ must hold at the optimum because because the objective is decreasing in b_i. Therefore, $b_i = a_i a_{i-1}$ holds at the optimum when constraint (21b) is satisfied. Similar argument also applies to constraint (21c). Overall, the above problem is a standard 0-1 ILP problem, which can be handled by standard exact algorithms, e.g., branch and bound method [18]. Notice that the problem has $M(N+3)$ binary variables, while the worst-case complexity of branch-and-bound method, as well as many other well-known exact algorithms for ILP, is as high as exhaustive search over all the binary variables. Therefore, the complexity of solving (P3) can still be high when either M or N is large, e.g., taking several minutes to compute when M equals several hundred. To reduce the complexity of solving a large-size ILP in real-time implementation, we investigate in the following sections an alternating minimization heuristic, where service caching placements and offloading decisions are optimized separately and iteratively.

V. Optimal Service Caching Placement

A. Structure of the Caching Causality

In this section, we assume a feasible offloading decision $\mathbf{a}$ is given in (P3) and optimize the service caching placement $\mathbf{X}$. By eliminating the unrelated terms, (P3) reduces to

$$(P4): \quad \underset{\mathbf{X}\in\{0,1\}^{M\times N}}{\text{maximize}} \quad \sum_{i\in\mathcal{A}} s_i^* x\left[i, t_i\right] \tag{22a}$$

$$\text{subject to} \quad (11b), (11c). \tag{22b}$$

where $\mathcal{A}$ denotes the index set of offloading tasks. The number of binary variables of the above ILP is MN. In the following, we exploit the graphical structure of the caching casuality constraint in (11c) to transform (P4) into an equivalent form with only $|\mathcal{A}|$ variables.

We show that it is sufficient to optimize only the caching placement for the offloading tasks, i.e., $\{x\left[i, t_i\right] | i \in \mathcal{A}\}$, while the other optimizing variables are redundant. As an illustrative example in Fig. 2, we consider 12 tasks to be executed in sequence, where 2 of the tasks (task 2 and 7) are computed locally while the rest are computed at the edge. Let us consider the tasks between two consecutive offloading tasks of the same service type, say task 6 and 10 of type-1 service. If $x[10, 1] = 1$, it must hold from the constraint of (11c) that $x[7, 1] = x[8, 1] = x[9, 1] = 1$. Intuitively, program 1 must be in the cache at the beginning of the 7-th task execution, because there is no other chance the program can be uploaded to the edge server between task 7 to 9. On the other hand, if $x[10, 1] = 0$, we can simply set $x[7, 1] = x[8, 1] = x[9, 1] = 0$ to satisfy

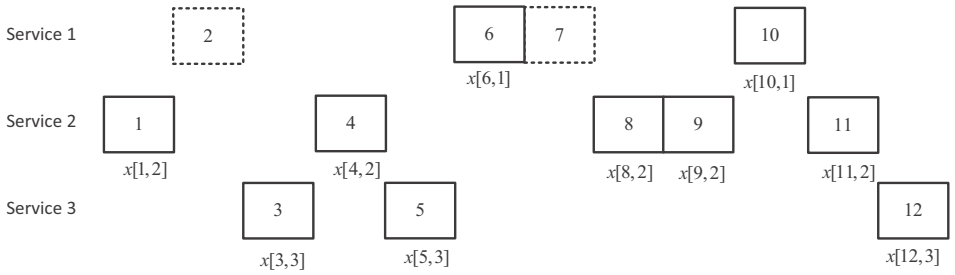

Fig. 2: An example task offloading decision. The solid (dashed) square indicates the task is computed at the edge (locally). The variables below the solid boxes are optimizing variables.

the caching casuality constraint (11c). To see this, we note that setting $x[i,1]=1$, for some $7 \leq i \leq 9$, does not help to the objective of (P4). Even worse, this tightens the i-th caching capacity constraint in (11b), leading to a potential deduction of the overall objective value. From the above discussion, by adding an additional constraint $x[7,1]=x[8,1]=x[9,1]=x[10,1]$ to (P4), we can safely remove the i-th caching causality constraint in (11c), where $i=7,8,9$, without affecting the optimal value of (P4). Notice that the newly added constraint indeed removes 3 redundant variables $\{x[7,1], x[8,1], x[9,1]\}$.

Intuitively, if we apply the above technique to all the consecutive offloading tasks of the same service type, all the causality constraints in (11c) will be completely removed and the optimizing variables will reduce to only $\{x\left[i,t_i\right] | i \in \mathcal{A}\}$. Formally, we denote ν_i^j as the index of the next type-j offloading task since the i-th task, i.e., $\nu_i^j = \{\min_{k \geq i} k | u_{k,j} = a_k = 1\}$. For instance, for the 11-th task in Fig. 2, we have $\nu_{11}^1 = \emptyset$, $\nu_{11}^2 = 11$, and $\nu_{11}^3 = 12$, where $\emptyset$ indicates no such offloading task exists. Then, for any $x[i,j]$, we can equivalently replace $x[i,j] = x[\nu_i^j, j]$ in (P4) which automatically satisfies the caching causality constraints. By doing so, (P4) is equivalently transformed to the following problem

$$(P4-Eq): \quad \underset{x[i,t_i]\in\{0,1\},\ \forall i\in\mathcal{A}}{\text{maximize}} \quad \sum_{i\in\mathcal{A}} s_i^* x\left[i,t_i\right] \tag{23a}$$

$$\text{subject to} \quad \sum_{j=1}^{N} c_j \cdot x[\nu_i^j, j] \leq C,\ i=1,\cdots,M. \tag{23b}$$

Notice that the above problem (P4-Eq) is a standard multidimensional knapsack problem (MKP) [19]. Compared to (P4), the number of binary variables is reduced from MN to only $|\mathcal{A}| \leq M$. Besides, as we will show in the next subsection, many constraints in (23b) are duplicated or redundant. When there is more than one effective constraint in (P4-Eq), there does not exist an fully polynomial-time approximation scheme (FPTAS) unless P = NP. However, for MKP problems of moderate size, plenty of algorithms include hybrid dynamic programming and

branch-and-bound methods can be applied to solve for the exact optimal solution in an acceptable computation time, e.g., within 0.1 second of computation time for overall 500 variables [30].

B. Optimal Caching Placement: A Case Study

In this subsection, we use the example in Fig. 2 to illustrate the problem transformation from (P4) to (P4-Eq). We first apply the above mentioned variable replacement technique to the M constraints in (11b) of (P4) one by one, to construct the corresponding M constraints in (23b) of (P4-Eq). Starting from the first constraint in (23b), we note that $\{\nu_1^1, \nu_1^2, \nu_1^3\} = \{6, 1, 3\}$, and thus focus on variables $\{x[6,1], x[1,2], x[3,3]\}$. Assuming that the service cache is initially empty, we have $x[6,1] = x[1,2] = x[3,3] = 0$. Therefore, the corresponding constraint in (23b) of (P4-Eq) is not necessary. For the second constraint in (23b), we can express the corresponding constraint as $C_2 : c_2 x[4,2] \leq C$ because $\nu_2^2 = 4$. After applying the similar variable replacement procedure to constraint $i = 3, \cdots, 12$, we obtain the M constraints of (P4-Eq), meanwhile eliminating all the redundant variables. Then, we remove the duplicated or redundant constraints in (P4-Eq). For instance, it can be easily verified that the 3-rd constraint C_3 in (23b) is the same as C_2. Besides, for the 6-th and 7-th constraints in (23b), we have

$$C_6 : c_2 x[8,2] + c_3 x[12,3] \leq C, \quad C_7 : c_1 x[10,1] + c_2 x[8,2] + c_3 x[12,3] \leq C,$$

where C_6 is evidently redundant if C_7 is satisfied.

