Utility Accrual Real-Time Scheduling Under Variable Cost Functions

Abstract

We present a utility accrual real-time scheduling algorithm called CIC-VCUA, for tasks whose execution times are functions of their starting times (and potentially other factors). We model such variable execution times using *variable cost functions* (or VCFs). The algorithm considers application activities that are subject to time/utility function time constraints, execution times described using VCFs, and mutual exclusion constraints on concurrent sharing of non-CPU resources. We consider the two-fold scheduling objective of (1) assure that the maximum interval between any two consecutive, successful completions of job instances *in an activity* must not exceed the activity period (an application-specific objective), and (2) maximizing the system's total accrued utility, while satisfying mutual exclusion resource constraints. Since the scheduling problem is intractable, CIC-VCUA is a polynomial-time heuristic algorithm. The algorithm statically computes worst-case task sojourn times, dynamically selects tasks for execution based on their potential utility density, and completes tasks at specific times. We establish that CIC-VCUA achieves optimal timeliness during under-loads, and tightly upper bounds inter- and intra-task completion times. Our simulation experiments confirm the algorithm's effectiveness and superiority.

Index Terms

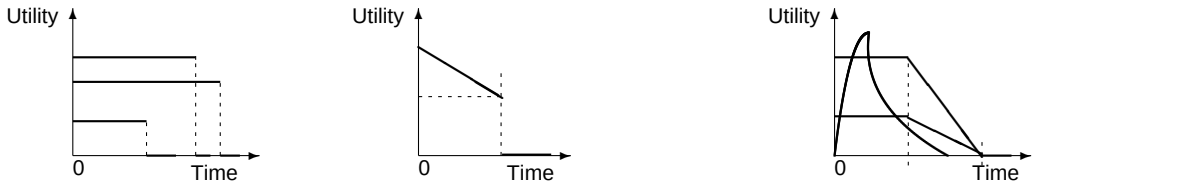
variable-cost functions, time/utility functions, utility accrual scheduling, real-time scheduling, overload scheduling, dynamic scheduling, resource management, mutual exclusion

I. Introduction

Embedded real-time systems that are emerging in many domains such as robotic systems in the space domain (e.g., NASA/JPL's Mars Rover [1]) and control systems in the defense domain (e.g., airborne trackers [2]) are fundamentally distinguished by the fact that they operate in environments with dynamically uncertain properties. These uncertainties include transient and sustained resource overloads due to context-dependent activity execution times and arbitrary activity arrival patterns. Nevertheless, such systems call for the strongest possible assurances on activity timeliness behavior. Another important distinguishing feature of these systems is their relatively long execution time magnitudes compared to traditional real-time systems—e.g., in the order of milliseconds to seconds, or seconds to minutes.

When resource overloads occur, meeting time constraints (for example, deadlines) of all application activities is impossible as the demand exceeds the supply. The urgency of an activity is typically orthogonal to the relative importance of the activity—e.g., the most urgent activity can be the least important; the most urgent can be the most important, and vice versa. Hence when overloads occur, completing the most important activities irrespective of activity urgency is often desirable. Thus, a clear distinction has to be made between the urgency and the importance of activities, during overloads. During under-loads, such a distinction need not be made, because deadline-based scheduling algorithms such as EDF are optimal for those situations [3]—i.e., they can satisfy all deadlines.

Deadlines by themselves cannot express both urgency and importance. Thus, we employ the abstraction of time/utility functions (or TUFs) [4] that express the utility of completing an application activity as a function of that activity's completion time. We specify a deadline as a binary-valued, downward "step" shaped TUF; Figure 1(a) shows examples; a classical deadline has unit utility values $[1, 0]$. Note that a TUF decouples importance and urgency—i.e., urgency is measured as a deadline on the X-axis, and importance is denoted by utility on the Y-axis.



(a) Some Step TUFs (b) MITRE/TOG AWACS TUF [2] (c) GD/CMU BM/C2 TUFs [5]

Fig. 1: Example TUF Time Constraints

Many real-time systems also have activities that are subject to *non-deadline* time constraints, such as those where the utility attained for activity completion *varies* (e.g., decreases, increases) with completion time. This is in contrast to general deadlines, where a positive utility is attained for completing the activity anytime before the deadline, after which zero, or negative utility is attained. Figures 1(b)–1(c) show examples of such time constraints from two actual experimental applications in the defense domain: (1) an Airborne Warning And Control System (AWACS) built by The MITRE Corporation and The Open Group (TOG) [2] and (2) a battle management (BM)/command and control (C2) application built by General Dynamics (GD) and Carnegie Mellon University (CMU) [5]. Details of these applications can be found in [2] and [5], respectively; for brevity, they are omitted here.

When activity time constraints are specified using TUFs, which subsume deadlines, the scheduling

criterion is based on accrued utility, such as maximizing sum of the activities' attained utilities. We call such criteria *utility accrual* (or UA) criteria, and call scheduling algorithms that optimize them UA scheduling algorithms.

UA algorithms that maximize summed utility under downward step TUFs (or deadlines) [6]–[8] default to EDF during under-loads, since EDF can satisfy all deadlines during those situations. Consequently, they obtain the maximum possible accrued utility during under-loads. When overloads occur, UA algorithms favor activities that are more important (since more utility can be attained from them), irrespective of their urgency. Thus, UA algorithms' timeliness behavior subsumes the optimal timeliness behavior of deadline scheduling.

In this paper, we focus on *variable cost* scheduling. In the context of this paper, "cost" means the duration of an activity, a term that comes from one of the interesting and important applications for such scheduling. The model presented requires scheduling activities consisting of sequences of jobs whose durations (e.g., execution times) vary depending on when they begin, or on how long the parent activity has been running, or on other factors. For the algorithms presented, the varying cost is specified by a cost function, which specifies the job's duration as a function of its start time. Thus, even if there were no new activity arrivals, the load to be scheduled changes while the activities are being performed.

Previous efforts on deadline-based and UA scheduling do not consider variable cost scheduling. For example, previous UA scheduling algorithms [6]–[8] do not allow task execution times to vary while tasks are being performed. The *imprecise computation* [9] and *IRIS (Increasing Reward with Increasing Service)* [10] models include optional parts in addition to the mandatory parts of task execution times. However, these models are different from UA and variable cost scheduling, because in these models, the longer the optional part executes, the higher the reward becomes. On the other hand, in UA scheduling, the utility (reward) can only be accrued by an activity when it is completed, and the utility value is decided by the completion time. Further, there are no optional parts in variable cost scheduling—task execution times only contain the mandatory parts and they may vary while the tasks are being performed.

The task model of the TAFT scheduler [11] allows variable task execution times. In TAFT, a task is allowed to have a main part and an exception part (which is executed when the main part misses its time constraint). The execution time of the main part is described as an "expected-case execution time" (the execution time of the exception part is described as a worst-case execution time). The authors describe the expected-case execution time of a task as a "measure for the time that a certain percentage of instances of

the task needs for a successful completion." This model is fundamentally different from our cost function model, where the task execution time depends upon when the task starts its execution (or other factors). The execution time represented on our cost function is a deterministic estimate, which is a function of time. In contrast, TAFT's expected-case execution time is *time-independent*.

Thus, no previous efforts have studied the problem space that intersects UA scheduling and variable cost scheduling. In this paper, we precisely focus on this unexplored problem space. We consider repeatedly occurring application activities whose time constraints are specified using TUFs. The execution times of activities are described by cost functions, which may vary as the activities are being performed. Activities may concurrently, but mutually exclusively, share non-CPU resources. For such a model, we consider a two-fold scheduling criterion: (1) assure that the maximum interval between any two consecutive, successful completions of job instances *in an activity* must not exceed the activity period (an application-specific criterion); and (2) maximize the system's summed utility.

This problem is $\mathcal{NP}$-hard. We present a polynomial-time heuristic algorithm for the problem, called *Completion Interval Constrained Variable Cost Utility Accrual Algorithm* (or CIC-VCUA). We prove several timeliness properties of the algorithm including optimal timeliness during under-loads, and tight upper bounds on completion times between tasks (i.e., activities), and between jobs of one task. Further, we establish that the algorithm is deadlock-free and safe. Finally, our experimental simulation studies confirm CIC-VCUA's effectiveness and superiority.

Thus, the paper's contribution is the CIC-VCUA algorithm. To the best of our knowledge, we are not aware of any other efforts that solve the problem solved by CIC-VCUA.

The rest of the paper is organized as follows: In Section II, we describe a motivating application for variable cost scheduling; in Section III, we outline our activity and timeliness models, and state the scheduling criterion. We present CIC-VCUA in Section IV and establish the algorithm's properties in Section V. The experimental measurements are reported in Section VI. Finally, we conclude the paper and identify future work in Section VII.

II. Motivating Application

One application context of interest to us for variable cost scheduling is an air-to-air radar tracking problem for which no scheduling algorithms and performance assurances have been publicly available. To motivate the work in this paper, we simplify and omit some characteristics of the tracking problem to expedite the creation of an initial plausible scheduling approach that can be generalized in subsequent

work. This problem is representative of a large class of related variable-cost scheduling problems which arise in sensor systems with both sensor collection and data processing times which are strongly dependent on the physical geometry of the sensor and target observation area.

This type of tracking problem employs an Active, Electronically Steerable Array (AESA) radar to provide an end-to-end tracking service supporting the evolution of a track through the phases described below. Examples of such systems include the radar systems installed in certain United States and European tactical aircraft and Naval surface craft [12]. An AESA radar uses an array of antennas to form a single virtual "beam," by varying the power, transmission frequencies, and sampling rates across the individual antenna elements. Consequently, the time required to deliver or measure a given amount of power in a particular direction (the pulse width) is a function of the relative geometry of the antenna, the desired beam shape and signal to noise ratio, and the relative geometry of the antenna and target.

Generally, a single task executed by such a radar consists of a coherent ensemble of "dwells", which are pulses of energy. Each such task is strongly a function of geometry and desired quality of the return signal to be collected. For instance, the number of individual dwells transmitted in an air tracking task dictates the amount of ambiguity in the resulting range and range-rate measurements. This ambiguity is also a function of the actual range and range-rate of the target, and thus the time required to achieve a consistently accurate collection varies as the relative geometry of the radar and the target varies.

Some common civilian applications of AESA radars such as the collection of synthetic aperture radar (SAR) imagery [13] for agricultural and forestry use and scientific research exploit a train of these tasks, scheduled coherently, over macroscopic timescales (10^1–10^2 seconds). The total end-to-end time required for such an extended task is again a function of the system geometry, and thus varies in sojourn and execution time.

The problem notionally consists of three component tasks[1]: (1) searching a segment of the airspace to find any airborne moving objects (*track initiation*); (2) maintaining a track for each of those objects until some deadline time; and (3) identifying the object using characteristics of the return pulses. An example of identification is the *Identification Friend or Foe (IFF)* system, but many more complicated mechanisms exist.

Those three tasks for a given object nominally occur in that order, but identification can occur almost any time while tracking. For each of the three tasks, the radar must make one or more measurements by

[1]Hereafter, we use the terms *task* and *activity* interchangeably.

illuminating the object with dwells, then await any return echoes. For convenience, we denote the entire sequence of transmit-wait-receive as a *dwell*. Tasks for any object may be interleaved with any other task by interleaving dwells.

