2009 21st Euromicro Conference on Real-Time Systems

Partitioned Fixed-Priority Preemptive Scheduling for Multi-Core Processors

Karthik Lakshmanan, Ragunathan (Raj) Rajkumar, and John P. Lehoczky
Carnegie Mellon University
Pittsburgh, PA 15213, USA
lakshmanan@cmu.edu, raj@ece.cmu.edu, jpl@stat.cmu.edu

Abstract*—Energy and thermal considerations are increasingly driving system designers to adopt multi-core processors. In this paper, we consider the problem of scheduling periodic real-time tasks on multi-core processors using fixed-priority preemptive scheduling. Specifically, we focus on the partitioned (static binding) approach, which statically allocates tasks to processing cores. The well-established 50% bound for partitioned multiprocessor scheduling [10] can be overcome by task-splitting (TS) [19], which allows a task to be split across more than one core. We prove that a utilization bound of 60% per core can be achieved by the partitioned deadline-monotonic scheduling (PDMS) class of algorithms on implicit-deadline tasksets, when the highest-priority task on each processing core is allowed to be split (HPTS). Given the widespread usage of fixed-priority scheduling in commercial real-time and non real-time operating systems (e.g. VxWorks, Linux), establishing such utilization bounds is both relevant and useful. We also show that a specific instance of PDMS_HPTS, where tasks are allocated in the decreasing order of size, called PDMS_HPTS_DS, has a utilization bound of 65% on implicit-deadline task-sets. The PDMS_HPTS_DS algorithm also achieves a utilization bound of 69% on lightweight implicit-deadline task-sets where no single task utilization exceeds 41.4%. The average-case behavior of PDMS_HPTS_DS is studied using randomly generated task-sets, and it is seen to have an average schedulable utilization of 88%. We also characterize the overhead of task-splitting using measurements on an Intel Core 2 Duo processor.

I. Introduction

Multi-core processors are quickly emerging as the dominant technology in the microprocessor industry. Massively multi-core processors are featured prominently in the future product roadmaps of many industry leaders. Most multi-core offerings largely resemble symmetric multiprocessors (SMPs) with co-located computation cores, which have significantly lower communication and co-ordination latencies. In this paper, we consider the problem of scheduling periodic real-time tasks on multi-cores to better utilize their parallel processing capability.

Real-time scheduling on multiprocessor systems is a well-studied problem in the literature. 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. In this paper, we focus only on the partitioned approach, and also assume that uniprocessor fixed-priority preemptive scheduling schemes are used. We perform task splitting where a task can be split to run across more than one processor[1]. This allows us to overcome the 50% bound imposed by the underlying bin-packing problem of allocating tasks to processors. Task splitting takes advantage of the co-located nature of the processing cores to increase the overall system utilization. Our algorithms presented in this paper do not split more than one task per processor, and therefore minimize any penalties of task splitting. Per-processor run-queues are still used to schedule tasks and thus retain the property of partitioned scheduling.

In this paper, we extend the class of partitioned deadline-monotonic scheduling (PDMS) algorithms by allowing the *highest-priority* task on a processor core to be split (HPTS) across more than one core. We prove that this extension achieves a schedulable per-processor utilization bound of 60% on implicit-deadline task-sets, under deadline-monotonic priority assignment. Under HPTS, a previously split task may be chosen again for splitting if it has the highest priority on a given core. A specific instance of this class of algorithms, in which tasks are allocated in the decreasing order of size (PDMS_HPTS_DS), can achieve a per-processor utilization bound of 65% on implicit-deadline task-sets. The behavior of PDMS_HPTS_DS on lightweight tasks is better, and a worst-case schedulable utilization of 69% on implicit-deadline task-sets is achieved when the individual task utilizations are less than 41.4%. When exact schedulability tests are used on the individual processors, PDMS_HPTS_DS is seen to achieve an average schedulable utilization of 88% for randomly generated implicit-deadline tasksets. In order to characterize the practical overhead of task-splitting, we present a case study using the Intel Core 2 Duo processor.

