

Time/Utility Function Decomposition Techniques for Utility Accrual Scheduling Algorithms in Real-Time Distributed Systems

Haisang Wu^{*}, Binoy Ravindran^{*}, E. Douglas Jensen[†], and Peng Li[‡]

^{*}ECE Dept., Virginia Tech

[†]The MITRE Corporation

[‡]Microsoft Corporation

Blacksburg, VA 24061, USA

Bedford, MA 01730, USA

Redmond, WA 98052, USA

{hswu02,binoy}@vt.edu

jensen@mitre.org

pengli@microsoft.com

Abstract

We consider Real-Time CORBA 1.2’s *distributable threads* (DTs), whose time constraints are specified using time/utility functions (TUFs), operating in *legacy environments*. In legacy environments, system node resources—both physical and logical—are shared among time-critical DTs and local applications that may also be time-critical. Hence, DTs that are scheduled using their propagated TUFs, as mandated by Real-Time CORBA 1.2’s Case 2 approach, may suffer performance degradation, if a node utility accrual (UA) scheduler achieves higher locally accrued utility by giving higher eligibility to local threads than to DTs. To alleviate this, we consider decomposing TUFs of DTs into “sub-TUFs” for scheduling segments of DTs. We present decomposition techniques called *UT*, *SCEQF*, *SCALL*, *OPTCON*, and *TUFS*, which are specific to different classes of UA algorithms, such as those that use utility density, and those that use deadline as their key decision metric. Our experimental studies show that *OPTCON* and *TUFS* perform best for utility density-based UA algorithms, and *SCEQF* and *SCALL* perform best for deadline-based UA algorithms.

Index Terms

distributable thread, time/utility function, time constraint decomposition, Real-Time CORBA 1.2

I. INTRODUCTION

The Object Management Group’s recent Real-Time CORBA 1.2 standard [1] (abbreviated here as RTC2) and Sun’s upcoming Distributed Real-Time Specification for Java

(DRTSJ) [2] specify *distributable threads* (or *DTs*) as the programming and scheduling abstraction for system-wide, end-to-end scheduling in real-time distributed systems.¹ A DT is a single thread of execution with a globally unique identifier that transparently extends and retracts through local and remote objects. Thus, a DT is an end-to-end control flow abstraction, with a logically distinct locus of control flow movement within/among objects and nodes.

A DT carries its execution context as it transits node boundaries, including its scheduling parameters (e.g., time constraints, execution time), identity, and security credentials. Hence, DTs require that Real-Time CORBA’s *Client Propagated* model be used, and not the *Server Declared* model. The propagated thread context is used by node schedulers for resolving all node-local

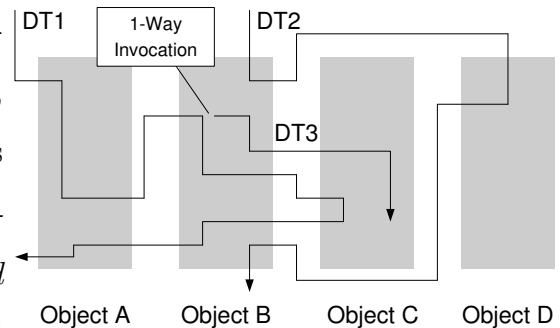


Fig. 1. Distributable Threads

resource contention among DTs such as that for node’s physical (e.g., CPU, I/O) and logical (e.g., locks) resources, and for scheduling DTs to optimize system-wide timeliness. Though this scheduling approach may only result in approximate, system-wide timeliness optimality, RTC2 explicitly supports it, called *Distributed Scheduling: Case 2* in the standard, due to its simplicity and capability for coherent end-to-end scheduling.² Figure 1 cited from [1] shows execution of DTs.

A. Time/Utility Functions and Utility Accrual Scheduling

In this paper, we focus on dynamic, adaptive, embedded real-time control systems at any level(s) of an enterprise—e.g., devices in the defense domain from phased array radars [6] up to entire battle management systems [7]. Such embedded systems include

¹Distributable threads first appeared in the Alpha OS [3], [4] and later in Alpha’s descendant, the MK7.3 OS [5].

²RTC2 also describes Cases 1, 3, and 4, which describe non real-time, global and multilevel distributed scheduling, respectively [1]. However, RTC2 does not support Cases 3 and 4.

time constraints that are “soft” (besides those that are hard) in the sense that completing an activity at any time will result in some (positive or negative) utility to the system, and that utility depends on the activity’s completion time. These soft time constraints are subject to optimality criteria such as completing all time-constrained activities as close as possible to their *optimal* completion times—so as to yield maximal collective utility. The optimality of the soft time constraints is generally as mission- and safety-critical as that of the hard time constraints.

Jensen’s time/utility functions [8] (or TUFs) allow the semantics of soft time constraints to be precisely specified. A TUF, which is a generalization of the deadline constraint, specifies the utility to the system resulting from the completion of an activity as a function of its completion time. Figure 2 shows example TUF time constraints: 2(a) shows an AWACS tracker’s time constraint [9]; 2(b) and 2(c) show a coastal air defense system’s time constraints [10]. Classical deadline is a binary-valued, downward “step” TUF; Figure 2(d) shows examples.

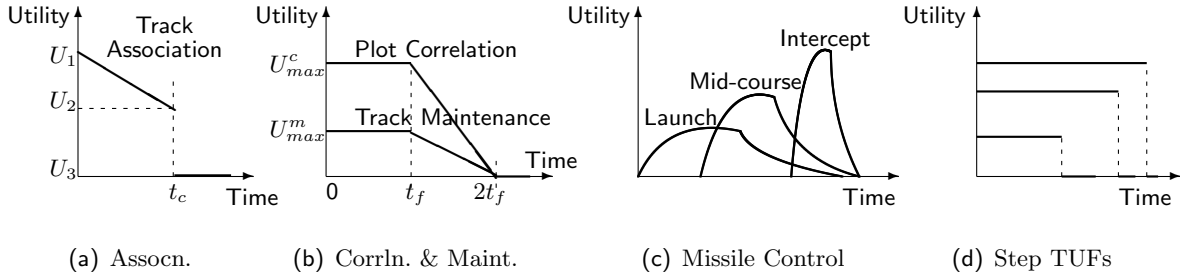


Fig. 2. Example TUF Time Constraints. (a): MITRE/TOG AWACS *association* TUF [9]; (b-c): GD/CMU coastal air defense *plot correlation*, *track maintenance*, & *missile control* TUFs [10]; (d): some step TUFs.

When activity time constraints are expressed with TUFs, the scheduling optimality criteria are based on maximizing accrued activity utility—e.g., maximizing the sum of the activities’ attained utilities. Such criteria are called *utility accrual* (or UA) criteria, and sequencing (scheduling, dispatching) algorithms that consider UA criteria are called UA sequencing algorithms. In general, the criteria may also include other factors such as resource dependencies. Several UA algorithms are presented in the literature [11]–[17].

RTC2 has IDL interfaces for the UA discipline, besides others such as fixed priority and earliest deadline first.

B. TUF Decomposition: Problem, Solution, and Paper Outline

In this paper, we consider integrating RTC2 applications, whose DT time constraints are specified using TUFs, into *legacy environments*. In legacy environments, a system node’s physical (processor, disk, I/O, etc.) and logical (locks, etc.) resources are shared between one or more RTC2 applications and other non-RTC2 applications, some of which include threads having TUF time constraints. We refer to such threads as “local threads.”

When DTs are subject to TUF time constraints, under RTC2’s Case 2 scheduling approach, scheduling of DTs is done using their propagated TUFs. The propagated TUFs are used by node-local UA scheduling algorithms to resolve local resource contentions and to construct local schedules that maximize *locally* accrued utility and approximate *global* accrued utility.

In legacy environments, resource-contention resolution and scheduling of DTs using their propagated TUFs may not always be the best approach—the propagated TUFs may fail to capture the timeliness requirements of DTs. For example, local threads on a node may always be favored by that node’s UA scheduler, at the expense of the DTs of RTC2 applications, because the node scheduler may find that favoring the local threads leads to higher *locally* accrued utility (due to the particular shape of the TUFs of local threads, for example). However, higher local utility does not necessarily imply higher system-wide utility in terms of the sum of utilities attained by all DTs, which is our optimization objective. Thus, DTs of an RTC2 application can suffer interference from local threads, and perform poorly.

Besides the shape of TUFs, other factors may also affect the performance of DTs. Example factors include the mixture of local threads and DTs, laxity of those local threads with deadlines, execution times of DTs and local threads, and node UA scheduling algorithms.

To help a DT properly compete with local threads and improve its performance in legacy environments, a “sub-TUF” for each segment of the DT can be derived from the DT’s TUF for node-local resource-contention resolution and UA scheduling. We call this process, *decomposition* of the TUF of a DT. Since we consider real-time distributed systems whose nodes use UA scheduling algorithms (as DTs are subject to TUF time constraints and UA scheduling optimality criteria), the DT TUF decomposition problem (abbreviated here as DTD) naturally raises fundamental questions such as “how to decompose TUFs for UA scheduling algorithms, both step shaped and non-step shaped?” Furthermore, “under what conditions can TUF decomposition help DTs improve their performance in legacy environments?”

In this paper, we answer these questions. We broadly categorize UA scheduling algorithms based on their key decision-making metrics such as potential utility density (e.g., DASA [13], GUS [11]) and deadline (e.g., LBESA [12]). For each UA algorithm class, we present class-appropriate decomposition techniques that derive sub-TUFs for DT segments. Our DTD techniques are called *UT*, *SCEQF*, *SCALL*, *OPTCON*, and *TUFS* (see Section IV for details on the techniques).

We conduct extensive and comprehensive experimental studies to identify the conditions that affect the performance of the DTD techniques for each algorithm class. Further, we identify those DTD techniques that perform superiorly for each class by considering a variety of factors including TUF shapes, resource dependencies, system load, and mixture of local threads and DTs. The major result from our study is that *OPTCON* and *TUFS* perform best for potential utility density-based UA algorithms, and *SCEQF* and *SCALL* perform best for deadline-based UA algorithms (see Section V-C that summarizes our comprehensive results).

