2012 IEEE/ACM Third International Conference on Cyber-Physical Systems
DOI 10.1109/ICCPS.2012.18

On Resource Overbooking in an Unmanned Aerial Vehicle

Dionisio de Niz, Lutz Wrage
Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA, U.S.A.
dionisio@sei.cmu.edu, lwrage@sei.cmu.edu

Nathaniel Storer, Anthony Rowe, and Ragunathan (Raj) Rajkumar
Electrical & Computer Engineering
Carnegie Mellon University
Pittsburgh, PA, U.S.A.
nstorer@andrew.cmu.edu, agr@ece.cmu.edu, raj@ece.cmu.edu

***Abstract*—Large variations in the execution times of algorithms characterize many cyber-physical systems (CPS). For example, variations arise in the case of visual object-tracking tasks, whose execution times depend on the contents of the current field of view of the camera. In this paper, we study such a scenario in a small Unmanned Aerial Vehicle (UAV) system with a camera that must detect objects in a variety of conditions ranging from the simple to the complex. Given resource, weight and size constraints, such cyber-physical systems do not have the resources to satisfy the hard-real-time requirements of safe flight along with the need to process highly variable workloads at the highest quality and resolution levels. Hence, tradeoffs have to be made in real-time across multiple levels of criticality of running tasks and their operating points. Specifically, the utility derived from tracking an increasing number of objects may saturate when the mission software can no longer perform the required processing on each individual object. In this paper, we evaluate a new approach called ZS-QRAM (Zero-Slack QoS-based Resource Allocation Model) that maximizes the UAV system utility by explicitly taking into account the diminishing returns on tracking an increasing number of objects. We perform a detailed evaluation of our approach on our UAV system to clearly demonstrate its benefits.**

I. Introduction

One of the key characteristics of CPS is their close interaction with their environment. UAV surveillance systems are a compelling example of CPS because they must perform timely adjustments to propulsion speeds and control surfaces to maintain flight, process video streams, and navigate a route avoiding obstacles. Timely execution of tasks in these types of systems typically relies on real-time scheduling along with physically independent (and often over-provisioned) subsystems. Real-time scheduling policies guarantee task response time bounds based on their worst-case execution times (WCET) and frequency of execution. Unfortunately, the use of complex CPS application algorithms, whose execution time depends on environmental conditions, make the WCET difficult to estimate. This is the case of the execution time of vision-based object recognition algorithms used for surveillance in UAV systems. This difficulty has motivated alternative schemes that can improve the schedulable utilization of the processors without compromising critical guarantees in the system. Zero-Slack Rate-Monotonic scheduling (ZSRM) is one of these approaches [1]. ZSRM is a general fixed-priority preemptive scheduling policy that is defined for a uniprocessor system. In ZSRM, a *criticality* value is associated with each task to reflect the task's importance to the CPS mission. Thus, ZSRM is specifically designed for mixed-criticality systems where tasks have different criticality levels and in the case of overloads, more critical tasks must execute to completion even at the total expense of less-critical tasks.

ZSRM allows the system designer to "overbook" CPU cycles at design time by allocating these cycles to more than one task from different criticality levels. In particular, tasks are characterized with two execution times: The *Nominal Case Execution Time* (NCET) and the *Overloaded Case Execution Time* (OCET). The NCET is the worst-case execution time of the task under normal execution when not overloaded. The OCET is the worst-case execution time of the task when it overloads, say, when the task has to process an unusually large number of objects in the camera's field-of-view. With this characterization, if a more-critical task needs to overrun its NCET, the task uses its overbooked cycles. Otherwise, a lower-criticality task will avail of those cycles.

In avionics systems, flight-critical control tasks are considered to be safety-critical as they are directly responsible for the safety of the vehicle. Other tasks are mission-oriented and are generally considered to be less critical than the safety-critical tasks.

While the criticality-overbooking in ZSRM provides the necessary protection of safety-critical tasks in avionics system, it proves to be insufficient when additional trade-offs are necessary within mission-critical tasks. A less-critical task can contribute more system utility than a more-critical task depending upon the current operating modes of those tasks. Different tasks may also overload at different times. Our objective is to maximize the total system utility under all operating conditions.

In this paper, we study the resource overbooking problem of a UAV surveillance system. We use this system to discuss the limitations of the ZSRM approach and evaluate the effectiveness of a new policy called ZS-QRAM [2] (Zero-Slack QoS-based Resource Allocation Model) that performs utility-based resource overbooking. ZS-QRAM combines

mechanisms from ZSRM and the Quality-of-Service-based Resource Allocation Model (Q-RAM) [3]. ZS-QRAM uses Q-RAM utility functions that map the resources allocated to a task to the utility that is obtained from the allocation. These utility functions generally exhibit what is called *diminishing returns*, i.e. every additional increment of a resource to a task returns less utility per unit of resource than the previous increment. The ratio of the utility increment to the resource increment is referred to as *marginal utility*.

In our UAV system, utility-based overbooking allows us to encode the utility saturation levels of a vision-based object-detection algorithm. Once marginal utility that is gained diminishes, allowing this object-detection algorithm to consume more CPU cycles may yield smaller utility than the utility lost by degrading (say) a video-streaming task to its next lower lower frame-per-second setting. Hence, instead of further degrading the video-streaming task, the object-detection task can be prevented from consuming more cycles.

