pCOMPATS: Period-Compatible Task Allocation and Splitting On Multi-Core Processors

Arvind Kandhalu, Karthik Lakshmanan, Junsung Kim, Ragunathan (Raj) Rajkumar
Department of Electrical and Computer Engineering
Carnegie Mellon University
Pittsburgh, PA 15213, USA
{akandhal, klakshma, junsungk, raj}@ece.cmu.edu

***Abstract*—Extensive research is underway to build chips with potentially hundreds of cores. In this paper, we consider the problem of scheduling periodic real-time tasks on multi-core processors. We develop a task partitioning algorithm called Period-COMPatible-Allocation and Task-Splitting (pCOMPATS) for fixed-priority scheduling of preemptive hard real-time tasks where the utilization of each of the tasks is less than 50%. pCOMPATS clusters *compatible* tasks together with task splitting to improve the achievable utilization. We show that as the number of cores increases, the least upper bound on schedulable utilization achieved using pCOMPATS approaches 100% per core. To the best of our knowledge, this is the first result that shows that the utilization bound improves as the number of processing cores in the system increases. We refer to tasks having utilization greater than or equal to 50% as heavy tasks and provide a task-partitioning algorithm called pCOMPATS-HT for allocating such tasks. We show that the upper bound on schedulable utilization when tasks are scheduled using pCOMPATS-HT is at most 72%. We also evaluate the performance of pCOMPATS and other well-known partitioning techniques, and show that using pCOMPATS provides much better schedulable utilization in the average case. We characterize the overhead of pCOMPATS using measurements on an Intel Core i7 processor running Linux/RK. The overheads are seen to be low on the platform, making pCOMPATS to be practical. Our results are especially useful in the context of future many-core processors with dozens to hundreds of cores per processor.**

I. Introduction

Multi-core processors that offer multiple processing cores on a single chip are quickly emerging as the dominant technology in the microprocessor industry. Major chip makers including AMD, IBM and Intel now have processors with 2, 4, 8 or 12 cores [8, 24]. Further, extensive research is underway to build chips with potentially hundreds of cores [29]. A major issue with multi-core technology however is in developing parallel software to effectively utilize the available processing power. Multi-core processors largely resemble symmetric multiprocessors (SMPs). Therefore, existing software experience with SMPs may prove useful. In multi-core processors, communication and coordination latencies are significantly reduced as the cores are co-located on the same chip. Also, shared levels in the cache hierarchy present in many modern processors facilitate inter-core data sharing. Software developed for multi-core processors should exploit these architectural features to achieve better performance. In this paper, we consider the problem of scheduling periodic real-time tasks on multi-core processors to better utilize their parallel-processing capability.

Real-time scheduling on multi-core processor systems is a well-studied problem. The scheduling algorithms developed for this problem are classified as partitioned (static binding) and global (dynamic binding) approaches, with each category having its own merits and de-merits. Recently, semi-partitioned scheduling approaches have been proposed where the taskset is partitioned, but some tasks migrate between selected processing cores at run-time.

In this paper, we focus on the semi-partitioned approach, and also assume that uniprocessor fixed-priority preemptive scheduling schemes are used. Our proposed algorithm does not split more than one task per core, and therefore minimizes any penalties of task-splitting. Our proposed algorithm called Period-COMPatible-Allocation and Task-Splitting (pCOMPATS) clusters *period-compatible* tasks together with task splitting. pCOMPATS considers tasks whose individual utilization is less than 50%. We show that as the number of processor cores increases, the least upper utilization bound on schedulable utilization achieved using pCOMPATS approaches 100% per core. We propose pCOMPATS-HT for allocating "heavy" tasks of utilization ≥ 0.5. We show that the upper bound on schedulable utilization using pCOMPATS-HT is at most 72%. To the best of our knowledge, this is the first result that shows that the utilization bound improves as the number of processing cores in the system increases. Multi-core processor scheduling approaches proposed so far perform *independent of the number of processor cores*. Our result is especially useful in the view of the fact that many chip makers already have processors with 4 or 8 cores and are planning to add substantially more processing such as Intel's 48-core Single-chip Cloud Computer (SCC) [13]. In order to characterize the practical overheads of pCOMPATS, we will present a case study using the Intel Core i7 processor running Linux/RK [28, 26, 27]. We study the cache overheads due to task migration and preemption using synthetic workloads and a media player application called *Mplayer*. We show that under realistic overheads, pCOMPATS is a practical mechanism of improving the overall system utilization in real-time multi-core scheduling.

II. Related Work

The best-known utilization bound of global fixed-priority scheduling is 38% [3], which is lower than the best-known result of partitioned fixed-priority scheduling of 50% [6].

Recently, semi-partitioned scheduling approaches which can exceed the maximum utilization bound of 50% of the partitioned scheduling have been proposed [19, 12]. Semi-partitioned scheduling has been studied with both dynamic priority scheduling [2, 7, 4, 5, 14, 16, 18, 10] and fixed priority scheduling [17, 15, 19, 12]. In semi-partitioned scheduling, most tasks are statically assigned to one fixed processor as in partitioned scheduling, while a few tasks are split into two subtasks assigned to different processors each. Task-splitting takes advantage of the co-located nature of the processing cores to increase the overall system utilization. Lakshmanan et al. [19] proposed the algorithm PDMS-HPTS-DS, which can achieve the worst-case utilization bound of 65%, and can achieve the bound of 69.3% for a special type of task sets only containing "light" tasks. Guan et al. [12] proposed an algorithm that assigns tasks in decreasing period order, and always selects the processor with the least workload assigned so far among all processors, to assign the next task. Their algorithm can achieve the Liu and Layland's utilization bound [12, 23]. For an exhaustive summary of research work related to utilization bounds on partitioned scheduling, please refer to [11].

In this paper, we propose a task-partitioning algorithm called pCOMPATS that allocates compatible tasks onto the same processing cores and splits one task per core to improve the achievable utilization. In order to cluster period-compatible tasks, pCOMPATS uses the R-BOUND [20]. We show that, under pCOMPATS, the schedulable utilization can reach up to 100% as the number of cores increases. In the average case, it goes up to 92% for only four cores and up to 99% for 32 cores.

III. Notation and Background

The multiprocessor platform consists of m processors $M_1, .., M_m$. We consider a set of independent periodic hard real-time tasks $\Gamma = \{\tau_1, \tau_2, ..., \tau_n\}$, where n is the number of tasks. Each task τ_i is given by $\{C_i, T_i, D_i\}$, where C_i is the worst-case execution time, T_i is the period, and D_i is the relative deadline from arrival time. We assume in this paper that $T_i = D_i$, i.e. we consider only implicit deadline tasks. The utilization of a task $\tau_i \in \Gamma$ is given by C_i/T_i. The utilization of a taskset assigned to a processor M_p is given by $U(p)$. The utilization bound for a processor M_p is given by $U_{bound}(p)$. In this paper, we will use the terms cores, processors and processor cores interchangeably.

A semi-partitioned scheduling algorithm consists of two parts: the partitioning algorithm and the scheduling algorithm. The partitioning algorithm determines if and how each of the tasks is split, and allocated to a fixed processor. The scheduling algorithm determines how the tasks allocated on a processor are scheduled. In this paper, we assume Rate-Monotonic Scheduling (RMS). With the partitioning algorithm, most tasks are assigned to a processor and execute on this processor at run-time. We call these tasks *non-split* tasks. Under our proposed algorithm, pCOMPATS, all but one task is non-split per processor core and therefore we reduce any penalties of task splitting. The other task is called a *split task*. Each subtask of split task τ_i is assigned to a different processor, and the sum of the execution time of all subtasks equals C_i. The subtasks of a task need to be synchronized to execute correctly. For example, if a task τ_i is split into two subtasks $\tau_i^{'}$ and $\tau_i^{''}$, then $\tau_i^{''}$ can be released only after the completion of $\tau_i^{'}$.