Thus, the paper’s contribution includes identification and formulation of the DTD problem, decomposition techniques, and determination of superior DTD techniques for each UA algorithm class. To the best of our knowledge, we are unaware of any other efforts that have studied the DTD problem (though deadline-decomposition—a special

case of the DTD problem—has been studied).

The rest of the paper is organized as follows: In Section II, we describe our application, timeliness, and system models. Section III describes the scheduling algorithms that we consider, and lists the possible factors affecting the DTD problem. Section IV discusses our decomposition techniques. In Section V, we describe our experimental setup, experimental evaluation, and analyze the results. In Section VI, we overview past research on time constraint decomposition in real-time distributed systems and contrast them with our work. Finally, the paper concludes and identifies future work in Section VII.

II. THE APPLICATION, TIMELINESS, AND SYSTEM MODELS

A. The Thread Model

We assume that the application consists of a set of DTs and local threads. A local thread is generated and executed at one and only one node. DTs follow the exact same semantics as in RTC2 and DRTSJ. A DT executes in objects that are distributed across computing nodes by location-independent invocations and returns. A DT always has a single execution point that will execute at a node when it becomes “most eligible” as deemed by the node scheduler. Within each node, a DT’s flow of control is equivalent to normal, local thread execution. One possible implementation is to map a DT to a local thread while it is executing in each node. We will refer to each node-local segment of a DT as an `object-level thread` (or simply as `OLT`) hereafter. Therefore, a DT can be assumed to consist of a series of OLTs that the DT is mapped to along its execution path.

As a shorthand, we use the notation $DT_k = \{OLT_i^k : 1 \leq i \leq m_k\}$ to represent the k^{th} DT that consists of m_k OLTs, where OLT_i^k means the i^{th} segment of DT_k .

We assume that the locus of control flow movement of the DTs are known. Thus, the chain of method invocations of each DT is assumed to be a-priori known. For many applications, it is possible to obtain this knowledge by static code analysis, such as for the GD/CMU air defense system [10]. Of course, for some applications, such static code

analysis will be difficult and hence it will be difficult to a-priori know the DT chain of method invocations.

B. Resource Model

Local threads and OLTs of different DTs can access non-CPU resources, which in general, are serially reusable. Examples include physical (e.g., disks) and logical (e.g., locks) resources.

Similar to fixed-priority resource access protocols (e.g., priority inheritance, priority ceiling) [18] and resource access mechanisms for UA algorithms [11], [13], we consider a single-unit resource model. Thus, only a single instance of a resource is present and an OLT or a local thread must explicitly specify the resource that it wants to access.

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

OLTs of different DTs can have precedence constraints. For example, an OLT OLT_i^k can become eligible for execution only after an OLT OLT_j^l has completed, because OLT_i^k may require results of OLT_j^l . As in [11], [13], we program such precedences as resource dependencies.

C. The System Model

We consider a distributed system architecture model, where a set of processing components, generically referred to as *nodes*, are interconnected via a communication network (Figure 3). Each node services OLTs of DTs and local threads generated at the node. The order of executing the threads—OLTs and local threads—on a node is determined by the scheduler that resides at the node. We consider RTC2’s Case 2 approach for thread scheduling—i.e., node schedulers do not collaborate and independently construct

schedules. Thus, scheduling decisions made by a node scheduler are independent of that made by other node schedulers.

Node schedulers make scheduling decisions using thread scheduling attributes, which typically include threads' time constraints (e.g., deadline, TUF), importance, and remaining execution time.

Nodes are assumed to be homogeneous in terms of hardware configurations (e.g., processing speed, instruction pipeline, primary memory/cache resources) and node scheduling algorithm (i.e., all nodes run the same scheduling algorithm).

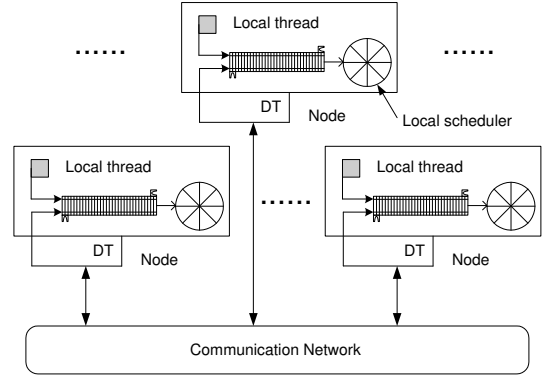


Fig. 3. The Distributed System Model

D. Timeliness Model

We use TUF to specify the time constraint of a DT, an OLT, or a local thread, generically referred to as T , and denote the TUF of a thread T as $U(T)$. Thus, the completion of T at a time t will yield a timeliness utility $U(T, t)$. DTs propagate their TUFs as they transit nodes, and we decompose the propagated TUF of a DT into sub-TUFs for node-local scheduling.

Though TUFs can take arbitrary shapes, here we focus on *unimodal* TUFs. Unimodal TUFs are those that have a *single*, optimal, completion time interval. Figure 2 shows examples. Our motivation to consider this TUF class is that they encompass the majority of time constraints in our motivating applications (see Figure 2).

Each TUF $U(T)$ is assumed to have an initial time $ar(T)$ and a deadline time $dl(T)$. Initial time is the earliest time for which the function is defined, while deadline time is the latest time at which the function drops to a zero utility value. In this paper, we assume that $ar(T)$ is equal to the arrival time of a thread T , which is simply the time at which the thread becomes ready for execution. The arrival time of a DT is the arrival

time of the first OLT of the DT. We also assume that $U(T)$ is defined from $ar(T)$ until the future indefinitely. Thus, the term “deadline time” (or “deadline” in abbreviation) is used to denote the last point that a TUF crosses the t -axis. For a downward step TUF, the deadline time is its discontinuity point. Further, we assume that $U(T, t) \geq 0, \forall t \in [ar(T), dl(T)]$ and $U(T, t) = 0, \forall t \geq dl(T)$.

The execution time of a DT/OLT/local thread T is denoted as $ex(T)$. We incorporate the transmission delays of all inter-OLT messages of a DT into the DT’s execution time. Thus, a DT’s execution time simply denotes the total workload of all its OLTs.

III. UA SCHEDULERS AND FACTORS CONSIDERED IN THE DTD PROBLEM

The type of the scheduling algorithm employed at a node can impact the DTD problem and thus DT performance. We consider UA scheduling algorithms including the Generic Utility Scheduling algorithm (GUS) [11], Dependent Activity Scheduling Algorithm (DASA) [13], Locke’s Best Effort Scheduling Algorithm (LBESA) [12], and D^{over} [14], and a non-UA algorithm—the Earliest Deadline First (EDF) [19] for comparison. LBESA, D^{over} , and EDF consider independent threads, while GUS and DASA allow threads to share resources.

These algorithms make scheduling decisions based on different metrics. EDF exclusively considers a thread’s deadline, and suffers significant domino effect during overloads [12]. D^{over} is a timer-based UA algorithm, which requires a timer for *Latest Start Time*. GUS, DASA, and LBESA use a metric called *Potential Utility Density* (or PUD). The PUD of a thread simply measures the amount of utility that can be accrued per unit time by executing the thread and the thread(s) that it depends upon for obtaining (non-CPU) resources.

At each scheduling event, GUS builds the dependency chain for each thread in the ready queue, and appends to the existing output schedule the highest PUD thread with the threads on which it depends. DASA also calculates PUD for each thread after building its dependency chain; it then inserts the threads, from high PUDs to low PUDs, into a

tentative schedule in an EDF order. Upon inserting a thread, DASA performs feasibility check, and ensures that the predicted completion time of each thread left in the tentative schedule never exceeds its deadline. DASA and D^{over} only deal with step TUFs, while GUS and LBESA can deal with arbitrary shapes of TUFs. LBESA examines threads in an EDF order, and performs feasibility check, where it rejects threads with low PUDs until the schedule is feasible.

GUS, DASA, and LBESA have the best performance among existing UA algorithms [11]. Moreover, DASA and LBESA mimic EDF to reap its optimality during under-loads. For independent threads with step TUFs, D^{over} has the optimal competitive factor [14].

Different characteristics of the scheduling algorithms require different DTD mechanisms to improve DT performance. To find good techniques for UA algorithms, we also analyze other factors that can influence the DTD problem, potentially affecting DT performance:

1) **TUF shape.** TUF shapes can affect DTD and thread scheduling. For example, it would be beneficial to schedule a thread with a decreasing TUF as early as possible to accrue high utility. However, it may be beneficial to delay the execution of a thread with a strictly concave TUF, so as to complete it at the time corresponding to the optimum TUF value. In this paper, we only consider unimodal TUFs (e.g., step TUFs, linear TUFs). Further, different scheduling algorithms focus on TUFs with different shapes.

2) **Task load.** We define *task load* (or *load*, for short) as the ratio of the rate of work generated to the total processing capacity of the system. The analytical expression for load is given in Section V-A starting from Page 15. Different scheduling algorithms produce different behaviors under different loads; thus, load may affect DTD for these algorithms. For example, EDF is optimal during under-load situations in terms of satisfying all deadlines [20], but suffers domino effects during over-load situations [12]. DASA and LBESA are also optimal during under-loads, but provides much better performance during overloads than EDF [12], [13]. To deal with arbitrary shapes of TUFs, GUS does not maintain the EDF order for the ready queue, so it yields different timeliness utility from EDF, DASA and LBESA [11].

3) **Global Slack Factor (GSF)**. We define a DT's *Global Slack Factor* (GSF) as the ratio of the DT's execution time to its TUF's definition period, which is the sum of the DT's execution time and its slack. For example, Figure 4 shows the slack and deadline time of a DT with four segments ($m = 4$). Thus, the DT's GSF which describes its global stack is $GSF(T) = \frac{ex(T)}{dl(T)-ar(T)}$. Intuitively, the larger $GSF(T)$, the less global slack of T ,

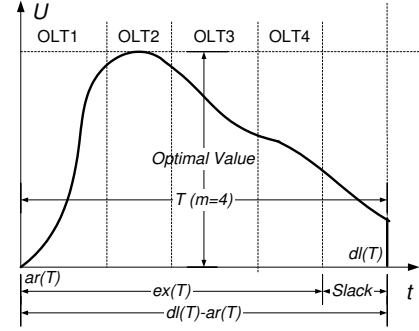


Fig. 4. Global Slack of a TUF

which means more stringent time constraint is imposed on T , and it is more prone to complete after its deadline and accrue zero utility. Thus, in the DTD process, a DT's GSF can be considered to change the relative importance of an OLT to compete with local threads.