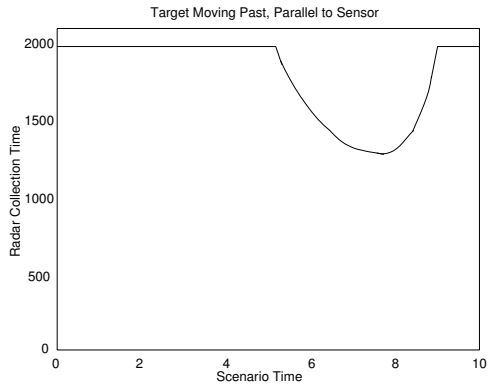
For the tracking tasks, the dwells occur at a revisit rate that is defined by the interval between two successive dwells—regular, but not necessarily periodic. The revisit rate for any particular object must be maintained for a long enough time to obtain acceptable values for certain application-level quality of service (AQoS) metrics.

One such critical AQoS metric is track quality [14], which is a measure of the error in our estimate of the given object's location and motion. Achieving any particular track quality value imposes a lower bound on the revisit rate of the object being tracked, since the estimate error increases quickly in time after each measurement. Optimal and minimum revisit rates are defined by the probability of detecting the object with the next dwell—failure to meet a minimum revisit rate implies increased chance of missing the object on the next dwell [15].

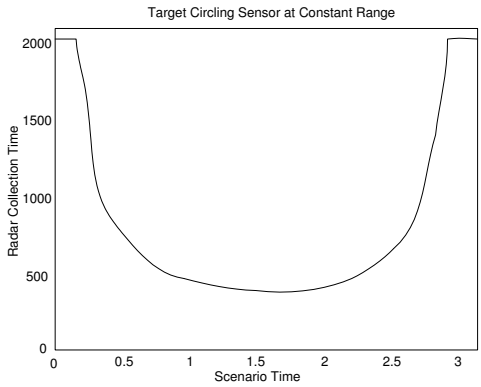
In this application, we associate with each task a cost function which specifies the required duration for a dwell as a function of execution time. This activity cost varies with many factors, including the type of dwell and the geometry of the sensor and target object. For instance, the number and duration of dwells required to search a segment of airspace depends on the relative positions of the radar platform, the scanned airspace, and the objects in that space. Depending on the relative motion of the radar platform and the object, it may be better either to procrastinate dwells (intentionally insert idle time in the radar schedule) or perform dwells early.

The cost function for each task varies with each object's range and look angle (azimuth off the sensor's nominal boresight)—i.e., having the form $f(r, \theta)$. The cost function is derived from the particular tracking problem and an equation known as the radar equation [16]. The radar equation relates the measured energy received to the geometry of the object, the sensor, and the emitted energy.

Two examples of cost functions are shown in Figure 2. Figure 2(a) shows the cost function—the amount of time required on the radar front-end to collect sufficient quantity and quality data—for a target object flying at a higher altitude and faster velocity than the radar platform; Figure 2(b) shows the cost function for a target object circling the radar platform at a constant range. In these cases, the cost achieves a minimum when the target object is along the sensor's boresight. Additionally, the cost increases as a polynomial function of the range (absolute distance) to the object. The specific cost functions can be



(a) Cost Function (Radar Collection Time) for a Target Flying at a Higher Altitude and Faster Velocity than Radar Platform



(b) Cost Function (Radar Collection Time) for a Target Circling the Radar Platform at a Constant Range

Fig. 2: Example Cost Functions

derived in part from the physical properties of the transceiver and antennae.

Note that radar dwell scheduling has been extensively studied by real-time researchers in the past—e.g., [17]–[21]. However, our work fundamentally differs from these works in the variable execution time model of the dwell jobs. These referenced works on dwell scheduling assume that the execution time of a radar dwell job is a *constant* and *time-independent*, enforcing this by forcing all dwells to take a fixed time. In contrast, our work (based on the applications of interest to us) assumes that the execution time of the dwell job depends on the time at which the job starts its execution, and the distance between the radar and the target (besides other factors), and consequently *varies* with time (see Section III-D).

III. Models and Objective

A. System and Task Model

We consider a preemptive system which consists of a set of periodic (dwell) tasks, denoted as $\mathbf{T} = \{T_1, T_2, \cdots, T_n\}$. Each task T_i contains a collection of instances. The period of a task T_i is denoted as P_i. Each task has a begin time and an end time between which execution of all jobs of the task must be completed.

An instance of a task is called a *job*, and we refer to the j^{th} job of task T_i, which is also the j^{th} invocation of T_i, as $J_{i,j}$. The basic scheduling entity that we consider is the job abstraction. Thus, we use J to denote a job without being task specific, as seen by the scheduler at any scheduling event; J_k can be used to represent a job in the scheduling queue.

B. Resource Model

Jobs can access non-CPU resources, which in general, are serially reusable. Examples include physical resources such as disks and logical resources such as locks. Similar to resource access models for fixed-priority scheduling [22] and that for UA scheduling [7], [23], we consider a single-unit resource model. Thus, only a single instance is present for each resource in the system and a job must explicitly specify the resource that it needs.

Resources can be shared and can be subject to mutual exclusion constraints. A job may request multiple shared resources during its lifetime. The requested time intervals for holding resources may be nested, overlapped, or disjoint. We assume that a job explicitly releases all granted resources before the end of its execution.

Jobs of different tasks can have precedence constraints. For example, a job J_k can become eligible for execution only after a job J_l has completed, because J_k may require J_l's results. As in [7], [23], we program such precedences as resource dependencies.

C. Timeliness Model

A job's time constraint is specified using a TUF. Jobs of a task have the same TUF. We use $U_i(\cdot)$ to denote task T_i's TUF, and use $U_{i,j}(\cdot)$ to denote the TUF of T_i's *jth* job. Without being task specific, $J_k.U$ means the TUF of a job J_k; thus completion of J_k at a time t will yield an utility $J_k.U(t)$.

TUFs can be classified into unimodal and multimodal functions. Unimodal TUFs are those for which any decrease in utility cannot be followed by an increase. Examples are shown in Figure 1. TUFs which are not unimodal are multimodal. In this paper, we focus on *non-increasing* unimodal TUFs, as they encompass the majority of the time constraints in our motivating applications. Figures 1(a) and 1(b), and two TUFs in Figure 1(c) show examples.

Each TUF $U_{i,j}, i \in \{1, \cdots, n\}$ has an initial time $I_{i,j}$ and a termination time $X_{i,j}$. Initial and termination times are the earliest and the latest times for which the TUF is defined, respectively. We assume that $I_{i,j}$ is equal to the arrival time of $J_{i,j}$. Further, the period P_i, and the relative termination time X_i of the task T_i, are both equal to $X_{i,j} - I_{i,j}$.

If a job's termination time is reached and its execution has not been completed, an exception is raised, and the job is immediately aborted. Our abortion model follows that of [7], [23], and is based on the observation that if time constraints are not satisfied, it is desirable to place the affected portions of

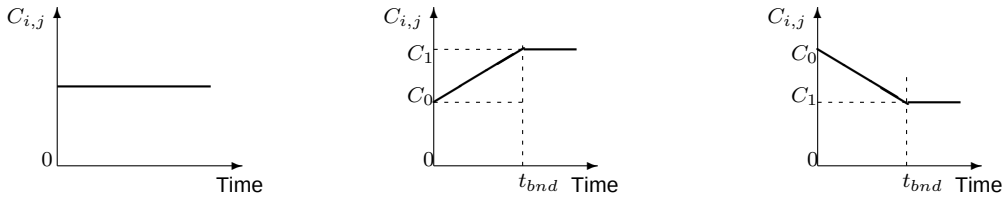
the system and the physical process being controlled into acceptable operating states. Aborting activities provides an opportunity to perform the necessary transformations. Although our algorithm does not require that all activities be aborted (when their time constraints are not met), it is advantageous to exploit the fact that some can be.

D. Task Execution Time Model

As motivated in Section II, after a job is released, its execution time may vary with time. Thus, we define a *variable cost function* (or VCF) for each job, which describes the job execution time as a function of its starting time. Jobs of a task have the same VCFs, so a VCF is also defined for a task. We use $C_i(\cdot)$ to denote task T_i's VCF, and use $C_{i,j}(\cdot)$ to denote T_i's *jth* job's VCF.

For a job $J_{i,j}$, the *x*-axis of its VCF is the absolute time relative to the job's arrival time; the *y*-axis represents its execution time $C_{i,j}(t)$, and the origin shows the execution time of $J_{i,j}$ when it is just released.

Cost functions of AESA applications can be increasing, decreasing, or strictly convex shaped. In this paper, we only consider *unimodal* VCFs. Figure 3 shows examples of VCFs. For jobs whose execution times do not vary with time after their arrivals, we define a constant VCF. Figure 3(a) shows a constant VCF. Figure 3(b) and Figure 3(c) show an increasing VCF and a decreasing VCF, respectively. From Figures 3(b) and 3(c), we can observe that the job $J_{i,j}$'s VCF starts from a non-zero value C_0. We also assume that $C_{i,j}(t)$ is bounded by another non-zero value C_1, which implies that after t_{bnd}, $C_{i,j}(t \geq t_{bnd}) = C_1$.



(a) Constant VCF (b) Increasing VCF (c) Decreasing VCF

Fig. 3: Example Variable Cost Functions for a Job $J_{i,j}$

Without being task specific, $J_k.\mathrm{VCF}$ or $J_k.C$ means the VCF of a job J_k; the execution time of J_k at a time t_{cur} will be $J_k.C(t_{cur}) = J_k.\mathrm{VCF}(t_{cur})$. Hereafter, in the discussion of TUFs and VCFs, we interchangeably use the terms *task* and *job* if no confusion is raised.

E. Scheduling Objective

A successful completion of a job means that the job has met its termination time. With this definition, we consider a two-fold scheduling criterion: (1) assure that the maximum interval between any two consecutive, successful completions of jobs *of a task* must not exceed the task period; and (2) maximize the system's summed utility. Furthermore, mutual exclusion constraints on all shared resources must be respected.

Note that with VCFs, it is difficult to statically calculate the system load, since it dynamically varies with time. For example, a constant load at a task arrival—one that is an under-load—can gradually increase, and can eventually become an overload even without new task arrivals, due to increasing VCFs. Thus, if the dynamic system load is so high such that scheduling objective (1) cannot be satisfied for each task, some tasks may be dropped and consequently aborted. In such cases, tasks that are not dropped are still subject to the two scheduling objectives and mutual exclusion constraints on all shared resources.

IV. The CIC-VCUA Algorithm

This section describes the CIC-VCUA algorithm. In Section IV-A, we first discuss the scheduling metric used by CIC-VCUA, the *Potential Utility Density* (or PUD). CIC-VCUA consists of two steps: static calculation (Section IV-B to IV-C), and the dynamic step (Section IV-D to IV-G).

In the static steps of CIC-VCUA, we first find the maximum possible execution time for each task based on its VCF. Then we label each task as either *selected* or *skipped*, based on their PUDs and contribution to the system load. For the *selected* tasks, the algorithm determines the worst case sojourn time of each task, and attempts to complete all jobs of the task at the same time relative to their arrivals.

