# Utility-Based Scheduling for Periodic Tasks with Multiple Parallelization Options

Dawei Li and Jie Wu
Department of Computer and Information Sciences
Temple University
Philadelphia, USA
{dawei.li, jiewu}@temple.edu

***Abstract*—Modern cloud computing systems have been using multiple processing units on servers to increase their processing capability. Recently, applications with multiple parallelization options have been witnessed, and serve as a promising model for efficiently utilizing the processing capacity of the system. In this paper, we consider utility-based scheduling for periodic multi-segment tasks with multiple parallelization options on platforms with multiple homogeneous processing units. Our goal is to maximize the system's overall utility achieved by scheduling the tasks. We show that the problem is closely related to another problem, which minimizes the density of each task separately. We consider two typical types of utility models, namely, a uniform utility model, where all tasks have equal utility, and a general utility model, where all tasks have different utility values. For the uniform utility model, we give the optimal solution for selecting and scheduling tasks. For the general utility model, we prove that the problem can be reduced to the classic 0-1 knapsack problem, and thus is NP-complete; we then provide the Fully Polynomial Time Approximation Scheme (FPTAS) for the problem. FPTAS algorithms are known for high time complexity, especially if we want to achieve near-optimal solutions. We then provide a simple 1/2 approximation algorithm based on a greedy strategy with significantly reduced time complexity. Simulations show that tasks with multiple parallelization options can improve system utility significantly; comparisons show that the 1/2 approximation algorithm can achieve near-optimal solutions under general conditions.**

***Index Terms*—Multi-core processors, utility-based scheduling, periodic tasks, parallel processing, multiple parallelization options.**

## I. Introduction

Recently, cloud computing has become an important service and business model. General users or small- or medium-sized companies can submit their application requests to cloud service providers, such as Microsoft Azure, Amazon Web Services, Google Cloud Platform, etc. By providing cloud services, for example, in the form of Software-as-a-Service (SaaS), cloud service providers receive a certain amount of payment, benefits, or utilities from the service requesters. In the rest of the paper, we will consistently use the word *utility* to describe what the cloud service provider gains by providing services. From the cloud service providers' point of view, application scheduling and resource provisioning can be optimized to maximize the utility gained, as long as the service level agreements with their customers are met.

To meet the increasing computation requirements of cloud applications, modern cloud computing systems have been using multiprocessors and/or multi-core processors to increase their computational capabilities. In this paper, we do not consider the hardware differences between multi-core processors and multiprocessors; what we are interested in is the total number of general purpose *processing units* in the system. We define a processing unit to be a logical thread core with a processing capability of 1.

Nowadays, the number of processing units on a single computer is common to go up to 8, 16, or higher. Applications that want to efficiently utilize these processing units evolve from single-threaded tasks to multi-threaded tasks. The model is further generalized into a general "multi-segment task model" in [1], [2], and [3], where each task consists of a sequence of segments, where each segment has multiple threads.

The aforementioned models assume that each task or each segment of the task has a fixed number of parallel threads. We call the number of parallel threads of each task or each segment of the task the *parallelization degree*. It is intuitive that if the number of parallel threads of each segment can be dynamically adjusted, the tasks can benefit more from a system with multiple processing units. This task model is called malleable, resizable, or elastic tasks [4]–[6]; we refer to this task model as tasks with multiple parallelization options. Authors in [5] develop a software framework, ReSHAPE for supporting dynamic resizing parallel tasks during runtime; experiments in the paper also show that dynamic resizing parallel tasks has great potential to improve system utilization and to reduce application completion time. Authors in [7] formally present the periodic multi-segment resizable task model, where each segment of the task has multiple options to chose a parallelization degree from.

Applications that can be modeled as tasks with multiple parallelization options provide profitable optimization spaces for cloud service providers. In this paper, we consider scheduling such tasks from the cloud service providers' point of view. Our goal is to maximize the overall system utility gained by selecting and scheduling a subset of (many presented) tasks.

### A. Motivational Example

In this subsection, we present a simple example to motivate our work in this paper. Assume that there are three single-segment tasks, $\tau_1$, $\tau_2$, and $\tau_3$. Each task has three parallelization options. All tasks are released at time 0 and has a deadline of 5. Their utilities, parallelization options, and corresponding

TABLE I
MOTIVATIONAL EXAMPLE

| Task | Utility | Options | NO. of Threads | Execution Time | |
|---|---|---|---|---|---|
| | | | | Single Thread | Total |
| $\tau_1$ | 8 | I | 1 | 7 | 7 |
| | | II | 2 | 4 | 8 |
| | | III | 3 | 3 | 9 |
| $\tau_2$ | 3 | I | 1 | 6 | 6 |
| | | II | 2 | 3.5 | 7 |
| | | III | 4 | 2 | 8 |
| $\tau_3$ | 4 | I | 1 | 6 | 6 |
| | | II | 3 | 3 | 9 |
| | | III | 4 | 2.5 | 10 |

execution times are provided in Table I. Assume that we have 3 processing units on the platform.

Since all tasks have a single-thread execution time greater than 5, if they are not partitioned and parallelized, none of them can meet their deadlines. In this simple example, we can see that the utility of $\tau_1$ is greater than the sum of $\tau_2$ and $\tau_3$'s utilities; since we want to maximize the total utility of the system, if $\tau_1$ is schedulable on the platform, we need to select $\tau_1$. With a quick observation, we can see that $\tau_1$ is not schedulable if it chooses the first parallelization option, i.e., using one thread for the task, because the execution time of using just one thread is 7, greater than its deadline. We can also tell that if we choose either the second or the third parallelization option for $\tau_1$, it is schedulable.

