Advances in Military Technology
Vol. 20, No. 1, 2025, pp. 107-122
ISSN 1802-2308, eISSN 2533-4123
DOI 10.3849/aimt.01943

Scheduling Techniques for Real-Time Systems for Defense Applications on Multicore Processors

P. Balakrishnan[1*], M. Rajesh[2] and R. Rajesh[3]

[1] *NPOL, DRDO, Cochin University of Science & Technologies, Kochi, Kerala, India*
[2] *National Institute of Electronics & Information Technology, Calicut, Kerala, India*
[3] *NPOL, DRDO, Kochi, Kerala, India*

The manuscript was received on 19 November 2024 and was accepted after revision for publication as a case study on 21 April 2025.

Abstract:

The real-time scheduling of tasks of any application software is an important requirement in defense applications. This paper explains partitioning and scheduling techniques in the real-time implementation of any defense system on embedded multicore processors with Linux as the operating system. A typical sonar system is considered as the reference to elaborate partitioning and scheduling techniques. This can be extended to any defense applications like radars, missiles, etc. This paper describes the Partitioned Earliest Deadline First (PART-EDF) scheduling algorithm integrated into LITMUS^{RT} real-time Linux kernel to enhance task schedulability. The proposed system partitions computationally intensive, periodic operations like filtering, beam forming, target classification, etc., and sporadic operations like input and control functions onto different clusters on the multi-core processors.

Keywords:

EDF, sonar, WCET, LITMUS^{RT}, ADC, beamforming

1 Introduction

The paper discusses the partitioned adaptive Earliest Deadline First (PART-EDF) scheduling with adaptive EDF scheduling policy for the realization of real-time signal processing tasks for applications like sonars, radars, etc., on multicore processors. With the availability of multicore processors having huge computational power, the signal processing functions of complex applications like sonars, radars, etc., are implemented on multicore processors. This paper describes the partitioning and scheduling techniques regarding the implementation of a typical sonar signal pro-

[*] *Corresponding author: Naval Physical and Oceanographic Laboratory, Defence Research and Development Organization, Kochi, Kerala, 682021, India. Phone: +91-9447 41 42 45, Email: bala.hema.bala@gmail.com. ORCID 0009-0003-0188-8646.*

cessing on any standard multicore. The partitioning and scheduling techniques explained can be extended to other applications like radar, missiles, etc.

Sonars are considered the eyes and ears of any underwater platform, especially submarines and hence, critical equipment. The submarines typically have integrated sonar suites consisting of multiple sonars to achieve better detection performance in different scenarios. In the past, nearly two decades back, signal processing for sonar systems was realized on digital signal processors from manufacturers like Analog Devices, Texas Instruments etc. Now, with the advancements in VLSI technology, multicore processors from various sources, especially Intel, are available which can meet the real time requirements of the sonar systems, radars etc. Embedded Multicore processors with multiple cores, with hyper-threading capability, and with Linux as the operating system are currently used in defense applications. Linux being an open source OS has a lot of advantages in the realization of systems. But to meet the real time requirements, the application developer, in the absence of good scheduling techniques, has to go through several iterations during software development so as to achieve the critical deadlines of the various tasks of the application software.

This paper explains the partitioning schemes and the scheduling techniques for realizing a typical complex real time signal processing system of a sonar system on any multicore processor with Linux as OS. The task partitioning techniques and the modification/improvement in the existing EDF scheduler in Linux is also explained in detail. These techniques lead to efficient utilization of the multicore processors, miniaturization of systems leading to enhanced reliability and low power consumption.

2 Literature Review

Many challenges have been addressed regarding the dynamic scheduling of real-time applications over multicore platforms, with partitioned EDF [1-6] as one of the main techniques enabling predictable execution of tasks. Many authors, such as Saranya and Hansdah, in their research work done in the year 2015 have shown that Dynamic partitioning provides the right level of tasks to right cores as per their demand, hence optimizing the available mode of cores [1]. Subsequent work by Han, Cai and Zhu in 2018 discusses the resource-aware partitioning for heterogeneous systems where the cores are not homogeneous in their nature [2]. Akram et al. (2019) discuss an inter-task affinity-aware allocation algorithm that partitions tasks based on several factors to reduce the worst-case execution time [3]. In Xu et al. (2019), a holistic cache and memory bandwidth algorithm for multi-core real-time systems is discussed [4]. Ma et al. (2021) elaborates the partitioning of shared resource in real-time systems [5]. LITMUS^{RT} was developed as a test-bed for evaluating multiprocessor scheduling solutions [6, 7]; Calandrino et al. (2006) presented LITMUS^{RT} as a benchmark for evaluating real-time schedulers. Stevanato et al. (2021) gives a detailed evaluation of adaptive partitioning of tasks in Linux [8]. These scheduling techniques are applicable to complex real-time systems used for defense applications like sonar and radar. Several research works have taken place in scheduling techniques for real-time systems on multi-core processors [9, 10]. A sonar array processing implementation based on multi-core processors is elaborated by Li et al. (2012) [11]. Stepping further, the new research is more oriented in energy-efficient and mixed critical systems. Digalwar, Gahukar and Mohan have developed energy efficient scheduling on multi-core processors [12] in their research work. Heterogeneous multi-core processors with the same instruction set architecture integrate complex high-performance big cores with power

efficient small cores on the same chip. [13] proposes a heterogeneous fairness-aware energy efficient framework to provide energy efficient scheduling.

3 Typical Sonar Signal Processing System

The paper discusses the partitioning and scheduling techniques [14, 15] used for realizing a typical sonar signal processing system. Partitioning techniques are arrived at the basis of the analysis conducted on the systems developed on multicore processors [16] and design requirements. The context diagram of any sonar signal processor is given in Fig. 1 below.