Fixed-priority preemptive scheduling is supported in a variety of commercial OSs including Linux, VxWorks, LynxOS and ThreadX. Industry standards like POSIX and AUTOSAR also provide standardized interfaces for fixed-priority scheduling. Given this widespread usage, our analysis of task-splitting in the fixed-priority context is both relevant and useful. Our main contribution is in establishing the utilization bounds for task-splitting in the context of partitioned preemptive fixed-priority preemptive scheduling.

A. Organization

The rest of this paper is organized as follows. First, we summarize prior research related to this work. We then provide a brief overview of the bin-packing problem, and show how splitting objects can achieve improved packing. We next

[1]In the rest of this paper, we will use the terms *processor*, *core* and *processor core* interchangeably.

978-0-7695-3724-5/09 $25.00 © 2009 IEEE
DOI 10.1109/ECRTS.2009.33

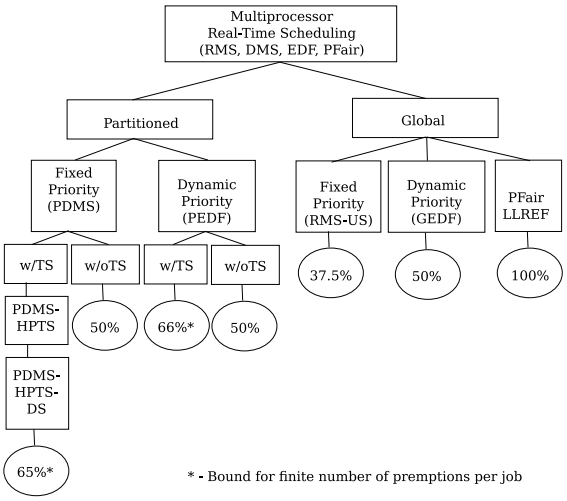


Fig. 1. Design space of Multi-Processor Real-Time Scheduling

translate the notion of dividing objects into splitting tasks, and compute the penalty of splitting the highest-priority task (HPTS). Utilization bounds are then developed for the class of PDMS algorithms under HPTS. The PDMS_HPTS_DS algorithm is then presented, its utilization bounds analyzed and its average-case performance evaluated. Task-splitting overheads are then measured on the Intel Core 2 Duo.

II. Related Work

The design space of the existing literature on multiprocessor real-time scheduling algorithms is shown in Fig. 1. Multiprocessor scheduling schemes are classified into global (*1*-queue *m*-server) and partitioned (*m*-queue *m*-server) systems. It has been shown that each of these categories has its own advantages and disadvantages [2]. Global scheduling schemes can better utilize the available processors, as best illustrated by PFair [3] and LLREF [24]. These schemes appear to be best-suited for applications with small working-set sizes. Although the last level of on-chip shared cache ultimately determines the caching behavior of an application, task migrations tend to generate significant additional cache traffic due to invalidations and cache-consistency protocols. Weak processor affinity and preemption overheads therefore need to be managed to fully exploit the benefits of global approaches [4]. On the other hand, partitioned approaches are severely limited by the low utilization bounds associated with bin-packing problems. The advantage of these schemes is their stronger processor affinity, and hence they provide better average response times for tasks with larger working set sizes.

Global scheduling schemes based on rate-monotonic scheduling (RMS) and earliest deadline first (EDF) are known to suffer from the so-called Dhall effect [5]. When heavy-weight (high-utilization) tasks are mixed with lightweight (low-utilization) tasks, conventional real-time scheduling schemes can yield arbitrarily low utilization bounds on multiprocessors. By dividing the task-set into heavy-weight and lightweight tasks, the RM-US [6] algorithm achieves a utilization bound of 33% for fixed-priority global scheduling. These results have been improved with a higher bound of 37.5% [7]. The global EDF scheduling schemes have been shown to possess a higher utilization bound of 50% [8]. PFair scheduling algorithms based on the notion of proportionate progress [9] can achieve the optimal utilization bound of 100%. Despite the superior performance of global schemes, significant research has also been devoted to partitioned schemes due to their appeal for a significant class of applications, and their scalability to massive multi-cores, while exploiting cache affinity.

