DREMS-OS: An Operating System for Managed Distributed Real-time Embedded Systems

Abhishek Dubey, Gabor Karsai, Aniruddha Gokhale, William Emfinger, Pranav Kumar
ISIS, Dept of EECS, Vanderbilt University, Nashville, TN 37235, USA

***Abstract*—Distributed real-time and embedded (DRE) systems executing mixed criticality task sets are increasingly being deployed in mobile and embedded cloud computing platforms, including space applications. These DRE systems must not only operate over a range of temporal and spatial scales, but also require stringent assurances for secure interactions between the system's tasks without violating their individual timing constraints. To address these challenges, this paper describes a novel distributed operating system focusing on the scheduler design to support the mixed criticality task sets. Empirical results from experiments involving a case study of a cluster of satellites emulated in a laboratory testbed validate our claims.**

I. Introduction

The emerging realm of mobile and embedded cloud computing, which leverages the progress made in computing and communication on mobile devices and sensors necessitates a platform for running distributed, real-time, and embedded (DRE) systems. Ensembles of mobile devices are being used as a computing resource in space missions as well: satellite clusters provide a dynamic environment for deploying and managing distributed mission applications; see, *e.g.* NASA's Edison Demonstration of SmallSat Networks, TanDEM-X, PROBA-3, and Prisma from ESA, and DARPA's System F6.

As an example consider a cluster of satellites that execute software applications distributed across the satellites. One application is a safety-critical cluster flight application (CFA) that controls the satellite's flight and is required to respond to emergency 'scatter' commands. Running concurrently with the CFA, image processing applications (IPA) utilize the satellites' sensors and consume much of the CPU resources. IPAs from different vendors may have different security privileges and so may have controlled access to sensor data. Sensitive camera data must be compartmentalized and must not be shared between these IPAs, unless explicitly permitted. These applications must also be isolated from each other to prevent performance impact or fault propagation between applications due to lifecycle changes. However, the isolation should not waste CPU resources when applications are dormant because, for example, a sensor is active only in certain segments of the satellite's orbit. Other applications should be able to opportunistically use the CPU during these dormant phases.

One technique for implementing strict application isolation is temporal and spatial partitioning of processes (see [1]). Spatial separation provides a physically separated memory address space for each process. Temporal partitioning provides a periodically repeating fixed interval of CPU time that is exclusively assigned to a group of cooperating tasks. Note that strictly partitioned systems are typically configured with a static schedule; any change in the schedule requires the system to be rebooted [1].

To address these needs, we have developed an architecture called **D**istributed **RE**altime **M**anaged **S**ystem (*DREMS*) [2]. This paper focuses on the design and implementation of key components of the operating system layer in *DREMS*. It describes the design choices and algorithms used in the design of the *DREMS* OS scheduler. The scheduler supports three criticality levels: critical, application, and best effort. It supports temporal and spatial partitioning for application-level tasks. Tasks in a partition are scheduled in a work-conserving manner. Through a CPU cap mechanism, it also ensures that no task starves for the CPU. Furthermore, it allows dynamic reconfiguration of the temporal partitions. We empirically validated the design in the context of a case study: a managed DRE system running on a laboratory testbed.

The outline of this paper is as follows: Section II presents the related research; Section III describes the system model and delves into the details of the scheduler design; Section IV empirically evaluates *DREMS* OS in the context of a representative space application; and finally Section V offers concluding remarks referring to future work.

II. Related Research

Our approach has been inspired by two domains: mixed criticality systems and partitioning operating systems. A mixed criticality computing system has two or more criticality levels on a single shared hardware platform, where the distinct levels are motivated by safety and/or security concerns. For example, an avionics system can have safety-critical, mission-critical, and non-critical tasks.

In [3], Vestal argued that the criticality levels directly impact the task parameters, especially the worst-case execution time (WCET). In his framework, each task has a maximum criticality level and a non-increasing WCET for successively decreasing criticality levels. For criticality levels higher than the task maximum, the task is excluded from the analyzed set of tasks. Thus increasing criticality levels result in a more conservative verification process. He extended the response-time analysis of fixed priority scheduling to mixed criticality task sets. His results were later improved by Baruah et al. [4] where an implementation was proposed for fixed priority single processor scheduling of mixed-criticality tasks with optimal priority assignment and response-time analysis.

