An Implementation of Partitioned Scheduling Scheme for Hard Real-Time Tasks in Multicore Linux with Fair Share for Linux Tasks

N. Saranya and R. C. Hansdah
Dept of Computer Science and Automation
Indian Institute of Science, Bangalore, India
Email: {saranya.n, hansdah}@csa.iisc.ernet.in

Abstract—The correctness of a hard real-time system depends upon its ability to meet all its deadlines. Existing real-time systems use either a pure real-time scheduler or a real-time scheduler embedded as a real-time scheduling class in the scheduler of an operating system (OS). Existing implementations of schedulers in multicore systems that support real-time and non-real-time tasks, permit the execution of non-real-time tasks in all the cores with priorities lower than those of real-time tasks, but interrupts and softirqs associated with these non-real-time tasks can execute in any core with priorities higher than those of real-time tasks. As a result, the execution overhead of real-time tasks is quite large in these systems, which, in turn, affects their runtime. In order that the hard real-time tasks can be executed in such systems with minimal interference from other Linux tasks, we propose, in this paper, an integrated scheduler architecture, called SchedISA, which aims to considerably reduce the execution overhead of real-time tasks in these systems. In order to test the efficacy of the proposed scheduler, we implemented partitioned earliest deadline first (P-EDF) scheduling algorithm in SchedISA on Linux kernel, version 3.8, and conducted experiments on Intel core i7 processor with eight logical cores. We compared the execution overhead of real-time tasks in the above implementation of SchedISA with that in SCHED_DEADLINE’s P-EDF implementation, which concurrently executes real-time and non-real-time tasks in Linux OS in all the cores. The experimental results show that the execution overhead of real-time tasks in the above implementation of SchedISA is considerably less than that in SCHED_DEADLINE. We believe that, with further refinement of SchedISA, the execution overhead of real-time tasks in SchedISA can be reduced to a predictable maximum, making it suitable for scheduling hard real-time tasks without affecting the CPU share of Linux tasks.

Keywords—Real-Time Scheduler, Hard Real-Time System, Multicore System, Linux Operating System

I. INTRODUCTION

Performance gain in single core processors is achieved by frequency scaling. However, increasing frequency boosts heat dissipation and power consumption in the single processor chip. The multicore technology came as a solution to this problem. Multicore architecture consists of two or more processing units placed in a single integrated circuit, or a number of such integrated circuits packed together. The multicore processors deliver higher performance, dissipate less heat and consume less power. Today, systems with number of cores ranging from two to 100 [1] are available and the multicore technology has found its application in many products including real-time systems. The real-time systems are classified as hard or soft real-time systems based on the system’s timing constraints. If the correctness of the system depends on meeting all its deadlines, it is called a hard real-time system and if a few deadline misses are tolerated, then it is called a soft real-time system. Multicore systems are being used widely for both hard and soft real-time systems.

Hard real-time capability is realized either at the hypervisor level or at the OS scheduler level. In case of hypervisor-based systems, the RTOS has a higher priority over the non-real-time OS. In such systems, the response times of the non-real-time tasks greatly suffer whenever there is an increased load on the RTOS. Examples of such systems can be found in [2] and [3]. At the OS scheduler level, real-time capability is realized using scheduling class. There are several scheduling classes in the same OS, and each scheduling class has its own scheduling policy. The scheduling classes are prioritized, and the priorities of tasks in a scheduling class are different from those in other scheduling classes. To realize real-time capability, real-time scheduling class is given the highest priority. In such an OS, the execution of other tasks greatly interfere with the execution of real-time tasks, which increases their execution overhead, and makes their runtime unpredictable. Examples of such systems can be found in [4] and [5]. In case of RTOS whose only objective is to execute real-time tasks, real-time scheduling class is the only scheduling class available.

Multicore schedulers implement the real-time scheduling class either as a partitioned scheduling scheme [6]–[8] with separate ready queue for real-time tasks in each core or as a global scheduling scheme [9]–[11] with a single ready queue for all the real-time tasks in the entire system. However, non-real-time tasks continue to be scheduled in all the cores albeit with lower priority. But, the execution of real-time tasks in the presence of non-real-time tasks gets affected due to the execution of interrupt service routines corresponding to hardware interrupts and softirqs belonging to non-real-time tasks, as they also can be executed in any core with priorities higher than those of real-time tasks. Execution overhead of a real-time task is the total time spent by a job of the task executing codes in the kernel which are not related to its function. Hence, scheduling activity corresponding to real-time tasks also contributes to their execution overhead. In order to use such a system for scheduling hard real-time tasks, execution overhead of real-time tasks needs to be brought down considerably. Another disadvantage of such a system is that the response time of non-real-time tasks increases greatly.
as the number of real-time tasks increases.