After removing all the duplicated/redundant constraints, (P4-Eq) becomes

$$\begin{aligned} \underset{x[i,t_i]\in\{0,1\},\ \forall i\in\bar{\mathcal{A}}}{\text{maximize}} \quad & \sum_{i\in\bar{\mathcal{A}}} s_i^* x\,[i,t_i] \\ \text{subject to} \quad & C_4 : c_2 x[4,2] + c_3 x[5,3] \leq C, \quad C_5 : c_2 x[8,2] + c_3 x[5,3] \leq C, \\ & C_7 : c_1 x[10,1] + c_2 x[8,2] + c_3 x[12,3] \leq C, \\ & C_9 : c_1 x[10,1] + c_2 x[9,2] + c_3 x[12,3] \leq C, \\ & C_{10} : c_1 x[10,1] + c_2 x[11,2] + c_3 x[12,3] \leq C, \end{aligned} \tag{24}$$

where $\bar{\mathcal{A}} \triangleq \{4, 5, 8, 9, 10, 11, 12\}$ denotes the indices of the remaining tasks. Compared to its original formulation in (P4), the numbers of binary variables and constraints are reduced from from $MN = 36$ to 7, and the number of constraints is reduced from $M = 12$ to 5. Besides, the original generic ILP is converted to a standard 0-1 MKP, for which many specialized exact and approximate solution algorithms are available.

After solving (24) optimally, we can easily retrieve the solution in (P4) from the caching causality property. For example, we see that for the 2-nd program, the optimal solutions are

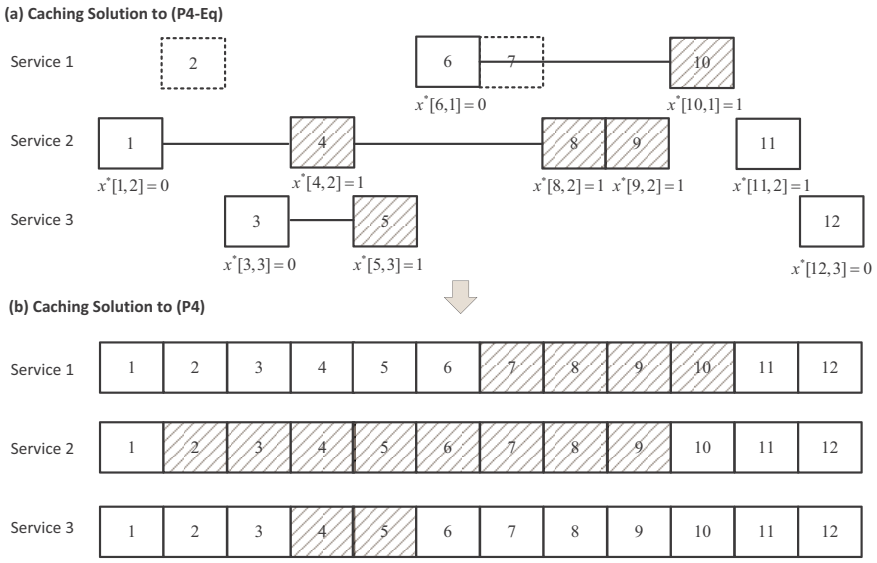

Fig. 3: An example caching solution adapted from Fig. 1. The figure above shows an optimal solution to (P4-Eq) and the figure below denotes the retrieved caching solution to (P4). A shaded (empty) box indicates $x[i,j] = 1$ (0).

retrieved from $\{x^*[4,2], x^*[8,2], x^*[9,2], x^*[11,2]\}$ as $x^*[2,2] = x^*[3,2] = x^*[4,2]$, $x^*[5,2] = x^*[6,2] = x^*[7,2] = x^*[8,2]$, $x^*[10,2] = x^*[11,2]$, while $x^*[i,2] = 0$ for the rest task i. The optimal solution of $x^*[i,1]$'s and $x^*[i,3]$'s can be similarly obtained. As an illustrating example, suppose that the optimal caching solution to (P4-Eq) for the example in Fig. 2 is in Fig. 3(a), the corresponding optimal caching placement solution to (P4) is illustrated in Fig. 3(b).

VI. Optimal Task Offloading Decision

In this section, we optimize the task offloading decision $\mathbf{a}$ given a caching placement decision $\mathbf{X}$ in (P3). Interestingly, we find that the only difficulty lies in optimizing the offloading decisions of the "uncached" tasks, which effectively reduces the number of binary variables.

A. Structure of Task Dependency

Notice that once $\mathbf{X}$ is given, (P3) is reduced to

$$\underset{\mathbf{a}}{\text{minimize}} \quad \sum_{i=1}^{M+1} \left\{ o_i^* \left(1 - a_{i-1}\right) a_i + l_i^* \left(1 - a_i\right) + \left(\lambda_i^* + \beta \tau_i^c\right) a_i + \beta \tau_i^d a_{i-1} \left(1 - a_i\right) \right\} \quad (25a)$$

$$\text{subject to} \quad u_{i-1,j} a_{i-1} \geq x[i,j] - x[i-1,j], \ \forall i, j, \quad (25b)$$

$$a_i \in \{0,1\}, \ i = 1, \cdots, M, \quad (25c)$$

where $a_0 = 0$ and $a_{M+1} = 0$. In the objective function, we have $\lambda_i^* = 0$ if $x\left[i, t_i\right] = 1$, indicating that the program for computing the i-th task is in the service cache, and $\lambda_i^* = s_i^*$ if $x\left[i, t_i\right] = 0$.

We first investigate the underlying graphical structures in the caching causality constraints (25b). We refer to a block of consecutive tasks with $x[i,j]=1$ for a specific program j as a *run*, and denote the index set of the first task of each run as $\mathcal{R}$. Formally, it is defined as $\mathcal{R}=\{i|x[i,j]>x[i-1,j],\forall i,j\}$. For instance, there are in total 3 runs in the caching solution $\mathbf{X}$ in Fig. 3(b) (shaded boxes) and $\mathcal{R}=\{2,4,7\}$. Since $\mathbf{X}$ is a feasible solution to (P3), by definition it must hold that $u_{i-1,j}=a_{i-1}=1$, for any j and $\forall i\in\mathcal{R}$, indicating that the task proceeding a run is of the same service type and must be executed on the edge to upload the program. For instance, $a_1=a_3=a_6=1$ must hold in Fig. 3(b). Meanwhile, for a task $i\notin\mathcal{R}$, the corresponding constraints in (25b) hold automatically regardless of the value of a_{i-1}. In other words, we can equivalent replace all the constraints in (25b) with $a_{i-1}=1$, $\forall i\in\mathcal{R}$, which essentially removes $|\mathcal{R}|$ variables and at the same time all the MN constraints in (25b).

