Resource Allocation in Distributed Mixed-Criticality Cyber-Physical Systems

Karthik Lakshmanan†, Dionisio de Niz⋆, Ragunathan (Raj) Rajkumar†, and Gabriel Moreno⋆
†Electrical & Computer Engineering and ⋆Software Engineering Institute
Carnegie Mellon University

***Abstract*—Large-scale distributed cyber-physical systems will have many sensors/actuators (each with local micro-controllers), and a distributed communication/computing backbone with multiple processors. Many cyber-physical applications will be safety-critical and in many cases unexpected workload spikes are likely to occur due to unpredictable changes in the physical environment. In the face of such overload scenarios, the desirable property in such systems is that the most critical applications continue to meet their deadlines. In this paper, we capture this *mixed-criticality* property by developing a formal overload-resilience metric called *ductility*. The generality of ductility enables it to evaluate any scheduling algorithm from the perspective of mixed-criticality cyber-physical systems. In distributed cyber-physical systems, this ductility is the result of both the task-to-processor packing (a.k.a bin packing) and the uniprocessor scheduling algorithms used. In this paper, we present a ductility-maximization packing algorithm to complement our previous work on mixed-criticality uniprocessor scheduling [6]. Our packing algorithm, known as *Compress-on-Overload Packing* (COP) is a criticality-aware greedy bin-packing algorithm that maximizes the tolerance of high-criticality tasks to overloads. We compare the ductility of COP against the Worst-Fit Decreasing (WFD) bin-packing heuristic used traditionally for load balancing in distributed systems, and show that the performance of COP dominates WFD in the average case and can reach close to five times better ductility when resources are limited. Finally, we illustrate the practical use of COP in distributed cyber-physical systems using a radar surveillance application, and provide an overview of the entire process from assigning task criticality levels to evaluating its performance.**

I. Introduction

Industrial-scale cyber-physical systems consist of multiple sensor, actuation, and computation subsystems running on a distributed platform. These systems are typically organized in sense-process-actuate pipelines that span across the entire platform. The physical phenomena interacting with the system impose strict timing constraints and potentially variable workloads on different pipeline stages. In such systems, the sensing and actuation tasks are typically pre-allocated to their individual dedicated subsystems, while the scheduling of *process* tasks and their allocation to processors presents an important resource allocation problem.

The motivating example for this paper is an Electronically Scanned Array (ESA) radar-surveillance system. This is an example of a large-scale distributed cyber-physical system, where the sensing of physical targets in the air space is accomplished by actuating the different radar arrays and sensing the corresponding echo. The processing is highly complex involving multiple tracking algorithms and filter implementations. The system experiences overload scenarios whenever there are more targets in the sky than it is able to process. Under such overload conditions, it is desirable that the most critical tasks, such as the ones tracking faster-moving or closer enemy targets, gain precedence in resource allocation over the less critical tasks tracking slower-moving or farther friendly targets. In order to formally capture this property, we develop a general *ductility* metric that characterizes the system behavior under overloads, and represents the value of resource allocation decisions in mixed-criticality systems.

In [6], we presented the *zero-slack rate-monotonic scheduling* (ZSRM) algorithm for mixed-criticality tasksets running on a single processor. ZSRM provides an asymmetric temporal protection guarantee under which low-criticality tasks cannot interfere with high-criticality tasks but high-criticality tasks can steal cycles from low-criticality tasks under overload scenarios. In our context, the high-criticality level tasks are thus treated as more important than all the other low-criticality tasks combined. In this paper, we study an important generalization of this problem to distributed systems, which are salient in many large-scale real-world settings.

When a distributed taskset is decomposed into a set of asynchronous tasks, these tasks can be scheduled using two different classes of schedulers: global scheduling and partitioned scheduling. A global scheduler uses a single global queue of ready tasks and assigns each of them to the different processors (or cores) according to their scheduling parameters (e.g. priority) at runtime. This has the effect that a task can run on any processor that the global scheduler finds appropriate at runtime. This also works only on Symmetric Multi-Processor (SMP) platforms due to the runtime costs involved in transferring the task state across the network. Partitioned scheduling, on the other hand, uses separate queues for each processor and each task is assigned to one processor *apriori*, and will only be scheduled on that processor. While the theoretical schedulable utilization of global scheduling is higher than that of partitioned scheduling, due to the costs associated with transferring the task state, we adopt the approach of partitioned scheduling in this paper.

In partitioned scheduling, the resource allocation decisions happen in two stages: (i) assignment of tasks to processors, and (ii) scheduling within a processor. The task-to-processor assignment problem is typically dealt as a bin-packing problem, where bin-packing algorithms are used to obtain near-

optimal solutions. In this paper, we present our bin-packing algorithm for mixed-criticality systems known as *Compress-on-Overload Packing* (COP) that extends the asymmetric protection scheme from ZSRM for distributed systems. We evaluate this algorithm using the ductility metric, and compare its performance against the WFD algorithm traditionally used for load balancing. Our experimental results show that COP dominates WFD in the average case, and can reach close to five times better ductility when resources are limited. Finally, we use a radar surveillance system case study to illustrate the practical benefits of our approach.

To the best of our knowledge, this is the first work to study the mixed-criticality scheduling problem in the context of distributed systems.

A. Related Work

Mixed-criticality systems are the subject of much recent research due to the emergence of cyber-physical systems. The US Air Force Research Laboratory has been leading a Mixed-Criticality Architecture Requirements (MCAR) [11] initiative to investigate building blocks to safely construct such mixed-criticality systems. Extensive real-time systems literature have already studied the offline overload scheduling problem [5, 19]. These studies are focused on uniprocessor systems, and try to maximize the accrued value from task completions. In contrast, our approach extends to distributed systems, and uses a notion of strict criticality ordering that is easier for system designers to deal with.

Researchers have also looked at online scheduling of overloads [2, 9, 15]. These schemes were not designed for mixed-criticality systems and do not include an explicit notion of criticality, and hence cannot take advantage of this notion. Approaches such as the elastic scheduling model [4] deal with overloads in a criticality-aware fashion by allowing tasks with higher elasticity to run at higher rates when required, whereas tasks with lesser elasticity are restricted to more steady rates. Earlier, we have developed the *zero-slack rate-monotonic scheduling* (ZSRM) algorithm [6] for mixed-criticality scheduling on a single processing setting. ZSRM provides asymmetric temporal protection guarantees under which low-criticality tasks cannot interfere with high-criticality tasks but high-criticality tasks can steal cycles from low-criticality tasks under overload scenarios to meet their deadlines. In this work, we extend this in two important directions: (i) We develop a performance metric to formally characterize the system behavior under different overload conditions, and (ii) We extend the algorithm to distributed systems through criticality-aware task allocation.

Scheduling tasks on a set of processors is a well-studied problem in the context of real-time systems. Traditional solutions are classified into [18, 23] (i) global scheduling, (ii) partitioned scheduling, and (iii) semi-partitioned scheduling. Among these solutions, partitioned scheduling algorithms incur the least runtime cost. Such cost includes task state migration, number of rescheduling instants, and a larger execution time of such rescheduling. In this work, we therefore focus on partitioned fixed-priority scheduling, while extensions and generalizations to other scheduling principles form a key component of our future work.