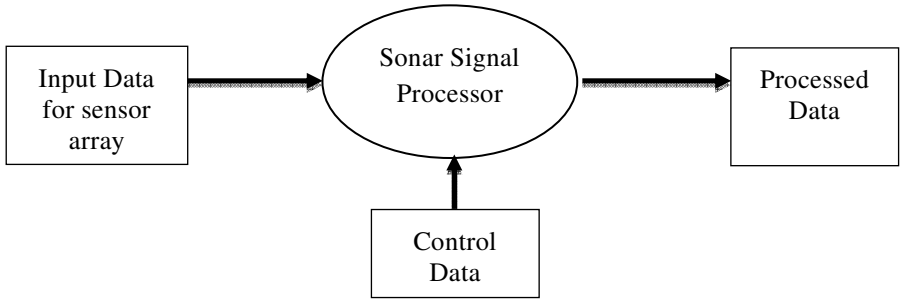

Fig. 1 Context diagram of a typical Passive Sonar Signal Processor

The function of a typical passive sonar system includes detection, tracking, spectrum processing, classification etc. of multiple targets. Sonar signal processor receives digitized data from the data acquisition system. Data acquisition of a typical sonar system having an array of 5 000 sensors, each sensor sampled at 50 kHz using 24-bit sigma delta ADC outputs data at the rate of 1.2 lakhs bits in every 20 microseconds. All the signal processing blocks like detection, beamforming, tracking, spectrum processing etc. shown in Fig. 2 have to be realized in software. The processing on each sample has to be strictly completed in every sample period. Normal block data processing is done and hence the time available for processing will be the data collection time for a block of data.

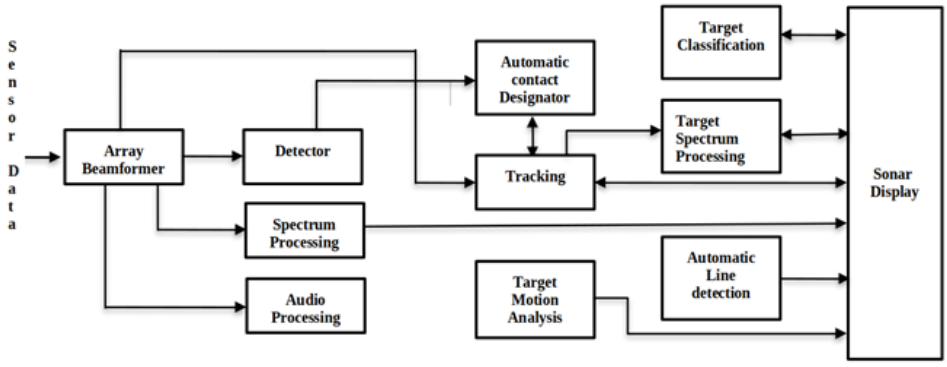

Fig. 2 Typical Passive Sonar Signal Processing System

4 Semi-Partitioned Scheduling

In any sonar signal processing system, three categories of data viz. input data, control data and processed data is handled by the signal processor as shown in Fig. 1. The input data from data acquisition is handled by the input tasks, the processed data is handled by output tasks and the control of the system as selected by the operator is handled by the control task of the application software. As shown in block diagram in Fig. 2, the input to each block is from the previous blocks, and hence, various processing tasks are to be in sync with the input data received from the data acquisition system. However, the control tasks are asynchronous; the computational requirement is to collect the control parameters and update the pre-defined variables that will be used by the processing tasks.

For realizing every block in the diagram given in Fig. 2, multiple cores will be required. The major functions involved in passive detection are Beamformer, Detector, Panoramic Spectrum processing, Tracking, spectrum processing of detected targets, Auto Contact Designation System (ACDS), Auto Line Detection System (ALDS) and Audio Processor. All these processing tasks include computationally intensive functions like Fast Fourier transforms, matrix inversions, filters etc. that require huge computational power proportional to the number of sensors of the array. The different tasks in any sonar system can be categorized as input reception, controller, signal processing tasks and processed data outputting task. Each task has to completed within the interval of repetition of the task. Extreme care has to be taken to collect all the data from the data acquisition without any miss. The typical data rate of the input data will be more than 1.2 Gbps for a sensor array having 1 500 sensors sampled at 32 kHz. The input data will be received in small chunks over Ethernet and the data reception process is initiated by interrupts.

For optimal scheduling, the total CPU utilization U on each core must satisfy:

$$U = \sum_i \frac{C_i}{T_i} \tag{1}$$

where

C_i: the worst-case execution time of task i,

T_i: the task period or inter-arrival time.

In a typical system, input tasks are periodic and occur at a higher frequency than processing tasks, as input data is received in small packets. Control tasks are sporadic in nature and to be addressed immediately. In such situations to avoid misses, input tasks & sporadic tasks like control tasks are given the highest priority. Hence these tasks will pre-empt the running processing tasks on the cores and will lead to deadline misses. In such cases CPU utilization will exceed as given in equation (2)

$$U = \frac{C_i + \delta + n \cdot C_\Delta}{T_i} \tag{2}$$

where

C_i – the execution time of any task i,

δ – the overhead due to pre-emptions

C_Δ – the high-priority task execution time,

n – the number of times when high priority task was executed during T_i,

T_i – the normal task period,

U – the CPU utilization of the core running the task T_i.

Hence, a cluster-based partitioning scheme is suggested for the implementation of various tasks of the sonar signal processing to ensure that input tasks, output tasks and control tasks are handled by one or two clusters as required, depending on the input data bandwidth. One cluster is formed for the processing tasks as shown in Fig. 3. In the modified EDF scheduler suggested, separate ready queues are maintained for each cluster to queue the tasks assigned to each cluster. In this paper, the EDF scheduler is modified with two ready queues for two cluster configurations.