If we choose the third parallelization option for $\tau_1$, we will need to use 9 units of execution time for $\tau_1$. Three processing units from time 0 to 5 can provide a maximum of 15 units of processing time. Thus, we have $15 - 9 = 6$ units of processing time remaining. For the same reason as $\tau_1$, neither $\tau_2$ nor $\tau_3$ can meet their deadlines if they choose their first parallelization options. However, if either $\tau_2$ or $\tau_3$ needs to use its second or third parallelization options, its execution requirement will be greater than or equal to 7; however, we only have 6 remaining. Thus, if we choose the third parallelization option for $\tau_1$, we cannot schedule either $\tau_2$ or $\tau_3$. The maximum utility we can gain is 8.

If we choose the second parallelization option for $\tau_1$, we will need to use 8 units of execution time for $\tau_1$. This leaves us 7 units of processing time remaining. 7 units of execution time can only satisfy the requirement of $\tau_2$'s second parallelization option. Thus, we are able to also schedule $\tau_2$. The overall utility achieved is $8{+}3 = 11$. Fig.1 shows an example scheduling for $\tau_1$ and $\tau_2$ with their second parallelization options, which is the best task selecting and scheduling strategy for this example.

We can see that, even in this simple example where each task has only one segment, how to choose the best parallelization option for each task is not straightforward. When each task has multiple segments, choosing the parallelization option for each segment is even more challenging. Also, in the simplest form, where each task has a fixed execution requirement and utility, the problem reduces to the classic 0-1 Knapsack problem, which is NP-complete. In this paper, we provide systematical approach to attack the general version of the problem.

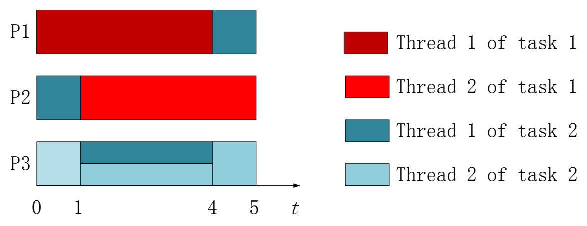


Fig. 1. An example scheduling for $\tau_1$ and $\tau_2$

## *B. Main Contributions*

In this paper, we consider selecting and scheduling a set of multi-segment tasks with multiple parallelization options, with the goal being to maximize the total utility of selected tasks. Our main contributions are as follows:

- First, we formulate the utility-based scheduling problem for multi-segment tasks with multiple parallelization options. The problem is of important practical meaning in modern cloud computing business models.
- Second, we provide two utility-model for the tasks, i.e., the uniform utility model, where all tasks have the same utility value, and the general utility model, where tasks may have arbitrarily different utility values. For the uniform utility model, we derive optimal solutions for the problem.
- Third, for the problem under the general utility model, we prove that the problem can be reduced to the classic 0-1 Knapsack problem and thus is NP-complete. We propose an FPTAS algorithm for the problem with $(1-\epsilon)$ approximation ratio. We also provide a 1/2 approximation algorithm for the problem with a significantly reduced time complexity.

## *C. Paper Organization*

The organization of this paper is as follows. Section II presents the system model and problem definition. Section III presents some fundamental results about scheduling multi-segment tasks with multiple parallelization options. In Section IV, we consider the problem under the uniform utility model, and provide optimal solutions for the problem. In Section V, we consider the problem under the general utility model. Simulations are conducted in Section VI. Conclusions are made in Section VII.

## II. SYSTEM MODEL AND PROBLEM DEFINITION

### *A. Platform and Task Models*

We consider a system with $m$ homogeneous processing units. Each processing unit can execute multiple parallel threads in a fine-grained time-shard fashion, as long as the total execution requirement of the parallel threads is less than or equal to 1. We consider a set of $n$ periodic tasks:

$$\tau = \{\tau_1, \tau_2, \cdots, \tau_n\}.$$

Each task $\tau_i$ is a sporadic task repeatedly arriving with a inter-release time $T_i$. We call each arrival of $\tau_i$ an *instance* of $\tau_i$. Each instance of $\tau_i$ has a relative deadline $D_i = T_i$, meaning that it should be completed before the next instance of $\tau_i$. Throughout this paper, we schedule each instance of a task repeatedly in the same way; our focus is the density of a task, which is also the density of each instance of the task; thus, we use the task and each instance of the task interchangeably. We assume that the total requirement of the given task set is usually greater than the system's total capacity. Thus, we need to select a subset of tasks to schedule from the given task set.

Each task $\tau_i$ has multiple sequential segments. We denote the $j$th segment of $\tau_i$ as $\tau_{i,j}$, where $1 \leq j \leq K_i$. $K_i$ is the total number of segments that $\tau_i$ has. Each segment of a task has multiple parallelization options that we can choose from. Let $P_{i,j}$ be the number of parallelization options that $\tau_{i,j}$ has. For the $p$th parallelization option, $1 \leq p \leq P_{i,j}$, the segment can spawn into $N^p_{i,j}$ threads and execute on $N^p_{i,j}$ cores in parallel. Without loss of generality, we assume that

$$N^1_{i,j} < N^2_{i,j} < \cdots < N^{P_{i,j}}_{i,j}.$$

This task model is quite common in large-scale scientific computations and has been discussed in several existing works [5], [7].

## B. Task Utility Model

In our model, if a cloud service provider can successfully schedule a task $\tau_i$, the system can gain a utility value $u_i$ from the service requester. Utility-based models are quite useful for real world problems; utility based routing and resource allocation have been studied in other settings [8], [9]. In our work, we introduce the utility-based model for task scheduling in the cloud computing business model. We consider two typical utility models for tasks.

*Uniform utility model*: all tasks have the same utility value, 1. If the task is scheduled, the system gains a utility of 1; if the task is not scheduled, the system gains 0 utility. This simple model is actually practical for situations where the tasks are with similar requirements.

*General utility model*: tasks have different utility values; the utility values of each task are quite arbitrary. This utility model is applicable to various pricing models agreed by the cloud service providers and the customers.