IV. Task Compatibility & Splitting

Compatible tasks are tasks that achieve a high processor utilization when scheduled together on the same processor. In [22], tasks that have periods closest to a power of two are considered to be compatible. Harmonic tasks are said to be perfectly compatible since they can reach 100% utilization with RMS. Consider the following task-set: $\tau_1 = (4, 10)$, $\tau_2 = (6, 14)$, $\tau_3 = (4, 10)$, $\tau_4 = (6, 14)$. Here, tasks τ_1 and τ_3 are said to be more compatible than τ_1 and τ_2 as bundling τ_1 and τ_3 yields a utilization bound of 1.0 while bundling τ_1 and τ_2 yields a utilization bound of 0.828. One way of achieving a higher utilization bound is to allocate *period-compatible* tasks on to the same processor [20, 6], and using task-splitting approaches [19].

As mentioned earlier, allocating compatible tasks to the same processor increases the minimum utilization bound of the task set assigned to that processor. R-BOUND [20] is a uniprocessor schedulability test which exploits such task compatibility. For the benefit of the reader, we repeat the R-BOUND test below.

For a set of n implicit-deadline tasks ordered by increasing periods, where the minimum period T_1 and maximum period T_n are fixed and known, and the ratio of any two periods is less than 2, the task set is guaranteed to be schedulable under RMS if the processor utilization is less than or equal to $B(r_p)$.

$$B(r_p) = (n-1)(r_p^{1/(n-1)} - 1) + 2/r_p - 1 \qquad (1)$$

where $r_p = T_n/T_1$ is the ratio of the maximum and minimum periods of tasks assigned to processor M_p. This bound $B(r_p)$ is a decreasing function of n, reaching its minimum value given by

$$\lim_{n \to \infty} B(r_p) = \ln r_p + 2/r_p - 1 \qquad (2)$$

Equation 1 assumes that the ratio of any two periods is less than 2. In order to satisfy this assumption for arbitrary tasksets, a "Scale Task Set" (STS) transformation is provided in [20].

Task splitting is an approach to achieve a higher utilization bound under multi-core processors as observed in [31, 15, 7, 4, 12]. [19] describes an algorithm, where tasks are allocated in the decreasing order of size (C_i/D_i), and the highest-priority task in each processor may be split to achieve a utilization bound of 65%. The motivation for splitting comes from the fact that the classical bin-packing problem is easy to solve when any object is permitted to be split into two pieces, and the sizes of split pieces add up to the size of the object. When an object does not fit into a bin, it is split so that the current bin is exactly filled with one piece, and the residual part is placed in the next bin. This continues with each bin being filled to capacity, and containing at most one additional split per bin. The final

Algorithm 1 $\Gamma' = PeriodTransform(\Gamma)$

```
Input: $\Gamma$ Output: $\Gamma'$
$T_{min} = min_{\forall \tau_i \in \Gamma} T_i$
for $\forall \tau_i \in \Gamma$ do
    $R = \lfloor \frac{T_i}{T_{min}} \rfloor$
    $T'_i = T_i / R$
    $C'_i = C_i / R$
return $\Gamma'$
```

bin will hold the remaining objects without further splitting. Since every bin is completely filled before a new bin is added, both first-fit and best-fit packing decreasing schemes yield the same result. When real-time tasks are considered, however, there are two issues that need to be considered: (i) each (bin) processor cannot be filled up to 100% utilization since deadlines can be missed at lower utilization under rate-monotonic scheduling, and (ii) the penalty of splitting from the utilization perspective is non-zero in practice. We design an algorithm called Period COMPatible Allocation and Task-Splitting (pCOMPATS) that combines the desirable properties of R-BOUND-MP-NFR and Highest Priority Task Splitting (HPTS) [19].

V. Period COMPatible Allocation and Task-Splitting (pCOMPATS)

The pCOMPATS algorithm transforms a given taskset to satisfy the period requirements of R-BOUND, and uses task splitting to reduce wasted cycles in each processor. In our discussion, we restrict our attention to tasksets, where the individual tasks have a utilization lesser than 50%. For tasks with utilization greater than 50%, the utilization overhead of task splitting is quite high as observed in [19], and these situations can be treated as a special case since no more than two tasks in this taskset will fit together in a processor. We will discuss heavy tasks in Section 5.5.

A. Period Transformation

In [20], the authors scale each taskset Γ using the STS transform so that, $1 \leq r = \frac{max_{\tau_i \in \Gamma} T_i}{min_{\tau_i \in \Gamma} T_i} < 2$ is satisfied. In our algorithm, we use the *period transformation* technique to achieve the same effect. Period transformation is a technique originally developed to provide graceful degradation under overload conditions by modifying the workload so that higher criticality tasks have higher scheduling priorities [30]. The period-transformation algorithm that we use in this paper is given in Algorithm 1. Henceforth, we will assume that the taskset has been transformed using Algorithm 1 and the scheduler performs run-time slicing to realize the transformed periods.

In order to apply task splitting in the R-BOUND context, we now define a complement of the Highest-Priority Task-Splitting (HPTS) [19] called Lowest-Priority Task-Splitting (LPTS). Under HPTS, if a task on a processor is to be split, the highest priority task τ_h on the processor is always chosen as the candidate for splitting. Let the split instances of the highest-priority task τ_h on a processor of interest be τ'_h and τ''_h, with τ'_h being scheduled on the same processor. The property that is exploited by HPTS is that the highest priority task τ'_h on a processor under fixed-priority scheduling has its worst-case response time equal to its worst-case computation time. The deadline of the second subtask τ''_h is therefore maximized. Please refer to [19] for more details. LPTS can leverage the property that if tasks are allocated in a non-decreasing order of periods, then splitting the lowest priority task on a processor results in a split task whose second piece has the highest priority among all unallocated tasks. *Please note that the task splitting is performed on the transformed task set. The transformed task set with task splitting is scheduled on the processor.* We now describe LPTS in more detail and develop its associated properties.

Algorithm 2 $\{\tau'_l, \tau''_l\} = LPTS(\tau_l, M_p)$

```
$\delta_{max}$ = HighestPrioritySlack($M_p$)
$\tau'_l = (C'_l = \delta_{max}, T'_l = T_l, D'_l = T^h_p)$
$\diamond$ $\tau'_l$ is given the second highest priority on $M_p$
$\tau''_l = (C''_l = C_l - \delta_{max}, T''_l = T_l, D''_l = T_l)$
$\diamond$ $\tau''_l$ is given an offset of $(C'_l + C^h_p)$
```

B. Lowest-Priority Task-Splitting (LPTS)

The LPTS procedure is described in Algorithm 2. We consider a scenario where the task being split has lower priority than all allocated tasks and higher priority than all remaining unallocated tasks. As in [19], we assume that τ can be split into τ' and τ'', which can be assigned to different processors, and should not execute at the same time. The first piece τ' must execute first, and only after the completion of τ' can τ'' execute. Furthermore, both τ' and τ'' must complete within the same relative deadline of the original task τ. Each of these subtasks will be assigned its own local deadline, and the second subtask will be released when the deadline of the first subtask is reached.

Definition 1. *We define a processor M_p as* non-full *if increasing the computation time of the highest-priority task allocated to M_p by a sufficiently small non-zero amount ϵ ($\epsilon \rightarrow 0$) does not make the task-set already allocated to M_p infeasible.*

The above definition implies that, on a non-full processor, the computation time of the highest priority task can be increased without any other task already allocated to the processor missing its deadline. We will use the notation δ_{max} (> 0) to represent the maximum additional computational time available at the highest priority level on a *non-full* processor such that all of the tasks already allocated on processor M_p remain schedulable. This means that increasing the computation time of the highest-priority task by δ_{max} would still result in a feasible task-set on M_p.