Partitioning operating systems have been applied to avionics (e.g., LynxOS-178 [5]), automotive (e.g., Tresos, the operating

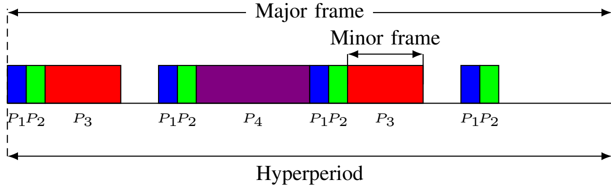


Fig. 1: A Major Frame. The four partitions (period,duration) in this frame are P_1 (2s, 0.25s), P_2 (2s, 0.25s), P_3 (4s, 1s), and P_4 (8s, 1.5s).

system defined in AUTOSAR [6]), and cross-industry domains (DECOS OS [7]). A comparison of the mentioned partitioning operating systems can be found in [8]. They provide applications shared access to critical system resources on an integrated computing platform. Applications may belong to different security domains and can have different safety-critical impact on the system. To avoid unwanted interference between the applications, reliable protection is guaranteed in both the spatial and the temporal domain that is achieved by using partitions on the system level. Spatial partitioning ensures that an application cannot access another application's code or data in memory or on disk. Temporal partitioning guarantees an application access to the critical system (CPU) resources via dedicated time intervals regardless of other applications.

Our approach combines mixed-criticality and partitioning techniques to meet the requirements of secure DRE systems. *DREMS* supports multiple levels of criticality, with tasks being assigned to a single criticality level. For security and fault isolation reasons, applications are strictly separated by means of spatial and temporal partitioning, and applications are required to use a novel secure communication method for all communications, described in *DREMS* [9].

Our work has many similarities with the resource-centric real-time kernel [10] to support real-time requirements of distributed systems hosting multiple applications. Though achieved differently, both frameworks use deployment services for the automatic deployment of distributed applications, and enforcing resource isolation among applications. However, to the best of our knowledge, [10] does not include support for process management, temporal isolation guarantees, partition management, and secure communication simultaneously.

III. DREMS Architecture

DREMS [9], [2], [11] is a distributed system architecture that consists of one or more computing nodes grouped into a cluster. It is conceptually similar to the recent Fog Computing Architecture [12]. Distributed applications, composed from cooperating processes called *actors*, provide services for the end-user. Actors specialize the notion of OS processes; they have persistent identity that allows them to be transparently migrated between nodes, and they have strict limits on resources that they can use. Each actor is constructed from one or more reusable components [13], [11] where each component is single-threaded.

A. Partitioning Support

The system guarantees spatial isolation between actors by (a) providing a separate address space for each actor; (b) enforcing that an I/O device can be accessed by only one actor at a time; and (c) facilitating temporal isolation between processes by the scheduler. Spatial isolation is implemented by the Memory Management Unit of the CPU, while temporal isolation is provided via ARINC-653 [1] style *temporal partitions*, implemented in the OS scheduler.

A temporal partition is characterized by two parameters: period and duration. The period reflects how often the tasks of the partition will be guaranteed CPU allocation. The duration governs the length of the CPU allocation window in each cycle. Given the period and duration of all temporal partitions, an execution schedule can be generated by solving a series of constraints, see [14]. A feasible solution, *e.g.* Figure 1, comprises a repeating frame of windows, where each window is assigned to a partition. These windows are called *minor frames*. The length of a window assigned to a partition is always the same as the duration of that partition. The repeating frame of minor frames, known as the *major frame*, has a length called the *hyperperiod*. The hyperperiod is the lowest common multiple of the partition periods.

B. Criticality Levels Supported by the DREMS OS Scheduler

The *DREMS* OS scheduler has the ability to manage CPU time for tasks at three different criticality levels: *Critical*, *Application* and *Best Effort*. The *Critical* tasks provide kernel level services and system management services. These tasks will be scheduled based on their priority whenever they are ready. *Application* tasks are mission specific and are isolated from each other. These tasks are constrained by temporal partitioning and can be preempted by tasks of the *Critical* level. Finally, *Best Effort* tasks are executed whenever no tasks of any higher criticality level are available.

Note that actors in an application can have different criticality levels, but all tasks associated with an actor must have the same criticality level, *i.e.* an actor cannot have both *Critical* tasks and *Application* tasks.