After the static step, at each scheduling event, CIC-VCUA builds the dependency chain for each job in the ready queue, and calculates its PUD. The algorithm then sorts them based on their PUDs, in a non-increasing order. Next, the algorithm inserts the jobs into a tentative schedule in the order of their critical times (earliest critical time first), while respecting their resource dependencies and timeliness feasibilities. Finally, CIC-VCUA determines the job to execute, as well as the amount of time for which it will be executed, so as to make sure all jobs of a task have identical sojourn times.

Finally in Section IV-H, we analyze the asymptotic time complexity of the algorithm.

A. Algorithm Rationale

The potential utility that can be accrued by executing a job defines a measure of its "return on investment." Because of the unpredictability of future events (e.g., during overloads), scheduling events that may happen later such as job completions and new job arrivals cannot be considered at the time when the scheduler is invoked. Thus, a reasonable heuristic is to favor "high return" jobs over "low return" jobs in the schedule. This will increase the likelihood of maximizing the summed utility.

The metric used by CIC-VCUA to determine the return on investment for a job is called the PUD, which was originally developed in [7]. The PUD of a job measures the amount of utility that can be accrued per unit time by executing the job itself and other job(s) that it depends upon (due to mutual exclusion constraints on resources held by the other jobs).

To compute job J_k's PUD at current time t_{cur}, CIC-VCUA considers J_k's expected completion time, which is denoted as $J_k.FinT$, and the expected utility by executing J_k and its dependent jobs. For each job J_l that is in J_k's dependency chain and needs to be completed before executing J_k, J_l's expected completion time is denoted as $J_l.FinT$. PUD of J_k is then computed as: $\frac{J_k.U(J_k.FinT)+\sum_{J_l \in J'_k s\ dependency\ chain} J_l.U(J_l.FinT)}{J_k.FinT-t_{cur}}$.

B. Static Job Selection

We assume that if a job cannot complete before its termination time even though it is scheduled immediately, it is infeasible and can be safely aborted. The process of testing the feasibility of a job will be described in Section IV-D.

To test for feasibility, we have to find the maximum possible task execution times. Depending on the VCF shapes, the maximum $C_{i,j}$ for each task can be calculated. For jobs with increasing VCFs, by solving the inequality $C_{i,j}(t) + t \leq X_{i,j}$, we can derive the latest possible starting time $t^b_{i,j}$ of job $J_{i,j}$, such that $C_{i,j}(t^b_{i,j}) + t^b_{i,j} = X_{i,j}$. $C_{i,j}(t^b_{i,j})$ corresponds to the maximum possible execution time of $J_{i,j}$, and $C_i(t^b_i)$ describes this parameter at the task level. For jobs with non-increasing VCFs, a job $J_{i,j}$'s maximum execution time is $C_{i,j}(t^0_{i,j})$.

Therefore, although a job's execution time changes with its starting time, it is possible for us to derive a system load bound $load_b$, which will never be exceeded by the system's dynamic load. For increasing VCFs, we derive $load_b = \sum_{i=1}^{n} \frac{C_i(t^b_i)}{P_i}$; for non-increasing VCFs, we define $load_b = \sum_{i=1}^{n} \frac{C_i(t^0_i)}{P_i}$. If a constant VCF is defined for each task, then a task's execution time is constant and $load_b$ here is the same as the system utilization definition in [24].

Since the system dynamic load may gradually increase even without new task arrivals, the task instances to be executed must be carefully selected in order to accrue more utility. Such selection process is guided by the PUD metric. Toward this, we use a job selection flag, which labels each job as *skipped* or *selected*. The selection process considers the parameters of the task set such as VCFs and TUFs. We associate with each job $J_{i,j}$ a label $SEL_{i,j}$, where $SEL_{i,j} = skipped$ indicates that the job is skipped and $SEL_{i,j} = selected$ indicates that it is selected for execution. At run-time, only jobs whose labels are set to *selected* are dispatched. Thus, the problem becomes choosing the job labels toward optimizing our scheduling objective.

We label jobs in a static and dynamic fashion, based on the workload information used by the scheduler. In the off-line (static) part of CIC-VCUA, we select task instances before the application starts. Initially, all tasks in $\mathbf{T}$ are labeled as *skipped*, i.e., $SEL_{i,j} = skipped$, $\forall i \in 1, \cdots, n, \forall j$. At $t_{cur} = t_i^0$, assuming that tasks are independent of each other, we calculate the PUD of each task, which in value is also the PUD of each task's first job, i.e., $\mathrm{PUD}_i = \frac{U_{i,1}\left(C_{i,1}(t_i^0)\right)}{C_{i,1}(t_i^0)}$. We also calculate the maximum possible execution time of each task ($C_i(t_i^b)$ for increasing VCFs and $C_i(t_i^0)$ for non-increasing VCFs), and then choose the sub task set $\mathbf{T}'$. $\mathbf{T}'$ consists of n' tasks with the largest PUDs, such that for increasing VCFs, $load_b' = \sum_{i=1'}^{n'} \frac{C_i(t_i^b)}{P_i} \leq 1$; and for non-increasing VCFs, $load_b' = \sum_{i=1'}^{n'} \frac{C_i(t_i^0)}{P_i} \leq 1$, and n' is the maximum possible number of tasks to be selected. Note that if $load_b \leq 1$, then $n' = n$. Thus, we favor tasks with larger PUDs, and label the n' tasks in $\mathbf{T}'$ as *selected* i.e., $SEL_{i,j} = selected$, $\forall i \in 1', \cdots, n', \forall j$.

C. Worst-Case Task Sojourn Times

CIC-VCUA's first objective is to assure that the maximum interval between any two consecutive, successful completions of jobs of a task T_i does not exceed its period P_i. In order to satisfy this scheduling objective, the algorithm determines the worst case sojourn time of each task, and attempts to complete all jobs of a task at the same time relative to their arrivals. Doing so ensures that all jobs $J_{i,j}$ of a task T_i have identical sojourn times, satisfying the algorithm's first objective.

As we know $X_i = P_i$, for tasks with step TUFs, the notion of termination time is the same as that of deadline. Thus, the *Earliest Deadline First* (or EDF) algorithm is also denoted as *Earliest Termination First* (abbreviated as EXF) in this paper. In the process of sojourn time calculation, we only consider $selected$ tasks, i.e., $\mathbf{T}'$.

For the selected task set $\mathbf{T}'$ with $load_b' \leq 1$, the on-line scheduling process of CIC-VCUA is essentially EXF (we describe this in Section IV-D). Thus, we define $T.wcST$ to denote the worst-case sojourn time

of each task T, when the task set $\mathbf{T}'$ is scheduled by EXF. For a job J of task T, we denote its worst-case sojourn time as $J.FinT$. This is also the latest possible time for jobs of T to complete without causing any load increase or abortions of other jobs.

Figure 4 shows the time line of the job $J_{i,j}$, where t_i^0 and $J_{i,j}.X$ indicate the release and termination times of $J_{i,j}$, respectively, and $J_{i,j}.FinT$ is less than $J_{i,j}.X$.

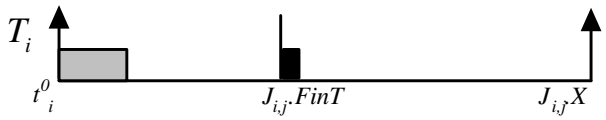


Fig. 4: A Job Example

For task set $\mathbf{T}'$ with $load'_b \leq 1.0$, our algorithm defaults to EXF. Hence, in order to find sojourn times under CIC-VCUA, we use the paradigm for finding sojourn times under EXF. Sojourn times under EXF can be determined using the notion of *deadline busy period*. A deadline d busy period is a busy period during which only jobs with absolute deadlines that are smaller than or equal to d execute [25], [26]. This is needed because, it allows us to determine how long it takes for each task to complete in the presence of other tasks.

First, it is necessary to calculate the synchronous busy period before calculating the individual deadline busy periods of tasks. The synchronous busy period, denoted L, is the interval of time, during which the processor is not idle. Further, if all the first job instances of tasks were to be released synchronously (the worst-case scenario), it would take L time units for all jobs to complete. Thus, the busy period bounds the individual completion times or deadline busy periods of tasks. The busy period is given by [27]:

$$L^0 = 0, \quad L^{m+1} = W\left(L^m\right), \text{ where } W(t) = \sum_{i=1}^{n'} \left\lceil \frac{t}{T_i} \right\rceil C_i.$$

The busy period is found when the iteration ends at $L^m = L^{m+1}$. After L is calculated, the individual deadline busy periods, L_i can be calculated. We need to determine which tasks will be executed before our target task. For a task T_i which arrives at time a, it is intuitive that before T_i's absolute termination time $a + X_i$, only tasks with termination times shorter than or equal to $a + X_i$ can be executed. The deadline busy period of a task T_i with an arrival time a is given by [26]:

$$L_i^0(a) = 0, \quad L_i^{m+1}(a) = W_i\left(a, L_i^m(a)\right) + \left(1 + \left\lfloor \frac{a}{X_i} \right\rfloor\right) C_i \tag{1}$$

where:

$$W_i(a,t) = \sum_{j \neq i, T_j.X \leq a + X_i} \texttt{min}\left(\left\lceil \frac{t}{X_j} \right\rceil, 1 + \left\lfloor \frac{a + X_i - X_j}{X_j} \right\rfloor \right) C_j$$

In calculating the deadline busy period L_i in Equation 1, the first term $W_i(a,t)$ calculates the higher priority workloads arriving in time window $[a,t]$ that have to be satisfied before executing task T_i, and the second term accounts for T_i's instances that have to be executed. The iterative computation will stop when $L_i^{m+1}(a) = L_i^m(a)$. Algorithm 1 shows the calculation of the maximum deadline busy period.

```
1: input: τ; output: (L_1, L_2, ...L_n) ;
2: Initialization : L_{n+1} := L;
3: for i = n down to 1 do
4:     let k be such that e_k ≤ L_{i+1} − C_i + X_i < e_{k+1};
5:     a := e_k − X_i;
6:     while L_i(a) ≤ a do
7:         let k be such that e_k ≤ L_i(a) − C_i + X_i < e_{k+1};
8:         a := e_k − X_i;
9:     L_i := L_i(a);
```

Algorithm 1: `maxDeadBusyP()`

Algorithm 1 uses the task list τ, which is ordered by non-decreasing termination times. The algorithm examines the list, starting from the task with the maximum X_i, which has the length L. Tasks that have absolute termination times shorter than that of T_i are inserted into proper termination time positions. Such positions are defined by $E = \bigcup_{i=1}^{n}(mX_i + X_i : m \geq 0) = (e1, e2...)$. After L_i is calculated, the bound for this task becomes L_i; it also becomes a bound for the next task in τ. The algorithm then moves down the list and calculates the maximum L_i for the next task, until all tasks are considered.

After determining L_i, the worst case sojourn time of a task T_i becomes $T_i.wcST = \texttt{max}(T_i.wcST(a))$ for $a \geq 0$, where $T_i.wcST(a) = \texttt{max}(C_i, L_i(a) - a)$. CIC-VCUA ensures that each job completes at its worst-case sojourn time after it is released, so that jobs can meet the bound constraint on consecutive completions.