From the perspective of partitioned scheduling, it is well-known that the problem of optimally allocating tasks to processors is analogous to bin-packing, and hence proven to be NP-hard in the strong sense. Approximate solutions based on bin-packing heuristics with known polynomial-time complexity can achieve near-optimal task partitioning across multiple processors [7, 8]. For fixed-priority scheduling, it has been proven that the First-Fit Decreasing (FFD) and Best-Fit Decreasing (BFD) heuristics have an utilization bound of $(n+1)(2^{\frac{1}{2}}-1)$ with rate-monotonic scheduling, where n is the number of processors [12]. However, these algorithms are criticality-agnostic in nature, and provide very little room for high-criticality tasks to expand under overload scenarios.

Worst-Fit Decreasing (WFD) [8] is a bin-packing heuristic that is typically used for load balancing in distributed systems. While WFD distributes the slack among multiple processors, thereby enabling tasks to better deal with overload scenarios, it still does not make any distinction among criticality levels. However, we observe that explicitly considering the task criticality level during task allocation leads to better system behavior under overload scenarios. In this paper, we therefore present Compress-on-Overload Packing (COP), which is a criticality-aware bin-packing algorithm that can provide significantly better resource allocation to high-criticality tasks under overload scenarios when used with ZSRM.

In this paper, we focus on the problem of scheduling independent tasks with varying criticality levels on a set of processors. Allocating tasks with shared resources and synchronization requirements was studied in [17], where tasks that share data were coalesced into virtual tasks to ensure co-location whenever possible. In [13], the authors proposed an approach using shared buffers to eliminate inter-task dependencies. Although these approaches may be applied to deal with resource sharing and task interdependencies in the mixed-criticality context, such extensions are beyond the scope of this paper and form an important part of our future work.

II. Resource Allocation in Mixed-Criticality Systems

Partitioned scheduling in distributed systems with a single criticality level is decomposed into (i) task allocation (or assigning tasks to processors) and (ii) uniprocessor task scheduling. When multiple levels of criticality are involved, the system is required to ensure that under overload scenarios the most critical tasks are guaranteed to meet their deadlines. Providing such an overload property presents an important resource allocation problem in distributed mixed-criticality systems, which needs to be addressed at both the task allocation and uniprocessor scheduling phases.

Before illustrating this problem in detail, let us introduce the notation used in this paper.

Each task τ_i is defined as:

TABLE I
EXAMPLE TASK SET

$Task$	C_i (ms)	C_i^o (ms)	$Period$ T_i (ms)	$Criticality$ $\kappa(\tau_i)$
τ_{h1}	4	6	10	1
τ_{h2}	4	6	10	1
τ_l	2	3	5	2

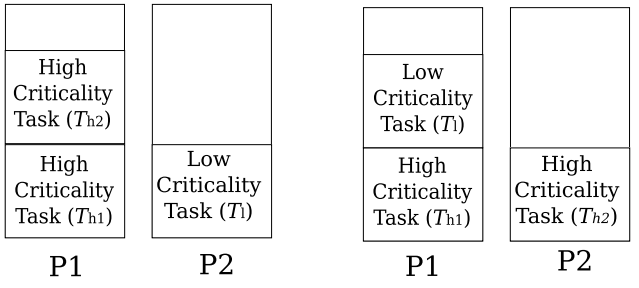
$$\tau_i = (C_i, C_i^o, T_i, \zeta_i)$$

where:

- C_i is its worst-case execution time under non-overloaded conditions,
- C_i^o is the overload execution budget,
- T_i is the period of the task and also its deadline.
- ζ_i is the criticality of the task. We follow the same convention as with scheduling priorities: lower value of ζ_i means higher criticality. For readability reasons, in order to explicitly represent the task under consideration, we will also use $\kappa(\tau_i)$ to denote the criticality ζ_i of task τ_i.

Within a processor, the criticality inversion problem can be addressed using ZSRM (as we will illustrate in subsection II-B), which guarantees that the low-criticality tasks cannot interfere with high-criticality tasks under overloads. Assuming ZSRM support, we first present the criticality inversion problem that arises during task allocation.

A. Criticality Inversion in Task Allocation



(a) Task Allocation that leads to Criticality Inversion

(b) Task Allocation that avoids Criticality Inversion

Fig. 1. Criticality Inversion in Allocation

In a distributed setting, the criticality inversion problem can arise at the task-allocation level, since a given allocation could favor a low-criticality task at the expense of a high-criticality one. For instance, consider three tasks τ_{h1}, τ_{h2}, and τ_l of high, high, and low criticality respectively, each having a normal utilization $\frac{C_i}{T_i}$ of 40% (shown in Table I). Say we only have two processors P_1 and P_2, and τ_{h1} is already deployed on P_1, and the three tasks do not fit on P_1 together, then we are forced to pack either τ_{h2} or τ_l on P_1 and the other to P_2. Packing τ_{h1} and τ_{h2} together, and τ_l by itself is a possible task allocation decision (see Figure 1(a)). In fact such an allocation decision is commonly used in legacy systems that try to isolate criticality levels. However, observe that such task allocation leads to a criticality inversion problem. In this scenario, if all the tasks overload, τ_{h2} may miss its deadline but τ_l will not i.e. our allocation decision protected τ_l (a low criticality task) at the expense of τ_{h2} (a high criticality task). Conversely, deploying τ_{h1} and τ_l together and τ_{h2} by itself removes this criticality inversion (see Figure 1(b)). Note that using a criticality-aware uniprocessor scheduling algorithm such as ZSRM will ensure that τ_l cannot steal cycles from τ_{h1} within processor P_1 under overload scenarios.

As illustrated by the above example, allocating tasks to processors can introduce criticality inversion, which affects the system performance under overloads. A criticality-aware task allocation algorithm can mitigate this problem by enabling more critical tasks to meet their deadlines under overload scenarios. In Section IV, we develop such a criticality-aware task allocation algorithm called Compress-on-Overload Packing (COP). For the benefit of the reader, we now provide a brief overview of the ZSRM algorithm used for criticality-aware uniprocessor scheduling.

B. ZSRM Overview

In [6] we characterized the criticality inversion problem for uniprocessors. In particular, we discussed how a symmetric temporal protection scheme can create a criticality inversion by stopping a high-criticality task when it tries to execute longer than its allocated execution budget, in order to let a lower-criticality task to run, making the former miss its deadline. ZSRM [6] provides an asymmetric protection scheme where high-criticality tasks can go into an overload budget by stealing cycles from low-criticality tasks.