## C. Problem Definition

Given a set of periodic tasks, $\tau$ and the platform with $m$ processing units, our goal is to select a subset of tasks from this task set, to

1.) determine the parallelization option for each segment of the selected tasks,

2.) derive the actual scheduling for all the tasks given their parallelization option,

3.) meet deadlines for all selected tasks, and

4.) maximize the utility gained by the system, which is the sum of all selected tasks' utility values.

## III. Preliminaries

Scheduling periodic sequential tasks on multiprocessor systems has received extensive research efforts. The earliest of these works dates back to 1978, when Dhall and Liu first presented the problem [10]. Authors in [11] first present a theoretically optimal algorithm, named Pfair, for scheduling periodic sequential tasks on multiprocessor systems. Pfair allows full migration and fully dynamic priorities, which may incur frequent scheduling and migration and therefore significant run-time overhead. After that, many researchers proposed optimal multiprocessor scheduling algorithms for periodic sequential tasks with less overheads, for example, PD$^2$ [12], and LLREF [13].

These algorithms are called optimal in the sense that if 1.) each task's requirement/density is less than 1, and 2.), the total requirement/density of all the tasks is less than the capacity of all the processing units, then, the entire task set can be scheduled on the platform without missing any task's deadline. Our scheduling method will adopt one of these optimal algorithms after deciding which tasks to select and what parallelization option to choose for each selected task.

## A. Using Intermediate Deadlines

We have three important steps to take to achieve the goal of maximizing overall utility. First, we need to select a subset of the tasks to schedule. Second, for each of the selected task, we need to decide the parallelization option for each segment of the task. Third, given the parallelization option, we still need to derive the detailed scheduling for the tasks.

For the third step, we decide to leverage one of the optimal scheduling algorithms, such as PD$^2$ [12] and LLREF [13], to derive the actual scheduling. To apply the optimal scheduling algorithms, we need to satisfy the two conditions of the algorithms. We take an approach that assigns an "intermediate deadline" $d_{i,j}$ to each segment $\tau_{i,j}$ for task $\tau_i$. $d_{i,j}$ is the relative deadline for all the parallel threads of $\tau_{i,j}$. Notice that assigning intermediate deadlines for each segment of a task is a straightforward approach to utilize traditional schedulability tests, and has been used in several existing works [2], [7]. Assume that $\tau_i$ is released at time $t$, then all the parallel threads of $\tau_{i,1}$ are released at the same time $t$, and should be completed by $t + d_{i,1}$. Then, the parallel threads of each following segment $\tau_{i,j}$ $(2 \leq j \leq K_i)$ are released when $\tau_{i,j-1}$'s absolute deadline $t + d_{i,1} + \cdots d_{i,j-1}$ has been reached and needs to be completed by $t + d_{i,1} + \cdots d_{i,j-1} + d_{i,j}$. With these intermediate deadlines, we can easily calculate each thread's density as the thread's execution requirement divided by its corresponding intermediate deadline. Applying one of the optimal scheduling algorithms, we have the following argument: if the intermediate deadlines are assigned to the segments of $\tau_i$ such that their sum is smaller than or equal to $D_i$, i.e., $\sum_{j=1}^{K_i} d_{i,j} \leq D_i$, and each thread's density is less than 1, and the total density of all active threads of all tasks is less than $m$, the total capacity of the system, then, task $\tau_i$ can be scheduled without missing its deadline.

We now consider the first two steps in our decision procedure. Notice that, the first step is actually dependent on the second step. Without selecting the parallelization option, we are unable to see whether the total density of a set of tasks is less than or equal to $m$, or not. For this reason, we consider the subproblem, which tries to determine which parallelization option can lead to a minimized task density for each task. For ease of presentation, we name this subproblem the Optimal Parallelization Selection (OPS) problem.

The OPS problem itself consists of two important steps: first, given parallelization options for each segment of the task, allocate intermediate deadlines such that the density of each thread of each segment is less than 1, and the total density of threads in the same segment is less than $m$, i.e., the peak density among all segments is less than $m$, and the total density is minimized; second, among all the parallelization options, determine which parallelization option to choose for each segment of the task.

Let $p_j$ be the parallelization option that $\tau_{i,j}$ decides on, $1 \leq p_j \leq P_{i,j}$. The WCET or execution requirement of the $l$th thread of segment $\tau_{i,j}$ under the $p_j$th parallelization option is $C^l_{i,j}(N^{p_j}_{i,j}), (1 \leq l \leq N^{p_j}_{i,j})$. Then, the maximum execution requirement among all of the $N^{p_j}_{i,j}$ threads can be denoted as:

$$C^{max}_{i,j}(N^{p_j}_{i,j}) = \max_{1 \leq l \leq N^{p_j}_{i,j}} C^l_{i,j}(N^{p_j}_{i,j}).$$

The total execution requirement of all the $N^{p_j}_{i,j}$ threads can be calculated as:

$$C^{total}_{i,j}(N^{p_j}_{i,j}) = \sum_{1 \leq l \leq N^{p_j}_{i,j}} C^l_{i,j}(N^{p_j}_{i,j}).$$

We define the "segment density" $\delta_{i,j}$ for segment $\tau_{i,j}$ as the sum of its thread densities, i.e.,

$$\delta_{i,j} = \sum_{l=1}^{N^{p_j}_{i,j}} \frac{C^l_{i,j}(N^{p_j}_{i,j})}{d_{i,j}} = \frac{C^{total}_{i,j}(N^{p_j}_{i,j})}{d_{i,j}}$$

Notice that, every time, only one segment of a task is active under our intermediate deadline approach. Thus, we define the "task peak density" $\delta_i$ for $\tau_i$ as as its largest segment density,

$$\delta_i = \max_{1 \leq j \leq K_i} \delta_{i,j}$$

The OPS problem for a task $\tau_i$ can be formally stated as follows:

$$\begin{array}{ll} min & \delta_i \\ s.t. & \sum_{j=1}^{K_i} d_{i,j} \leq D_i. \\ & \frac{C^l_{i,j}(N^{p_j}_{i,j})}{d_{i,j}} \leq 1, \forall 1 \leq l \leq N^{p_j}_{i,j}. \\ & \delta_i \leq m. \end{array}$$

The optimization variables are $d_{i,j}$'s for deadline allocation, and $p_j$ for parallelization selection for all segments. The objective is to minimize $\tau_i$'s density.