In this paper, we propose an integrated scheduler architecture, called SchedISA, which aims to ameliorate the above difficulties of scheduling hard real-time tasks in a multicore Linux with a fair share for Linux tasks. The architecture is based on the concept of scheduling class, but it also makes use of the capabilities of modern day multicore processors and other features of Linux. The architecture essentially is as follows.

1) Partition the set of available cores into three sets, viz., a set of real-time cores, a service core, and a set of Linux cores.

2) Redirect all interrupts related to Linux to Linux cores.

Real-time cores execute only real-time tasks and service cores execute tasks which are related to scheduling of real-time tasks. This is achieved by ensuring that jobs of only appropriate tasks are enqueued in the ready queue of each core. Since the hardware interrupts are redirected to Linux cores, and the scheduling of real-time tasks is carried out by the service core, the execution overhead of real-time tasks is expected to reduce considerably. This also allows for the simultaneous execution of real-time and Linux tasks in their respective cores.

SchedISA combines two existing implementation techniques to reduce the execution overhead of real-time tasks. First technique is the concept of vertically partitioning the set of processors into two different groups for handling real-time and Linux tasks separately. This concept has been implemented in [12] and [13], and both these implementations shield the real-time cores from the Linux tasks. Second technique is the concept of service core which takes care of all the scheduling activities of real-time cores. This concept dates back to the days of Spring Kernel [14]. Recently, this concept has been used to implement global scheduling schemes in [15]. In [12] and [13], the concept of service core has not been used. Besides, in [15], there is no vertical partitioning of processors, and the response times of non-real-time tasks is high. The motivation for combining these two techniques comes from the requirements to have a much tighter bound for the execution overhead of real-time tasks and fair CPU share for the Linux tasks. We believe that multicore embedded systems and applications would greatly benefit from these kinds of systems.

In order to test the effectiveness of the proposed scheduler, we implemented P-EDF scheduling algorithm for hard real-time tasks in SchedISA using Linux kernel, version 3.8, on the Intel core i7 processor with eight logical cores. Further, we compared the execution overhead of real-time tasks in the above implementation of SchedISA with that in SCHED_DEADLINE P-EDF’s implementation. The results of the experiment show considerable reduction in execution overhead in our approach compared to that in SCHED_DEADLINE. Our approach also has the advantage that the response times of non-real-time tasks would not get affected by the increased load of real-time tasks.

The rest of the paper is structured as follows. We review the related works in section II. In section III, we provide the basic background of the system model, kernel and hardware features used by SchedISA.

II. RELATED WORK

SCHED_DEADLINE [5], [16] implements earliest deadline first (EDF) algorithm, and it is implemented as a scheduling class in the Linux kernel. However, unlike in SchedISA, this class has a lower priority than Linux real-time scheduling classes.

RTAI [3] is a hypervisor based real-time extension to the Linux kernel which provides hard real time capability, and allows the real-time (RT) and the Linux kernel to coexist simultaneously. In RTAI, RT kernel is given the highest priority and the Linux kernel acts as an idle task for the RT kernel. RTLinux [2] and Xenomai are the two real-time extensions to the Linux kernel that use hypervisor and provide hard real-time capability. Like RTAI, SchedISA can support hard real-time tasks but it does not use hypervisor. Besides, unlike in case of RTAI, it does not reduce the CPU share of Linux tasks as the number of real-time tasks increases.

LITMUS<sup>RT</sup> [17] is a real-time extension to the Linux kernel which provides an experimental platform for assessing multicore real-time scheduling and synchronization algorithms. LITMUS<sup>RT</sup> scheduler framework supports many different real-time scheduling algorithms. The different scheduling algorithms are implemented as plug-in components which are activated at runtime. LITMUS<sup>RT</sup> has been used to test the schedulability and also compare the performance of many multiprocessor real-time scheduling algorithms on different multiprocessor architectures ranging from 4-processor Intel Xeon SMP platform to much larger Sun Niagara multicore platform with 32 logical processors [6], [9], [18] and [19]. However, in LITMUS<sup>RT</sup>, non-real-time Linux tasks are executed only when there are no real-time tasks available for execution.

ExSched [20] is a scheduler framework that enables different real-time scheduling algorithms to be implemented across different OS platforms. ExSched prototype implements many different scheduling algorithms in Linux and VxWorks as external plugins that are not modified across different OS platforms. ExSched uses high priority kernel thread to migrate tasks in interrupt context. SchedISA also uses kernel thread but these kernel threads do not compete with the real-time tasks for processors. ExSched runs the real-time tasks on the Linux scheduler as SCHED_FIFO, and therefore, a Linux real-time task of higher priority will interfere with the execution of ExSched tasks of lower priority in Linux.

III. BACKGROUND

In this section, we first briefly describe the basic task model, and the kernel and hardware features used by SchedISA.

