Defence Science Journal, Vol. 75, No. 2, March 2025, pp. 263-268, DOI : 10.14429/dsj.20751

Partitioned Cache Aware Dynamic Scheduling for Real-Time Applications on Multicore Processors

Balakrishnan P.[#,*], Rajesh M.[$] and Rajesh R.[!]

[#]*Cochin University of Science and Technology, Kochi - 682 022, India*
[$]*National Institute of Electronics and Information Technology, Calicut - 673 601, India*
[!]*DRDO-Naval Physical and Oceanographic Laboratory, Kochi - 682 021, India*
[*]*E-mail: bala.hema.bala@gmail.com*

ABSTRACT

Efficient task partitioning and scheduling on multicore processors are critical for optimizing performance and resource utilization in real-time systems. This paper explores a dynamic approach to task partitioning and scheduling, leveraging Intel Cache Allocation Technology (CAT) and pseudo-locking to enhance predictability and reduce inter-core interference. By dynamically allocating cache resources to critical tasks, partitioning high-frequency tasks into a separate cluster and isolating them from contention, the system achieves improved schedulability. Additionally, an adaptive Earliest Deadline First (EDF) scheduling algorithm is introduced, which allocates the tasks to free cores in real time based on workload variations and resource availability. The proposed techniques are validated through typical applications in signal processing and other similar systems, where high throughput, low latency, and strict timing constraints are paramount. Experimental results of the Modified-EDF approach demonstrated a reduction of 4.6 % in Worst-Case Execution Time (WCET) compared to SCHED_FIFO and a decrease of 2.3 % in CPU utilization Similarly, it achieved a 4.2 % improvement in WCET over SCHED_RR and a 2.3 % improvement over SCHED_DEADLINE., highlighting its efficiency gains through deadline sensitivity and cache-awareness, thus making this approach highly suitable for safety-critical and high-performance computing environments.

Keywords: EDF; Signal processing; Real-Time; Linux; Cache allocation technology

NOMENCLATURE

CLOS	: Classes of service
LLC	: Last-level cache
CAT	: Cache allocation technology
RTOS	: Real-time operating system
CBM	: Capacity bit mask
EDF	: Earliest deadline first
WCET	: Worst case execution time
MSR	: Memory specific register
OS	: Operating system
FFT	: Fast fourier transform

1. INTRODUCTION

Multicore Processors are used extensively for the implementation of real-time embedded systems for defence applications like radar, sonar, missiles, etc. With the advent of multicore processors, they are extensively used in systems where massive computation and better response with low power consumption are required[1,3,9]. A complex application is partitioned into smaller blocks by the developer to run on multiple cores but had synchronisation issues between the various cores to meet critical timelines of system[2,4-5].

Linux being open source is used in multicore processors as an Operating System(OS)[6,8,11]. The OS handles the scheduling of tasks depending on the scheduling policy selected by the application developer. To achieve the real-time characteristics of the system, the developer has to go through several iterations, which leaves a great burden on him. This paper discusses the issues and their solutions in the implementation of a complex real-time embedded system on multicore processors, a typical sonar signal processor is considered as a reference.

Received : 02 December 2024, Revised : 05 February 2025
Accepted : 06 February 2025, Online published : 24 March 2025

2. METHODOLOGY

A partitioned scheduling technique is developed for real-time embedded applications implemented on Multicore processors. The existing scheduler in Linux is modified and Intel's cache allocation technology is used for cache partitioning and locking.

2.1 Typical Multicore Architecture

In homogeneous multicore processors, multiple identical cores share common resources like memory, IO devices[7,10,14,16], etc. Each core has a dedicated L1 instruction & data cache and L2 cache. But L3, the Last Level Cache (LLC) has a large capacity and is shared by the cores of the multicore processor. A typical multicore processor architecture is shown in Fig 1. The cores of the processor share common resources leading to contention which affects the processing Time leads to deadline misses, not tolerable in critical real-time systems like sonar and radar.