ZSRM schedules such tasks as a meta-scheduler working on top of the rate-monotonic scheduler (RMS). It is based on the observation that criticality inversion only matters under overload conditions. We use this observation to create two execution modes for each task τ_i: (i) the normal mode (*N mode*), and (ii) the critical mode (*C mode*). Jobs of each task τ_i switch from their *N mode* to *C mode* at a relative *zero-slack instant* Z_i (for details on calculating Z_i see [6]). In the first mode, each job J_i of τ_i is scheduled using its RMS priority. When a job J_i of τ_i enters its second mode (i.e. does not complete within Z_i time units from its release), jobs of lower-criticality tasks will be suspended until J_i finishes. This suspension effectively blocks the interference of lower-criticality jobs under overloads, until the completion of the higher-criticality jobs experiencing the overload (It must be noted that τ_i itself can also be suspended by a task $\tau_c | \zeta_c < \zeta_i$ in τ_c's *C mode*).

For example, consider a processor P_1 in which tasks τ_{h1} and τ_l from Table I are allocated (as shown in Figure 1 (b)). The jobs of each task are assumed to have a relative deadline equal to the task period. In this example, τ_l has a shorter period and hence a higher RMS scheduling priority than τ_{h1}. The scheduling behavior under different overload scenarios is illustrated in Figure 2. Under ZSRM, the zero-slack instant of τ_{h1} was calculated as $Z_{h1} = 6$. As can be observed in Figure 2(a), when both τ_{h1} and τ_l overload simultaneously

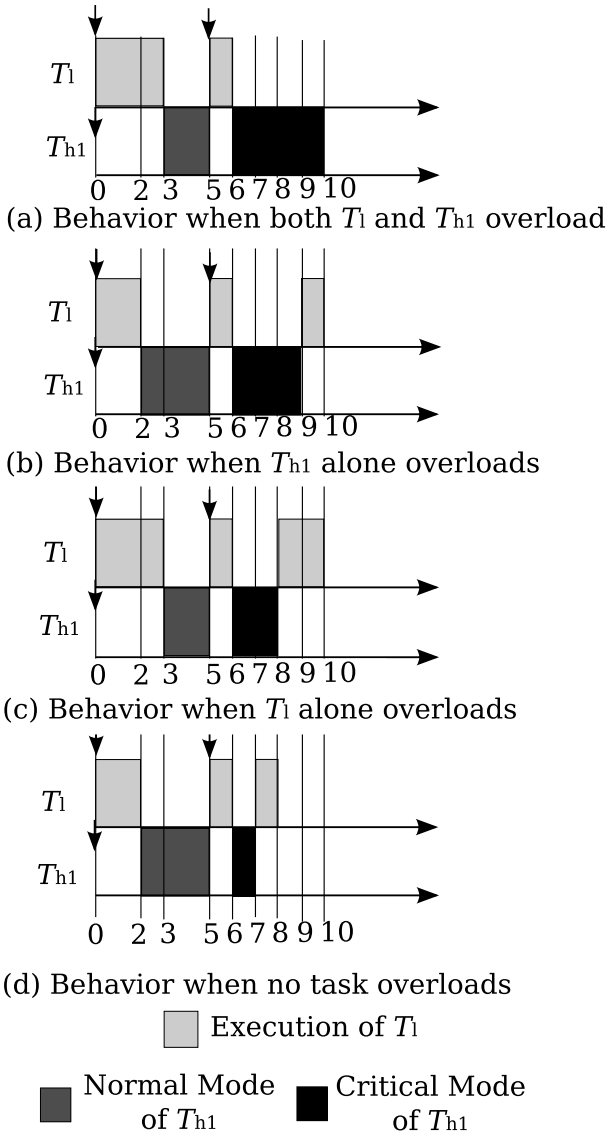


Fig. 2. Illustration of ZSRM

the second job of τ_l misses its deadline. The *zero-slack* instant of τ_{h1} occurs at time 6, thus effectively blocking τ_l until 10 causing the second job of τ_l to miss its deadline of 10 ensuring that the high-criticality task τ_{h1} meets its deadline of 10. In all other overload scenarios (shown in Figures 2(b), 2(c) and 2(d)), all the tasks meet their implicit deadlines.

As can be seen this example, ZSRM can ensure that the high-criticality task τ_{h1} can overload by stealing cycles from the low-criticality task τ_l. However, the high-criticality task τ_{h1} is protected from any overloads experienced by τ_l.

III. System Ductility

Mixed-criticality systems comprise of applications with different criticality levels and timing constraints co-located on a distributed set of processors. Such co-location is largely driven by cost and physical space considerations. The desirable property in mixed-criticality systems is that in the face of overload scenarios the most critical applications continue to meet their deadlines. In order to capture this property formally, we introduce a matrix called the *ductility matrix* to fully describe the potential behaviors of the system with respect to two factors: (1) the level of overload faced by tasks, and (2) tasks that miss their deadlines due to a given overload.

A. Illustration

In order to illustrate the characterization of a ductility matrix, let us consider the taskset given in Table I and allocation given in Figure 1(b). Observe that the task τ_{h2} allocated to processor P_2 will meet its deadline under all overload scenarios with utilization not exceeding 1, since it is allocated a processor on its own. We are interested in the scheduling behavior of τ_{h1} and τ_l allocated to processor P_1.

The different possible workload scenarios are:

1) Both tasks in criticality level 1 (τ_{h1} and τ_{h2}) and tasks in criticality level 2 (τ_l) experience an overload. Let us denote this workload as $w_3 =< 1, 1 >$, with the vector $< 1, 1 >$ representing that tasks in both criticality levels 1 and 2 are experiencing an overload.
2) Tasks in criticality level 1 (τ_{h1} and τ_{h2}) experience an overload, and tasks in criticality level 2 (τ_l) are under normal workloads. Let us denote this workload as $w_2 =< 1, 0 >$, with the vector $< 1, 0 >$ representing that tasks in criticality level 1 are overloaded while tasks in criticality level 2 are normal.
3) Tasks in criticality level 1 (τ_{h1} and τ_{h2}) are under normal workloads, and tasks in criticality level 2 (τ_l) experience an overload. Let us denote this workload as $w_1 =< 0, 1 >$, with the vector $< 0, 1 >$ representing that tasks in criticality level 2 are overloaded while tasks in criticality level 1 are normal.
4) Both tasks in criticality level 1 (τ_{h1} and τ_{h2}) and tasks in criticality level 2 (τ_l) are under normal workloads. Let us denote this workload as $w_0 =< 0, 0 >$, with the vector $< 0, 0 >$ representing that task in both criticality levels 1 and 2 are under normal workloads.

For any given workload vector W, we can determine the tasks that are guaranteed to meet their deadlines (a deadline guarantee vector). In this example taskset, based on Figure 2 observe that:

1) When $w_3 =< 1, 1 >$, task τ_l will miss its deadline, leading to a deadline guarantee vector of $D_3 =< 1, 0 >$ (This deadline guarantee vector D_3 denotes that tasks in criticality level 1 are guaranteed to meet their deadlines, while tasks in criticality level 2 are not guaranteed to meet their deadlines).
2) When $w_2 =< 1, 0 >$, all tasks will meet their deadlines, leading to a deadline guarantee vector of $D_2 =< 1, 1 >$.
3) When $w_1 =< 0, 1 >$, all tasks will meet their deadlines, leading to a deadline guarantee vector of $D_1 =< 1, 1 >$.
4) When $w_0 =< 0, 0 >$, all tasks will meet their deadlines, leading to a deadline guarantee vector of $D_0 =< 1, 1 >$.