Having converted the caching causality constraints in (25b), we introduce auxiliary variables $b_i=a_{i-1}a_i$, $i=1,\cdots,M$, as in (21), which transforms (25) to the following ILP:

$$\begin{aligned}\underset{\mathbf{a},\mathbf{b}}{\text{minimize}}\quad & \textstyle\sum_{i=1}^{M}\left(o_i^*+\lambda_i^*+\beta\tau_i^c+\beta\tau_{i+1}^d-l_i^*\right)a_i-\sum_{i=1}^{M}\left(o_i^*+\beta\tau_i^d\right)b_i+\sum_{i=1}^{M}l_i^*,\\ \text{subject to}\quad & b_i\le\frac{1}{2}\left(a_{i-1}+a_i\right),\ i=2,\cdots,M,\\ & a_i,b_i\in\{0,1\},\ i=1,\cdots,M,\quad a_{i-1}=1,\ \forall i\in\mathcal{R},\end{aligned}\tag{26}$$

In general, the problem has $2M-|\mathcal{R}|$ binary variables. In the following, we study the properties of optimal offloading decisions to further reduce the complexity of solving a large-size ILP.

B. Reduced-Complexity Decomposition Method

For simplicity of illustration, we denote $y_i\triangleq x[i,t_i]$ to indicate whether the program for computing the i-th task is in the service cache, where $i=1,\cdots,M$. We refer to a task with $y_i=1$ as a *cached task*, and an *uncached task* otherwise. To facilitate illustration, we set $y_0=1$ and $y_{M+1}=0$ for the two virtual tasks without affecting both the objective and constraints of (25). As an illustrative example in Fig. 4, the caching state vector $\mathbf{y}$ consists of alternating patterns of consecutive 0's and 1's. Here, we refer to a block of consecutive tasks with $y_i=1$ as a *cached segment*, and a block of consecutive tasks with $y_i=0$ as an *uncached segment*, such as the three cached segments and three uncached segments in Fig. 4. By this assumption, the M tasks always start with a cached segment and ends with an uncashed segment. Therefore, we always have equal number of cached and uncached segments, which is denoted by $K\ge1$. In the following, we separate our discussions according to the value of K.

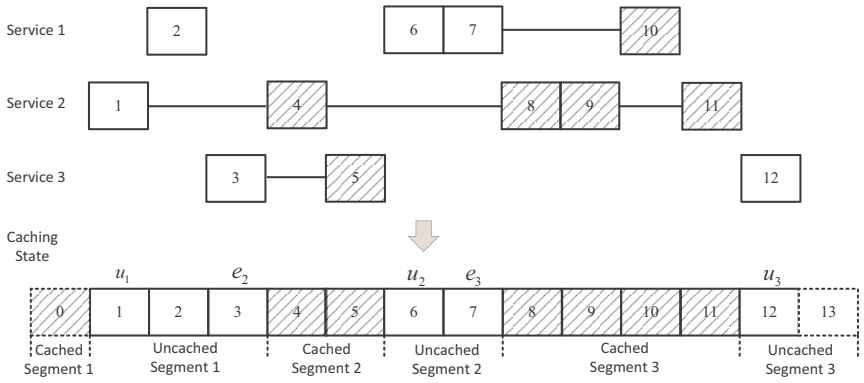

Fig. 4: An example caching state derived from a caching placement. An empty (shaded) box indicates $y_i = 1$ (0).

1) $K = 1$*:* Note that $y_1 = 0$ always holds because the service cache is assumed empty initially (i.e., $x[1, j] = 0$ for all j). Therefore, the first cached segment always has only one task (i.e., task 0). $K = 1$ indicates that $y_i = 0$ for $i = 1, \cdots, M + 1$, i.e., all the tasks are uncached. This indeed is the most difficult case to handle that we need to solve the ILP by setting $\lambda_i^* = s_i^*$ in (26), where $i = 1, \cdots, M$, without any improvement on computational complexity. In practice, however, this case rarely occurs when a proper initial caching placement is set.

2) $K > 1$*:* In this case, there exists some cached task i for $1 < i \leq M$. We denote e_k and u_k as the indices of the uncached tasks preceding and following the k-th cached segment, respectively, while e_1 is not defined. For instance, $u_1 = 1$, $\{e_2, u_2\} = \{3, 6\}$ and $\{e_3, u_3\} = \{7, 12\}$ in Fig. 4. Notice that $u_k = e_{k+1}$ may occur when there is only one task in an uncached segment. For simplicity of illustration, we denote

$$\psi_i = o_i^* \left(1 - a_{i-1}\right) a_i + l_i^* \left(1 - a_i\right) + \left(\lambda_i^* + \beta \tau_i^c\right) a_i + \beta \tau_i^d a_{i-1} \left(1 - a_i\right), \tag{27}$$

such that the objective of (25) is expressed as $\Psi \triangleq \sum_{i=1}^{M+1} \psi_i$. Alternatively, Ψ can be decomposed based on $\{e_k, u_k\}$'s, and problem (25) can be recast as following

$$\underset{\mathbf{a} \in \{0,1\}^M}{\text{minimize}} \qquad \Psi = \sum_{k=1}^{K} \left(\phi_{k,1} + \phi_{k,0}\right) \tag{28a}$$

$$\text{subject to} \qquad a_{i-1} = 1, \ \forall i \in \mathcal{R}, \tag{28b}$$

where

$$\phi_{k,1} = \begin{cases} o_1^* a_1, & k = 1, \\ \sum_{i=e_k+1}^{u_k - 1} \psi_i + \left[o_{u_k}^* \left(1 - a_{u_k - 1}\right) a_{u_k} + \beta \tau_{u_k}^d a_{u_k - 1} \left(1 - a_{u_k}\right)\right] & k = 2, \cdots, K, \end{cases} \tag{29}$$

and

$$\phi_{k,0} = \left[l_{u_k}^* \left(1 - a_{u_k}\right) + \left(\lambda_{u_k}^* + \beta \tau_{u_k}^c\right) a_{u_k}\right] + \sum_{i=u_k+1}^{e_{k+1}} \psi_i, \quad k = 1, \cdots, K. \tag{30}$$

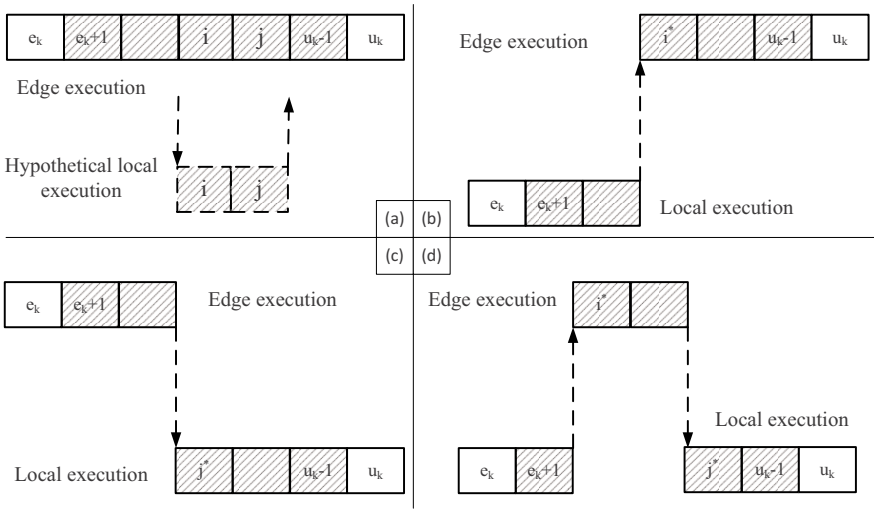

Fig. 5: Optimal offloading decision given the values of a_{e_k} and a_{u_k}.