For a task T_i, the finish time of the first job of the task is $J_{i,1}.FinT = T_i.wcST$. We can determine the finish times of the subsequent jobs of a task as $J_{i,j}.FinT = J_{i,j-1}.FinT + P_i$.

During schedule construction, CIC-VCUA "pushes back" the completion times of jobs further toward their termination times, as system load changes with variable execution times. In order to satisfy the bound constraint across the range of $load_b \leq 1.0$, the algorithm uses the worst-case sojourn times as predicted finish times. For $load_b > 1.0$, CIC-VCUA pushes job finish times such that they occur slightly before the

job termination times. Pushing finish times closer to termination times for higher loads is necessary for two reasons: First, the worst case sojourn time calculation becomes more unpredictable for different VCF shapes (since the calculation uses $C_i(t_i^0)$). Second, task sojourn times are already close to their termination times because of the load.

D. Dynamic UA Scheduling

After the initial static steps, CIC-VCUA selects the largest sub task set consisting of the highest PUD tasks, whose dynamic load will not cause a system overload. The algorithm then adopts the preemptive earliest termination time first (or EXF) scheduling policy which is optimal from the feasibility point of view [24].

At each arrival of a job $J_{i,j}$, its finish time $J_{i,j}.FinT$ is calculated from the task sojourn time $T_i.wcST$, the period P_i, and its predecessor's finish time. After we have $J_{i,j}.FinT = J_{i,j-1}.FinT + P_i$, the job is executed until only a very small amount of execution time of the job, denoted Δ, is left to be executed. At this time, if the absolute time is far from $J_{i,j}.FinT$, then job $J_{i,j}$ is preempted. Later, it will be selected again at $J.FinT$ to be completed.

Δ is a small quantity of time selected ($J.C \gg \Delta$) so that the interference caused by the execution of Δ time units (to finish the job) to other jobs is negligible. Δ is used to delay the completion of jobs, so that at their finish times $J.FinT$, they only need to run Δ units of time to finish. If two or more jobs have identical finish times, then Δ is also used to break the tie. When a job J's remaining execution time is only Δ, and it is preempted and will be resumed at $J.FinT$, we say that the job J is *ready to complete*.

Since tasks are preemptive, CIC-VCUA's scheduling events include: (1) a job's arrival; (2) the expiration of a time constraint such as the arrival of a TUF's termination time, when the CPU is idle; (3) a job's completion; (4) a resource request; and (5) a resource release.

To describe the algorithm, we define the following variables and auxiliary functions:

- $\mathcal{J}_r = \{J_1, J_2, \cdots, J_m\}$ is the current unscheduled job set; σ is the ordered output schedule. $J_k \in \mathcal{J}_r$ is a job. $J_k.X$ is its termination time; $J_k.FinT$ is its finish time, and $J_k.SEL$ is the job selection flag.
- `selectJob(`σ`)` returns a job to execute with the amount of time it will execute.
- `headOf(`σ`)` returns the first job in σ.
- `sortByPUD(`σ`)` returns σ ordered by non-increasing PUD. If two or more jobs have the same PUD, then the job(s) with the largest execution time will appear before any others with the same PUDs.

- `owner(`R`)` denotes the set of jobs that are currently holding resource R; `reqRes(`J`)` returns the resource requested by job J.
- `insert(`J`,`σ`,`I`)` inserts job J in the ordered list σ at the position indicated by index I; if there are already entries in σ at the index I, J is inserted before them. After insertion, the index of J in σ is I.
- `remove(`J`,`σ`,`I`)` removes job J from ordered list σ at the position indicated by index I; if J is not present at the position I in σ, the function takes no action.
- `lookup(`J`,`σ`)` returns the index value associated with the first occurrence of job J in the ordered list σ.
- `feasible(`σ`)` returns a boolean value indicating schedule σ's feasibility. For σ to be feasible, the predicted completion time of each job in σ must never exceed its termination time.

```
1:  input  : T = {T_1, ..., T_n}, J_r = {J_1, ..., J_m}
2:  output : selected job J_exe
3:  Initialization: t := t_cur, σ := ∅;
4:  for ∀J_k ∈ J_r do
5:      if feasible(J_k)=false then
6:          abort(J_k);
7:      else
8:          J_k.Dep := buildDep(J_k);
9:          J_k.PUD := calculatePUD (J_k);
10: σ_tmp :=sortByPUD(J_r);
11: for ∀J_k ∈ σ_tmp from head to tail do
12:     if J_k.PUD ≥ 0 and J_k.SEL = selected then
13:         σ :=insertByEXF(σ, J_k);
14:     else break;
15: J_exe:=selectJob(σ);
16: return J_exe
```

Algorithm 2: CIC-VCUA: Dynamic Part Description

A high-level description of CIC-VCUA is shown in Algorithm 2. At the beginning of each scheduling event, when CIC-VCUA is invoked at time t_{cur}, the algorithm first checks the feasibility of all the jobs in the current ready queue. If a job is infeasible, then it can be safely aborted (line 6). Otherwise, the algorithm constructs the job's dependency list (line 8), and then calculates its PUD (line 9).

At line 10, jobs are sorted by their PUDs, in a non-increasing order. In each step of the `for` loop from line 11 to line 14, the job with the largest PUD and its dependencies are inserted into σ by `insertByEXF()`. Thus, σ becomes a feasible schedule that is ordered by job termination times, in a non-decreasing order. Then, the `selectJob` () function finds a job in σ and returns it for execution.

For each job J, CIC-VCUA will compare Δ with $J.FinT - t_{cur}$ when the job has only Δ remaining

execution time units. If $\Delta < J.FinT - t_{cur}$, then job J is preempted, and another job may be selected for execution. Later, when $J.FinT - t_{cur} = \Delta$, job J will preempt the current running task, so that it can finish at $J.FinT$.

Such monitoring, preemption, and resumption are realized by the procedure `selectJob()`. This procedure selects a job with the earliest finish time from σ. If this job is not *ready to complete*, then it ensures that the job executes $J.C - \Delta$ time units. Otherwise, `selectJob()` runs that job to completion. After finishing such jobs, the algorithm seeks another job to execute.

E. Resource and Deadlock Handling

Before CIC-VCUA can compute job partial schedules, the dependency chain of each job must be determined. This is described in Algorithm 3.

```
1: input: Job J_k; output: J_k.Dep ;
2: Initialization : J_k.Dep := J_k; Prev := J_k;
3: while reqRes(Prev) ≠ ∅ ⋀ owner(reqRes(Prev)) ≠ ∅ do
4:     J_k.Dep :=owner(reqRes(Prev)) ·J_k.Dep;
5:     Prev := owner(reqRes(Prev));
```

Algorithm 3: `buildDep()`

Algorithm 3 follows the chain of resource request and ownership. For convenience, the input job J_k is also included in its own dependency list. Each job J_l other than J_k in the dependency list has a successor job that needs a resource which is currently held by J_l. Algorithm 3 stops either because a predecessor job does not need any resource or the requested resource is free. Note that "**.**" denotes an append operation. Thus, the dependency list starts with J_k's farthest predecessor and ends with J_k.

To handle deadlocks, we consider a deadlock detection and resolution strategy, instead of a deadlock prevention or avoidance strategy. Our rationale for this is that deadlock prevention or avoidance strategies normally pose extra requirements—for example, resources must always be requested in ascending order of their identifiers.

Further, restricted resource access operations that can prevent or avoid deadlocks, as done in many priority-based resource access protocols, are not appropriate for the class of application systems that we focus here. For example, the Priority Ceiling protocol [22] assumes that the highest priority of jobs accessing a resource is known. Likewise, the Stack Resource policy [28] assumes preemptive "levels" of threads *a priori*. Such assumptions are too restrictive for our application systems—the resources that

will be needed, the length of time for which they will be needed, and the order of accessing them are all statically unknown.

Recall that we are assuming a single-unit resource request model. For such a model, the presence of a cycle in the resource graph is the necessary *and* sufficient condition for a deadlock to occur. Thus, the complexity of detecting a deadlock can be mitigated by a straightforward cycle-detection algorithm.

```
1:  input: Requesting job J_k, t_cur;
2:  /* deadlock detection */;
3:  Deadlock := false;
4:  J_l := owner(reqRes(J_k));
5:  while J_l ≠ ∅ do
6:      J_l.LoPUD := J_l.U(J_l.FinT) / J.C(t_cur);
7:      if J_l = J_k then
8:          Deadlock := true;
9:          break;
10:     else
11:         J_l := owner(reqRes(J_l));
12: /* deadlock resolution if any */;
13: if Deadlock = true then
14:     abort(The job J_m with the minimal LoPUD in the cycle);
```

Algorithm 4: Deadlock Detection and Resolution

The deadlock detection and resolution algorithm (Algorithm 4) is invoked by the scheduler whenever a job requests a resource. Initially, there is no deadlock in the system. By induction, it can be shown that a deadlock can occur if and only if the edge that arises in the resource graph due to the new resource request lies on a cycle. Thus, it is sufficient to check if the new edge resulting from the job's resource request produces a cycle in the resource graph.

To resolve the deadlock, some job needs to be aborted. If a job J_l were to be aborted, then its timeliness utility is lost. To minimize such loss, we compute the Local PUD (or LoPUD) of each job at t_{cur}. A job's LoPUD is defined as the utility that the job can potentially accrue by itself at the current time, if it were to continue its execution. The algorithm aborts the job with the minimal LoPUD in the cycle to resolve a deadlock. Before aborting the job, the resources held by the job is released and returned to a consistent state.

F. Manipulating Partial Schedules

The `calculatePUD()` algorithm (Algorithm 5) accepts a job J_k and its dependency list, and determines J_k's PUD. It assumes that jobs in $J_k.Dep$ finish at their predicted finish times $J.FinT$ from the current position in the schedule, while following the dependencies.

```
1: input: J_k; output: J_k.PUD;
2: Initialization: t_c := 0, U := 0;
3: for ∀J_l ∈ J_k.Dep, from head to tail do
4:     t_c := t_c + J_l.C(t_cur);
5:     U := U + J_l.U(J_l.FinT);
6: J_k.PUD := U / t_c;
7: return J_k.PUD;
```

Algorithm 5: `calculatePUD()`

To compute J_k's PUD, CIC-VCUA considers each job J_l that is in J_k's dependency chain $J_k.Dep$, which needs to be completed before executing J_k, since they hold resources that J_k needs. (Note that `buildDep()` includes J_k's dependents and J_k in $J_k.Dep$.) First, the algorithm calculates the total utility U that can be accrued by executing J_k and its dependents and completing them at their respective finish times $J.FinT$. The total execution times of J_k and its dependents is aggregated in the variable t_c. `calculatePUD` () determines J_k's PUD as U/t_c (line 6).