## *B. Practical Assumptions and Fundamental Results*

In practical situations, we have the following conditions for each task:

$$C^{total}_{i,j}(N^1_{i,j}) \leq C^{total}_{i,j}(N^2_{i,j}) \leq \cdots \leq C^{total}_{i,j}(N^{P_{i,j}}_{i,j}),$$

$$C^{max}_{i,j}(N^1_{i,j}) \geq C^{max}_{i,j}(N^2_{i,j}) \geq \cdots \geq C^{max}_{i,j}(N^{P_{i,j}}_{i,j}).$$

The meaning for the first inequality is that, if we parallelize the segment to more threads, the total execution requirement of the segment will be increased due to increased parallelization overhead. The meaning for the second inequality is that, by slicing the task segment into more threads, the maximum execution requirement among the parallel threads decreases; otherwise, we may not want to slice the task segment into more threads.

Thanks to the fundamental work in [7], under these two conditions, the OPS problem can be solved optimally. We adopt the method in [7] to solve the OPS problem. From now on, we only consider the minimum density of each task; thus, we simply use $\delta_i$ to represent $\tau_i$'s minimum density. Here, we deal with some special cases: if $\delta_i > m$, we can safely eliminate $\tau_i$ from the candidate task set, because it will never be schedulable; if $\sum_{i=1}^{n} \delta_i \leq m$, we can select all of the tasks to maximize the system utility. In the follow-up discussions, we do not consider these special cases anymore.

## *C. Illustration Using Single Segment Tasks*

Instead of presenting all the details in [7], we use a simpler model to illustrate the solution. In this section, we consider a single-segment task model, where each task has only one parallelizable segment, i.e., $K_i = 1$. Then, the deadline for the task is the deadline for the only segment of the task. Similarly, the execution time of the $l$th thread of the single segment of $\tau_i$, with the $p$th optimization option is $C^l_{i,1}(N^p_{i,1})$. The maximum and total execution requirements of the same segment under the $p$th parallelization option are $C^{max}_{i,1}(N^p_{i,1})$ and $C^{total}_{i,1}(N^p_{i,1})$, respectively. The total execution time of all the threads of the segment has the following relation:

$$C^{total}_{i,1}(N^1_{i,1}) \leq C^{total}_{i,1}(N^2_{i,1}) \leq \cdots < C^{total}_{i,1}(N^{P_{i,1}}_{i,1}).$$

The maximum execution time among all of the threads has the following relation:

$$C^{max}_{i,1}(N^1_{i,1}) \geq C^{max}_{i,1}(N^2_{i,1}) \geq \cdots \geq C^{max}_{i,1}(N^{P_{i,1}}_{i,1}).$$

To apply PD$^2$ or LLREF algorithms, we have to satisfy two conditions: first, the execution time of each thread of the task should be less than or equal to $D_i$, i.e., the maximum execution time among all threads of the task should be less than or equal to $D_i$; second, the total density of all threads of the task should be less than $m$. For single segment tasks, the single segment's relative $d_{i,1}$ is equal to the task's deadline $D_i$. Since $C^{max}_{i,1}$ decreases with the parallelization degree, if $C^{max}_{i,1}(N^1_{i,1}) > D_i$, we can try increasing the parallelization degree to $N^2_{i,1}$; if $C^{max}_{i,1}(N^2_{i,1}) > D_i$, we can try increasing the parallelization degree to $N^3_{i,1}$. Assuming that the first parallelization degree that has $C^{max}_{i,1} \leq D_i$ is $N^j_{i,1}$, then we should use this parallelization degree for task $\tau_i$, because, further increasing the parallelization degree will only increase the total density of this task, which will risk the entire task being unschedulable. The idea here is that, we should provide each task *just enough* parallelization degree. Thus, the problem of minimizing the density of a task is to select the *just enough* parallelization degree such that the maximum execution time of each thread is less than or equal to $D_i$. When there are multiple segments in each task, the main idea is still to select just enough parallelization degree for each task, though the solution is not as obvious as that of the single segment tasks.

Take a look at tasks $\tau_1$, $\tau_2$, and $\tau_3$ in the motivational example as shown in Table I again. Any of the three tasks cannot meet their deadline if we choose the first parallelization option, i.e., using just one thread to execute the task, because their deadlines of using just one thread are 7, 6, and 6, which are greater than their deadline of 5. We can also see that the second parallelization option provides enough parallelization degree such that they can meet their deadline. Thus, the minimum density of each task can be determined by choosing the second parallelization option for each task.

## IV. Uniform Utility Model

Under the uniform utility model, to maximize the system utility is equivalent to maximizing the number of tasks to be scheduled. In order to schedule the maximum number of tasks, we need to select the tasks with the least density first. Thus, our algorithm first sorts the tasks in ascending order of their minimum densities, $\delta_i$. After that, we pick the tasks one by one as long as the total density is less than or equal to $m$. We refer to this algorithm as the Optimal Task Selection (OTS) algorithm.

*Theorem 1:* The OTS algorithm achieves optimal task selection in polynomial time for the problem under the uniform utility model.

*Proof:* Obviously, the algorithm is with polynomial time complexity: deriving the minimum density for each task and scheduling each task is polynomial time solvable [7]. The time complexity of sorting all the tasks according to their densities is $\Theta(nlogn)$, and the task selection procedure is $\Theta(n)$.

