Computation Offloading for Frame-Based Real-Time Tasks under Given Server Response Time Guarantees

Anas Toma[1] **and Jian-Jia Chen**[2]

1 **Department of Informatics, Karlsruhe Institute of Technology, Germany**
anas.toma@student.kit.edu
2 **Department of Informatics, TU Dortmund University, Germany**
jian-jia.chen@cs.uni-dortmund.de

Abstract

Computation offloading has been adopted to improve the performance of embedded systems by offloading the computation of some tasks, especially computation-intensive tasks, to servers or clouds. This paper explores computation offloading for real-time tasks in embedded systems, provided given response time guarantees from the servers, to decide which tasks should be offloaded to get the results in time. We consider frame-based real-time tasks with the same period and relative deadline. When the execution order of the tasks is given, the problem can be solved in linear time. However, when the execution order is not specified, we prove that the problem is $\mathcal{NP}$-complete. We develop a pseudo-polynomial-time algorithm for deriving feasible schedules, if they exist. An approximation scheme is also developed to trade the error made from the algorithm and the complexity. Our algorithms are extended to minimize the period/relative deadline of the tasks for performance maximization. The algorithms are evaluated with a case study for a surveillance system and synthesized benchmarks.

2012 ACM Subject Classification Software and its engineering, Scheduling, Computer systems organization, Real-time systems

Keywords and phrases Computation offloading, task scheduling, real-time systems

Digital Object Identifier 10.4230/LITES-v001-i002-a002

Received 2013-02-28 **Accepted** 2014-08-22 **Published** 2014-11-14

1 Introduction

In the recent years, a significant increase in the development of mobile devices has been achieved. They have become devices that provide various computation-intensive services and applications, including video, audio, images, etc. Also, mobile robots have become more and more popular and important in the recent years. For instance, the sales of service robots for personal and household purposes have been increased significantly in the past years, i. e., 35 % increase in 2010 [7]. Furthermore, the number of service robots sold per year is also expected to increase in the next few years [7].

However, due to the resource constraints on both mobile devices and robots, their computation capabilities are still quite limited. For some applications on these devices, if the peak performance requirement happens rarely or is not always required, designing the embedded system for the extreme case to achieve the peak performance is usually too pessimistic, as most resources will be wasted. Moreover, when increasing the performance of an embedded system, we will also usually increase the power consumption, the weight, and also the cost of the devices.

Improving the embedded systems just for extreme cases, for executing some computation-intensive applications, may waste the device resources in normal cases if the extreme case is needed rarely. Therefore, *computation offloading* can be used to move a task from a resource-constrained device (here, we call it a *client*) to one or more devices (here, we call them *servers*). Figure 1

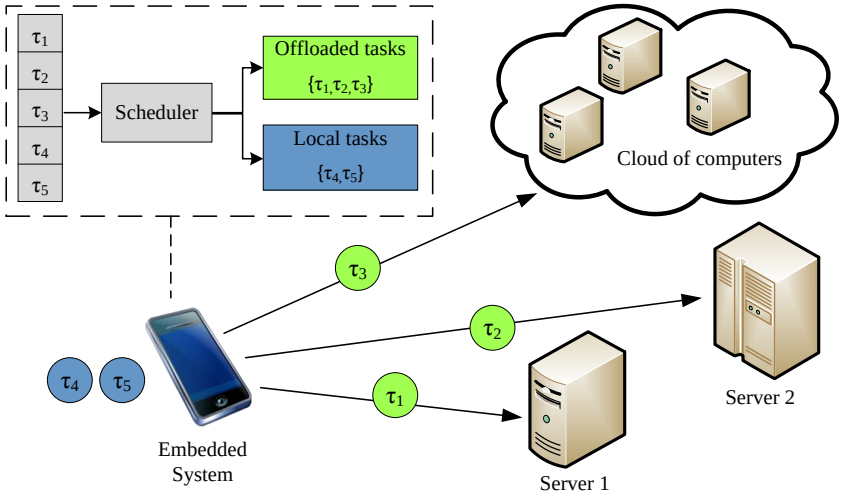


Figure 1 Offloading Mechanism.

illustrates the computation offloading mechanism. The task can be a part of an active program (e. g., function, class, etc.) or a complete one. The servers can either provide faster execution in general (e. g., powerful desktop, an array of high-performance blade servers, cloud of computers, etc.) or accelerate the execution for some specific tasks (e. g., digital signal processing (DSP) units for signal decoding/encoding, General-purpose computing on graphics processing units (GPGPU) for accelerating, etc.). Furthermore, the server may be slower than the client. For such a case, the offloading may also be beneficial. Because the computation is done remotely, the energy consumption of the client can be reduced, or another task can be executed on the client while awaiting the results from the servers.

For example, some computation-intensive real-time tasks may be required to run on the *Electronic Control Units* (**ECU**s), that are distributed in the the automobiles, for specific time. However, this resource-constrained ECUs may not be able to finish the tasks execution in time. Improving the ECUs just for the extreme cases, if they happen rarely, to execute computation-intensive tasks may waste the resources in the normal cases and increases their cost. Therefore, offloading the computation-intensive tasks to a server (i. e., an additional processing unit inside the automobile with timing predictable communication), that serves all the ECUs in the extreme cases, is a cheaper and more flexible solution.

The idea of computation offloading has been studied previously [16, 9, 17, 10, 6, 3, 12, 8]. The existing approaches decide whether to execute a task locally or offload it without changing the execution order for the independent tasks. So, the client remains idle during the remote execution of an offloaded task until the result of this task returns from the server. Also, they consider, implicitly, a dedicated server for each client to run the offloaded task immediately. Furthermore, most of the existing computation offloading approaches either do not consider the timing satisfaction requirement for real-time properties, e. g., in [9, 16, 10, 6, 17], or use pessimistic offloading mechanism for deciding whether a task can be offloaded [12]. Timing requirements are important for real-time embedded systems, in which the results may become useless or even harmful to the client if the deadlines are missed.

Our Contributions. In this paper, computation offloading is exploited for real-time systems to meet the timing constraints. We consider frame-based real-time tasks with the same period and relative deadline under given response time guarantees from the servers. Our model is more

applicable for real-time embedded systems than the existing related work [16, 9, 17, 10, 6, 3, 12], in which (1) the client can execute another task locally while some offloaded tasks are executed on the servers, and (2) the server is not dedicated to a client to provide the service immediately, but provide a certain response timing assurance for the offloaded tasks. Our contributions are as follows:

- We prove that the offloading problem is $\mathcal{NP}$-complete even for frame-based real-time tasks with the same period and relative deadline without a specified execution order.
- We develop algorithms for deciding which tasks to be offloaded and how the tasks are executed to meet the timing constraints, for frame-based real-time tasks. We consider two cases, depending on whether the execution order of the tasks on the client is given or not. In case the order is given, the problem can be solved efficiently. Otherwise, we develop a pseudo-polynomial-time algorithm to derive a feasible schedule, if and only if it exists.
- We also provide an approximation scheme to trade the error made from the algorithm and the time/space complexity.
- Our algorithms can also be extended to maximize the sampling rate of the frame-based tasks by minimizing the period/relative deadline of the tasks.
- We evaluate for our proposed algorithms using a case study of a real-world application and randomly synthesized benchmarks. In our case study, a surveillance system is used to capture images periodically and execute four tasks within a deadline (i. e. sampling period).

The remainder of this paper is organized as follows: Section 2 summarizes the related work on computation offloading. Section 3 provides system model. Section 4 presents an efficient algorithm when the execution order is given. The hardness of the studied problem is shown in Section 5. Section 6 presents our approaches when the execution order is not given. Experimental results are presented in Section 7, and Section 8 concludes the paper.

2 Related Work

Computation offloading has been adopted in the literature to satisfy real-time requirements [12], improve performance [16], save energy [9, 17, 10, 6], and improve the quality of service [3].

For reducing the execution time and also the response time, Nimmagadda et al. [12] propose an offloading framework for mobile robots to satisfy the real-time constraints. Also, Wolski et al. [16] formulate computation offloading as a statistical decision problem by considering both the client and the servers are in computational grids. Offloading decisions in both of the above approaches are based on the comparison between two values: (1) the local execution time, and (2) the summation of the expected remote execution time in the server(s) and data transfer time. If the second value is less than the first one for a specific task, then this task is offloaded to the server(s) [12, 16].

Hong et al. [6] present an offloading strategy with three offloading options to reduce the energy consumption. Their strategy is dedicated for content-based image retrieval applications in mobile systems. For handheld devices, Li et al. [9] develop a scheme to run a program (task) by characterizing its corresponding client subtasks and server subtasks for executing on the client and servers, respectively. They build a cost graph for each program and use a branch-and-bound algorithm to minimize the energy consumption of the client. Moreover, Li et al. [10] also develop a computation offloading scheme by applying the standard maximum-flow/minimum-cut algorithm for deciding the server and client subtasks. For reducing the energy consumption, Xian et al. [17] apply timeout mechanism so that a task will be offloaded to a server if it cannot be finished before the timeout (timestamp) set for it. A middleware for mobile Android platforms is developed by Kovachev et al. [8] to offload the computation-intensive tasks from the mobile device to a remote

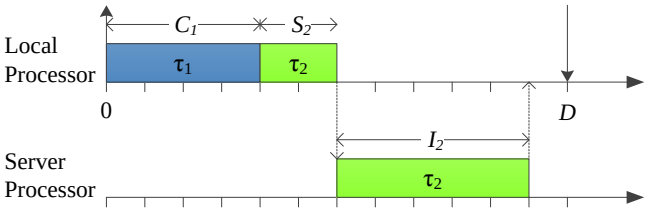


Figure 2 Timing parameters for two tasks.

cloud. The Offloading decision is represented as an optimization problem and solved using Integer Linear Programming (ILP).