We will also denote the period of the highest-priority task (τ^h_p) allocated to a processor M_p as T^h_p, and the period of the lowest priority task (τ^l_p) allocated to M_p as T^l_p. Correspondingly, the worst-case execution time of the highest-priority task allocated to processor M_p is C^h_p, and the worst-case execution time of the lowest priority task allocated to M_p is C^l_p. δ_{max} can be obtained by using a similar technique as that of $MaximalSplit$ from [19, 12]. Each task τ_i assigned to processor M_p is considered and the maximum value of the computation time of the highest priority task ($C_p^h + \delta$, $\delta > 0$) for which τ_i still meets its deadline using an exact schedulability test is determined. This can be determined for all τ_i allocated in processor M_p and the minimum value of δ over all these tasks can be assigned as δ_{max}. An alternative way of obtaining δ_{max} is using $B(r_p)$ from Equation 1. Since $B(r_p)$, $U(p)$ and T_p^h are already known, δ_{max} is given by $\delta_{max} = (B(r_p) - U(p))\, T_p^h$. When a fast response is required, this method can be used for deciding δ_{max}. In this paper, we use this technique.

C. pCOMPATS Algorithm Description

The pCOMPATS algorithm first transforms the given taskset using the period transformation scheme of Algorithm 1. The period-transformed tasks are then sorted in a non-decreasing order of periods. Tasks are then packed on to a processor until unschedulablity is reported when a task τ_l is added. If the processor M_p is non-full (without the task τ_l being considered for M_p) , then we apply Lowest-Priority Task-Splitting (LPTS) to split τ_l into τ_l' and τ_l'' such that τ_l' fills up processor M_p when assigned a local deadline equal to T_p^h and explicitly made the second highest-priority task on M_p. The second piece τ_l'' is allocated to the next processor M_{p+1}, and we continue allocation onto the next processor M_{p+1}. Alternatively, if the processor M_p is already full, then the task is added to the next processor M_{p+1}, and we continue with the allocation.

In this paper, we only consider tasksets with implicit-deadline tasks with task deadlines equal to periods. We also assume rate-monotonic scheduling on each processor. Under pCOMPATS, the only task that violates this assumption is τ_l', which is assigned a deadline T_p^h that is shorter than its period T_l. We also assign the second highest scheduling priority to task τ_l'. The schedulability of split task τ_l' will be proved separately in Lemma 1, and the impact of τ_l' on other tasks will be quantified in Theorem 1. The pCOMPATS task allocation algorithm is described in Algorithm 3.

Algorithm 3 $pCOMPATS(\Gamma)$

```
Γ' = $PeriodTransform(\Gamma)$
Sort tasks in $\Gamma'$ in non-decreasing order of periods
for Each task $\tau_l$ in $\Gamma'$ do
    if non-full($M_p$) and (not schedulable($M_p \cup \tau_l$)) then
        $\{\tau_l', \tau_l''\} := LPTS(\tau_l, M_p)$
        $M_p := M_p \cup \tau_l'$
        $M_{p+1} := M_{p+1} \cup \tau_l''$
        $p := p + 1$
        if $p > m$ then
            return unschedulable
    else
        if (not non-full($M_p$)) then
            $p := p + 1$
            if $p > m$ then
                return unschedulable
        $M_p := M_p \cup \tau_l$        ▷ $M_p \cup \tau_l$ is schedulable
return schedulable
```

Lemma 1. *Under pCOMPATS, if adding a task $\tau_l : (C_l, T_l)$ to a non-full processor M_p renders the task-set allocated to M_p infeasible, then τ_l is split using LPTS, and the split task $\tau_l = \{\tau_l' \cup \tau_l''\}$ will be schedulable on processors M_p and M_{p+1}, when $\forall \tau_i, U_i < 0.5$.*

Proof: If task τ_l is the candidate for splitting, task τ_l has the *lowest priority* among all allocated tasks, since pCOMPATS allocates tasks in non-decreasing order of periods. Recall that period transformation ensures that $1 \leq r_p < 2$. Let the split instances of τ_l be τ_l' and τ_l'' with τ_l' being scheduled on M_p. The split instances of τ_l are given by:

$$\tau_l' : (C_l', T_l', D_l'), \tau_l'' : (C_l'', T_l'', D_l'')$$

where, $C_l' = \delta_{max}$, $C_l'' = C_l - \delta_{max}$, $D_l' = T_p^h$, $D_l'' = T_l$, $T_l' = T_l$, $T_l'' = T_l$.

On the processor M_p, the split task τ_l' is assigned a deadline of T_p^h, and the second highest scheduling priority. Any job of τ_l' can face only a single preemption from the task with the highest scheduling priority on M_p, with period $T_p^h = D_l'$. The worst-case response time $W(\tau_l')$ is therefore given by $W(\tau_l') = C_p^h + \delta_{max} = C_p^h + C_l'$. Also, since the period and deadline of τ_l'' are equal to that of τ_l, τ_l'' will retain its highest priority on the next processor M_{p+1} due to the arrangement of unallocated tasks in the non-decreasing order of periods.

This implies that *the worst-case response time* of τ_l'' on M_{p+1} will just be equal to C_l''. In other words, $W(\tau_l'') = C_l'' = C_l - \delta_{max}$. Then, the worst-case response time of task τ_l is given by $W(\tau_l) = W(\tau_l') + W(\tau_l'') = C_p^h + C_l$.

For task τ_l to be schedulable, we need to ensure that $W(\tau_l) \leq T_l$. Since $U_i < 0.5$, $\frac{C_p^h}{T_p^h} + \frac{C_l}{T_l} < 1$. Also, since $T_l \geq T_p^h$, $\frac{C_p^h}{T_l} + \frac{C_l}{T_l} < 1$ and $C_p^h + C_l < T_l$ hold. Hence, task τ_l is schedulable on processors M_p and M_{p+1}. ■

Lemma 2. *Under pCOMPATS, if a task τ_i: (C_i, T_i) is added to a non-full processor M_p, and M_p is schedulable when adding τ_i, then all tasks in M_p will remain schedulable.*

Proof: Consider the scenario when we add any new task τ_l to processor M_p. There are two possible cases: *(i)* Task τ_l can be added to M_p and M_p remains schedulable with τ_l. Here, by the algorithm definition, any task τ_i already allocated to M_p is schedulable along with all other tasks on M_p. *(ii)* Task τ_l gets split using LPTS, and the piece τ_l' with computation time δ_{max} and deadline $D_l' = T_p^h$ is added to M_p. In this case, processor M_p should remain schedulable since δ_{max} is by definition the maximum slack available at the highest priority level such that the taskset allocated to the processor M_p is schedulable, and τ_l' has the second highest priority with a period $T_l \geq T_p^h$. Hence, all tasks, once allocated to a processor M_p under pCOMPATS, continue to be schedulable. ■

D. Utilization Bound Analysis for pCOMPATS

Compatible tasks are tasks that achieve a high processor utilization when scheduled together on the same processor.

Theorem 1. *Given an R-BOUND ratio r_p for a task-set allocated to a processor M_p using pCOMPATS, the schedulable utilization bound of processor M_p, (denoted by $U_{bound}(p)$), is given by:*

$$U_{bound}(p) \geq \ln r_p - \tfrac{3r_p - 5}{2r_p}$$

when $\forall \tau_i, U_i < 0.5$

Proof: There are two cases to consider.