In essence, our ZS-QRAM approach considers overloaded execution conditions and marginal utility at different task operating points in order to allocate resources (and thereby assign operating points) to tasks in the system. The objective of ZS-QRAM is to maximize the total utility accrued from these allocations under both normal and overloaded conditions. Under non-overloaded conditions, tasks will have been assigned the highest QoS operating points that are feasible with the available resources. Under overloaded conditions, our approach will select at run-time the tasks to degrade such that the total system utility is minimally reduced. Many different combinations of task overloads can occur, and ZS-QRAM needs to optimize system utility for all these possibilities.

A. Related Work

In this section, we describe previous work in the area of mixed-criticality overload scheduling that can potentially be used in our case study.

Multiple papers have been published related to overload scheduling. For example, [4] and [5] use a form of criticality together with a value assigned to job completions. Their approach is then to maximize the accrued value. In our case, we combine the accrued value of job completion with criticality which is not included in their approaches. In [6], the authors describe an approach to map the semantic importance of tasks to QoS service classes to improve resource utilization. They ensure that the resources allocated to a high-criticality task are never less than the allocation to a lower-criticality one. In our case, the use of ZSRM in our approach supports criticality-based graceful degradation while also maximizing total utility.

The elastic task model proposed in [7] provides a scheme for overload management. In this scheme, tasks with higher elasticity are allowed to run at higher rates when required, whereas tasks with lesser elasticity are restricted to a more steady rate. In our scheme, we also change rates (periods) but they are pre-defined and only change at a "zero-slack" instant [1]. We call this *period degradation* and the degradation is carried out in decreasing order of marginal utility, leading to a minimal loss of system utility.

Figure 1. Our UAV Quadrotor Platform

In [8, 9], the authors propose the Own Criticality-Based Priority (OCBP) schedulability test to find the appropriate priority ordering for mixed-criticality schedulability. Such a scheme is aimed at certification requirements. In contrast, we are focused on an overbooking approach to improve the total utility of the system.

The rest of the paper is organized as follows. Section II presents our UAV CPS. Section III discusses the utility-based maximization approach. In Section IV, we present the implementation of our approach and evaluate its efficacy particularly in comparison with standard techniques. Finally, in Section V, we present our conclusions.

II. The UAV Cyber-Physical System

A. System Overview

UAVs exemplify cyber-physical systems that consist of both hard and soft real-time requirements. Low-level control components required for stable flight have an extremely low tolerance to timing jitter. Higher-level software required for mission planning, perceptual sensor processing and coordination tend to have utility-based requirements where performance degrades given fewer resources. The miniaturization requirement for small UAVs tends to force both of these functional sets onto a single underlying compute platform where resources must be intelligently shared. In this section, we discuss the details of our UAV platform and highlight a few of the practical challenges related to resource scheduling.

Our small UAV platform shown in Figure 1 is based on the Parrot AR Drone quadrotor [10]. The platform consists of a 468 MHz ARM9 with 128 MB of flash, 128 MB of RAM and built-in 802.11b wireless communication running Linux

kernel v2.6.27. This platform is unique in that the low-level control functionality required for maintaining stable flight executes in Linux user space, making it an ideal testbed for scheduling. As part of the Drone-RK project [11], we developed a hardware expansion board and a software development environment that allows users to create autonomous UAV applications.

Figure 2 shows a block diagram of the UAV's hardware components. The system is composed of a main single-computer board running Linux and connected over two serial ports to micro-controller-powered sensor boards. The stock sensor board on the AR Drone is responsible for periodically measuring angle, rate of rotation and altitude (using ultrasound transducers). The Drone-RK expansion board contains a GPS receiver, a barometric pressure sensor, a compass and IR rangefinders. A set of C APIs is provided so as to allow onboard access to sensor data and the ability to perform basic navigation operations. The software framework consists of a low-level flight controller task, a video capturing task, an actuation task, and two sensing tasks (one for the onboard navigation sensors and one for our auxiliary sensors). The periods and execution times of these task sets are outlined in Table I. The other columns in the table capture related task utility properties and will be discussed later. Figure 3 illustrates the data flows between the tasks in the system.

B. Mixed-Criticality Concerns in UAVs

When designing software to run on our UAV platform, it became apparent that we require resource isolation to protect various system components. An early adopter of our system had the need to increase the amount of CPU allocated to a vision processing task. As one might expect, they raised the task priority and quickly saw that under high load the UAV platform became unstable and could no longer fly correctly. In other cases, if one adheres strictly to classic real-time design paradigms like rate-monotonic scheduling (RMS), the developer ends up in a situation where tasks