A. System Model

We consider a system of <i>m</i> identical processors. Each task <i>τ</i><sub><i>i</i></sub> is represented using the tuple (<i>C</i><sub><i>i</i></sub>, <i>D</i><sub><i>i</i></sub>, <i>T</i><sub><i>i</i></sub>), where <i>C</i><sub><i>i</i></sub> is the
worst case execution time (WCET), $D_i$ is the relative deadline and $T_i$ is the period of the task. Job is an instance of a task and $j_{ij}$ is the $j$th job of task $r_i$. A job of $r_i$, released at time $t$ has its deadline at $D_i + t$. The utilization, $U_t$, of a task $r_i$ is $C_i/T_i$. Every job of task $r_i$ executes for time less than or equal to $C_i$ and the relative deadline $D_i$ is always greater than $C_i$. Besides, we consider a constraint deadline system, where $D_i \leq T_i$ for all tasks $r_i$.

We consider independent tasks where the execution of one task does not depend on the other. At any instant of time, no two processors can execute the same task. Besides, job $j_{i(j+1)}$ can begin its execution only after the completion of execution of job $j_{ij}$.

### B. Hardware and Kernel components

In this section, we provide some basic information about the hardware and the Linux features that SchedISA uses in its implementation.

1) **Local APIC:** Each logical processor in a multicore system comes with an Advanced Programmable Interrupt Controller referred to as the local APIC. It receives various interrupts from both internal and external sources. The local APIC is also responsible for sending/receiving inter-processor interrupt (IPI) messages to/from other logical processors. The cores in SchedISA use IPI messages to communicate with each other. Besides, each local APIC has a highly accurate clock source, and a set of counters which can be used to realize high resolution timers with nanosecond accuracy.

2) **High Resolution timers (hrtimers):** Hrtimers are independent of the kernel’s periodic tick and can operate at a very high resolution. The hrtimer uses the system’s clock event device to interrupt the system. Clock event device can be programmed to interrupt the system after a certain time interval from the current time. The system’s clock source provides the time at which this clock event occurs. When this interrupt arrives, the hrtimer interrupt handler, called the hrtimer_interrupt, is executed.

3) **IRQ affinity:** In Linux, the interrupt requests (IRQ) can be assigned to a specific core or a group of cores, and this facility is known as SMP IRQ affinity. For every IRQ in the Linux kernel, there is a corresponding directory in /proc/irq/. The processor affinity of an IRQ can be changed by modifying the bitmask present in the smp_affinity_file in its corresponding IRQ directory.

4) **isolcpus:** The kernel configuration isolcpus isolates the specified processors from the Linux scheduler and load balancer. Isolcpu parameter is configured by appending “isolcpus = <cpu1, cpu2,.....,cpum>” to GRUB_CMDLINE_LINUX_DEFAULT line in /etc/default/grub file or by modifying the “isolcpu = ” kernel boot time parameter at the bootloader command line during system boot. A task can be executed on the isolated processor by setting its CPU affinity using the sched_setaffinity system call.

5) **Kthreads:** Kthreads are kernel tasks with no userspace component, and they are created using internal kernel thread API module. On receiving an interrupt, the interrupt handler is executed with other interrupts blocked leading to high interrupt latency. Using kthreads for interrupt processing reduces the amount of time the other interrupts remain blocked, and hence the interrupt latency.

### IV. IMPLEMENTATION

In this section, we give a detailed description of SchedISA. We first describe the overall system architecture, followed by the details of scheduling and the various implementation issues.

### A. System Design

1) **Processor Group:** The set of $m$ cores in the system are divided into three disjoint groups, viz., Linux, real-time and service group. The service group consist of only one core, called the service core. The cores from the real-time and Linux group are called real-time and Linux cores respectively. The service core along with all the cores from the real-time group form a real-time scheduling entity.

The cores in real-time and service group are removed from kernel load balancing and scheduling algorithm by using the isolcpus kernel configuration. IRQ affinity is used to redirect all external interrupts to the Linux cores. However, the IPIs and timer interrupts are not directed away from the real-time and service cores.