We prove optimality by contradiction. For ease of presentation, we assume that the tasks are sorted in ascending order of their densities, in other words, $\delta_1 \le \delta_2 \le \cdots \le \delta_n$. Assume that the OTS algorithm described above selects $j$ tasks. If $j = n$, i.e., all tasks are selected, it is obviously an optimal solution. If $j < n$, then the greedy algorithm will select the first $j$ tasks and we have

$$\delta_1 + \delta_2 + \cdots + \delta_j \le m,$$

and

$$\delta_1 + \delta_2 + \cdots + \delta_j + \delta_{j+1} > m.$$

Assume that there exists a better valid solution which can select $j+1$ tasks, i.e., $\tau_{i_1}$, $\tau_{i_2}$, $\cdots$, $\tau_{i_{j+1}}$. Since all tasks are sorted in ascending order of their densities, we have

$$\delta_{i_1} + \cdots + \delta_{i_{j+1}} > \delta_1 + \cdots + \delta_{j+1} > m.$$

The above inequality indicates that the "better" solution does not exist. Thus, there does not exist any valid solution that can select more tasks than the OTS algorithm. The theorem is hence proved. ∎

## V. General Utility Model

*Theorem 2:* Under the general utility model, the problem of selecting and scheduling tasks to achieve maximum system utility is NP-complete.

*Proof:* First, we know that deriving the minimum density for each task is a polynomial time problem, according to [7]. After achieving the minimum density for each task, the problem becomes selecting a subset of tasks such that, their total minimum density is less than or equal to $m$, and the achieved utility is maximized. Each task can either be selected or not be selected. It is easy to notice that the task selection problem reduces to the classic 0-1 Knapsack problem, which is a well known NP-complete problem. ∎

**Algorithm 1** DPOTS$(u, \delta, m)$

```
Input:
    Utility vector of tasks, $u = \{u_1, \cdots, u_n\}$; density vector,
    $\delta = \{\delta_1, \cdots, \delta_n\}$; the number of processing units, $m$;
1:  Let $\Delta_{(n+1)\times(U(\tau)+1)}$ be a two dimensional array.
2:  $\Delta[0,0] = 0$;
3:  for $U := 1$ to $U(\tau)$ do
4:      $\Delta[0, U] = \infty$;
5:  for $i := 1$ to $n$ do
6:      for $U := 1$ to $U(\tau)$ do
7:          if $u_i \le U$ then
8:              $\Delta[i, U] = \min(\Delta[i-1, p], \Delta[i-1, U-u_i] + \delta_i)$
9:          else
10:             $\Delta[i, U] = \Delta[i-1, U]$
11: $OPT = \max\{U : 0 \le U \le U(\tau)$ and $\Delta[i, U] \le m, \forall 0 \le i \le n\}$
12: Using $\Delta$ to find a subset $S$ of profit $OPT$ and total weight
    less than or equal to $m$.
13: return $S$
```

### A. Polynomial Time Solution for Integer Utility Values

If the utility of all the tasks are integer values, we provide the following pseudo-polynomial time dynamic programming algorithm that achieves the optimal solution.

Define $U(S)$ to be the total utility of all task of task set $S$, where $S$ is a subset of all tasks:

$$U(S) = \sum_{\tau_i \in S} u_i, \forall S \subseteq \{\tau_1, \tau_2, \cdots, \tau_n\}.$$

Define $D(S)$ to be the total density of all tasks of task set $S$:

$$D(S) = \sum_{\tau_i \in S} \delta_i, \forall S \subseteq \{\tau_1, \tau_2, \cdots, \tau_n\}.$$

Then, $U(\tau)$ will be the total utility value of all the tasks. Define

$$\Delta[i, U] = \min\{D(S) : U(S) = U, \forall S \subseteq \{\tau_1, \tau_2, \cdots, \tau_i\}\},$$

i.e., $\Delta[i, U]$ denotes the minimum possible density of any subset of the first $i$ items such that the subset's total utility is exactly $U$. When there is no such a subset of profit exactly $U$, then we define $\Delta[i, U] = \infty$. Remember that we need to select a subset of density at most $m$ with the maximum utility. This maximum utility is given by $\max\{U : 0 \le U \le U(\tau), \Delta[i, U] \le m, \forall 1 \le i \le n\}$. This means that if we can compute all values in $\Delta[i, U]$, then we can compute the maximum utility. From the values in $\Delta[i, U]$, we can also compute a subset whose utility is the optimal value. $\Delta[i, U]$ has the following recursive relation:

**Algorithm 2** FPTAS$(u, \delta, m, \epsilon)$

```
1: $u_{max} = \sum_{1\leq i\leq n} u_i$;
2: for $i := 1$ to $n$ do
3:     $u_i^* = \lceil \frac{u_i}{(\epsilon/n)max} \rceil$;
4: $S = DPOTS(u^*, \delta, m)$;
5: return $U(S)$
```

*Lemma 1:*

$$\Delta[i,U]=\begin{cases} 0 & U=0 \\ \infty & i{=}0,\ U{>}0 \\ \Delta[i-1,U] & i{>}0,\ 0{<}U{<}u_i \\ \min\{\Delta[i{-}1,U], \Delta[i{-}1,U{-}u_i]{+}\delta_i\} & i{>}0,\ U{\geq}u_i \end{cases}$$

*Proof:* The first two cases are predefined. We consider how to calculate $\Delta[i, U]$ in general cases. If $u_i > U$, then $\tau_i$ cannot be included in the subset $S$, $S \subseteq \{\tau_1, \tau_2, \cdots, \tau_i\}$, whose total utility sums to $U$; this indicates the third case in the recursive relation. If $u_i \leq U$, we have two options to select a subset $S$, $S \subseteq \{\tau_1, \tau_2, \cdots, \tau_i\}$: first, include $\tau_i$, then, the minimal total density is $\Delta[i{-}1, U{-}u_i]{+}\delta_i$; second, do not include $\tau_i$, then, the minimal total density is $\Delta[i-1, U]$. As defined $\Delta[i, U]$ should choose the smaller density of these two cases, which proves the fourth case. ■

Algorithm 1 sketches the Dynamic Programming algorithm for Optimal Task Selection (DPOTS), when utility values of all tasks are integers. Its time complexity is $\Theta(nU(\tau))$.