Case 1: A task τ_l gets split on the processor M_p into τ_l' and τ_l'' with τ_l' assigned to M_p, and τ_l'' assigned to M_{p+1}. From the definitions of the R-BOUND ratio r_p and the highest-priority task period T_p^h on M_p, we have $T_l = r_p T_p^h$. We know from the definition of `LPTS` that $D_l' = T_p^h$, and D_l' is smaller than the period $T_l' = T_l$ of τ_l'. We accommodate for this constrained deadline of τ_l' by considering a task τ_l^* with period $T_l^* = D_l' = T_p^h$, computational requirement $C_l^* = C_l'$, deadline $D_l^* = D_l'$, and the second highest scheduling priority on processor M_p. If the taskset allocated to processor M_p is schedulable with τ_l^* (instead of τ_l') under rate-monotonic scheduling, then the taskset should also be schedulable with τ_l' since τ_l' represents a task with a longer period but equal priority as τ_l^* [9].

We know that $T_l = r_p T_p^h$. We therefore have the synthetic utilization[1] [1] of τ_l' as $U^*(\tau_l') = \frac{\delta_{max}}{T_p^h}$. For task τ_l'', we have $U(\tau_l'') = \frac{C_l - \delta_{max}}{T_l} = \frac{C_l - \delta_{max}}{r_p T_p^h}$. Hence, the utilization *inflation* χ of task τ_l due to task splitting is given by:

$$U^*(\tau_l') + U(\tau_l'') - U(\tau_l) = (r_p - 1)\tfrac{\delta_{max}}{r_p T_p^h} = \tfrac{(r_p-1)}{r_p} U^*(\tau_l')$$

Since we use $B(r_p)$ to obtain δ_{max}, we know that $\delta_{max} = (B(r_p) - U(p))\,T_p^h$ satisfies. We also know that $\delta_{max} < C_l$ since the task τ_l would not be split otherwise, and $U(\tau_l)$ is less than 0.5 based on the assumption. Therefore, $U^*(\tau_l') < 0.5$, and the inflation χ of task splitting in pCOMPATS is thus limited by $\chi < \frac{r_p - 1}{2r_p}$.

The utilization bound of processor M_p without splitting obtained using the R-BOUND (Equation 2) is $\lim_{n\to\infty} B(r_p) = \ln r_p + 2/r_p - 1$. Accounting for the *inflation* χ of task splitting gives us the utilization bound for pCOMPATS as $U_{bound}(p) \geq [\ln r_p + 2/r_p - 1] - \frac{(r_p-1)}{2r_p}$. Rearranging, we have

$$U_{bound}(p) \geq \ln r_p - \frac{3r_p - 5}{2r_p} \tag{3}$$

Case 2: No task gets split on the processor M_p. The utilization bound in this scenario is given in [20], where $U_{bound}(p) = \ln r_p + \frac{2}{r_p} - 1 = \ln r_p - \frac{2r_p - 4}{2r_p}$.

We have $2r_p - 4 \leq 3r_p - 5$, since $r_p \geq 1$. Hence,

$$U_{bound}(p) \geq \ln r_p - \tfrac{3r_p - 5}{2r_p}$$

■

Figure 1 shows the utilization bound achieved under pCOMPATS as the r-bound ratio varies from 1 to 2.

[1]The synthetic utilization is defined as the ratio of the computation time of a task to its deadline.

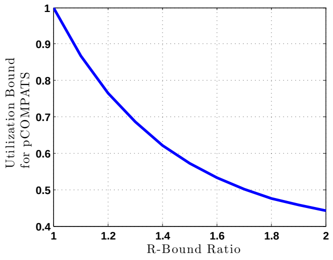


Figure 1: The utilization bound for pCOMPATS as the r-bound ratio varies from 1 to 2.

Corollary 1. *As the ratio of the longest and the shortest periods of the tasks assigned to a processor M_p, i.e. r-bound ratio of processor M_p, approaches 1.0, the utilization bound of processor M_p approaches 1.0 under pCOMPATS.*

Proof: From Equation 3, as r_p decreases, $U_{bound}(p)$ increases. And,

$$\lim_{r_p \to 1} \{\ln r_p - \frac{3r_p - 5}{2r_p}\} = 1 \tag{4}$$

■

We will now study the utilization bound achieved by pCOMPATS. We will do this by studying the first $m-1$ processors and the m^{th} processor separately as the overhead of task-splitting is present only on the first $m-1$ processors.

Lemma 3. *The utilization bound of a task-set Γ with a r-bound ratio r (after period transformation) achieved on the first $k = m-1$ processors, where $m \geq 2$ is given by:*

$$U_{bound}^k = \tfrac{1}{k}\sum_{p=1}^{k} U_{bound}(p) = \frac{1}{k}\sum_{p=1}^{k}(\ln r^{\frac{1}{k}} - \frac{3r^{\frac{1}{k}} - 5}{2r^{\frac{1}{k}}})$$

when $\forall \tau_i, U_i < 0.5$ and $p < m$.

Proof: Let τ_p^h be the highest-priority (shortest period) task assigned to processor M_p, and τ_p^l be the lowest priority (longest period) assigned to processor M_p. Task τ_1 is assigned to processor M_1, and τ_n is assigned to processor M_k. Then, $r_p = \frac{T_p^l}{T_p^h}$ holds good. Hence,

$$r_1 r_2 ... r_p r_{p+1} ... r_k = \frac{T_1^l}{T_1}\frac{T_2^l}{T_2^h}....\frac{T_n}{T_k^h} \leq \frac{T_2^h}{T_1}\frac{T_3^h}{T_2^h}....\frac{T_n}{T_k^h}$$

Since tasks are allocated in non-decreasing order of periods or non-increasing order of priorities $T_i^l \leq T_{i+1}^h$. Therefore,

$$r_1 r_2 ... r_p r_{p+1} ... r_k \leq r \tag{5}$$

where, r is the maximum ratio between any two tasks in the period-transformed taskset.

In order to minimize the per-processor utilization bound, we need to minimize (from Equation 3):

$$\frac{1}{k}\sum_{p=1}^{k} U_{bound}(p) = \frac{1}{k}\sum_{p=1}^{k}(\ln r_p - \frac{3r_p - 5}{2r_p}) \tag{6}$$

subject to Constraint 5. A necessary condition for the minimizer is that the gradient of the Lagrangian function be zero (see example 14.13 in [25]). Let λ denote the Lagrangian

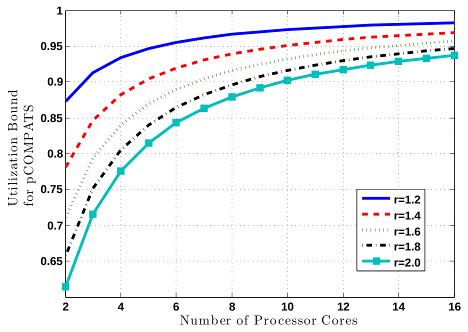


Figure 2: The utilization bound for pCOMPATS for various r-bound ratios as the number of processors is increased from 2 to 16.

multiplier for Constraint 5, where $\lambda \geq 0$. Hence, necessary conditions for the local minimizer are:

$$\frac{1}{k}\left(\frac{1}{r_i} - \frac{5}{2{r_i}^2}\right) = \lambda(r_1 r_2 ... r_{i-1} r_{i+1} ... r_k), \forall i, 1 \leq i \leq k$$

$$\lambda(r_1 r_2 ... r_i r_{i+1} ... r_k - r) = 0$$

For the given equations above, either of $\lambda = 0$ or $r_1 r_2 ... r_i r_{i+1} ... r_k = r$ should be satisfied. If $\lambda = 0$, $\frac{1}{r_i} - \frac{5}{2{r_i}^2} = 0, \forall i, 1 \leq i \leq k$ because $k \neq 0$. Then, we get $r_i = 2.5$, which contradicts the assumption that $r < 2$. Hence, $\lambda > 0$.