The SchedISA real-time tasks are distinguished from the Linux real-time tasks using a different scheduling policy. The term Linux task is inclusive of both Linux’s real-time and non-real-time tasks. SchedISA real-time tasks are scheduled to run only on real-time cores, and are scheduled using P-EDF scheduling algorithm. Although, in this implementation, we have implemented P-EDF scheduling algorithm, SchedISA

<table>
<thead>
<tr>
<th>System Call</th>
<th>Functionality</th>
</tr>
</thead>
<tbody>
<tr>
<td>sched_setparam_real</td>
<td>Changes task’s policy to SCHED_IS</td>
</tr>
<tr>
<td>sched_setparam_linux</td>
<td>Changes SCHED_IS task to Linux task</td>
</tr>
<tr>
<td>sched_do_job_release</td>
<td>Starts executing the real-time job</td>
</tr>
<tr>
<td>sched_do_job_complete</td>
<td>Completes execution of real-time job</td>
</tr>
<tr>
<td>sched_task_complete</td>
<td>Completes execution of real-time task</td>
</tr>
<tr>
<td>sched_compl_ipi_func</td>
<td>Callback function for completion IPI</td>
</tr>
<tr>
<td>sched_do_hrtimer</td>
<td>Callback function for release hrtimer</td>
</tr>
</tbody>
</table>

**TABLE I:** System calls in SchedISA
scheduler architecture, in general can implement any scheduling algorithm with some modification. Linux’s tasks are scheduled to run only on Linux cores. Unlike other implementation, SchedISA allows the real-time and Linux tasks to execute simultaneously in their respective groups. The service core manages the scheduling of real-time tasks that execute on the real-time cores.

The number of cores allocated to the real-time and non-real-time tasks depends on the application requirement, and also on the number of real-time cores that a service core can handle without affecting the system’s schedulability. The experimental evaluation in this paper is done with one service core, one (to three) Linux core(s), six (to four) real-time cores forming one real-time scheduling entity with six (to four) real-time cores and one service core.

2) Queue Structure: The real-time scheduling entity contains three different types of queue data structure. There is one global release queue, one per core ready queue and one per core release queue for every real-time core in the real-time scheduling entity. The global release queue holds the yet to be released jobs. The global release queue is local to the service core, i.e., all queue operations on it are performed by the service core. The per core ready queue of a real-time core holds all runnable tasks that are waiting for execution on that core. The per core release queue of a real-time core, called the temporary release queue, holds all those tasks that have completed their execution in the current period and are waiting to be moved to the global release queue. Queue operations on ready and temporary release queues are performed by the core that hosts the queues and the service core. Locks are used to prevent simultaneous access. All the queue data structures are implemented using red-black tree. The ready queue is sorted in decreasing order of priority with the leftmost node having the highest priority. The global release queue is sorted in increasing order of release time with the leftmost node in having earliest release time. These data structures are as shown in figure 1.

3) Task Construction: The worst case execution time, period and deadline of a SchedISA real-time task are known prior to its execution. All SchedISA real-time tasks start as non-real-time tasks. These non-real-time tasks with SchedISA real-time segment are initially started on the Linux cores and then migrated to the service core where they change their policy to the one implemented in SchedISA using a system call. Once in the service core the real-time segment of these tasks are scheduled to run in the real-time cores. While the tasks execute their real-time segment, they run exclusively on the real-time core. A SchedISA real-time task exits its real-time segment by making a system call and is moved to the service core where, depending upon its requirement, it either completes its execution or is migrated to one of the Linux cores. Hrtimers are used to program the task’s release at its release times.

Every SchedISA task is associated with a hrtimer. When the job of a task begins its execution, it first programs its hrtimer to expire at the task’s next release time. When the jth job, $j_t$ of task $\tau_t$ completes its execution, $\tau_t$ suspends itself and waits for the release of job $j_t(j+1)$.

B. Scheduling Details

In this section, we explain how a SchedISA task is scheduled within the real-time scheduling entity. The implementation of SchedISA is obtained from the implementation of SCHED_DEADLINE by making the necessary modifications and including additional system calls. The SchedISA scheduler architecture implements a scheduling class, SCHED_IS, which has the highest priority amongst all the other scheduling classes in the Linux kernel. A summary of the new system calls and modifications made in the kernel are given in table I. In P-EDF scheduling, we assume that assignment of tasks to processor is done prior to scheduling.

1) Task Creation Phase: As discussed in the task construction section, a SchedISA real-time task is first started as a non-real-time task in the Linux core and is then migrated to the service core. In the service core, it makes two system calls. The first system call sched_setparam_real is responsible for changing the task’s policy to SCHED_IS. The task’s WCET, period and deadline are passed as arguments to this function and are included in the task’s structure. The system call sched_do_job_release immediately follows the sched_setparam_real. CPU affinity and the release time of the task’s first job are passed as arguments to this function. sched_do_job_release sets up an hrtimer to expire at the mentioned release time and sets the task’s CPU affinity in its task’s structure. The sched_do_job_release system call then removes the task from the service core’s ready queue, puts it in the global release queue and suspends the execution of the task.

When the sched_setparam_real function is called for the first time, i.e., the first SchedISA task is launched, two Linux real-time kernel threads temp_to_global and global_to_ready are created. On their creation, these threads are bound to the service core and are put to sleep. These threads have SCHED_FIFO policy and are scheduled to run with the highest real-time priority in the service core.