3 System Model

We consider a system of one client and one or more servers for computation enhancement. Servers may provide higher computation capability than the client. On the client side, a set of frame-based real-time tasks arrive periodically and require execution within a common relative deadline. The tasks can be offloaded to the servers, but the results should be returned in time, i. e., no later than the deadline. The tasks are independent in execution without precedence constraints. The client has to schedule task executions to satisfy the real-time constraints.

3.1 Client Side

Suppose that we are given a set $\mathcal{T}$ of n independent frame-based real-time tasks. Each task τ_i in $\mathcal{T}$ (for $i = 1, 2, \ldots, n$) represents an execution unit, and it can be considered as an infinite sequence of instances, which called jobs. All the tasks have the same arrival time 0, period D and relative deadline D, i. e., with implicit deadlines. Each task $\tau_i \in \mathcal{T}$ is associated with the following timing parameters:

- **Worst-case *local* execution time C_i**: If task τ_i is decided to be executed locally on the client, the worst-case execution time required to finish task τ_i is up to C_i.
- **Setup time S_i:** is the execution time required on the client so that the required information can be sent to a corresponding server for offloading. It includes transmission time to the server and any local pre-processing operations such as data compression and transformation. As a result, when a task τ_i is offloaded, it has to be executed on the client for up to S_i amount of time, we say that τ_i is *settled for offloading*. After the setup time finishes on the client for offloading, the corresponding server can start task processing on its side.[1]
- **Round-trip offloading time I_i:** the interval length starting from the end of setting up S_i for task τ_i until getting the result from the server. If a server, or a processor, with a speed-up factor of α is dedicated for each offloaded task, then I_i is equal to the execution time on its side which can be computed as $\frac{C_i}{\alpha}$. Otherwise, when the server may handle more than one task, the server has its own scheduling policy and it provides a response time guarantee I_i for each task. The client contacts the server/s before scheduling to get the values of I_i. Subsection 3.3 describes how this value can be calculated.

[1] If the transmission time can be estimated with the worst case when the communication fabric is timing predictable, the worst-case transmission time can be used for guaranteeing the setup time. Otherwise, a pessimistic estimation can be used for providing soft timing analysis. For example, the transmission time can be computed as $\frac{Z}{\beta}$, where Z is the estimated maximum size of the offloaded data and β is the estimated lowest network bandwidth between the client and the server.

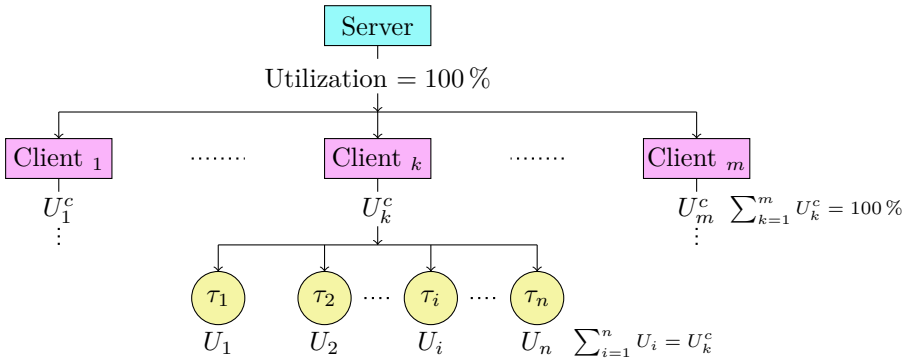


Figure 3 The distribution of the server's utilization.

Figure 2 shows these timing parameters for two tasks, where task τ_1 is locally executed and task τ_2 is offloaded. We assume that the results returned from the servers need very short post processing time, which is negligible. For instance, the returned results in our case study are the coordinates of the moving object or the distance between it and the cameras. Therefore, an offloaded task is said to meet the deadline/timing constraint if the result can return before the deadline. Our model is a special case of a general model (where each task has its own arrival time, period and relative deadline) that has never been considered before for offloading. Also, we prove that the offloading problem for this model is $\mathcal{NP}$-complete.

We say that a task is *locally executed* if it is processed on the client, while a task is called *offloaded* if it is processed on a server. The *finishing time* of a locally-executed task is the time that the task finishes its local execution. The finishing time of an offloaded task is its round-trip offloading time plus the time that this task is settled for offloading.

3.2 Server Side

The server can provide offloading services for more than one client, and the offloading decisions from a client will not control how the servers schedule the tasks. The servers can have their own scheduling policies to handle the tasks that are offloaded from the clients. They can decide how to provide the response time guarantee by themselves. For example, servers can use Earliest Deadline First (EDF) scheduling algorithm or resource reservation servers to ensure I_i. The response time guarantee can be either (1) hard if the scheduling in the servers and the communication fabric between the client and the servers are both timing predictable, or (2) soft if only the scheduling in the servers is timing predictable. However, for each case, when the client has the information about the round-trip offloading time, the open problem is how to meet the timing constraint by exploiting the services provided from the servers.

3.3 Calculating the Value of I_i

To calculate the value of I_i, the server has to provide a response time guarantee for the offloaded tasks. *Resource reservation* technique [2] (the resource here is the CPU of the server) can be used to provide such guarantee, and then satisfy the real-time constraints. In *Resource Reservation Server*[2] (RRS) model, the client can be given a bandwidth or a budget guarantee. In this paper, we consider the *Total Bandwidth Server* (TBS) model [14, 15] as a RRS on the server side. In this

[2] This is a logical server, inherited from the literature.

model, the server reserves a specific bandwidth (or utilization) U_k^c for each requesting client, if it is possible. U_k^c represents the fraction of the processor bandwidth of the server that is assigned to the client $_k$, where $1 \leq k \leq m$ and m is the total number of clients. The total reserved (or given) utilization for all clients should not exceed 100 %, i. e., $\sum_{k=1}^{m} U_k^c = 100\,\%$. Using this technique, the server is able to provide offloading services for more than one client without violating the real-time constraints.

For a client k with a given bandwidth of U_k^c and n tasks, the server allocates a TBS for each task τ_i with a utilization of U_i, such that $\sum_{i=1}^{n} U_i = U_k^c$ to preserve the system feasibility. Figure 3 shows how the utilization of the server can be distributed. A client $_k$ with a given utilization of U_k^c can divided it equally over all of its tasks, i. e., $U_i = \frac{U_k^c}{n}$, or with different ratios based on a specific algorithm. A task τ_i with a given utilization of U_i seems to be executed alone on a processor (TBS) which is $\frac{1}{U_i}$ times slower than the processor of the server. The TBS assigns an absolute deadline $d_i(t)$ for each offloaded task τ_i as follows:

$$d_i(t) = \max\{t, d_i(t^-)\} + \frac{R_i}{U_i}\,,$$

where t is the arrival time of the task at the server side, $d_i(t^-)$ is the absolute deadline of the previous instance (or frame), R_i is the remote execution time of the task (the execution time on the server side), and $d_i(0^-)$ is defined as 0. The offloaded tasks are scheduled on the server side using the Earliest Deadline First (EDF) algorithm based on the assigned TBS deadlines.

The candidate tasks for offloading are the tasks with $S_i + I_i \leq D$, i. e., feasible for offloading. Therefore, all the offloaded tasks finish within the deadline D (before the next frame), and then $t > d_i(t^-)$. Also, the task τ_i arrives at the server side immediately after the setting up time S_i (the transmission time is included in S_i). So, the round-trip offloading time can be calculated as $I_i = \frac{R_i}{U_i}$. In this way, each task can be executed independently of the behavior or the order of the other tasks.

3.4 Problem Definition

The problem explored in this paper is defined as follows:
Given a set $\mathcal{T}$ of n frame-based real-time tasks, the SElective Real-Time Offloading (SERTO) problem is to schedule the tasks and to decide when and what to offload without violating timing constraints for a client.

We consider two types of input instances of the SERTO problem, depending on whether the task execution ordering on the client is given or not. When the execution order is given and has to be preserved, we suppose that τ_i is executed on the client (either with S_i amount of time for offloading or C_i amount of time for local execution) before τ_j if $i < j$.

A schedule of a set $\mathcal{T}$ of tasks for the SERTO problem is an assignment of the executions of the tasks either on the client locally or on a remote server with computation offloading. A schedule is *feasible* if the finishing times of all locally-executed and offloaded tasks are within the deadline D. A scheduling algorithm is said to be *optimal offloading scheduling* algorithm if it is able to find a feasible schedule, if and only if one exists. Moreover, as we are dealing with frame-based real-time tasks, we always consider how to scheduling within a frame, starting from time 0. Therefore, the response time of a task is the same as the finishing time of a task.