4) **Homogenous/heterogeneous TUFs**. Each DT's end-to-end time constraint can possibly be specified using different TUFs. This can lead to DTs having homogenous or heterogenous TUFs. As different TUF shapes can affect the performance of DTD, it is possible that a given DTD strategy can perform better for one kind of TUFs while perform poorly for another kind of TUFs. Thus, TUF homogeneity/heterogeneity can affect TUF decomposition and DT performance.

5) **Resource Dependencies**. It is possible that there exist resource dependencies among OLTs of different DTs, among local threads, and among OLTs and local threads. Such dependencies include resource access constraints such as mutual exclusion constraints and inter-thread precedence relationships. Dependencies cause additional interference to the contention among OLTs and local threads, under which a thread is more likely to miss its deadline. Thus, DTD under dependencies is more unlikely to improve DT performance.

6) **Local threads**. On each node, the OLTs of DTs compete with each other, and they also compete with the local threads. In addition, there also exists contention among local threads. The three types of contention, which we describe as global-global, global-local,

and local-local, should be resolved by the node-local schedulers. Even though we do not decompose the TUFs of local threads, their properties will affect two types of contention, i.e., global-local and local-local, which can in turn affect the performance of DTs whose TUFs are decomposed and allocated to their OLTs.

IV. TUF DECOMPOSITION METHODS

We categorize UA scheduling algorithms based on their key decision-making metrics as follows: D^{over} is a timer-based algorithm; EDF is completely deadline-based, and GUS is PUD-based. LBESA and DASA consider both PUDs and deadlines in their scheduling, and thus they fall between the classes of EDF and GUS. But a more precise categorization of the algorithms cannot be given until we experimentally test their performance (we do so in Section V-B.1).

Based on this broad algorithm classification, we propose heuristics to derive DTD methods. The DTD methods that we propose can be broadly categorized into three classes: (1) those that change no properties (we describe this in Section IV-A); (2) those that change an OLT's deadline time (Sections IV-B and IV-C describe this); and (3) those that change an OLT's PUD (we describe this in Sections IV-D and IV-E).

A. *Ultimate TUF (UT)*

Without any specific knowledge on the execution times of the OLTs, the only available measure of their time requirement is the deadline and shape of their DT's TUF. Thus, a simple strategy is to set the deadline time of an OLT to be equal to the deadline time of its DT, i.e., $dl(OLT_i^k) = dl(DT_k), 1 \leq i \leq m_k$, and to set $U(OLT_i^k) = U(DT_k)$. We call this strategy, **Ultimate TUF (UT)**. Thus, *UT* does not decompose a DT's TUF. The propagated (ultimate) TUF is used by all node schedulers for scheduling OLTs of a DT.

B. *Scaling based on EQF (SCEQF)*

A problem with UT is that the time for the execution of a later OLT of a DT is considered slack to an earlier OLT. This may give the scheduler incorrect information about how much time an OLT can be delayed in its execution. To avoid this, we can “slice” the deadline of a DT’s TUF to change the deadline time for each of its OLT. We adopt the Equal Flexibility (or EQF) strategy presented in [21] for such slicing. EQF decomposes the end-to-end deadline of a global task (a DT in our case)

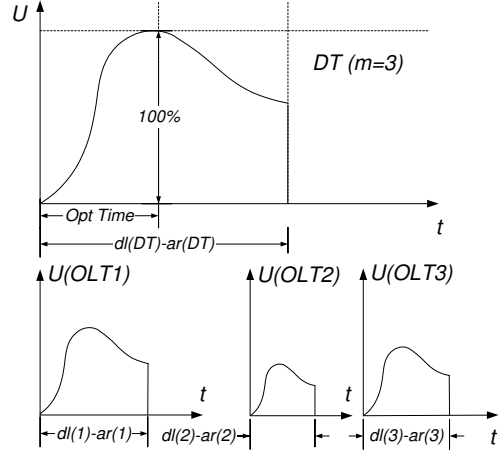


Fig. 5. The $SCEQF$ Technique

into deadlines for subtasks (OLTs in our case), by dividing the total remaining slack among the OLTs in proportion to their execution times; it can significantly improve the performance of global tasks [21]. The slicing is done such that higher the execution time of a subtask, longer is its deadline.

After deriving the deadlines of the OLTs of a DT from its TUF using EQF , we also change the height of a sub-TUF to convey more accurate information to the node-local scheduler. We call this DTD strategy **Scale on EQF** ($SCEQF$). The sub-TUF of an OLT is defined as the scaled TUF of the DT between the OLT’s arrival time and deadline. The scaling factor is selected to be $\frac{dl(OLT_i^k) - ar(OLT_i^k)}{dl(DT_k) - ar(DT_k)}$ to make the sub-TUFs’ heights in proportion to the OLTs’ relative deadline times.

Figure 5 illustrates the method with an example DT whose segment number m is 3.

C. Scaling based on TUF shape ($SCALL$)

We also try to slice the deadline of a DT’s TUF based on its shape. We define the time at which the TUF reaches its extremum as Opt (for unimodal TUFs that we consider here, the extremum is the maximum). The utility value that corresponds to the Opt time is denoted $OptValue$. One intuition in DTD is that, a DT should complete near its Opt to accrue as much utility as possible. Thus, we modify $SCEQF$ so that only

when $ar(OLT_i^k) + ex(OLT_i^k) > Opt$, we derive $dl(OLT_i^k)$ by EQF. Otherwise, we set $dl(OLT_i^k) = Opt$.

We then scale the height of the TUF by the factor $\frac{dl(OLT_i^k) - ar(OLT_i^k)}{dl(DT_k) - ar(DT_k)}$, and allocate the scaled sub-TUF to each OLT. We call this strategy the **SCALL** method.

D. Decomposing into Linear-Constant TUF (OPTCON)

To change the PUD of an OLT, we design a method to decompose the DT's TUF into a linear-constant TUF and allocate it to the OLT, without slicing the deadline of a DT's TUF as *SCEQF* and *SCALL*. This method, which is called **OptValue and Constant Value Function (OPTCON)**, is illustrated in Figure 6.

The terms *Opt* and *OptValue* have the same meanings as before. As shown in the figure, for DT_k and its i^{th} segment OLT_i^k , we first set $dl(OLT_i^k) = dl(DT_k)$. The times t_1 and t_2 are then defined as $t_1 = ar(OLT_i^k)$ and $t_2 = ar(OLT_i^k) + ex(OLT_i^k)$. *OPTCON* decomposes a DT's TUF to allocate sub-TUFs to OLTs with the following steps (Figure 6 also illustrates the steps):

(1) Let the OLT_i^k obtain its highest utility at its expected finish time t_2 , which means we set $U(OLT_i^k, t_2) = OptValue$.

(2) We then increase the TUF of the OLT from time t_1 with utility value $U(OLT_i^k, t_1)$ linearly until the utility value $OptValue$ at time t_2 , after which we keep the TUF constant at $OptValue$ until its deadline time $dl(OLT_i^k)$.

(3) The TUF of the OLT_i^k is then scaled by a factor $fctr$ so that the expected PUD of the OLT is increased. The factor is determined as: $fctr =$

$$\max\left(\frac{PUD_{local}}{PeakPUD_{OLT_i^k}}, 1\right), \text{ where } PeakPUD_{OLT_i^k} = \frac{OptValue}{ex(OLT_i^k)}, \text{ and } PUD_{local} \text{ is the PUD of}$$

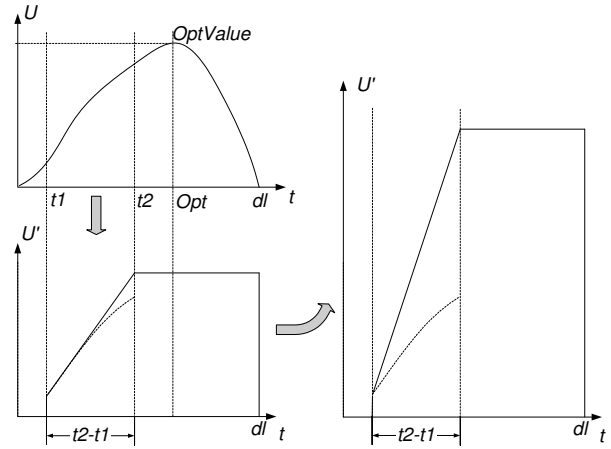


Fig. 6. The *OPTCON* Technique

local threads. Therefore, $fctr$ is chosen in such a way that an OLT's TUF is not scaled if its peak PUD is larger than the PUD of local threads, otherwise it is scaled *up* by the ratio of PUDs of local threads to PUDs of OLTs. The OLT's TUF will never be scaled down because our goal is to improve DTs' performance instead of local threads' performance.

E. Scaling the TUF (TUFs)

Instead of the complicated method *OPTCON*, for comparison, we also consider a simple strategy to improve the OLT's PUD that is seen by the local scheduler. This is realized by: (1) using the DT's deadline as the OLT's deadline, and (2) scaling the DT's propagated TUF and assigning the scaled TUF as the OLT's TUF. The scaling of the DT's TUF can be done by using an factor ($fctr$) that is determined exactly the same as in *OPTCON*. We call this strategy **Time/Utility Function Scaling (TUFs)**.

V. EXPERIMENT EVALUATION

We experimentally study the DTD strategies by conducting simulation experiments. We believe that simulation is an appropriate tool for this study as that would allow us to evaluate a number of different DTD techniques under a broad range of system conditions. We first present the simulation model, and then discuss the results.