When $r_1 r_2 ... r_i r_{i+1} ... r_k = r$, we get $\forall i, 1 \leq i < m$, $\frac{1}{k}\left(\frac{1}{r_i} - \frac{5}{2{r_i}^2}\right) = \frac{\lambda r}{r_i}$ since $r_1 r_2 ... r_{i-1} r_{i+1} ... r_k = \frac{r}{r_i}$. Rearranging, $\forall i, 1 \leq i < m$, $r_i = \frac{5}{2(1-\lambda r k)}$. Hence, $r_1 = r_2 ... = r_p ... = r_k = r^{\frac{1}{k}}$.

Thus, we have:

$$\frac{1}{k}\sum_{p=1}^{k} U_{bound}(p) = \frac{1}{k}\sum_{p=1}^{k}\left(\ln r^{\frac{1}{k}} - \frac{3r^{\frac{1}{k}} - 5}{2r^{\frac{1}{k}}}\right) \qquad (7)$$

■

Figure 2 depicts the utilization bound for pCOMPATS for various r-bound ratios as the number of processor cores is increased.

Theorem 2. *The utilization bound of a task-set* Γ *with an r-bound ratio* r *(after period transformation) for* m *processors, where* $m \geq 2$*, is given by:*

$$U^m_{bound} = \begin{cases} \frac{1}{m}\{(m-1)(\ln(\frac{5}{2})^{\frac{1}{m}} - \frac{3}{2} + \frac{\frac{5}{2}}{(\frac{5}{2})^{\frac{1}{m}}}) + \\ \ln(\frac{4}{5}(\frac{5}{2})^{\frac{1}{m}}) + \frac{\frac{5}{2}}{(\frac{5}{2})^{\frac{1}{m}}} - 1\}, \textit{if } 2 \leq m < 5 \\ \frac{1}{m}\{(m-1)(\ln(2^{\frac{1}{m-1}}) - \frac{3}{2} + \frac{5}{2 \cdot 2^{\frac{1}{m-1}}}) + 1\}, \\ \textit{if } m \geq 5 \end{cases}$$

, when $\forall \tau_i, U_i < 0.5$

Proof: pCOMPATS splits a task if and only if a task cannot be scheduled on the current processor. Upon splitting, all the "available" utilization on that processor is used up by the first sub-task. It can be observed that given m processors, splitting will take place only on processors M_1 to M_{m-1}. This implies that processor M_m will not incur the penalty of utilization inflation due to task splitting. Hence, the utilization bound for each of the first $m-1$ processors is given by Equation 3 and the utilization bound for the m^{th} processor is given by R-BOUND [20]. Let the r-bound ratio for tasks allocated by pCOMPATS on processors $M_1, .., M_{m-1}$ be β and processor M_m be ψ. Let $\beta\psi = \alpha$. From Equation 7, the total utilization bound is given by $U^m_{bound} = \frac{1}{m}\left\{(m-1)\left(\ln(\beta^{\frac{1}{m-1}}) - \frac{3\beta^{\frac{1}{m-1}} - 5}{2\beta^{\frac{1}{m-1}}}\right) + ln\frac{\alpha}{\beta} + \frac{2\beta}{\alpha} - 1\right\}$. The above function is decreasing in α when $1 \leq \alpha < 2$. In our case, we use period transformation to limit the *maximum* value of r-bound ratio to 2 (Please refer to Section 5.1). Hence, the utilization bound is computed with $\alpha = 2$. Therefore, the total utilization bound is given by $U^m_{bound} = \frac{1}{m}\left\{(m-1)\left(\ln(\beta^{\frac{1}{m-1}}) - \frac{3\beta^{\frac{1}{m-1}} - 5}{2\beta^{\frac{1}{m-1}}}\right) + ln\frac{2}{\beta} + \beta - 1\right\}$.

The above utilization bound function is decreasing until $\beta = (\frac{5}{2})^{1-\frac{1}{m}}$, and is increasing beyond this value. We know that $\beta \leq 2$. Hence we have, $\beta = \begin{cases} (\frac{5}{2})^{1-\frac{1}{m}}, & \text{if } 2 \leq m < 5 \\ 2, & \text{if } m \geq 5 \end{cases}$

From the above equations, we have:

$$U^m_{bound} = \begin{cases} \frac{1}{m}\{(m-1)(\ln(\frac{5}{2})^{\frac{1}{m}} - \frac{3}{2} + \frac{\frac{5}{2}}{(\frac{5}{2})^{\frac{1}{m}}}) + \\ \ln(\frac{4}{5}(\frac{5}{2})^{\frac{1}{m}}) + \frac{\frac{5}{2}}{(\frac{5}{2})^{\frac{1}{m}}} - 1\}, \text{if } 2 \leq m < 5 \\ \frac{1}{m}\{(m-1)(\ln(2^{\frac{1}{m-1}}) - \frac{3}{2} + \frac{5}{2 \cdot 2^{\frac{1}{m-1}}}) + 1\}, \\ \text{if } m \geq 5 \end{cases}$$

■

Corollary 2. *As the number of processors increases, the utilization bound of a task-set* Γ *scheduled using pCOMPATS approaches* 1.

Proof: From Theorem 2, we have:

$$\lim_{m\to\infty} \frac{1}{m}\{(m-1)(\ln(2^{\frac{1}{m-1}}) - \frac{3}{2} + \frac{5}{2 \cdot 2^{\frac{1}{m-1}}}) + 1\} = 1 \qquad (8)$$

■

Equation 12 is depicted in Figure 3.

Proposition 1. *Under fixed-priority partitioned scheduling without task-splitting, the utilization bound for tasks, where each tasks's utilization is less than or equal to* 0.5*, is no more than* $\frac{2}{3} + \frac{1}{3k}$*, where* k *is the number of processors.*

Proof: Consider k processors with $2k+1$ tasks, each with utilization of $\frac{1}{3}+\epsilon$. This taskset is not schedulable since no 3 tasks can fit on the same processor. The utilization bound is therefore no more than $\frac{1}{3}(\frac{2k+1}{k}) = \frac{2}{3} + \frac{1}{3k}$ ■

pCOMPATS Example: It can be observed that although r for a given task-set could be just less than 2 in the worst case, the individual r_p for each processor p does not have to be necessarily better for improving the schedulability bounds. For example, consider the following taskset obtained after period transformation is: Γ' τ_1: $(20, 100, 100)$, τ_2: $(36, 120, 120)$, τ_3: $(75, 150, 150)$, τ_4: $(80, 160, 160)$, τ_5: $(100, 180, 180)$, τ_6: $(38, 190, 190)$. Let the system comprise three processors M_1, M_2 and M_3. According to the proposed

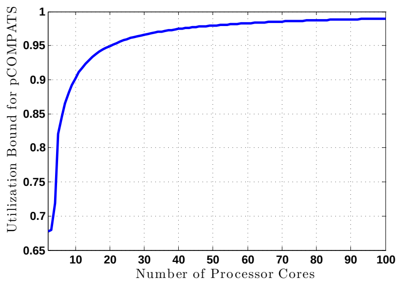


Figure 3: Utilization Bound for pCOMPATS as the number of processor cores increases.

$pCOMPATS$ task allocation algorithm, tasks are allocated in increasing order of periods. Observe that after tasks τ_1 and τ_2 fit into M_1, M_1 is non-full. If task τ_3 is added to M_1, it will have a worst-case response time of 187 causing it to miss its deadline of 150. Therefore, task τ_3 will be split into $\tau_3^{'}$: $(44, 150, 100)$ having the second highest priority on M_1, and $\tau_3^{''}$: $(31, 150, 150)$ having the highest priority on M_2. Similarly, on processor M_2, task τ_4 can be allocated, and task τ_5 has to be split into $\tau_5^{'}$: $(31, 180, 150)$ having the second highest priority on M_2, and $\tau_5^{''}$: $(69, 180, 180)$ having the highest priority on M_3. Task τ_6 can be allocated to M_3. The R-BOUND ratio for the entire taskset is $r = 190/100 = 1.9$. Now, let us look at the R-BOUND ratios of the individual processors: $r_1 = 1.5$, $r_2 = 1.067$ and $r_3 = 1.056$. As we can see, the individual R-BOUND ratios are better than r, and this effect offsets the utilization inflation due to task-splitting.

