Utility Accrual Real-Time Scheduling Under Variable Cost Functions

Umut Balli

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science
in
Computer Engineering

Dr. Binoy Ravindran, Chair
Dr. Amitabh Mishra
Dr. Y. Thomas Hou

July 28, 2005
Blacksburg, Virginia

Keywords: variable-cost functions, time/utility functions, utility accrual scheduling,
real-time scheduling, resource management, overload scheduling

Utility Accrual Real-Time Scheduling Under Variable Cost Functions

Umut Balli

(ABSTRACT)

We present a utility accrual real-time scheduling algorithm called CIC-VCUA, for tasks whose execution times are functions of their starting times. We model such variable execution times employing *variable cost functions* (or VCFs). The algorithm considers application activities that are subject to time/utility function time constraints (or TUFs), execution times described using VCFs, and concurrent, mutually exclusive sharing of non-CPU resources. We consider the multi-criteria scheduling objective of (1) assuring that the maximum interval between any two consecutive, successful completions of jobs *of a task* must not exceed a specified upper bound, and (2) maximizing the system's total accrued utility, while satisfying mutual exclusion resource constraints. Since the scheduling problem is intractable, CIC-VCUA statically computes worst-case sojourn times of tasks, selects tasks for execution based on their potential utility density, and completes them at specific times, in polynomial-time. We establish that CIC-VCUA achieves optimal timeliness during under-loads. Further, we identify the conditions under which timeliness assurances hold. Our simulation experiments illustrate CIC-VCUA's effectiveness and superiority.

Acknowledgments

I would like to express my deepest gratitude to my Advisor, Dr. Binoy Ravindran. His constant guidance, encouragement and trust in my abilities helped me carve my way to this thesis research. Also, his professional and friendly approach taught me valuable lessons which will be forever with me. I would also like to thank Dr. Amitabh Mishra and Dr. Y. Thomas Hou for their inputs and comments on my work.

I like to thank Dr. Jensen who guided us on the VCF problem, helped us understand the radar application, and got MITRE to fund this work. I also would like to thank The MITRE Corporation for funding our research since January 2005 and also Jonathan S. Anderson for his immense help with the motivating application.

I would like to thank also the Real-Time Systems Lab family Haisang Wu, Hyeonjoong Cho, and Dr. Peng Li for their support and company. I would especially like to express my appreciation to Haisang Wu for all the help and guidance he provided in all the areas of my research.

Also, I would like to thank all my friends for their constant support, understanding, and help. They were always with me whether it was my worst day or my best. I especially want to thank Orkun Akar, Geremy Michel, Rahmi Zorlu, Umur Yilmaz, Ahmet Bayram, Dr. Maurizio Porfiri, Kerem Eraydin and Banu Cankaya. I would also like to thank the Turkish community at Virginia Tech as a whole.

Last, but not the least, I want to thank my family members. They never ceased their immense moral support and guidance for one moment, and always made me remember the reasons I pursued my studies. I will be forever in debt for everything they have ever done for me.

Contents

List of Tables

List of Figures

List of Algorithms

Chapter 1

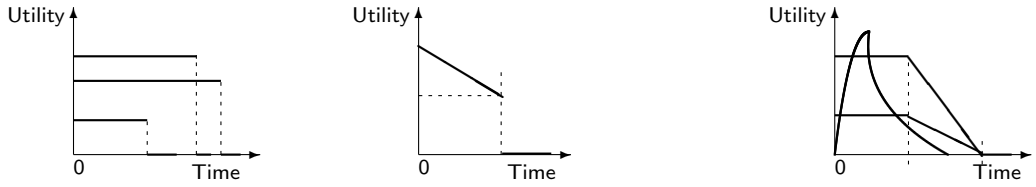
Introduction

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 [4]) 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 desire the strongest possible assurances on activity timeliness behavior. Another important distinguishing feature of these systems is their relatively long execution time magnitudes—e.g., in the order of milliseconds to seconds, or seconds to minutes.

When resource overloads occur, meeting 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, and vice versa; 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 [7]—i.e., they can satisfy all deadlines.

Deadlines by themselves cannot express both urgency and importance. Thus, we consider the abstraction of time/utility functions (or TUFs) [9] that express the utility of completing an application activity as a function of that activity's completion time. We specify deadline as a binary-valued, downward "step" shaped TUF; Figure 1.1(a) shows examples. 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 [15]

Figure 1.1: Example TUF Time Constraints

Many embedded 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 deadlines, where a positive utility is accrued for completing the activity anytime before the deadline, after which zero, or infinitively negative utility is accrued. Figures 1.1(b)–1.1(c) show example such time constraints from two real 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) [15]. Details of these applications can be found in [2] and [15], 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 scheduling algorithms that optimize them, as UA scheduling algorithms.

UA algorithms that maximize summed utility under downward step TUFs (or deadlines) [3, 13, 23] default to EDF during under-loads, since EDF can satisfy all deadlines during those situations. Consequently, they obtain the maximum total utility during under-loads. When overloads occur, they 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 thesis, we focus on variable cost scheduling. By variable cost scheduling, we mean scheduling activities that have durations (e.g., tasks with execution times) which vary while being performed—e.g., depending on when they begin, or on how long they have been running, or on other factors. In this context, cost means the duration of the activity—a term that comes from one of the interesting and important applications for such scheduling. The variable cost is specified by a cost function. Thus, even if there were no new activity arrivals, the load to be scheduled changes while the activities are being performed.

Past efforts on deadline-based and UA scheduling do not consider variable cost scheduling. For example, past UA scheduling algorithms [3, 13, 23] do not allow task execution times to vary while tasks are being performed. The *imprecise computations* [12] and *IRIS (Increasing Reward with Increasing Service)* [6] models propose optional parts in addition to the mandatory parts of task execution times. But, these concepts are different from UA and variable cost scheduling, because (1) in these models the longer the optional part executes, the higher the reward is, while in UA scheduling the utility (reward) can only be accrued by an activity when its completed, and the utility value is decided by the completion time; and (2) 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.

Thus, no past efforts have been devoted to the analysis of the intersection of UA scheduling and variable cost scheduling. In this thesis, we precisely focus on this unexplored domain. We consider repeatedly occurring, real-time application activities whose time constraints are specified using TUFs. The activities execution times are described by cost functions, which may vary as the activities are being performed. For such an application model, 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 a specified upper bound; 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 Constrainted Variable Cost Utility Accrual Algorithm* (or CIC-VCUA). The algorithm optimizes the dual criteria while (1) variable cost differs and/or (2) resource dependency exists in the system. We prove several timeliness properties of CIC-VCUA including optimal timeliness during under-loads, and identify the conditions under which timeliness assurances hold. Finally, our simulation studies confirm CIC-VCUA's effectiveness and superiority.

Thus, the contribution of the thesis 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 thesis is organized as follows: In Chapter 2, we describe the motivating application for variable cost scheduling; in Chapter 3, we outline our activity and utility models, and state the scheduling criterion. We present CIC-VCUA in Chapter 4 and establish the algorithm's timeliness properties in Chapter 5. The experimental measurements are reported in Chapter 6. Finally, we conclude the thesis and identify future work in Chapter 7.

Chapter 2

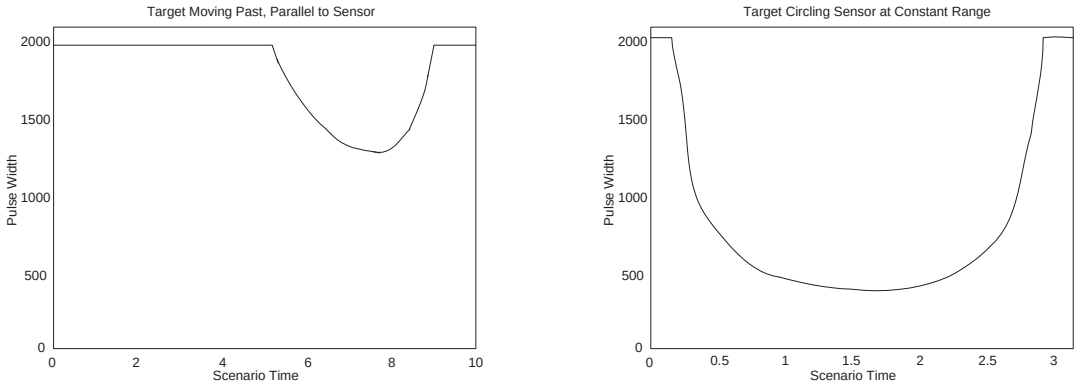
Motivating Application

The application context of interest to us for variable cost scheduling is an air-to-air radar tracking problem for which few 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 type of tracking problem employs an Active, Electronically Steerable Array radar (AESA) to provide an end-to-end tracking service. Examples of such systems include the radar systems installed in certain United States and European tactical aircraft and Naval surface craft. [14]

The problem notionally consists of three component tasks: (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



(a) Example Cost Function 1

(b) Example Cost Function 2

Figure 2.1: Example Cost Functions for (a) a Target Flying at a Higher Altitude and Faster Velocity than the Radar Platform, and (b) a Target Circling the Radar Platform at a Constant Range

occur almost any time while tracking. For each of the three tasks, the radar must make one or more measurements by illuminating the object with pulses of energy called 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 maintain acceptable values for certain application-level quality of service (AQoS) metrics.

One of such critical AQoS metric is track quality [8], 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. [20]

Another metric of interest is efficiency of the radar utilization—e.g., the track quality that can be maintained for a given percentage utilization of the radar. Contrary to intuition, maximizing radar utilization is neither necessary nor sufficient to attain various AQoS metrics; intentional radar idle time can result in improved AQoS metrics.

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 [19]. 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 2.1. Figure 2.1(a) shows the cost function for a target object flying at a higher altitude and faster velocity than the radar platform; Figure 2.1(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 derived in part from the physical properties of the transceiver and antennae.