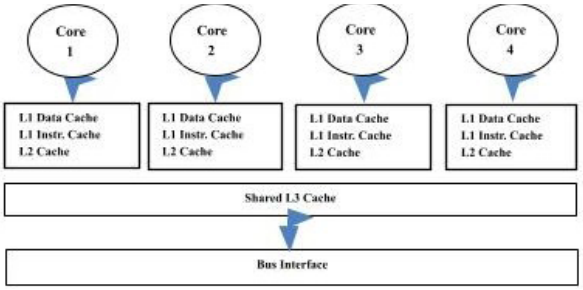

Figure 1. Typical 4 core multicore architecture.

Application developers had to resort to an iterative approach to partition the tasks to cores, and this may lead to errors, resulting in situations where certain critical tasks may miss the deadlines and overall effect of deadline misses is the equipment failure. This paper discusses the issues and the solutions in the development of complex real-time systems on multicore processors in the Linux environment.

2.1.1 Multicore Utilisation (Ui)

Consider a multicore processor with N cores (M_1,M_2,M_3,…..M_N) and a task set consisting of m independent tasks (τ_1,τ_2,…..τ_m) of any real-time signal processor application software. Each task i is shown as a quadruple:

$$\tau_i = \left(C_i, D_i, T_i, P_{ioh}, I_i\right) \quad (1)$$

where, C_i is the Worst-Case Execution Time, D_i is the deadline, T_i is the period and l_i is interference encountered by the task τ_i. P_{ioh} is defined as the pre-emption overhead incurred by the i^{th} task. Pre-emption occurs when a high-priority task as compared to the running tasks is ready for execution. Interference l_i is due to the read-write operations to the shared cache by co-running tasks; the effect of both interferences and pre-emptions is in the enhancement of C_i. The hyper period of any task set, H, is defined as the least common multiple (LCM) of the periods of all the periodic tasks of the task set:

$$H = LCM\left\{T_i\right\},\ i = 0,1,....(n-1) \quad (2)$$

O_i is the number of occurrences of the task τ_i in the hyper period H:

$$O_i = \frac{H}{T_i} \quad (3)$$

Due to interferences by the read-write operations of co-running tasks, the C_i of any task will be C_i+l_i. The interference l_i is different from one instance of the task to the next, as the co-running tasks on other cores will be different. With Interference,

$$U_i = \frac{\left(C_i + I_i\right)}{T_i} \forall\ i \quad (4)$$

When task pre-emptions are allowed, the overhead due to pre-empting any task, cache evictions, and context switches result in an increase in execution time[12]. With task pre-emptions and interferences,

$$U_i = \frac{\left(C_i + I_i + P_{ioh}\right)}{T_i} \forall\ i \quad (5)$$

With no pre-emptions or interferences from Linux tasks or other tasks, CPU utilisation is

$$U_i = \frac{C_i}{T_i} \forall\ i \quad (6)$$

The execution time of the task depends on the core operating frequency and the core architecture[13]. Estimation of l_i and P_{ioh} is a difficult problem due to its random nature, it depends on the multicore architecture, and hence the design of the signal processor system should take care of the effects of these unpredictable sources[18,20].

A typical frequency domain beam former is shown in Fig. 2. The data samples from the M sensors are subjected to N point FFT. A typical submarine sonar will have arrays with thousands of sensors, and the data samples from all the sensors to be processed. The input data is normally received on high-speed interfaces like Ethernet. The processing hardware configuration of any sonar system should be capable of handling high-frequency interrupts and large-size data processing. The context diagram of a typical sonar signal processor is given in Fig 3.