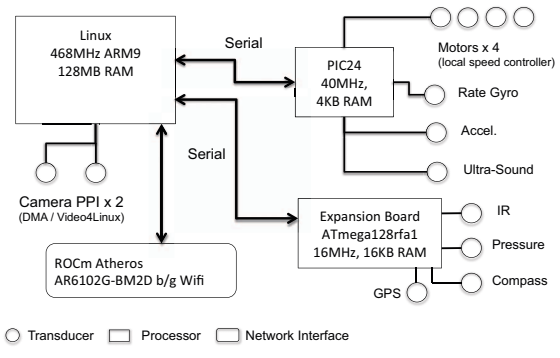


Figure 2. UAV Hardware Architecture

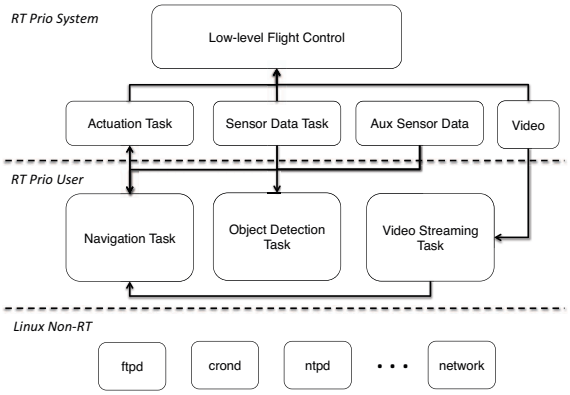


Figure 3. UAV Task Interactions

receive more CPU cycles solely based on their periods and not the end utility provided to the system. For example, it is quite common for a UAV to stream image data as part of a surveillance application. Image processing typically operates at a fixed rate derived from its attached camera, but in some cases it may not require a hard deadline for completion. For example, image processing tasks to detect objects or avoid obstacles might operate concurrently at a much lower rate. Often these tasks are more critical to the UAVs operation, but are forced to run at a lower priority since they have longer periods. As a result, if an overload occurs, the lowest-priority task is the first one to miss its deadline, even if it is more critical to the system. This is known as *criticality inversion* [1] and ZSRM can be used to correct this problem.

C. Utility Inversion

Unfortunately, the notion of criticality inversion does not capture all the issues that arise in a CPS. In particular, it does not capture the nature of diminishing returns exhibited by mission tasks when the latter are allocated additional resources. This scenario is better described by using the general notion of utility inversion. *Utility inversion* is said to occur when a lower-utility task prevents the execution of a higher-utility task. The utility inversion of a task is also represented by the duration that it is blocked by a lower-utility task. Such utility inversion can occur because the lower-utility task has higher scheduling priority. This occurs

Process	Util_1	Util_2	C	C^o	T_1	T_2	ZS
Actuation			1	1		30	30
Sensor Data			0.1	0.1		65	65
Aux Sensor Data			0.5	0.5		50	50
Navigation			11	11		50	49
Object Detection		7	15	40		100	87
Video Streaming	2	4	10	10	120	40	40

Table I
SURVEILLANCE EXPERIMENT TASK SET (TIME IN MS)

Task	NCET	OCET	Period_1	Period_2	Util_1	Util_2
Task 1	25	25	200	200	10	10
Task 2	200	300	800	400	5	6
Task 3	250	500	1600	800	7	8

Table II
BASIC TASKSET

in RMS-based systems when a task with a longer period yields higher utility than a task with a shorter period. It is worth noting that the allocation precedence can reverse when a task consumes more resources and its marginal utility diminishes. That is, the next resource increment to the higher-utility task may return a lower marginal utility than the one obtained if this resource increment is allocated to a lower-utility task. Such dynamic changes cannot occur in ZSRM where criticality inversion stays constant over time. As a result, using ZSRM mechanisms to prevent criticality inversion can still lead to utility inversion when the allocation precedence changes due to diminishing returns.

In the next sections, we show how our proposed ZS-QRAM approach can be used to help address these situations and maximize total system utility.

III. USING UTILITY TO MAXIMIZE MISSION VALUE

Our utility-maximizing ZS-QRAM approach is designed for tasks whose different QoS levels are implemented using different task periods and NCET and OCET as defined in ZSRM. Task periods are mapped to allocation points along two utility functions, one based on NCET and another on OCET. ZS-QRAM first considers NCET utility functions, and utilizes Q-RAM to do an initial allocation (at design time) where each increment in the allocation is represented by an increasingly shorter period[1]. If a task overloads at runtime, an overload management mechanism is used to degrade tasks (by selecting a longer period) to keep the taskset schedulable. This overload management mechanism uses task utility functions based on their OCET to select the tasks that render the least utility per unit of CPU utilization (marginal utility).

As an example, consider the taskset in Table II. For instance, Task 2 can execute with two different periods 800 and 400, with corresponding utilities of 5 and 6.

Figure 4(a) depicts the *nominal utility functions* that Q-RAM builds for the taskset of Table II based on their CPU utilization when they run at their NCET, i.e., $\frac{NCET}{Period}$. These functions are then used to perform the initial resource allocation. Q-RAM follows these functions one task and step at a time until the resources are exhausted or all the steps are successful. These allocation decisions happen at design time and so no penalty is incurred at runtime.

[1]The reader may correctly observe that algorithmic variants can also be used to yield different NCET and OCET values at different task operating points. Such variations are beyond the scope of this paper.

Similarly, Figure 4(b) depicts the *overload utility functions* that the ZS-QRAM overload management mechanism uses for our UAV taskset shown in Table II based on their OCET. These functions would be used to select the tasks to degrade under overload conditions so as to minimally reduce the total utility of the system. As a result, in an overload, ZS-QRAM will execute a sequence of cumulative degradation steps until the taskset becomes schedulable again. These steps can be calculated at design time, leaving only a degradation *sequence* to be explored at runtime.

Figures 5(a) and 5(b) depict the utility functions and diminishing returns of the *Video Streaming* and the *Object Detection* tasks for our UAV surveillance system using the information from Table I. It is worth noting that, in the overload utility functions of Figure 5(b), the marginal utility of the video streaming task at $\frac{10}{120}$ is larger than that of the object detection function at $\frac{40}{100}$. As a result, under one possible overload scenario, while the degradation of the video streaming period from 40 to 120 is the first step, the second step (if one were required) would be the degradation of the object detection.