Cost functions of real applications can be increasing, decreasing, or strictly convex shaped.

In this thesis, we make the simplifying assumption that cost functions are unimodal. We further note that the timescales associated with the cost and time/utility functions can and do vary widely. In this problem, the TUFs associated with each dwell may have critical times in the range of tens to hundreds of microseconds; the VCFs may only change significantly over the course of tens or hundreds of seconds. This facet of the model is not exploited in the work which follows.

For the purposes of the analysis and algorithms presented below we make the simplifying assumption that dwell jobs are preemptible. This is not generally true of AESA systems, but it is a simplifying assumption commonly made in such analyses and we propose to loosen this constraint in future work.

Chapter 3

Models and Objective

3.1 System and Task Model

We consider a preemptive system which consists of a set of periodic 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 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 job scheduling queue.

3.2 Resource Model

Jobs can access non-CPU resources, which in general, are serially reusable. Examples include physical resources (e.g., disks) and logical resources (e.g., critical sections guarded by mu-

texes). Similar to fixed-priority resource access protocols (e.g., priority inheritance, priority ceiling) [16] and that for UA algorithms [3, 10], we consider a single-unit resource model. Thus, only a single instance of a resource is present and a job must explicitly specify the resource that it wants to access.

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 [3, 10], we program such precedences as resource dependencies.

3.3 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; completion of J_k at a time t will yield a 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 1.1. TUFs which are not unimodal are multimodal. In this thesis, we restrict our focus to *non-increasing*, unimodal TUFs i.e., those TUFs for which utility never increases as time advances. Figures 1.1(a) and 1.1(b) 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 $J_{i,j}$, and $X_{i,j} - I_{i,j}$ is equal to the period P_i of the task T_i. If a job's termination time is reached and its execution has not been completed, an exception is raised. Normally, this exception will cause the job's abortion and execution of exception handlers.

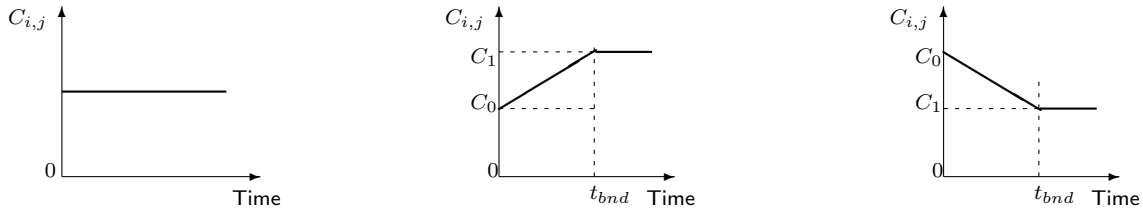
3.4 Task Execution Time Model

After a job is released, the job's 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 the VCF of T_i's *jth* job.

For a job $J_{i,j}$, the x-axis of its VCF is the absolute time according 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.

In this thesis, we consider *unimodal* VCFs. Figure 3.1 shows examples of VCFs. For jobs whose execution times do not vary with time after their arrivals, we define a constant VCF for each job. Figure 3.1(a) shows a constant VCF. Figure 3.1(b) and Figure 3.1(c) show an increasing VCF and a decreasing VCF, respectively. From Figure 3.1(b) and 3.1(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}, $C_{i,j}(t)$ equals C_1.

Without being task specific, J_k.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 $J_k.C(t_{cur}) = J_k.\text{VCF}(t_{cur})$. Hereafter, in discussion of TUF and VCF, we interchangeably use the terms *task* and *job* if no confusion is raised.



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

Figure 3.1: Variable Cost Funtions of Job $J_{i,j}$

3.5 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 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. Even a constant load at the task arrivals—one that is an under-load—may gradually becomes a heavier load, and may eventually become an overload even without new tasks arriving, due to increasing VCFs. Therefore, if the dynamic system load is so high such that scheduling objective (1) cannot be satisfied for each task, some tasks may be dropped. In such cases, tasks that are not dropped are still subject to the two scheduling objectives. We propose the CIC-VCUA algorithm to solve this problem, and its details are described in Chapter 4.

Chapter 4

The CIC-VCUA Algorithm

4.1 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, 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 into the schedule. This will increase the likelihood of maximizing the aggregate utility.

The metric used by CIC-VCUA to determine the return on investment for a job is called the *Potential Utility Density* (or PUD), which was originally developed in [3]. 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.

To compute 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: $\frac{J_k.U(J_k.FinT)+\sum_{J_l \in J_k.Dep} J_l.U(J_l.FinT)}{J_k.FinT-t_{cur}}$.

4.2 Static Selection of CIC-VCUA

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. This process is called feasibility check. Details of feasibility check will be described in Section 4.4.

In order to perform feasibility check, we have to find the maximum possible execution times of tasks. Depending on the shapes of VCFs, 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 $C_{i,j}(t^b_{i,j}) + t^b_{i,j} = X_{i,j}$. Apparently, $C_{i,j}(t^b_{i,j})$ 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, the maximum execution time of a job $J_{i,j}$ 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 [11].

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 performance metric PUD. To this aim, we use a job selection flag in which each task instance (job) is labeled 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 for our scheduling performance objective.

We perform the labeling of 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., $\text{PUD}_i = \frac{U_{i,1}\left(C_{i,1}(t_i^0)\right)}{C_{i,1}(t_i^0)}$. 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 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., $SEL_{i,j} = selected$, $\forall i \in 1', \cdots, n', \forall j$.

4.3 Worst-Case Sojourn Times of Tasks in CIC-VCUA

The first objective of this algorithm is to assure that the maximum interval between any two consecutive, successful completions of jobs *of a task* T_i must not exceed its period P_i. In order to satisfy this scheduling objective, CIC-VCUA tries to find the worst case sojourn time of each task, and makes all jobs *of a task* finish on the same time relative to their arrivals. Thus, CIC-VCUA ensures that all jobs $J_{i,j}$ of a task T_i have identical sojourn times, so as to achieve the first objective of the algorithm. We should note that the notion of *termination time* is the same as that of *deadline*, for tasks with step TUFs where $X_i = D_i$. So the *Earliest Deadline First* (EDF) algorithm is also denoted as *Earliest Termination First* (EXF) in this

thesis. 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, as we can see in Section 4.4. 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 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 load increase or abortions of other jobs. Figure 4.1 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 $J_{i,j}.X$.

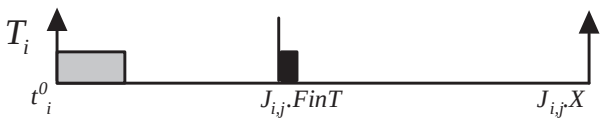


Figure 4.1: A Job Example

For task set $\mathbf{T}'$ with $load'_b \leq 1.0$, our algorithm resembles EXF. Hence, in order to find sojourn times under CIC-VCUA, the paradigm for finding response times in the EXF algorithm is used. As explained in [17], response times under EXF are found using a notion of *deadline busy period*. This is needed because we can find out how long it takes for each task to complete when there are other tasks present in the system.

First, it is necessary to calculate the synchronous *busy period* length before calculation of individual *deadline busy period* lengths of tasks. The synchronous *busy period* (denoted as L) is the interval of time, during which the processor is not idle. Further, if all the first instances of tasks were to be released synchronously (the worst care scenario), it would take L amount of time for all jobs to complete. Thus, it is clear to see *busy period* bounds the individual completion times or *deadline busy period* lengths of task. The busy period is given by [18]:

$$L^0 = 0$$

$$L^{m+1} = W(L^m) \tag{4.1}$$

where:

$$W(t) = \sum_{i=1}^{n'} \left\lceil \frac{t}{T_i} \right\rceil C_i \tag{4.2}$$

Busy period is found when the iteration ends at $L^m = L^{m+1}$. After L is calculated, individual *deadline busy period* lengths, L_i can be calculated. We need to find out which tasks will have to be executed before our target task. It is intuitive that in the absolute termination time X_i of a task T_i, only tasks with termination times shorter than or equal to X_i can be executed. The deadline busy period of a task T_i with an arrival time a is given by [17]:

$$L_i^0(a) = 0$$

$$L_i^{m+1}(a) = W_i(a, L_i^m(a)) + (1 + \left\lfloor \frac{a}{X_i} \right\rfloor)C_i \tag{4.3}$$

where:

$$W_i(a,t) = \sum_{i \neq j, X_j \leq a + X_i} \texttt{min} \left(\left\lceil \frac{t}{X_j} \right\rceil, 1 + \left\lfloor \frac{a + X_i - X_j}{X_j} \right\rfloor \right) C_j \tag{4.4}$$

In calculating the deadline busy period length L_i in Equation 4.3, 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 T_i, and the second term accounts for the T_i's instances that have to be executed. The iterative computing will stop when $L_i^{m+1}(a) = L_i^m(a)$. Algorithm 1 shows the calculation of maximum deadline busy periods.

As Algorithm 1 suggests, the task list τ (which is initially ordered by PUD and then by non-decreasing termination times) is used. Starting from the task with the maximum X_i, which consequently could have the length of L, tasks that have absolute termination times $X \leq X_i$ 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 consequently becomes a bound for the next task in τ. So, the algorithm moves down to the list to calculate the maximum L_i possible for each task.

```
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+1} − C_i + X_i < e_{k+1};
8:         a := e_k − X_i;
9:     L_i := L_i(a);
```

Algorithm 1: `maxDeadBusyP()`

After calculation of 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 $T_i.wcST(a) = \texttt{max}(C_i, L_i(a) - a)$. The sojourn time is used to make sure that each job is completed on this specific point after it is released, so that jobs can meet the bound constraint on consecutive completions.