A. Simulation Model

Our simulator is written with the simulation tool OMNET++ [22], which provides a discrete event simulation environment. Each simulation experiment (generating a single data point) consists of three simulation runs with different random seeds, each lasting 200 sec (at least 10,000 events are generated per run; many more for high load experiments). Since the basic time unit in OMNET++ is a second, we will refer to a time unit as a second hereafter. The structure of our simulation model follows the conceptual model described in Section II, with the following characteristics:

- **Nodes.** There are k homogeneous nodes in the system. Each node services their threads (OLTs of DTs and local threads) according to a given real-time scheduling algorithm. We consider both UA and non-UA scheduling algorithms such as GUS, DASA, LBESA, D^{over} , and EDF for our experiments.
- **Local threads.** Local threads are generated at each node according to a Poisson distribution with mean inter-arrival time $1/\lambda_{local}$ seconds. (Poisson distributions are typically used in experimental studies like ours because of their simplicity and because they yield useful insights.) Since there are k nodes, the total average arrival rate is k/λ_{local} per second. Execution times of local threads are exponentially distributed with mean in $1/\mu_{local}$ seconds. The rate of work due to local threads is thus $k\lambda_{local}/\mu_{local}$.
- **DTs.** Similar to local threads, DTs are generated as n streams of Poisson processes with mean inter-arrival $1/\lambda_{DT}$ time. For simplicity, we assume that DTs are homogeneous. In particular, we assume that all DTs consist of the same number of segments, and the execution times of all the segments (OLTs) follow the same exponential distribution with a mean $1/\mu_{OLT}$ seconds. We assume that the number of OLTs contained in a DT is m . The total execution times of DTs thus follow an m -stage Erlang distribution with mean m/μ_{OLT} . The rate of work due to DTs is therefore $m\lambda_{DT}/\mu_{OLT}$. The execution node of an OLT is selected randomly and uniformly from the k nodes.
- **System Load.** We define the *normalized load* (or *load*, for short) as the ratio of the rate of work generated to the total processing capacity of the system. That is,

$$load = \left(\frac{n \cdot m \cdot \lambda_{DT}}{\mu_{OLT}} + \frac{k \cdot \lambda_{local}}{\mu_{local}} \right) / k \quad (1)$$

For a stable system, we have $0 \leq load \leq 1$. We also define *frac_local* as the least fraction of PUD that can possibly be contributed by local threads in node-local scheduling, i.e.,

$$frac_local = \frac{PUD_{local}}{PUD_{local} + PeakPUD_{OLT}} \quad (2)$$

- **TUFs.** Different threads have different TUFs. Each DT has a propagated TUF, and we define five classes of TUFs to evaluate our methods. The parameter setting of our baseline experiment is summarized in Table I. We select different shapes of TUFs, but

keep $OptValue$ of each TUF to be 10, and its deadline time to be 20. In Table I, TUF_1 is the right half of a quadratic function in the first quadrant, and TUF_2 is a linear function, so they are both non-increasing. TUF_3 is a downward step function. TUF_4 is a complete quadratic function in the first quadrant, so it is strictly concave. TUF_5 is the combination of different TUFs. The TUFs of local threads are downward step functions with variable heights.

TABLE I
BASELINE SETTINGS

Overload Management Policy	Abort threads later than deadlines
Local Scheduler	GUS, DASA, LBESA, D^{over} , EDF
μ_{OLT}	1.0 (When $load$ varies)
μ_{local}	1.0 (When $load$ varies)
λ_{DT}	1/8.33 (When GSF varies)
λ_{local}	1/8.33 (When GSF varies)
k (#of nodes)	8
m (# of OLTs in a DT)	4
n (# of DT streams)	6
$frac_{local}$	0.80 (When $load$ or GSF varies)
TUF_1	$f_1(t) = \begin{cases} -0.025t^2 + 10, & \text{when } 0 \leq t \leq 20 \\ 0, & \text{otherwise} \end{cases}$
TUF_2	$f_2(t) = \begin{cases} -0.5t + 10, & \text{when } 0 \leq t \leq 20 \\ 0, & \text{otherwise} \end{cases}$
TUF_3	$f_3(t) = \begin{cases} 10, & \text{when } 0 \leq t \leq 20 \\ 0, & \text{otherwise} \end{cases}$
TUF_4	$f_4(t) = \begin{cases} -0.1t^2 + 2t, & \text{when } 0 \leq t \leq 20 \\ 0, & \text{otherwise} \end{cases}$
TUF_5	combination of $TUF_1 \sim TUF_4$
Local threads	Downward Step Functions

B. Experimental Results

The primary performance metric that we use to evaluate the DTD methods is accrued utility ratio (or AUR). AUR is simply the ratio of the accrued utility to the maximum possible total utility (which is the sum of each DT's maximum utility).

We use the notation $AUR_A^B(C)$ to denote the AUR obtained under a scheduling algorithm $A \in \{GUS, DASA, LBESA, D^{over}, EDF\}$, a DTD technique $B \in \{UT, SCEQF, SCALL, OPTCON, TUFs\}$, and a TUF $C \in \{TUF_1, TUF_2, TUF_3, TUF_4, TUF_5\}$. Thus, for example, $AUR_{GUS}^{SCEQF}(TUF_4)$ denotes the AUR that DTs can accrue under the GUS

scheduler, *SCEQF* decomposition technique, and *TUF₄*.

In Section III, we evaluate the effects of all factors presented on system performance in a collective way, so all simulations are performed with variation of several parameters describing the factors.

1) *DTD Methods with Different Scheduling Algorithms*: We first study how our designed DTD techniques work on different scheduling algorithms. We consider downward step functions (i.e., *TUF₃*) for the DTs, as all the algorithms, with the exception of GUS and LBESA, cannot deal with arbitrarily-shaped TUFs. Further, we do not consider resource dependencies, as LBESA, *D^{over}*, and EDF cannot directly address resource dependencies.

The TUFs of local threads are set to downward step functions with heights of 40, and the maximum heights of DT TUFs are bounded at 10. Thus, the *frac_{local}* in these experiments is 0.8. Since we only focus on *TUF₃* here, for simplicity, we denote $AUR_A^B(TUF_3)$ as AUR_A^B . AURs of different methods are recorded as the *load* and *GSF* varies from 0 to 1. When we refer to the *GSF* of a task set, it has the same value as the *GSF* of each task in the set. As shown in Table I, when *load* varies, we keep the execution time $\mu_{OLT} = \mu_{local} = 1.0$, but change the mean inter-arrival time $1/\lambda_{DT}$ and $1/\lambda_{local}$; when *GSF* varies, we keep $\lambda_{DT} = \lambda_{local} = 1/8.33$, but change μ_{OLT} and μ_{local} .

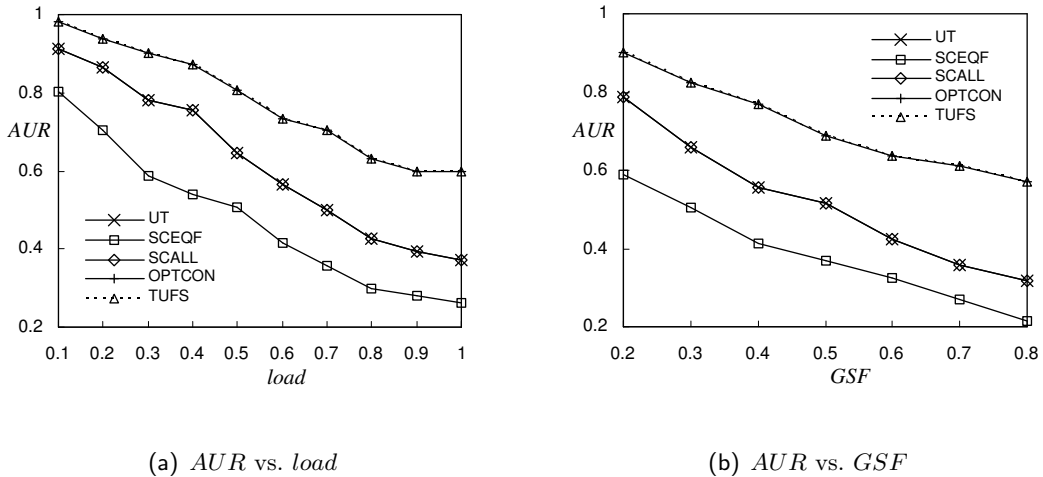


Fig. 7. Accrued Utility Ratio with GUS under *TUF₃*

Figure 7 shows AUR_{GUS} of the DTD methods under varying $load$ and GSF . From Figure 7, we observe that the curves in both figures indicate similar trends. For example, in Figure 7(a), when $load$ increases, AUR_{GUS} of all DTD strategies decreases. This is reasonable, because more threads miss their deadlines and thus less utilities are accrued. But GUS under different DTD strategies accrues different utilities, even when the $load$ is very light. AUR_{GUS}^{SCEQF} has the worst performance among all strategies. AUR_{GUS}^{UT} and AUR_{GUS}^{SCALL} are identical because they perform the same operations with downward step functions, and they perform better than AUR_{GUS}^{SCEQF} . The methods $OPTCON$ and $TUFS$ perform the best. For example, at $load = 0.6$, $AUR_{GUS}^{SCEQF} = 42\%$, while $AUR_{GUS}^{SCALL} = AUR_{GUS}^{UT} = 56\%$, and $AUR_{GUS}^{OPTCON} = AUR_{GUS}^{TUFS} = 73\%$.

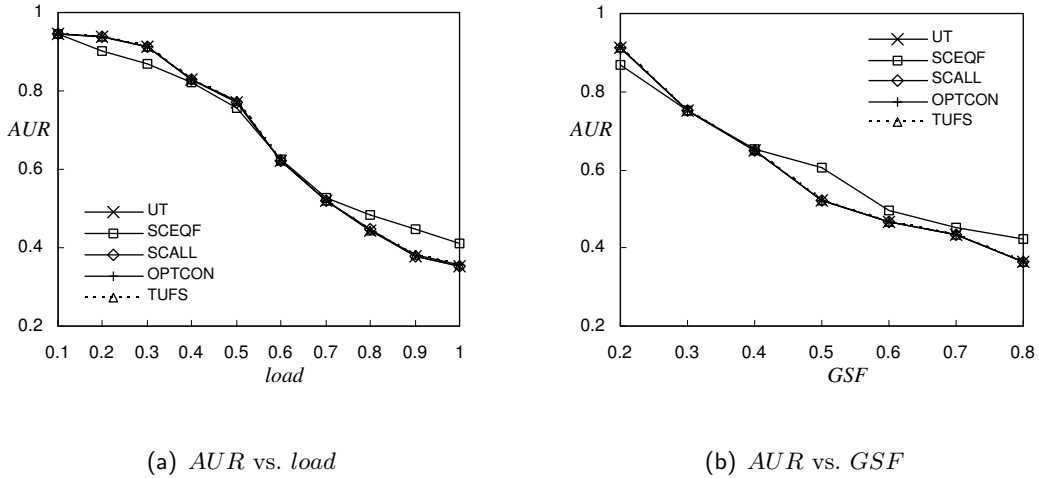


Fig. 8. Accrued Utility Ratio with LBESA under TUF_3

The performance results of the DTD strategies under the LBESA algorithm are shown in Figures 8(a) and 8(b). In both figures, $SCEQF$ performs better than others, especially at high $load$ and GSF ; and the other four strategies produce almost identical results.