These deadline guarantee vectors $D_3...D_0$ collectively describe the performance guarantees provided by the system under different overload conditions. These vectors can be represented as a matrix that we denote as the *Ductility Matrix* D. In this example scenario, $D = [D_3\ D_2\ D_1\ D_0]^t$, which is

$$D = \begin{pmatrix} 1 & 0 \\ 1 & 1 \\ 1 & 1 \\ 1 & 1 \end{pmatrix}$$

We now proceed to generalize this concept of a *Ductility Matrix* for arbitrary tasksets.

B. Ductility Matrix

Consider a set of n tasks $\{\tau\} = \{\tau_1, \tau_2, ..., \tau_n\}$ and k criticality levels $\{\kappa\} = \{\kappa_1, \kappa_2, ..., \kappa_k\}$. Let each task τ_i have a criticality value $\kappa(\tau_i) \in \{\kappa\}$. Let lower values denote higher criticalities, i.e. $\kappa(\tau_i) < \kappa(\tau_j)$ implies that τ_i is more critical than τ_j. Without loss of generality, the given set of criticality levels $\{\kappa\}$ is assumed to be in the decreasing order of criticalities i.e. $\kappa_s < \kappa_t\ \forall s < t$.

Although the concept of ductility can be extended in a straight-forward fashion to deal with any number γ of operating states, we restrict our attention to 2 operating states (C_i and C_i^o) for simplicity of presentation. From a practical perspective, system designers may find it harder to deal with a multitude of operating states.

The workload of the system can be characterized by the *workload vector* $< W_1, W_2, ..., W_m >$, where W_i is an indicator variable that denotes the operating state of tasks with criticality value κ_i. $W_i = 0$ denotes that all tasks with criticality κ_i are in the normal operating state. $W_i = 1$ denotes that a task with criticality κ_i is in the overload operating state. In addition, we define a scalar equivalent known as *system workload* (w) that is a comparable quantity to work with, which is computed from the *workload vector* as:

$$w = \sum_{g=1}^{k} \{2^{k-g} W_g\}$$

Observe that the *system workload* as defined above results in a strict ordering among the overloads faced by criticality levels. Criticality level κ_g is assigned a weight of 2^{k-g}, therefore, any additional overload in criticality level κ_g results in more *system overload* (at least 2^{k-g} additional system overload) than it is possible to add through the maximum overloading of all lower-criticality levels $\{\kappa_l\}\ \forall l > g$ (at most $\sum_{l=g+1}^{k} 2^{k-l} = (2^{k-g} - 1)$ additional system overload). This property captures the requirement that a high-criticality level be treated as more important than all the other low-criticality levels combined.

The *ductility matrix* defines the system timing behavior under possible overload conditions. It is a $2^k \times k$ matrix, where the rows correspond to different possible *system workload* values in decreasing order $\{2^k - 1, 2^k - 2, ..., 0\}$ i.e. row r ($r \in [1, 2^k]$) has a *system workload* of $2^k - r$. The columns correspond to the different possible criticality values in the decreasing order $\kappa_1, \kappa_2, ..., \kappa_k$. The ductility matrix $D = (d_{r,c})$ encodes whether all the tasks in criticality level κ_c meet their deadlines under a *system workload* of $w = 2^k - r$.

- When $d_{r,c} = 1$, it means that all the tasks in criticality level κ_c are guaranteed to meet their deadlines under workload $w = 2^k - r$.
- When $d_{r,c} = 0$, it means that at least one task in criticality level κ_c is not guaranteed to meet its deadlines under workload $w = 2^k - r$.

$$D = \begin{pmatrix} d_{1,1} & d_{1,2} & ... & d_{1,k} \\ d_{2,1} & d_{2,2} & ... & d_{2,k} \\ ... & ... & ... & ... \\ d_{2^k,1} & d_{2^k,2} & ... & d_{2^k,k} \end{pmatrix}$$

The ductility matrix describes the system performance with respect to criticality levels and all possible overload scenarios. Given the taskset and scheduling algorithm used in the system, the ductility matrix can be calculated to describe the deadline guarantees provided by the system.

C. Normalized Ductility

The ductility matrix is a comprehensive description of the system performance with respect to criticality levels. To simplify the evaluation of different scheduling algorithms, we define a scalar equivalent of the ductility matrix that can be ordered based on magnitudes. To obtain this scalar we define the projection function P_d to map a ductility matrix D to a scalar as:

$$P_d(D) = \sum_{c=1}^{k} \{\frac{1}{2^c} \frac{\sum_{r=1}^{2^k} d_{r,c}}{2^k}\}$$

The key properties of this projection function P_d are:

1) All entries within each column belong to the same criticality level, and are therefore treated equally without assigning different weights to each row (using $\sum_{r=1}^{2^k} d_{r,c}$).
2) The contribution to the final scalar $P_d(D)$ of having a 1 in any column c is larger than the contribution of having all ones in all other columns l with lower criticality. Thus, each criticality level κ_c is treated as absolutely more important than any other lower criticality level κ_l $\forall c < l \leq k$. This is accomplished by normalizing the contribution of each column c to the range $[0, 1]$ (by applying a scale of $\frac{1}{2^k}$ to $\sum_{r=1}^{2^k} d_{r,c}$) and subsequently applying a weight of $\frac{1}{2^c}$ to impose a strict ordering among columns.

Note that the maximum value of P_d will be

$$\sum_{c=1}^{k} \frac{1}{2^c} = 1 - \frac{1}{2^k}.$$

Therefore, we obtain the normalized ductility ν (normalized to the range $[0, 1]$) as:

$$\nu = \frac{P_d(D)}{1 - \frac{1}{2^k}}$$

In this paper, we will use this *normalized ductility* ν to compare the performance of various scheduling algorithms. Other projection functions are also possible. However, we believe that P_d succinctly captures the mixed-criticality scheduling requirements from the system designer's perspective. It should be noted here that our definition of the ductility matrix does not preclude the possibility of using other projection functions. For mixed-criticality systems, it would be desirable to assign larger weights to higher criticality tasks. We use the projection function more as a metric for comparing different allocation algorithms rather than as a guiding objective function in our bin-packing algorithms, therefore, our Compress-On-Overload Packer will not be affected by choosing other such projection functions.