E. Heavy Tasks

Theorem 2 assumes that $\forall \tau_i$, $U_i < 0.5$. In this section, we propose pCOMPATS-HT which is specifically designed for handling heavy tasks, i.e. tasks with utilization ≥ 0.5.

1) pCOMPATS-HT: The pCOMPATS-HT scheme is given in Algorithm 4. The algorithm arranges the transformed taskset in non-decreasing order of priorities and allocates each of the tasks into each of the empty processors. If no empty processor is found and if the current processor is non-full, then Highest-Priority Task Splitting (HPTS) [19] is employed.

In pCOMPATS-HT, due to the non-decreasing priority order arrangement of the taskset, the task currently being considered for allocation to a processor will be the highest priority task in that processor.

For heavy tasks, HPTS is used instead of LPTS. Under LPTS, the response times of the subtasks are increased by the response times of the higher priority tasks. For heavy tasks, such an increase could potentially cause a deadline miss. Hence, in order to avoid deadline misses, HPTS is employed.

Theorem 3. *The upper bound on schedulable utilization of a task-set Γ (after period transformation) when scheduled using pCOMPATS-HT is at most 72% (when $\forall \tau_i$, $U_i \geq 0.5$).*

Proof:

Consider the case when the number of processors $m = 3$ and, three tasks of utilization 0.5 and one task of utilization $0.656 + \epsilon$ ($\epsilon \to 0$) is allocated using pCOMPATS-HT. Given that the utilization of each of the tasks is ≥ 0.5 and that the Liu and Layland 2-task utilization bound is 0.828, it can be seen that no more than two tasks can be allocated onto a processor. Hence, in this case, each of the tasks of utilization ≥ 0.5 will be allocated to each of the processors. One task of utilization $0.656 + \epsilon$ will be split into three sub-tasks of 0.328, 0.328 and ϵ. Each of the sub-tasks will be allocated onto each of the processors. The utilization bound is then $\approx 72\%$. This example can be extended for m processors. Consider n tasks with utilization ≥ 0.5 where $n > m$. The first m tasks will each be allocated to a processor. Now the $(m+1)^{th}$ task τ_{m+1} will not fit into any of the processors due to the 2-task utilization bound of 0.828. Say τ_{m+1} is split across p processors. Then, $p-1$ processors will be filled up to 0.828 and the p^{th} processor will have the remaining piece ϵ. In the worst case this piece could be $U(\epsilon) \to 0$ and the already allocated task have an utilization of 0.5. The total utilization is then $U_p = \frac{(p-1)*0.828+(0.5+\epsilon)}{p}$. When $p = 2$, U_p is $\approx 75\%$ and when $p = 3$, U_p is $\approx 72\%$. When p is greater than 3, U_p only increases. Thus, the upper bound on schedulable utilization of a task-set Γ where each of the task's utilization is ≥ 0.5 when allocated using pCOMPATS-HT is $\approx 72\%$. ■

Algorithm 4 $pCOMPATS - HT(\Gamma)$

```
$\Gamma' = PeriodTransform(\Gamma)$
Sort tasks in $\Gamma'$ in non-decreasing order of priority
Assign the $m$ lowest priority tasks to each of the $m$ processors.
$p := 1$
**for** Each task $\tau$ remaining in $\Gamma'$ **do**
    **while Not** *schedulable*$(M_p \cup \tau)$ **do**
        $\{\tau', \tau''\} = HPTS(\tau, M_p)$        ▷see [19]
        $M_p = M_p \cup \tau$
        $\tau := \tau''$
        $p := p + 1$
        **if** $p > m$ **then**
            return *unschedulable*
    $M_p = M_p \cup \tau$
    **if** $p > m$ **then**
        return *unschedulable*
return *schedulable*
```

VI. Evaluation for Average-Case behavior

We have so far analyzed the worst-case performance of the pCOMPATS algorithm and obtained its utilization bound. The worst-case performance occurs when the R-BOUND ratio for the task-set is less than 2 by an infinitesimally small non-zero amount, and the R-BOUND ratios for each of the processors are equal and less than $2/m$ by an infinitesimally small non-zero amount. Additionally, when a task τ_i is split such that τ_i' is allocated to processor M_p and τ_i'' is allocated to processor M_{p+1}, $U(\tau_i')$ should be almost equal to that of τ_i and $U(\tau_i'')$ is just greater than zero. These conditions represent extreme situations, and the average-case performance of pCOMPATS is expected to be far better than

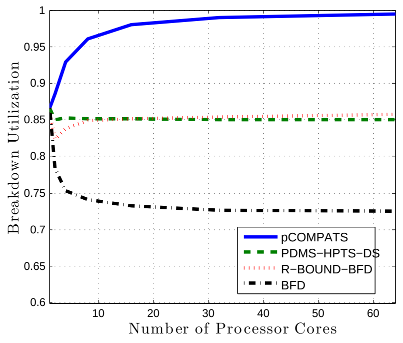


Figure 4: Breakdown utilization of various task-partitioning algorithms as the number of processor cores increases for tasks with period ranging between 100 to 200.

its utilization bound.

In this evaluation, we will study the average-case performance of our algorithm on randomly generated tasksets and compare it against PDMS-HPTS-DS [19], R-BOUND with Best Fit Decreasing heuristic (R-BOUND-BFD) and plain Best Fit Decreasing (BFD) heuristic. The breakdown utilization values were computed as given in [21]. The task period T for each task was chosen in an uniformly random fashion from the interval [100, 200] in the first experiment and between [100, 1000] in the second experiment. The computation time C for each task was chosen using a uniform distribution from the interval [0, 0.5T]. Tasks were generated till the total utilization exceeded $m \times 100\%$. These computation times were then proportionally scaled down to compute the breakdown utilization (for more details , please see [21]).

In Figure 4, where average breakdown utilization is plotted from 1000 runs, one can observe that as the number of cores increases, the breakdown utilization for pCOMPATS keeps increasing while that of PDMS-HPTS-DS and R-BOUND converge to about 85%. Also, it is to be noted that even for a smaller number of processors, pCOMPATS does much better than its worst-case utilization bound. PDMS-HPTS-DS does not go beyond 85% as it suffers from task splitting overhead. R-BOUND-BFD on the other hand suffers from bin-packing effects i.e. there could be unused “space” in the processor into which a task could not fit as it does not leverage task-splitting. The BFD algorithm does not leverage the compatibility of tasks within a given task-set and also does not perform task-splitting, resulting in inefficient packing. In the second experiment, we increased the range of the periods of the tasks to [100,1000]. The results of this experiment are shown in Figure 5. Here, we see that all the algorithms perform better but the performance of R-BOUND-BFD remains the same. This is because R-BOUND-BFD uses STS and hence the increase in task period does not affect it. Meanwhile, PDMS-HPTS-DS performs better due to lower loss in utilization due to task-splitting. Still, pCOMPATS continues to perform the best.