Figure 9 shows the performance of DTD techniques under the DASA algorithm, which has very similar trends as those under GUS. From the figure, we observe that under DASA, as $load$ and GSF varies, the performance trend is $OPTCON = TUFS > UT = SCALL > SCEQF$.

In Figure 10, we show performance of the DTD methods under the D^{over} algorithm.

The AUR of DTs in Figure 10 drops to under 0.2 when $load$ or GSF is larger than 0.5. All curves are very close to each other, which indicates that the different DTD methods have little effect on D^{over} 's scheduling decisions.

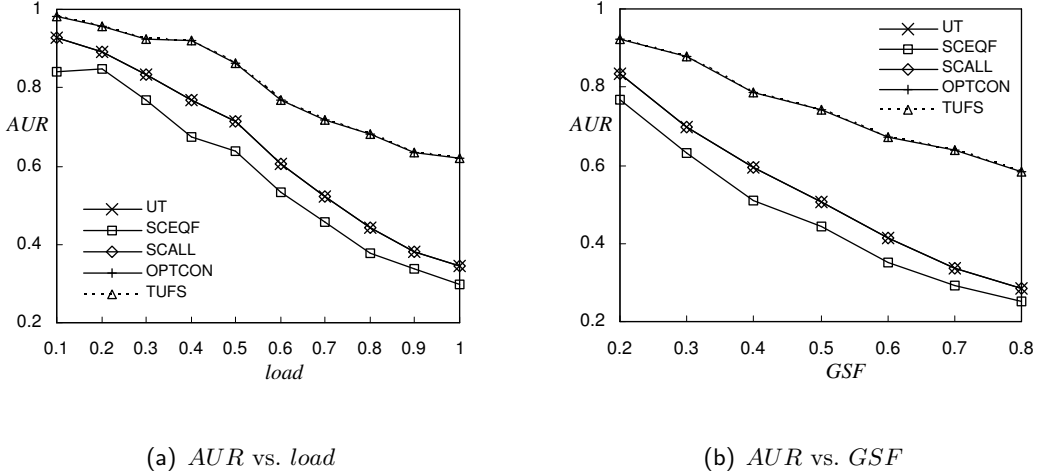
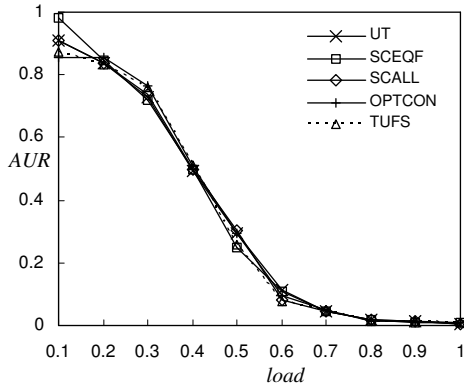
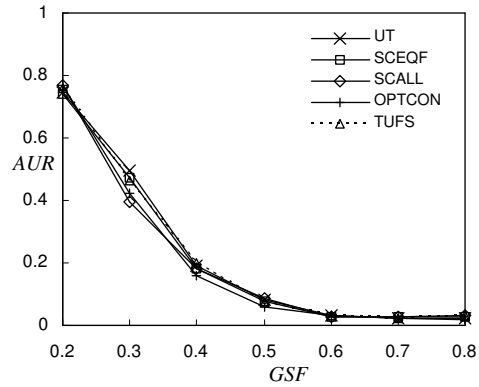


Fig. 9. Accrued Utility Ratio with DASA

As a comparison, the performance of the EDF algorithm under the different DTD strategies is shown in Figure 11. Under EDF, $SCEQF$ performs better than the others when $load$ or GSF is low (in the figure, approximately less than 0.5), because deadline slicing gives OLTs shorter deadline times, which in turn gives them higher priorities for accessing resources under EDF. But when $load$ or GSF approach 1.0, $SCEQF$ shows worse performance than the others. This is because if slack is too tight or the $load$ too high, with $SCEQF$, it is easy for an OLT in the DT to miss its sub deadline, which is shorter than the DT's deadline, resulting in failure of the DT.

EDF is not an UA scheduler, but its primary performance measure—i.e., deadline miss ratio, can be converted to AUR . Intuitively, the more deadlines are met by DTs, the more utilities are accrued. Thus, the deadline miss ratio performance metric of EDF can be a reasonable metric for UA scheduling. The results in Figure 11(a) are consistent with Kao and Garcia-Molina's experimental results on the serial subtask problem (SSP) presented in [21], where the deadline miss ratio of EQF is lower than the ultimate deadline (UD) strategy as $load$ varies from 0.1 to 0.5. Kao and Garcia-Molina also argue in [21] that “if

(a) *AUR vs. load*(b) *AUR vs. GSF*Fig. 10. Accrued Utility Ratio with D^{over}

the slack is too tight or if the load is too high, no matter what SSP policy we use, many deadlines will be missed.”

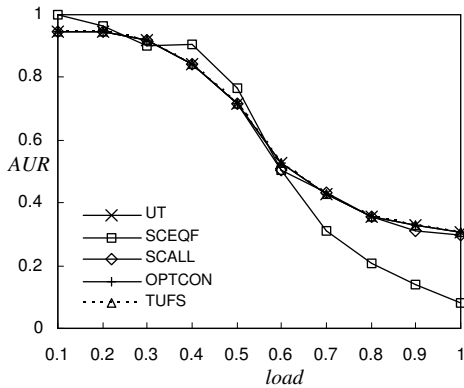
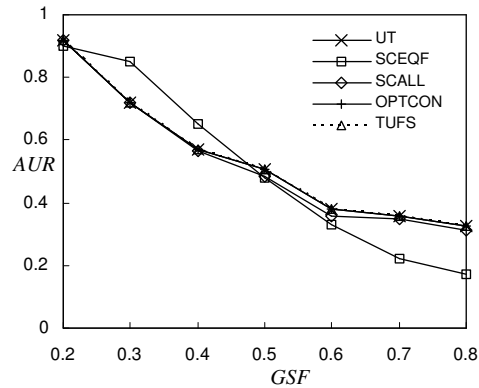
(a) *AUR vs. load*(b) *AUR vs. GSF*

Fig. 11. Accrued Utility Ratio with EDF

From Figure 7 to Figure 11, we can infer that different metrics employed in the scheduling algorithms result in their different scheduling behaviors; thus, the performance of a given DTD method will be differently influenced by different underlying schedulers. For this reason, large performance gaps, in terms of *AURs*, are particularly interesting to us.

To understand such differences, we consider three types of resource competition among

threads. These include local-local, local-global, and global-global. DTs are subject to local-global and global-global contention. A scheduling algorithm resolves the contention mainly by its scheduling metric. In Section IV, we categorize the DTD methods into three classes, and they alter only the OLTs’ metrics used by the schedulers, so they only impact local-global and global-global contention. Thus, the choice of a DTD strategy significantly affects DTs and their *AURs*.

According to our algorithm descriptions in Section III, GUS resolves resource contention mainly by comparing the PUDs of threads. Our simulation reveals that, under GUS, the performance of *OPTCON* and *TUFS*, which increase OLTs’ PUDs, are better than that of others, for various parameter settings. We notice that slicing the deadline of a DT and allocating sub deadlines to its OLTs can also increase the OLTs’ PUDs. But such deadline slicing technique i.e., *SCEQF*, performs poorly for DTs. The reason is that a DT consists of a series of OLTs, and if any OLT in the DT misses its sub deadline and is aborted by the scheduler, the parent DT fails. Thus, deadline slicing techniques sometimes can be detrimental to DTs, if too “tight” sub deadline constraints are assigned to their OLTs.

It is interesting to observe that D^{over} , as a timer based scheduling algorithm, is almost not affected by TUF decomposition. Both LBESA and DASA compare deadlines and PUDs for scheduling, but in different ways. From their performance under different DTD methods, we can infer that DASA is more “PUD-based” and LBESA is more “deadline-based.”

Thus, from the experimental results in this section, we can derive a more precise scheduler categorization than that in Section IV. We now have three classes of schedulers: deadline-based (LBESA and EDF), PUD-based (GUS and DASA), and timer-based (D^{over}). We summarize the performance of DTD strategies for each class of schedulers in Section V-C.

2) Effect of TUF Shape: Our designed DTD methods will decompose different shapes of TUFs differently, possibly yielding different performance of DTs. Thus, in this section,

we study the effect of DTD with different shapes of TUFs. Our study focuses on GUS and LBESA, as they are the only ones that allow arbitrarily-shaped TUFs.

We conduct experiments with non-increasing TUFs (TUF_1 , TUF_2 , and TUF_3), and a strictly concave TUF (TUF_4). Note that step TUFs are also non-increasing. To obtain an average performance of the DTD methods on various TUFs, we also consider TUF_5 , which is a combination of different TUFs. The experimental results show that, performance of DTs with non-increasing TUFs are quite similar, and the major performance difference can be observed with TUF_4 , so we only list plots with TUF_4 .