The details of `insertByEXF()` in line 13 of Algorithm 2 are shown in Algorithm 6. `insertByEXF()` updates the tentative schedule σ by attempting to insert each job along with all of its dependents in σ. The updated σ is an ordered list of jobs, where each job is placed according to the termination time that it should meet. Note that the time constraint that a job should meet is not necessarily the job's termination time. In fact, the index value of each job in σ is the actual time constraint that the job must meet.

```
1: input  : J_k and an ordered job list σ;
2: output : the updated list σ;
3: if J_k ∉ σ then
4:     copy σ into σ_tent: σ_tent := σ;
5:     insert(J_k, σ_tent, J_k.X);
6:     CuXT = J_k.X;
7:     for ∀J_l ∈ {J_k.Dep − J_k} from tail to head do
8:         if J_l ∈ σ_tent then
9:             XT=lookup(J_l, σ_tent);
10:            if XT < CuXT then continue;
11:            else remove(J_l, σ_tent, XT);
12:        CuXT:=min(CuXT, J_l.X);
13:        insert(J_l, σ_tent, CuXT);
14:    if feasible(σ_tent) then
15:        σ := σ_tent;
16: return σ;
```

Algorithm 6: `insertByEXF()`

A job may need to meet an earlier termination time in order to enable another job to meet its time constraint. Whenever a job J is considered for insertion in σ, it is scheduled to meet its own termination time. However, J's dependents must execute before J can execute, and therefore, must precede it in the schedule. The index values of the dependencies can be changed with `insert()` in line 13 of Algorithm 6.

The variable $CuXT$ is used to keep track of this information. Initially, it is set to be the termination time of job J_k, which is tentatively added to the schedule (line 6, Algorithm 6). Thereafter, any job in $J_k.Dep$ with a later time constraint than $CuXT$ is required to meet $CuXT$. If, however, a job has a tighter termination time than $CuXT$, then it is scheduled to meet the tighter termination time, and $CuXT$ is advanced to that time since all jobs left in $J_k.Dep$ must complete by then (lines 12–13, Algorithm 6). Finally, if this insertion produces a feasible schedule, then the jobs are included in the schedule; otherwise, the schedule is not changed (lines 14–15).

It is worth noting that the real time constraint that a job has to meet is its finish time $J.FinT$. The procedure `insertByEXF()` resolves resource dependencies and, accordingly, may change the order of task execution.

G. Selecting a Job for Execution

The procedure `selectJob()` (Algorithm 7) determines the job that will be executed, as well as the amount of time for which it needs to be executed.

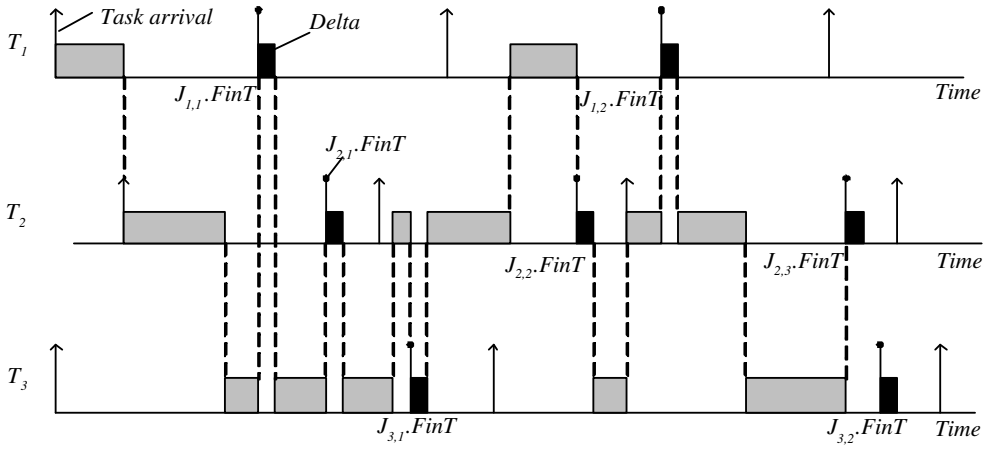


Fig. 5: Example of a Task Set

At the beginning of the algorithm, the job with the earliest finish time in σ, denoted $Earliest$, is found. `selectJob()` starts by checking whether the currently running job $CurRunning$ holds resources (line 3). If so, the algorithm ensures that this job is executed so that the held resources are freed. Then, $Earliest$ is selected to be the running task, if its finish time has arrived (line 6) and the algorithm returns with $J_{exe} = Earliest$.

If line 6 cannot determine the job J_{exe} that needs to complete, then the algorithm checks if a previous job, $Prev$, exists (line 10). If $Prev$ exists, then it means that $Prev$ is currently holding resources, and

```
1:  input: σ, CurRunning, Prev, t_cur; output: J_exe ;
2:  Initialization : Earliest := minFinT(σ); J_i = NULL;
3:  if CurRunning.HeldRes ≠ ∅ then
4:      Prev := CurRunning;
5:  /* a job's about to finish */;
6:  if Earliest.FinT = t_cur then
7:      J_exe := Earliest;
8:      setExeTimer(Δ);
9:      return J_exe
10: if Prev ≠ NULL then
11:     J_exe := Prev;
12:     setExeTimer(J_exe.C(t_cur) − Δ);
13:     return J_exe
14: for ∀J_k ∈ σ from head to tail do
15:     if J_k.C(t_cur) > Δ then
16:         J_i := J_k;
17:         break;
18: /* is there a job to run?*/;
19: if J_i = ∅ then
20:     J_exe := NULL;
21:     setExeTimer(Earliest.FinT);
22: else
23:     J_exe := J_i;
24:     setExeTimer(J_exe.C(t_cur) − Δ);
25: return J_exe
```

Algorithm 7: `selectJob()`

has to be executed to release those resources. However, at the same time, jobs that are *ready to complete* must be finished without delay. Therefore, jobs holding resources can only be preempted by ready jobs and $Prev$'s execution has to follow that of the ready jobs. So, lines 10–13 ensure that $Prev$ is favored for execution after a job completes.

If line 6 cannot return J_{exe} and no $Prev$ exists, then the algorithm seeks to select a job that can be executed in σ (lines 14–17). The first job with $J.C(t_{cur}) > \Delta$ in σ is selected to execute until its remaining execution time is only Δ (line 24). With task arrivals and completions, the contents and the order of σ change, in terms of both resource dependencies and finish times. However, the algorithm ensures that each job J is selected to complete at its finish time $J.FinT$. If no tasks can be found to execute at line 19, then the algorithm idles the processor until either the earliest finish time or the arrival of a new job.

An example of how CIC-VCUA executes jobs is shown in Figure 5. Upward arrows indicate both job arrivals and termination times, and black boxes denote Δ. In this example, jobs $J_{1,1}$ and $J_{3,1}$ arrive at the same time. However, since $J_{1,1}$'s finish time is earlier, CIC-VCUA selects $J_{1,1}$ for execution and creates a preemption point at time $J_{1,1}.C(t_{cur}) - \Delta$. As $J_{1,1}$ is preempted, T_2 arrives with an earlier finish time than that of $J_{3,1}$ and runs until $J_{2,1}.C(t_{cur}) - \Delta$. After $J_{2,1}$'s preemption, $J_{3,1}$ is executed, but it gets preempted because $J_{1,1}$'s finish time $J_{1,1}.FinT$ arrives and $J_{1,1}$ executes to completion. Then, $J_{3,1}$ resumes, however

it is again preempted to let $J_{2,1}$ finish. After this, $J_{3,1}$ resumes and completes at its finish time.

H. Asymptotic Time Complexity

To analyze the complexity of CIC-VCUA (Algorithm 2), we consider a ready queue of n jobs and a maximum of r resources. In the worst-case, `buildDep()` will build a dependency list with a length n; so the `for`-loop from line 4 to 9 will be repeated $O(n^2)$ times in the worst-case. `sortByPUD()`'s complexity is $O(n \log n)$.

Complexity of the `for`-loop body starting from line 11 is dominated by `insertByEXF()` (Algorithm 6). Its complexity is dominated by the `for`-loop (line 7–13, Algorithm 6), which requires $O(n \log n)$ time since the loop will be executed no more than n times, and each execution requires $O(\log n)$ time for `insert()`, `remove()` and `lookup()` operations on the tentative schedule. Therefore, CIC-VCUA's worst-case complexity is $2 \times O(n^2) + O(n \log n) + n \times O(n \log n) = O(n^2 \log n)$.

CIC-VCUA's asymptotic cost is similar to that of many past UA scheduling algorithms such as [6]–[8]. Our prior implementation experience with UA scheduling at the middleware-level have shown that the overheads are in the magnitude of sub-milliseconds [29] (sub-microsecond overheads may be possible at the kernel-level). We anticipate a similar overhead magnitude for CIC-VCUA (on a similar platform).

As mentioned before, systems such as AESA that we consider in this paper are distinguished by their relatively long execution time magnitudes—e.g., milliseconds to seconds, or seconds to minutes. Thus, although CIC-VCUA has a higher overhead than traditional real-time scheduling algorithms, this high cost is justified for applications with longer execution time magnitudes such as those that we focus here (of course, this high cost cannot be justified for every application).[2]

V. Algorithm Properties

A. Non-Timeliness Properties

We now discuss CIC-VCUA's non-timeliness properties, i.e., deadlock-freedom, correctness, and mutual exclusion. CIC-VCUA respects resource dependencies by ensuring that the job selected for execution can execute immediately. Thus, no job is ever selected for normal execution if it is resource-dependent on some other job.

[2]When UA scheduling is desired with low overhead, solutions and tradeoffs exist. These include linear-time stochastic UA scheduling [30], UA scheduling with non-blocking synchronization for concurrent, mutually exclusive resource sharing [31], and using special-purpose hardware accelerators for UA scheduling (analogous to floating-point co-processors) [32].

Theorem 1: CIC-VCUA ensures deadlock-freedom.

Proof: A cycle in the resource graph is the sufficient *and* necessary condition for a deadlock in the single-unit resource request model. CIC-VCUA does not allow such a cycle by deadlock detection and resolution; so it is deadlock free. ■

Lemma 2: In `insertByEXF()`'s output, all the dependents of a job must execute before the job can execute, and therefore, must precede it in the schedule.

Proof: `insertByEXF()` maintains an output queue that is ordered by job termination times, while respecting resource dependencies. Consider job J_k and its dependent J_l. If $J_l.X$ is earlier than $J_k.X$, then J_l will be inserted before J_k in the schedule. If $J_l.X$ is later than $J_k.X$, then $J_l.X$ is advanced to be $J_k.X$ by the operation with $CuXT$. According to the definition of `insert()`, after advancing the termination time, J_l will be inserted before J_k. ■

Theorem 3: When a job J_k that requests a resource R is selected for execution by CIC-VCUA, J_k's requested resource R will be free. We call this, CIC-VCUA's correctness property.

Proof: From Lemma 2, the output schedule σ is correct. Thus, CIC-VCUA is correct. ■

Thus, if a resource is not available for a job J_k's request, jobs holding the resource will become J_k's predecessors. We present CIC-VCUA's mutual exclusion property by a corollary.

Corollary 4: CIC-VCUA satisfies mutual exclusion constraints in resource operations.

B. Timeliness Properties

We now consider CIC-VCUA's timeliness properties, and compare the algorithm with other algorithms. Specifically, we consider the following two conditions: (1) a set of independent periodic tasks subject to step TUFs; and (2) sufficient processor cycles exist for meeting all task termination times—i.e., there is no overload, and $load_b \leq 1$.