Naturally, for the first instance of the task, $T_i.wcST$ becomes this job's finish time, i.e., $J_{i,1}.FinT = T_i.wcST$. We can find finish times of the subsequent jobs of a task by the equation $J_{i,j}.FinT = J_{i,j-1}.FinT + P_i$.

The completion times of jobs are pushed back further to their termination times as system load changes with variable execution times. In order to keep the bound constraint, across the range of $load_b \leq 1.0$, our algorithm uses the worst case sojourn times as predicted finish times. For $load_b > 1.0$, the algorithm places finish times slightly before jobs' termination times. Pushing finish times closer to termination times for higher loads is necessary because, in this load region, (1) the worst case sojourn time calculation gets more unpredictable for different shapes of VCF (since the calculation uses $C_i(t_i^0)$, and (2) sojourn times of tasks are already close to their termination times because of the load.

4.4 Dynamic Utility Accrual Scheduling

After the initial static steps, CIC-VCUA reserves the maximum sub task set consisting of the highest PUD tasks, whose dynamic load will not cause the system overload. CIC-VCUA then adopts the preemptive earliest termination time first (or EXF) scheduling policy which is known to be optimal from the feasibility point of view [11].

At every job $J_{i,j}$ arrival, 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 is left. At this time, if the absolute time is far from $J_{i,j}.FinT$, job $J_{i,j}$ is preempted. Later it will be picked up again at $J.FinT$ to be finished.

We use Δ to denote this small amount of time. Δ is a very small quantity of time selected ($J.C \gg \Delta$) so that its interference of finishing a 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 a tiny amount time Δ to finish. It is also used to break the tie, if two or more jobs have the identical finish times. When a job J's remaining execution time is only Δ, and it is preempted and will be resumed at $J.FinT$, we say the job J is *ready to complete*.

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

- $\mathcal{J}_r = \{J_1, J_2, \cdots, J_m\}$ 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.
- `findSEL(`σ`)` returns the first job in σ whose selection flag $SEL = selected$.

- `selectedJob(`σ`)` returns a job to execute with amount of time it will execute.
- `headOf(`σ`)` returns the first job in σ.
- `sortByPUD(`σ`)` returns a schedule by non-increasing PUD. If two or more jobs have the same PUDs, then the job(s) with the largest execution time should appear before any others with the same PUDs.
- Function `owner(`R`)` denotes the jobs that are currently holding resource R; `reqRes(`T`)` returns the resource requested by T.
- `insert(`T`,`σ`,`I`)` inserts T in the ordered list σ at the position indicated by index I; if there are already entries in σ at the index I, T is inserted before them. After insertion, the index of T in σ is I.
- `remove(`T`,`σ`,`I`)` removes T from ordered list σ at the position indicated by index I; if T is not present at the position in σ, the function takes no action.
- `lookup(`T`,`σ`)` returns the index value associated with the first occurrence of T 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.

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 proceeds to find out resource dependencies among jobs and constructs the dependency list for each job (line 8), and 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 14, the job with the largest PUD and its dependencies are inserted into

```
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:          calculatePUD (J_k);
10: σ_tmp :=sortByPUD(σ);
11: for ∀J_k ∈ σ_tmp from head to tail do
12:     if J_k.PUD ≥ 0 and J_k.SEL then
13:         insertByEXF(σ, J_k);
14:     else break;
15: J_exe:=selectedJob(σ);
16: return J_exe
```

Algorithm 2: CIC-VCUA: Dynamic Part Description

σ by `insertByEXF()`. Thus, σ becomes a feasible schedule sorted by the jobs' termination times, in an non-decreasing order. Then, the `selectedJob` () function finds a job in σ and returns it for execution.

For every job J in its execution life, when the job has only Δ as the remaining execution time, the system will compare Δ with $J.FinT - t_{cur}$. If $\Delta < J.FinT - t_{cur}$, J is preempted, and other jobs may be selected. Later, when $J.FinT - t_{cur} = \Delta$, J will preempt the current running task at, so it can finish at $J.FinT$.

Such monitoring and preemption/resumption steps are realized by the procedure `selected-Job` (). It picks a job with the earliest finish time from σ. If this job is not *ready to complete*, it makes sure that the job executes $J.C - \Delta$ time units. Otherwise, `selectedJob` () runs this job to completion. After finishing such jobs, the algorithm seeks another job to execute.

4.5 Resource and Deadlock Handling

Before CIC-VCUA can compute job partial schedules, the dependency chain of each job must be determined, as shown 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/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 "$\cdot$" 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 resource access protocols, are not appropriate for the class of embedded real-time systems that we focus on. For example, the Priority Ceiling protocol [16] assumes that the highest priority of jobs accessing a resource is known. Likewise, the Stack Resource policy [1] assumes preemptive "levels" of threads *a priori*. Such assumptions are too restrictive for the class of systems that we focus on (due to their dynamic nature).

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 LoPUD of each job at t_{cur}, and the algorithm aborts the job with the minimal LoPUD in the cycle to resolve a deadlock.

4.5.1 Manipulating Partial Schedules

The `calculatePUD()` algorithm (Algorithm 5) accepts a job J_k (with its dependency list), and on completion determines PUD for J_k. It assumes that jobs in $J_k.Dep$ are finished

at their prospective 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 in J_k's dependency chain, which needs to be completed before executing J_k. First, the algorithm calculates utilities that can be accrued by the dependencies and J_k, assuming that they will finish at their respective finish times $J.FinT$. CIC-VCUA also calculates the total execution times of J_k and its dependents. Finally, J_k's PUD is calculated in line 6 of the procedure `calculatePUD` ().

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 dependencies to σ. The updated σ is an ordered list of jobs, where each job is placed according to the termination time it should meet. Note that the time constraint that a job should meet is not necessarily the job termination time. In fact, the index value of each job in σ is the actual time constraint that the job must meet.

A job may need to meet an earlier termination time in order to enable another job to meet its time constraint. Whenever a job is considered for insertion in σ, it is scheduled to meet its own termination time. However, all of the jobs in its dependency list must execute before it 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).

```
 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()`

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 has to be mentioned that the real time constraint that a job has to meet is its finish time $J.FinT$. The procedure `insertByEXF()` makes sure to resolve resource dependency issues and, accordingly, may change the order of tasks' execution.

4.5.2 Selecting a Job for Execution

The procedure `selectJob()` (Algorithm 7) finds the job that will be executed, as well as the amount of execution time it needs to be executed. At the beginning of the algorithm, the job with the earliest finish time *Earliest* in σ is found.

At the beginning of Algorithm 7, the procedure checks whether the currently running job *CurRunning* holds resources (line 3). If it is holding resources, then the algorithm makes sure that this job is executed so that the held resources can be freed. Then, *Earliest* is selected to be the running task, if its finish time has arrived (line 6), and the algorithm returns with $J_{exe} = Earliest$.

```
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()`

If line 6 does not find a job to finish, then the algorithm checks if there's a previous job *Prev* exists at line 10. Its existence means that *Prev* is currently holding resources, and has to be executed to release these resources. However, at the same time, we have to make sure that jobs that are *ready to complete* finish without delay. Therefore, jobs holding resources could only be preempted by ready jobs and the execution of *Prev* has to follow completions of

ready jobs. So, lines 10– 13 ensures that $Prev$ is favored for execution after a job completes.

If there's no finishing jobs and no $Prev$ exists, then the algorithm seeks for a job that could 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 picked at its finish time $J.FinT$ to complete. If no tasks can be found to execute at line 19, then the algorithm idles the processor until either the earliest finish time or a new job arrival.

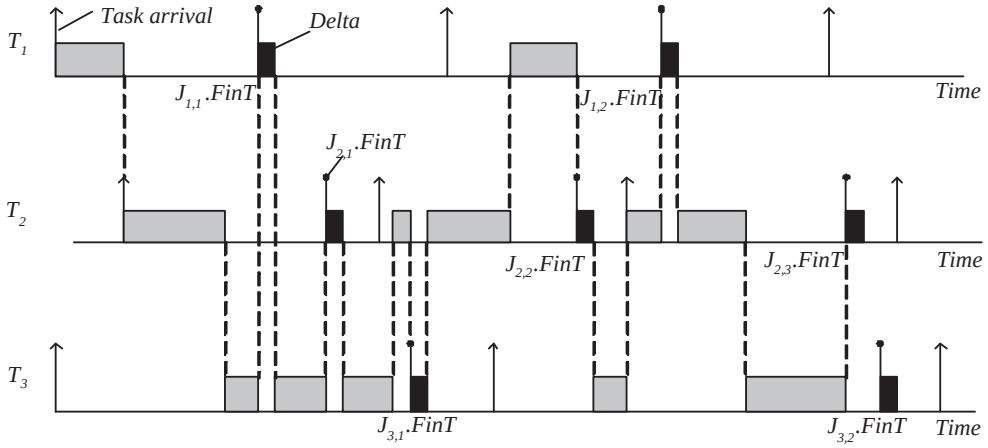


Figure 4.2: Example of a Task Set

An example of how CIC-VCUA executes jobs is shown in Figure 4.2. Upward arrows indicate both job arrivals and termination times, and black blocks denotes Δ. In this example, tasks $J_{1,1}$ and $J_{3,1}$ arrive at the same time, however because $J_{1,1}$'s finish time is earlier, scheduler 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, $J_{3,1}$ takes over the scheduler, however it is again preempted to let $J_{2,1}$ finish. After this, $J_{3,1}$ is run again and and completes at its finish time.