C. Multiple partitions

To support the different levels of criticality, we extend the *runqueue* data structure of the Linux kernel [15]. A runqueue maintains a list of tasks eligible for scheduling. In a multicore system, this structure is replicated per CPU. In a fully preemptive mode, the scheduling decision is made by evaluating which task should be executed next on a CPU when an interrupt handler exits, when a system call returns, or when the scheduler function is explicitly invoked to preempt the current process. We created one runqueue per temporal partition per CPU. Currently, the system can support 64 **Temporal partitions**, also referred to as Application partitions in the sequel. One extra runqueue is created for the critical tasks. These tasks are said to belong to the **System partition**. The Best effort tasks are managed through the Linux Completely Fair Scheduler (default) runqueue and are considered for execution as part of the System partition when no other tasks are eligible to run.

D. CPU Cap and Work Conserving Behavior

The schedulability of the *Application* level tasks is constrained by the current load coming from the *Critical* tasks and the temporal partitioning used on the *Application* level. Should the load of the *Critical* tasks exceed a threshold the system will not be able to schedule tasks on the *Application* level. A formal analysis of the response-time of the *Application* level tasks will not be provided in this paper, however, we present a description of the method we will use to address the analysis which will build on available results from [4], [16], [17].

The submitted load function $H_i(t)$ determines the maximum load submitted to a partition by the task τ_i itself after its release together with all higher priority tasks belonging to the same partition. The availability function $A_S(t)$ returns for each time instant the cumulative computation time available for the partition to execute tasks. In the original model [16] $A_S(t)$ is the availability function of a periodic server. The response-time of a task τ_i is the time when $H_i(t)$ intersects the availability function $A_S(t)$ for the first time. In our system $A_S(t)$ is decreased by the load of the available *Critical* tasks which, if unbounded, could block the application level tasks forever. This motivates us to enforce a bound on the load of the *Critical* tasks. This bound is referred to as **CPU cap**.

In *DREMS* OS, the CPU cap can be applied to tasks on the *Critical* and *Application* level to provide scheduling fairness within a partition or hyperperiod. Between criticality levels, the CPU cap provides the ability to prevent higher criticality tasks from starving lower criticality tasks of the CPU. On the *Application* level, the CPU cap can be used to bound the CPU consumption of higher priority tasks to allow the execution of lower priority tasks inside the same partition. If the CPU cap enforcement is enabled, then it is possible to set a maximum CPU time that a task can use, measured over a configurable number of major frame cycles.

The CPU cap is enforced in a work conserving manner, *i.e.*, if a task has reached its CPU cap but there are no other available tasks, the scheduler will continue scheduling the task past its ceiling. In case of *Critical* tasks, when the CPU cap is reached, the task is not marked ready for execution unless (a) there is no other ready task in the system; or (b) the CPU cap accounting is reset. This behavior ensures that the kernel tasks, such as those belonging to network communication, do not overload the system, for example in a denial-of-service attack. For the tasks on the *Application* level, the CPU cap is specified as a percentage of the total duration of the partition, the number of major frames, and the number of CPU cores available all multiplied together. When an *Application* task reaches the CPU cap, it is not eligible to be scheduled again unless the following is true: either (a) there are no *Critical* tasks to schedule and there are no other ready tasks in the partition; or (b) the CPU cap accounting has been reset.

E. Dynamic Major Frame Configuration

During the configuration process that can be repeated at any time without rebooting the node, the kernel receives the major frame structure that contains a list of minor frames and it also contains the length of the hyperperiod, partition periodicity, and duration. Note that major frame reconfiguration can only be performed by an actor with suitable capabilities. More details on the *DREMS* capability model can be found in [9].

Before the frames are set up, the process configuring the frame has to ensure that the following three constraints are met: (C0) The hyperperiod must be the least common multiple of partition periods; (C1) The offset between the major frame start and the first minor frame of a partition must be less than or equal to the partition period: $(\forall p \in \mathbb{P})(O_1^p \leq \phi(p))$; (C2) Time between any two executions should be equal to the partition period: $(\forall p \in \mathbb{P})(k \in [1, N(p)-1])(O_{k+1}^p = O_k^p + \phi(p))$, where $\mathbb{P}$ is the set of all partitions, $N(p)$ is the number of partitions, $\phi(p)$ is the period of partition p and $\Delta(p)$ is the duration of the partition p. O_i^p is the offset of ith minor frame for partition p from the start of the major frame, H is the hyperperiod.

The kernel checks two additional constraints: (1) All minor frames finish before the end of the hyperperiod: $(\forall i)(O_i.start + O_i.duration \leq H)$ and (2) minor frames cannot overlap, i.e. given a sorted minor frame list (based on their offsets): $(\forall i < N(O))(O_i.start + O_i.duration \leq O_{i+1})$, where $N(O)$ is the number of minor frames. Note that the minor frames need not be contiguous, as the update procedure fills in any gaps automatically.