Suppose that x_i is equal to 1 if task τ_i is decided to be offloaded; otherwise, x_i is 0. We use a vector $\vec{x}_n = (x_1, x_2, \ldots, x_n)$ to denote an *offloading decision* for the given n tasks.

Algorithm 1 GMF

```
1: t_1 ← 0;
2: for i = 1 to n do
3:    if S_i < C_i and t_i + S_i + I_i ≤ D then
4:       τ_i is assigned for offloading;
5:       t_{i+1} ← t_i + S_i;
6:    else if t_i + C_i ≤ D then
7:       τ_i is assigned for local computation;
8:       t_{i+1} ← t_i + C_i;
9:    else
10:      return "There is no feasible schedule";
11:   end if
12: end for
```

4 Greedy Minimum Finishing Algorithm

In this section we consider a set $\mathcal{T}$ of tasks with a given execution order. Let the tasks be indexed based on the given execution order from 1 to n, where n is the number of tasks. The problem is to decide whether a task should be executed locally or to be offloaded without violating timing constraints.

Under the given ordering, the SERTO problem can be solved by a greedy algorithm, called *Greedy Minimum Finishing (**GMF**)*. Suppose that t_i is the time when task τ_i can start to execute in the client, either offloaded or locally executed. The greedy algorithm simply makes the decision to offload a task τ_i if it is beneficial and feasible: that is, if $S_i < C_i$ (beneficial for offloading) and $t_i + S_i + I_i \leq D$ (feasible for offloading). If it is either not beneficial ($S_i \geq C_i$) or not feasible ($t_i + S_i + I_i > D$) for offloading, the algorithm checks if it can be executed locally, i. e., $t_i + C_i \leq D$. Otherwise, there is no feasible solution. Algorithm 1 presents the pseudo-code of the *GMF* algorithm. The time complexity of the algorithm is $O(|\mathcal{T}|)$.

▶ **Theorem 1.** *The GMF algorithm is an optimal offloading scheduling algorithm for the SERTO problem when the execution ordering is given.*

Proof. This theorem can be proved by an induction on the value t_i. We claim that t_{i+1} in the GMF algorithm is the *earliest* time on the client that τ_i finishes its local execution or is settled for offloading and τ_{i+1} can start to run by following the given ordering. For the **base case**, when $i = 1$, the statement is correct by definition.

Inductive step: Assume that t_{k+1} is the *earliest* time on the client that τ_k finishes its local execution or is settled for offloading and τ_{k+1} can start to run, for $k \geq 2$. There are two cases to run task τ_{k+1}:

- τ_{k+1} is offloaded: For such a case, we know that $t_{k+1} + S_{k+1} + I_{k+1} \leq D$ and $S_{k+1} \leq C_{k+1}$.
- τ_{k+1} is locally executed: For such a case, we know that either $t_{k+1} + S_{k+1} + I_{k+1} > D$ or $S_{k+1} > C_{k+1}$.

For both cases, we know that t_{k+2} is also the earliest time on the client that τ_{k+1} finishes its local execution or is settled for offloading and τ_{k+2} can start to run.

Clearly, if task τ_{k+1} cannot finish before the deadline D, the schedule is infeasible and there is no feasible schedule for the first $k + 1$ tasks. Therefore, based on the induction hypothesis, this theorem is proved. ◀

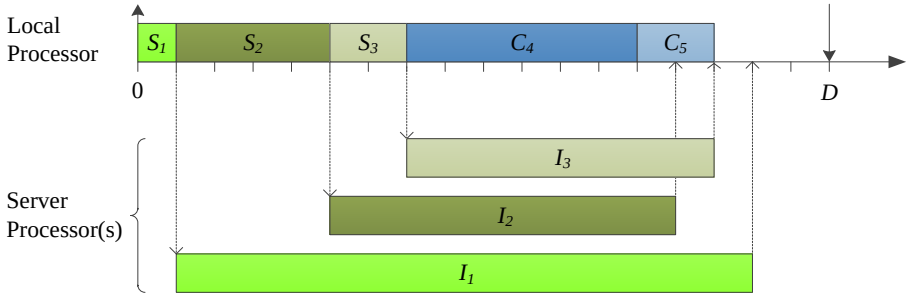


Figure 4 Example of optimal ordering for a set of tasks.

5 Hardness of the SERTO Problem

This section presents the $\mathcal{NP}$-completeness of the SERTO problem. Throughout this section, we implicitly consider the case that the execution ordering is not specified. Before presenting the hardness, we need the following lemma for deciding the optimal execution order on the client, provided that the computation offloading decisions have been made.

▶ **Lemma 2.** *If the execution order is not specified, all the offloaded tasks should be executed before any locally-executed task.*

Proof. Suppose that τ_i is decided to be locally executed, while task τ_j is to be offloaded. If a feasible schedule executes τ_i on the client before the next task τ_j in the schedule starts on the client, we can also swap the execution ordering of τ_i and τ_j on the client to be still feasible. Let τ_i starts to run on the client at time t at the original schedule. So, the total finishing time of the two tasks τ_i and τ_j is equal to $t + C_i + S_j + I_j \leq D$ (because the schedule is feasible). After swapping, the finishing time of τ_j is now at most $t + S_j + I_j$, the finishing time of τ_i now is at most $t + S_j + C_i$, and the total finishing time of both tasks is at most $\max\{t + S_j + I_j, t + S_j + C_i\}$. Therefore, the the total finishing time of the two tasks after swapping is less than before swapping without violating the feasibility of the schedule, because $\max\{t + S_j + I_j, t + S_j + C_i\} < t + C_i + S_j + I_j \leq D$.

After swapping, the worst-case finishing time of the other tasks does not change. By repeating the above procedure, we know that the statement in the lemma holds. ◀

When the offloading decision $\vec{x}_n$ for the tasks is known, we define $d_i = x_i(D - I_i) + (1 - x_i)D$ as the *virtual offloaded deadline*. If there is a feasible schedule based on an offloading decision $\vec{x}_n$, then executing the tasks by following the order of $d_i = x_i(D - I_i) + (1 - x_i)D$ non-decreasingly is also a feasible schedule. This ordering is called Earliest Virtual Offloaded Deadline First (**EVODF**). Please refer to Figure 4, as an illustration example for an optimal ordering for a given set of five tasks. We have the following lemma for **EVODF**.

▶ **Lemma 3.** *If the execution order is not specified and there is a feasible schedule based on the offloading decisions, the schedule by using **EVODF** is also a feasible schedule.*

Proof. Suppose that a given schedule is feasible. By Lemma 2, we can reorder the execution ordering, such that any locally-executed task should be executed after the offloaded tasks, to maintain the feasibility. Now, for two consecutively offloaded tasks τ_i and τ_j in that feasible schedule, if $d_i > d_j$ and τ_i starts its execution at time t on the client before τ_j, we can still swap these two jobs to maintain the feasibility. Suppose that the server returns the result of task τ_i at time $f_i = t + S_i + I_i$ and τ_j at time $f_j = t + S_i + S_j + I_j$, respectively. By the definition of $d_i > d_j$, we know that $I_i < I_j$.

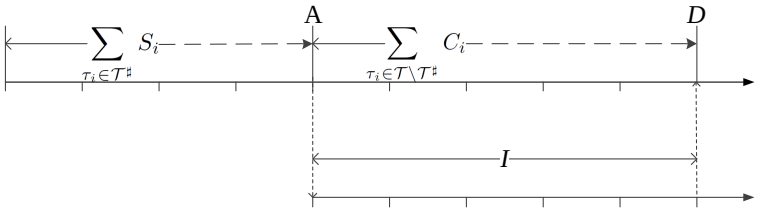


Figure 5 Illustration for the $\mathcal{NP}$-completeness proof of the SERTO problem.

Clearly, due to the feasibility before swapping, we know that $f_i = t + S_i + I_i \leq D$ and $f_j = t + S_i + S_j + I_j \leq D$. Therefore, the finishing time f'_j of τ_j after swapping is $f'_j = t + S_j + I_j \leq D$, and the finishing time of τ_i after swapping is $f'_i = t + S_j + S_i + I_i < t + S_j + S_i + I_j \leq D$. Clearly, after swapping, the worst-case finishing time of the other tasks does not change.

By repeating the above procedure, we know that the schedule by using (**EVODF**) is also a feasible schedule. ◀

Based on Lemma 3, we have the following lemma for testing whether an offloading decision results in a feasible schedule or not.

▶ **Lemma 4.** *Suppose that tasks $\tau_i \in \mathcal{T}$ for $i = 1, 2, ..., n$ are ordered non-decreasingly according to $D - I_i$. An offloading decision $\vec{x}_n$, $x_i = \{0, 1\}$, results in a feasible schedule (by using **EVODF**) if and only if (a) $\sum_{j=1}^{n} x_j S_j + (1 - x_j) C_j \leq D$ and (b) $x_k I_k + \sum_{j=1}^{k} x_j S_j \leq D, \forall k = 1, 2, \ldots, n$.*

Proof. This comes directly from Lemma 3. ◀

Now, we will prove the $\mathcal{NP}$-completeness of the SERTO problem when the execution ordering is unknown.