4.6 Asymptotic Time Complexity

To analyze the complexity of CIC-VCUA (Algorithm 2), we consider n jobs in the ready queue and a maximum of r resources. In the worst case, `buildDep()` may build a dependency list with a length n; so the `for`-loop from line 4 to 9 can be repeated $O(n^2)$ times in the worst case. The complexity of procedure `sortByPUD()` 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 to perform `insert()`, `remove()` and `lookup()` operations on the tentative schedule. Therefore, the worst-case complexity of the CIC-VCUA algorithm is $2 \times O(n^2) + O(n \log n) + n \times O(n \log n) = O(n^2 \log n)$. Although the overhead of CIC-VCUA is higher than EDF which has a complexity of $O(n)$, such cost is justified because of the long execution time magnitude of the problem—e.g., in the order of milliseconds to seconds, or seconds to minutes.

Chapter 5

Properties of CIC-VCUA

5.1 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.

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 it can execute, and therefore, must precede it in the schedule.*

Proof `insertByEXF()` seeks to maintain an output queue ordered by jobs' 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 $J_k.X$, $J_l.X$ 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.*

5.2 Timeliness Properties of CIC-VCUA

We consider timeliness properties of CIC-VCUA, and compare it with a number of well-known algorithms. Specifically, we consider the following two conditions: (1) there is a set of independent periodic tasks with step TUFs; and (2) there are sufficient processor cycles 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 [7] is also produced by CIC-VCUA, yielding equal total utilities. Not coincidentally, this is simply a termination time ordered schedule as finish times result in same ordering.*

Proof We prove this by examining Algorithms 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 [7]. As proved in [7, 11], 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 [5].

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

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

Theorem 7 *CIC-VCUA assures that the maximum interval between any two consecutive, successful completions of jobs of a task does not exceed the length of 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 in task T_i. Also, let $T_i.wcST$ be the worst case sojourn time of Task T_i (and of all its jobs). In this scheme, the maximum interval between these two jobs 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 $J_{i,1}.FinT = J_{i,1}.ST = T_i.wcST$. From our algorithm, we know 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 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 the tasks with 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.

Chapter 6

Experimental Results

6.1 Experimental Settings

To evaluate the efficiency of CIC-VCUA, we perform simulation experiments to validate our conclusions. Our simulator is written with the simulation tool OMNET++ [21], which provides a discrete event simulation environment.

Table 6.1: 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	$2.4 \sim 9.6$	$[1, 10]$	$\langle 0\text{–}0.1, E(C_o) \rangle$

We select task sets with 16 tasks in three applications for our study. Their parameters are summarized in Table 6.1. Within each range, the period P is uniformly distributed. The synthesized task sets simulate the varied mix of short and long periods. The U^{max}s of the TUFs in A_1, A_2, and A_3 are uniformly generated within each range. We define a linear increasing VCF $= k \times t + C_o$, a linear decreasing VCF $= -k \times t + C_o$, and a constant VCF $= C_o$ for each task. k is uniformed generated within the range $[0, 0.1]$. We change the

mean value of C_o, and generate normally-distributed values to adjust the system $load_b$. In all of the experiments, Δ value is set to be 2e-4. Finish times of tasks $J.FinT$ are pushed to their termination times when $load_b > 0.9$ in order to avoid unpredictability of sojourn time calculations.

6.2 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 the system $load_b$ 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 all tasks. We define the former as maximum intra-task completion interval, and the latter as maximum inter-task completion interval.

We experimented with two kinds of settings when considering VCFs for task sets: *homogeneous* and *heterogeneous*. A task set that consists of tasks with *only* one type of VCF shapes is referred to as *homogeneous set* hereon. On the other hand, in *heterogeneous* sets, tasks of the set can have any VCF shapes that we specified in Table 6.1. In the following experiments, TUFs are specified to be step functions.

6.2.1 Homogeneous VCFs

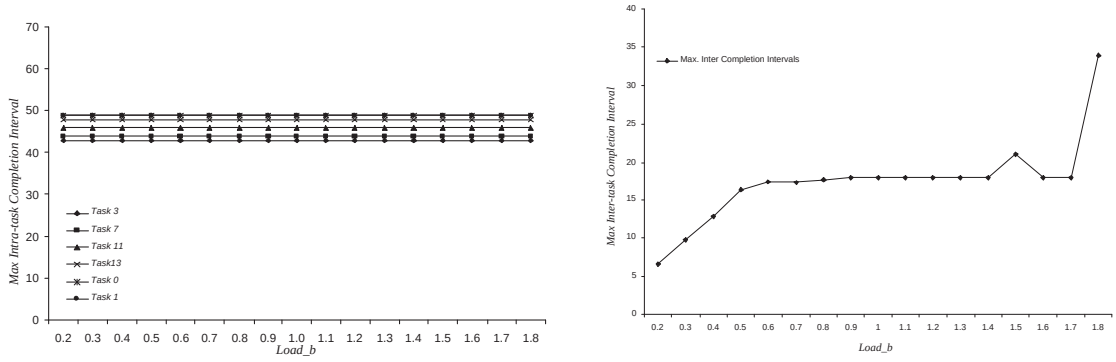
In the experiments of this section, we used constant VCFs for all tasks. Figure 6.1 shows the maximum intra- and inter-task completion intervals, as $load_b$ varies. In Figure 6.1(a), we only show 6 tasks as examples selected from the task set to study their maximum intra-task completion interval.

To validate Theorem 7, in Table 6.2, we list periods of the selected tasks from Figure 6.1(a). From Figure 6.1(a) with Table 6.2, we observe that in all $load_b$ regions, the maximum intra-

Table 6.2: Tasks and Their Periods for Homogeneous Set

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

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 6.1(a) validate Theorem 7.



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

Figure 6.1: 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 6.1(b). The minimum period of the task set is 3. From the figure, we observe that during underload, the maximum inter-task completion interval is less than 20 and, during overloads, the maximum inter-task completion may go beyond 20 in order to satisfy intra-task completion bounds.

Experiments for homogeneous sets with monotonic increasing and decreasing VCFs with various TUFs yield similar results shown in Figure 6.1. They are omitted here for brevity, however they can be seen in Appendix A.

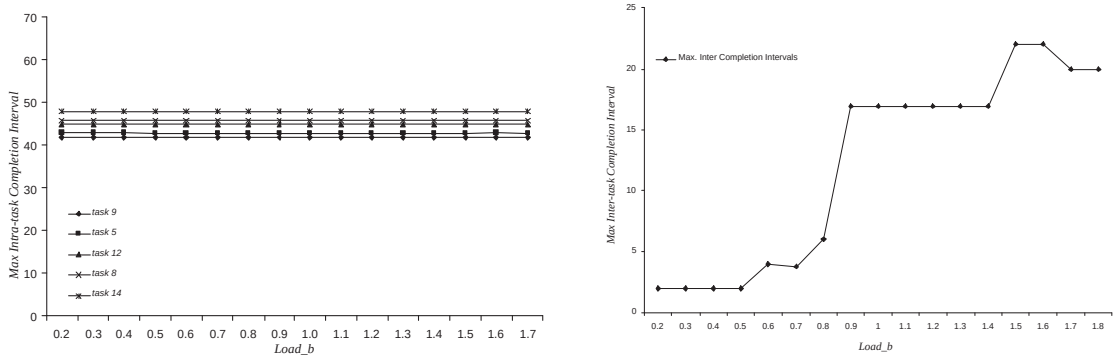
6.2.2 Heterogeous VCFs

In the experiments below, we generated random VCFs for each task. These shapes we used for VCFs are described in Section 3.4. Figure 6.2 shows the maximum intra- and inter-task completion intervals, as $load_b$ varies. In Figure 6.2(a), we selected 5 tasks to study their maximum intra-task completion interval. These task's periods are shown in Table 6.3 to justify Theorem 7.

Table 6.3: Tasks and Their Periods for Heterogeneous Set

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

From this figure, we again 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, similar to the case with *homogeneous set*, *skipped* tasks with low PUDs never get a chance to execute. Hence, plots in Figure 6.2(a) also validate Theorem 7.



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

Figure 6.2: Maximum Intra- and Inter-Task Completion Interval for Heterogenous Set

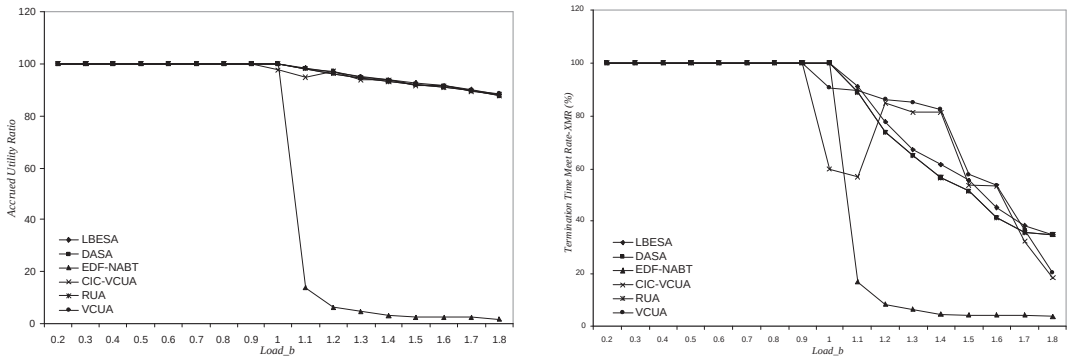
Figure 6.2(b) shows the maximum inter-task completion interval of the whole task set. The minimum period of the task set is 3. We observe similar results to *homogeneous set*. Figures showing heterogeneous VCFs with various TUF shapes are omitted here for brevity, but could be observed in Appendix A.

6.3 Performance on Utility Accrual

We then conduct experiments to study the performance of CIC-VCUA on the scheduling objective (2). The VCF is defined to be constant in the CIC-VCUA algorithm, so we can compare it with other UA algorithms that cannot deal with non-constant VCFs and varying execution times.