If the constraints are satisfied, then the task is moved to the first core, *CPU0* if it is not already on *CPU0*. This is done because the global tick (explained in next subsection) used for implementing the major frame schedule is also executed on *CPU0*. By moving the task to *CPU0* and disabling interrupts, the scheduler ensures that the current frame is not changed while the major frame is being updated. At this point the task also obtains a spin lock to ensure that no other task can update the major frame at the same time. In this procedure the scheduler state is also set to `APP_INACTIVE` (see Table I), to stop the scheduling of all application tasks across other cores. The main scheduling loop reads the scheduler state before scheduling application tasks. A scenario showing dynamic reconfiguration can be seen in Figure 2.

TABLE I: The states of the DREMS Scheduler

APP_INACTIVE	Tasks in temporal partitions are not run
APP_ACTIVE	Inverse of APP_INACTIVE

It is also possible to set the global tick (that counts the hyperperiods) to be started with an offset. This delay can be used to synchronize the start of the hyperperiods across nodes of the cluster. This is necessary to ensure that all nodes schedule related temporal partitions at the same time. This ensures that for an application that is distributed across multiple nodes, its *Application* level tasks run at approximately the same time on all the nodes which enables low latency communication between dependent tasks across the node level.

F. Main Scheduling Loop

A periodic tick running at 250 Hz[1] is used to ensure that a scheduling decision is triggered at least every 4 ms. This tick

[1]The kernel tick value is also called 'jiffy' and can be set to a different value when the kernel image is compiled

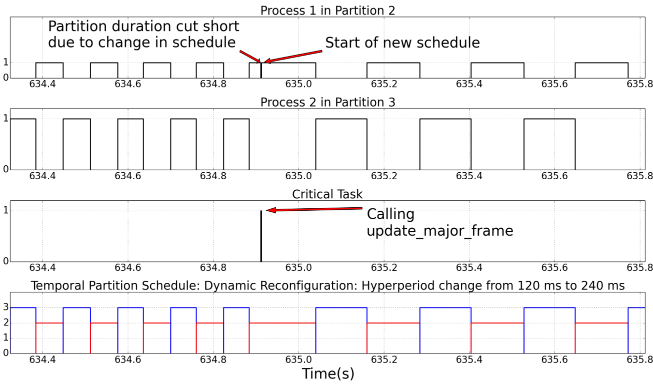


Fig. 2: Two single-threaded processes run in separate partitions with a duration of $60ms$ each. The schedule is dynamically reconfigured so that each partition duration is doubled. A *Critical* task is responsible for calling the update_major_frame system call. Duration of the active partition is cut short at the point when update_major_frame function is called.

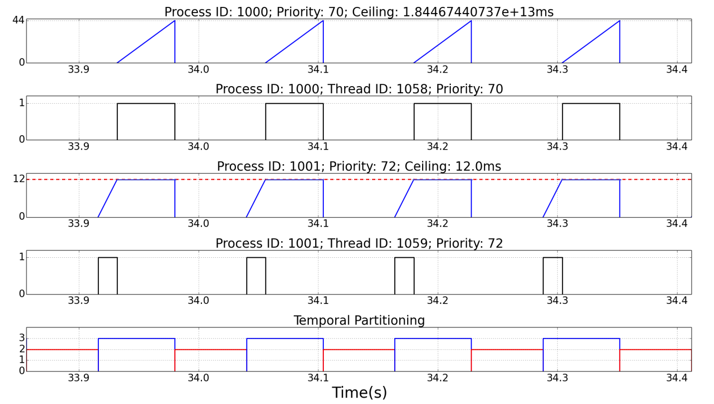


Fig. 3: Single Threaded processes 1000 and 1001 share a partition with a duration of $60ms$. Process 1000 has 100% CPU cap and priority 70; process 1001 has 20% CPU cap, and higher priority 72. Since process 1001 has a CPU cap less than 100%, a ceiling is calculated for this process: 20% of $60ms$ = $12ms$. The average jitter was calculated to be 2.136 ms with a maximum jitter of 4.0001 ms.

runs with the base clock of *CPU0* and executes a procedure called $GlobalTick$ in the interrupt context only on *CPU0*. This procedure enforces the partition scheduling and updates the current minor frame and hyperperiod start time (`HP_start`). The partition schedule is determined by a circular linked list of minor frames which comprise the major frame. Each entry in this list contains that partition's duration, so the scheduler can easily calculate when to switch to the next minor frame.