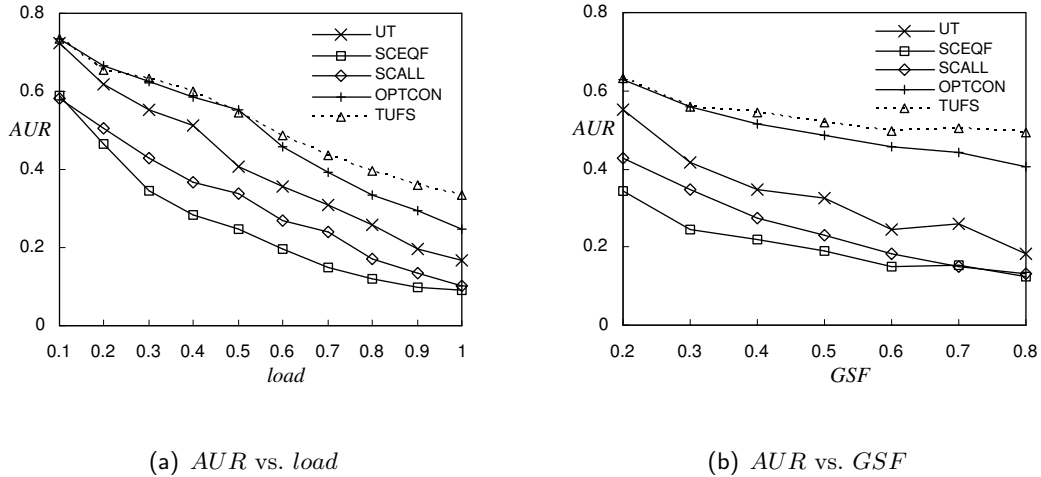


Fig. 12. Accrued Utility Ratio with GUS under TUF_4

Figure 12 shows the results of GUS under TUF_4 . In contrast to the results under non-increasing TUFs, different DTD strategies show different performance under strictly concave TUFs, respectively. But at any *load* or *GSF*, TUF_5 performs the best, while $SCEQF$ and $SCALL$ perform the worst.

We show the corresponding results of LBESA with TUF_4 in Figure 13. With strictly concave TUFs, we obtain different results from those with non-increasing TUFs. Figure 13 shows that $SCEQF$ only outperforms others when *load* and *GSF* approach 1.0; at light *load* and *GSF*, $SCALL$ has the best performance.

Among the four shapes of TUFs considered in our experiments, strictly concave TUFs have the most apparent impact on DTD, in the sense that the most apparent performance

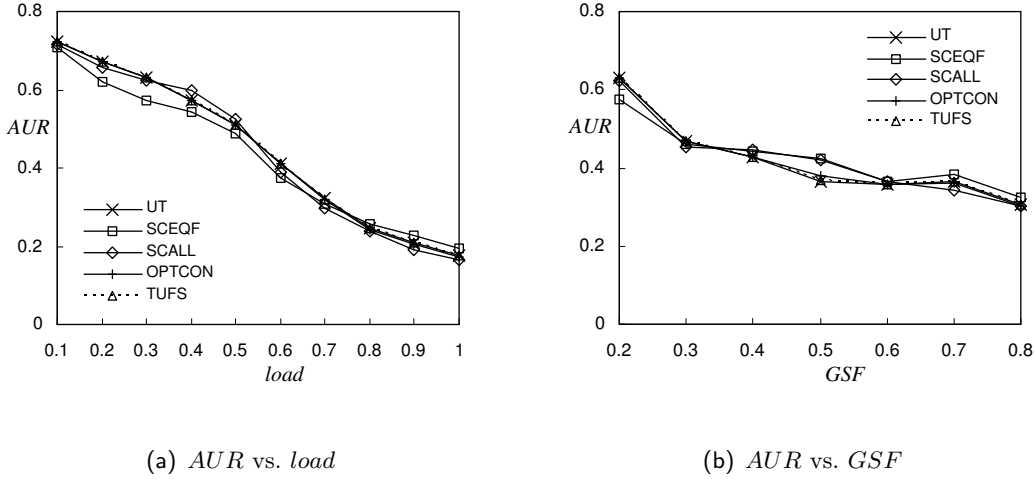


Fig. 13. Accrued Utility Ratio with LBESA under TUF_4

gap can be observed among different DTD strategies. However, the results observed in Figure 12, Figure 13 and other experiments are consistent with those in Section V-B.1—DTD strategies that change an OLT’s PUD ($OPTCON$ and $TUF5$) work better for GUS, and those that slice a DT’s deadline to allocate to its OLTs ($SCEQF$ and $SCALL$) work better for LBESA.

3) Effect of Local Threads: Comparing the AUR s of the different DTD strategies, we observe that as the $load$ and GSF increases, the performance of all strategies deteriorate. However, all of our previously reported experiments are carried out with $frac_local = 0.80$.

We had hypothesized that by changing the deadlines or PUDs of OLTs, various DTD methods can help DTs to “grab” resources (including CPU) from local threads, since deadlines or PUDs are key scheduling metrics used by UA scheduling algorithms. For example, the likelihood of DTs to obtain resources can be improved by reducing the deadlines of OLTs (under LBESA) or by increasing the PUDs of OLTs (under GUS and DASA).

To verify this, we vary the relative proportion of the two kinds of threads, by varying $frac_local$ from 0.3 to 0.88, and study AUR of different algorithms. We set $\mu_{OLT} = \mu_{local} = 1.0$ and $\lambda_{DT} = \lambda_{local} = 1/8.33$, and therefore $load$ is about 0.5.

Figures 14 and 15 show the performance of GUS and DASA with different DTD methods, under TUF_3 . We observe that GUS and DASA have similar performance. As $frac_local$ increases (i.e., more PUD is contributed by local threads), $AUR_{GUS/DASA}^{UT}(TUF_3)$ drops dramatically. When GUS and DASA are used for scheduling OLTs and local threads, and when $frac_local$ is less than or equal to 0.5, the two classes of threads (OLTs and local threads) have almost equal PUDs according to the definition of $frac_local$, and thus have almost equal chances to be selected to execute. In such cases, intuitively UT should perform similar to methods that increase PUDs of DTs, but better than deadline slicing methods. This is illustrated in the plots.

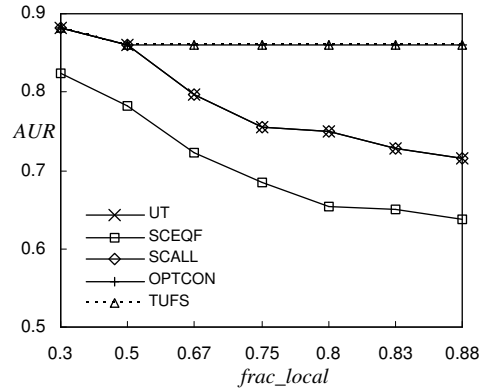
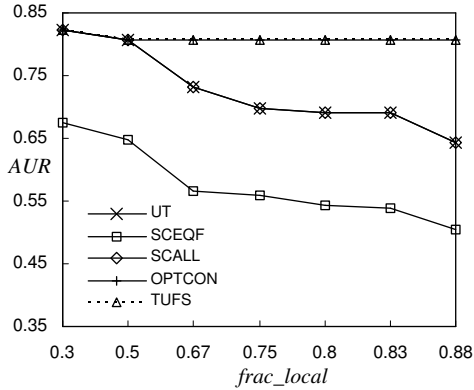


Fig. 14. AUR vs. $frac_local$ with GUS under TUF_3 Fig. 15. AUR vs. $frac_local$ with DASA under TUF_3

But when $frac_local$ exceeds 0.5, OLTs with sub- TUF s allocated by UT will be at a disadvantage under the PUD-based scheduling of GUS and DASA. Thus, we observe that higher the $frac_local$, the worse UT performs. The performance of $OPTCON$ and $TUF3$ are almost a constant as $frac_local$ increases, because these methods maintain PUDs of OLTs to be comparable with PUDs of local threads.

Figures 16 and 17 show the performance of LBESA and D^{over} with different DTD methods under TUF_3 . In Figure 16, we observe that $SCALL$, $OPTCON$, and $TUF3$ are almost constant as $frac_local$ increases, but only slightly outperform the others. This is reasonable, because LBESA is in the class of “deadline-based”, and its scheduling decisions are only partially dependent on PUDs. On the other hand, in Figure 17, all

curves converge in a narrow zone between 0.24 and 0.29 (corresponding to the point of $load = 0.5$ in Figure 10(a)). Since D^{over} does not consider PUDs, the results of D^{over} do not exhibit any regular pattern in Figure 17.

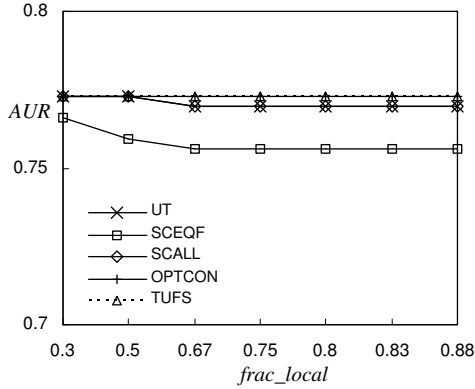


Fig. 16. AUR vs. $frac_local$ with LBESA under TUF_3

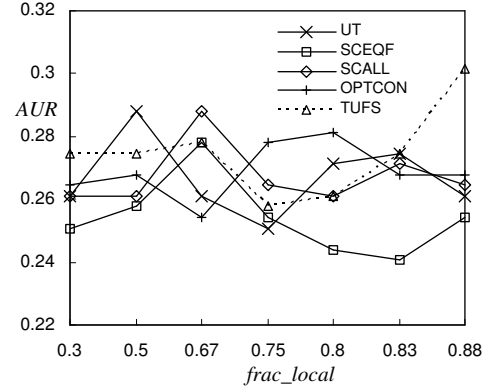


Fig. 17. AUR vs. $frac_local$ with D^{over} under TUF_3

To compare the performance of GUS and LBESA with changing $frac_local$ under other shapes of TUFs, we also conduct experiments with TUF_1 , TUF_2 , and TUF_4 . Results with TUF_1 and TUF_2 are similar to those in Figure 14 and Figure 16. But interesting results can be observed when GUS and LBESA are scheduling TUF_4 that is strictly concave, and the results are shown in Figure 18 and Figure 19.