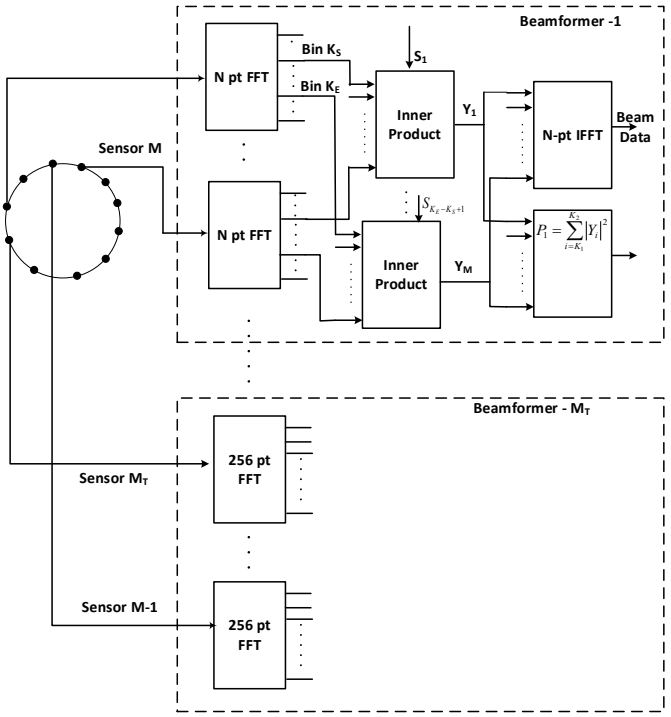

Figure 2. Typical frequency domain beam former.

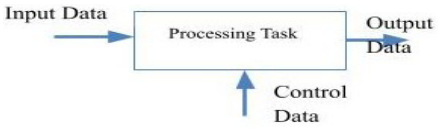

Figure 3. Context diagram of a typical sonar signal processor.

The tasks of sonar signal processor are categorised into three (1) Input task (2) output task processed data to display (3) Processing task (4) control task. The input data received by way of interrupts occur at regular intervals. The input task occurs at fast rates, processing time is small.

For example, in an array with 5,000 sensors sampled at 50 KHz, using a 24-bit sigma-delta ADC, the data size is 15 KB in 20 microsec. The 15 KB of data will be received as Ethernet packets each with 1536 bytes, the data reception time of each Ethernet packet is approx. 12 microsec. for a Gigabit link. There will be approx.1000 occurrences (15 kb / 1536)) of

input task during the execution of any processing task shown in the signal processing system (Fig. 2). Every occurrence of input task will give rise to interferences and pre-emptions. So the execution time of the task C_i in the isolated condition is less than C_i in the actual condition,

$$\left(C_{i\ isol} < C_{i\ actual}\right) \Rightarrow \left(\frac{C_{i\ actual}}{T_i}\right) > \left(\frac{C_{i\ isol}}{T_i}\right) \quad (7)$$

implies that the core utilisation U_i increases and will lead to deadline misses.

2.1.2 Grouping of Tasks Into Clusters

The characteristics of the tasks of any sonar signal processor are analysed, and classified into two clusters, one having low C_i & T_i and the other having higher T_i. The input tasks and control tasks are assigned higher priority and have a higher frequency of occurrence. The input tasks in the example above, if assigned to the same core running processing tasks, will lead to a large no of pre-emptions and interferences to the processing tasks leading to increased execution time. The larger the execution time of any processing task, the higher will be the number of pre-emptions. Hence high frequency tasks may be grouped into one cluster and the processing tasks into another cluster.

2.1.3 Issues in Using Multicore Processors

The major issues in using a multicore processor with an operating system like Linux are (1) As Input tasks occur at fast rates of the order of a few micro-seconds, task pre-emptions of processing tasks will increase; disabling pre-emptions will lead to deadline misses of high-frequency tasks (2) Lack of efficient schedulers to meet the critical timelines of the real-time signal processor.

The solution suggested in Fig. 4 divides the cores into two clusters, one to handle high frequency & high priority tasks like input tasks, control tasks and output tasks. The core affinity and task priority can be set in the application program. Another cluster is formed for processing tasks where a global scheduling policy may be adopted.

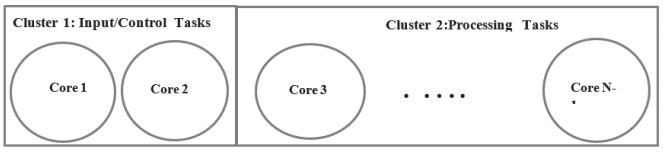

Figure 4. CPU clustering.