After the global tick handles the partition switching, the function to get the next runnable task is invoked. This function combines the *mixed criticality* scheduling with the *temporal partition* scheduling. For mixed criticality scheduling, the *Critical* system tasks should preempt the *Application* tasks, which themselves should preempt the *Best Effort* tasks. This policy is implemented by *Pick_Next_Task* subroutine, which is called first for the system partition. Only if there are no runnable *Critical* system tasks and the scheduler state is not inactive, i.e. the application partitions are allowed to run[2], will *Pick_Next_Task* be called for the *Application* tasks. Thus, the scheduler does not schedule any *Application* tasks during a major frame reconfiguration. Similarly *Pick_Next_Task* will only be called for the *Best Effort* tasks if there are both no runnable *Critical* tasks and no runnable *Application* tasks.

G. *Pick_Next_Task and CPU Cap*

The *Pick_Next_Task* function returns either the highest priority task from the current temporal partition (or the system partition, as application) or an empty list of there are no runnable tasks. If CPU cap is disabled, the *Pick_Next_Task* algorithm returns the first task from the specified runqueue. For the best effort class, the default algorithm for the Completely Fair Scheduler policy in the Linux Kernel [18] is used.

If the CPU cap is enabled, the *Pick_Next_Task* algorithm iterates through the task list at the highest priority index of the runqueue, because unlike the Linux scheduler, the tasks may have had their disabled bit set by the scheduler if it had enforced their CPU cap. If the algorithm finds a disabled task in the task list, it checks to see when it was disabled; if the task was disabled in the previous CPU cap window, it reenables the task and sets it as the $next_task$. If, however, the task was disabled in the current CPU cap window, the algorithm continues iterating through the task list until it finds a task which is enabled. If the algorithm finds no enabled task, it returns the first task from the list if the current runqueue belongs to an application partition.

This iteration through the task list when CPU cap enforcement is enabled increases the complexity of the scheduling algorithm to $O(n)$, where n is the number of tasks in that temporal partition, compared to the Linux scheduler's complexity of $O(1)$. Note that this complexity is incurred when CPU cap enforcement is enabled and there is at least one actor that has partial CPU cap (less than 100%). In the worst case, if all actors are given a partial CPU cap, the scheduler performance may degrade necessitating more efficient data structures.

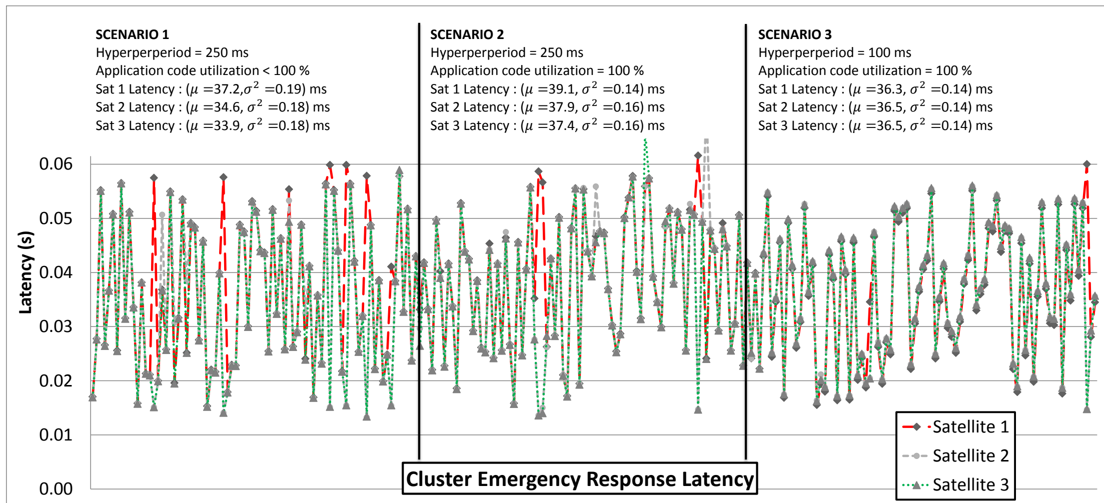
To complete the enforcement of the CPU cap, the scheduler updates the statistics tracked about the task and then updates the disabled bit of the task accordingly. Figure 3, shows the above mentioned scheduler decisions when CPU cap is placed on processes that share a temporal partition. To facilitate analysis, the scheduler uses a logging framework that updates a log every time a context switch happens. Figure 3 clearly shows the lower priority actor executing after the higher priority actor has reached its CPU cap.