IV. Compress-on-Overload Packer (COP)

The task allocation problem in distributed mixed-criticality systems has a dual objective. (i) We need to minimize the deadline misses of high-criticality tasks under all possible overload cases. (ii) We also need to minimize the number of processors needed by the system. In order to achieve this, we design a two-phase algorithm. In the first phase, we allocate the tasks ensuring that all of them are schedulable even if they run their corresponding C^o. For this packing we explore three variants of bin packers based on: Best-Fit Decreasing (BFD), First-Fit Decreasing (FFD), and Worst-Fit Decreasing (WFD). These packers try to fit tasks ordered first by criticality, and then in decreasing order of overload utilization ($\frac{C^o}{T}$). Each task considers each available processor in an order based on the overloaded fullness level ($\sum_{\forall i} \frac{C_i^o}{T_i}$) as defined by the different variants: decreasing for BFD, increasing for WFD, and without any order for FFD. Any task that is unschedulable is kept aside. This first phase thus completely eliminates the consequence of overloading and ensures that no allocated task can miss its deadline on overload. At the end of this phase, we will have a set of tasks that did not fit. These tasks are now packed using a modified worst-fit decreasing (WFD) packing. Under this packing, tasks are ordered first by criticality, and then in the decreasing order of normal utilization ($\frac{C}{T}$). Each task considers each available processor in the increasing order of normal fullness level. For this second phase, we let ZSRM compress the schedule by allowing it to calculate the zero-slack instant that can potentially allow the higher-criticality tasks to suspend lower-criticality tasks (stealing their cycles). Such a compression reduces the number of processors needed to schedule the tasks under ZSRM at the expense of deadline misses incurred by the lower-criticality tasks under overloads. At the end of this second phase, some tasks might still be unschedulable and left outside of the packed set. These unpacked tasks are evaluated in the *ductility* metric as tasks that always miss their deadlines. The effect of bin count is evaluated in the next section.

TABLE II
Task Types

Task τ_i	C_i	C_i^o	T_i
τ_1	10	50	100
τ_2	20	100	200
τ_3	30	200	400

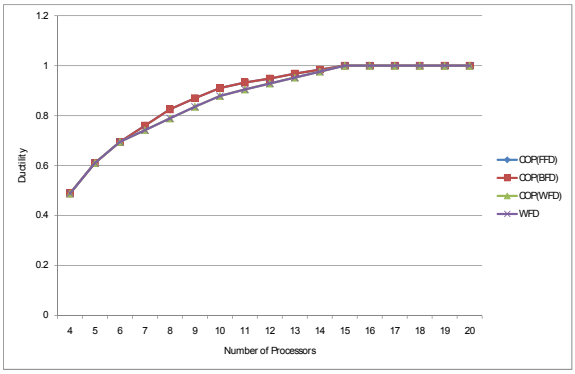


Fig. 3. Comparison at criticality vector <0,1,2>

V. Evaluation

The performance of mixed-criticality scheduling algorithms needs to be evaluated along two dimensions: (i) normal schedulability, and (ii) overload behavior. Classical bin-packing algorithms for (non-mixed-criticality) distributed systems are typically evaluated exclusively along the dimension of normal schedulability. For any given taskset, the performance of different bin-packing algorithms along the dimension of normal schedulability can be compared by determining the number of processors required by each algorithm. However, in our case, we want to evaluate the effectiveness of the algorithms to extract the maximum ductility out of a given number of processors. Therefore, the ductility that the algorithms can obtain for different processor counts is compared. Our three COP algorithm variants are compared to WFD (all of them using ZSRM for uniprocessor scheduling) given that it is designed to balance the load, and hence the slack, across all the available processors. In our implementation of WFD, tasks with equal sizes were sorted using the increasing order of periods. For our evaluation we first use a specific taskset that allows us to better discuss the behavior of our algorithms and then we evaluate their average case behavior.

Let us start by evaluating the performance of the COP variants and WFD with the specific taskset shown in Table II. Each task of type τ_i was assigned to criticality level L_i, and the criticality vector $< L_1, L_2, L_3 >$ is varied along the x-axis as $< 0, 1, 2 >$,$< 0, 2, 1 >$, $< 1, 0, 2 >$,$< 1, 2, 0 >$,$< 2, 0, 1 >$, $< 2, 1, 0 >$. These variations represent the level of priority-to-criticality misalignment with $< 0, 1, 2 >$ being fully aligned and $< 2, 0, 1 >$ fully misaligned. We only present these two criticality variations given that they clearly present the inter-

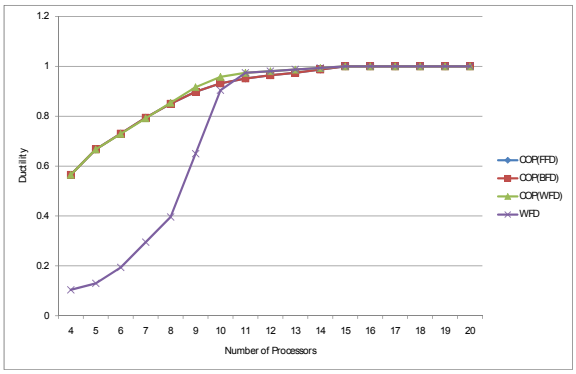


Fig. 4. Comparison at criticality vector <2,1,0>

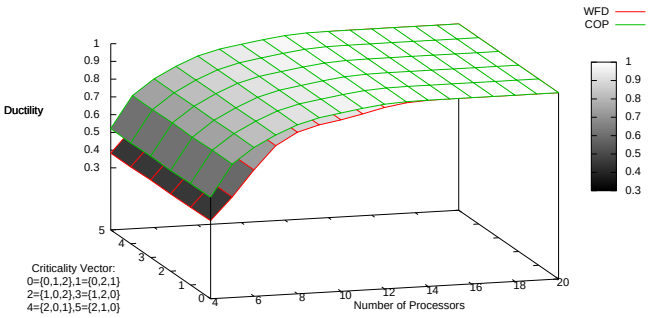


Fig. 5. Surface of Average Performance

esting behaviors. Figure 3 shows the performance at criticality assignment $< 0, 1, 2 >$. In this scenario, the task criticalities are assigned to task priorities. As shown in Figure 3, all COP variants as well as WFD exhibit very similar performance since there is no criticality inversion and the sorting order of WFD matches COP (since all the tasks have equal sizes and are hence sorted by WFD based on priority, which equals criticality under assignment $< 0, 1, 2 >$). However, under a criticality assignment of $< 2, 1, 0 >$, the taskset experiences maximum criticality inversion since the criticalities are exactly in the reverse order of priorities. Figure 4 shows that the COP variants achieve significantly better performance compared to WFD when a small number of processors is available (almost five-fold when only a minimum number of processors are available). As the number of processors increases, the performance difference between the COP variants and WFD decreases. When the number of available processors approaches a large enough value that is sufficient to schedule the overloaded tasksets themselves, WFD performs slightly better than the FFD and BFD COP variants (whose lines are completely overlapping) while the WFD COP variant aligns itself to WFD. This is largely due to the approximate nature of the heuristics themselves. The FFD and BFD COP variants used in its first phase, can perform worse than WFD for specific tasksets.