A. ZS-QRAM Task Model

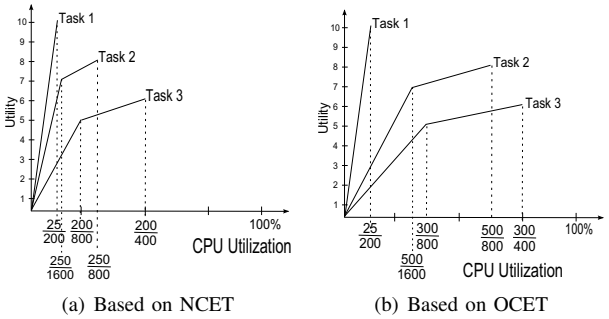
In this section, we briefly outline the task model that our system uses. ZS-QRAM models multi-period tasks as modal tasks. Modal tasks are defined as the ordered sequence: $\tau_i = \langle \tau_{i,x} \rangle$, where each $\tau_{i,x}$ is a mode of τ_i and the sequence is ordered by decreasing marginal utility to be defined shortly. *Each* mode[2] is defined as

$$\tau_{i,x} = (C_i, C_i^o, T_{i,x}, U_{i,x}, U_{i,x}^o)$$

where

- C_i is the NCET of the task,
- C_i^o is the OCET of the task,
- $T_{i,x}$ is the task period in this mode,
- $U_{i,x}$ is the utility of the mode when it runs for $C_{i,x}$, and

[2]A task mode can also be considered to be an "operating point" of a task using Q-RAM terminology.



(a) Based on NCET (b) Based on OCET

Figure 4. Utility Functions

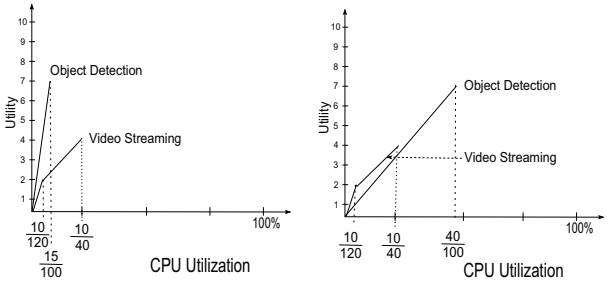
- $U^o_{i,x}$ is the utility of the mode when it runs for $C^o_{i,x}$.

With this definition, we can calculate the marginal utilties of the modes. These marginal utilities allow us to build the allocation sequence using the nominal utility functions and the degradation sequence using the overload utility functions. The modal marginal utilities are calculated as follows.

- $M_{i,x}$ is the marginal utility of the mode when it runs for its NCET, C_i. In this case, the task is said to be in its *nominal execution mode* and its marginal utility is known as the *nominal marginal utility* in that mode. While other definitions are possible, in this paper, we define the nominal marginal utility as:

$$M_{i,x} = \begin{cases} \frac{U_{i,1}}{\frac{C_i}{T_{i,1}}} & \text{if } x = 1 \\ \\ \frac{U_{i,x} - U_{i,x-1}}{\frac{C_i}{T_{i,x}} - \frac{C_i}{T_{i,x-1}}} & \text{otherwise} \end{cases}$$

  The above definition is used to capture the following semantics. Marginal utility represents the utility gained per unit of allocated resource. Resource allocation is quantified as utilization in the traditional scheduling sense of $\frac{C}{T}$ (i.e. ratio of worst-case execution time to period). We also assume that every task derives a utility of 0 when it is not allocated any resource. Hence, for any non-zero resource allocation (i.e. utilization), the first mode of a task τ_i has the marginal utility given by the ratio of $U_{i,1}$ to its resource utilization. Here, $U_{i,x}$ represents the absolute value of utility derived in mode x. For all other modes of task τ_i, the marginal utility at mode $x > 1$, is given by the ratio of the increase in utility from the previous mode $(x-1)$ to the increase in resource utilization relative to the previous mode.
- $M^o_{i,x}$ is the *overload marginal utility* of the mode when it runs for C^o_i, and the task is said to be running in its *overloaded execution mode*. Analogous to the case of nominal marginal utility, the overload marginal utility is defined as:



(a) Based on NCET (b) Based on OCET

Figure 5. UAV Utility Functions

Task	Mode	C	C^o	T	U/U^o	M	M^o
τ_1	$\tau_{1,1}$	25	25	200	10	80	80
τ_2	$\tau_{2,1}$	200	300	800	5	20	13.3
	$\tau_{2,2}$	200	300	400	6	4	2.6
τ_3	$\tau_{3,1}$	250	500	1600	7	44.8	22.4
	$\tau_{3,2}$	250	500	800	8	6.4	3.2

Table III

$$M^o_{i,x} = \begin{cases} \frac{U^o_{i,1}}{\frac{C^o_i}{T_{i,1}}} & \text{if } x = 1 \\ \\ \frac{U^o_{i,x} - U^o_{i,x-1}}{\frac{C^o_i}{T_{i,x}} - \frac{C^o_i}{T_{i,x-1}}} & \text{otherwise} \end{cases}$$

  This definition mirrors that of the nominal marginal utility, except that overloaded execution parameters are used instead of nominal execution parameters.

The modes in a task τ_i are ordered by decreasing length of period ($T_{i,x}$) such that $T_{i,x} > T_{i,x+1}$. Marginal utilities are restricted to follow the same order, i.e., $M_{i,x} > M_{i,x+1} \wedge M^o_{i,x} > M^o_{i,x+1}$. These conditions construct concave utility functions (with monotonically-decreasing marginal utility), which correspond to the simplest utility functions handled by Q-RAM.

Table III presents the taskset from Table II as modal tasks along with their marginal utilities.

B. ZS-QRAM Operation

As mentioned earlier, Q-RAM is used first to assign a mode $\tau_{i,x}$ for every task, with system schedulability verified at each allocation step. Then, given the operating modes for all tasks, a ZSRM-like zero-slack instant, zs_i, of each task is computed at design time [1]. In our current situation, the zero-slack instant is the last instant at which the system can be degraded to ensure that $\tau_{i,x}$ does not miss its deadline.