IV. Experiment: A 3-node Satellite Cluster

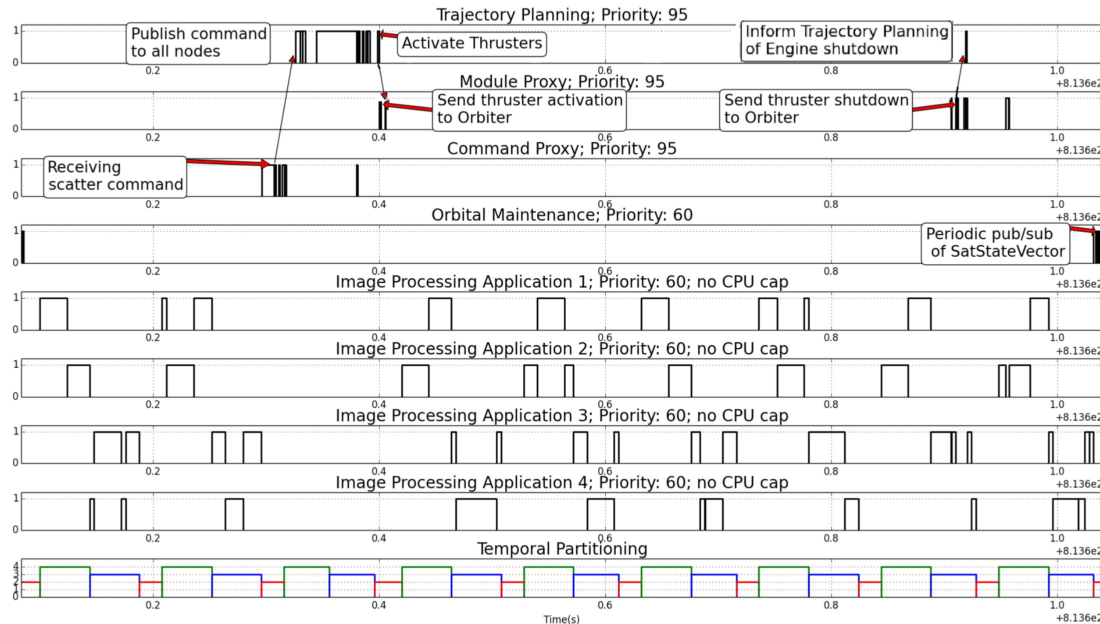
To demonstrate the DREMS platform, a multi-computing node experiment was created on a cluster of fanless computing nodes with a 1.6 GHz Intel Atom N270 processor and 1 GB of RAM each. On these nodes, a cluster of three satellites was emulated and each satellite ran the example applications described in Section I. Because the performance of the cluster flight control application is of interest, we explain the interactions between its actors below.

The mission-critical cluster flight application (CFA) (Figure 5) consists of four actors: *OrbitalMaintenance*, *Trajectory-Planning*, *CommandProxy*, and *ModuleProxy*. *ModuleProxy* connects to the Orbiter space flight simulator (http://orbit.

[2]The OS provides support for pausing all application partitions and ensuring that only system partition is executed



(a) This is the time between reception of the *scatter* command by satellite 1 and the activation of the thrusters on each satellite, corresponding to interactions *CommandProxy* to *ModuleProxy*. The three regions of the plot indicate the three scenarios: (1) image processing application has limited use of its partitions and has a hyperperiod of 250 ms, (2) image processing application has full use of its partitions and has a hyperperiod of 250 ms, and (3) image processing application has full use of its partitions and has a hyperperiod of 100 ms. The averages and variances for the satellites' latencies are shown for each of the three scenarios.



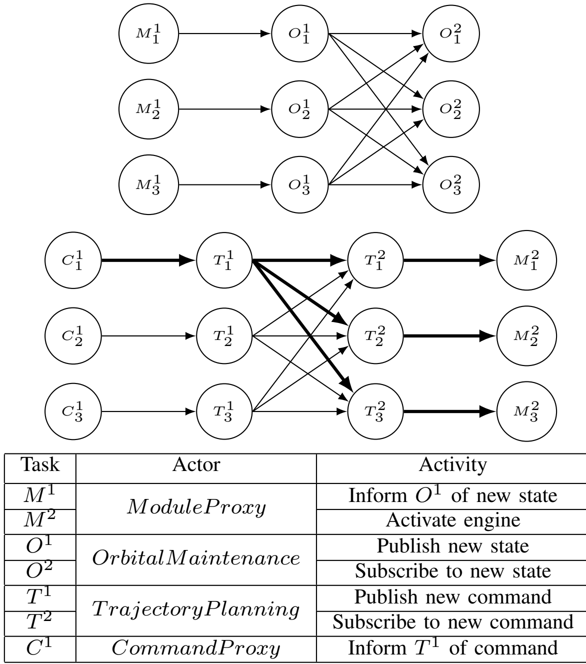
(b) The engine activation following reception of a *scatter* command is annotated for the relevant actors for scenario 2 shown above. The *scatter* command causes the *TrajectoryPlanning* to request *ModuleProxy* to activate the thrusters for 500 ms. Notice that the image processing does not run while the mission-critical tasks are executing - without halting the partition scheduling. Also note that the context switching during the execution of the critical tasks is the execution of the secure transport kernel thread. Only the application tasks are shown in the log; the kernel threads and other background processes are left out for clarity.