We consider step and decreasing TUFs in our experiments. Our first experiments compare CIC-VCUA with RUA [23], DASA [3], LBESA [13], VCUA [22], and EDF without abortion (EDF-NABT) [7] algorithms to show performance under step TUFs. We also compare CIC-VCUA's performance under decreasing TUFs with those of RUA, DASA, VCUA and LBESA.

Figures 6.3 and 6.4 show the accrued utility ratios (or AUR) and termination time meet rates (or XMR) of the algorithms as the system $load_b$ increases. AUR is defined as the ratio of accrued aggregate utility to the maximum possible utility, and XMR is the ratio of the number of jobs meeting their termination times to the total number of jobs releases.

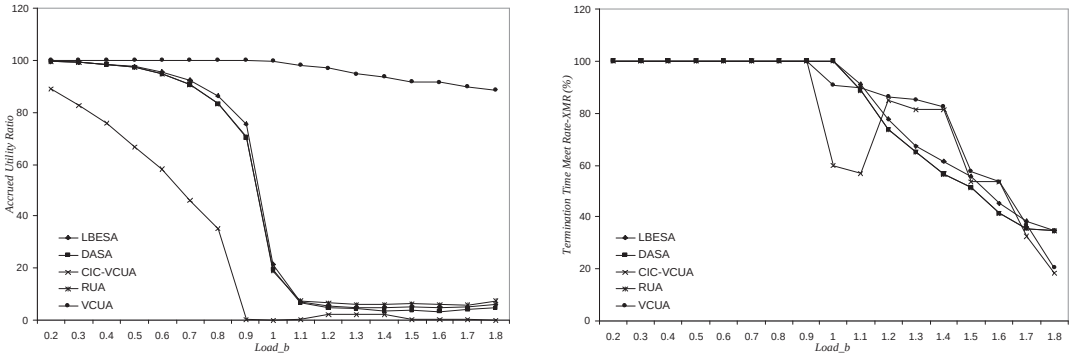


(a) AUR (b) XMR

Figure 6.3: AUR and XMR of CIC-VCUA and other UA Algorithms, Constant VCFs, Step TUFs

Figure 6.3 depicts AUR and XMR comparison of CIC-VCUA with other UA algorithms under step TUFs. Figure 6.3(a) demonstrates that utility accrual of CIC-VCUA is as good as that of DASA, RUA, and LBESA. XMRs of the algorithms are demonstrated in Figure 6.3(b).

We observe that the CIC-VCUA algorithm suffers higher termination time misses than the others during high loads. This is because CIC-VCUA needs to satisfy the completion interval bounds, so it has to statically label some tasks as *skipped*.



(a) AUR (b) XMR

Figure 6.4: AUR and XMR of CIC-VCUA and other UA Algorithms, Constant VCFs, Decreasing TUFs

The algorithm's performance under decreasing TUFs is shown in Figure 6.4(a). The algorithm accrues less utility for decreasing TUF, as depicted in Figure 6.4(a). Figure 6.4(b) compares XMRs of UA algorithms with CIC-VCUA. We observe that XMR of CIC-VCUA drops rapidly, similar to the results with step TUFs, because CIC-VCUA first pursues to achieve objective (1), and XMR is not its optimization consideration.

Figures 6.3 and 6.4 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 6.3 validate Theorem 5 and Corollary 6.

Experiments concerning monotonic increasing and decreasing VCFs yield similar results to those shown in Figures 6.3 and 6.4. These results are omitted here for brevity, however they can be seen in Appendix A.

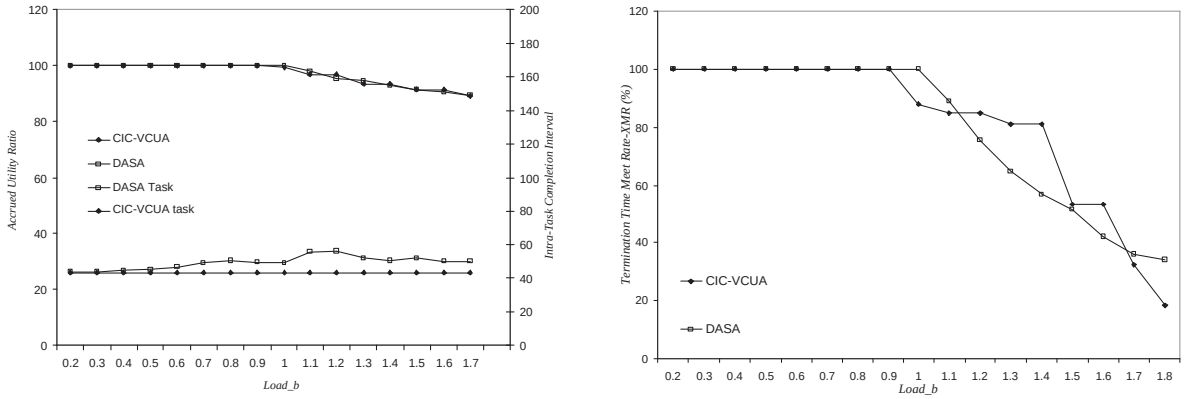
6.4 Results under Resource Dependency

To construct dependent task sets, we consider task sets where jobs may randomly request and release resources from some available set of resources during their life spans. The resource request and release times are uniformly distributed within a job's life cycle before job's *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 6.4 displays 6 tasks selected to study their maximum inter- and intra-task completion intervals.

Table 6.4: Tasks and Their Periods used under Resource Dependency

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

Figure 6.5 shows the comparison between CIC-VCUA and DASA. 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.



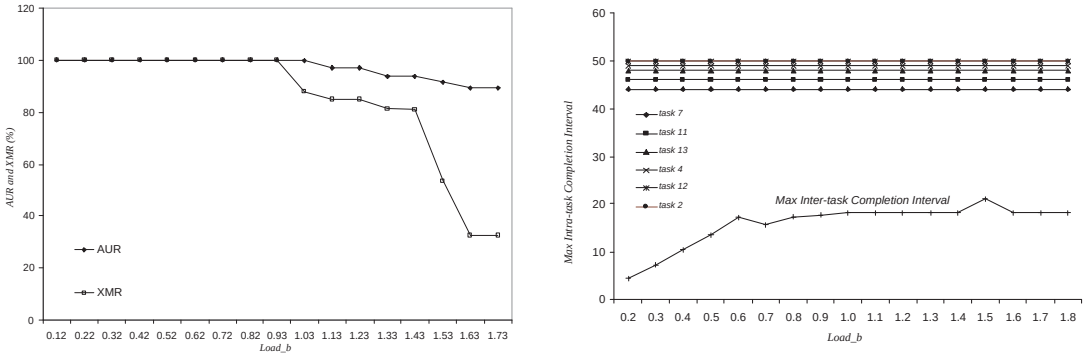
(a) AUR (b) XMR

Figure 6.5: CIC-VCUA vs DASA under Resource Dependency, Constant VCFs, Step TUFs

Figure 6.5(a) shows both AURs of CIC-VCUA and DASA, and the intra-task completion intervals of a randomly selected task under both algorithms as $Load_b$ varies. In terms of

AUR, CIC-VCUA performs as well as DASA. Also we observe from Figure 6.5(a) that the studied intra-task completion interval of DASA increases as load increases, and it cannot be kept within the bound of one period. However, CIC-VCUA keeps this intra-task completion interval constant to task's period with different system loads. Additionally, Figure 6.5(b) displays the XMR comparison of both algorithms. During overloads, in terms of XMR, DASA and CIC-VCUA present different behaviors because of their different scheduling processes.

Figure 6.6 shows the performance of CIC-VCUA under *decreasing* VCFs and step TUFs. As it can be observed from Figure 6.6(a), CIC-VCUA algorithm presents good AUR even with task sets with resource dependencies. The XMR decrease is due to the static selection which favors high PUD tasks. Figure 6.6(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.

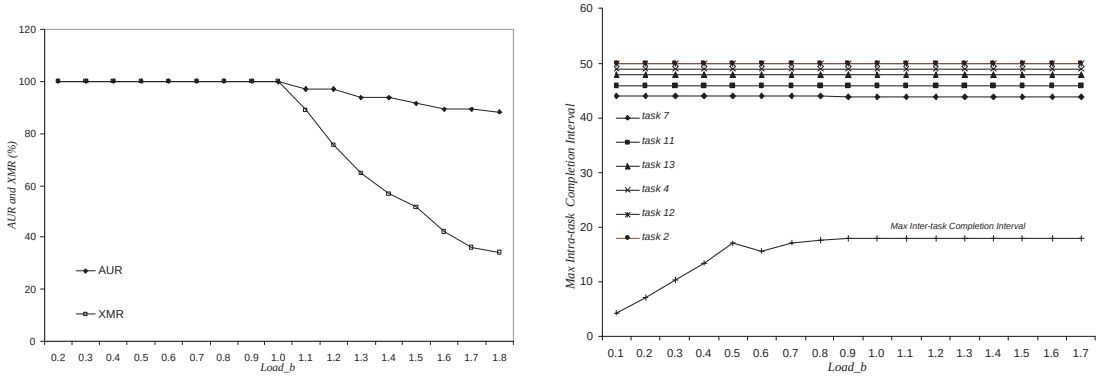


(a) AUR and XMR (b) Completion intervals

Figure 6.6: CIC-VCUA Performance under Resource Dependency, Decreasing VCFs, Step TUFs

Figure 6.7 shows CIC-VCUA performance under *increasing* VCFs and step TUFs, and also maximum intra-task and inter-task completion intervals. Even when increasing VCFs are considered, CIC-VCUA still respects resource dependency. AUR and XMR metrics are similar to the case with non-increasing VCFs, as Figure 6.7(a) suggests. Figure 6.7(b) displays inter- and intra-completion intervals of some *selected* tasks. Apparently, CIC-VCUA

keeps the intra-task completion interval bound to one period.