Let us now evaluate the average case performance of our algorithms. Since our results do not show any significant difference between the three COP variants, for the average case evaluation we only present the BFD variant. We will refer to this variant just as COP. Figure 5 shows the average ductility achieved using COP in comparison with the average ductility achieved using WFD. These results were obtained using randomly generated tasksets having 30 tasks each. In order to isolate the effects of bin packing from any rate-monotonic scheduling effects that may arise from non-harmonic task period ratios, we constrained our task sets to have harmonic task periods T_i from the set {100,200,400,800,1600}. The overloaded computation time C_i^o of each task was chosen in an uniformly random fashion between $\frac{1}{6}T_i$ and $\frac{1}{2}T_i$. Subsequently, the normal computation time C_i of each task was chosen in an uniformly random fashion between $\frac{1}{12}T_i$ and $\frac{1}{2}C_i^o$. We did not choose an overload utilization greater than $\frac{1}{2}$ since such tasks are typically allocated to their own processors, and using such tasks would therefore unfairly affect WFD. We also ensure that the utilization is at least $\frac{1}{12}$ because it is the smallest object size that can be part of a wost-case packing [1] ($\frac{1}{6}$ for C^o to enable the selection of a C as a fraction of C^o that is still larger than $\frac{1}{12}$). The overload workload was also restricted to be within a factor of two from the normal workload to focus on more stressful task sets. The tasks were also assigned to a criticality level in an uniformly random fashion from $< L_1, L_2, L_3 >$.

The criticality levels are varied in the same fashion as in the previous experiment. The number of available processors was increased from 4 to 20 along the y-axis. The z-axis presents the average ductility value (1000 experiments for each data point) given the criticality assignment and number of available processors. Results in Figure 5 show that COP outperforms WFD significantly when the system has a fewer number of available processors. This behavior is largely due to the fact that WFD performs its allocation decisions in a criticality-agnostic fashion, thereby potentially packing high-criticality tasks in the same processor resulting in poor performance under overload conditions. COP, on the other hand, spreads the high-criticality tasks among the available processors, thus resulting in much better performance during system overloads. As the number of available processors increases, both COP and WFD are able to allocate more slack in each processor, which leads to better overload behavior. When the number of available processors is increased beyond 15 all tasks become schedulable even with their overloaded utilization $\frac{C_i^o}{T_i}$. Therefore, both COP and WFD achieve the maximum ductility of 1.0.

Our evaluation results show that COP performs better for mixed-criticality systems compared to traditional WFD, by taking into account both task criticality and sizes. We illustrated this using the ductility metric developed in Section III. Based on the evaluation, COP is best suited for mixed-criticality systems where (i) there are fewer number

of processors than required to schedule the overloaded taskset itself, and (ii) criticality of tasks are misaligned with their priorities.

VI. A Radar Surveillance Case Study

In this section, we demonstrate an application of mixed-criticality scheduling in the context of phased-array radars. Electronically scanned antenna (ESA) or phased-array radars have the ability to quickly redirect the radar beam in any position without inertia [3]. This provides greater flexibility compared to mechanically scanned antennas because the two main functions performed by the radar: (i) search for new targets, and (ii) track targets, need not be driven by a common period. Furthermore, it is possible to illuminate targets with different frequency depending on appropriate criterion, such as their range, so that some tracks are updated more often than others. At the same time, there are deadlines that must be met in order to keep track of targets.

Briefly, the tracking process in a phased-array radar is as follows. A search task scans the surveillance area looking for new targets that the radar is not yet tracking. When a new target is found, a confirmation task is started to illuminate the candidate target two more times to make sure it is not spurious. If confirmed, a new track representing the target is created. For each track held by the radar, the tracking process periodically determines the speed and direction of the target using tracking algorithms based on the Kalman filter and $\alpha\beta\gamma$ filter [14]. These algorithms combine the radar observation of a target (with uncertainty), with previous track data to form an estimate of the actual speed and direction. The next time a track has to be updated, the radar beam is aimed at the estimated position of the target (considering the time elapsed since the last update) to get a new observation. Since the error of this estimate gets larger as time passes, missing a deadline to update a track may cause the radar beam to miss the target. If that happens, the radar loses track of the target and the track is dropped.

Scheduling of tasks in phased-array radars is an interesting and multifaceted problem [10, 16, 22]. We focus on the mixed-criticality aspect of this problem. Different functions performed by the radar have different criticality (e.g., searching for new targets is more critical than tracking existing ones). Even within the same radar function, there can be mixed-criticality (e.g., tracking a hostile target is more critical than tracking a friendly one). In order to demonstrate the benefits of using appropriate mixed-criticality scheduling, we developed a simulation of a phased-array radar with a taskset that exhibits such mixed-criticality nature.

Figure 6 shows the architecture of our radar simulation. The *TrackGen* task maintains a model of the airspace where the simulated radar operates, simulating the movement of targets. This model is maintained in a repository that can be read by other tasks. *TrackGen* can also vary the system workload by adjusting the number of targets in the airspace according to predetermined settings. The *Display* task provides the graphical user interface used in our simulation. The *TrackSend* task, periodically broadcasts messages over the network to update the location of targets in the airspace. This enables one or more displays to be run on different computers. The *Search* task scans the airspace periodically, looking for new targets. When a new target is found, it is delegated to one of the tracking tasks so that it can be tracked.

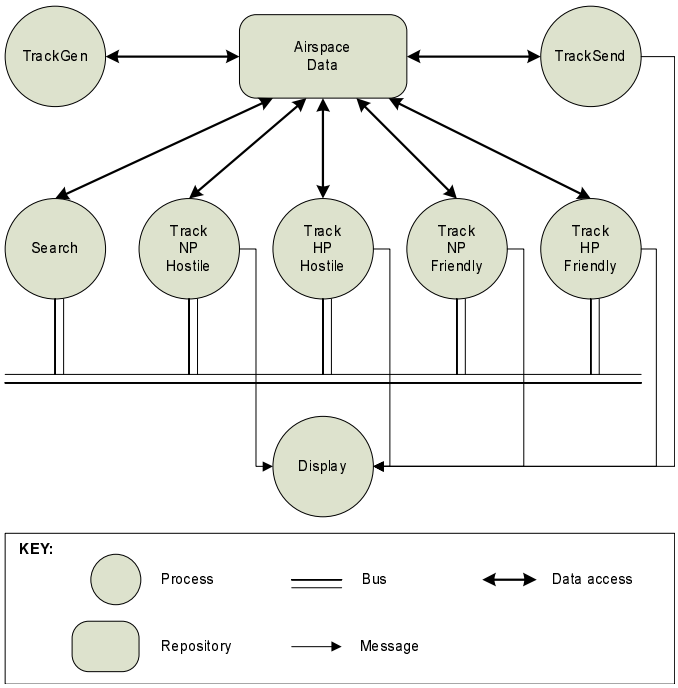


Fig. 6. Architecture of the radar simulation

Tracks are classified along two dimensions: range, and type. Depending on the range, they can be normal priority (NP) or high priority (HP) tracks, with the latter being closer in range, and, consequently, requiring more frequent updates and higher precision. Along the other dimension, they are classified as either friendly or hostile, with the latter being more critical. There are four *Track* tasks to handle these different track classes. The *Search* task and the tracking tasks are connected by a bus that allows the *Search* task to delegate a track to the appropriate tracking task. Even after a track is delegated by *Search* to a tracking task, the track can cross the range boundary between NP and HP and may have to be transferred to a different tracking task. Note that this does not affect the task periods or criticality, and only affects the task computation time that depends on the number of tracks being handled by the task.