Theorem 5: Under conditions (1) and (2), a schedule produced by EDF [3] is also produced by CIC-VCUA, yielding equal total utilities. Not coincidentally, this is simply a termination time-ordered schedule.

Proof: We prove this by examining Algorithm 2. For periodic tasks, during non-overload situations, σ from Algorithm 2 is termination time-ordered, due to the properties of the procedure `insertByEXF()`. The termination time that we consider is analogous to a deadline in [3]. As proved in [3], [24], a deadline-ordered schedule is optimal (with respect to meeting all deadlines) for preemptive task sets when there are no overloads. Thus, σ yields the same total utility as preemptive EDF. ■

Some important corollaries about CIC-VCUA's timeliness behavior during non-overload situations can be deduced from EDF's optimality [33].

Corollary 6: Under conditions (1) and (2), CIC-VCUA always meet all task termination times.

With the previous theorems and corollaries, we derive algorithm properties in terms of CIC-VCUA's scheduling objective.

Theorem 7: CIC-VCUA assures that the maximum time interval between any two consecutive, successful completions of jobs of a task does not exceed the task period.

Proof: Let $J_{i,j}.ST$ and $J_{i,j+1}.ST$ be the sojourn times of two consecutive, successfully completed jobs $J_{i,j}$ and $J_{i,j+1}$ of task T_i, respectively. Also, let $T_i.wcST$ be the worst-case sojourn time of task T_i (and of all its jobs). Under CIC-VCUA, the maximum time interval between completions of $J_{i,j}$ and $J_{i,j+1}$ will be equal to $P_i + J_{i,j+1}.ST - J_{i,j}.ST$, i.e., $J_{i,j+1}.FinT - J_{i,j}.FinT$. So, in order to have a maximum interval bound of P_i, we should have $J_{i,j+1}.ST = J_{i,j}.ST$.

We know that the first job of T_i has $J_{i,1}.FinT = J_{i,1}.ST = T_i.wcST$. Under CIC-VCUA, the consecutive completions of the following jobs will keep $J_{i,j+1}.ST = J_{i,j}.ST = T_i.wcST$, i.e., sojourn times of all jobs are equal to the task's worst-case sojourn time. Therefore, $J_{i,j+1}.FinT - Ji, j.FinT = P_i$. So for any task, the time interval between two consecutive, successful completions of its jobs does not exceed the length of the task period. ■

Following theorem 7, during under-loads, every job of task T_i completes within their completion time bound $T_i.wcST$ after its arrival. Other jobs of other tasks abide by the same rule. During system overloads, when $load_b > 1$, CIC-VCUA dynamically selects tasks with the highest PUDs among the task set, until total $load'_b \leq 1$. Therefore, the bound on consecutive job completions still holds for the selected sub task set.

VI. Experimental Results

We experimentally evaluated CIC-VCUA through a detailed simulation study. We first describe our experimental settings, and then report our results.

A. Experimental Settings

We selected task sets with 16 tasks in three applications, denoted A_1, A_2, and A_3. Task parameters are summarized in Table I. Within each range, the period P is uniformly distributed. The synthesized task sets simulate the varied mix of short and long periods.

TABLE I: Task Settings

Applications	♯ Tasks	Period	U^{max}	$\langle k, C_o \rangle$ (VCF $= \pm k \times t + C_o$)
A_1	4	$22 \sim 28$	$[50, 70]$	$\langle 0\text{–}0.1, E(C_o) \rangle$
A_2	18	$50 \sim 70$	$[300, 400]$	$\langle 0\text{–}0.1, E(C_o) \rangle$
A_3	8	$2.4 \sim 9.6$	$[1, 10]$	$\langle 0\text{–}0.1, E(C_o) \rangle$

The U^{max}s of the TUFs in A_1, A_2, and A_3 are uniformly generated within each range. We define a linearly increasing $\text{VCF} = k \times t + C_o$, a linearly decreasing $\text{VCF} = -k \times t + C_o$, and a constant $\text{VCF} = C_o$ for each task. The parameter k is uniformly generated within the range $[0, 0.1]$. We change the mean value of C_o, and generate normally-distributed values to adjust the system load $load_b$. In all of our experiments, the Δ value is set to be 2×10^{-4}. Finish times of tasks $J.FinT$ are pushed to their termination times when $load_b > 0.9$, in order to avoid the unpredictability of sojourn time calculations.

B. Performance on Completion Interval

We assign to each task a step TUF, and first consider CIC-VCUA's performance on scheduling objective (1). For the 16 tasks, we vary $load_b$ from less than 0.1 to larger than 1.8, and evaluate the maximum interval between any two consecutive, successful completions of jobs of each task, and of the whole task set (containing all tasks). We define the former as the maximum *intra-task* completion interval, and the latter as the maximum *inter-task* completion interval.

We consider two classes of VCFs for the task sets: *homogeneous* and *heterogeneous*. A task set that consists of tasks with only one type of VCF shapes is referred to as a *homogeneous set*. On the other hand, in a *heterogeneous set*, tasks of the set can have any VCF shapes specified in Table I. In the following experiments, we consider step TUFs.

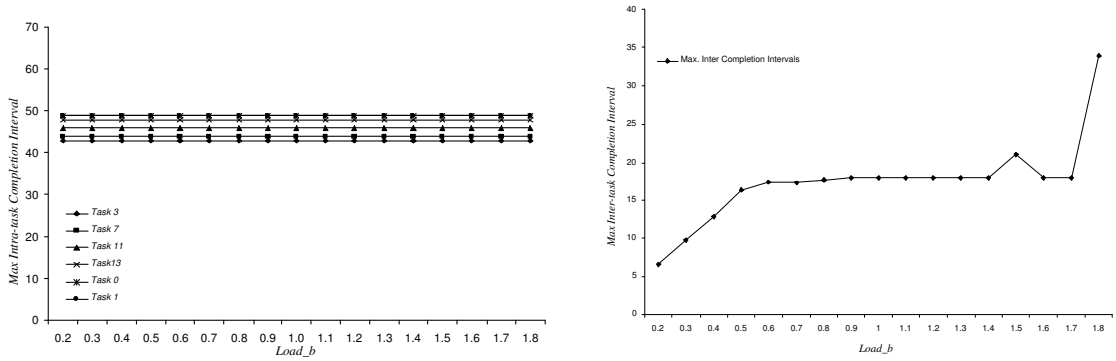
1) Homogeneous VCFs: In the experiments of this section, we use constant VCFs for all tasks. Figure 6 shows the maximum intra- and inter-task completion intervals, as $load_b$ varies. In Figure 6(a), we only show 6 tasks as examples selected from the task set to study their maximum intra-task completion interval.

TABLE II: Tasks and Their Periods for Homogeneous Set

Task ID	0	1	3	7	11	13
Period	49	49	43	44	46	48

To validate Theorem 7, we show periods of the selected tasks from Figure 6(a) in Table II. From Figure 6(a) and Table II, we observe that in all $load_b$ regions, the maximum intra-task completion interval

of each task is less than or equal to the length of its period. During overloads, the selected tasks are labeled as *selected* since they have high PUDs. So they can always satisfy their bound constraints. Therefore, plots in Figure 6(a) validate Theorem 7.



(a) Intra-Task (b) Inter-Task

Fig. 6: Maximum Intra- and Inter-Task Completion Interval for Homogeneous Set, Constant VCFs, Step TUFs

As a comparison, we also study the maximum inter-task completion interval of the whole task set in Figure 6(b). The minimum period of the task set is 3. From the figure, we observe that during under-loads, the maximum inter-task completion interval is less than 20, and during overloads, the maximum inter-task completion interval may exceed 20 in order to satisfy intra-task completion bounds.

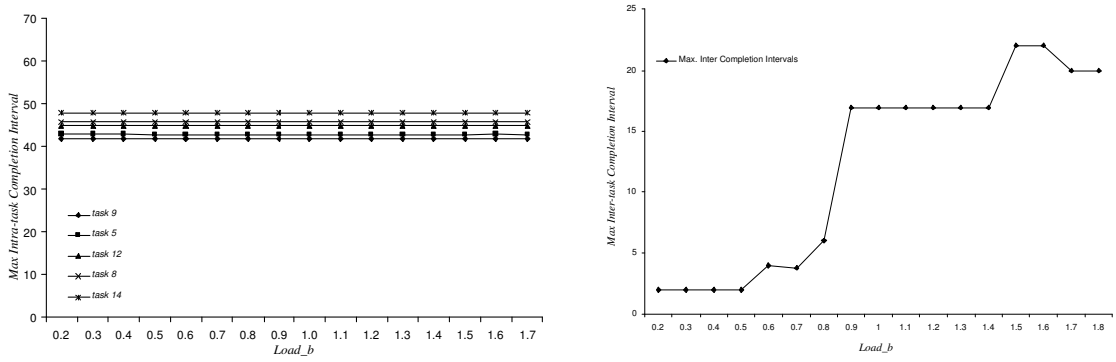
Experiments for homogeneous sets with monotonically increasing and decreasing VCFs under various TUF shapes yield results similar to those shown in Figure 6. These are omitted here for brevity, however they can be found in [34].

2) Heterogeous VCFs: For the experiments in this section, we generate random VCFs for each task. The shapes we use for VCFs are described in Section III-D. Figure 7 shows the maximum intra- and inter-task completion intervals, as $load_b$ varies. In Figure 7(a), we selected 5 tasks to study their maximum intra-task completion interval. The periods of these tasks are shown in Table III.

TABLE III: Tasks and Their Periods for Heterogeneous Set

Task ID	5	8	9	12	14
Period	46	43	48	45	42

From Figure 7(a), we again observe that the maximum intra-task completion interval of each task is less than or equal to the length of its period, in all $load_b$ regions. During overloads, similar to the homogeneous set scenario, *skipped* tasks with low PUDs never get a chance to execute. Hence, plots in Figure 7(a) also validate Theorem 7.



(a) Intra-Task (b) Inter-Task

Fig. 7: Maximum Intra- and Inter-Task Completion Interval for Heterogenous Set

Figure 7(b) shows the maximum inter-task completion interval of the whole task set. The minimum period of the task set is 3. We observe results similar to that of the homogeneous set scenario. Results for heterogeneous VCFs under various TUF shapes show consistent results. These are again omitted here for brevity, but can be found in [34].

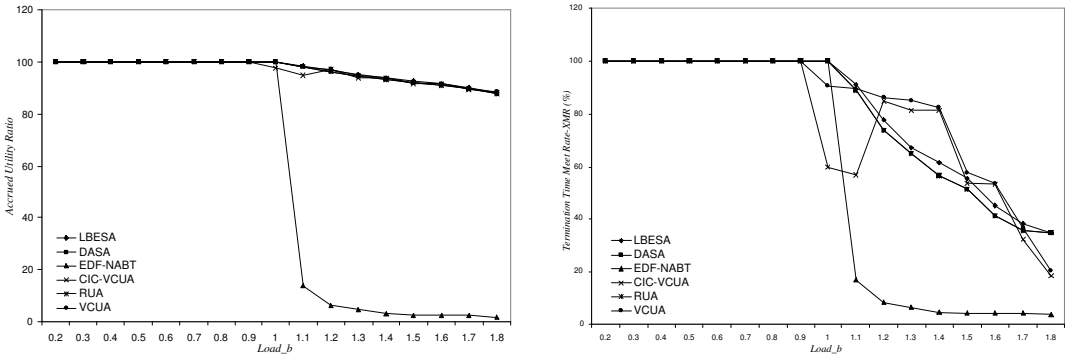
C. Performance on Utility Accrual

We now evaluate CIC-VCUA's performance on the scheduling objective (2). In these experiments, we consider constant VCFs. For such VCFs, CIC-VCUA can be compared with other UA algorithms (that cannot deal with non-constant VCFs and varying execution times).