(a) AUR and XMR (b) Completion intervals

Figure 6.7: CIC-VCUA Performance under Resource Dependency, Increasing VCFs, Step TUFs

As it can be noticed, the performance under resource dependency is similar to that under no resource dependency. However, there's a small performance loss due to mutual exclusion requirements. The higher number of shared resources, the more performance decrease might be observed. 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. So with dependent task sets, CIC-VCUA cannot provide performance assurances and suffers losses, especially during high loads. However, as experiments suggest, CIC-VCUA yields fair results even under resource dependency.

Experiments concerning monotonic increasing and decreasing VCFs with other various TUF shapes yield similar results shown in Figures 6.7 and 6.6. These results are omitted here for brevity, however they can be seen in Appendix B.

Chapter 7

Conclusions

In this thesis, we present a real-time scheduling algorithm called CIC-VCUA. The algorithm considers tasks which are subject to TUF time constraints, mutually exclusive sharing of non-CPU resources, and tasks' whose execution times are functions of their starting times, described using unimodal VCFs.

This research is exploring the conjunction of Utility Accrual UA scheduling and variable cost scheduling. CIC-VCUA algorithm meets a two-fold objective: (1) it bounds the maximum interval between any two consecutive, successful completion of jobs *of a task* to the task's period and (2) it maximizes the system's total utility. The algorithm optimizes the dual criterion while variable cost differs and/or resource dependency exists in the system. This problem can be shown to be NP-hard. Our heuristic algorithm CIC-VCUA can solve this problem in polynomial time with the worst-case complexity of $O(n^2 \log n)$.

We establish that CIC-VCUA achieves optimal timeliness during under-loads, and identify the conditions under which timeliness assurances hold. Our experimental study confirms CIC-VCUA's effectiveness and superiority. We emphasize that the higher asymptotic cost of the algorithm and the consequent higher overhead are justified by the longer execution time

magnitudes of the VCF problem.

This thesis only scratched the surface of the VCF scheduling problem; so many problems are open for further research. The first direction for further research include relaxing the many simplifying task model assumptions made here. These include assumptions on VCF shapes, VCF time scales (with respect to TUF time scales), and arbitrary job preemptions, so that they are consistent with the AESA motivating application problem described in Chapter 2.

Another direction include developing TUF/UA VCF scheduling would be to develop TUF/UA VCF scheduling algorithms that can provide individual task timing assurances for satisfying a lower bound on each task's maximum utility. As for some applications, although they might have soft real time constraints, we might still need task-level timing assurances.

Yet another direction is to consider stochastic VCF models such as those with probabilistic task execution times. Non-deterministic nature of some of the real-time systems domains may be a target for this kind of research. If uncertainties in task execution times could be modeled stochastically, it may be possible to provide algorithmic solutions for such VCFs.

Bibliography

[1] T. P. Baker. Stack-based scheduling of real-time processes. *Journal of Real-Time Systems*, 3(1):67–99, March 1991.

[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*, volume 1586 of *LNCS*, pages 353–362. Springer-Verlag, April 1999.

[3] R. K. Clark. *Scheduling Dependent Real-Time Activities*. PhD thesis, CMU, 1990. CMU-CS-90-155.

[4] R. K. Clark, E. D. Jensen, and N. F. Rouquette. Software Organization to Facilitate Dynamic Processor Scheduling. In *IEEE Workshop on Parallel and Distributed Real-Time Systems, IEEE Parallel and Distributed Processing Symposium*, April 2004.

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

[6] 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 Trans. on Computers*, 45(7):802–813, July 1996.

[7] W. Horn. Some Simple Scheduling Algorithms. *Naval Research Logistics Quaterly*, 21:177–185, 1974.

[8] T. W. Jeffrey. Track quality estimation for multiple-target tracking radars. In *Proceedings of the 1989 IEEE National Radar Conference*, pages 76–79, March 1989.

[9] 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*, pages 112–122, December 1985.

[10] P. Li. *Utility Accrual Real-Time Scheduling: Models and Algorithms*. PhD thesis, Virginia Tech, 2004. `http://scholar.lib.vt.edu/theses/available/etd-08092004-230138/` (last accessed: January 22, 2005).

[11] C. L. Liu and J. W. Layland. Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment. *Journal of the ACM*, 20(1):46–61, 1973.

[12] J. W. S. Liu, W.-K. Shih, K.-J. Lin, R. Bettati, and J.-Y. Chung. Imprecise computations. *Proceedings of the IEEE*, 82(1):83–94, Janurary 1994.

[13] C. D. Locke. *Best-Effort Decision Making for Real-Time Scheduling.* PhD thesis, Carnegie Mellon University, 1986. CMU-CS-86-134, `http://www.real-time.org` (last accessed: January 22, 2005).

[14] J. A. Malas. F-22 radar development. In *Proceedings of the IEEE 1997 National Aerospace and Electronics Conference*, volume 2, pages 831–839, July 1997.

[15] 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. Technical report, Department of Computer Science, Carnegie Mellon University, December 1988. Archons Project Technical Report 88121.

[16] L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers*, 39(9):1175–1185, 1990.

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

[18] J. A. Stankovic, M. Spuri, K. Ramamritham, and G. C. Buttazzo. *Deadline Scheduling for Real-Time Systems*, chapter Chapter 3: (Fundamentals of EDF Scheduling), pages 27–67. Kluwer Academic Publishers, 1998.

[19] G. W. Stimson. *Introduction to Airborne Radar.* SciTech Publishing, second edition, January 1998.

[20] G. van Keuk and S. S. Blackman. On phased-array radar tracking and parameter control. *IEEE Transactions on Aerospace and Electronic Systems*, 29(1):186–194, January 1993.

[21] A. Varga. OMNeT++ Discrete Event Simulation System. `http://www.omnetpp.org/` (last accessed: January 22, 2005).

[22] 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 Real-Time and Embedded Computing Systems and Applications (RTCSA)*, August 2005.

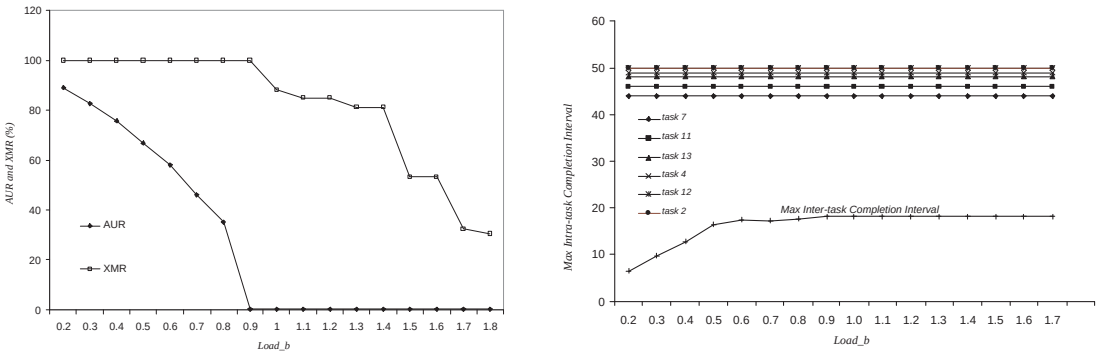
[23] H. Wu, B. Ravindran, E. D. Jensen, and U. Balli. Utility accrual scheduling under arbitrary time/utility functions and multiunit resource constraints. In *IEEE Real-Time and Embedded Computing Systems and Applications*, pages 80–98, August 2004.

Appendix A

Performance and Completion Intervals

A.1 Homogeneous VCFs

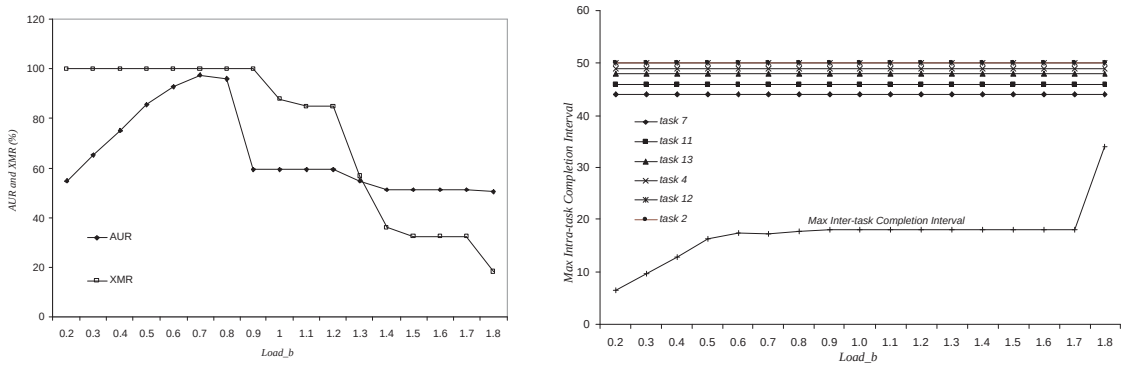
A.1.1 Decreasing VCF



(a) AUR and XMR

(b) Completion Intervals

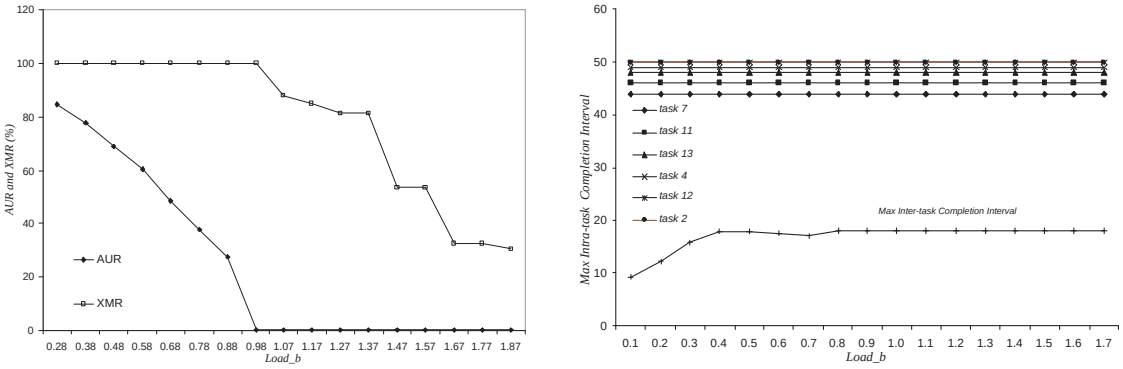
Figure A.1: CIC-VCUA Performance and Maximum Intra- and Inter-Task Completion Interval for Homogeneous Set, Decreasing VCFs, Decreasing TUFs