▶ **Theorem 5.** *The SERTO problem is $\mathcal{NP}$-complete if the execution order is not given.*

Proof. Due to Lemma 3, verifying whether an offloading decision with **EVODF** scheduling is feasible or not can be done in polynomial time. Therefore, the SERTO problem is in $\mathcal{NP}$. The $\mathcal{NP}$-completeness can be proved by a reduction from the SUBSET SUM problem [4]: Given a set of integers $\mathcal{V} = \{v_1, v_2, \ldots, v_n\}$ and an integer A, the problem is to find a subset $\mathcal{V}^\sharp$ of $\mathcal{V}$ such that $\sum_{v_i \in \mathcal{V}^\sharp} v_i = A$. For each v_i in $\mathcal{V}$, the reduction creates τ_i with

- $C_i = 2v_i$, $S_i = v_i$,
- $I_i = I = 2((\sum_{v_j \in \mathcal{V}} v_j) - A)$, and
- $D = 2\sum_{v_j \in \mathcal{V}} v_j - A$.

Since all the tasks have the same round-trip offloading time and by Lemma 4, for an offloaded task set $\mathcal{T}^\sharp$ (with the corresponding set $\mathcal{V}^\sharp$), the resulting **EVODF** schedule is feasible if and only if $\sum_{\tau_i \in \mathcal{T}^\sharp} S_i \leq D - I$ and $\sum_{\tau_i \in \mathcal{T}^\sharp} S_i + \sum_{\tau_i \in \mathcal{T} \setminus \mathcal{T}^\sharp} C_i \leq D$, see Figure 5.

By the construction of task set $\mathcal{T}$, we know that

$$\sum_{v_i \in \mathcal{V}^\sharp} v_i = \sum_{\tau_i \in \mathcal{T}^\sharp} S_i \leq D - I = A \tag{1}$$

and

$$\begin{aligned} & \sum_{\tau_i \in \mathcal{T}^\sharp} S_i + \sum_{\tau_i \in \mathcal{T} \setminus \mathcal{T}^\sharp} C_i \leq D \\ \Rightarrow \quad & 2\sum_{v_i \in \mathcal{V}} v_i - \sum_{v_i \in \mathcal{V}^\sharp} v_i \leq 2\sum_{v_j \in \mathcal{V}} v_j - A \quad \Rightarrow \quad \sum_{v_i \in \mathcal{V}^\sharp} v_i \geq A. \end{aligned} \tag{2}$$

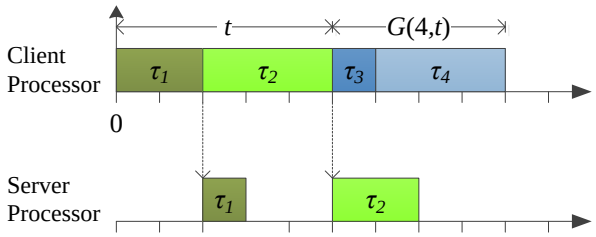


Figure 6 An example for illustrating the dynamic programming parameters.

Therefore, by (1) and (2), we know that there exists such a $\mathcal{V}^\sharp$ with $\sum_{v_i \in \mathcal{V}^\sharp} v_i = A$, if and only if the reduced input instance for the SERTO problem has a feasible schedule by offloading the corresponding task set $\mathcal{T}^\sharp$ created from $\mathcal{V}^\sharp$.

Since the reduction is in linear time complexity, we know that the SERTO problem is $\mathcal{NP}$-complete. ◀

6 Algorithms for Tasks without Specified Ordering

In this section, we consider real-time tasks without any specified ordering, and present our proposed scheduling algorithms for the SERTO problem. We will present a pseudo-polynomial-time algorithm and an approximation algorithm with polynomial-time complexity. At the end of the section, we will extend our algorithms to find the minimum D for executing the frame-based real-time tasks to maximize the performance.

6.1 Dynamic Real-time Scheduling Algorithm

Based on dynamic programming, we introduce *Dynamic Real-time Scheduling (*DRS) algorithm to find a feasible solution for the SERTO problem. At the beginning, tasks $\tau_i \in \mathcal{T}$ for $i = 1, 2, ..., n$ are ordered non-decreasingly according to $D - I_i$.

An offloading decision $\vec{x}_i$ for the first i tasks, i. e., $\{\tau_1, \tau_2, \ldots, \tau_i\}$, is said *partially feasible for offloading* (or a *partially feasible offloading decision*) if the offloaded tasks can finish the execution in the servers before the given deadline D. Similar to Lemma 4, we know that a vector $\vec{x}$ is partially feasible for offloading for $\{\tau_1, \tau_2, \ldots, \tau_i\}$ if and only if $x_k I_k + \sum_{j=1}^{k} x_j S_j \leq D, \forall k = 1, 2, \ldots, i$.

Our strategy is to build a dynamic programming table by maintaining and storing some scheduling results for the partially feasible offloading decisions for the first i tasks. Specifically, among all the partially feasible offloading decisions for $\{\tau_1, \tau_2, \ldots, \tau_i\}$, let $G(i, t)$ be the minimum total local execution time for the locally-executed tasks under the constraint that the total setup time for the offloaded tasks in $\{\tau_1, \tau_2, \ldots, \tau_i\}$ is less than or equal to t. Figure 6 presents an example of four tasks $\{\tau_1, \tau_2, \tau_3, \tau_4\}$ with the dynamic programming parameters, where $\{\tau_1, \tau_2\}$ are offloaded and $\{\tau_3, \tau_4\}$ are executed locally. That is, for a given i and t, the value $G(i, t)$ is the objective function of the following integer linear programming (ILP):

$$\text{minimize} \sum_{j=1}^{i} (1 - x_j) C_j \tag{3a}$$

$$\text{s.t.} \sum_{j=1}^{i} x_j S_j \leq t \tag{3b}$$

$$x_k I_k + \sum_{j=1}^{k} x_j S_j \leq D \qquad \forall k = 1, 2, \ldots, i \tag{3c}$$

$$x_j \in \{0, 1\} \qquad \forall j = 1, 2, \ldots, i. \tag{3d}$$

For notational brevity, when the above ILP has no feasible solution, $G(i,t)$ is defined as ∞. Moreover, $G(i,t) = \infty$ when $t < 0$.

The optimal solution for (3) is also called *optimal partially offloaded decision* when the total local setup time for the offloaded tasks in these i tasks is no more than t. Clearly, when i is 1, we know that

$$G(1,t) = \begin{cases} 0 & \text{if } S_1 \leq t \leq D - I_1 \\ C_1 & \text{otherwise} \end{cases} . \tag{4}$$

Instead of solving the above ILP for building $G(i,t)$, the construction of $G(i,t)$, for $i \geq 2$, can be achieved by using the following recurrence:

$$G(i,t) = \min \begin{cases} \begin{cases} G(i-1, t-S_i) & \text{if } t \leq D - I_i \\ \infty & \text{otherwise} \end{cases} \\ \\ G(i-1,t) + C_i \end{cases} . \tag{5}$$

Suppose that $\vec{x}^{\sharp}_{i-1}$ is the corresponding partially feasible offloading decision for $\{\tau_1, \tau_2, \ldots, \tau_{i-1}\}$ with respect to the result in $G(i-1, t-S_i)$. Similarly, let $\vec{x}^{\dagger}_{i-1}$ be the corresponding partially feasible offloading decision for $\{\tau_1, \tau_2, \ldots, \tau_{i-1}\}$ with respect to the result in $G(i-1,t)$. The recursive function in (5) represents the selection of the minimum solution by comparing two cases:

- Case 1: task τ_i is offloaded when its local setup execution finishes at time t. For such a case, if $t + I_i > D$, offloading τ_i is an infeasible offloading decision; otherwise, we consider the offloading decision $\vec{x}^{\sharp}_{i-1}$ for the first $i-1$ tasks, in which the total local execution time of this solution is as the same as that in $\vec{x}^{\sharp}_{i-1}$, i. e., $G(i-1, t-S_i)$.
- Case 2: task τ_i is locally executed. Therefore, we consider the offloading decision $\vec{x}^{\dagger}_{i-1}$ for the first $i-1$ tasks. As a result, the total local execution time of this solution is the sum of C_i and the total local execution time in solution $\vec{x}^{\dagger}_{i-1}$. That is, $C_i + G(i-1,t)$.

We assume in this subsection that S_i is a non-negative integer for a task τ_i in $\mathcal{T}$. The standard dynamic-programming procedure can be applied by constructing a table with n rows for $i = 1, 2, \ldots, n$ and $\lfloor D \rfloor + 1$ columns for $t = 0, 1, 2, \ldots, \lfloor D \rfloor$.

▶ **Lemma 6.** *For given integers i and t, the recursive function defined in (4) and (5) computes the optimal solution for $G(i,t)$.*

Proof. The optimality is proved by induction on i. For the **base case**, $G(1,t) = 0$ if there is enough time for feasible offloading of task τ_1. Otherwise τ_1 is locally executed, and then $G(1,t) = C_1$, which is optimal.

Inductive step: Assume that $G(i-1,t)$ is optimal for the subproblem for the first $i-1$ tasks with $i \geq 2$ for any given $t \geq 0$ (i. e., the ILP described in (3)). Recall that the two offloading decisions $\vec{x}^{\sharp}_{i-1}$ and $\vec{x}^{\dagger}_{i-1}$ which represent the optimal partially offloading decisions for $\{\tau_1, \tau_2, \ldots, \tau_{i-1}\}$ when the total local setup time for the offloaded tasks in these $i-1$ tasks is no more than $t - S_i$ and t, respectively.