Intuitively, $\phi_{k,1}$ and $\phi_{k,0}$ correspond to the TEC induced by the k-th cached and uncached segments, respectively. Besides, the sets of optimizing variables in $\phi_{k,1}$ and $\phi_{k,0}$ are

$$\mathcal{A}_{k,1} = \begin{cases} a_1, & k = 1, \\ \{a_i | i = e_k, e_k + 1, \cdots, u_k\}, & k = 2, \cdots, K, \end{cases} \tag{31}$$

and

$$\mathcal{A}_{k,0} = \{a_i | i = u_k, u_k + 1, \cdots, e_{k+1}\}, \ k = 1, \cdots, K. \tag{32}$$

A close observation on (31) and (32) shows that once the values of $\{a_{e_k}, a_{u_k}\}$'s are fixed, for $i = 1, \cdots, K$, $\phi_{k,1}$'s and $\phi_{k,0}$'s can be separately optimized with disjoint sets of variables. In the following, we first discuss the optimal offloading decisions of the cached tasks that minimize $\phi_{k,1}$'s. Without loss of generality, we focus on the k-th cached segment, supposing that $\{a_{e_k}, a_{u_k}\}$ are given. Depending on the values of $\{a_{e_k}, a_{u_k}\}$, there are four cases, as illustrated in Fig. 5.

1) $a_{e_k} = a_{u_k} = 1$, as shown in Fig. 5(a). In this case, the optimal offloading solution is $a_i = 1$, for $i = e_k + 1, \cdots, u_k - 1$. That is, all the cached tasks are executed at the edge server. Due to the page limit, we only provide a sketch of proof here, by contradiction. Suppose that tasks i to j are computed locally in Fig. 5(a). This will not only incur additional time and energy on downloading (uploading) the input (output) of the i-th (j-th) task compared to computing them on the edge, but also additional time and energy on local computation, because $f_0 > f_{max}$ and the energy consumption on edge computation is neglected.
2) $a_{e_k} = 0$ and $a_{u_k} = 1$, as shown in Fig. 5(b). For the optimal offloading decision, there must exist an optimal task $i^* \in \{e_k + 1, \cdots, u_k - 1\}$, such that for each $i = e_k + 1, \cdots, u_k - 1$,

we have

$$a_i^* = \begin{cases} 0, & i < i^*, \\ 1, & i \geq i^*. \end{cases} \tag{33}$$

This indicates that the computation result is offloaded to the edge server exactly once within the segment. The proof follows that in the first case and is omitted for brevity. In this case, i^* can be found via a simple linear search.

3) $a_{e_k} = 1$ and $a_{u_k} = 0$, as shown in Fig. 5(c). For the optimal offloading decision, there must exist an optimal task $j^* \in \{e_k + 1, \cdots, u_k - 1\}$, such that for each $i = e_k + 1, \cdots, u_k - 1$, we have

$$a_i^* = \begin{cases} 1, & i < j^*, \\ 0, & i \geq j^*. \end{cases} \tag{34}$$

The proof also follows the idea in the first case. This indicates that the computation result is downloaded to the MU exactly once within the segment, where j^* can be found using a linear search.

4) $a_{e_k} = a_{u_k} = 0$, as shown in Fig. 5(d), implying that the computations start and end both at the MU. This corresponds to the case in [23], which shows that the optimal computation offloading strategy satisfies a "one-climb" policy where either the tasks are offloaded to the edge server for exactly once, or all executed locally at the MU. There must exist $i^* \leq j^*$, such that optimal solution of a_i, $i = e_k + 1, \cdots, u_k - 1$, is

$$a_i^* = \begin{cases} 0, & i < i^* \text{ or } i \geq j^*, \\ 1, & i^* \leq i < j^*. \end{cases} \tag{35}$$

The optimal $\{i^*, j^*\}$ can be efficiently obtained through a two-dimensional search.

From the above discussion, the optimal value $\phi_{k,1}$ under the above four cases can be efficiently obtained. Let us denote the optimal values by $v_k^{(1)}$, $v_k^{(2)}$, $v_k^{(3)}$, and $v_k^{(4)}$ for the four cases, respectively. Moreover, the calculations of $\left\{v_k^{(1)}, v_k^{(2)}, v_k^{(3)}, v_k^{(4)}\right\}$'s can be performed in parallel for different segments. This way, $\phi_{k,1}$ can be expressed as

$$\phi_{k,1} = v_k^{(1)} a_{e_k} a_{u_k} + v_k^{(2)} \left(1 - a_{e_k}\right) a_{u_k} + v_k^{(3)} a_{e_k} \left(1 - a_{u_k}\right) + v_k^{(4)} \left(1 - a_{e_k}\right) \left(1 - a_{u_k}\right). \tag{36}$$

By substituting (36) into (28), we eliminate all the offloading decision variables corresponding to the cached tasks, and leaving only the variables for the uncached tasks, i.e., $\{a_i | y_i = 0, i = 1, \cdots, M\}$. In the following, we transform (28) into an equivalent ILP problem.

C. Equivalent ILP Formulation

The basic idea is similar to that for (P3) in Section IV-B, where the new challenge is in the multiplicative terms in (36). By denoting $\hat{a}_i \triangleq 1 - a_i$, where $\hat{a}_i \in \{0, 1\}$, we rewrite (36) as

$$\phi_{k,1} = v_k^{(1)} a_{e_k} \left(1 - \hat{a}_{u_k}\right) + v_k^{(2)} \left(1 - a_{e_k}\right) a_{u_k} + v_k^{(3)} a_{e_k} \left(1 - a_{u_k}\right) + v_k^{(4)} \left(1 - a_{e_k}\right) \hat{a}_{u_k}. \tag{37}$$

We further define $q_k \triangleq a_{e_k} a_{u_k}$ and $\hat{q}_k \triangleq a_{e_k} \hat{a}_{u_k}$, and express the above equation as

$$\omega_{k,1} \triangleq \left(v_k^{(1)} + v_k^{(3)}\right) a_{e_k} + v_k^{(2)} a_{u_k} + v_k^{(4)} \hat{a}_{u_k} - \left(v_k^{(1)} + v_k^{(4)}\right) \hat{q}_k - \left(v_k^{(2)} + v_k^{(3)}\right) q_k. \tag{38}$$

By substituting (38) into (28) and introducing auxiliary variables $b_i = a_{i-1} a_i$, we have

$$\begin{aligned}
\underset{\mathbf{a}, \hat{\mathbf{a}}, \mathbf{b}, \mathbf{q}, \hat{\mathbf{q}}}{\text{minimize}} \quad & \textstyle\sum_{k=1}^{K} \left(\omega_{k,1} + \phi_{k,0}\right) && \text{(39a)} \\
\text{subject to} \quad & a_{i-1} = 1, \ \forall i \in \mathcal{R}, && \text{(39b)} \\
& b_i \leq \frac{1}{2} \left(a_{i-1} + a_i\right), \ \forall i \in \mathcal{A}_{k,0} \setminus u_k, k = 1, \cdots, K, && \text{(39c)} \\
& q_k \leq \frac{1}{2} \left(a_{e_k} + a_{u_k}\right), \ k = 2, \cdots, K, && \text{(39d)} \\
& \hat{q}_k \leq \frac{1}{2} \left(a_{e_k} + \hat{a}_{u_k}\right), \ k = 2, \cdots, K, && \text{(39e)} \\
& \hat{a}_{u_k} + a_{u_k} = 1, \ k = 2, \cdots, K, && \text{(39f)} \\
& a_i, \hat{a}_i, b_i, q_k, \hat{q}_k \in \{0, 1\}, \ \forall i, k. && \text{(39g)}
\end{aligned}$$

We see that the inequalities (39c)-(39e) are equivalent to $b_i = a_{i-1} a_i$, $q_k = a_{e_k} a_{u_k}$, and $\hat{q}_k = a_{e_k} \hat{a}_{u_k}$, respectively, because the objective decreases with $\{b_i, q_k, \hat{q}_k\}$'s. Similar to (26), the problem above is also a pure 0-1 integer optimization problem. Compared to (26), it reduces $2|\mathcal{A}_1|$ variables that correspond to the cached tasks, where $\mathcal{A}_1 = \{i | y_i = 1, i = 1, \cdots, M\}$, while introducing additional $3(K-1)$ auxiliary variables. In general, the formulation can effectively reduce the computational complexity when the number of cached tasks is much larger than the number of segments, which is often the case in practice and will be demonstrated in simulations.