(a) AUR and XMR (b) Completion Intervals

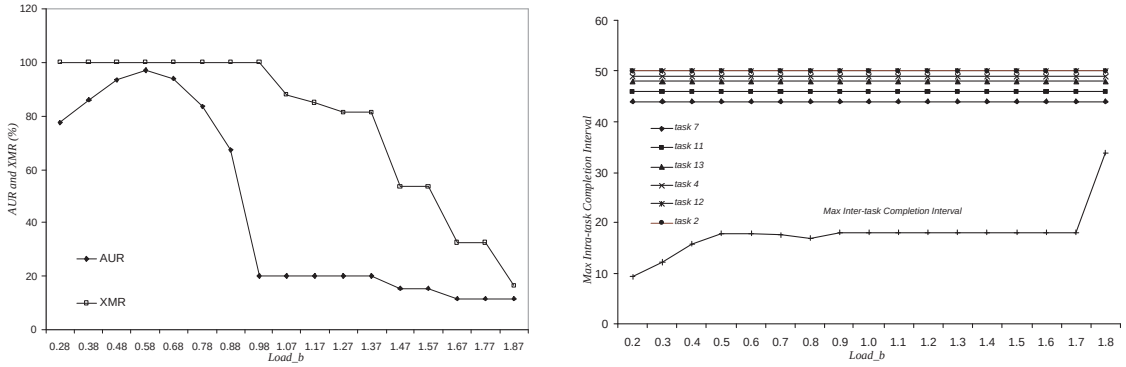
Figure A.2: CIC-VCUA Performance and Maximum Intra- and Inter-Task Completion Interval for Homogeneous Set, Decreasing VCFs, Parabolic TUFs

A.1.2 Increasing VCF



(a) AUR and XMR (b) Completion Intervals

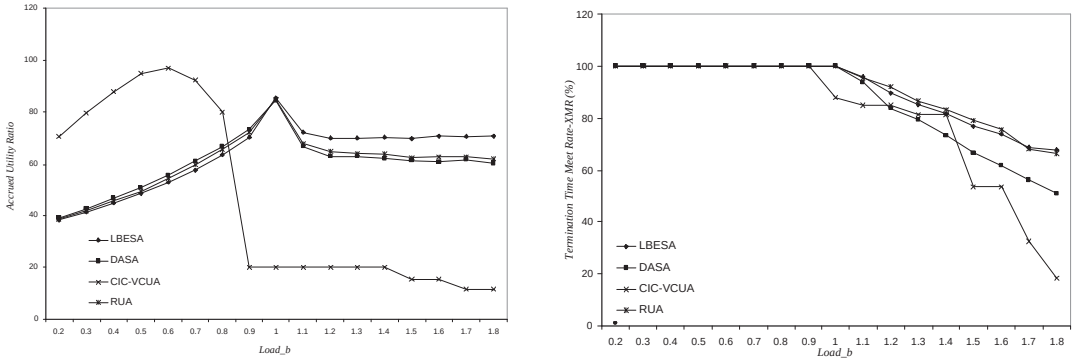
Figure A.3: CIC-VCUA Performance and Maximum Intra- and Inter-Task Completion Interval for Homogeneous set, Increasing VCFs, Decreasing TUFs



(a) AUR and XMR (b) Completion Intervals

Figure A.4: CIC-VCUA Performance and Maximum Intra- and Inter-Task Completion Interval for Homogeneous Set, Increasing VCFs, Parabolic TUFs

A.1.3 Constant VCF



(a) AUR (b) XMR

Figure A.5: CIC-VCUA vs other UA Algorithms, Constant VCFs, Parabolic TUFs

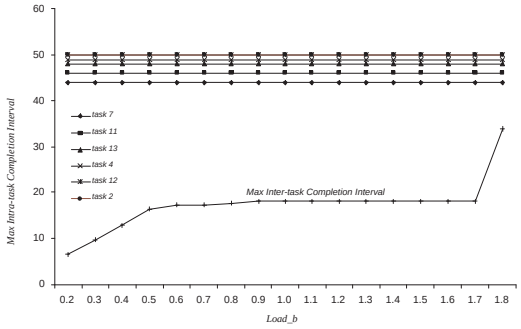
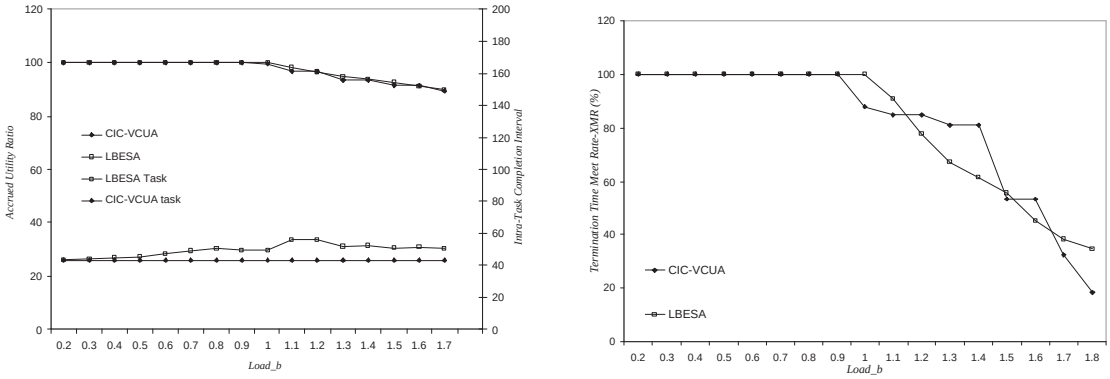


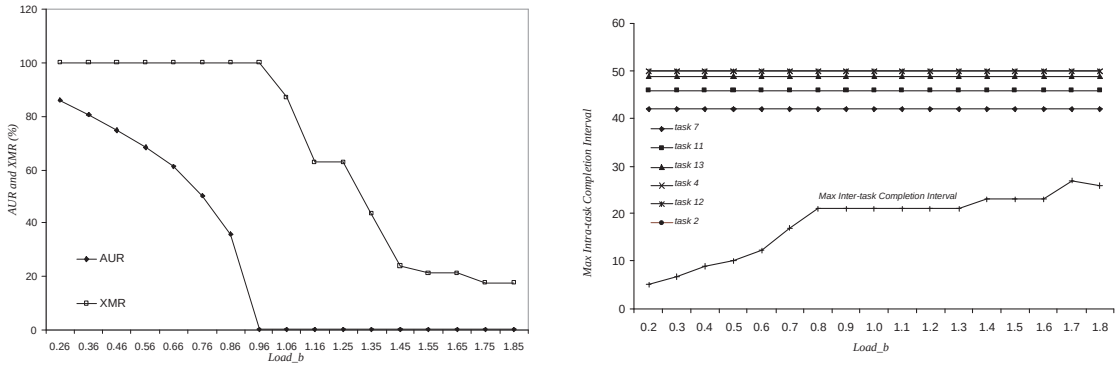
Figure A.6: Maximum Intra- and Inter-Task Completion Interval for Homogeneous Set, Constant VCFs, Parabolic TUFs



(a) AUR (b) XMR

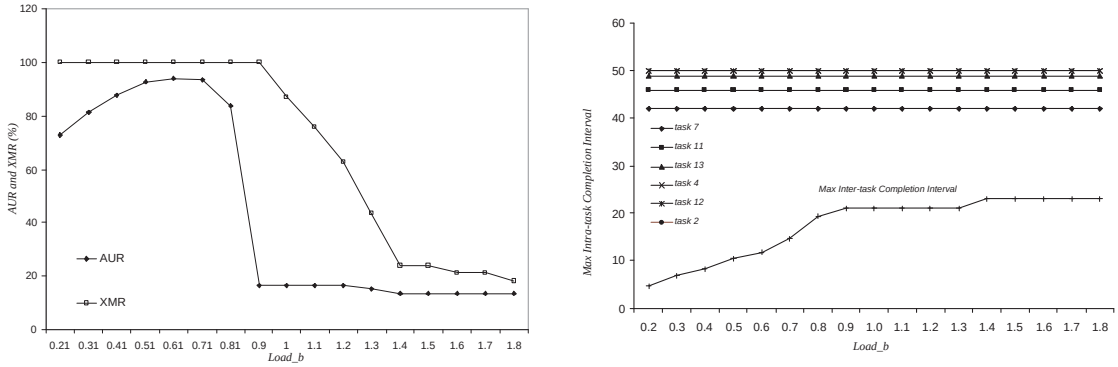
Figure A.7: CIC-VCUA Performance vs LBESA, Constant VCFs, Step TUFs

A.2 Heterogeneous VCFs



(a) AUR and XMR (b) Completion Intervals

Figure A.8: CIC-VCUA Performance and Maximum Intra- and Inter-Task Completion Interval for Heterogeneous Set, Mixed VCFs, Decreasing TUFs