Suppose for contradiction that $\vec{x}^{*}_{i}$ is a partially feasible offloading decision for $\{\tau_1, \tau_2, \ldots, \tau_i\}$ in which $\sum_{j=1}^{i} x^*_j S_j \leq t$ and $\sum_{j=1}^{i}(1 - x^*_j)C_j < G(i,t)$. There are two cases for task τ_i in $\vec{x}^{*}_{i}$.

- **Case 1:** x^*_i is 0 (τ_i is locally executed) in $\vec{x}^{*}_{i}$. Clearly, under the assumption $\sum_{j=1}^{i}(1-x^*_j)C_j < G(i,t)$, we know that

$$\sum_{j=1}^{i} C_j(1-x^*_j) = C_i + \sum_{j=1}^{i-1} C_j(1-x^*_j) < G(i,t) \leq_1 C_i + \sum_{j=1}^{i-1} C_j(1-x^{\dagger}_j),$$

where $\leq_1$ comes from the construction of $G(i,t)$ in (5). Hence, the offloading decision $\vec{x}^*_{i-1}$ by excluding x^*_i from $\vec{x}^*_i$ is a partially feasible offloading decision for the first $i-1$ tasks with $\sum_{j=1}^{i-1} S_j x^*_j \leq t$ and $\sum_{j=1}^{i-1} C_j(1-x^*_j) < \sum_{j=1}^{i-1} C_j(1-x^\dagger_j)$, which contradicts the optimality of $G(i-1,t)$.

- **Case 2:** x^*_i is 1 (τ_i is offloaded) in $\vec{x}^*_i$. Clearly, we know that $S_i \leq t \leq D - I_i$ for such a case; otherwise, $\vec{x}^*_i$ is not a partially feasible offloading decision. With this case, we know that

$$\sum_{j=1}^{i} C_j(1-x^*_j) = \sum_{j=1}^{i-1} C_j(1-x^*_j) < G(i,t) \leq \sum_{j=1}^{i-1} C_j(1-x^\sharp_j).$$

Therefore, the offloading decision $\vec{x}^*_{i-1}$ by excluding x^*_i from $\vec{x}^*_i$ is a partially feasible offloading decision for the first $i-1$ tasks with $\sum_{j=1}^{i-1} S_j x^*_j \leq t - S_i$ and $\sum_{j=1}^{i-1} C_j(1-x^*_j) < \sum_{j=1}^{i-1} C_j(1-x^\sharp_j)$, which contradicts the optimality of $G(i-1,t-S_i)$.

Hence, based on the induction hypothesis, the lemma is proved. ◀

Now, based on Lemma 6, for an input task set $\mathcal{T}$, to verify whether a feasible schedule exists for the SERTO problem or not, we just have to check whether there exists $0 \leq t \leq D$ with $G(n,t) + t \leq D$.

▶ **Theorem 7.** *There exists t with $G(n,t) + t \leq D$ if and only if there exists a feasible schedule for the SERTO problem.*

Proof. If: Suppose $\vec{x}_n$ is the corresponding offloading decision for a feasible schedule of the SERTO problem. Let ℓ be the maximum index with $x_\ell = 1$. That is, x_j is 0 for $j > \ell$. As the schedule is feasible, we know that $\vec{x}_n$ is also a partially feasible offloaded decision when t is set to $\sum_{j=1}^{\ell} S_j x_j$. Therefore, based on Lemma 6, we have $\sum_{j=1}^{n} C_j(1-x_j) \geq G(n, \sum_{j=1}^{\ell} S_j x_j)$. The necessary condition for being a feasible schedule for the SERTO problem, is $\sum_{j=1}^{\ell} S_j x_j + \sum_{j=1}^{n} C_j(1-x_j) \leq D$. This implies that $\sum_{j=1}^{\ell} S_j x_j + G(n, \sum_{j=1}^{\ell} S_j x_j) \leq D$. Therefore, when t is $\sum_{j=1}^{\ell} S_j x_j$, we know that $G(n,t) + t \leq D$.

Only-If: Suppose t^* is with $G(n,t^*) + t^* \leq D$. We can backtrack the dynamic programming table to obtain an offloading decision $\vec{x}^*_n$ for the given n tasks such that it satisfies the constraints described in (3) when t is set to t^* and i is set to n. Since $G(n,t^*) + t^* \leq D$, based on Lemma 4, we know that the resulting schedule by using **EVODF** scheduling policy is a feasible schedule. ◀

▶ **Theorem 8.** *The DRS algorithm is an optimal offloading scheduling algorithm with time complexity $O(n \log n + nD)$ for the SERTO problem when there is no specified execution ordering.*

Proof. The optimality comes from Theorem 7. The time complexity of constructing $G(i,t)$ for $i = 1, 2, \ldots, n$ and $t = 0, 1, 2, \ldots, \lfloor D \rfloor$ is $O(n \log n + nD)$, since it takes $O(n \log n)$ to sort task set $\mathcal{T}$ and $O(nD)$ to build the dynamic-programming table. For back-tracking a solution from an entry t with $G(n,t^*) + t^* \leq D$, the time complexity is $O(n)$. Therefore, the overall time complexity is $O(n \log n + nD)$, which is pseudo-polynomial time. The space complexity is $O(nD)$, but it can be improved to $O(D)$ by discarding the entries $G(i-2,t)$ when building $G(i,t)$. ◀

6.2 Approximation for DRS Algorithm

As the SERTO problem is $\mathcal{NP}$-complete, solving the problem in polynomial time is not possible unless $\mathcal{NP} = \mathcal{P}$. To allow a polynomial-time algorithm, some approximation is needed. This subsection presents a methodology to reduce the time complexity so that the user can trade the complexity with the approximation of the derived solution.

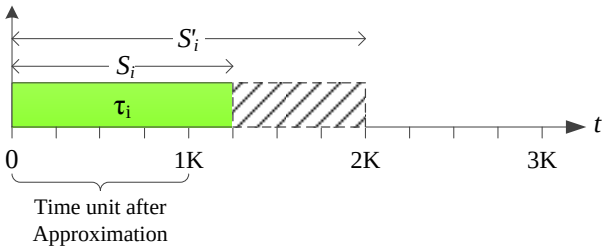


Figure 7 Approximation example.

Let $K = \frac{\epsilon D}{n}$ be the time unit after approximation, where $\epsilon > 0$ is a user-specified parameter that determines the approximation granularity. This means that a time unit after approximation is equal to K amount of time before approximation. The exact algorithm requires the assumption that all the timing parameters are integers and has pseudo-polynomial complexity. However, if the timing parameters are real numbers, the algorithm will not work. In this case, the real numbers can be rounded up to the nearest integers. But, this will affect the accuracy of the algorithm. Also, in the case of a large value of D, the time and space complexities of the algorithm will be high. Therefore, the approximation is used to trade the accuracy with time and space complexities for both cases, depending on the user parameter ϵ. Both complexity and accuracy are inversely proportional to the value of ϵ, which determines the value of K. If the value of K is less than 1, the timing parameters are scaled up to increase the accuracy. But, it will also increase the complexity of the algorithm. On the other hand, if the value of K is greater than 1, the timing parameters are scaled down which is used to reduce the complexity of the algorithm for a large value of D. As a consequence, we will get a less accurate result. For $K = 1$, the approximation does not have any effect.

For each task τ_i, we construct a corresponding task τ'_i as follows:

- $S'_i = K \left\lceil \frac{S_i}{K} \right\rceil$ (rounded up to the nearest time unit, i. e. integer multiples of K),
- $I'_i = I_i - (S'_i - S_i)$, and
- $C'_i = C_i$.

Figure 7 shows an approximation example, where the time unit after approximation (K) is equal to 4 and the setup time S_i is rounded-up to the next time unit ($2K$).

Let $\mathcal{T}'$ be the resulting task set after transformation. Moreover, we also set D' either to D or to $(1+\epsilon)D$, to be explained later. As all the setup times are integer multiples of K, we can construct the dynamic programming table by considering only the integer multiples of K. Therefore, we define $G'(i,t)$ as the minimum total local execution time for the locally executed tasks under the constraint that the total *rounded-up* setup time for the offloaded tasks in $\{\tau'_1, \tau'_2, \ldots, \tau'_i\}$ is less than or equal to $t \cdot K$

$$G'(1,t) = \begin{cases} 0 & \frac{S'_1}{K} \leq t \leq \frac{D'-I'_i}{K} \\ C'_1 & otherwise \end{cases} . \tag{6}$$

For $i \geq 2$,

$$G'(i,t) = min \begin{cases} \begin{cases} G'(i-1, t-\frac{S'_i}{K}) & \forall t \leq \frac{D'-I'_i}{K} \\ \infty & otherwise \end{cases} \\ \\ G'(i-1,t) + C'_i \end{cases} . \tag{7}$$

The time t in the equations above represents the time after approximation, which is represented in K unit of time. Therefore, the timing parameters S'_i and $D' - I'_i$ should be divided by K to be consistent with the new time scale.