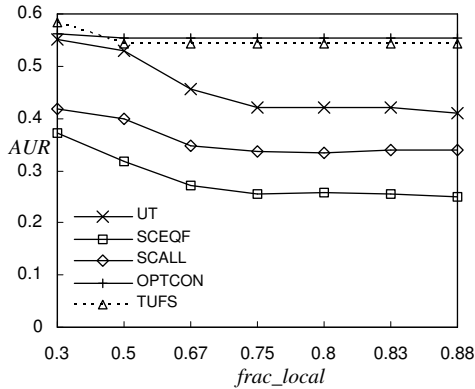


Fig. 18. AUR vs. $frac_local$ with GUS under TUF_4

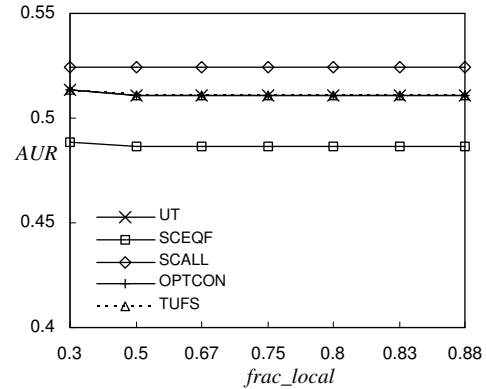


Fig. 19. AUR vs. $frac_local$ with LBESA under TUF_4

In Figure 18, *OPTCON* and *TUFS* are apparently better than the others, while *SCEQF* performs the worst. For strictly concave TUFs, *OPTCON* and *TUFS* allow enough slack for each OLT, and also improve the OLT’s PUD with respecting the TUF shape of the DT. Therefore, they show superior performance. In Figure 19, performance gaps are more pronounced than those in Figure 16. With LBESA, the mechanism *SCALL*, which slices the DT’s deadline and considers the *Opt* of the DT performs the best. *OPTCON* and *TUFS* show very similar performance as that of *UT*, which implies improving PUDs of OLTs has little effect on LBESA’s scheduling decisions. We have to note that, in Figure 19 the performance gaps among all DTD mechanisms are within 5%, which means *frac_local* has only limited effect on DTD strategies under LBESA.

The performance of various DTD strategies under EDF is not affected by *frac_local*. This is because varying *frac_local* only changes the PUD contributed by the local threads, but it does not affect the scheduling decisions made by EDF.

4) Effect of Resource Dependencies: There are no resource dependencies among OLTs in the experiments that we have conducted so far. In this section, we study how resource dependencies among OLTs affect the performance of DTD strategies.

We impose resource dependencies among OLTs of different DTs, and among OLTs and local threads. That is, we incorporate local-global and global-global dependencies at the same time. We then study the performance of the DTD strategies under GUS and DASA, as only these two UA algorithms consider TUFs under resource dependencies. For OLTs within a single DT, there are no resource dependencies, but only precedence dependencies. We first consider step TUF (TUF_3) for these experiments, and then other shapes.

Figures 20 and 21 show AUR_{GUS} and AUR_{DASA} , respectively, as *load* and *GSF* increase. We also vary *frac_local* to study the impact of resource dependencies on performance. Figures 22 and 23 show the *AURs* of GUS and DASA with dependencies, as *frac_local* varies, respectively.

From Figure 20 to Figure 23, GUS and DASA exhibit similar performance with resource

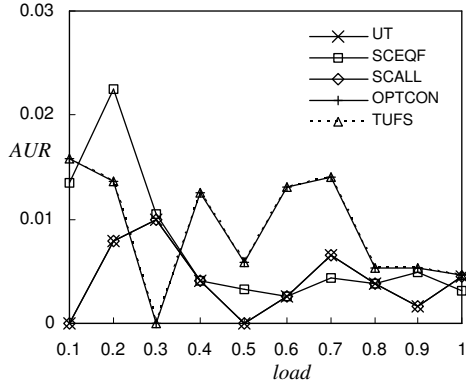
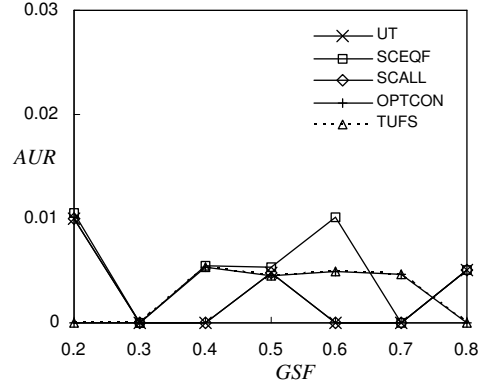
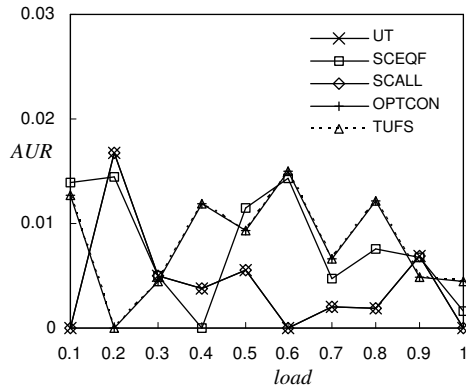
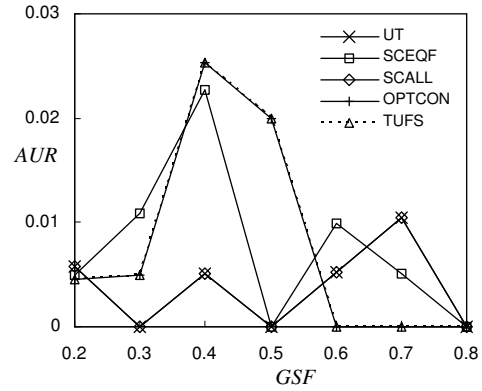
(a) *AUR vs. load*(b) *AUR vs. GSF*Fig. 20. Accrued Utility Ratio with GUS Under TUF_3 and Resource Dependencies(a) *AUR vs. load*(b) *AUR vs. GSF*

Fig. 21. Accrued Utility Ratio with DASA Under Resource Dependencies

dependencies. From the figures, we observe that the performance drops when there are resource dependencies; *AURs* of GUS and DASA under different DTD techniques decrease to less than 1%. Moreover, we do not observe any regular pattern in the results when *load*, *GSF*, and *frac_local* are varied.

This is reasonable, because DTD cannot compensate for major interference among OLTs. Resource access operations and consequent dependencies among OLTs and local threads can cause unexpected, sometimes large, interference to DTD methods, resulting in their poor performance. We observed similar results for GUS with other TUF shapes

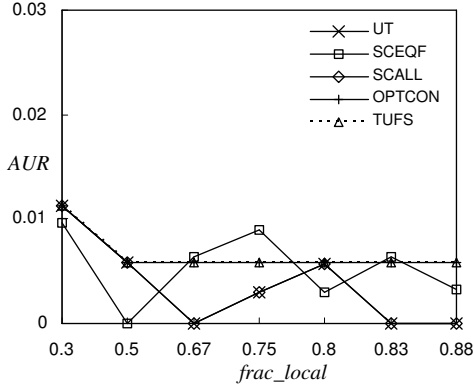


Fig. 22. AUR vs. $frac_local$ with GUS Under TUF_3 and Resource Dependencies

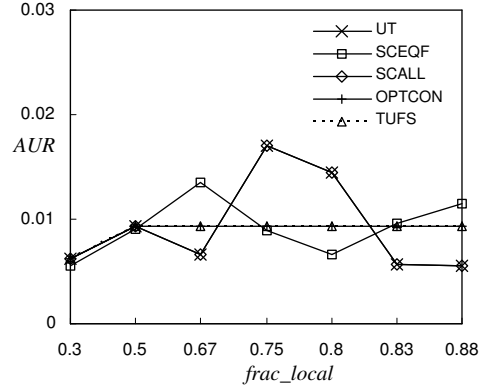


Fig. 23. AUR vs. $frac_local$ with DASA Under Resource Dependencies

and resource dependencies.

C. Summary of Experiments and Conclusions

Our experiments in Section V-B verified our proposed heuristics on DTD strategies for UA scheduling, by considering possible factors that can affect the DTD performance. We now summarize our results and draw conclusions.

We use UT as a baseline method, and measure the performance gains of other DTD methods over UT . For each class of UA algorithms, we summarize our conclusions about class-appropriate DTD techniques as follows:

- 1 With “PUD-based” UA algorithms (GUS and DASA), $OPTCON$ and $TUF3$ always perform better than UT , and $SCEQF$ and $SCALL$ always perform worse, or no better than UT , for different TUF shapes. For strictly concave TUFs, $TUF3$ performs the best.
- 2 With “deadline-based” UA algorithms (e.g., LBESA), $SCEQF$ and $SCALL$ always perform better than, or at least as well as, UT , and $OPTCON$ and $TUF3$ always perform no better than UT , for different TUF shapes.
- 3 With “timer-based” UA algorithms (e.g., D^{over}), different DTD strategies show little difference in improving DTs’ performance.

- 4 For downward step TUFs with GUS and DASA, the increase of *frac_local* causes quick drops on *AURs* of *UT* and *SCALL*. However, *OPTCON* and *TUFS* perform better, while *SCEQF* strategies perform worse than *UT*, because the former methods can improve the OLTs’ PUDs seen by the node-local UA scheduler when *frac_local* increases. With LBESA and increasing *frac_local*, no DTD strategies outperform *SCALL*.
- 5 With resource dependencies among OLTs of different DTs, all DTD strategies show very poor performance because of the unexpected and large interference caused by resource access operations (and consequent dependencies).

VI. PAST WORK

There are relatively few studies on the DTD problem. Most of the past efforts on time constraint decomposition in real-time distributed systems focus on the deadline constraint. We summarize these efforts and contrast them with our work.

A legacy issue that is analogous to the one that motivates our DTD problem is presented in Real-Time CORBA 1.0 (RTC1) [23], where any node may be shared by both RTC1 applications and non-CORBA applications. RTC1 priorities are mapped into each node’s local priority space, which is shared by RTC1 and non-CORBA applications. In all three cases of RTC1, there are end-to-end timeliness requirements that must be decomposed into timeliness requirements on each node involved in a multi-node computation. On those nodes, these decomposed timeliness requirements contend for resources with the timeliness requirements of strictly node-local computations. The challenge is for the scheduler to resolve this contention in a manner that is optimal according to application-specific criteria for both the distributed computations and the local computations. A multi-node computation with an end-to-end deadline has local sub-computations with per-node deadlines derived from the end-to-end deadline; these deadlines contend with the node-local computation deadlines.