4.1 Scheme for Partitioning the Tasks of a Typical Sonar System

The context diagram of any typical sonar system is explained in Fig 1. The tasks to be executed in any sonar system can be categorized as input tasks, output tasks, control tasks and processing tasks. Input task will collect the data from data acquisition system for processing and output task will forward the processed data to the display. The input and output processes are to be executed before the arrival of the next block of data to maintain real time nature. A partitioned scheduling [17, 18] scheme is suggested to avoid misses of the input data and to ensure proper CPU utilization. Cores are partitioned into two or three clusters to handle different categories of tasks. In an N core processor with Linux as the operating system, one or two processors are grouped into one or two clusters and earmarked for input process, output process and the control data process, while the remaining cores are grouped into another cluster for processing tasks. One core is assigned to run Linux processes and remaining cores are isolated as explained in Fig. 3 to run the application program.

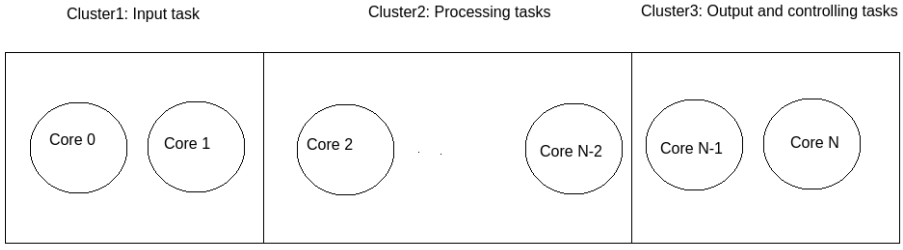

Fig. 3 Partitioned Scheduling a typical Passive Sonar Signal Processing use case

5 Scheduling Techniques

Linux is the preferred operating system for multicore processors, but the real time requirement and efficient CPU utilization cannot be achieved by the available scheduling policies. The modification of the EDF scheduler with two ready queues for a semi partitioned approach to achieve maximum CPU utilization is explained in the subsequent sections.

The Linux scheduler schedules the tasks of the application program onto the free cores of the multicore processor. Each scheduler available in Linux has advantages and disadvantages. The real time policies available in Linux are SCHED_FIFO, SCHED _RR and SCHED_DEADLINE. The policy available for meeting the critical timelines of any real time system is SCHED_DEADLINE (Earliest Deadline First EDF), but it is not used due to its limitations. For using SCHED_DEADLINE, the

application developer has to feed the following parameters: worst case execution time (Ci), deadline (D_i) and period (T_i) in the application program. C_i depends on various parameters, such as processor frequency, processor architecture and the interference of co-running tasks with other cores of the multicore processor. In Linux kernel version 3.14 onwards SCHED_DEADLINE policy is introduced. The priorities of the tasks are assigned based on the deadlines of the tasks; the task having the smallest deadline will be on the top of the ready queue.

5.1 Scheduling Techniques in Linux

The major difficulty in using SCHED_DEADLINE policy is that the application developer has to estimate the WCET and feed it into the application program. For this a normal practice by the application developer is to execute all the task/thread in any core of the multicore processor with other cores in idle condition and this value of WCET is used in the application program. But the WCET estimate will vary when that task is running with other tasks on other cores of the multicore processor. The allocation of tasks to any core is done based on the execution time and hence this approach will affect the CPU utilization; moreover, it may exceed CPU utilization "1" in certain situations leading to deadline misses. These deadline misses will be random in nature and will be extremely difficult to track while the application program is running on any multicore processor. To overcome this problem, a modified EDF scheduler has been developed that dynamically estimates the worst-case execution time in real time and tasks are assigned to the free core after ensuring that the utilization is below 1.

6 System Model

Consider a multicore processor system with M identical cores ($P_1,\ldots, P_M$) and an application program consisting of a task set with N tasks/threads, i.e., ($\tau_1,\ldots, \tau_N$). Let the execution time be denoted by ($C_1,\ldots, C_N$) and the period be denoted by ($T_1,\ldots, T_N$). In any real-time system, almost all the tasks except the control tasks will be periodic. All the periodic tasks will be repeated in certain intervals, and the pattern of repetition will be the same in any major cycle. The period of the major cycle is the LCM of periods of all the tasks of the task set. This is verified by simulating a typical task set for multiple major cycles.

6.1 CPU Utilization in a Major Cycle

In a major cycle M_i, each task i will be repeating every m_i time

Major cycle M is the LCM of the periods of the tasks in the task set. The utilization of each core in a major cycle is given as

$$U_{\mathrm{M}i} = m_1 \frac{C_1}{T_1} + m_2 \frac{C_2}{T_2} + \cdots \tag{3}$$

where CPU utilization $U_{\mathrm{M}i} < 1$ for efficient core utilization.

7 Modified EDF Scheduler

The modified EDF scheduler is an improvement over the existing EDF policy in Linux in which the worst-case execution time (WCET) is estimated online as the application

program is running. When the application program is executed for the first time, the initial value of the execution time is fed by the application developer.

7.1 Computation of WCET in Real-Time

A scheme to compute the WCET when the application program is running is elaborated on in this section. In any multicore processor, the execution time of a task will depend on the interference of the co-running tasks. In any typical real-time sonar system, the application program will consist of a large number of threads/processes with different execution times, release times, deadlines and periodicity. In any major cycle each task '*i*' will be executed m_i times depending on the period of that task. The worst-case execution time of any task is the maximum of the execution time of the task in any major cycle. The scheme is elaborated on in section 6.3, and the corresponding flowchart is given in Fig. 4.