The scheduling policies available in Linux for real-time applications are SCHED_FIFO, SCHED_RR and SCHED_DEADLINE. The SCHED_DEADLINE is generally not used due to its limitations in meeting deadlines. The developer has to feed the "runtime" of all tasks/threads and should have a good understanding of the processor architecture, cache configuration etc.

2.2 Modified SCHED_DEADLINE

An adaptive scheduler developed eliminates the limitations of the existing SCHED_DEADLINE policy. The scheduler and the task partitioning techniques were implemented on an Intel Xeon Board with the following specifications as given below in Table 1.

Table 1. Intel xeon board specification

CPU	Max freq.	L1D	L1I	L2	L3	No. of Cores
Intel Xeon D-1548@ 2.0 GHz	2.3 GHZ	32 KB	32 KB	256 KB	2 MB	8

The main limitation of the SCHED_DEADLINE policy is due to the inaccurate run-time of the tasks fed into the application program. The run time is inaccurate as it is estimated when the task is run in isolation. But when the task is running along with other tasks on different cores, the execution time is different from that estimated in isolation. In the modified scheduler, the worst-case estimation time of each task is measured online when all tasks/threads of the application program are running on all the cores of the multicore processor. In every instance of the task, if the measured execution time is higher, it replaces the existing value. After a few iterations worst-case execution time of all the tasks is obtained.

In the modified scheduler, a new task is assigned to any free core only after estimating the ratio of ΣC_i and hyper-period (H) of all tasks already assigned to that core. The ratio is again estimated after including the C_i of the new task and the new task is assigned to that core only if the ratio is less than ONE. This ensures maximum utilisation of the cores and eliminates deadline misses.

2.2.1 Cache Partitioning and Locking Implementation

Intel's Cache Allocation Technology (CAT) is a resource management feature that provides fine-grained control over the last-level cache (LLC). It allows software to determine, limit the amount of cache allocated to specific threads, applications, virtual machines, or containers 15. This capability is especially useful in environments like data centre's, where managing resource contention among multiple workloads is critical.

CAT operates by introducing Classes of Service (CLOS) and Capacity Bitmasks (CBMs), enabling flexible cache allocation. CLOS acts as a resource control tag, grouping threads or applications, while CBMs define how much cache can be allocated to a specific CLOS. This technology is realised by Model-Specific Registers (MSRs) and they are read/written by using ordinary CPU instructions such as CPUID.

2.2.2 Pseudo-Locking

Pseudo-locking in Intel Cache Allocation Technology (CAT) is a method applied to provide cache regions in the shared Last-Level Cache (LLC) for predictable and performance-critical applications. This reduces cache conflicts as workloads are grouped to a specific portion of the cache and predictability is also improved. The cache contention is reduced by giving higher priority to important workloads. Nevertheless, it does not imply changes at the hardware level, but it builds on existing CAT functionality. Unlike the dynamic nature of true locking, the cache partitions must be configured manually according to the workload. If the reserved cache region is under-utilised, it can lead to inefficient use of

LLC resources. Intel CAT allows partitioning of the LLC into regions using Capacity Bit Masks (CBMs). Pseudo-locking exploits this feature to "lock" specific regions of the cache for critical workloads.

2.2.3 Pseudo-locking-Working Principle

A specific region of the LLC is defined using CAT. Data are accessed by the application to load it into specified cache lines. Once the region is initialized, the application operates primarily out of the reserved region 17,19. Coherency transactions reduce cache misses and interferences from other applications thereby achieving high determinism. One additional feature of pseudo-locking is the ability to map out other workloads to prevent them from removing reserved data from the cache.