(a) AUR and XMR (b) Completion Intervals

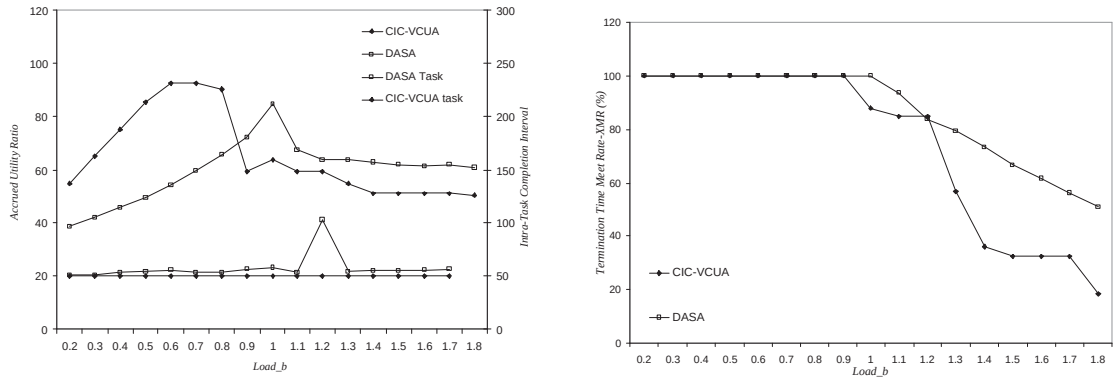
Figure A.9: CIC-VCUA Performance and Maximum Intra- and Inter-Task Completion Interval for Heterogeneous Set, Mixed VCFs, Parabolic TUFs

Appendix B

Performance under Resource Dependency

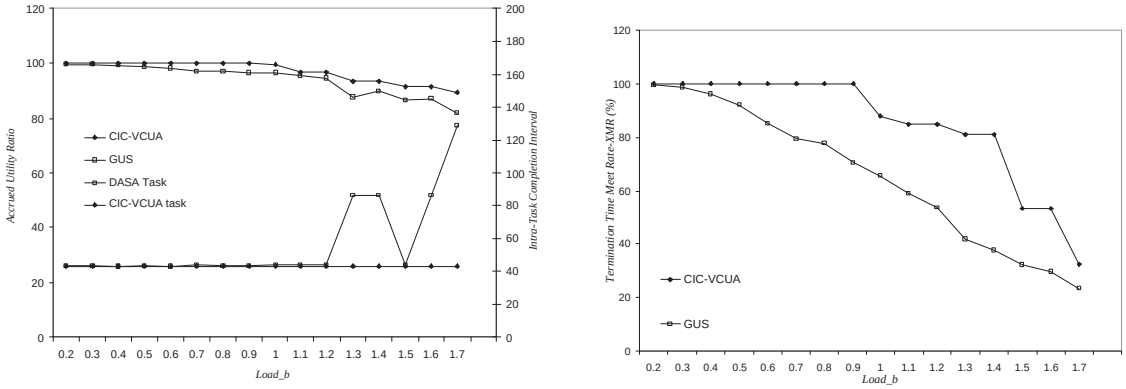
B.1 Homogeneous VCFs

B.1.1 Constant VCF



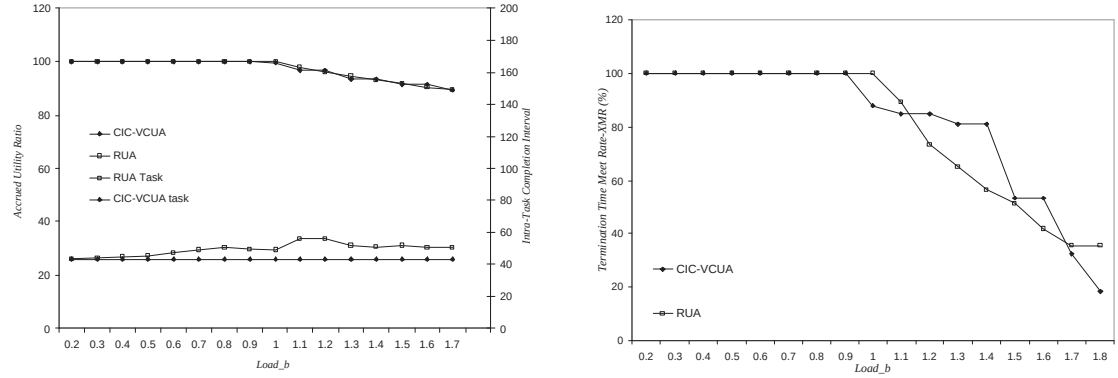
(a) AUR (b) XMR

Figure B.1: CIC-VCUA vs DASA under Resource Dependency, Constant VCFs, Parabolic TUFs



(a) AUR (b) XMR

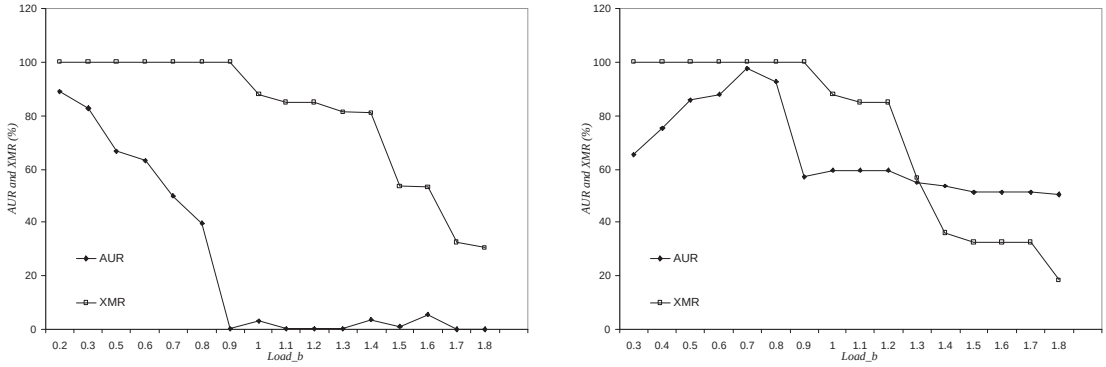
Figure B.2: CIC-VCUA vs GUS under Resource Dependency, Constant VCFs, Step TUFs



(a) AUR (b) XMR

Figure B.3: CIC-VCUA vs RUA under Resource Dependency, Constant VCFs, Step TUFs

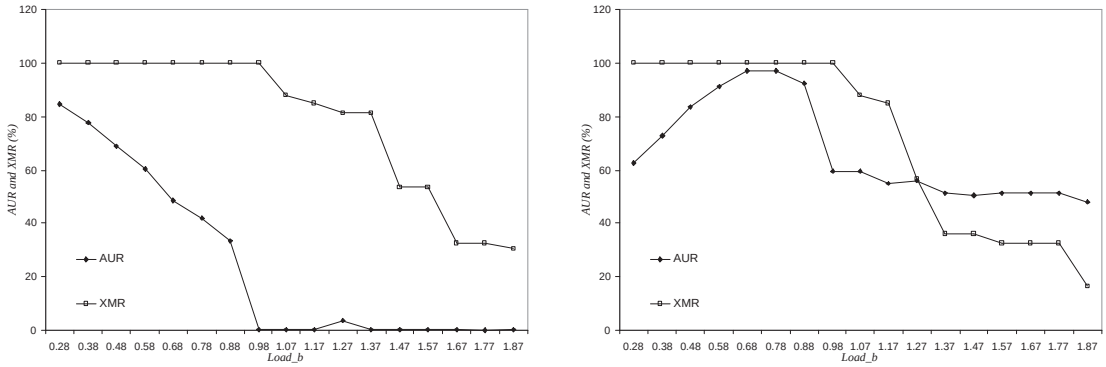
B.1.2 Decreasing VCF



(a) AUR and XMR; Decreasing TUF (b) AUR and XMR; Parabolic TUF

Figure B.4: CIC-VCUA Performance under Resource Dependency with Decreasing VCFs

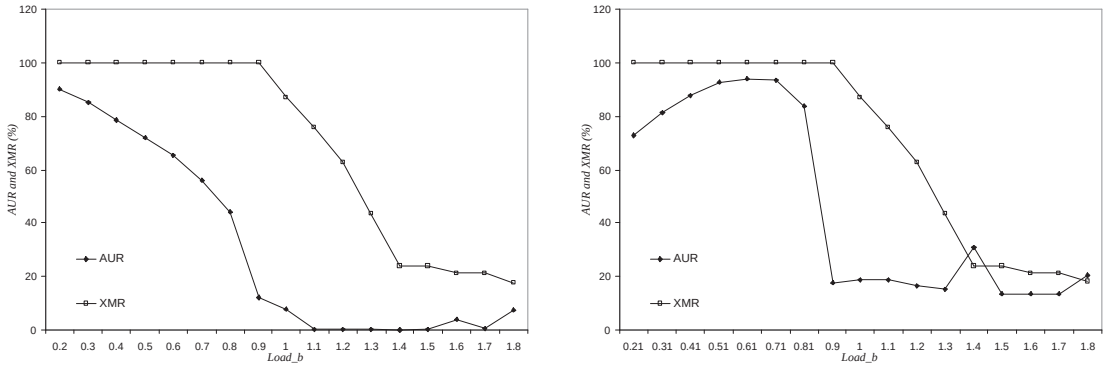
B.1.3 Increasing VCF



(a) AUR and XMR; Decreasing TUF (b) AUR and XMR; Parabolic TUF

Figure B.5: CIC-VCUA Performance under Resource Dependency with Increasing VCFs

B.2 Heterogeneous VCFs



(a) AUR and XMR; Decreasing TUF (b) AUR and XMR; Parabolic TUF

Figure B.6: CIC-VCUA Performance under Resource Dependency with Mixed VCFs