7.2 CPU Isolation

In Linux, it is possible to pin a specific core to a certain process to reduce variation in its reaction time and protect it from interruptions of the processes that are less important. By pinning CPUs, the other tasks, non-CPU intensive, kernel threads, and interrupt handling cannot occur on the isolated cores, making it suitable for real-time applications. This can be done through patched kernel boot parameters such as isolcpus, run time control tools such as taskset and cset, and the duty to handle the interrupt such as irq balance. CPU isolation benefits system determinism by decreasing context switches and by guaranteeing the timely execution of important loads with minimal disturbances.

7.3 Scheduling of Tasks in the Modified Scheduler

Scheduling of the tasks [19, 20] to any free core is based on CPU utilization of any core in a major cycle. Whenever a task is going to be assigned to any core, the summation of WCET, i.e., ΣCi of all tasks assigned to that core in the major cycle is calculated. The ratio of ΣCi of all tasks and the major cycle M_i is taken and if the ratio is less than 1, then only the task will be assigned to that core, i.e., $U_{Mi} = (m_1C_1/T_1 + m_2C_2/T_2 \ \ldots)/Mi$ should be less than 1. This ensures that the core utilization is maintained below 1.

7.4 LITMUS^{RT} Implementation Framework

LITMUS^{RT} [6, 15] is designed to run in a dual-kernel architecture, enabling real-time capabilities while preserving standard Linux functionality. This approach differs from other approaches - the real-time control of tasks differs from generic Linux operations. The real-time kernel performs the dynamic scheduling of tasks with hard timing constraints using specific real-time scheduling add-ons. However, non-real-time tasks can still be executed in the standard environment based on the Linux kernel. Such segregation effectively eliminates interactions between real-time and non-real-time processes, thus adhering to core ideas of system efficiency and reliability. According to the present work, the utilization of a dual-kernel design enables the customization of scheduling policies to be more manageable since researchers do not necessarily have to heavily tinker with the regular Linux kernel to analyze real-time behavior. This is

realized through APIs and hooks for issuing task migration, scheduling decisions and timing assessment within the Real-Time Kernel LITMUS^{RT}.

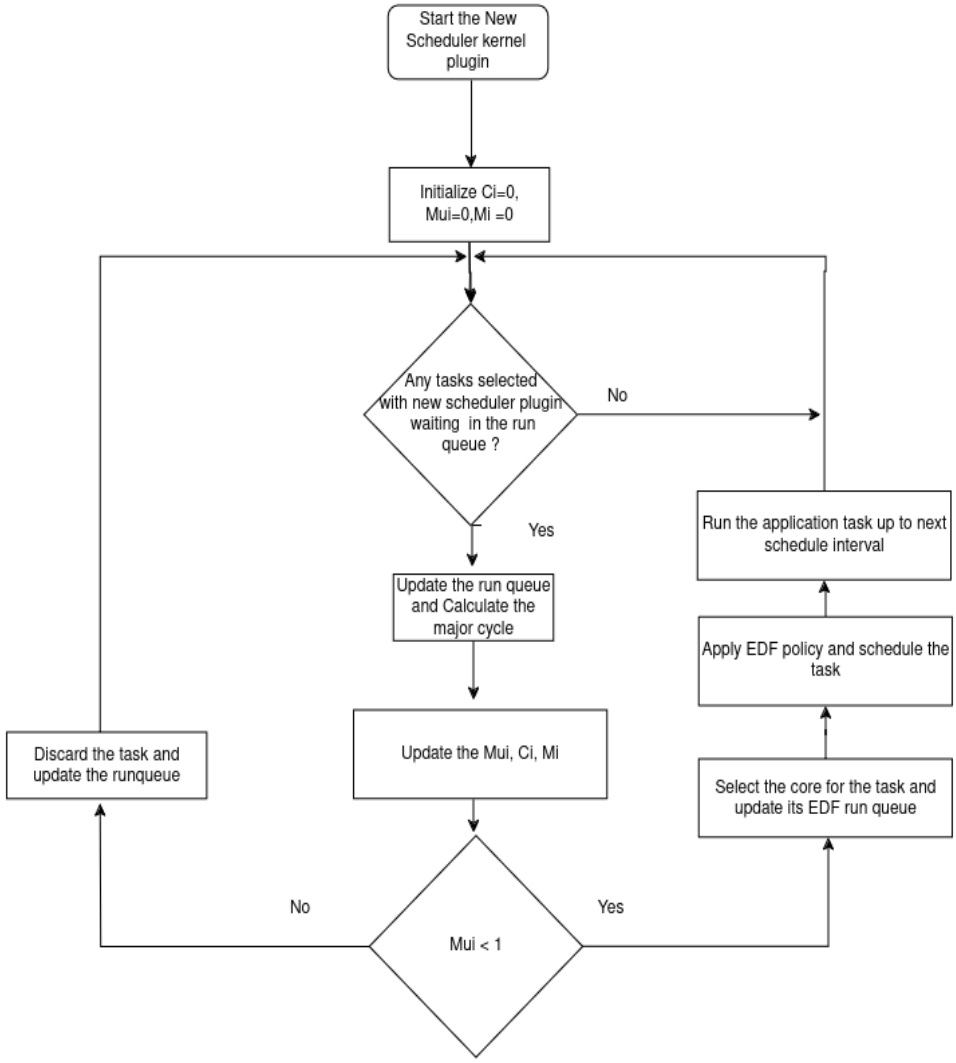

Fig. 4 Flow diagram

7.5 LITMUS^{RT} Environment

LITMUS^{RT} is essentially a real-time patch for the Linux kernel built as a research prototype for experimenting with real-time scheduling algorithms. It is widely used for the analysis of working behavior and performance of real-time systems under different scheduling policies. Some of the benefits of LITMUS^{RT} include accurate control over the task and flexibility to use the modular scheduler plugins. Its approach involves a dual-kernel structure, through which new schedulers can be designed and easily incorporated while other normal Linux operations may continue to run in the background. The concept provides means and substrates, such as RT-launch for running

real-time tasks, and APIs for their synchronization, timing and latency measurement hooks. Scheduler plugins are developed by writing specific scheduling code as per the LITMUS^{RT} API, kernel build consisting of a new plugin, and application deployment through the RT-launch tool to assess performance and timing promises under real-time conditions.