To simplify the discussion of our algorithms, we define two functions: $\Gamma_i(\Pi{:}\text{taskset})$, and $\Gamma^o_i(\Pi{:}\text{taskset})$. These functions return the interfering sets (in the nominal and overloaded execution modes respectively) of tasks for the selected modes assigned to the tasks in Π. These tasksets are defined in a similar fashion to ZSRM replacing criticality by utility at the task mode level[3].

ZS-QRAM preserves its schedulability guarantee across overloads with two overload management mechanisms: period degradation and task suspension. Both these mechanisms are designed to be triggered based on the zero-slack instant of the admitted modes.

Task suspension happens when the zero-slack instant of a task mode $\tau_{i,x}$ elapses and it has executed for a time less than or equal to C_i. In this case, all task modes $\tau_{j,y}$ with

[3]For a formal treatment and a discussion of the calculation of the zero-slack instant, please see [2].

a marginal $M^o_{j,y} < M^o_{i,x}$ are temporarily suspended. If the current job of $\tau_{i,x}$ completes without exceeding C_i, then all modes $\tau_{j,y}$ resume their execution but they are stopped if they try to execute for longer than $T_{i,x}$.

Period degradation is triggered if the zero-slack instant of a task mode $\tau_{i,x}$ elapses *and* its execution time exceeds C_i, i.e., task $\tau_{i,x}$ goes into overload. In this case, all task modes $\tau_{j,y}|M^o_{i,x} > M^o_{j,y}$ are degraded to a mode $\tau_{j,z}|M^o_{i,x} \leq M^o_{j,z}$.

The period degradation triggered by $\tau_{i,x}$ can also degrade $\tau_{i,x}$ itself. In this case, the degradation happens at its zero slack instant.

Note that, even though we have two overload management mechanisms, a task mode $\tau_{i,x}$ only has the two execution modes described in Section III-A. This is because, suspending or degrading the lower marginal utility modes reduces preemption time. In other words, suspension works as a temporary period degradation that reduces the preemption on mode $\tau_{i,x}$. This temporary degradation may become permanent if $\tau_{i,x}$ overruns C_i. For formal proofs of these algorithms, please see [2].

IV. System Implementation

We created three implementations of the ZS-QRAM overload manager. The first one was implemented as a resource reservation in Linux/RK [12], a resource kernel. This implementation has precise reservation accounting but it requires modification to the Linux kernel. This presented an inconvenience for our UAV platform due to difficulty obtaining the kernel sources (customized for the AR Drone). As a result, we created a second version that was implemented as a kernel module that does not require kernel modifications. The third version is a daemon-based scheduler that works on any version of Linux.

A. Linux/RK Reservation

Our Linux/RK implementation of ZS-QRAM performs enforcement in two stages. First, if the zero-slack instant is shorter than the response time of the nominal-case execution time, a marginal-utility-as-criticality enforcement is used. This means that the marginal utility of a task is used as its criticality and all the tasks with lower criticality are suspended. Secondly, we use the execution time enforcement of Linux/RK to trigger a system-wide period degradation when the task has executed beyond its NCET. In particular, we degrade all the modes $\tau_{j,y}$ in the system that have an $M^o_{j,y}$ smaller than the $M^o_{i,x}$ of the enforcing task mode $\tau_{i,x}$.

If the zero-slack instant is greater than the response time of the NCET, only period degradation takes place and no marginal-utility-as-criticality enforcement is necessary.

A system-wide marginal utility is maintained at all times representing the minimum marginal utility mode that is allowed to be active. This is used to keep track of period degradations that happen one on top of the other. For instance, consider a period degradation of all modes with marginal utility lower than 5 just after another degradation operation that degrades all modes with marginal utility lower than 3. When the degradation of marginal utility 5 ends, it returns to the system-wide marginal utility 3 keeping all modes with lower marginal utility degraded.

B. Non-Intrusive Kernel Module

Our non-intrusive kernel module was inspired by [13]. It uses high-precision kernel timers and a kernel thread to reschedule the processes attached to reserves. Because we do not intercept the context switching of processes inside the kernel, we do not have precise accounting of the CPU time consumed by each process. Instead, we only use the zero-slack timer for period degradation.

A zero-slack timer implements the zero slack enforcement. However, in order to implement a more precise utility-as-criticality enforcement with imprecise accounting, we take a different approach. Specifically, at the zero-slack instant, we perform period degradation immediately as if the task would have already executed beyond its C time. Then, when the degraded tasks have the opportunity to run and signal their completion (by calling the special system call `wait_for_next_period()`), we restore the original period (as if no period degradation had occurred) if we can verify that they completed before the original period and the system-wide marginal utility is low enough to re-enable its non-degraded mode.

C. User-Level Scheduler

A final implementation option is to create an entirely user-space scheduler to manage our taskset. This approach is similar to that of the kernel-module except that instead of relying on a system timer callback, we use a high-priority daemon process to perform scheduling and send signals to each waiting process. If the daemon process executes at the highest priority of all of the managed tasks, then it can suspend itself for specific periods of time and then signal the wakeup of any pending tasks. It can provide a primitive form of enforcement by decreasing the priority of any executing tasks at the zero-slack instant to the lowest level in the system. Due to the use of signals and the user-level daemon thread, it incurs a higher overhead.