D. Alternating Minimization

Sections V.A and VI.C show that we can compute the optimal caching placement $\mathbf{X}^*$ with low complexity when the offloading decision $\mathbf{a}$ is given, and vice versa. This allows us to scheme an alternating minimization that iteratively updates the two set of variables $\mathbf{X}$ and $\mathbf{a}$. Starting from an initial $\mathbf{a}^{(0)}$, we iteratively compute the optimal $\mathbf{X}^{(i)}$ given $\mathbf{a}^{(i-1)}$, and the optimal $\mathbf{a}^{(i)}$ given $\mathbf{X}^{(i)}$ for $i = 1, 2, \cdots$, until the improvement on the objective function of (P3) becomes marginal. Because the objective of (P3) is bounded below and non-increasing as the iterations proceed, the

TABLE I: Simulation Parameters

$B = 10^6$ Hz	$f_{max} = 0.5$ GHz	$O_i \in [2, 5]$ Mb	$d_M = 30$ meters
$\sigma^2 = 10^{-10}$ Watt	$\kappa = 10^{-26}$	$L_i \in [50, 200] \cdot 10^6$ Cycles	$d_e = 2.6$
$P_0 = 1$ Watt	$\beta = 0.1$	$s_j \in [0.5, 1.5]$ Mb	$A_d = 4.11$
$f_0 = 10$ GHz	$M = 400$	$D_j = 3$ seconds, $\forall j$	$f_c = 915$ MHz
$P_{max} = 0.1$ Watt	$N = 6$	Normalized $C = 3$	

alternating minimization method is asymptotically convergent. The detailed algorithm description is omitted for brevity.

VII. Simulation Results

In this section, we evaluate the performance of the proposed algorithms through numerical simulations. All the computations are solved in MATLAB on a computer with an Intel Core i7-4790 3.60-GHz CPU and 16 GB of memory. Besides, Gurobi optimization tools are used to solve the ILP problems [31]. In all simulations, we assume that the average channel gain $\bar{h}_i$ follows a path-loss model $\bar{h}_i = A_d \left(\frac{3 \cdot 10^8}{4\pi f_c d_M}\right)^{d_e}$, $i = 1, \cdots, M$, where A_d denotes the antenna gain, f_c denotes the carrier frequency, d_e denotes the path loss exponent, and d_M denotes the distance between the MU and the edge server. The time-varying fading channel h_i follows an i.i.d. Rician distribution with LOS link power equals to $0.2\bar{h}_i$. Unless otherwise stated, the parameters used in the simulations are listed in Table I, which correspond to a typical outdoor MEC system. For simplicity of illustration, we assume that c_j's are equal for all the programs, such that the caching capacity C is normalized to indicate the number of programs that can cache.

All results in the simulations are the average performance of 50 independent simulations. In each simulation, we first randomly generate M tasks that belong to $N = 6$ types of programs, where the types of the sequential tasks follow a Markov chain with a random initial state. Specifically, the Markov transition probability $P_{i,j} \triangleq \Pr(t_{k+1} = j | t_k = i) = 0.4$ if $i = j$, and $P_{i,j} = 0.12$ if $i \neq j$, $\forall i, j, k$, where t_k denotes the program type of the k-th task. Then, the parameters of each task (O_i and L_i) and each type of program (s_j) are uniformly generated from the ranges specified in Table I for $i = 1, \cdots, M$ and $j = 1, \cdots, N$.

In the following, we evaluate the performance of the considered joint optimization (in Section IV-B) and alternating minimization (in Section VI-D) methods. Specifically, we initialize $a_i = 1$ for all i in the alternating minimization. Besides, we also consider the following benchmark methods for performance comparison:

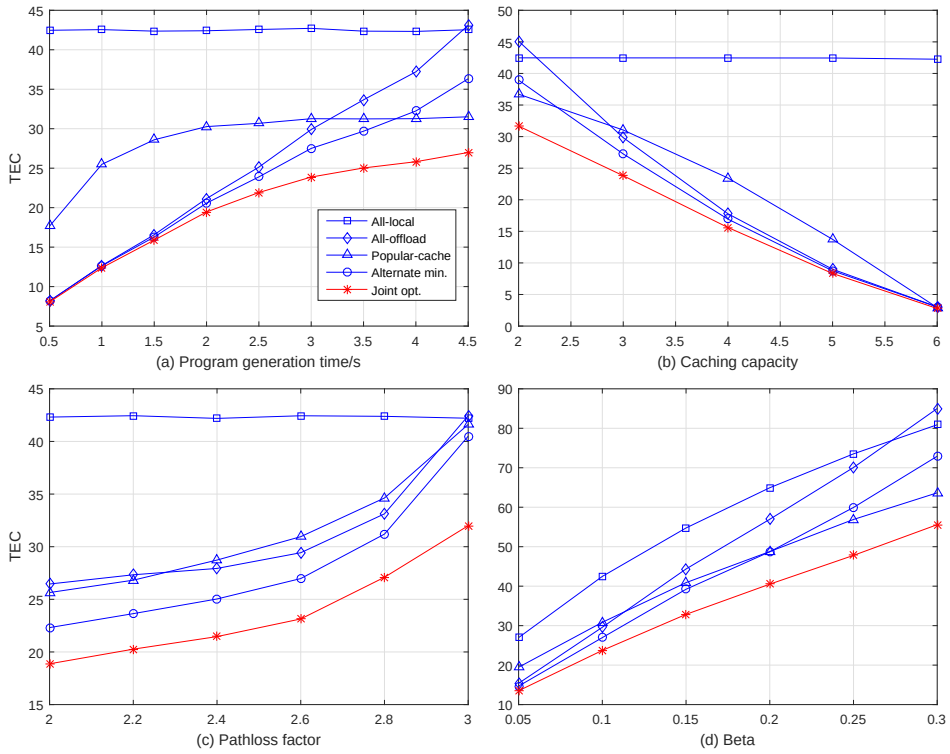

Fig. 6: TEC performance comparisons of different methods.

- Popular-cache: we cache the most popular programs that are executed most frequently by the MU throughout the time. Then, we optimize the offloading decision given the fixed caching placement using the method in Section VI-C.
- All-offload: offload all tasks for edge execution and then optimize the caching placement using the method in Section V-A.
- All-local: all the tasks are computed locally at the MU.