8 Partitioned EDF (PART-EDF) – A New Scheduler Plugin

The newly developed PART-EDF scheduler plugin will group the processors into ***n*** clusters and ***n***-ready queues. The plugin software takes care of the clusters. PART-EDF is a fixed-priority scheduling algorithm used in real-time systems where each task is statically bound to a particular processor (core). Both of them are implemented independently, and each processor uses the modified EDF [21-22] algorithm to schedule tasks on its core according to the task's deadline and CPU utilization. LITMUS^{RT}, which is built as an extension of the Linux kernel, offers a set of experimental real-time scheduling policies, including PART-EDF.

8.1 Working Principle of Partitioned EDF (PART-EDF) – Scheduler Plugin

Task Partitioning:

- Tasks are partitioned into two or three clusters depending on their criticality.
- Tasks with high interrupt frequency are grouped into one cluster, while normal processing tasks into a processing cluster.
- EDF Scheduler on Each Core:
- Each cluster maintains a separate queue for the PART-earliest deadline first (EDF) scheduling policy.
- Where the due date of two activities is alike, then they are arranged according to the time when the activities are arriving.
- No Task Migration:
- Contrary to global EDF, tasks do not move from one core to another. This makes it easier to schedule, but load partitioning must be efficient to ensure that the loads are well distributed.

8.2 PART-EDF Plugin Module Pseudocode

```
PROCEDURE part_edf_domain(pedf, cpu)  // Initialize Part-EDF Domain
  SET pedf.edf_domain ⟵ initialized
  SET pedf.assigned_cpu ⟵ cpu
  SET pedf.scheduled_task ⟵ NULL
PROCEDURE requeue(task, edf)  // Requeue Task
  IF task.state = RUNNING THEN RETURN
  ADD task TO edf.ready_queue IF task.state = READY ELSE edf.release_queue
PROCEDURE preempt(pedf)  // Preemption Handling
  IF pedf.scheduled_task IS preemptable THEN TRIGGER preemption
```

```
PROCEDURE part_edf_preempt_check(pedf)
  IF preemption required THEN CALL preempt(pedf)
PROCEDURE part_edf_check_resched(edf)
  CALL part_edf_preempt_check(edf.domain)
PROCEDURE job_completion(task)  // Job Completion
  TRACE job completion
  RESET task.status
  PREPARE task FOR next period
PROCEDURE part_edf_schedule(prev)  // Scheduling Decision
  LOCK scheduling
  IF prev BLOCKED OR OUT-OF-TIME THEN HANDLE job completion
  HANDLE preemption
  IF prev NEEDS requeue THEN CALL requeue(prev, edf)
  SET next_task ⟵ GET next ready task OR KEEP current task
  TRACE scheduling decision
  UNLOCK scheduling
  RETURN next_task\
PROCEDURE part_edf_task_new(task, on_rq, is_scheduled)  // Task Management
  LOCK scheduling
  INITIALIZE job parameters
  IF NOT is_scheduled AND on_rq THEN CALL requeue(task, edf)
  UNLOCK scheduling
PROCEDURE part_edf_task_wake_up(task)
  LOCK scheduling
  IF sporadic_release_required THEN HANDLE job release
  IF task NOT scheduled THEN CALL requeue(task, edf)
  UNLOCK scheduling
PROCEDURE part_edf_task_block(task)
  TRACE task blocking event
PROCEDURE part_edf_task_exit(task)
  LOCK scheduling
  REMOVE task FROM ready_queue IF in queue
  IF task IS scheduled THEN SET scheduled_task ⟵ NULL; CALL preempt(pedf)
  UNLOCK scheduling
```

8.3 Explanation of the Pseudocode

The PART-EDF Scheduling Procedures manage task execution, scheduling, and pre-emption in a Partitioned Earliest Deadline First (PART-EDF) scheduling system. The "part_edf_domain" initializes the EDF domain, assigning it to a specific CPU. The requeue function ensures that tasks are placed in the appropriate queue based on their state. Pre-emption handling is managed by checking whether the currently scheduled task should be pre-empted and if required, pre-emption is triggered. The "part_edf_check_resched" procedure calls the pre-emption check to determine if re-scheduling is necessary. Upon job completion, the task status is reset. The "part_edf_schedule" function makes scheduling decisions by locking the scheduler, handling the job completion, checking for any pre-emptions, and selecting the next task. Task management is handled by "part_edf_task_new", which initializes job pa-

rameters and enqueues tasks if necessary. The function "part_edf_task_wake_up" manages sporadic task releases and requeues unscheduled tasks. The function "part_edf_task_block" logs task blocking events, while "part_edf_task_exit" ensures proper task removal from the ready queue and triggers pre-emption if needed. These procedures together ensure efficient and predictable task execution in PART-EDF scheduling.