Fig. 4: *DREMS* Mixed Criticality Demo

medphys.ucl.ac.uk/) that simulates the satellite hardware and orbital mechanics for the three satellites in low Earth orbit. *CommandProxy* receives commands from the ground network. *OrbitalMaintenance* keeps track of every satellite's position and updates the cluster with its current position. This is done by a group publish subcribe interaction between all *OrbitalMaintenance* actors across all nodes.

Additionally, four image processing application (IPA) actors (one actor per application instance) are deployed as application tasks. The IPA design allows the percentage of CPU cycles consumed by them to be configurable. The four IPAs are assigned to two partitions, such that each partition contains two IPA actors. A third, shorter, partition runs the *OrbitalMaintenance* actor; since it is a periodic task, it updates the satellite state every second and is not critical in an emergency.

Figures 4a and 4b show the results from three different scenarios: 1) hyperperiod of 250 ms, with IPA consuming less than 50 percent CPU. 2) hyperperiod of 250 ms, with IPA consuming 100 percent CPU and 3) hyperperiod of 100 ms, with IPA consuming 100 percent CPU. As shown in figure 4a,



Task	Actor	Activity
M^1	$ModuleProxy$	Inform O^1 of new state
M^2		Activate engine
O^1	$OrbitalMaintenance$	Publish new state
O^2		Subscribe to new state
T^1	$TrajectoryPlanning$	Publish new command
T^2		Subscribe to new command
C^1	$CommandProxy$	Inform T^1 of command

Fig. 5: *DREMS* tasks : *ModuleProxy* tasks control thruster activation in Orbiter and state vector retrieval from Orbiter. *OrbitalMantenance* tasks track the cluster satellites' state vectors and disseminate them. *TrajectoryPlanning* tasks control the response to commands and satellite thruster activation. *CommandProxy* tasks inform the satellite of a command from the ground network. For these tasks, the subscript represents the node ID on which the task is deployed. The total latency of the interaction $C_1^1 \rightarrow M_N^2$ represents the total emergency response latency between receiving the *scatter* command and activating the thrusters. This interaction pathway is bolded.

the emergency response latency over the three nodes was quite low with very little variance, and did not correlate with either the image application's CPU utilization or the application's partition schedule. Since we show that the emergency response has very low latency with little variance between different application loads on the system, we provide a stable platform for deterministic and reliable emergency response. As such, the satellite cluster running the *DREMS* infrastructure is able to quickly respond to emergency situations despite high application CPU load and without altering the partition scheduling. Figure 4b demonstrates the proper preemption of the image processing tasks by the critical CFA tasks for scenario 2.

V. Conclusions and Future Work

This paper propounds the notion of managed distributed real-time and embedded (DRE) systems that are deployed in mobile computing environments. To that end, we described the design and implementation of a distributed operating system called *DREMS* OS focusing on a key mechanism: the scheduler. We have verified the behavioral properties of the OS scheduler, focusing on temporal and spatial process isolation, safe operation with mixed criticality, precise control of process CPU utilization and dynamic partition schedule reconfiguration. We have also analyzed the scheduler properties of a distributed application built entirely using this platform and hosted on an emulated cluster of satellites.

We are extending this operating system implement a to build an open-source FACE[tm] Operating System Segment [19], called COSMOS (**C**ommon **O**perating **S**ystem for **M**odular **O**pen **S**ystems). To the best of our knowledge this is the first open source implementation of its kind that provides both ARINC-653 and POSIX partitions.

Acknowledgments: This work was supported by the DARPA under contract NNA11AC08C. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not reflect the views of DARPA.