2) Job Release Phase: When a task’s release hrtimer expires on the service core, the release hrtimer handler wakes up the global_to_ready service thread, sets the task’s timer_expired flag, sets the global global_timer_expired flag to true and returns. The service thread executing on the service core sets the global_timer_expired flag to false, moves all the those tasks whose timer_expired flag is set to true from the global release queue to the real-time core’s ready queue based on the task’s affinity. At the end of the above process, it checks the global_timer_expired flag again, and if it is set to true, it loops through the process again, otherwise, it suspends its execution. The release hrtimer interrupt handler can set the global_timer_expired flag to true while global_to_ready service thread executes. After placing the task in the ready queue, the global_to_ready service thread checks to see if the new task is of higher priority than the current task. If it is of higher priority, the service core sends a reschedule IPI to the real-time core. On receiving the reschedule IPI from the service core, the real-time core preempts the current task and schedules the next highest priority task in its ready queue to run.

Algorithm 1 illustrates global_to_ready thread. The
Algorithm 1

```c
function global_to_ready_thread_function(void) do
    while global_timer_expired ≠ 0 do
        global_timer_expired ← 0;
        i ← 0;
        lock globrelq;
        nextTask ← globrelq.leftmost;
        while true do
            if nextTask.timer_expired ≠ 0 then
                task[i] ← dequeueTask(globrelq);
                i ← i + 1;
                nextTask ← nextTask.next;
            else
                break;
            end if
        end while
        unlock globrelq;
        k ← 0;
        while k < i do
            t ← taskaffinity(task[k])
            lock rcpu[t].readyq;
            enqueueTask(rcpu[t].readyq, task[k]);
            k ← k + 1;
            if rcpu[t].currentTask.pri < task[k].pri then
                send resched IPI to rcpu[t]
            end if
            unlock rcpu[t].readyq;
        end while
    end function
```

The global release queue is represented by the `globrelq`. `globrelq.leftmost` is global release queue’s left-most task and has the earliest release time. `nextTask.next` is the task with next earliest release time after task, `nextTask`, in the global release queue, `nextTask.timer_expired` is the task’s timer expired flag, `rcpu[i]` is the `i`th real-time core in the real-time scheduling entity, `rcpu[i].readyq` is `rcpu[i]`’s ready queue, `rcpu[i].currentTask` is the task executing on `rcpu[i]`, `rcpu[i].currentTask.pri` is the priority of `rcpu[i].currentTask`, `task[i].pri` is the priority of `task[i]`. The function `taskaffinity(task)` returns the CPU affinity of the task. The function `enqueueTask(queue, task)` adds task to queue. The function `dequeueTask(queue, task)` removes the left-most task from the queue.

3) Job Completion Phase: A real-time job completes execution by calling `sched_do_job_complete` system call. The `sched_do_job_complete` system call executing on the current real-time core removes the current task from its ready queue, adds it to its temporary release queue, notifies the service core by sending a completion IPI, suspends it and schedules the next highest priority task to execute.

The real-time core sets the `tasks_in_temp` per core flag to true after it enqueues a task in its temporary release queue. On receiving the completion IPI, the IPI’s interrupt handler executing in the service core wakes up the `temp_to_global` service thread and returns. This service thread checks the `tasks_in_temp` flag of all the real-time cores in the scheduling entity. If it is set for a real-time core, it acquires the lock on the core’s temporary release queue, removes all the tasks from this queue, adds them to the global release queue and sets `tasks_in_temp` flag to false.

Algorithm 2 illustrates `temp_to_global_thread` function thread. The global release queue is represented by the `globrelq`, `rcpu[i]` is the `i`th real-time core in this real-time scheduling entity, `rcpu[i].releaseq` is `rcpu[i]`’s temporary release queue, `rcpu[i].releaseq.ntasks` is the number of tasks in `rcpu[i]`’s temporary release queue, `rcpu[i].tasks_in_temp` is a per core flag which is set when temporary release queue contains some tasks.

4) Task Completion Phase: A task executing in the real-time core exits its real-time segment by making `sched_task_complete` system call which migrates the task to the service core’s ready queue.

A SCHED_IS task calls `sched_setparam_linux` function to change its scheduling policy from SCHED_IS. In SchedISA, when the task changes its policy from SCHED_IS, the `sched_setparam_linux` system call migrates the task from the real-time scheduling entity to the Linux cores. Tasks are allowed to change their policy from SCHED_IS to other Linux policy and back again using `sched_setparam_linux` and `sched_setparam_real` system call respectively.

C. Other Issues

In this section, we discuss the various implementation issues and the means to handle them.