Memory-centric scheduling is a technique used in RTOS to enhance and manage memory operations of a memory-centric scheduler. In this, each thread is partitioned between the memory-computation cycle and the execution cycle. Every time a thread executes an operation in the memory, it needs permission from the scheduler and is called memory prefetching. During the pre-fetch phase, it uses Intel's Cache Allocation Technology (CAT) and the pseudo-locking concept, securing the memory region to a given thread. This also leads to private cache and helps to reduce cache conflict. In case multiple threads try to pre-fetch, the current thread trying to pre-fetch is forced to wait until the lock is given back by the other thread. After the memory pre-fetch phase is complete the thread moves into the compute phase and creates the lock. This process dramatically decreases the cases in which one thread has to contend for access to the cache with another one. The memory pre-fetch in combination with Intel's pseudo-locking is explained in the algorithm 1 & 2 and the pseudo-code is given in Algorithm 1 & Algorithm 2 respectively.

Algorithm 1: Memory pre-fetch into cache and computation

1. Initialization:
   - Initialize lock_table such that all memory_region entries are UNLOCKED.
   - Initialize queue as an empty list.
2. Function request_memory_access(thread_id, memory_region):
   - If lock_table[memory_region] == UNLOCKED:
     - Set lock_table[memory_region] = LOCKED_BY(thread_id).
     - Return GRANTED.
   - Else:
     - Enqueue (thread_id, memory_region) into queue.
     - Return DENIED.
3. Function release_memory_lock(memory_region):
   - Set lock_table[memory_region] = UNLOCKED.
   - If queue contains threads waiting for memory_region:
     - Dequeue the next thread, next_thread, from the queue.
     - Set lock_table[memory_region] = LOCKED_BY(next_thread.thread_id).
     - Signal next_thread that access has been granted.
4. Function thread_task(thread_id, memory_region):
   - While request_memory_access(thread_id, memory_region) == DENIED:
     - Wait until the lock for memory_region is released.
   - Perform:
     - Memory Prefetch Phase:
       - Prefetch memory_region into the cache using prefetch_to_cache(memory_region).
     - Computation Phase:
       - Compute on the prefetched data using compute_on_data(memory_region).
     - Release Lock:
       - Call release_memory_lock(memory_region) to unlock the memory region.

Algorithm 1: for Memory Pre-fetch into Cache and Computation

Input:

- lock_table: Table to track the lock status of memory regions.
- Queue: Queue to hold threads waiting to access memory regions.
- thread_id: ID of threads requesting memory access.
- memory_region: Memory-region accessed by thread.

Output:

- Controlled access to memory regions for threads with pre-fetching and computation phases.

Highlights:

- Global variables lock_table and queue maintain the lock status and manage threads waiting for memory access
- Memory-access requests are either granted or queued depending on the lock status of the requested memory region
- Thread releases lock after its computation phase, allowing the next waiting thread in the queue to gain access
- The memory prefetching phase ensures data is ready in the cache for computation.

Algorithm 2: Kernel-based memory lock management with prefetching and computation

   - Initialize kernel_queue as an empty list.
2. Function kernel_request_memory_access(thread_id, memory_region):
   - Acquire kernel_lock to ensure mutual exclusion.
   - If kernel_lock_table[memory_region] == UNLOCKED:
     - Set kernel_lock_table[memory_region] = LOCKED_BY(thread_id).
     - Release kernel_lock.
     - Return GRANTED.
   - Else:
     - Enqueue (thread_id, memory_region) into kernel_queue.
     - Release kernel_lock.
     - Return DENIED.
3. Function kernel_release_memory_lock(memory_region):
   - Acquire kernel_lock to ensure mutual exclusion.
   - Set kernel_lock_table[memory_region] = UNLOCKED.
   - If kernel_queue contains threads waiting for memory_region:
     - Dequeue the next thread, next_thread, from kernel_queue.
     - Set kernel_lock_table[memory_region] = LOCKED_BY(next_thread.thread_id).
     - Wake up next_thread.
   - Release kernel_lock.
4. Function kernel_thread_task(thread_id, memory_region):
   - While kernel_request_memory_access(thread_id, memory_region) == DENIED:
     - Put thread_id to sleep until it is woken up.
   - Perform:
     - Memory Prefetch Phase:
       - Prefetch memory_region into cache using kernel_prefetch_to_cache(memory_region).
     - Computation Phase:
       - Perform computation on prefetched data using kernel_compute_on_data(memory_region).
     - Release Lock:
       - Call kernel_release_memory_lock(memory_region) to unlock the memory region.