VII. Case Study

In this section, we present a case study characterizing the practical overheads associated with pCOMPATS due to task splitting and period transformation on the Intel Core i7 processor running Linux/RK. The processor has four cores

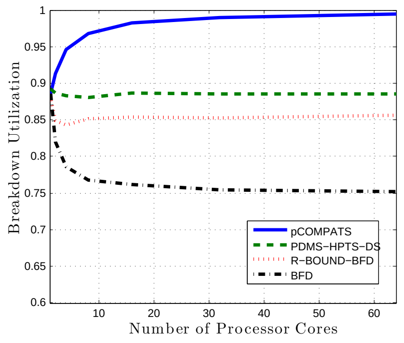
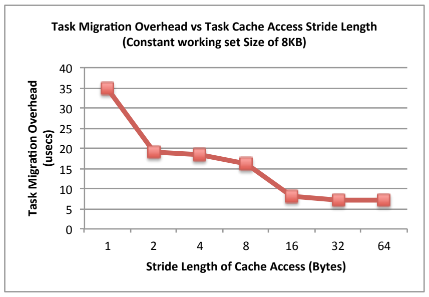
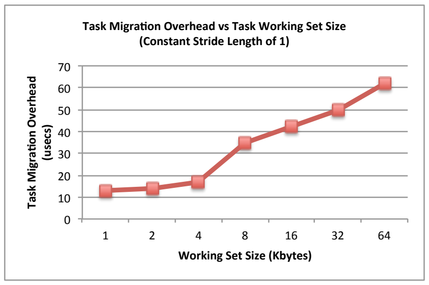


Figure 5: Breakdown utilization of various task-partitioning algorithms as the number of processor cores increases for tasks with period ranging between 100 to 1000.



(a) Migration overheads vs Cache access stride lengths



(b) Migration overheads vs Working-set sizes

Figure 6: Task migration overheads measured on Intel Core i7 platform

with private L1 and L2 caches and a shared L3-cache. The L1-cache (64 KB) is a split cache with both Instruction and Data caches having a size of 32 KB each. The L2-cache is of size 256 KB. The L3-cache is a unified cache of size 8 MB. L1, L2 and L3 are on-chip resources, and the cache line size is 64 bytes.

A resource kernel [28, 26, 27] is a resource centric approach for building real-time kernels that provide timely, guaranteed and enforced access to system resources. Linux/RK [26, 27] is a resource kernel within the Linux operating system. There are two basic abstractions in Linux/RK, a *reserve* and a resource set. A reserve represents a share of a single computing resource such as CPU time, physical memory pages, a network bandwidth and so on. A resource set is a grouping of reserves on different resources into a single set that is accessible by user processes. In this work, we are concerned with the CPU resource. Each reservation is analogous to a fixed-priority aperiodic server and will consist of the (C, T, D) model, where C is the execution budget, T is the replenishment period of the reservation and D is the deadline. Tasks that are split will have a budget for each of the cores it is executed on. Hence, the reservation for split tasks will consist of a CPU affinity apart from (C, T, D) and the ordering of the migration.

Linux/RK uses the High-Resolution timer (HRtimer) to maintain the fine-grained timing information. The HRtimer is a per-core hardware counter with nanosecond precision, which has been adopted in most mainstream processor architectures. Each HRtimer is related with a time-ordered event tree. When it reaches the counting time of current pending event, the current event's callback function is invoked and the next pending event is reloaded. Using the HRtimer, task accounting and release is performed. The CPU core that a task should be scheduled on is set by using the *sched_setaffinity()* kernel function. The migration of a split task will be handled by setting the CPU affinity in the HRtimer callback function.

In order to understand the impact of task migration on cache performance, we evaluated a series of synthetic cache workloads. These workloads had varying working-set sizes (from 1 KB to 64 KB) and stride-lengths (1 to 64 bytes). The performance of these workloads are shown in Figure 6a and Figure 6b. The overall overhead was acceptably low (less than 60 microseconds) for these cache-workloads. These workloads exercised only the data cache, and the instruction cache effects were neglected in these experiments. The timing measurements were done at a fixed processor frequency of 3 GHz. The results show that the overheads are generally lower at smaller working set sizes, as expected. At lower stride lengths, the cache overhead of task-splitting is higher. As the stride length increases, the performance difference introduced by task-splitting diminishes. When a split-task migrates from one core to another, it has to re-create the cache state on the new core. A task with spatially sparse access patterns will load its entire cache state much faster than those with sequential access patterns. Most modern cores provide extensive support for out-of-order execution, load-store queues, and cache pre-fetching, which reduce the impact of task migration. In such cores, tasks with spatially sparse access patterns generate multiple parallel requests to different cache lines, whereas sequential access patterns result in stalling on the same cache line. This analysis was done with R/W access patterns; however, similar behavior was also observed with read-only access. Although not shown here, tasks with low temporal locality will also have lower cache overheads due to task splitting.

In an effort to characterize the impact of task migration on a real-world application, we looked at the media player application called *Mplayer*. We observed that the overhead due to task splitting was negligible. Mplayer was scheduled such that the *decode_frame* module gets split across the two cores at exactly mid-way through the processing. The FFmpeg library with libavcodec was used and a mpeg4 file was being played. The core computational loop in the video player is comprised solely of *decode_frame* in addition to minor accounting updates. Without task splitting, the average

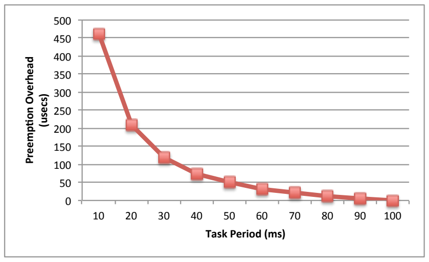


Figure 7: Period Transform overheads measured on Intel Core i7 platform

time taken by a single call to *decode_frame* was 3.79 ms. After performing task splitting, the average time taken for a *decode_frame* call was increased by approximately 0.03 ms. The reason for the overhead of about 30 microseconds is probably the fact that it incurs heavy cache overheads. The video was being played at approximately 45 frames/second (a period of 22 ms), and this cache overhead translates to 0.135% in terms of *decoder* task utilization. It is to be noted here that the overhead numbers include both OS overhead in migrating the task, as well as cache-overheads.

To measure the overhead of period transformation, we consider two tasks τ_1 and a test task τ_2. Both tasks are CPU-bound to avoid cache overheads since we are concerned only with the preemption overheads due to period transformation in this case. We first let task τ_2 execute without any interruption (with the highest priority in the RT scheduling class), and record its execution time, as the *net time*. Then we let τ_2 run with τ_1, where τ_2 has lower priority than τ_1. We record task τ_2's execution time in this case, as the *gross time*. The overhead equals the gross time minus the net time. The measured overhead includes the overhead of scheduler invocation, context switch and queueing operations. We measured the average preemption overhead to be 5 microseconds. When period transformation is applied, the number of preemptions faced by the lower priority tasks will increase. This increase will also depend on the number of tasks in the system. As an example, we let τ_1 have an utilization of 40% with a period of 100 ms. This experiment is repeated over different period transform ratios of task τ_1. The measurements were averaged over one million runs for each of the experiments. The measurement results are shown in Figure 7, which shows the overhead when task τ_2 faces every preemption from τ_1. In reality, τ_1 might be scheduled during idle intervals. We see that as the period of τ_1 is reduced, the overhead increases due to the increased number of preemptions. When the period is reduced by a factor of 10 (from 100 ms to 10 ms), the overhead is still only around 450 μs over a 1000 ms interval, corresponding to an overhead of 0.045%. In pCOMPATS, period transformation is performed only to bring the ratio of the task periods to be within 1 and 2, thus limiting the overhead caused by period transform.