	Scheduler	Kernel Module	User Space
Build Requirement	kernel build	kernel src	none
Overhead	minimal	+1 ctx swap	+2 ctx swap
Timing	HR timer	kernel tick	kernel tick
Enforcement	full	zs instant	zs instant
Accounting	full	none	none
Protection	full	rt taskset	rt taskset
Sustainability	difficult	easy	easiest

Table IV
Linux real-time scheduling options

D. A Comparison of Different Implementations

Table IV shows a comparison of the different scheduling options that could be adopted for legacy Linux systems. Modifying the scheduler requires the least amount of overhead and provides the best performance and the highest level of runtime assurance (protection and accountability). However, it relies on being able to work with the latest Linux kernel versions. In our case, this was not possible since the correct kernel sources were not available. The kernel module approach is the next best option and does not require building a new kernel. It does however require access to kernel source, which is required under the Linux GPL license. The kernel module incurs approximately one extra context swap of performance overhead as compared to using a modified scheduler. In terms of protection, it cannot always strictly guarantee that higher-priority tasks within the system do not starve the real-time taskset. In practice, it is a reasonable assumption that users will not launch miscellaneous tasks that gain real-time scheduling priorities and starve the execution of the ZS-QRAM tasks. The final option of using a user-space scheduler is similar to that of the kernel module with the additional overhead of having to context swap in an entire process to perform scheduling computations. With a $1ms$ OS timer, having two context swaps per scheduling operation could represent significant overhead.

Based on our constraints, we opted to use the kernel module-based scheduling approach for our study.

E. Utility Maximization in the UAV Mission-Critical Layer

In order to maximize system utility, ZS-QRAM ensures that the last task to miss its deadlines is the one that provides the most utility. In other words, in an overload, the overload management of ZS-QRAM degrades the periods of the tasks starting with the one with the smallest marginal utility, then the second smallest, continuing until the degraded taskset (and workload) becomes schedulable again. As a result, at any given time, the active tasks are the ones that provide the largest utility per unit of resource.

In order to measure the benefit of ZS-QRAM, we developed a metric called *Utility Degradation Resilience* (UDR). UDR measures the capacity of the resource allocator and overload management mechanisms to preserve the total utility of the system as tasks run beyond their NCET and trigger a load-shedding mechanism that degrades the utility of the system.

UDR is measured in a similar fashion to *ductility* [14] in ZSRM. It is defined as a matrix that evaluates all possible overloading conditions and the resulting consequences over the deadlines of the different tasks. However, instead of accruing only unit values when a task meets a deadline, we accrue the utility of the task. This is formally defined as:

$$\begin{matrix} \langle O_n, O_{n-1}, \ldots, O_1 \rangle \\ \cdots \\ \langle O_n, O_{n-1}, \ldots, O_1 \rangle \end{matrix} \begin{pmatrix} D_n U_n, D_{n-1} U_{n-1}, \ldots, D_1 U_1 \\ \cdots \\ D_n U_n, D_{n-1} U_{n-1}, \ldots, D_1 U_1 \end{pmatrix} \quad (1)$$

where:

- O_i is a zero-or-one variable that indicates whether the task with the i highest utility overruns, where larger i means higher utility,
- D_i is a zero-or-one variable indicating whether the task mode with the i highest utility meets its deadlines, and
- U_i is the utility of the i highest utility task mode.

Applying this metric to the *Video Streaming* (τ_1) and *Object Detection* (τ_2) tasks from our UAV taskset (Table I), when scheduled with ZS-QRAM, we obtain an UDR matrix as follows.

$$\begin{matrix} \langle 1,1 \rangle \\ \langle 1,0 \rangle \\ \langle 0,1 \rangle \\ \langle 0,0 \rangle \end{matrix} \begin{pmatrix} 1*7 & 0*4 & 1*2 \\ 1*7 & 0*4 & 1*2 \\ 1*7 & 1*4 & 0*2 \\ 1*7 & 1*4 & 0*2 \end{pmatrix} \quad (2)$$

The overloading vectors correspond to tasks $\langle \tau_2, \tau_1 \rangle$ and we enumerate all possible overloadings of the modes $(\tau_{2,1}, \tau_{1,2}, \tau_{1,1})$ of the possible periods available to the tasks. That is, the first row is when both τ_2 and τ_1 overload, the second one when only τ_2 overloads, the third when only τ_1 overloads, and the fourth when no task overloads. The consequences of the overloads on the system when using ZS-QRAM are as follows. For the first overloading row, $\tau_{2,1}$ meets its deadline, but task τ_1 is degraded from mode $\tau_{1,2}$ to mode $\tau_{1,1}$, which meets its deadline. The second overloading row also degrades task τ_1 in the same fashion as in the previous row. In the third overloading row, ZS-QRAM allows $\tau_{1,2}$ to overrun meeting its deadline because τ_2 did not overrun. Finally, in the fourth overloading row, no task overloads and, hence, both $\tau_{2,1}$ and $\tau_{1,2}$ meet their deadlines. It is worth noting that only one mode per task can be counted as meeting its deadlines.

We project the UDR matrix into an UDR scalar by simply adding the resulting utility of the tasks. That is, we add up all the cells in the matrix. In the case of Matrix (2), the total utility resilience obtained is 40.