Algorithm 2: For Kernel-Based Memory Lock Management with Prefetching and Computation

Input:

- kernel_lock_table: Tracks lock status of memory regions.
- kernel_queue: Queue to manage threads waiting for memory-region access.

- thread_id: Identifier for requesting thread.
- memory_region: Memory-region for which access is requested.

Output:
- Controlled access to shared memory regions for threads, including memory pre-fetching and computation phases.

Highlights:
- Mutual Exclusion: Use of kernel_lock ensures that the kernel_lock_table and kernel_queue are accessed safely in a concurrent environment
- Thread Management: Threads denied access are queued and put to sleep, minimizing memory contention
- Efficient Memory Access: Pre-fetching memory into cache before computation reduces latency
- Fairness: Threads in kernel_queue are processed in FIFO order, ensuring equal access to shared resources.

3. RESULTS

This section analyses the performance of various scheduling mechanisms, including SCHED_FIFO, SCHED_RR, SCHED_DEADLINE, and Modified-EDF, based on metrics such as WCET (worst-case Case Execution Time), CPU utilisation, and adherence to deadlines. The target application selected for benchmarking was a typical signal processing application involving a mix of threads that are compute-intensive with memory-access operations. These threads represent typical real-time workloads with varying computational demands (Table 2).

Table 2. Comparison of test results in terms of schedulability improvements

Scheduler	Execution time (units)	Deadline (units)	WCET (Ci)	CPU utilisation (%)	Key observations
Sched_FIFO	1.75	NA	1.803	38.961	Predictable but suffers from resource contention.
Sched_RR	1.75	NA	1.795	38.995	Fair scheduling but higher context-switching overhead.
Sched_Deadline	1.75	4.5	1.761	38.693	Deadline-aware but lacks cache optimisation.
Modified-EDF	1.75	4.5	1.72	38.067	Cache-aware, deadline-sensitive, and reduced throttling.

The results compare the different scheduling mechanisms in managing tasks with mixed criticality. SCHED_FIFO and SCHED_RR provide predictable execution times but suffer from deadline misses and higher CPU utilisation due to context-switching overhead and resource contention.

4. DISCUSSION

Existing SCHED_DEADLINE in Linux, based on EDF principles, offers significant improvements as compared to the other two policies achieving better WCET, lower CPU utilisation and less deadline misses. The Modified-EDF enhances schedulability, prevents deadline misses, and reduces cache evictions by incorporating the pseudo-locking facility of Intel CAT for cache partitioning. In Modified-EDF execution time of tasks is constantly monitored to arrive at the worst-case execution time and this value is used to decide the task allocation to free cores. This reduces the throttling time observed in EDF scheduling. Compared to SCHED_FIFO, the Modified-EDF approach reduces worst-case execution time (WCET) by approximately 4.6 % (from 1.803 to 1.72), lowers CPU utilization by around 2.3 % (from 38.961 % to 38.067 %). Against SCHED_RR, WCET improved by 4.2%, and against SCHED_DEADLINE it improved by 2.3 %.

5. CONCLUSIONS

This paper presented a comprehensive study on Task Partitioning and Scheduling of signal processing software on Multicore Processors for complex real-time defence-related applications. The existing scheduling policies, viz, SCHED_FIFO, SCHED_RR, and SCHED_DEADLINE, were compared with Modified-EDF and their effectiveness in managing computationally intensive and memory-bound tasks, which are characteristic of complex real-time signal processing systems, were analysed. Modified-EDF, combined with Intel's pseudo-locking features, demonstrated promising results, significantly reducing WCET and enhancing CPU utilisation while ensuring optimal task schedulability. The integration of cache partitioning and online estimation of execution time resulted in reducing resource contention, and improving schedulability, making Modified-EDF well-suited for high-performance real-time defence applications where efficient resource utilization and strict deadlines are critical. The findings of this study highlight the importance of combining hardware-aware scheduling with adaptive task management to meet the demanding performance and reliability requirements of defence signal processing applications.