▶ **Lemma 9.** *For given integers i and t, the recursive function defined in (6) and (7) computes the optimal solution for a partially feasible offloading decision for the first i tasks in $\mathcal{T}'$.*

Proof. This is similar to the proof for Lemma 6 ◀

The following theorem shows the feasibility by adopting the dynamic programming for the resulting solutions.

▶ **Theorem 10.** *When D' is set to D, and there exists t with $0 \leq t \leq \left\lfloor \frac{D'}{K} \right\rfloor$ and $G'(n,t)+t\cdot K \leq D'$, the offloading decision by backtracking the dynamic programming table built for $\mathcal{T}'$ is a feasible schedule of the original task set $\mathcal{T}$ by using* **EVODF** *scheduling policy.*

Proof. This basically comes directly from the definition of $\mathcal{T}'$. Suppose that $\vec{x}_n$ is an offloading decision for such a t after by backtracking the dynamic programming table built for $\mathcal{T}'$. Therefore, with the fact $\sum_{j=1}^{n} x_j S'_j \leq t \cdot K$, we know that

$$t \cdot K \geq \sum_{j=1}^{n} x_j S'_j \geq \sum_{j=1}^{n} x_j S_j,$$

and, for all $k = 1, 2, \ldots, n$, as $I'_k + S'_k$ is equal to $I_k + S_k$ and $x_k I'_k + \sum_{j=1}^{k} x_j S'_j \leq D$, we have

$$D \geq x_k(I_k + S_k) + \sum_{j=1}^{k-1} x_j S'_j \geq x_k I_k + \sum_{j=1}^{k} x_j S_j.$$

Therefore, we know that $\vec{x}_n$ is a partially feasible offloading decision with $\sum_{j=1}^{n} x_j S_j \leq t \cdot K$. Since $G'(n,t) + t \cdot K \leq D'$, the statement holds due to Lemma 4. ◀

▶ **Theorem 11.** *If there exists a feasible schedule for $\mathcal{T}$, then there exists t with $0 \leq t \leq \left\lfloor \frac{D'}{K} \right\rfloor$ and $G'(n,t) + t \cdot K \leq D'$ when D' is set to $(1+\epsilon)D$.*

Proof. Suppose that $\vec{x}^*_n$ is the offloading decision of a feasible schedule for $\mathcal{T}$. By Lemma 4, $\sum_{j=1}^{n} x^*_j S_j + (1 - x^*_j) C_j \leq D$ and $x^*_k I_k + \sum_{j=1}^{k} x^*_j S_j \leq D$ for $k = 1, 2, \ldots, n$.

By the definition of $\mathcal{T}'$ and $K = \frac{\epsilon D}{n}$, we know that

$$\sum_{j=1}^{n} x^*_j S'_j \leq \sum_{j=1}^{n} x^*_j (S_j + K) \leq K \cdot n + \sum_{j=1}^{n} x^*_j S_j \leq \epsilon D + \sum_{j=1}^{n} x^*_j S_j. \tag{8}$$

Similarly, for $k = 1, 2, \ldots, n$, we have

$$x^*_k I_k + \sum_{j=1}^{k} x^*_j S_j \leq x^*_k (I_k + S_k) + \sum_{j=1}^{k-1} x^*_j S'_j \leq x^*_k (I_k + S_k) + \epsilon D + \sum_{j=1}^{k-1} x^*_j S_j \leq (1+\epsilon) D.$$

Let t' be $\frac{\sum_{j=1}^{n} x^*_j S'_j}{K}$, in which t' is an integer and $t' \leq \left\lfloor \frac{D'}{K} \right\rfloor$. Then, by Lemma 9 for the optimality in $G'(i,t')$, we know that $G'(n,t') \leq \sum_{j=1}^{n} (1 - x^*_j) C_j$. Therefore, together with (8),

we know that

$$G'(n,t') + t' \cdot K = G'\left(n, \frac{\sum_{j=1}^{n} x_j^* S_j'}{K}\right) + \sum_{j=1}^{n} x_j^* S_j'$$
$$\leq \sum_{j=1}^{n}(1-x_j^*)C_j + \sum_{j=1}^{n} x_j^* S_j + \epsilon D \leq (1+\epsilon)D = D',$$

which proves the theorem. ◀

We now analyze the time complexity.

▶ **Theorem 12.** *For a given $\epsilon > 0$ and D', evaluating whether there exists exists t with $0 \leq t \leq \left\lfloor \frac{D'}{K} \right\rfloor$ and $G'(n,t) + t \cdot K \leq D'$ is with time complexity $O(\frac{n^2}{\epsilon})$.*

Proof. The construction of task set $\mathcal{T}'$ takes only $O(n)$. The construction of $G'(i,t)$ requires $O(n\frac{D}{K}) = O(\frac{n^2}{\epsilon})$, since K is set to $\frac{\epsilon D}{n}$. ◀

6.3 Maximizing the Sampling Rate

The SERTO problem so far is for determining a feasible schedule if there exists. Another extension is to minimize the deadline/period D for the frame-based real-time tasks so that the sampling rate of the frame-based tasks can be maximized. The DRS algorithm can be adopted to find the optimal value of D with a binary search. Suppose that D^{lower} and D^{upper} are the lower and upper bounds of the feasible deadlines in the current iteration in the binary search, respectively. Initially, D^{upper} is $\sum_{i=1}^{n} C_i$ and D^{lower} is $\sum_{i=1}^{n} \min\{S_i, C_i\}$.

Moreover, suppose that $\vec{x}_n$ is the offloading decision for a feasible schedule by setting D to $\frac{D^{lower}+D^{upper}}{2}$. We also know that setting D to

$$D^\sharp = \max\left\{ \begin{array}{l} \max_{1\leq k\leq n}\{x_k I_k + \sum_{j=1}^{k} x_j S_j\}, \\ \sum_{j=1}^{n}(1-x_j)C_j + x_j S_j \end{array} \right\}$$

is also feasible. Therefore, if such an offloading decision $\vec{x}_n$ is found, the efficiency, with respect to the time complexity, of the binary search can be further improved by setting the next D to $\frac{D^{lower}+D^\sharp}{2}$, as any $D > D^\sharp$ has feasible schedules. Clearly, the whole procedure is still with pseudo-polynomial time.

When the approximation in Section 6.2 is adopted, the above binary search still works with polynomial-time complexity. Due to Theorems 10 and 11, the derived solution is at most $(1+\epsilon)$ times the minimum feasible deadline of the input instance, by ignoring the error due to the termination condition of the binary search.

7 Experimental Results

In our experiments, our *DRS* algorithm, with and without approximation, is evaluated by adopting a surveillance system as a case study and synthesis workload simulation.

7.1 Case Study of a Surveillance System

We use a surveillance system that performs four real-time tasks to evaluate our *DRS* algorithm, and compare it with Nimmagadda et al. [12] algorithm and by offloading all the tasks. The system captures two images at the same time, left and right, periodically. Left and right images are used for a stereo vision task. For the other tasks, one image is used for processing. The tasks are frame-based real-time tasks and independent, described as follows:

Table 1 Timing parameters of case study tasks (ms).

τ_i	Description	C_i	With encoding			Without encoding		
			S_i	I_i - Multi.	I_i - Single	S_i	I_i - Multi.	I_i - Single
τ_1	Motion Detection	30	31	33	117	7	21	141
τ_2	Object Recognition	220	3	102	102	2	102	102
τ_3	Stereo Vision	88	34	47	115	16	41	127
τ_4	Motion Recording	18	31	29	115	7	14	148

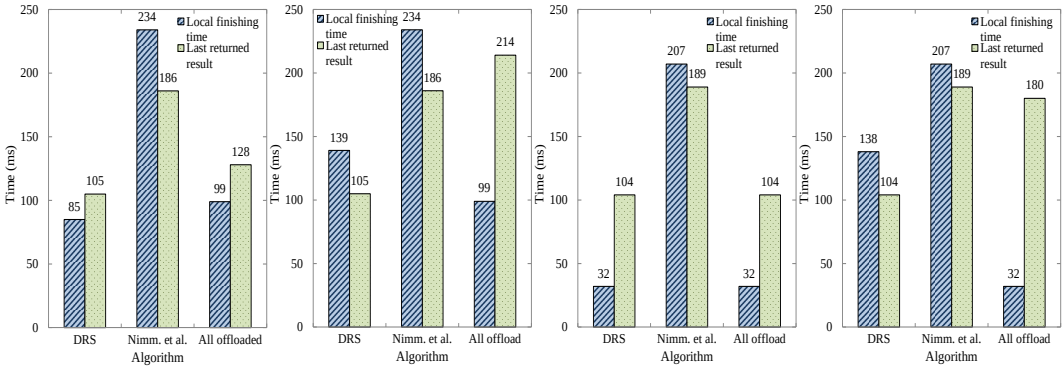
- **Motion Detection:** The motion is detected using the background subtraction technique [13]. The system computes the running average of the captured frames, and each new frame is subtracted from the moving average. Then, the output is processed to get the contours of the moving objects [5].
- **Object Recognition:** It is used to recognize a given object from the input image. Scale Invariant Feature Transform (SIFT) method [11] is used to extract features, which are not affected by object size, position or rotation.
- **Stereo Vision:** Stereo vision is used to generate a depth map for left and right images to calculate the distance between the surveillance system and the object of interest. Stereo imaging [1] is adopted in the implementation.
- **Motion Recording:** It records video for detected motion for further human check.