In order to appreciate the value obtained from UDR Matrix (2), let us now present the result of a *bad* overload management policy. This represents the behavior of the rate-monotonic scheduling (RMS) policy under overload conditions. This policy may favor low-utility tasks under different overload conditions instead of high-utility tasks, leading to the following UDR matrix:

$$\begin{matrix} \langle 1,1 \rangle \\ \langle 1,0 \rangle \\ \langle 0,1 \rangle \\ \langle 0,0 \rangle \end{matrix} \begin{pmatrix} 0*7 & 1*4 & 0*2 \\ 0*7 & 1*4 & 0*2 \\ 1*7 & 1*4 & 0*2 \\ 1*7 & 1*4 & 0*2 \end{pmatrix} \quad (3)$$

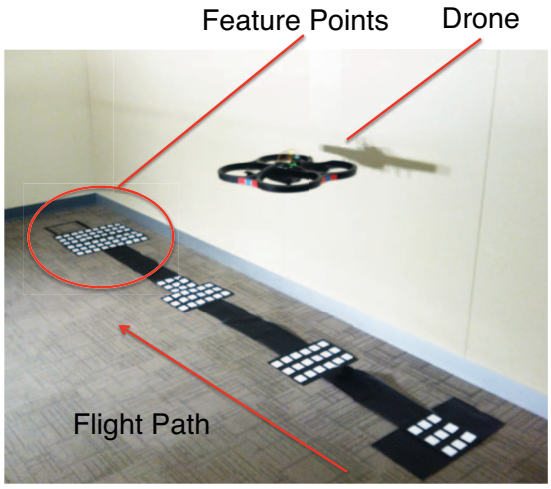


Figure 6. Experimental Setup

In this case, the resulting total utility resilience is 30.

F. The UAV Surveillance System Experiment

We now demonstrate the effectiveness of ZS-QRAM in a UAV surveillance mission. The mission is composed of the following three tasks: (1) a navigation task where the drone follows a wall along a corridor while using the IR sensors to avoid obstacles (2) a video processing task that uses the downward-facing camera to detect objects of interest, and (3) a lower-utility video streaming process that relays data from the forward-facing camera to a remote station.

Figure 6 shows our experimental setup. The drone is instructed to automatically takeoff starting at one end of the course. Using IR sensors on the front and side, the drone follows the wall as it hovers over the white features that can be detected by the downward facing camera. Wall following is achieved using a PID controller while the front-facing IR detectors is responsible for obstacle avoidance. In this example, if an obstacle is detected, the drone simply lands. While these control loops are running, the front-facing camera is collecting, compressing and transmitting video data over an 802.11 link to a monitoring PC. As the number of features increase, we expect to see an increase in the object-tracking task's CPU demand. This increase needs to be isolated from the wall following and low-level control tasks (Actuation, Sensor Data, Aux Sensor Data, and Navigation). Otherwise, the system will become unstable. To achieve isolation, we assign a significantly higher utility to these tasks as compared to the other two. The video streaming task is of a less-critical nature and hence if more CPU demand is required for downward object tracking, a reduction in streaming frame-rate would be acceptable. Table I shows the utilities of each of these tasks and their appropriate zero-slack instants. It is important to note that

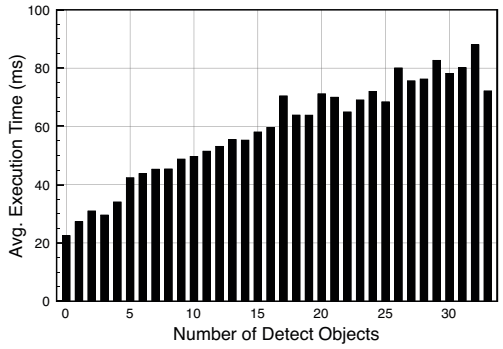


Figure 7. Average computational overhead vs number of blobs

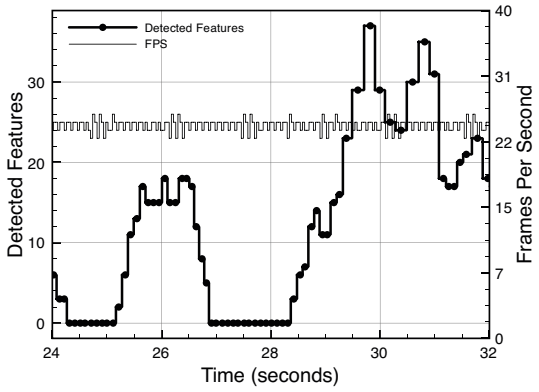


Figure 8. Streaming and object detection process using RMS

the streaming task operates at a higher rate and hence it will be assigned a higher scheduling priority by RMS. However, since it has a lower utility, it will be degraded to its longer period when the zero-slack instant of the object detection task is reached.