### B. FPTAS algorithm for General Utility Values

We now design the FPTAS algorithm for the problem when utilities of tasks are not necessarily integers. We define $u_{max} = \max_{i=1}^{n} u_i$, and the new utility value for task $\tau_i$ as

$$u_i^* = \lceil \frac{u_i}{(\epsilon/n)u_{max}} \rceil,$$

where $0 < \epsilon < 1$. Notice that, the new utility values $u_i^*$ will be integers within range $[1, \lceil n/\epsilon \rceil]$; thus, we can apply the dynamic programming algorithm on the integer utility values, and derive the achieved total utility value. Algorithm 2 sketches the FPTAS algorithm for our problem. Denote the real maximum (optimal) utility for the problem as $OPT$.

*Theorem 3:* Algorithm 2 computes in $O(n^3/\epsilon)$ time. The solution achieved is at least $(1-\epsilon)OPT$.

*Proof:* $u_i^* \leq \lceil n/\epsilon \rceil$; thus, the total profit is at most $n\lceil n/\epsilon \rceil$. Thus the total run time is $\Theta(n \times n\lceil n/\epsilon \rceil) = \Theta(n^3/\epsilon)$. Let $S_{opt}$ denote an optimal subset, that is, a subset of tasks whose total density is at most $m$ such that $U(S_{opt}) = OPT$. Let $S^*$ denote the subset returned by the algorithm. Let $U^*(S)$ be the total converted utility values of tasks in subset $S$. Because $S^*$ is optimal for the new profits, we have $U^*(S^*) \geq U^*(S_{opt})$. Due to the integer rounding procedure, we have

$$\frac{u_i}{(\epsilon/n)u_{max}} \leq u_i^* \leq \frac{u_i}{(\epsilon/n)u_{max}} + 1.$$

$OPT$ can be derived as follows:

$$\begin{aligned} OPT &= \sum_{\tau_i \in S_{opt}} u_i \\ &\leq \sum_{\tau_i \in S_{opt}} (\epsilon/n)u_{max}u_i^* \\ &\leq (\epsilon/n)u_{max}U^*(S_{opt}) \\ &\leq (\epsilon/n)u_{max}U^*(S^*) \\ &\leq (\epsilon/n)u_{max}\sum_{\tau_i \in S^*}(\frac{u_i}{(\epsilon/n)u_{max}} + 1) \\ &= U(S^*) + (\epsilon/n)u_{max}|S^*| \\ &\leq U(S^*) + \epsilon u_{max} \\ &\leq U(S^*) + \epsilon OPT. \end{aligned}$$

Thus, $U(S^*) \geq (1-\epsilon)OPT$. ■

**Algorithm 3** Greedy$(u, \delta, m)$

```
1: Sort items in non-increasing order of $u_i/\delta_i$.
2: Greedily add tasks until we hit an item $\tau_i$ that is too big,
   i.e., $\sum_{\tau_i} \delta_i > m$.
3: Pick the better of $\{\tau_1, \tau_2, \cdots, \tau_{i-1}\}$ and $\tau_i$.
```

### C. 1/2-Approximation Algorithm

Using the FPTAS algorithm, we can achieve a solution that is arbitrarily close to the optimal solution. However, the time complexity of the algorithm is too high when we want a solution that is close to the optimal solution. Here, we provide another algorithm, with a time complexity of $\Theta(n)$, that has a satisfactory bound on the utility we can achieve. The algorithm is described in Algorithm 3.

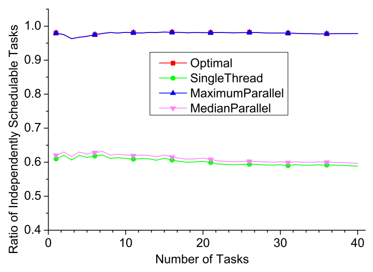
*Theorem 4:* Algorithm 3 is a $1/2$-approximation algorithm for the original problem.

*Proof:* We employed a greedy approach. Therefore we can say that if our solution is suboptimal, we must have some leftover space $\Delta\delta$ at the end. Imagine our algorithm was able to take a fraction of a task. Then, by adding $\frac{\Delta\delta}{\delta_i}u_i$ to our knapsack value, we would either match or exceed $OPT$. Therefore, either $\sum_{k=1}^{i-1} u_k \geq 1/2\ OPT$, or $u_i \geq \frac{\Delta\delta}{\delta_i}u_i \geq 1/2\ OPT$. By selecting the better of these two, we can conclude that the algorithm can achieve an approximation ratio of $1/2$. ■

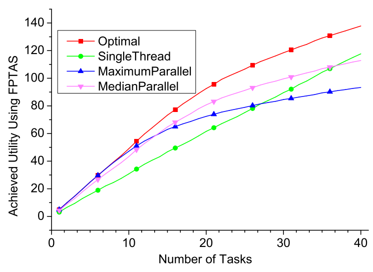
## VI. Simulations

### A. Comparing Methods

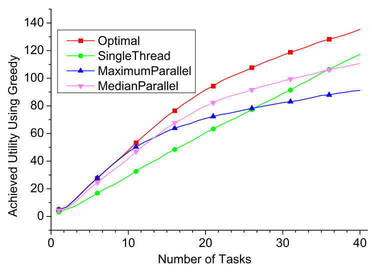
We compare several task scheduling methods for the multi-segment task model. Specifically, we compare the optimal density minimization and task scheduling methods against methods that minimize the density of each task given fixed parallelization for each segment of a task. We denote the optimal density minimization and task scheduling method as "Optimal", which is the method we use in this paper. For methods with fixed parallelization options, we consider three typical cases: using a single thread for each segment of a task, choosing the maximum number of threads for each segment, and choosing the median parallelization option for each segment. We denote the three methods as "SingleThread," "MaximumParallel," and "MedianParallel," respectively. Notice that, given fixed parallelization option for each segment of a task, the offline optimal deadline allocation among the segment to minimize the task density is considered in [2] by using linear programming solvers. We identify that, if the parallelization option is fixed beforehand, the problem



(a) % of schedulable tasks.