A. TEC Performance Evaluation

We first evaluate the TEC performance under different system setups. In Fig. 6(a), we vary the program generation time D_j from 0.5 to 4.5 seconds, which naturally results in an increase of TEC for all the methods (except for the All-local scheme). Meanwhile, we notice that the Popular-cache method performs closely to the optimal scheme when D_j is small, e.g., $D_j \leq 2$, because the saving in the computation time and energy at the edge server outweighs the overhead of offloading/installing new programs. However, its performance degrades as D_j further increases and is even worse than the All-local scheme when $D_j = 4.5$ because frequent installation of new programs becomes extremely costly. The alternating minimization method has similar trend as

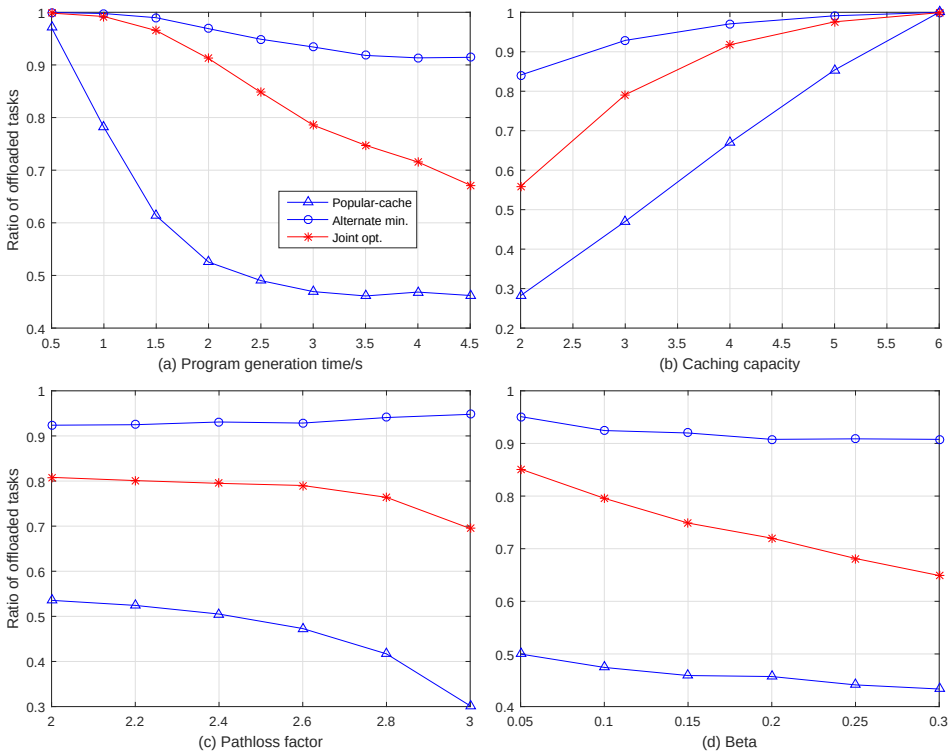

Fig. 7: Ratio of offloaded tasks when different methods are applied.

the All-offload scheme, because it is largely affected by the initialization where all the tasks are offloaded to the edge. Meanwhile, the TEC performance of the Popular-cache method gradually converges as D_j increases. To examine the underlying cause, we plot in Fig. 7(a) the ratio of offloaded tasks. As expected, the offloading ratios of all methods decrease with the program generation time. Specifically, the offloading ratio of the Popular-cache method converges to around 0.5 when D_j is large. This is because the Popular-cache method results in a fixed caching placement throughout the time, where $C = 3$ out of the $N = 6$ programs are cached. Meanwhile, the cached tasks are more likely to be offloaded for computation at the edge server when D_j is large. Accordingly, around half of the tasks are offloaded for edge execution when D_j is large, which also leads to a convergent TEC performance.

In Fig. 6(b) and Fig. 7(b), we vary the normalized caching capacity from 2 to 6. Because a larger caching capacity translates to more savings in program offloading and installation, the TEC decreases and the task offloading ratio increases for all the schemes considered. Specifically, when $C = 6$, i.e., all the programs can be stored in the cache, the All-offload scheme approaches the optimal scheme. In Fig. 6(c) and Fig. 7(c), we vary the path-loss factor d_e from 2 to 3, which leads to drastic decrease of wireless channel gains. As expected, the weaker channels suffer lower

offloading ratio and higher TEC because the higher cost of transmitting the task and program data discourages task offloading. The proposed joint optimization has significant performance gain over all the other schemes considered, especially when d_e is large. Specifically, it reduces the TEC by more than 25% compared with all the other schemes when $d_e = 3$.

At last, in Fig. 6(d) and Fig. 7(d), we vary the weighting parameter β from 0.05 to 0.3, where a smaller (larger) β indicates higher emphasis on minimizing the energy consumption (delay). We notice that the TEC increases with β because the value of delay dominates that of energy consumption (e.g., one order of amplitude larger). Meanwhile, the relatively high program installation delay discourages the tasks to be offloaded for edge execution, leading to a decreased offloading ratio when β increases. As a result, the All-offload scheme performs the worst when $\beta = 0.3$ because of the high delay cost on program generation at the edge server.

Overall, the joint optimization scheme has evident TEC performance advantage over the other schemes. The All-offload scheme performs well only when the program generation time is small, the channels are good, or under less stringent delay requirement. The Popular-cache scheme performs poorly in most cases due to its negligence to the task offloading decisions. This is in contrast to the traditional content caching schemes, where caching popular contents (e.g., large and most frequently accessed files) usually performs well. The alternating minimization has relatively good performance in most scenarios. However, too many tasks are offloaded than actually required in the optimal solution. In the following, we evaluate the computational complexity of the optimal scheme and the alternating minimization method, which outperforms the other benchmark algorithms in most cases.x

B. Complexity Evaluation

We first evaluate in Fig. 8(a) the average number of iterations needed by alternating minimization when the number of tasks M varies from 100 to 600. We see that the average number of iterations does not vary significantly and is below 3 for each M. Meanwhile, we also plot in Fig. 8(b) the average number of segments K and cached tasks $|\mathcal{A}_1|$ during the iterations. It is evident that the K is significantly smaller than $|\mathcal{A}_1|$. On average, $|\mathcal{A}_1|$ is more than 3 times larger than K. This indicates that the proposed decomposition method in Section VI-B is effective in reducing the number of binary variables of the offloading optimization problem in (26).

In Fig. 9(a), we compare the TEC performance of the considered methods when the number of tasks varies. The optimal scheme and the alternating minimization significantly outperforms

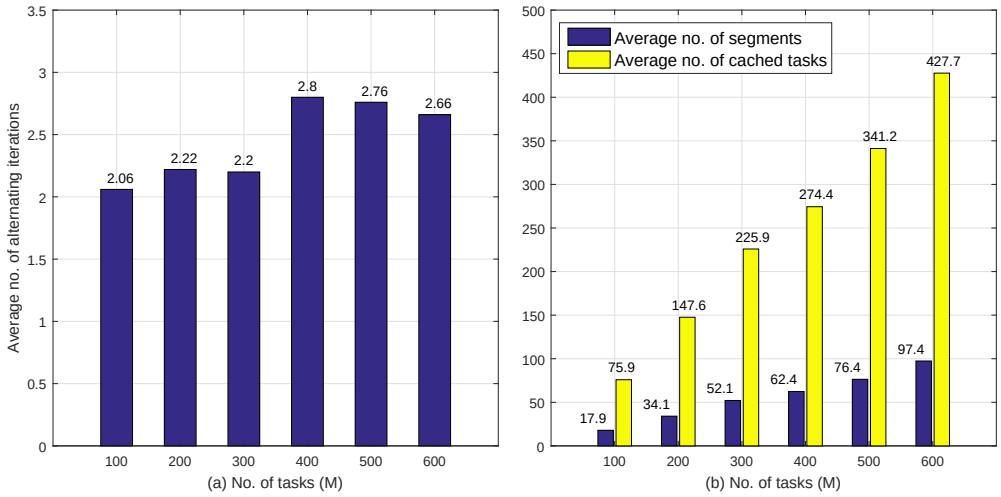

Fig. 8: Performance of the alternating minimization method: (a) average number of iterations used; (b) average number of segments and cached tasks during optimization.