8.4 Application Code – Pseudo Code – LITMUS^{RT}

```
INPUT: tasks[] = (task_id, period, execution_time, cpu_id)
INITIALIZE LitmusRT environment
FOR each task IN tasks DO:
   CREATE_THREAD(task_thread, task)
   SET_CPU_AFFINITY(task.cpu_id)
   SET_REALTIME_PARAMS(task.execution_time, task.period, task.cpu_id)
   START_THREAD()
WAIT_FOR_ALL_THREADS()  // Optional
PROCEDURE task_thread(task):
   SET_REALTIME_PARAMS(task.execution_time, task.period, task.cpu_id)
   TRANSITION_TO(LITMUS_RT_TASK)
   WHILE true DO:
      SLEEP_UNTIL_NEXT_PERIOD()
      EXECUTE_TASK_LOGIC()
      LOG_EXECUTION_DETAILS()
   TRANSITION_TO(BACKGROUND_TASK)
```

8.5 Explanation of the Pseudocode

The Part-EDF Scheduling Algorithm initializes a LITMUS^{RT} environment to manage real-time tasks efficiently. It takes a list of tasks as input, each defined by a unique task ID, period, execution time, and assigned CPU core. The system first initializes the LITMUS^{RT} environment, ensuring that real-time scheduling is enabled. For each task, a dedicated thread is created, and CPU affinity is set to bind the task to a specific core. Real-time parameters, including execution time, period, and CPU assignment, are configured before the thread starts execution. Each task then transitions into real-time mode, where it repeatedly waits for its next activation based on its defined period. Upon activation, the task executes its logic, logs execution details, and then returns to a waiting state until the next period. This cyclic execution continues to ensure Earliest Deadline First (EDF) scheduling, optimizing task execution and meeting deadlines effectively. Finally, when the task is no longer needed, it transitions back to background mode, ensuring efficient resource utilization and system stability.

9 Results and Discussions

In Fig. 5, it shows how to set up the PART-EDF plugin in LITMUS^{RT}. Once enabled, we ensure that the PART-EDF plugin is enabled or not with the help of supported commands, which have been depicted in the following Fig. 5.

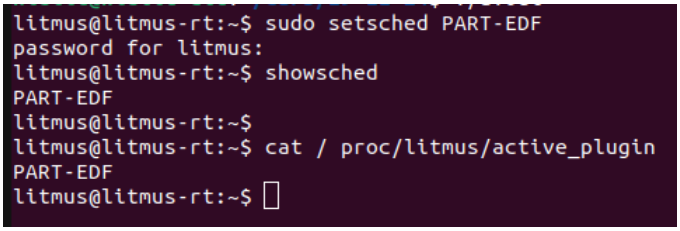

Fig. 5 Enabling PART-EDF plugin module in LITMUS^{RT} environment

The PART-EDF scheduling strategy proposed in this paper was evaluated on a task set with diversified execution time and period, the major cycle is 600 ms. The task set is described with the execution time, deadline, period, and individual utilization value and the total execution count in the major cycle is shown in Tab. 1. The task allocation strategy used in PART-EDF of our scheduler plugin provided in Tab. 2 is employed to distribute high-priority, high-frequency tasks into one cluster and other periodic tasks to another cluster to reduce the overhead due to pre-emptions and thus enhance CPU utilization. The time for executing the tasks is represented by Fig. 6, which is plotted through a bespoke Python toolkit for the visualization of efficient resource scheduling and performing tasks exhibited in the PART-EDF plugin. This holistic approach brings into focus the ability of a scheduler to tackle real-time tasks on the multicore processors.

Tab. 1 Taskset with its parameters

Thread ID	Execution Time (*C*) (Measured WCET)	Deadline (*D*)	Period (*T*)	Major cycle	Iterations in Major cycle
T1	10 ms (14.02 ms)	20 ms	20 ms	600 ms	30
T2	10 ms (10.93 ms)	20 ms	20 ms	600 ms	30
T3	15 ms (16.34 ms)	40 ms	40 ms	600 ms	15
T4	5 ms (7.15 ms)	20 ms	20 ms	600 ms	30
T5	1 µs (1.03 µs)	10 µs	10 µs	600 ms	60 000
T6	5 ms (6.05 ms)	25 ms	25 ms	600 ms	24
T7	8 ms (9.21 ms)	30 ms	30 ms	600 ms	20
T8	8 ms (8.34 ms)	30 ms	30 ms	600 ms	20
T9	6 ms (6.76 ms)	30 ms	30 ms	600 ms	20

Tab. 2 Core Utilization and Core assignment PART-EDF

Core	**Assigned Tasks**	**Total Theoretical Utilization**	**Measured Utilization**
Core 1	T1, T4, T5	0.85	**1.165**
Core 2	T2, T6	0.7	**0.788**
Core 3	T3, T9	0.575	**0.634**
Core 4	T7, T8	0.534	**0.585**

Tab. 1 shows the increase in the execution time when the tasks are run on a typical multicore processor having four cores. Tab. 2 gives the theoretical and the measured utilization for each core. In Core 1, the utilization goes beyond ONE due to the task T5 having high periodicity and high frequency of occurrence. T5 will pre-empt the other tasks T1 and T4 (increase in execution time is given in Tab. 1), and hence, the core utilization goes beyond ONE, and it leads to deadline misses. To avoid this, it is recommended to assign high-frequency and high-priority tasks in any real-time embedded systems like sonar and radar to separate core.