Partitioned multiprocessor scheduling techniques have largely been restricted by the underlying bin-packing problem. The utilization bound of strictly partitioned scheduling schemes is known to be 50%. This optimal bound has been achieved for both fixed-priority algorithms [10] and dynamic-priority algorithms based on EDF [11]. Most modern multi-core processors provide some level of data sharing through shared levels of the memory hierarchy. Therefore, it could be useful to split a bounded number of tasks across processing cores to achieve a higher system utilization [18]. Partitioned dynamic-priority scheduling schemes with task splitting have been explored in this context [12], [13]. Fixed-priority scheduling with task-splitting support is relatively less analyzed in the literature. In this paper, we characterize the behavior of partitioned deadline-monotonic scheduling (PDMS) with task-splitting. Specifically, we focus on the effects of deadline-monotonic priority assignments and splitting the highest-priority task (HPTS), and show that this can lead to a utilization bound of 60% for this category of algorithms on implicit-deadline task-sets. A specific instance of this class, where tasks are allocated in the decreasing order of size using PDMS_HPTS_DS, is shown to have a higher utilization bound of 65%. This algorithm also performs better on lightweight task-sets achieving a utilization bound of 69%, when the individual task utilizations are restricted to be less than 41.4%.

The main advantages of PDMS_HPTS_DS are:

1) a higher utilization bound than conventional partitioned multiprocessor scheduling schemes,
2) a tight bound on the number of tasks straddling processing core boundaries (1 per core),
3) the ability to use both utilization-based tests and exact schedulability conditions,
4) high average-case utilization (88%), and
5) strong processor affinity due to the partitioned nature.

In the area of real-time multi-core scheduling, there has also been previous work on cache-aware approaches to real-time scheduling [14]. We focus more on exploiting the shared caches to minimize the overhead of task splitting, rather than choosing cache-collaborative tasks to run in parallel. The partitioning algorithm may be modified to choose cache-collaborative tasks to be co-located on the same processing core. However, the effects of such partitioning schemes is the subject of future research.

III. Task Partitioning

In this section, we first introduce the necessary notation, and introduce the basic idea behind task-splitting.

A. Notation

We shall use the following notation throughout this paper.

We consider a task-set $\{\tau_1, \tau_2, \cdots, \tau_n\}$ comprising of n periodic tasks. This task-set must be run on m processor cores.

We use the classical (C,T,D) model to represent the parameters of a task τ, where C is the worst-case computation time of each job of τ, T is the period of τ, and D is the deadline of each job of τ relative to job release time. Tasks $\tau_i : (C_i, T_i, D_i)$ are ordered such that $i < j$ implies $D_i < D_j$.

The *utilization* U of task (τ) is given by $\frac{C}{T}$.

The *size*[2] S of a task τ is given by $\frac{C}{D}$. The size of a task quantifies the peak processing demand posed by an individual job of a task. If $T = D$, then $U = S$.

B. Partitioning Tasks

The task partitioning problem is that of dividing the given set of tasks, $\{\tau_1, \tau_2, \cdots, \tau_n\}$, into m subsets such that each subset is schedulable on a single processing core.

The problem of task partitioning involves two components:

1) **Bin-Packing**: Dividing the taskset into m sub-sets.
2) **Uniprocessor Schedulability**: The sub-problem of ensuring that each subset is schedulable.