1) Service Core: The concept of using a dedicated processor for handling interrupts to release jobs was experimented by Brandenberg et al. in [9]. They used this concept for reducing

Algorithm 2

```c
function temp_to_global_thread_function(void) do
    i ← 0;
    while i < r do
        if rcpu[i].tasks_in_temp ≠ 0 then
            lock rcpu[i].releaseq;
            j ← 0;
            while j < rcpu[i].releaseq.ntasks do
                task[j] ← dequeueTask(rcpu[i].releaseq);
                j ← j + 1;
            end while
            unlock rcpu[i].releaseq;
            lock globrelq;
            k ← 0;
            while k < j do
                enqueueTask(globrelq, task[k]);
                k ← k + 1;
            end while
            unlock globrelq;
            i ← i + 1;
        end if
    end while
    sleep
end function
```
execution overhead of real-time tasks in global scheduling algorithm, whereas, in SchedISA, we try to reduce execution overhead of real-time tasks by redirecting all interrupts that are not required in the real-time cores to service core and Linux cores as needed. Besides, in SchedISA, the service core is used for executing the non-real-time section of the SchedISA tasks, sending resched IPI to the real-time cores, receiving completion IPI from the real-time cores, handling the IPI’s interrupt handlers, executing the hrtimer callback function and service threads. The interrupt handler executes with all other IPIs blocked. To reduce this blocking time, SchedISA employs a threaded approach to interrupt handling.

Scheduling other processes on the service core will not affect the schedulability of the real-time tasks because the highest priority service threads will preempt the current process. Besides, interrupt handlers will always execute in the context of the current process.

2) Real-time Cores: Apart from executing SchedISA real-time jobs, the real-time cores send completion IPI to the service core, performs dequeue and enqueue operation on its ready and temporary release queue respectively.

3) Service Thread: The two kernel threads global_to_ready and temp_to_global have the highest real-time priority on the service core. However, they can be blocked during the execution of IPI and hrtimer interrupt handlers.

As discussed earlier, the temp_to_global thread moves all the SchedISA tasks from all the temporary release queue to the global release queue. When woken up, the global_to_ready service thread dequeues all those tasks with timer_expired flag set from the global release queue and moves them to their respective ready queue. We use this method of moving all the available tasks instead of moving them individually to increase the system throughput.

4) Hrtimer: The task’s release hrtimer is set to expire on the service core on its first release. On subsequent release, it is set on the real-time core and migrated to the service core when the job completes. If a job completes its execution close is set on the real-time core and migrated to the service core on its first release. On subsequent release, it moves the task from the real-time core’s temporary release queue to its ready queue. An alternate design would be to set the task’s release hrtimer to expire on the real-time core completes, the real-time core moves the task from its ready queue to its temporary release queue and sends an IPI to the service core. Job completion overhead is the time taken by the real-time core to perform this process.

d) Job Completion Overhead: When a job executing on a real-time core completes, the real-time core moves the task from its ready queue to its temporary release queue and sends an IPI to the service core. Job completion overhead is the time taken by the real-time core to perform this process.

e) Completion IPI Latency: It is the time elapsed from the generation of the completion IPI in the real-time core, to the start of its callback function.

f) Job Dequeue Overhead: The time elapsed from the generation of completion IPI on the real-time core to time when the task is enqueued in the global release queue is the job dequeue overhead.

g) Preemption Cache Overhead: When a preempted job resumes its execution, the time taken to restore its memory context on the cache is the preemption cache overhead.

h) Context Switching Latency: The time taken to acquire the lock on the temporary release and ready queue to initiate the context switching of tasks is the context switching latency.

i) Context Switching Overhead: This is the actual time taken to perform context switching between two tasks.

j) Context Switching Time: The sum of context switching overhead and context switching latency is the context switching time.

k) TLB Overhead: The time taken by the CPU to perform the required TLB operation on receiving a TLB interrupt from another CPU is the TLB overhead.

l) TLB Latency: On receiving a TLB interrupt, the interrupt handler makes the necessary changes in the core’s TLB and sends an acknowledgment of the same to the CPU that sent the TLB interrupt. The amount of time the CPU generating the TLB IPI spends waiting for this acknowledgment from all the cores to which it sent the IPI is the TLB latency.

V. Runtime Overhead

This section discusses the various overheads and latencies in SchedISA.

a) Job Release Overhead: When job’s release hrtimer expires on the real-time core, the time taken by the real-time core to process the hrtimer’s interrupt and enqueue the task in its ready queue is the job release overhead.

b) Release Timer Latency: It is the time elapsed from the generation of the release hrtimer interrupt to the start of its callback function.

c) Job Enqueue Overhead: The time elapsed from the hrtimer’s expiry on the service core to the completion of the task’s enqueue operation on the real-time core’s ready queue by the service thread is the job enqueue overhead.

d) Job Completion Overhead: When a job executing on a real-time core completes, the real-time core moves the task from its ready queue to its temporary release queue and sends an IPI to the service core. Job completion overhead is the time taken by the real-time core to perform this process.

f) Job Dequeue Overhead: The time elapsed from the generation of completion IPI on the real-time core to time when the task is enqueued in the global release queue is the job dequeue overhead.
m) Job Execution Overhead: It is the total time spent by a job performing tasks inside the kernel that are not related to its function. In SchedISA, job enqueue and dequeue overhead do not contribute to the job execution overhead. However, a large job enqueue and dequeue overhead can cause delays in release of real-time jobs, which can lead to deadline misses if overheads are not accounted for appropriately.