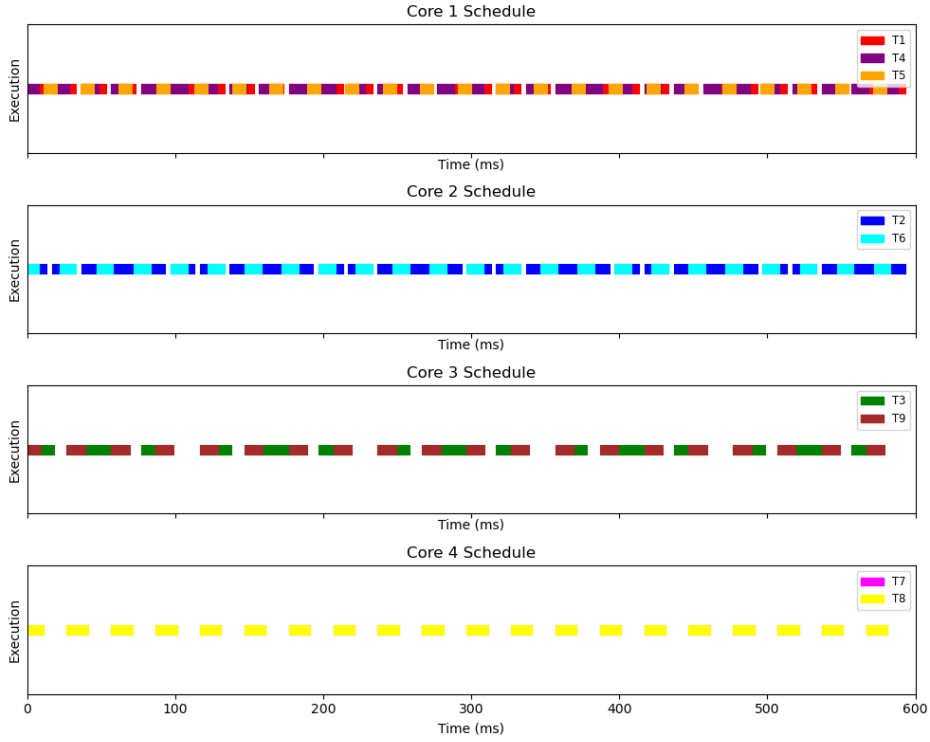

Fig. 6. PART-EDF plugin thread switching behavior

The work distribution and execution on each core as described in this paper are demonstrated in the Python framework used in generating Fig. 6 covering the entire 600 ms major cycle. This visualization demonstrates how the PART-EDF function enables the efficient scheduling of tasks into cores with little or no downtime and within set time.

As can be observed from the results, Thread 5 with its very high priority due to the very high repetition rate over the major cycle, severely affects the execution time of other threads running in Core 1. Since T5 has high priority and high frequency, it tends to pre-empt other tasks and consume more of the CPU time. In real-time, there can be situations where the CPU utilization of any core may exceed 1 due to the overhead caused by pre-emptions. This behavior exposes some of the issues that real-time systems developers encounter when handling high-frequency tasks. To this end, in the design of our PART-EDF scheduler, we have suggested that partitioning the tasks

based on the category into multiple clusters will lead to reduced pre-emptions and enhanced schedulability.

10 Conclusion and Future Scope

This work provides an adaptive Partitioned Earliest Deadline First (PART_EDF) scheduler for real-time defense applications like radar, sonar, etc. In this work, a typical sonar signal processing system is implemented with the LITMUS^{RT} real-time Linux kernel. To support the static partitioning of tasks across cores, a custom scheduler plugin, PART_EDF, with a dual-ready queue was designed and incorporated into the LITMUS^{RT} architecture. The system was evaluated in a multi-core testbed with emulated computationally demanding and periodic tasks relevant to sonar signal processing such as filtering, beamforming and target identification. The partitioning scheme where different categories of tasks are partitioned into two or three clusters results in efficient CPU utilization, leading to miniaturization and low power consumption, an important requirement in defense applications.

The experimental results prove that PART_EDF improves the core utilization, decreases the deadline miss rate, and guarantees definite task scheduling under high load conditions. One of the major advantages of the partitioning scheme into clusters is reduced inter-core migration overhead, making the scheme ideal for resource-limited and real-time application environments. This implementation illustrates that PART_EDF is a robust algorithm to be implemented for high-risk defense applications, where predictable performance and meeting the strict constraints on access time are crucial.

It is still worthwhile in future to conduct more experiments with PART_EDF on systems with more cores and/or heterogeneous architectures to understand the performance of the algorithm in the most extreme scenarios. Further, incorporating energy-related scheduling mechanisms into portable defense platforms would technically be beneficial in the energy consumption disparity, which is crucial in the portability of such defense systems. Last but not least, it must be noted that the proposed PART_EDF can be incorporated into hybrid systems where original sonar signal processing techniques and machine learning target classification are incorporated for making smart decisions in real time. To the extent that these advancements would increase the resilience, versatility, and generality of PART_EDF, they would also expand its usefulness beyond defense systems to real-time scheduling problems in general.

References

[1] SARANYA, N. and R.C. HANSDAH. Dynamic Partitioning Based Scheduling of Real-Time Tasks in Multicore Processors. In: *2015 International Symposium on Real-Time Distributed Computing*. Auckland: IEEE, 2015, pp. 190-197. DOI 10.1109/ISORC.2015.23.

[2] HAN, J.J., W. CAI and D. ZHU. Resource-Aware Partitioned Scheduling for Heterogeneous Multicore Real-Time Systems. In: *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC).* San Francisco: IEEE, 2018. DOI 10.1109/DAC.2018.8465907.

[3] AKRAM, N., Y. ZHANG, S. ALI and D.H.M. AMJAD. Efficient Task Allocation for Real-Time Partitioned Scheduling on Multi-Core Systems. In: *2019 16th*

International Bhurban Conference on Applied Sciences and Technology (IBCAST). Islamabad: IBCAST, 2019, pp. 492-499. DOI 10.1109/IBCAST.2019.8667139.