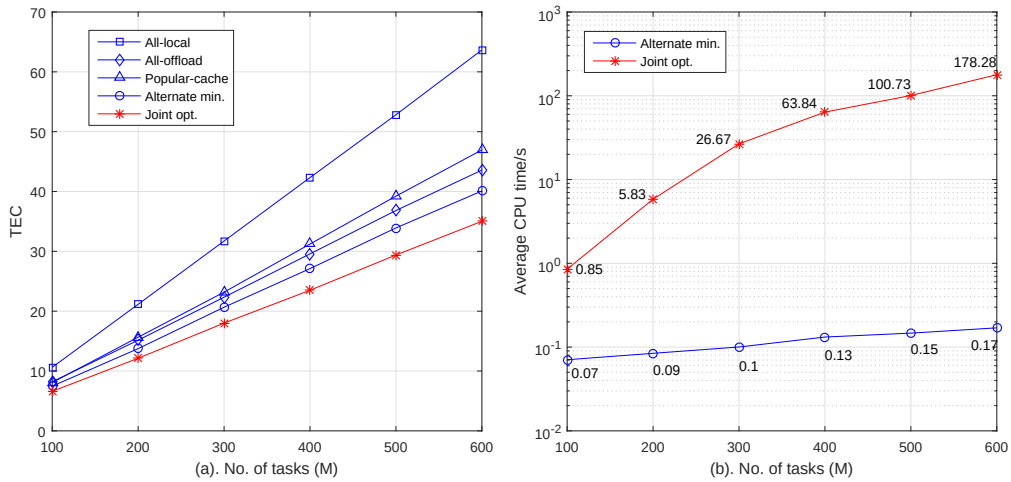

Fig. 9: TEC and CPU time comparisons when the number of tasks varies.

the others. In particular, the optimal scheme achieves on average 13.5% lower TEC than the alternating minimization method. However, the performance advantage comes at a cost of drastic increase of computational complexity. To see this, we plot the average CPU time of the two methods in Fig. 9(b), where the CPU time of alternating minimization increases slightly from 0.07 to 0.17 seconds when M increases by 6 times. In vivid contrast, the CPU time of the joint optimization method increases by more than 200 times from 0.78 to around 3 minutes. The exponential increase of CPU time may result in unaffordable delay in practice when M is large. Therefore, the alternating minimization method provides a reduced-complexity alternative for real-time implementation.

VIII. Conclusions

In this paper, we have considered a cache-assisted MEC system, where an MU uploads and runs its customized programs at the edge server, while the server can selectively cache the previously generated programs for future service reuse. To minimize the computation delay and energy consumption of the MU, we studied the joint optimization of service caching placement, computation offloading decisions, and system resource allocation. We first transformed the complicated MINLP problem to a pure 0-1 ILP problem after deriving the closed-form solution of the optimal resource allocation. Then, we analyzed the graphical structures of the optimal caching placements and offloading decisions, and accordingly proposed reduced-complexity alternating minimization method that optimizes them separately and iteratively. Extensive simulations show that the joint optimization achieves substantial resource savings of the MU compared to other representative benchmark methods considered. In particular, a sub-optimal alternating minimization method achieves a good balance of system performance and computational complexity.

Appendix A
Proof of Lemma 1

Proof: We take the derivative of the objective of (15) with respect to τ_i^u

$$\frac{dL}{d\tau_i^u} = \beta + \frac{(1-\beta)\sigma^2}{h_i}\left(2^{\frac{O_{i-1}}{B\tau_i^u}} - 1 - \ln 2 \cdot 2^{\frac{O_{i-1}}{B\tau_i^u}} \cdot \frac{O_{i-1}}{B\tau_i^u}\right) \tag{40}$$

Because the objective L is strictly convex in $\tau_i^u \geq 0$, $\frac{dL}{d\tau_i^u}$ is an increasing function in τ_i^u. Therefore, if $\frac{dL}{d\tau_i^u} \geq 0$ holds at $\tau_i^u = \frac{O_{i-1}}{R_i^{max}}$, the minimum of L is achieved at $\tau_i^u = \frac{O_{i-1}}{R_i^{max}}$. By substituting $\tau_i^u = \frac{O_{i-1}}{R_i^{max}}$ in (40) and setting $\frac{dL}{d\tau_i^u} \geq 0$, we have

$$\begin{aligned} &\beta + (1-\beta)P_{max}\left[1 - \ln(1+q_i)\left(\frac{1}{q_i}+1\right)\right] \geq 0 \\ \Rightarrow &\ln(1+q_i) \leq \left(1 + \frac{\beta}{(1-\beta)P_{max}}\right)\left(1 - \frac{1}{1+q_i}\right) \Rightarrow \ln\left(\frac{1}{1+q_i}\right) \geq -A + \frac{A}{1+q_i}, \end{aligned} \tag{41}$$

where $q_i \triangleq \frac{h_i P_{max}}{\sigma^2}$ and $A \triangleq 1 + \frac{\beta}{(1-\beta)P_{max}}$. By taking a natural exponential operation at both sides of (41), we have

$$\exp\left(-\frac{A}{1+q_i}\right)\left(\frac{1}{1+q_i}\right) \geq \exp(-A) \Rightarrow \exp\left(-\frac{A}{1+q_i}\right)\left(-\frac{A}{1+q_i}\right) \leq -A\exp(-A).$$

Evidently, the RHS of the above inequality satisfies $e^{-1} \leq -A\exp(-A) \leq 0$. Then, the above inequality can be equivalently expressed as

$$-\frac{A}{1+q_i} \leq \mathcal{W}\left(-A\exp(-A)\right), \tag{42}$$

where $\mathcal{W}(x)$ denotes the Lambert-W function, which is the inverse function of $f(z) = z\exp(z) = x$, i.e., $z = \mathcal{W}(x)$. The equivalence holds because $\mathcal{W}(x)$ is an increasing function when $x \geq -1/e$. From (42), the condition that $\frac{dL}{d\tau_i^u} \geq 0$ at $\tau_i^u = \frac{O_{i-1}}{R_i^{max}}$ is equivalent to

$$h_i \leq \frac{\sigma^2}{P_{max}}\left(\frac{A}{-\mathcal{W}\left(-A\exp\left(-A\right)\right)} - 1\right). \tag{43}$$

Otherwise, if (43) does not hold, i.e., $\frac{dL}{d\tau_i^u} < 0$ at $\tau_i^u = \frac{O_{i-1}}{R_i^{max}}$, we set the $\frac{dL}{d\tau_i^u} = 0$ in (19) to find the minimum. That is,

$$\begin{aligned}\frac{dL}{d\tau_i^u} &= \frac{(1-\beta)\sigma^2}{h_i}\left[\frac{\beta h_i}{(1-\beta)\sigma^2} - 1 - 2^{\frac{O_{i-1}}{B\tau_i^u}}\left(\ln 2\cdot\frac{O_{i-1}}{B\tau_i^u} - 1\right)\right] \\ &= \frac{(1-\beta)\sigma^2 e}{h_i}\left[e^{-1}\left(\frac{\beta h_i}{(1-\beta)\sigma^2} - 1\right) - e^{\ln 2\frac{O_{i-1}}{B\tau_i^u}-1}\left(\ln 2\cdot\frac{O_{i-1}}{B\tau_i^u} - 1\right)\right] = 0,\end{aligned} \tag{44}$$

which is equivalent to

$$\ln 2\frac{O_{i-1}}{B\tau_i^u} - 1 = \mathcal{W}\left(e^{-1}\left[\frac{\beta h_i}{(1-\beta)\sigma^2} - 1\right]\right). \tag{45}$$