VI. EXPERIMENTAL EVALUATION

In this section, we demonstrate the runtime performance of SchedISA. The experiments are conducted on Intel’s core i7 3632-QM platform. It is a quad core processor running at 2.2 GHz clock speed with 8 logical cores, i.e., two threads per physical core. The execution overhead of real-time tasks in the P-EDF’s implementation of SchedISA is compared with that in SCHED_DEADLINE implementing P-EDF scheme.

TABLE II: Number of TLB interrupts handled by the core

<table>
<thead>
<tr>
<th>CPU</th>
<th>SchedISA</th>
<th>SCHED_DEADLINE</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>0</td>
<td>68</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>10</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>4951</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>942</td>
</tr>
</tbody>
</table>

A. TLB shootdowns

In this section, we report the number of TLB interrupts handled by the real-time cores in SchedISA and SCHED_DEADLINE.

1) Experimental Setup: Our scheduling entity in SchedISA consists of four real-time cores and one service core. The periodic tasks are randomly generated and partitioned among four cores in both the implementations. The real-time tasks were executed in four cores for twelve hours under a similar system load.

We used the /proc/interrupt file to calculate the number of TLB shootdowns handled by a core during the execution of the real-time tasks. The /proc/interrupt file lists the number of interrupts of each type, which includes both external interrupts from other devices and internal interrupts handled by the core since system boot.

2) Results: Table II gives the number of TLB shootdowns recorded over twelve hours run. In both SchedISA and SCHED_DEADLINE, each real-time tasks is bound to one of the four cores and is not migrated. In SchedISA, the Linux tasks execute only on Linux cores while the SchedISA real-time tasks execute on the real-time cores. Since the real-time cores are isolated using isolcpus, and within these cores, P-EDF algorithm does not allow for migration of tasks, the TLB interrupts handled by the real-time cores is zero. In SCHED_DEADLINE, though the P-EDF algorithm does not allow for migration of SCHED_DEADLINE tasks, the Linux tasks executing in these cores contribute to the TLB shootdown.

B. Job Enqueue Overhead and Job Dequeue Overhead

In this section, we determine the job enqueue and dequeue overhead which are unique to SchedISA. These two overheads can be critical to the schedulability of the system under certain conditions. We determined these overheads by timestamping the SchedISA implementation code.

1) Experimental setup: The real-time scheduling entity in this experiment consists of five real-time cores and one service core. The job enqueue and job dequeue overhead depends on the number of real-time tasks executing in the real-time scheduling entity. We randomly generated tasks in which each task has a utilization in the range of [0.1, 0.9]. We divided these tasks randomly into many different groups such that the number of tasks in each group is in the range [10, 80] in increments of 10. Each group constitutes a task set and we conducted experiment on each of the task set. 100 different task sets were generated for each increment of 10 in the given range, and the average and the maximum overhead value were computed from the execution of thousand jobs for each task.

2) Results: Figure 2 and figure 3 show the results of the experiment, as maximum and average job enqueue and job dequeue overhead respectively. We see that the maximum values of both job enqueue and job dequeue overheads are in the range of tens of microseconds, and hence the maximum delay incurred by a job from the time of its release to the time it is enqueued in the ready queue is small and does not affect the schedulability of the task. It is to be noted that these overheads are borne by the service core, and do not add to the execution time of real-time tasks in the real-time cores. The job enqueue and dequeue overhead tend to increase as the number of tasks increases, and hence, if the real-time cores are heavily loaded, there are chance of deadline misses if these overheads are not accounted for appropriately.

From the two figures, we also observe that as the number of tasks increases, there is a small decrease in the average job dequeue and enqueue overhead. The temp_to_global service thread moves all tasks from all the temporary release queues in the real-time scheduling entity to the global release queue. As the number of real-time tasks being executed increases, the probability of temp_to_global service thread moving a task to the global release queue before its IPI is handled in the service core increases. Since the task is moved to the global release queue before its IPI is handled, its job dequeue overhead gets reduced. This is the reason

![Fig. 2: Job enqueue overhead of task sets with different number of tasks](image-url)
for the decreasing trend in the average job dequeue overhead as the number of tasks increases. Similarly, when a release hrtimer expires, it sets the task’s timer_expired flag. The global_to_ready service thread checks this per task flag as it loops through the global release queue and if set, it moves the task to the real-time core based on the task’s affinity. Hence, there is a small decrease in the average job enqueue overhead.