The system remains idle until a motion is detected. Then, it starts executing all the tasks above. Before sending an image to the server(s), scaling, encoding, or both of them may be performed on the image. Although scaling and encoding can reduce the size of the transfered image for reducing the communication overhead, they consume more time on the local device for scaling and encoding. The time used for scaling, encoding, and sending on the client for a task is considered as the setup time in our case study. For the server side, we consider two cases. First, a dedicated server (or processor) for each task, if offloaded. Second, we assume that we have only one server where a scheduling algorithm is used to schedule all the offloaded tasks, in which I_i may be larger than C_i for some task τ_i.

We consider four scenarios: (*Scenario 1*) images are encoded before sending and a dedicated server is used for each offloaded task (multiple servers), (*Scenario 2*) images are encoded before sending and only one server is used for all the offloaded tasks (single server), (*Scenario 3*) images are sent without encoding and a dedicated server is used for each offloaded task, and (*Scenario 4*) images are sent without encoding and only one server is used for all the offloaded tasks.

Timing parameters for the tasks in the four scenarios are given in Table 1, where the time values are in milliseconds based on measurements. If the system performs all the tasks locally, they will finish by 356 ms, which will be considered as the deadline (sampling period) in our case study here. We explore the three offloading approaches to reduce the local finishing time, i. e., increase the sampling rate.

Figure 8 shows the total local finishing time on the client side, and the time at which the last result returns back from the server side. In Scenario 1, tasks τ_2 and τ_3 are offloaded in both *DRS* and Nimmagadda et al. [12] algorithms. Although the offloading decisions in the previous scenario are the same for both algorithms, the total finishing time in *DRS* is shorter. This is because *DRS* algorithm continues local execution after offloading, while Nimmagadda et al. [12] remains idle waiting for the results from the server side. The decision of the Nimmagadda et al. [12] algorithm in Scenario 2 is the same as in Scenario 1. It does not change by having multiple servers because Nimmagadda et al. [12] algorithm remains idle during offloading. *DRS* algorithm just offloads task τ_2 in Scenario 2 and 4.



(a) Scenario 1 **(b)** Scenario 2 **(c)** Scenario 3 **(d)** Scenario 4

Figure 8 Case study results.

In Scenarios 3, *DRS* algorithm offloads all the tasks because their setup time S_i is less than their local execution time C_i, and they are feasible for offloading. Nimmagadda et al. [12] algorithm offloads all the tasks except task τ_4 in Scenario 3 and 4, because its local time is less than the summation of the expected remote execution time and the data transfer time. We observe that all the three evaluated algorithms reduce the local finishing time, but our algorithm has the minimum finishing time in all scenarios.

7.2 Simulation Setup and Results

We also perform simulations by using synthetic workload for task τ_i generated as follows:

- C_i: Randomly generated integer values from 1 to 50 ms with uniform distribution.
- S_i: Randomly generated integer values from 1 to C_i with uniform distribution.
- I_i: $I_i = \frac{C_i}{\alpha}$, where α is the speed-up factor of the server, i. e., the response time from the server is α times faster than the execution time of the local client. α is randomly generated such that $0 < \alpha \leq m$, where m is the maximum value of α.

We perform 100 rounds in the experiment. In each round, a set of 25 frame-based real-time tasks is randomly generated according to the above conditions. Each task set is evaluated by ten different settings according to m values, where $m = \{0.005, 0.025, 0.05, 0.1, 0.25, 0.5, 1, 2, 4, 8\}$. By using small m values, we simulate servers that require longer response time than the local execution time for tasks. While by using large m values, we simulate servers that are faster than the client and can process almost immediately for any offloaded tasks.

The *normalized finishing time reduction* of an algorithm for a task set is the finishing time for the task set execution after using the derived schedule divided by the finishing time for the same task set if all tasks are executed locally. Also, the *normalized sampling period* is the finishing time for the input task set using the approximation *DRS* algorithm described in 6.2 divided by the finishing time for the same task set using the *DRS* algorithm.

For the rest of this section, we will discuss the simulation results for the three offloading approaches using the task sets described above. Also, we evaluate the approximation *DRS* algorithm described in Section 6.2. Figure 9 shows the number of offloaded tasks for different m values. Nimmagadda et al. [12] algorithm offloads tasks only when the server is faster than the client. But, *DRS* algorithm offloads tasks even to server(s) with longer response time, while they are feasible. Also, we observe that the number of the offloaded tasks increases proportionally to m value.

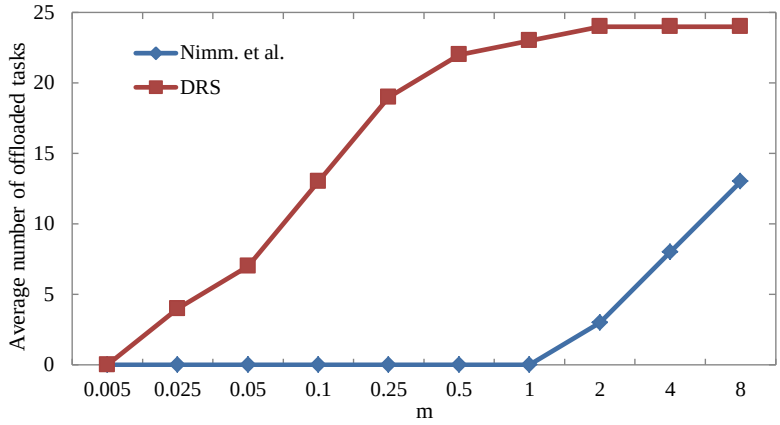


Figure 9 Number of offloaded tasks for synthesized tasks.

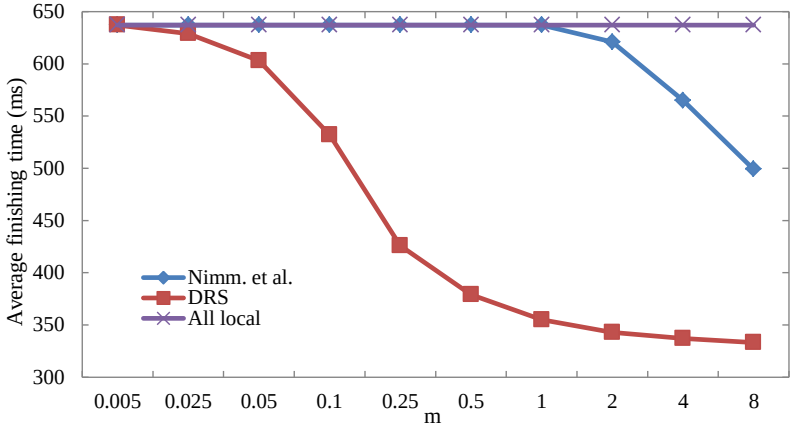


Figure 10 Finishing time for synthesized tasks.

Figure 10 illustrates the average finishing time for the generated task sets. Nimmagadda et al. [12] algorithm reduces the local execution time only when the server is faster than the client, because it doesn't offload tasks with $m \leq 1$ as shown in Figure 9. The improvement of *DRS* algorithm, compared to Nimmagadda et al. [12] algorithm, is up to 44.7 %.

Figure 11 shows the average normalized finishing time reduction. Again, Nimmagadda et al. [12] algorithm does not help in finishing time reduction for the same reason in Figures 9 and 10. Furthermore, the finishing time reduction in *DRS* algorithm is more than in Nimmagadda et al. [12] algorithm because Nimmagadda et al. [12] algorithm remains idle during offloading. In Figure 11, offloading all the tasks is not useful for $m \leq 2$ and the finishing time exceeds the summation of local execution for all the tasks, because the round-trip offloading time for most of the tasks is relatively large. *DRS* algorithm reduces the finishing time in all cases because it offloads only the beneficial and optimal tasks for offloading. The average finishing time using *DRS* algorithm is reduced up to 52 % of the local execution.

Figure 12 shows the average execution time of DRS algorithm, where n is the number of input tasks. The algorithm is evaluated with different number of input tasks (5, 10, 15, 20 and 25) and different deadlines (300, 400, 500, 600 and 700 ms). As the deadline value increases, the average execution time also increases, but more rapidly for larger number of tasks. Nevertheless, the execution time of the algorithm is very short and negligible relative to the deadline.

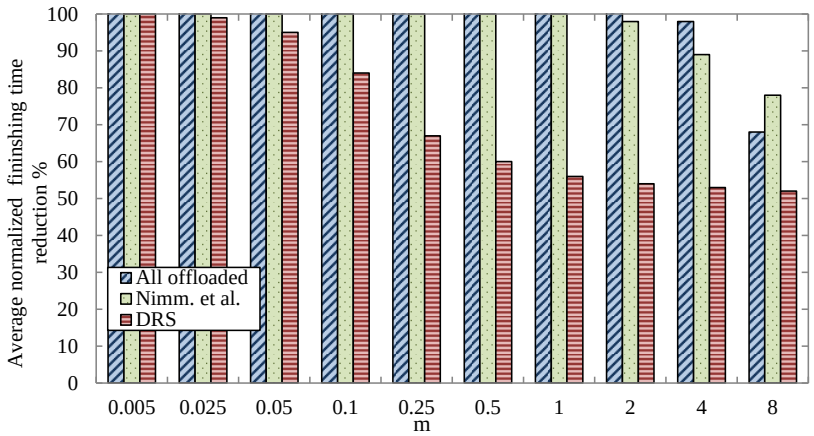


Figure 11 Finishing time reduction for synthesized tasks.