References

[1] *Document No. 653: Avionics Application Software Standard Inteface (Draft 15)*, ARINC Incorporated, Annapolis, Maryland, USA, Jan. 1997.

[2] G. Karsai, D. Balasubramanian, A. Dubey, and W. R. Otte, "Distributed and managed: Research challenges and opportunities of the next generation cyber-physical systems," in *2014 IEEE 17th International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing*, June 2014, pp. 1–8.

[3] S. Vestal, "Preemptive Scheduling of Multi-Criticality Systems with Varying Degrees of Execution Time Assurance," in *Proc. of 28th IEEE Real-Time Systems Symposium*, Tucson, AZ, Dec. 2007, pp. 239–243.

[4] S. Baruah, A. Burns, and R. Davis, "Response-Time Analysis for Mixed-Criticality Systems," in *Proceedings of the 2011 32nd IEEE Real-Time Systems Symposium*, Vienna, Austria, Nov. 2011, pp. 34–43.

[5] LynuxWorks, "RTOS for Software Certification: LynxOS-178." [Online]. Available: http://www.lynuxworks.com/rtos/rtos-178.php

[6] Autosar GbR, "AUTomotive Open System ARchitecture," http://www.autosar.org/. [Online]. Available: http://www.autosar.org/

[7] R. Obermaisser, P. Peti, B. Huber, and C. E. Salloum, "DECOS: An Integrated Time-Triggered Architecture," *e&i journal (Journal of the Austrian Professional Institution for Electrical and Information Engineering)*, vol. 123, no. 3, pp. 83–95, Mar. 2006.

[8] B. Leiner, M. Schlager, R. Obermaisser, and B. Huber, "A Comparison of Partitioning Operating Systems for Integrated Systems," in *Computer Safety, Reliability and Security*, ser. Lecture Notes in Computer Science. Springer, 2007, vol. 4680/2007, pp. 342–355.

[9] A. Dubey, W. Emfinger, A. Gokhale, G. Karsai, W. Otte, J. Parsons, C. Szabo, A. Coglio, E. Smith, and P. Bose, "A Software Platform for Fractionated Spacecraft," in *Proceedings of the IEEE Aerospace Conference, 2012*. Big Sky, MT, USA: IEEE, Mar. 2012, pp. 1–20.

[10] K. Lakshmanan and R. Rajkumar, "Distributed Resource Kernels: OS Suppport for End-To-End Resource Isolation ," in *Proceedings of the 2008 IEEE Real-Time and Embedded Technology and Applications Symposium*, St. Louis, MO, Apr. 2008, pp. 195–204.

[11] S. Eisele, I. Madari, A. Dubey, and G. Karsai, "Riaps:resilient information architecture platform for decentralized smart systems," in *20th IEEE International Symposium On Real-Time Computing*, IEEE. Toronto, Canada: IEEE, 05/2017 2017.

[12] L. M. Vaquero and L. Rodero-Merino, "Finding your way in the fog: Towards a comprehensive definition of fog computing," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 5, pp. 27–32, 2014.

[13] W. R. Otte, A. Dubey, S. Pradhan, P. Patil, A. Gokhale, G. Karsai, and J. Willemsen, "F6COM: A Component Model for Resource-Constrained and Dynamic Space-Based Computing Environment," in *Proceedings of the 16th IEEE International Symposium on Object-oriented Real-time Distributed Computing (ISORC '13)*, Paderborn, Germany, Jun. 2013.

[14] A. Dubey, G. Karsai, and N. Mahadevan, "A Component Model for Hard Real-time Systems: CCM with ARINC-653," *Software: Practice and Experience*, vol. 41, no. 12, pp. 1517–1550, 2011.

[15] A. Garg, "Real-time linux kernel scheduler," *Linux Journal*, vol. 2009, no. 184, p. 2, 2009.

[16] L. Almeida and P. Pedreiras, "Scheduling within Temporal Partitions: Response-time Analysis and Server Design," in *Proc. of the 4th ACM Int Conf on Embedded Software*, Pisa, Italy, Sep. 2004, pp. 95–103.

[17] G. Lipari and E. Bini, "A Methodology for Designing Hierarchical Scheduling Systems," *Journal of Embedded Computing*, vol. 1, no. 2, pp. 257–269, Apr. 2005.

[18] W. Mauerer, *Professional Linux Kernel Architecture*, ser. Wrox professional guides. Wiley, 2008. [Online]. Available: http://books.google.com/books?id=4eCr9dr0uaYC

[19] OpenGroup. [Online]. Available: http://www.opengroup.org/face