[4] XU, M., L.T.X. PHAN, H.-Y. CHOI, Y. LI, H. LI and C.A. LU. Holistic Resource Allocation for Multicore Real-Time Systems. In: *Real-Time and Embedded Technology and Applications Symposium (RTAS).* Montreal: IEEE, 2019, pp. 345-356. DOI 10.1109/RTAS.2019.00036.

[5] MA, K., W. HU, J. LIU and D.Y. GAN. Partition Scheduling Algorithm for Shared Resources in Real-Time Systems. In: *International Conference on Systems, Man, and Cybernetics (SMC)*. Melbourne: IEEE, 2021, pp. 679-684. DOI 10.1109/SMC52423.2021.9659117.

[6] CALANDRINO, J., D. BAUMBERGER, T. LI, S. HAHN and J. ANDERSON. LITMUS^{RT}: A Testbed for Empirically Comparing Real-Time Multiprocessor Schedulers. In: *Real-Time Systems Symposium (RTSS)*. Rio de Janeiro: IEEE, 2006. DOI 10.1109/RTSS.2006.1.

[7] BARUAH, S. Partitioned EDF Scheduling: A Closer Look. *Real-Time Systems Journal*, 2013, **49**(6), pp. 715-729. DOI 10.1007/s11241-013-9186-0.

[8] STEVANATO, A., T. CUCINOTTA, L. ABENI and D.B. de OLIVEIRA. An Evaluation of Adaptive Partitioning of Real-Time Workloads on Linux. In: *24th International Symposium on Real-Time Distributed Computing (ISORC)*. Daegu: IEEE, 2021, pp. 53-61. DOI 10.1109/ISORC52013.2021.00018.

[9] SHEIKH, S.Z. and M.A. PASHA. A Dynamic Cache-Partition Schedulability Analysis for Partitioned Scheduling on Multicore Real-Time Systems. *IEEE Letters of the Computer Society,* 2020, **3**(2), pp. 46-49. DOI 10.1109/LOCS.2020.3013660.

[10] ZHANG, X., S. HUANG and J. LI. Implementation of Real-Time Scheduling Algorithm on Multi-Core Platform. In: *International Conference on Computer Network, Electronic and Automation.* Xi'an: ICCNEA, 2020, pp. 66-71. DOI 10.1109/ICCNEA50255.2020.00023.

[11] LI, B., M. XIAOCHUAN, Y. SHEFENG and Y. LI. A Sonar Array Processing System Based on Multicore DSPs. In: *2012 IEEE 11th International Conference on Signal Processing*. Beijing: IEEE, 2012, pp. 421-424. DOI 10.1109/ICoSP.2012.6491690.

[12] DIGALWAR, M., P. GAHUKAR and S. MOHAN. Energy Efficient Real Time Scheduling on Multi-core Processor with Voltage Islands. In: *International Conference on Advances in Computing, Communications and Informatics.* Bangalore: ICACCI, 2018, pp. 1245-1251. DOI 10.1109/ICACCI.2018.8554680.

[13] SALAMI, B., H. NOORI and M. NAGHIBZADEH. Fairness-Aware Energy Efficient Scheduling on Heterogeneous Multi-Core Processors. *IEEE Transactions on Computers,* **70**(1), pp. 72-82, 2021. DOI 10.1109/TC.2020.2984607.

[14] FAN, M., Q. HAN, G. QUAN and S. REN. Multi-Core Partitioned Scheduling for Fixed-Priority Periodic Real-Time Tasks with Enhanced RBound. In: *International Symposium on Quality Electronic Design*. Santa Clara: 2014, pp. 284-291. DOI 10.1109/ISQED.2014.6783338.

[15] CHEN, J.J., J. SHI, G. von der BRÜGGEN and N. UETER. Scheduling of Real-Time Tasks with Multiple Critical Sections in Multiprocessor Systems. *IEEE Transactions on Computers,* 2022, **71**(1), pp. 146-160. DOI 10.1109/TC.2020.3043742.

[16] ANDERSON, J.H., J.M. CALANDRINO and U.C. DEVI. Real-Time Scheduling on Multicore Platforms. In: *Real-Time and Embedded Technology and Applications Symposium (RTAS'06).* San Jose: IEEE, 2006, pp. 179-190. DOI 10.1109/RTAS.2006.35.

[17] LIU, C.L. and J.W. LAYLAND. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *Journal of the AC.* 1973, **20**(1), pp. 46-61. DOI 10.1145/321738.321743.

[18] BAKER, T.P. A Comparison of Global and Partitioned EDF Schedulability Tests for Multiprocessors. In: *International Conference on Real-Time and Network Systems.* Paris: Citeseer, 2005.

[19] KURZAK, J., A. BUTTARI and J. DONGARRA. Solving Systems of Linear Equations on the CELL Processor Using Cholesky Factorization. *IEEE Transactions on Parallel and Distributed Systems*, 2008, **19**(9), pp. 1175-1186. DOI 10.1109/TPDS.2007.70813.

[20] RAMESH P. and U. RAMACHANDRAIAH. Performance Evaluation of Real Time Scheduling Algorithms for Multiprocessor Systems. In: *International Conference on Robotics, Automation, Control and Embedded Systems.* Chennai: RACE, 2015. DOI 10.1109/RACE.2015.7097297.

[21] BOUAKAZ, A., T. GAUTIER and J.P. TALPIN. Earliest-Deadline First Scheduling of Multiple Independent Dataflow Graphs. In: *Workshop on Signal Processing Systems (SiPS).* Belfast: IEEE, 2014. DOI 10.1109/SiPS.2014.6986102.

[22] DAVIS R. and A. BURNS. A Survey of Hard Real-Time Scheduling for Multiprocessor Systems. *ACM Computing Surveys*, 2011, **43**(4). DOI 10.1145/1978802.1978814.