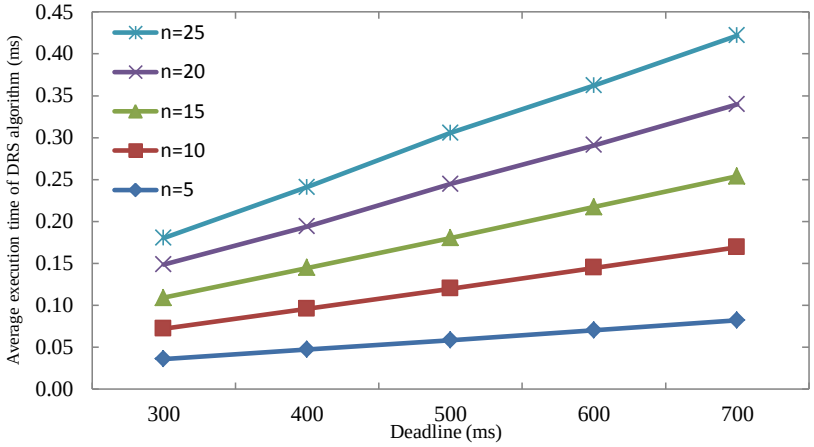


Figure 12 Execution time of DRS algorithm.

The above results are based on the *DRS* algorithm. Now, we will present the results based on the approximation in Section 6.2. Figure 13 shows the effect of the approximation parameter ϵ on the finishing time of approximation *DRS* algorithm for different m values. As the m value increases, which also implies an increase in the number of offloaded tasks, the average normalized sampling period also increases, because the offloading decision is affected by the rounded-up setup time for the offloaded tasks. For $m \geq 0.5$, the average normalized sampling period is nearly the same because almost all of the tasks are offloaded in this case. Clearly, the ϵ value affects the accuracy of the approximation for *DRS* algorithm. When the value ϵ increases for worse approximation, the finishing time of the tasks also usually, but not always, increases.

8 Conclusion

In this paper, we present two offloading algorithms, *GMF* and *DRS*, for real-time embedded systems. Our algorithms can be used to schedule tasks with and without specified execution order to meet the deadline. Also, they can be used to maximize the sampling rate for tasks execution. Our experimental results show that, even by offloading to server(s) with shorter response time, using *DRS* algorithm can result in significant finishing time reduction. The experiments also

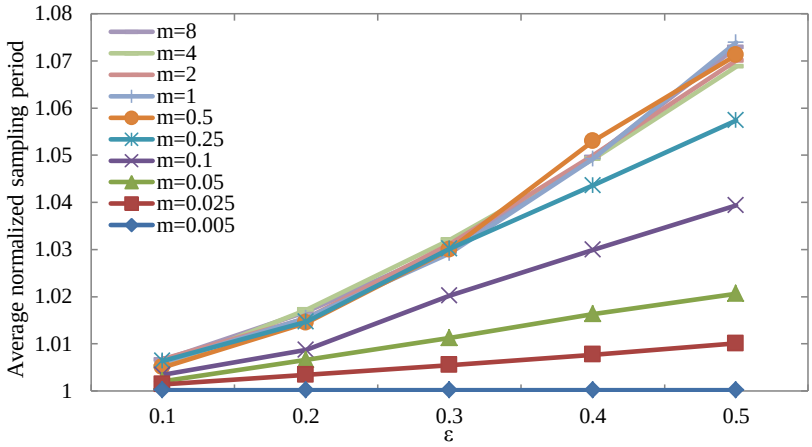


Figure 13 Normalized sampling period for synthesized tasks.

reveal that *DRS* algorithm reduces the finishing time up to 52 % of the total local execution time, and improves the finishing time of other existing offloading algorithms up to 44.7 %.

References

1 Gary R. Bradski and Adrian Kaehler. *Learning OpenCV – computer vision with the OpenCV library: software that sees.* O'Reilly, 2008. URL: `http://www.oreilly.de/catalog/9780596516130/index.html`.

2 Giorgio C. Buttazzo. *Hard Real-time Computing Systems.* Springer US, 2011. URL: `http://www.springer.com/978-1-4614-0675-4`.

3 Luis Lino Ferreira, Guilherme D. Silva, and Luís Miguel Pinho. Service offloading in adaptive real-time systems. In Zoubir Mammeri, editor, *IEEE 16th Conf. on Emerging Technologies & Factory Automation (ETFA'11), Toulouse, France, September 5–9, 2011*, pages 1–6. IEEE, 2011. `doi:10.1109/ETFA.2011.6059236`.

4 M. R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman, 1979.

5 Rafael C. Gonzalez and Richard E. Woods. *Digital Image Processing (3rd Edition).* Prentice-Hall, Inc., NJ, USA, 2008. URL: `http://www.imageprocessingplace.com/`.

6 Yu-Ju Hong, Karthik Kumar, and Yung-Hsiang Lu. Energy efficient content-based image retrieval for mobile systems. In *Int'l Symp. on Circuits and Systems (ISCAS'09), 24–17 May 2009, Taipei, Taiwan*, pages 1673–1676. IEEE, 2009. `doi:10.1109/ISCAS.2009.5118095`.

7 IFR International Federation of Robotics. Service Robot Statistics, September 2011. URL: `http://www.ifr.org/service-robots/statistics/`.

8 Dejan Kovachev, Tian Yu, and Ralf Klamma. Adaptive computation offloading from mobile devices into the cloud. In *10th IEEE Int'l Symp. on Parallel and Distributed Processing with Applications (ISPA'12), Leganes, Madrid, Spain, July 10–13, 2012*, pages 784–791. IEEE, 2012. `doi:10.1109/ISPA.2012.115`.

9 Zhiyuan Li, Cheng Wang, and Rong Xu. Computation offloading to save energy on handheld devices: a partition scheme. In *2001 Int'l Conf. on Compilers, Architecture, and Synthesis for Embedded Systems (CASES'01)*, pages 238–246, 2001. URL: `http://portal.acm.org/citation.cfm?id=502217.502257`.

10 Zhiyuan Li, Cheng Wang, and Rong Xu. Task allocation for distributed multimedia processing on wirelessly networked handheld devices. In *16th Int'l Parallel and Distributed Processing Symp. (IPDPS'02), 15–19 April 2002, Fort Lauderdale, FL, USA, CD-ROM/Abstracts Proceedings.* IEEE Computer Society, 2002. `doi:10.1109/IPDPS.2002.1015589`.

11 David G. Lowe. Object recognition from local scale-invariant features. In *Int'l Conf. on Computer Vision (ICCV'99), Vol. 2*, pages 1150–1157, 1999. URL: `http://dl.acm.org/citation.cfm?id=850924.851523`.

12 Yamini Nimmagadda, Karthik Kumar, Yung-Hsiang Lu, and C. S. George Lee. Real-time moving object recognition and tracking using computation offloading. In *2010 IEEE/RSJ Int'l Conf. on Intelligent Robots and Systems (IROS'10), October 18–22, 2010, Taipei, Taiwan*, pages 2449–2455. IEEE, 2010. `doi:10.1109/IROS.2010.5650303`.

13 Massimo Piccardi. Background subtraction techniques: a review. In *2004 IEEE Int'l Conf. on Systems, Man & Cybernetics (ICSMC'04), The Hague, Netherlands, 10–13 October 2004*, pages 3099–3104. IEEE, 2004. `doi:10.1109/ICSMC.2004.1400815`.

14 Marco Spuri and Giorgio C. Buttazzo. Efficient aperiodic service under earliest deadline scheduling. In *15th IEEE Real-Time Systems Symp. (RTSS'94), San Juan, Puerto Rico, December 7–*

9, 1994, pages 2–11. IEEE Computer Society, 1994. doi:10.1109/REAL.1994.342735.

15 Marco Spuri and Giorgio C. Buttazzo. Scheduling aperiodic tasks in dynamic priority systems. *Real-Time Systems*, 10(2):179–210, 1996. doi:10.1007/BF00360340.

16 Richard Wolski, Selim Gurun, Chandra Krintz, and Daniel Nurmi. Using bandwidth data to make computation offloading decisions. In *22nd IEEE Int'l Symp. on Parallel and Distributed Processing (IPDPS'08), Miami, Florida USA, April 14–18, 2008*, pages 1–8. IEEE, 2008. doi:10.1109/IPDPS.2008.4536215.

17 Changjiu Xian, Yung-Hsiang Lu, and Zhiyuan Li. Adaptive computation offloading for energy conservation on battery-powered systems. In *13th Int'l Conf. on Parallel and Distributed Systems (ICPADS'07), December 5–7, 2007, Hsinchu, Taiwan*, pages 1–8. IEEE Computer Society, 2007. doi:10.1109/ICPADS.2007.4447724.