REFERENCES

1. Bo, Z.; Qiao, Y.; Leng, C.; Wang, H.; Guo, C. & Zhang, S. Developing real-time scheduling policy by deep reinforcement learning. IEEE 27th real-time and embedded technology and applications symposium (RTAS), Nashville, TN, USA, 2021, 131-142. doi: 10.1109/RTAS52030.2021.00019
2. Bhuiyan, A.; Liu, D.; Khan, A.; Saifullah, A.; Guan, N. & Guo, Z. Energy-efficient parallel real-time scheduling on clustered multi-core. *IEEE Transact. Parallel and Distributed Syst.*, 2020, **31**(9), 2097-2111. doi: 10.1109/TPDS.2020.2985701.
3. Zeng, L.; Xu, C. & Li, R. Partition and scheduling of the mixed-criticality tasks based on probability. *IEEE Access*, 2019, **7**, 87837-87848. doi: 10.1109/ACCESS.2019.2926299
4. Sheikh, S.Z. & Pasha, M.A. Energy-efficient cache-aware scheduling on heterogeneous multicore systems. *IEEE Transact. Parallel and Distributed Syst.*, 2022, **33**(1) 206-

217.
doi: 10.1109/TPDS.2021.3090587.
5. Krishnakumar, A.; Arda, S.E.; Goksoy, A.A.; Mandal, S.K.; Ogras, U.Y.; Sartor, A.L. & Marculescu, R. Runtime task scheduling using imitation learning for heterogeneous many-core systems. IEEE Transact. *Comput.-Aided Design of Integrated Circuits and Syst.*, 2020, **39**(11), 4064-4077. doi: 10.1109/TCAD.2020.3012861.
6. Saranya, N. & Hansdah, R.C. Dynamic partitioning based scheduling of real-time tasks in multicore processors. IEEE 18th International Symposium on Real-Time Distributed Computing, Auckland, New Zealand, 2015, 190-197. doi: 10.1109/ISORC.2015.23.
7. Osmolovskiy, S.; Ivanova, E.; Shakurov, D.; Fedorov, I. & Vinogradov, V. Hierarchical real-time scheduling for multicore systems. *In* 18th Conference of Open Innovations Association and Seminar on Information Security and Protection of Information Technology (FRUCT-ISPIT), St. Petersburg, Russia, 2016, 248-256. doi:10.1109/FRUCT-ISPIT.2016.7561535.
8. Chniter, H.; Mosbahi, O.; Khalgui, M.; Zhou, M. & Li, Z. Improved multi-core real-time task scheduling of reconfigurable systems with energy constraints. *IEEE Access*, 2020 **8**, 95698-95713. doi: 10.1109/ACCESS.2020.2990973.
9. Mok, A.K.; Feng, X. & Chen, D. Resource partition for real-time systems. Proceedings Seventh IEEE Real-Time Technology and Applications Symposium, Taipei & Taiwan, 2001, 75-84. doi: 10.1109/RTTAS.2001.929867.
10. Paul, S.; Chatterjee, N.; Ghosal, P. & Diguet, J.-P. Adaptive task allocation and scheduling on NoC-based multicore platforms with multitasking processors. *ACM Trans. Embed. Comput. Syst.*, 2021, **20**(1). doi: 10.1145/3408324
11. K. Kang, D. Ding, H. Xie, Q. Yin & J. Zeng. Adaptive DRL-Based Task Scheduling for Energy-Efficient Cloud Computing. IEEE Transact. *Network and Serv. Manage.*, 2022, **19**(4), 4948-4961. doi: 10.1109/TNSM.2021.3137926.
12. Ali, I.; Seo, J.-h. & Hoon Kim, K. A dynamic power-aware scheduling of mixed-criticality real-time systems. *In* IEEE International Conference on Computer and Information Technology; Ubiquitous Computing and Communications; Dependable, Autonomic and Secure Computing; Pervasive Intelligence and Computing, Liverpool, UK, 2015. doi: 10.1109/CIT/IUCC/DASC/PICOM.2015.63.