We consider step and decreasing TUFs. Our first experiments compare CIC-VCUA with RUA [8], DASA [7], LBESA [6], VCUA [35], and EDF without abortion (or EDF-NABT) [3] to evaluate performance under step TUFs (all these algorithms allow step TUFs). We then compare CIC-VCUA with RUA, DASA, VCUA and LBESA, under decreasing TUFs (these algorithms allow decreasing TUFs).

Figures 8 and 9 show the accrued utility ratio (or AUR) and termination time meet rate (or XMR) of the algorithms as $load_b$ increases. AUR is the ratio of the total accrued utility to the total maximum utility, and XMR is the ratio of the number of jobs meeting their termination times to the total number of job releases.

Figure 8 shows the AUR and XMR of the algorithms under step TUFs. From Figure 8(a), we observe that CIC-VCUA has almost the same AUR as that of DASA, RUA, and LBESA. However, from Figure 8(b), we observe that CIC-VCUA suffers higher termination time misses than other algorithms during high loads. This is because, CIC-VCUA statically labels some tasks as $skipped$, so that it can satisfy the completion interval bounds.

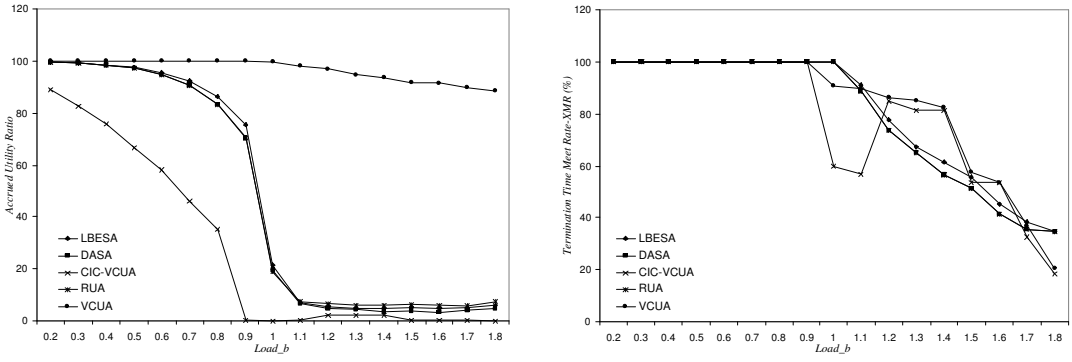


(a) AUR (b) XMR

Fig. 8: AUR and XMR of CIC-VCUA and other UA Algorithms, Constant VCFs, Step TUFs

The AUR and XMR of the algorithms under decreasing TUFs is shown in Figures 9(a) and 9(b), respectively. We observe that CIC-VCUA yields less AUR and less XMR than other algorithms for decreasing TUFs, and much less so than under step TUFs. This is clearly due to the algorithm's procrastination of jobs to satisfy the completion time interval bound—CIC-VCUA's primary scheduling objective. None of the other algorithms are designed to satisfy the completion time interval bound (see Section VI-D for results that illustrate this). Maximizing AUR is only CIC-VCUA's secondary objective.

Thus, job procrastination results in reduced AUR and XMR for CIC-VCUA with respect to other algorithms. Further, this reduction is more significant under decreasing TUFs than under step TUFs, clearly because earlier completion results in greater AUR under decreasing TUFs but not under step TUFs.



(a) AUR (b) XMR

Fig. 9: AUR and XMR of CIC-VCUA and other UA Algorithms, Constant VCFs, Decreasing TUFs

Figures 8 and 9 show that, when $load_b > 0.9$, CIC-VCUA starts to miss termination times and its XMR drops, but its AUR drops much more slowly, since tasks with higher PUDs are statically selected. Further,

AURs and XMRs in Figure 8 validate Theorem 5 and Corollary 6.

Our experiments with monotonically increasing and decreasing VCFs yield similar results to those shown in Figures 8 and 9. Those are again omitted here, but can be found in [34].

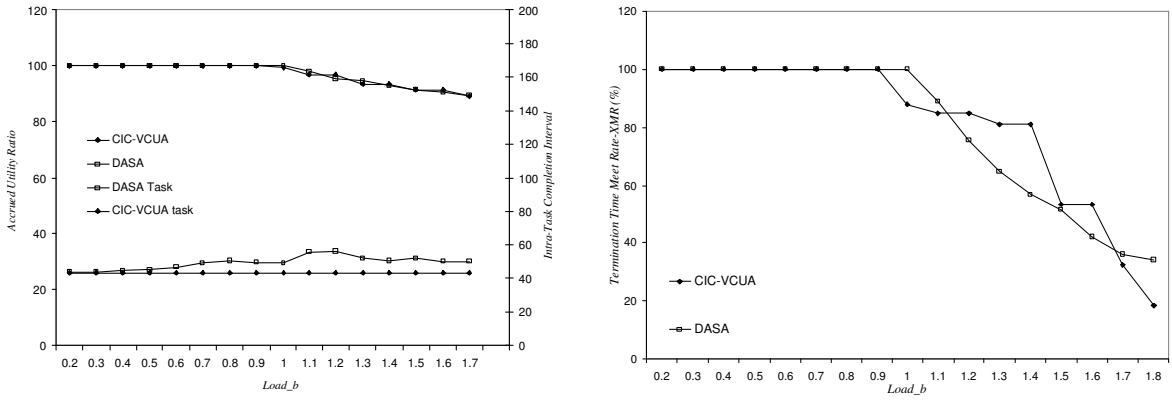
D. Results under Resource Dependency

To construct dependent task sets, we consider task sets where jobs may randomly request and release resources from an available set of resources during their life spans. The resource request and release times are uniformly distributed within a job's life cycle before the job is *ready to complete*. That is, resource request and release are serviced before the job's remaining execution time is only Δ. We conducted experiments on task sets and five shared resources. Table IV displays 6 tasks that we selected to study their maximum inter- and intra-task completion intervals.

TABLE IV: Tasks and Their Periods for Resource Dependency Experiments

Task ID	7	11	13	4	12	2
Period	44	46	48	49	50	50

For these experiments, we compare CIC-VCUA with DASA, since DASA is a UA scheduling algorithm that allows resource dependencies, and exhibits good performance. Figure 10 shows the results. With our experimental settings, we have only limited performance loss in our simulation, but we expect more performance drop with larger task sets and more resources.



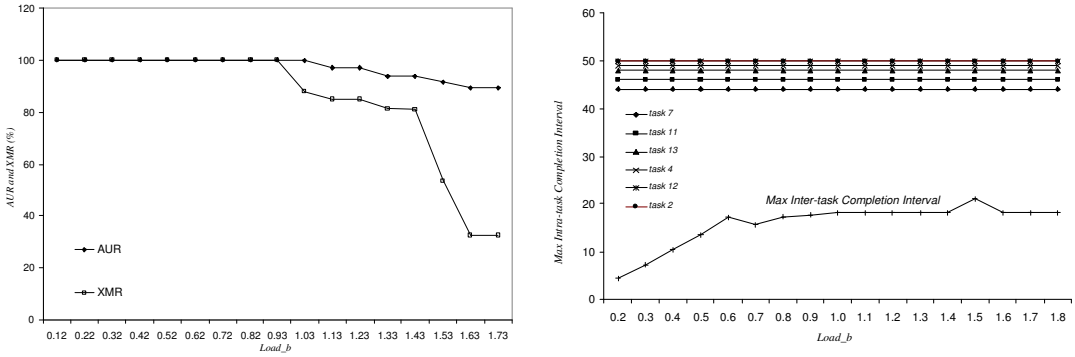
(a) AUR (b) XMR

Fig. 10: CIC-VCUA vs DASA under Resource Dependency, Constant VCFs, Step TUFs

Figure 10(a) shows both AURs (on the left Y-axis), and the intra-task completion intervals (on the right Y-axis) of a randomly selected task, for CIC-VCUA and DASA, as $Load_b$ varies. In terms of AUR,

CIC-VCUA performs as well as DASA. Also, we observe from Figure 10(a) that the intra-task completion interval of DASA increases as load increases, and it exceeds the bound of one period. However, CIC-VCUA maintains the intra-task completion interval as a constant, equal to the task's period, under different system loads. Additionally, Figure 10(b) shows the XMR comparison of both algorithms. During overloads, in terms of XMR, DASA and CIC-VCUA exhibit different behaviors because of their different schedule construction process.

Figure 11 shows the performance of CIC-VCUA under decreasing VCFs and step TUFs. As Figure 11(a) shows, CIC-VCUA exhibits good AUR even for task sets with resource dependencies. The XMR decrease is due to the static selection which favors high PUD tasks. Figure 11(b) shows the maximum intra-task and inter-task completion intervals of the $selected$ tasks. Clearly, CIC-VCUA bounds the intra-task completion interval to be one period.

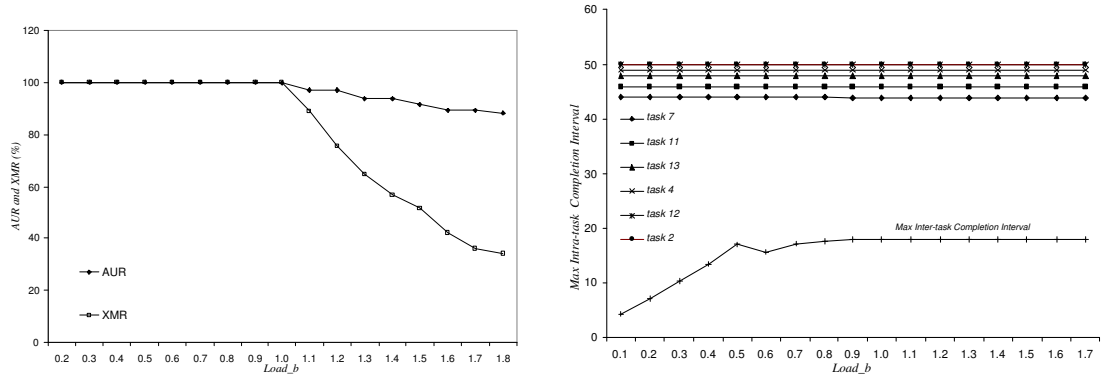


(a) AUR and XMR (b) Completion intervals

Fig. 11: CIC-VCUA Performance under Resource Dependency, Decreasing VCFs, Step TUFs

Figure 12 shows CIC-VCUA's performance under increasing VCFs and step TUFs. As Figure 12(a) shows, the algorithm AUR and XMR are similar to the case of non-increasing VCFs. Figure 12(b) shows the inter- and intra-task completion intervals of $selected$ tasks. Again, we observe that CIC-VCUA bounds the intra-task completion interval to one period.