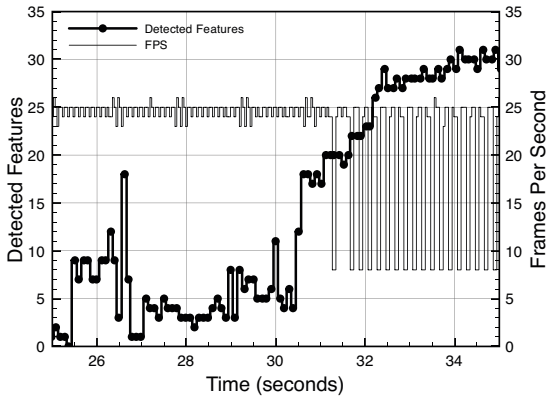


Figure 9. Streaming and object detection process using ZS-QRAM

Figure 7 shows the average execution time required per frame in relation to the number of features. As can be seen in Figure 6, the course that the drone navigates consists of clusters with an increasing number of objects followed by blank regions in between. As the drone encounters an increasing number of objects, the execution time of the feature detector increases. Figure 8 shows a plot of the number of frames per second along with the number of objects tracked as the drone traverses the course using RMS. We see a snapshot towards the end of the run where the drone flies over the last two clusters of obstacles. The second cluster has more features which is why we see a larger second peak in the graph. Since each additional tracked object tends to increase the task execution time, once the drone detects about 25 objects, the object detection task begins to exceed its allotted execution time of C_i. This first causes the object detection task to drift in time, meaning that the expected rate of sensor updates decreases. One can notice this effect by looking at the spacing between updates on the detected features line. This leads to the system missing frames and hence detecting fewer objects. This is reflected in the dip seen at time 31 seconds. Figure 9 shows a similar plot running ZS-QRAM. Note that since these are two distinct runs, the value may not be exactly identical, but the trends should be similar. Towards the end of the run, we see that the number of tracked features increases to about 25 when the streaming task is degraded. The utility of the streaming task is lower as compared to the feature detector and hence we see a period degradation (with the corresponding change in priority) to allow higher overall utility. After the feature detector completes, it re-enables the non-degraded period of the streaming task which returns to 25 frames per second for one cycle. With ZS-QRAM, we see not only consistent sensor computation timing but also a higher total number of objects being detected during the duration of the run. This CPS prototype illustrates how ZS-QRAM reallocates CPU cycles during an overload to keep the total utility of the system as high as possible without affecting critical tasks.

V. Conclusions

In this paper, we address the problem of large variations in the execution times of algorithms that are often found in cyber-physical systems. For example, computer vision and other image-processing tasks in real-world environments will demand potentially substantially more execution time when the sensor's field of view contains multiple artifacts. In other words, the characteristics of the physical environment can determine the resource demands placed on the cyber-components. We proposed a utility-based approach called ZS-QRAM (Zero-Slack QoS-based Resource Allocation Model) to allocate resources to tasks such that the benefits accrued to the system are maximized while CPS safety constraints are satisfied. In particular, we showed how the use of Q-RAM utility functions [3] and overbooking at design time can be combined with a utility degradation scheme at runtime to yield the highest system utility at any level of overloading. We also presented the *Utility Degradation Resilience* (UDR) metric that is used at design time to measure the capacity of the system to accrue utility under overload conditions. Using UDR, we were able to quantify and evaluate the difference between ZS-QRAM and traditional rate-monotonic scheduling when used in an UAV system. Finally, we conducted experiments where we measured the performance of object detection and video streaming tasks of our UAV system when a large number of objects in the Field-of-View (FoV) of the camera used by the former makes the latter overrun its nominal execution time. These experiments allowed us to verify the practical effects of our approach that degrades the frame rate of the video stream in order to enable the processing of the plethora of objects in the FoV. Such an adaptation is performed without disturbing the safety-critical tasks which are isolated from the mission-critical tasks.

References

[1] D. de Niz, K. Lakshmanan, and R. Rajkumar. On the scheduling of mixed-criticality real-time task sets. In *Real-Time Systems Symposium, 2009, RTSS 2009. 30th IEEE*, pages 291 –300, dec. 2009.

[2] Dionisio de Niz, Lutz Wrage, Antony Rowe, and Ragunathan (Raj) Rajkumar. ZS-QRAM: An Utility-Based Overbooking Scheduler for CPS (To be Published).

[3] R. Rajkumar, C. Lee, J. Lehoczky, and D. Siewiorek. A resource allocation model for qos management. In *Real-Time Systems Symposium, 1997. Proceedings., The 18th IEEE*, pages 298 –307, dec 1997.

[4] Pedro Mejia-Alvarez, Rami Melhem, and Daniel Mosse. An incremental approach to scheduling during overloads in real-time systems. In *Proceedings of the 21st, IEEE RTSS*, 2000.

[5] Giorgio Buttazzo, Marco Spuri, and Fabrizio Sensini. Value vs deadline scheduling in overload conditions. In *Proceedings of the 16st, IEEE Real-Time Systems Symposium*, 1995.

[6] Chi-Sheng Shih, Phanindra Ganti, and Lui Sha. Schedulability and fairness for computation tasks in surveillance radar systems. In *Proceedings of the 10th RealTime and Embedded Computing Systems and Applications Conference*, 2004.

[7] Giorgio Buttazzo, Giuseppe Lipari, and Luca Abeni. Elastic task model for adaptive rate control. *IEEE RTSS*, pages 286–295, 1998.

[8] Sanjoy Baruah, Haohan Li, and Leen Stougie. Towards the design of certifiable mixed-criticality systems. *Proceedings of the IEEE Real-Time Technology and Applications Symposium*, April 2010.

[9] Sanjoy K. Baruah, Alan Burns, and Robert I. Davis.

Response-time analysis for mixed criticality systems. In *RTSS*, pages 34–43, 2011.

[10] Parrot AR Drone: http://ardrone.parrot.com/ (viewed 10/21/2011).

[11] Drone-RK: http://www.drone-rk.org/ (viewed 10/21/2011).

[12] S. Oikawa and R. Rajkumar. Portable rk: a portable resource kernel for guaranteed and enforced timing behavior. In *Real-Time Technology and Applications Symposium, 1999. Proceedings of the Fifth IEEE*, pages 111 –120, 1999.

[13] S. Kato, R. Rajkumar, and Y. Ishikawa. A loadable real-time scheduler suite for multicore platforms. Technical report, Department of Electrical and Comptuter Engineering, Carnegie Mellon University, 2009.

[14] Karthik Lakshmanan, Dionisio de Niz, Ragunathan (Raj) Rajkumar, and Gabriel Moreno. Resource allocation in distributed mixed-criticality cyber-physical systems. *Distributed Computing Systems, International Conference on*, 0:169–178, 2010.