Therefore, we have

$$(\tau_i^u)^* = \frac{\ln 2\cdot O_{i-1}}{B\cdot\left[\mathcal{W}\left(e^{-1}\left[\frac{\beta h_i}{(1-\beta)\sigma^2} - 1\right]\right) + 1\right]}. \tag{46}$$

The above results lead to the proof of Lemma 1. ■

REFERENCES

[1] Q. Zhang, L. Cheng, and R. Boutaba, "Cloud computing: state-of-the-art and research challenges," *J. Internet Serv. Appl.*, vol. 1, no. 1, pp. 7-18, May 2010.

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

[3] Y. Cui, W. He, C. Ni, C. Guo, and Z. Liu, "Energy-efficient resource allocation for cache-assisted mobile edge computing," in *Proc. IEEE LCN*, pp. 640-648, Oct. 2017.

[4] P. Liu, G. Xu, K. Yang, K. Wang, and X. Meng, "Jointly optimized energy-minimal resource allocation in cache-enhanced mobile edge computing systems," *IEEE Access*, vol. 7, pp. 3336-3347, Dec. 2018.

[5] G. Lee, W. Saad, and M. Bennis, "Online optimization for low-latency computational caching in fog networks," in *Proc. IEEE FWC*, 2017.

[6] Y. Sun, Z. Chen, M. Tao, and H. Liu, "Bandwidth gain from mobile edge computing and caching in wireless multicast systems," submitted for publication, available on-line at arxiv.org/abs/1702.00606.

[7] S-W. Ko, K. Huang, S-L. Kim, and H. Chae, "Live prefetching for mobile computation offloading," *IEEE Trans. Wireless Commun.*, vol. 16, no. 5, pp. 3057-3071, May 2018.

[8] E. Jonas, J. Schleier-Smith, V. Sreekanti, et al., "Cloud programming simplified: a Berkeley view on serverless computing," available on-line at arxiv.org/abs/1902.03383.

[9] P. K. Gunda, L. Ravindranath, C. A. Thekkath, Y. Yu, and L. Zhuang, "Nectar: automatic management of data and computation in datacenters," in *Proc. OSDI*, 2010.

[10] T. He, H. Khamfroush, S. Wang, T. L. Porta, and S. Stein, "It's hard to share: joint service placement and request scheduling in edge clouds with sharable and non-sharable resources," in *Proc. ICDC*, pp. 365-375, 2018.

[11] K. Poularakis, J. Llorca, A. M. Tulino, I. Taylor, and L. Tassiulas, "Joint service placement and request routing in multi-cell mobile edge computing networks," to appear in *Proc. IEEE INFOCOM 2019*, available on-line at arxiv.org/abs/1901.08946.

[12] Q. Xie, Q. Wang, N. Yu, H. Huang, and X. Jia, "Dynamic service caching in mobile edge networks," in *Proc. IEEE MASS*, 2018.

[13] T. Zhao, I.-H. Hou, S. Wang, and K. Chan, "Red/LeD: an asymptotically optimal and scalable online algorithm for service caching at the edge," *IEEE J. Sel. Areas in Commun.*, vol. 36, no. 8, pp. 1857-1870, Aug. 2018.

[14] L. Chen, J. Xu, S. Ren, and P. Zhou, "Spatio-temporal edge service placement a Bandit learning approach," *IEEE Trans. Wireless Commun.*, vol. 17, no. 12, pp. 8388-8401, Dec. 2018.

[15] J. Xu, L. Chen, and P. Zhou, "Joint service caching and task offloading for mobile edge computing in dense networks," in *Proc. IEEE INFOCOM*, pp. 207-215, 2018.

[16] S. S. Manvi and G. K. Shyam. "Resource management for Infrastructure as a Service (IaaS) in cloud computing: A survey," *J. Netw. Comput. Appl.*, vol. 41, pp. 424-440, May 2014.

[17] Y. Hao, M. Chen, L. Hu, M. S. Hossain, and A. Ghoneim, "Energy efficient task caching and offloading for mobile edge computing," *IEEE ACCESS*, vol. 6, pp. 11365-11373, Mar. 2018.

[18] C. H. Papadimitriou and K. Steiglitz, *Combinatorial Optimization: Algorithms and Complexity*, Courier Corporation, New York, Dover, 1998.

[19] A. Freville, "The multidimensional 0-1 knapsack problem: An overview," *Eur. J. Oper. Res.*, vol. 155, no. 1, pp. 1-21, 2004.

[20] C. You, K. Huang, H. Chae, and B.-H. Kim, "Energy-efficient resource allocation for mobile-edge computation offloading," *IEEE Trans. Wireless Commun.*, vol. 16, no. 3, pp. 1397-1411, Mar. 2017.

[21] M.-H. Chen, B. Liang, and M. Dong, "Joint offloading decision and resource allocation for multi-user multi-task mobile cloud," in *Proc. IEEE Int. Conf. Commun. (ICC)*, Kuala Lumpur, Malaysia, May 2016, pp. 1-6.

[22] M. Liu and Y. Liu, "Price-based distributed offloading for mobile-edge computing with computation capacity constraints," *IEEE Wireless Commun. Lett.*, vol. 7, no. 8, pp. 420-423, Jun. 2018.

[23] J. Yan, S. Bi, Y. J. Zhang, and M. Tao, "Optimal offloading and resource allocation in mobile-edge computing with inter-user task dependency," submitted for publication, available on-line at arxiv.org/abs/1810.11199

[24] F. Wang, J. Xu, X. Wang, and S. Cui, "Joint offloading and computing optimization in wireless powered mobile-edge computing systems," *IEEE Trans. Wireless Commun.*, vol. 17, no. 3, pp. 1784-1797, Mar. 2018.

[25] S. Bi and Y. J. Zhang, "Computation rate maximization for wireless powered mobile-edge computing with binary computation offloading," *IEEE Trans. Wireless Commun.*, vol. 17, no. 6, pp. 4177-4190, Jun. 2018.

[26] L. Huang, S. Bi, and Y. J. Zhang, "Deep reinforcement learning for online offloading in wireless powered mobile-edge computing networks," submitted for publication, available on-line at arxiv.org/abs/1808.01977.

[27] Y. Wang, M. Sheng, X. Wang, L. Wang, and J. Li, "Mobile-edge computing: partial computation offloading using dynamic voltage scaling," *IEEE Trans. Commun.*, vol. 64, no. 10, pp. 4268-4282, Oct. 2016.

[28] S. Boyd and L. Vandenberghe, *Convex Optimization*, Cambridge University Press, 2004.

[29] S. S. Rao, *Engineering Optimization: Theory and Practice*, 4th ed. Hoboken, NJ, USA: Wiley, 2009.

[30] K. Berger and F. Galea, "An efficient parallelization strategy for dynamic programming on gpu," in *Proc. IEEE IPDPSW*, pp. 1797-1806, May 2013.

[31] Gurobi Optimization [Online]. Available: http://www.gurobi.com/