Preemptions also cause cache related overheads. To measure this cache related overhead, competing task τ_1 and test task τ_2 read and write to an array of size equal to the working set size. First, we execute test task τ_2 to ensure that its working set has been loaded into the cache. Then,

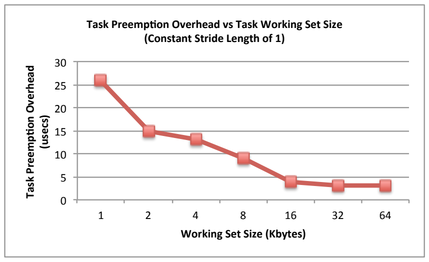


Figure 8: Task preemption overhead for varying working set size measured on Intel Core i7 platform

we execute τ_2 again to measure the net time of τ_2. Next, we let τ_2 to run with the competing task τ_1 where τ_1 has a higher priority. We record the test task's execution time in this case as the gross time. The experiment is repeated for different working set sizes and is shown in Figure 8. Our measurements shows that in general the cache-related overhead due to task migrations is greater than local context switches but only by a small margin. This is due to the shared lower-hierarchy caches (L3 cache in our case): in both local context switches and task migrations, most of the working space of the preempted/to-migrate task will be replaced out from the private cache (L1 and L2 cache in our case), and stay in the shared lower-hierarchy caches. If an application has generally very small working space (much smaller than the size of private cache), the cache-related delay of local context switches will be smaller than task migrations, since there is a better chance for the working space of the preempted task to stay in the private cache, until it resumes execution.

Our evaluation of task migration and period transformation on a real platform shows that the cache and preemption overheads caused by them can be expected to be negligible in multi-core platforms. Hence, pCOMPATS that uses task-splitting and period transform is a practical approach to improve the overall system utilization in partitioned real-time multi-core scheduling.

VIII. Concluding Remarks

In this paper, we have considered the problem of scheduling periodic real-time tasks on multi-core processors. Specifically, we focus on the semi-partitioned approach, which statically allocates each task to a processing core and splits some tasks across cores. We propose an algorithm called pCOMPATS (period Compatible Allocation and Task Splitting) that clusters *compatible* tasks together with task splitting and show that as the number of processor cores increases, the least upper utilization bound on schedulable utilization achieved using pCOMPATS approaches 100%. pCOMPATS does not split more than one task per processor, and therefore minimizes any penalties of task-splitting. We call tasks having utilization greater than or equal to 50% as "heavy tasks" and provide a task partitioning algorithm called pCOMPATS-HT for allocating such tasks. We show that the upper bound on the schedulable utilization using pCOMPATS-HT is at most 72%. To the best of our knowledge, this is the first result that shows that the schedulable utilization bound improves as the number of processing cores increases. Multi-core processor scheduling approaches proposed so far perform independent of the number of processor cores. We have evaluated the performance of pCOMPATS and other well-known partitioning techniques, and show that using pCOMPATS provides much better utilization bound in the average case. Our results are especially useful in the context of future many-core processors such as Intel's 48-core Single-chip Cloud Computer (SCC) [13]. We have measured the cache and preemption overhead caused by task splitting and period transformation by pCOMPATS on the Intel Core i7 processor running Linux/RK. The overheads are seen to be low on the platform, making pCOMPATS to be practical.

References

[1] T. Abdelzaher, V. Sharma. A synthetic utilization bound for aperiodic tasks with resource requirements. *Euromicro Conference on Real-Time Systems, 2003.*

[2] J. H. Anderson, V. Bud, and U. C. Devi. An edf-based scheduling algorithm for multiprocessor soft real-time systems. *ECRTS 2005.*

[3] B. Andersson. Global static-priority preemptive multiprocessor scheduling with utilization bound 38%. *International Conference on Principles of Distributed Systems*, pages 73–88, 2008.

[4] B. Andersson and K. Bletsas. Sporadic multiprocessor scheduling with few preemptions. *Euromicro Conference on Real-Time Systems 2008.*

[5] B. Andersson, K. Bletsas, and S. Baruah. Scheduling arbitrary-deadline sporadic task systems on multiprocessors. *Real-Time Systems Symposium 2008.*

[6] B. Andersson and J. Jonsson. The utilization bounds of partitioned and pfair static-priority scheduling on multiprocessors are 50%. *ECRTS 2003.*

[7] B. Andersson and E. Tovar. Multiprocessor scheduling with few preemptions. *Polytechnic Institute of Porto, Portugal HURRAY-TR-060811 2006.*

[8] S. Borkar. Thousand core chips: a technology perspective. *Design Automation Conference 2007.*

[9] A. Burns and S. Baruah. Sustainability in real-time scheduling. In *Journal of Computing Science and Engineering*, 2008.

[10] A. Burns, R. Davis, P. Wang, and F. Zhang. Partitioned edf scheduling for multiprocessors using a c=d scheme. *RTNS 2010.*

[11] R. Davis and A. Burns. A survey of hard real-time scheduling algorithms and schedulability analysis techniques for multiprocessor systems. techreport YCS-2009-443, University of York, Department of Computer Science, 2009.

[12] N. Guan, M. Stigge, W. Yi, and G. Yu. Fixed-priority multiprocessor scheduling with liu and layland's utilization bound. *RTAS 2010.*

[13] Intel. *www.intel.com*, 2010.

[14] S. Kato and N. Yamasaki. Real-time scheduling with task splitting on multiprocessors. *RTCSA 2007.*

[15] S. Kato and N. Yamasaki. Semi-partitioned fixed-priority scheduling on multiprocessors. *RTAS 2009.*

[16] S. Kato and N. Yamasaki. Portioned edf-based scheduling on multiprocessors. *EMSOFT 2008.*

[17] S. Kato and N. Yamasaki. Portioned static-priority scheduling on multiprocessors. *IPDPS 2008.*

[18] S. Kato, N. Yamasaki, and Y. Ishikawa. Semi-partitioned scheduling of sporadic task systems on multiprocessors. *ECRTS 2009.*

[19] K. Lakshmanan, R. Rajkumar, and J. Lehoczky. Partitioned fixed-priority preemptive scheduling for multi-core processors. *ECRTS 2009.*

[20] S. Lauzac, R. Melhem, and D. Mosse. An efficient rms admission control and its application to multiprocessor scheduling. *Parallel Processing Symposium 1998.*

[21] J. Lehoczky, L. Sha, and Y. Ding. The rate monotonic scheduling algorithm: exact characterization and average case behavior. *Real Time Systems Symposium 1989.*

[22] J. Liebeherr, A. Burchard, Y. Oh, and S. H. Son. New strategies for assigning real-time tasks to multiprocessor systems. *IEEE Trans. Comput. 1995.*

[23] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, 20(1):46–61, 1973.

[24] L. Mosley. Power delivery challenges for multicore processors. In *Proceedings of CARTS*, 2008.

[25] S. Nash and A. Sofer. *Linear and Nonlinear optimization*. McGraw-Hill, 1996.

[26] S. Oikawa and R. Rajkumar. Linux/rk: A portable resource kernel in linux. *IEEE Real-Time Systems Symposium Work-In-Progress*, 1998.

[27] S. Oikawa and R. Rajkumar. Portable rk: A portable resource kernel for guaranteed and enforced timing behavior. *RTAS*, 1999.

[28] R. Rajkumar, K. Juvva, A. Molano, and S. Oikawa. Resource kernels: A resource-centric approach to real-time systems. *SPIE/ACM Conference on Multimedia Computing and Networking*, 1998.

[29] E. Seo, J. Jeong, S. Park, and J. Lee. Energy efficient scheduling of real-time tasks on multicore processors 2008. *IEEE Transactions on Parallel and Distributed Systems.*

[30] L. Sha, J. P. Lehoczky, and R. Rajkumar. Solutions for some practical problems in prioritized preemptive scheduling. RTSS 1986.

[31] Shinpei and N. Yamasaki. Real-time scheduling with task splitting on multiprocessors. *International Workshop on Real-Time Computing Systems and Applications*, 2007.