13. Altin, L.; Topcuoglu, H.R. & Gürgen, F.S. Latency-aware multi-objective fog scheduling: Addressing real-time constraints in distributed environments. *IEEE Access*, 2024, **12**, 62543-625557. doi: 10.1109/ ACCESS.2024.3395664.
14. Han, J.-J.; Wang, Z.; Gong, S.; Miao, T. & Yang, L.T. Resource-aware scheduling for dependable multicore real-time systems: Utilisation bound and partitioning algorithm. *IEEE Transact. Parallel and Distributed Syst*, 2019, **30**(12), 2806-2819. doi: 10.1109/TPDS.2019.2926455.
15. Seo, E.; Jeong, J.; Park, S. & Lee, J. Energy efficient scheduling of real-time tasks on multicore processors. *IEEE Transact. Parallel and Distributed Syst.*, 2008, **19**(11), 1540-1552. doi: 10.1109/TPDS.2008.104.
16. Liang, Y.; Li, H.; Shen, F.; Xu, Q.; Hua, S. & Zhu, S. Adaptive multi-core real-time scheduling based on reinforcement learning. *In* IEEE 18th International Conference on Control & Automation (ICCA), Reykjavík, Iceland, 2024, pp. 148-153. doi: 10.1109/ICCA62789.2024.10591927.
17. Digalwar, M.; Gahukar, P. & Mohan, S. Energy efficient real-time scheduling on multi-core processor with voltage islands. *In* International Conference on Advances in Computing, Communications and Informatics (ICACCI), Bangalore, India, 2018, pp. 1245-1251. doi: 10.1109/ICACCI.2018.8554680.
18. Ekberg, P. & Baruah, S. Partitioned scheduling of recurrent real-time tasks. IEEE Real-Time Systems Symposium (RTSS), Dortmund, DE, 2021, 356-367. doi: 10.1109/RTSS52674.2021.00040.
19. Ding, J.; Song, L.; Li, S.; Wu, C.; He, R.; Su, Z. & Lü, Z. A Heuristic method for data allocation and task scheduling on heterogeneous multiprocessor systems under memory constraints. *In* 23rd International Conference on Algorithms and Architectures on Parallel Processing, ICA3PP, Tianjin, China, 2023. doi: 10.1007/978-981-97-0801-7_21.
20. Sun, Z.Y. & Han, W.M. & Gao, L.L. Real-time scheduling for dynamic workshops with random new job insertions by using deep reinforcement learning. *Adv. in Production Engin. & Manage.*, 2023, **18**, 137-151. doi:10.14743/apem2023.2.462.

CONTRIBUTORS

Mr Balakrishnan P. is currently working as an Advisor, at Bharat Electronics Limited and also at Cochin University of Science and Technology. His areas of interest include: Embedded systems/signal processing systems and also working on scheduling techniques for real-time applications.
His contribution towards this research is in the design of the scheduling scheme, algorithm, and analysis as well as in the preparation of the manuscript.

Mr Rajesh M. obtained his PhD from National Institute of Technology (NIT), Meghalaya and working as a Scientist at the NIELIT, Kozhikode. His areas of interest include: Research and development in the areas of Linux internals and real-time operating systems (RTOS) for Embedded applications.
His contribution to this research is in the software implementation of the scheduling schemes and its evaluation.

Dr. Rajesh R. obtained PhD Degree in physics from Jamia Millia Islamia, New Delhi and working at DRDO-NPOL, Kochi. His areas of interest include: Laser-aided plasma diagnostic systems, high-power chemical lasers for directed energy applications and also development of fibre-optic underwater acoustic sensors.
In the current study, he guided the research activity and critically reviewed the manuscript.