C. Execution Overhead

In this section, we show the evaluation of how much extra time a task spends inside the kernel because of scheduling and other unrelated activities, assuming that a task needs to execute only in the user space. This extra time is the execution overhead for the real-time tasks. The total CPU time of a process is the sum of its user time and system time. User time is the time spent executing the code in user space and system time is the time spent executing the code in kernel space. To see the effect of interrupts and softirqs on the execution overhead, we compare the total CPU time of real-time tasks in SchedISA and SCHED_DEADLINE.

1) Experimental Setup: The real-time scheduling entity in this experiment consists of five real-time cores and one service core. For a given system utilization, we randomly generated tasks in which each task has execution time in the range [10ms, 70ms], starting from 10ms in increments of 10ms, and having period in the range [50ms, 100ms], until the required CPU utilization is reached. The tasks so generated constitute a task set for a particular utilization. Hundred such task sets were generated for each system utilization in the range of [10%, 80%] in steps of 10%. Each task in a task set execute thousand jobs. The actual execution time of tasks in the user space is realized using a calibrated for-loop. The CPU time for the jobs is obtained using Linux getrusage command. Experiments were conducted on each task set to calculate the CPU time of each job. For each task set, the experiment was performed under both high and normal system load. High system load was realized by running cat /dev/zero > /dev/null bash command and other Linux programs.

2) Results: For each of the system utilization considered, experiments were performed on each of its hundred task set. From amongst the many CPU time values obtained for a given execution time and a given system utilization, the CPU time of the job having the maximum value was computed. Figure 4(a) and 4(b), plots the maximum CPU time of SchedISA and SCHED_DEADLINE tasks under normal and high system load respectively under 50% system utilization. We observe that the CPU time of real-time tasks in SCHED_DEADLINE is greater than that in SchedISA in both figure 4(a) and 4(b). In SCHED_DEADLINE, real-time tasks are executed concurrently with the non-real-time tasks in the same set of cores, and the interrupts and softirqs associated with these non-real-time tasks adds to real-time task’s CPU time. However, in SchedISA, the real-time cores execute only the real-time tasks, the interrupts associated with non-real-time tasks are redirected to the Linux cores, and the service core performs the real-time task’s scheduling activities. As a result, the execution overhead of real-time tasks in SchedISA, and hence, CPU time of tasks in SchedISA is lesser than that in SCHED_DEADLINE. However, there is still some execution overhead of real-time tasks in SchedISA as softirqs can still execute in any core. With increased system load, there is an increase in the execution overhead of real-time tasks for both SchedISA and SCHED_DEADLINE as shown in figure 4(b), since there is a corresponding increase in interrupts and softirqs in the real-time cores.

Figure 5(a) and 5(b) illustrate the maximum CPU time of tasks executed under 30% and 70% system utilization in SchedISA under a normal and high system load. From the figures, we observe that CPU time of SchedISA task is slightly greater than the actual execution time. This is due to the execution of the softirqs in the real-time scheduling entity. This deviation of the CPU time from actual execution time in SchedISA increases as the system load increases. This is because with the increase in system load, the softirq load also increases. Isolating the real-time scheduling entity from softirqs is left open for future work.

VII. CONCLUSIONS AND FUTURE WORK

In this paper, we present an integrated scheduler architecture, called SchedISA, for scheduling hard real-time tasks in Linux. A basic objective of the framework is to restrict execution overhead of real-time tasks, and thereby improve predictability. The division based design reduces the execution overhead in the real-time cores and guarantees a fair share of CPU time for the Linux tasks. Experimental results from the implementation of partitioned scheduling scheme show that we are indeed able to considerably restrict the execution overhead. However, as the experimental results show, the softirqs continue to be executed in the real-time cores, and they also make a small contribution to the execution overhead of real-time tasks, our future works include isolating the real-time scheduling entity from softirqs and reducing execution overheads even when tasks are migrated.

ACKNOWLEDGMENT

This work is partially supported by the Joint Advanced Technology Programme of Indian Institute of Science, Bangalore, India, under a research grant on ”Real-Time Scheduling Schemes for Multicore Operating Systems” (JATP 133, IISc).

REFERENCES

DEADLINE tasks with execution time in [10ms, 70ms].

Fig. 4: CPU time against actual execution time of SchedISA vs SCHED_DEADLINE tasks with execution time in [10ms, 70ms].

Fig. 5: CPU time against actual execution time of SchedISA tasks with execution time between [10ms, 70ms].