The tracking tasks send updates with the estimated track parameters to the display. If a tracking task misses its deadline, and consequently fails to report the updated tracks to the display, the display will provide a visual indication by fading the corresponding tracks.

Table III shows the relevant subset of tasks from our radar simulation platform running on an Intel Core 2 Extreme processor with four processing cores, each clocked at

TABLE III
RADAR SURVEILLANCE TASK SET

$Task$	C_i (ms)	C_i^o (ms)	$Period$ T_i (ms)	$Criticality$ $\kappa(\tau_i)$
HP Hostile	40	58	100	1
NP Hostile	83	106	200	1
HP Friendly	40	58	100	2
NP Friendly	83	106	200	2

TABLE IV
DEADLINE MISSES

Packer	Deadline misses (% misses { Task })		
	2300 Tracks	2400 Tracks	2500 Tracks
WFD	0	24.32 {NP Hostile}	69.56 {NP Hostile}
COP	0	10.34 {HP Friendly}	59.18 {HP Friendly}

2.526GHz with 32KB Instruction Cache, 32KB Data Cache, and 6MB Unified L2 Cache. We use the Linux/RK [20, 21] infrastructure, which we have extended to provide support for mixed-criticality resource reservations using ZSRM. *HP Hostile* and *HP Friendly* are examples of near-range tracking algorithms that require a higher sampling rate (10Hz or a 100ms period), whereas, *NP Hostile* and *NP Friendly* are examples are far-range tracking algorithms that only need a lower sampling rate (5Hz or a 200ms period). The criticality levels reflect whether the tasks are used to track hostile (*HP Hostile* and *NP Hostile*) or friendly (*HP Friendly* and *NP Friendly*) objects. There are two execution times for each task: C is the execution time when the radar has a normal workload, and C^o is the execution requirement for each task when the system is overloaded.

A. Task Allocation and Scheduling

In our setup, there are four available processing cores P_1,P_2,P_3, and P_4. The *TrackGen*, *TrackSend*, and *Search* tasks were allocated to processing core P_3, and the processing core P_4 was dedicated for display. For simplicity of illustration, we restrict our attention to the allocation and scheduling of the *HP Hostile*, *NP Hostile*, *HP Friendly*, and *NP Friendly* tasks among processing cores P_1 and P_2.

The summary of our performance evaluation is shown in Table IV.

1) Performance Evaluation under WFD: Under Worst-Fit Decreasing (WFD) task allocation and Zero-Slack Rate-Monotonic (ZSRM), the *HP Hostile* and *NP Hostile* tasks are assigned to processor P_1, and the *HP Friendly* and *NP Friendly* tasks are assigned to processor P_2.

The ductility matrix $D(WFD)$ under this scenario is given by:

$$D(WFD) = \begin{array}{l} w = 3 =< 1,1 > \\ w = 2 =< 1,0 > \\ w = 1 =< 0,1 > \\ w = 0 =< 0,0 > \end{array} \begin{pmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 0 \\ 1 & 1 \end{pmatrix}$$

The ductility matrix is developed as follows:

1) When $w = 3 =< 1,1 >$, both criticality levels 1 and 2 are overloaded. The *NP Hostile* task (in criticality level 1) will miss its deadline in processor P_1 ($d_{1,1} = 0$)and *NP Friendly* task (in criticality level 2) will miss its deadline in processor P_2 ($d_{1,2} = 0$).
2) When $w = 2 =< 1,0 >$, criticality level 1 is overloaded. The *NP Hostile* task (in criticality level 1) will miss its deadline in processor P_1 ($d_{2,1} = 0$) and all other tasks will meet their deadlines ($d_{2,2} = 1$).
3) When $w = 1 =< 0,1 >$, criticality level 2 is overloaded. The *NP Friendly* task (in criticality level 2) will miss its deadline in processor P_2 ($d_{3,2} = 0$) and all other tasks will meet their deadlines ($d_{3,1} = 1$).
4) When $w = 0 =< 0,0 >$, both criticality levels are in normal conditions. All tasks meet their deadlines ($d_{4,1} = d_{4,2} = 1$).

The ductility is given by $P_d(D(WFD)) = (\frac{1}{2}\frac{1}{2} + \frac{1}{2^2}\frac{1}{2}) = 0.375$, since tasks in κ_1 meets its deadlines only under $w = 1$ and $w = 0$ and gets a weight of $\frac{1}{2}$ (high criticality), while κ_2 meets its deadlines only under $w = 2$ and $w = 0$ and gets a weight of $\frac{1}{2^2}$ (low criticality).

The normalized ductility $\nu(WFD) = \frac{0.375}{0.75} = 0.5$ (since the maximum ductility for two criticality levels is $1 - \frac{1}{2^2} = 0.75$).

As seen in Table IV, as the system workload increased beyond the maximum normal workload of 2300 tracks, the WFD packing scheme resulted in *NP Hostile* missing deadlines due to the overloading of *HP Hostile* on processor P_1. This behavior is clearly not desirable since the resource allocation decision of assigning *NP Hostile* to P_2 would not only protect *NP Hostile*, but also enable it to steal cycles from other tasks under overload. In terms of the ductility metric, this results in deadline misses at criticality level 1, which has a high weight of $\frac{1}{2}$.

2) Performance Evaluation under COP: Under Compress-on-Overload Packing (COP) and Zero-Slack Rate-Monotonic (ZSRM) scheduling, the *HP Hostile* and *HP Friendly* task are allocated to processor P_1, and the *NP Hostile* and *NP Friendly* tasks are allocated to processor P_2.

The ductility matrix $D(COP)$ under this scenario is given by:

$$D(COP) = \begin{array}{l} w = 3 =< 1,1 > \\ w = 2 =< 1,0 > \\ w = 1 =< 0,1 > \\ w = 0 =< 0,0 > \end{array} \begin{pmatrix} 1 & 0 \\ 1 & 1 \\ 1 & 1 \\ 1 & 1 \end{pmatrix}$$

The ductility matrix is developed as follows:

- When $w = 3 =< 1,1 >$, both criticality levels 1 and 2 are overloaded. The *HP Friendly* task (in criticality level 2) will miss its deadline in processor P_1 and *NP Friendly* task (in criticality level 2) will miss its deadline in processor P_2 ($d_{1,2} = 0$) and all other tasks will meet their deadlines ($d_{1,1} = 1$).
- When $w = 2 =< 1,0 >$, criticality level 1 is overloaded. All tasks meet their deadlines ($d_{2,1} = d_{2,2} = 1$).
- When $w = 1 =< 0,1 >$, criticality level 2 is overloaded. All tasks meet their deadlines ($d_{3,1} = d_{3,2} = 1$).