From Figure 12, we observe that the algorithm performance under resource dependencies is similar to that under no resource dependencies. However, there is a small performance loss due to mutual exclusion requirements. The higher the number of shared resources, the greater is this performance loss. This is because, CIC-VCUA respects resource dependencies in scheduling, which in the worst case may cause jobs to be executed in the reverse order of PUDs or termination times. With such dependent task sets, the algorithm suffers performance losses, especially during high loads.



(a) AUR and XMR (b) Completion intervals

Fig. 12: CIC-VCUA Performance under Resource Dependency, Increasing VCFs, Step TUFs

Our experiments with monotonically increasing and decreasing VCFs under various other TUF shapes yield similar results to those shown in Figures 11 and 12 [34].

VII. Conclusions, Future Work

In this paper, we present a real-time scheduling algorithm called CIC-VCUA that focuses on the problem space intersecting UA scheduling and variable cost scheduling. The algorithm considers tasks which are subject to TUF time constraints and mutual exclusion constraints on shared non-CPU resources, and whose execution times are functions of their starting times. CIC-VCUA considers a two-fold objective: (1) bound the maximum interval between any two consecutive, successful completion of jobs of a task to the task's period, and (2) maximize the system's total utility, while satisfying all resource dependencies. This problem can be shown to be NP-hard. CIC-VCUA heuristically solves the problem in polynomial-time. We establish that CIC-VCUA achieves optimal total utility during under-loads, and tight upper bounds on inter- and intra-task completion times. Our experimental studies confirm the algorithm's effectiveness and superiority.

This paper only scratched the surface of the VCF scheduling problem; so many problems are open for further research. Immediate research directions include relaxing some of our task model assumptions. For example, our work assumed that dwell jobs are arbitrarily preemptible. This is not generally true of AESA systems, as preempting a dwell is sometimes expensive. (Our preliminary work [35] that led to this work considered a fully non-preemptive task model, which is also restrictive.) Thus, CIC-VCUA can be extended for a task model, which includes non-preemption and preemption with non-negligible cost.

Further, timescales associated with VCFs and TUFs can vary widely in AESA systems. For example,

the TUFs associated with each dwell may have termination times in the range of tens to hundreds of milliseconds; the VCFs may only change significantly over the course of tens or hundreds of seconds. This facet of the model can be exploited in future work.

Our periodic task arrival model can also be relaxed—e.g., to the unimodal arbitrary arrival model (or UAM) [31]. UAM embodies a "stronger" adversary than most arrival models.

ACKNOWLEDGEMENTS

This work was supported by the US Office of Naval Research under Grant N00014-00-1-0549 and The MITRE Corporation under Grant 52917. Co-author E. Douglas Jensen's contributions to this work were sponsored by the MITRE Corp. Technology Program. The authors thank Dr. Raymond Clark of The MITRE Corporation for his inputs on the VCF problem. Preliminary results of this work appeared in [35].

REFERENCES

[1] R. K. Clark, E. D. Jensen, and N. F. Rouquette, "Software Organization to Facilitate Dynamic Processor Scheduling," in *Proceedings of IEEE Parallel and Distributed Processing Symposium*, April 2004.

[2] R. Clark, E. D. Jensen, A. Kanevsky, J. Maurer, P. Wallace, T. Wheeler, Y. Zhang, D. Wells, T. Lawrence, and P. Hurley, "An Adaptive, Distributed Airborne Tracking System," in *Proceedings of The IEEE Workshop on Parallel and Distributed Systems*, ser. LNCS, vol. 1586. Springer-Verlag, April 1999, pp. 353–362.

[3] W. Horn, "Some Simple Scheduling Algorithms," *Naval Research Logistics Quaterly*, vol. 21, pp. 177–185, 1974.

[4] E. D. Jensen, C. D. Locke, and H. Tokuda, "A Time-Driven Scheduling Model for Real-Time Systems," in *Proceedings of IEEE Real-Time Systems Symposium*, December 1985, pp. 112–122.

[5] D. P. Maynard, S. E. Shipman, R. K. Clark, J. D. Northcutt, R. B. Kegley, B. A. Zimmerman, and P. J. Keleher, "An Example Real-Time Command, Control, and Battle Management Application for Alpha," Department of Computer Science, Carnegie Mellon University, Tech. Rep., December 1988, Archons Project Technical Report 88121.

[6] C. D. Locke, "Best-Effort Decision Making for Real-Time Scheduling," Ph.D. dissertation, Carnegie Mellon University, 1986, CMU-CS-86-134, http://www.real-time.org (last accessed: June 22, 2005).

[7] R. K. Clark, "Scheduling Dependent Real-Time Activities," Ph.D. dissertation, Carnegie Mellon University, 1990, CMU-CS-90-155, http://www.real-time.org (last accessed: June 22, 2005).

[8] H. Wu, B. Ravindran, E. D. Jensen, and U. Balli, "Utility Accrual Scheduling under Arbitrary Time/utility Functions and Multi-unit Resource Constraints," in *Proceedings of 10th International Conference on Real-Time and Embedded Computing Systems and Applications (RTCSA)*, August 2004, pp. 80–98.

[9] J. W. S. Liu, W.-K. Shih, K.-J. Lin, R. Bettati, and J.-Y. Chung, "Imprecise Computations," *Proceedings of the IEEE*, vol. 82, no. 1, pp. 83–94, Janurary 1994.

[10] J. K. Dey, J. F. Kurose, and D. Towsley, "On-Line Scheduling Policies for a Class of IRIS (Increasing Reward with Increasing Service) Real-Time Tasks," *IEEE Transactions on Computers*, vol. 45, no. 7, pp. 802–813, July 1996.

[11] L. B. Becker, E. Nett, S. Schemmer, and M. Gergeleit, "Robust Scheduling in Team-Robotics," *Journal Of Systems and Software*, vol. 77, no. 1, pp. 3–16, 2005.

[12] J. A. Malas, "F-22 Radar Development," in *IEEE National Aerospace and Electronics Conference*, vol. 2, July 1997, pp. 831–839.

[13] J. C. Curlander and R. N. McDonough, *Synthetic Aperture Radar : Systems and Signal Processing*. Wiley-Interscience, 1991.

[14] T. W. Jeffrey, "Track Quality Estimation for Multiple-Target Tracking Radars," in *Proceedings of the 1989 IEEE National Radar Conference*, March 1989, pp. 76–79.

[15] G. van Keuk and S. S. Blackman, "On Phased-Array Radar Tracking and Parameter Control," *IEEE Transactions on Aerospace and Electronic Systems*, vol. 29, no. 1, pp. 186–194, January 1993.

[16] G. W. Stimson, *Introduction to Airborne Radar*, 2nd ed. SciTech Publishing, January 1998.

[17] S. Gopalakrishnan, M. Caccamo, C.-S. Shih, C.-G. Lee, and L. Sha, "Finite-Horizon Scheduling of Radar Dwells with Online Template Construction," *Real-Time Syst.*, vol. 33, no. 1-3, pp. 47–75, 2006.

[18] S. Gopalakrishnan, P. G. Chi-Sheng Shih, M. Caccamo, L. Sha, and C.-G. Lee, "Radar Dwell Scheduling with Temporal Distance and Energy Constraints," in *Proceedings of the International Radar Conference*, October 2004.

[19] C.-S. Shih, P. Ganti, S. Gopalakrishnan, M. Caccamo, and L. Sha, "Synthesizing Task Periods for Dwells in Multi-Function Phased Array Radars," in *Proceedings of the IEEE Radar Conference*, April 2004, pp. 145 – 150.

[20] C.-S. Shih, S. Gopalakrishnan, P. Ganti, M. Caccamo, and L. Sha, "Template-Based Real-Time Dwell Scheduling with Energy Constraint," in *IEEE Real-Time and Embedded Technology and Applications Symposium*, May 2003.

[21] ——, "Scheduling Real-Time Dwells using Tasks with Synthetic Periods," in *IEEE Real-Time Systems Symposium*, 2003.

[22] L. Sha, R. Rajkumar, and J. P. Lehoczky, "Priority Inheritance Protocols: An Approach to Real-Time Synchronization," *IEEE Transactions on Computers*, vol. 39, no. 9, pp. 1175–1185, 1990.

[23] P. Li, H. Wu, B. Ravindran, and E. D. Jensen, "A Utility Accrual Scheduling Algorithm for Real-Time Activities With Mutual Exclusion Resource Constraints," *IEEE Transactions on Computers*, vol. 55, no. 4, pp. 454–469, April 2006.

[24] C. L. Liu and J. W. Layland, "Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment," *Journal of the ACM*, vol. 20, no. 1, pp. 46–61, 1973.

[25] L. George, N. Roivierre, and M. Spuri, "Preemptive and Non-Preemptive Real-Time Uni-Processor Scheduling," INRIA, Le Chesnay Cedex, France, Tech. Rep. Rapport de Recherche RR-2966, 1996.

[26] J. A. Stankovic, M. Spuri, K. Ramamritham, and G. C. Buttazzo, *Deadline Scheduling for Real-Time Systems*. Kluwer Academic Publishers, 1998, ch. 4: Response Times under EDF Scheduling, pp. 67–87.

[27] ——, *Deadline Scheduling for Real-Time Systems*. Kluwer Academic Publishers, 1998, ch. 3: Fundamentals of EDF Scheduling, pp. 27–67.

[28] T. P. Baker, "Stack-based Scheduling of Real-Time Processes," *Journal of Real-Time Systems*, vol. 3, no. 1, pp. 67–99, March 1991.

[29] P. Li, B. Ravindran, S. Suhaib, and S. Feizabadi, "A Formally Verified Application-Level Framework for Real-Time Scheduling on POSIX Real-Time Operating Systems," *IEEE Transactions on Software Engineering*, vol. 30, no. 9, pp. 613–629, September 2004.

[30] P. Li and B. Ravindran, "Fast Real-Time Scheduling Algorithms," *IEEE Transactions on Computers*, vol. 53, no. 8, pp. 1159–1175, September 2004.

[31] H. Cho, "Utility Accrual Scheduling with Non-Blocking Synchronization on Uniprocessors and Multiprocessors," PhD Dissertation Proposal, Virginia Tech, 2005, http://www.ee.vt.edu/~realtime/cho_proposal05.pdf.

[32] J. D. Northcutt, *Mechanisms for Reliable Distributed Real-Time Operating Systems: The Alpha Kernel*. Academic Press, 1987.

[33] M. Dertouzos, "Control Robotics: the Procedural Control of Physical Processes," *Information Processing*, vol. 74, 1974.

[34] U. Balli, “Utility Accrual Real-Time Scheduling Under Variable Cost Functions,” Master’s thesis, Virginia Tech, 2005, http://scholar.lib.vt.edu/theses/available/etd-08052005-155355/.

[35] H. Wu, U. Balli, B. Ravindran, and E. D. Jensen, “Utility Accrual Real-Time Scheduling Under Variable Cost Functions,” in *Proceedings of 11th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, August 2005, pp. 213–219.