In RTC1 systems, an RTC1 thread has a 15-bit CORBA (“global”) priority; when a

client invokes a servant, the client CORBA priority is mapped into the servant node's local operating system (much smaller) priority space using an application- or system-specific mapping; these priorities contend with the node-local computation priorities. (The mapping is reversed when an invocation returns; care must be taken in defining the mappings so as to retain priority fidelity end-to-end.)

The DTD problem is also analogous to the problem of deadline decomposition in distributed real-time systems whose time constraints are deadlines. In fact, deadline decomposition is a special case of DTD, where all TUFs are binary-valued, downward step-shaped functions.

Bettati and Liu [24], [25] present an approach for scheduling pre-allocated flow-shop (sequential) tasks in a hard real-time distributed environment. In their model, global tasks consist of (same) set of subtasks to be executed on nodes in the same order. The goal is to devise efficient off-line algorithms for computing a schedule for the subtasks such that all deadlines are met (if such a schedule exists). In their work, local deadlines are assigned by distributing end-to-end deadlines evenly over tasks, and then tasks are non-preemptively scheduled using a deadline-based priority scheme.

Kao and Garcia-Molina present multiple strategies for automatically translating the end-to-end deadline into deadlines for individual subtasks in distributed soft real-time systems [21], [26]. They reduce the subtask deadline assignment problem (SDA) into two subproblems: the serial subtask problem (SSP) and parallel subtasks problem (PSP). The authors present decomposition strategies called Ultimate Deadline (UD), Effective Deadline (ED), Equal Slack (EQS), and Equal Flexibility (EQF) for the SSP problem. Furthermore, they propose a strategy called DIV-x for the PSP problem. The techniques are aimed at systems with complete *a priori* knowledge of task-processor assignment.

Di Natale and Stankovic [27] present the end-to-end deadline slicing technique for assigning slices to tasks using the critical-path concept. The strategy used for finding slices is to determine a critical path in the task graph that minimizes the overall laxity of the path. The slicing technique is optimal in the sense that it maximizes the minimum

task laxity. The optimality applies to task assignments and communication costs that are *a priori* known.

In [28], García and Harbour present an approach that derives deadlines for preemptive, deadline-monotonic task scheduling. Given an initial local deadline assignment, the strategy seeks to find an improved deadline assignment using a heuristic iterative approach.

Saksena and Hong present a deadline-distribution approach for pre-allocated tasks in [29] and [30]. The approach specifies the end-to-end deadline as a set of local deadline-assignment constraints, and calculates the largest value of a scaling factor based on a set of local deadline assignments known *a priori*. The scaling factor is then applied to the task execution time. The local deadline assignment is chosen to maximize the largest value of the scaling factor.

In [31], Jonsson and Shin present a deadline-distribution scheme that distributes task deadlines using adaptive metrics. The authors experimentally show that their scheme yields significantly better performance in the presence of high resource contention. The deadline distribution problem that is addressed in [31] focuses on distributed hard real-time systems with relaxed locality constraints. Thus, schedulability analysis is performed at pre-run-time and only a subset of the tasks are constrained by pre-assignment to specific processors.

Thus, to the best of our knowledge, all past efforts on time constraint decomposition have focussed on deadlines. We are not aware of any work that considered TUF decomposition.

VII. CONCLUSIONS AND FUTURE WORK

In this paper, we consider integrating RTC2 applications, which are subject to TUF time constraints and UA optimality criteria into legacy environments which contain time-critical, non-RTC2 applications. Since system node resources (that use UA scheduling algorithms) are now shared between RTC2 and non-RTC2 applications, DTs of RTC2 applications may suffer performance degradation due to interference from non-RTC2

application threads.

To alleviate this, we consider decomposing TUFs of DTs into sub-TUFs for scheduling segments of DTs. We present decomposition techniques called *UT*, *SCEQF*, *SCALL*, *OPTCON*, and *TUFS*, which are specific to different classes of UA algorithms, such as those that use utility density, and those that use deadline as their key metric. Our experimental studies identify the conditions that affect the performance of the decomposition techniques. Further, our studies show that *OPTCON* and *TUFS* perform best for utility density-based UA algorithms, and *SCEQF* and *SCALL* perform best for deadline-based UA algorithms.

There are several interesting directions for future work. One direction is to consider TUF decomposition under emerging stochastic UA scheduling algorithms [11]. Another direction is to develop scheduling algorithms for scheduling DTs according to RTC2's Cases 3 and 4.

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.

REFERENCES

- [1] OMG, "Real-time CORBA 2.0: Dynamic Scheduling Specification," Object Management Group, Tech. Rep., September 2001, OMG Final Adopted Specification, <http://www.omg.org/docs/ptc/01-08-34.pdf> (last accessed: October 22, 2004).
- [2] E. D. Jensen, A. Wellings, R. Clark, and D. Wells, "The distributed real-time specification for java: A status report," in *Proceedings of The Embedded Systems Conference*, 2002.
- [3] J. D. Northcutt, *Mechanisms for Reliable Distributed Real-Time Operating Systems: The Alpha Kernel*. Academic Press, 1987.
- [4] E. D. Jensen and J. D. Northcutt, "Alpha: A non-proprietary operating system for large, complex, distributed real-time systems," in *IEEE Workshop on Experimental Distributed Systems*, 1990, pp. 35–41.
- [5] The Open Group, "Mk7.3a release notes," <http://www.real-time.org/docs/RelNotes7.Book.pdf>, Cambridge, MA 02142, 1998.
- [6] GlobalSecurity.org, "Multi-Platform Radar Technology Insertion Program," <http://www.globalsecurity.org/intell/systems/mp-rtip.htm/> (last accessed: October 22, 2004).
- [7] —, "BMC3I Battle Management, Command, Control, Communications and Intelligence," <http://www.globalsecurity.org/space/systems/bmc3i.htm/> (last accessed: October 22, 2004).
- [8] 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.

- [9] 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.
- [10] 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, Archons Project TR-88121, December 1988, <http://www.real-time.org> (last accessed: October 22, 2004).
- [11] P. Li, "Utility Accrual Real-Time Scheduling: Models and Algorithms," Ph.D. dissertation, Virginia Tech, 2004, <http://scholar.lib.vt.edu/theses/available/etd-08092004-230138/> (last accessed: October 22, 2004).
- [12] 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: October 22, 2004).
- [13] 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: October 22, 2004).
- [14] G. Koren and D. Shasha, "D-Over: An Optimal On-line Scheduling Algorithm for Overloaded Real-Time Systems," in *IEEE Real-Time Systems Symposium*, December 1992, pp. 290–299.
- [15] K. Chen and P. Muhlethaler, "A Scheduling Algorithm for Tasks Described by Time Value Function," *Journal of Real-Time Systems*, vol. 10, no. 3, pp. 293–312, May 1996.
- [16] D. Mosse, M. E. Pollack, and Y. Ronen, "Value-Density Algorithm to Handle Transient Overloads in Scheduling," in *IEEE Euromicro Conference on Real-Time Systems*, June 1999, pp. 278–286.
- [17] J. Wang and B. Ravindran, "Time-Utility Function-Driven Switched Ethernet: Packet Scheduling Algorithm, Implementation, and Feasibility Analysis," *IEEE Transactions on Parallel and Distributed Systems*, vol. 15, no. 2, pp. 119–133, February 2004.
- [18] 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.
- [19] W. Horn, "Some Simple Scheduling Algorithms," *Naval Research Logistics Quarterly*, vol. 21, pp. 177–185, 1974.
- [20] 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.
- [21] B. Kao and H. Garcia-Molina, "Deadline Assignment in a Distributed Soft Real-time System," *IEEE Transactions on Parallel and Distributed Systems*, vol. 8, no. 12, pp. 1268–1274, December 1997.
- [22] A. Varga, "OMNeT++ Discrete Event Simulation System," <http://www.omnetpp.org/> (last accessed: October 22, 2004).
- [23] OMG, "Real-Time CORBA Joint Revised Submission," Object Management Group, Tech. Rep. OMG TC Document ptc/99-05-03, May 1998, <http://www.omg.org/docs/ptc/99-05-03.pdf> (last accessed: October 22, 2004).
- [24] R. Bettati and J. W. S. Liu, "Algorithms for End-to-End Scheduling to Meet Deadlines," in *IEEE Symposium on Parallel and Distributed Processing*, 1990, pp. 62–67.
- [25] —, "End-to-End Scheduling to Meet Deadlines in Distributed Systems," in *IEEE Real-Time Systems Symposium*, 1992, pp. 452–459.
- [26] B. C. Kao, "Scheduling in Distributed Soft Real-Time Systems With Autonomous Components," Ph.D. dissertation, Princeton University, November 1995.
- [27] M. D. Natale and J. A. Stankovic, "Dynamic End-to-End Guarantees in Distributed Real-Time Systems," in *IEEE Real-Time Systems Symposium*, 1994.
- [28] J. J. G. Garcia and M. G. Harbour, "Optimized Priority Assignment for Tasks and Messages in Distributed Hard Real-Time Systems," in *IEEE Workshop on Parallel and Distributed Real-Time Systems*, 1995, pp. 124–132.
- [29] M. Saksena and S. Hong, "An Engineering Approach to Decomposing End-to-End Delays on a Distributed Real-Time System," in *IEEE Workshop on Parallel and Distributed Real-Time Systems*, 1996, pp. 244–251.
- [30] —, "Resource Conscious Design of Distributed Real-Time Systems: An End-to-End Approach," in *IEEE International Conference on Engineering of Complex Computer Systems*, 1996, pp. 306–313.

- [31] J. Jonsson and K. G. Shin, "Robust Adaptive Metrics for Deadline Assignment in Distributed Hard Real-Time Systems," *Real-Time Systems*, vol. 23, no. 3, pp. 239–271, November 2002.