- When $w = 0 =< 0, 0 >$, both criticality levels are in normal conditions. All tasks meet their deadlines ($d_{4,1} = d_{4,2} = 1$).

The ductility under ZSRM is given by $P_d(D(COP)) = (\frac{1}{2} + \frac{1}{2^2}\frac{3}{4}) = 0.6875$ since tasks in κ_1 meets its deadlines only under all overloads and gets a weight of $\frac{1}{2}$ (high criticality), while κ_2 meets its deadlines for $w < 3$ and gets a weight of $\frac{1}{2^2}$ (low criticality).

The normalized ductility $\nu(COP) = \frac{0.6875}{0.75} = 0.9167$ (since the maximum ductility for two criticality levels is 0.75).

As seen in Table IV, as the system workload increases beyond the maximum normal workload of 2300 tracks, the COP packing scheme results in *HP Friendly* missing deadlines due to the overloading of *HP Hostile* on processor P_1. This is much better behavior since *NP Hostile* was protected from the overload behavior of *HP Hostile*. In terms of the ductility metric, this results in deadline misses at criticality level 2, which has a lower weight of $\frac{1}{4}$. In our experimental surveillance setup, the *HP Hostile* task overloaded before the *NP Hostile* task, therefore, the *NP Friendly* task did not miss any deadlines on processor P_2.

As can be seen from the radar surveillance example, allocating high-criticality tasks by spreading them across the processing cores enables them to expand under overload, resulting in much better system *ductility*. Compress-on-Overload Packing (COP) can therefore achieve a higher *ductility* value 0.9167 than the conventional criticality-agnostic task allocation heuristics like WFD 0.5. The higher ductility value directly translates to better system performance with respect to criticality levels under overload scenarios (as shown in Table IV).

VII. Conclusions

In this paper, we developed the *ductility* metric to characterize the overload behavior of mixed-criticality cyber-physical systems. A highly ductile system has the property that under overload scenarios, the high-criticality tasks continue to meet their deadlines by stealing cycles from low-criticality tasks. We illustrated the importance of task allocation in distributed mixed-criticality systems from the perspective of increasing ductility. To maximize the ductility on distributed systems, we developed a new bin-packing algorithm for mixed-criticality real-time systems known as *Compress-on-Overload Packing* (COP) that works on top of the *zero-slack rate-monotonic* scheduler. We evaluated the average-case ductility of COP, and compared it against the average-case ductility of Worst-Fit Decreasing (WFD) that is typically used in load balancing. Our experiments show that COP dominates WFD in the average case and can provide close to five times more ductility when resources are limited. Finally, we illustrated the practicality of our approach with a radar surveillance application, which is an example of a large-scale distributed cyber-physical system. We used this example to discuss the assignment of criticalities in a real setting, and evaluated the system behavior.

References

[1] Brenda S. Baker. A new proof for the first-fit decreasing bin-packing algorithm. *Journal of Algorithms*, 6:49–70, 1985.

[2] S.K. Baruah and J.R. Haritsa. Scheduling for overload in real-time systems. *IEEE Transactions on Computers*, 46(9):1034–1039, 1997.

[3] Samuel S. Blackman. Multiple-target tracking with radar applications. 1986.

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

[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] Dionisio de Niz, Karthik Lakshmanan, and Raj Rajkumar. On the scheduling of mixed-criticality real-time task sets. *Proceedings of the 30th Real-Time Systems Symposium*, 2009.

[7] D.S.Johnson. Near-optimal bin packing algorithms. *Ph.D. Thesis, Department of Mathematics, Massachusetts Institute of Technology*, 1973.

[8] D.S.Johnson. Fast algorithms for bin packing. *Journal of Computr. Syst. Sci. 8*, pages 272–314, 1974.

[9] M.K. Gardner and Liu J.W.S. Performance algorithms for scheduling real-time systems with overrun and overload. In *Proceedings of the 11th ECRTS*, 1999.

[10] Sourav Ghosh, Ragunathan Rajkumar, Jeffery Hansen, and John Lehoczky. Scalable qos-based resource allocation in hierarchical networked environment. *In Proceedings of 11th IEEE Real-Time Technology and Applications Symposium*, 2005.

[11] David Homan. Designing future systems for airworthiness certification. http://www.cse.wustl.edu/~cdgill/CPSWEEK09_MCAR/MCAR%20overview%20BoF%20CPS%20PA%20approved.pdf, 2009.

[12] J.M.Lopez, J.L.Diaz, and D.F.Garcia. Minimum and maximum utilization bounds for multiprocessor rm scheduling. *Proceedings of the 13th Euromicro Conference on Real-Time Systems*, 2001.

[13] Sharath Kodase, Shige Wang, Zonghua Gu, and Kang G. Shin. Improving scalability of task allocation and scheduling in large distributed real-time systems using shared buffers. *Real-Time and Embedded Technology and Applications Symposium, IEEE*, 0:181, 2003.

[14] Michael Kolawole. Radar systems, peak detection, and tracking. 2002.

[15] Gilad Koren and Dennis Shasha. Dover; an optimal on-line scheduling algorithm for overloaded real-time systems. In *RTSS'92*.

[16] Tei-Wei Kuo, Yung-Sheng Chao, Chin-Fu Kuo, and Cheng Chang. Real-time dwell scheduling of component-oriented phased array radars. *IEEE Trans. Computers 54(1): 47-60 (2005)*.

[17] Karthik Lakshmanan, Dionisio de Niz, and Ragunathan Rajkumar. Coordinated task scheduling, allocation and synchronization on multiprocessors. *Real-Time Systems Symposium, IEEE International*, 0:469–478, 2009.

[18] Karthik Lakshmanan, Raj Rajkumar, and John Lehoczky. Partitioned fixed-priority preemptive scheduling for multi-core processors. *Proceedings of the 21st Euromicro Conference on Real-Time Systems*, 2009.

[19] 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.

[20] Shui Oikawa and Raj Rajkumar. Portable rk: A portable resource kernel for guaranteed and enforced timing behavior. *In Proceedings of the IEEE Real-Time Technology and Applications Symposium*, June 1999.

[21] Shuichi Oikawa and Raj Rajkumar. Linux/rk: A portable resource kernel in linux. *proc. of the 19th, IEEE RTSS*, 1998.

[22] Chi-Sheng Shih, Phanindra Ganti, and Lui Sha. Schedulability and fairness for computation tasks in surveillance radar systems. *In Proceedings of the 10th RTCSA conference (Real-time and Embedded Computing Systems and Applications Conference)*, 2004.

[23] S.Lauzac, R.Melhem, and D.Mosse. Comparison of global and partitioning schemes for scheduling rate monotonic tasks on a multiprocessor. *Proceedings of the 10th Euromicro Workshop on Real-Time Systems*, pages 188–195, 1998.