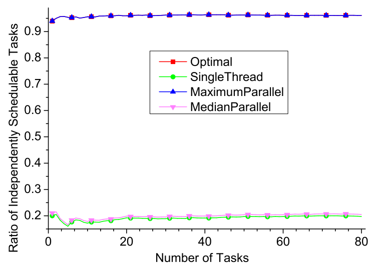


(b) Achieved utility using FPTAS.

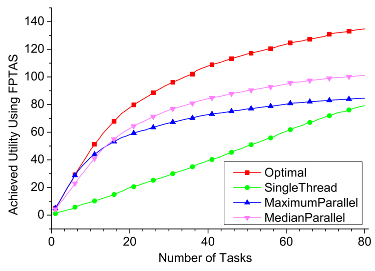


(c) Achieved utility using Greedy.

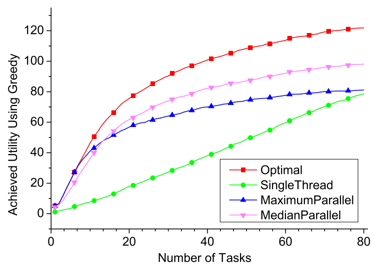
Fig. 2. 40 tasks on 20 processing units



(a) % of schedulable tasks.



(b) Achieved utility using FPTAS.



(c) Achieved utility using Greedy.

Fig. 3. 80 tasks on 20 processing units; tasks with increased densities.

becomes a simpler instance of our general problem, and thus can be solved using the same algorithm as that of Optimal.

For task selection algorithms, we consider the Fully Polynomial Time Approximation Scheme, denoted as "FPTAS," and the 1/2-approximation algorithm, denoted as "Greedy." Our objective is to show what the actual performance differences are between these two task selection algorithms for our problem under various situations.

## B. Simulation Settings

Modern computational systems usually have tens of, if not hundreds of, cores. In our simulation, we target systems with 20 processing units, such as the Intel Xeon Processor E7-8870 [14] and Intel Broadwell-E Core i7-6950X [15].

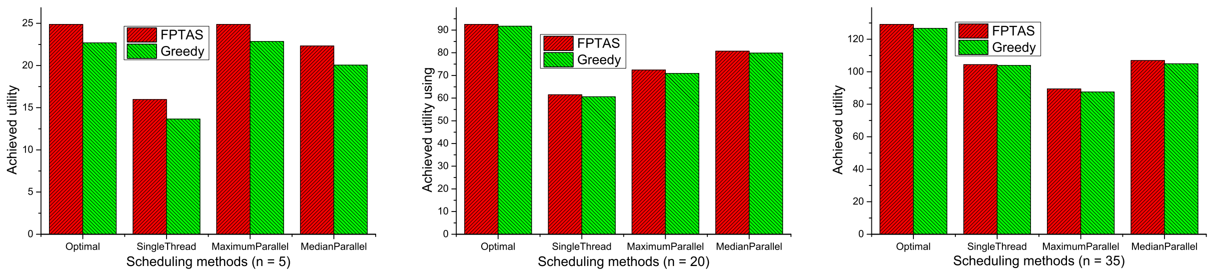
To consider the utility achieved under various situations, each task is randomly generated, with random deadline tightness, parallelization overhead, and $C^{max}$ and $C^{total}$ values. We first generate 40 such random tasks; each time, we only consider scheduling the first $n$ tasks ($1 \leq n \leq 40$); that is, we try to select a subset of tasks from the first $n$ tasks, such that the selected tasks can be scheduled without missing each task's deadline, and the total utility of all selected tasks be maximized. For each specific setting, we repeat the simulation 100 times, and compare the average values.

## C. Simulation Results

We call a task "independently schedulable" if the task can be scheduled onto the platform when it is the only task on the platform. Then, a task is independently schedulable if and only if each thread of the task has a density less than or equal to 1, and the maximum density of the task is less than or equal to $m$. In our task generation scheme, we do not allow a task to have density greater than or equal to $m$. Thus, if a task is not independently schedulable, it means that the segments of the task do not have the parallelization option that can use enough cores for parallel processing, or the parallelization does not reduce the single thread execution requirement significantly.

Fig. 2(a) shows the percentage of independently schedulable tasks under each method, i.e., Optimal, SingleThread, MaximumParallel and MedianParallel. We can see from Fig. 2(a) that, when there are a large number of tasks, there almost always exists some task(s) that are not "independently schedulable". Optimal has the highest percentage of independently schedulable tasks; MaximumParallel has the same percentage of independently schedulable tasks, because, using a larger parallelization options reduces the maximum execution requirement among the threads for each segment of each tasks; though it may increase the total execution requirement for each segment of the task, it will not affect the schedulability of the task as long as the total execution requirement is not greater than $m$. SingleThread has the lowest schedulable percentage because it cannot utilize multiple threads for parallel processing. MedianParallel has a higher schedulable percentage than SingleThread, but it is still lower than Optimal and MaximumParallel, because its ability to utilize multiple threads are limited.

Fig. 2(b) shows the utility achieved under each method using the FPTAS algorithm for selecting tasks to schedule. It is no surprise that the Optimal achieves the maximum utility. When the number of tasks is small, for example, less than 14, MaximumParallel achieves a higher utility than MedianParallel, because it has a large set of schedulable tasks to choose from, while MedianParallel has a limited number of schedulable tasks to choose from. When the number of tasks is large, MaximumParallel achieves a lower utility than MedianParallel. The reason is that though MaximumParallel has a large set of schedulable tasks, when using the maximum number of threads for each segment of a task, the total execution requirement



(a) n = 5 (b) n = 20 (c) n = 30

Fig. 4. FPTAS vs Greedy

of each task is much higher than MedianParallel; given the number of processors on the system, the total number of tasks that can be scheduled is thus lower than MedianParallel, and consequently, the total achieved utility is lower. Similarly, when the number of tasks is smaller than 24, SingleThread achieves a lower utility value than MaximumParallel; when the number of tasks is greater than 24, SingleThread achieves a higher utility than MaximumParallel.

Fig. 2(c) shows the utility achieved under each method using the Greedy algorithm for selecting tasks to schedule. It demonstrates similar patterns as in Fig. 2(b).

To compare the performances of the task selection algorithms, FPTAS and Greedy, we compare their achieved utility values using various scheduling algorithms. When the number of tasks is small, as shown in Fig. 4(a), Greedy achieves obviously lower utilities. When the number of tasks is large, the differences between their utilities values are minimal (Figs. 4(b) and 4(c)). Since FPTAS can be considered as the optimal solution, we can conclude that the Greedy algorithm achieves near-optimal utility under general cases.

We also conduct simulations when there are a total of 80 tasks and the tasks have increased execution density. The results are presented in Fig. 3. As we can see, the percentage of independently schedulable tasks further decreases for SingleThread and MedianParallel due to tasks' increased density.

## VII. Conclusion

We consider utility-based scheduling for periodic tasks on platforms with multiple homogeneous processing units, with the goal of maximizing the system's overall utility achieved by scheduling the tasks. We consider two typical types of utility models, namely, the uniform utility model and the general utility model. For the first utility model, we give the optimal solution for selecting and scheduling the tasks; for the second model, we prove that the problem can be reduced to the classic 0-1 Knapsack problem, and thus is NP-complete; we then provide an FPTAS algorithm with an arbitrary approximation ratio for the problem. The FPTAS algorithms are known for their high time complexity, especially if we want to achieve a good approximation ratio. We provide a 1/2 approximation algorithm with a significantly reduced time complexity. Simulations and comparisons show that our proposed algorithms have good performances in terms of maximizing system utility, and the 1/2 approximation algorithm can achieve near-optimal solutions under general conditions.

## Acknowledgement

This research was supported in part by NSF grants CNS 1449860, CNS 1461932, CNS 1460971, CNS 1439672, CNS 1301774, and ECCS 1231461.

## References

[1] A. Saifullah, K. Agrawal, C. Lu, and C. Gill, "Multi-core real-time scheduling for generalized parallel task models," in *32nd IEEE Real-Time Systems Symposium (RTSS)*, Nov. 2011, pp. 217–226.

[2] G. Nelissen, V. Berten, J. Goossens, and D. Milojevic, "Techniques optimizing the number of processors to schedule multi-threaded tasks," in *Proc. of the 24th Euromicro Conference on Real-Time Systems*, July 2012, pp. 321–330.

[3] H. S. Chwa, J. Lee, K. M. Phan, A. Easwaran, and I. Shin, "Global edf schedulability analysis for synchronous parallel tasks on multicore platforms," in *25th Euromicro Conference on Real-Time Systems (ECRTS)*, July 2013, pp. 25–34.

[4] J. Blazewicz, M. Y. Kovalyov, M. Machowiak, D. Trystram, and J. Weglarz, "Preemptable malleable task scheduling problem," *IEEE Transactions on Computers*, vol. 55, no. 4, pp. 486–490, Apr. 2006.

[5] R. Sudarsan and C. J. Ribbens, "Scheduling resizable parallel applications," in *IEEE International Symposium on Parallel Distributed Processing*, May 2009, pp. 1–10.

[6] F. Liu and J. B. Weissman, "Elastic job bundling: An adaptive resource request strategy for large-scale parallel applications," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2015, pp. 33:1–33:12.

[7] J. Kwon, K. W. Kim, S. Paik, J. Lee, and C. G. Lee, "Multicore scheduling of parallel real-time tasks with multiple parallelization options," in *Proc. of IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, Apr. 2015, pp. 232–244.

[8] M. Xiao, J. Wu, and L. Huang, "Time-sensitive utility-based routing in duty-cycle wireless sensor networks with unreliable links," in *Proc. of the 31st IEEE Symposium on Reliable Distributed Systems (SRDS)*, Oct 2012, pp. 311–320.

[9] J. A. Joao, M. A. Suleman, O. Mutlu, and Y. N. Patt, "Utility-based acceleration of multithreaded applications on asymmetric cmps," in *Proc. of the 40th Annual International Symposium on Computer Architecture*. ACM, 2013, pp. 154–165.

[10] S. K. Dhall and C. L. Liu, "On a real-time scheduling problem," *Oper. Res.*, vol. 26, no. 1, Feb. 1978.

[11] S. K. Baruah, N. K. Cohen, C. G. Plaxton, and D. A. Varvel, "Proportionate progress: A notion of fairness in resource allocation," in *Proc. of the 25th Annual ACM Symposium on Theory of Computing*, 1993, pp. 345–354.

[12] A. Srinivasan and J. H. Anderson, "Fair scheduling of dynamic task systems on multiprocessors," *J. Syst. Softw.*, vol. 77, no. 1, pp. 67–80, 2005.

[13] H. Cho, B. Ravindran, and E. D. Jensen, "An optimal real-time scheduling algorithm for multiprocessors," in *27th IEEE International Real-Time Systems Symposium*, Dec 2006, pp. 101–110.

[14] "Intel Xeon Processor E7-8870," http://ark.intel.com/products/53580/Intel-Xeon-Processor-E7-8870-30M-Cache-2_40-GHz-6_40-GTs-Intel-QPI, accessed: 2016-05-03.

[15] "Intel Broadwell-E Core i7-6950X," http://wccftech.com/intel-broadwell-e-core-i7-6950x-price/, accessed: 2016-05-03.