We briefly describe the classical bin-packing problem. Each bin B_j is of unit size, and each object O_i in the list of objects $\{O_1, O_2, \cdots, O_n\}$ to be packed has a size S_i ranging from 0 to 1. The bin-packing problem is to allocate the n objects to m bins, subject to the following constraint:

$$\forall j (1 \leq j \leq m) \sum_{\forall O_i \in B_j} S_i \leq 1$$

The average size (AS) for a given bin-packing problem instance is defined as:

$$AS = \frac{\sum_{i=1}^{n} S_i}{m}$$

The worst-case size-bound SB for bin-packing is defined as the greatest lower bound on the average size AS of all instances of the bin-packing problem that are unsolvable. Therefore, given a problem instance with average size $AS \leq SB$, there exists a solution to the bin-packing problem.[3]

The underlying bin-packing problem becomes the bottleneck for partitioned real-time scheduling schemes. It restricts the worst-case schedulable utilization to 50% per processor (see [10]), although uniprocessor scheduling is known to have a higher utilization bound (69% for RMS, 100% for EDF).

The classical bin-packing problem is simple to solve when any object is permitted to be split into two pieces, the sizes of which sum to the size of the object. The objects can be packed in any order, one bin at a time. When an object does not fit, it is split so that the current bin is exactly filled, and the residual part is placed in the next bin. This continues with each bin being filled to capacity and containing at most one split per bin. The final bin(s) will hold the remaining objects without any further splitting, since $AS \leq 1$. The total number of split objects cannot exceed $m - 1$. This discussion outlines the motivation behind our approach of splitting one task per processor to achieve a utilization considerably higher than the 50% restriction placed by conventional bin-packing when objects are not allowed to be split. In our context of fixed-priority scheduling, each bin (i.e. processor) cannot be filled up to 100% utilization since deadlines can be missed earlier. Since a task τ has three parameters (C, T, D), we need to define an appropriate mechanism for splitting tasks. We would like to do this with an eye towards improving schedulability. In our greedy object-splitting algorithm, we assumed that there was no penalty while splitting. That is, after splitting, the total size of the split objects did not change from the original.

When our real-time tasks are considered for splitting, however, the penalty of splitting can be non-zero. If a task τ is split into τ' and τ'', the subtasks τ' and τ'' will be assigned to different cores and both 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 task τ. While other approaches can be adopted, we assume that each of these subtasks is assigned its own local deadline, and the second subtask will be released when the deadline of the first subtask is reached. For instance, task τ with parameters (C, T, D) will be split into τ' and τ'' such that:

$$\tau' : (C', T, D') \qquad \tau'' : (C - C', T, D - R')$$
$$(D' \leq D, R' \leq D')$$

where, R' represents the worst-case response time of subtask τ' on its allocated processor. This value may be obtained through an analysis of the critical instant of τ', as performed in the exact schedulability test for fixed-priority scheduling. Subtask τ'' will be eligible to execute on its processor R' time-units after τ' becomes eligible on its corresponding processor.

In our scheme, 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. We refer to this scheme as HPTS (Highest-Priority Task 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. If $\tau_h : (C_h, T_h, D_h)$ is split, we generate:

$$\tau' : (C'_h, T_h, D_h) \qquad \tau'' : (C_h - C'_h, T_h, D_h - C'_h)$$

The special property that we exploit is that the highest-priority task τ'_h on a processor under fixed-priority scheduling has its worst-case response time R'_h equal to its worst-case computation time C'_h. The deadline of the second subtask τ'' is therefore maximized.

IV. PDMS_HPTS

In this section, we analyze the performance of Partitioned Deadline-Monotonic Scheduling (PDMS) under deadline-monotonic priority assignments, when used with Highest-Priority Task-Splitting (HPTS). Conventional partitioned deadline-monotonic scheduling algorithms consist of a bin-packing strategy, which allocates tasks to processing cores in some sequence. Each time a task is allocated to a core, a

[2]We use the term *size* to reference the term *density* used earlier in the literature since it corresponds to the bin-packing nature of the problem

[3]This formulation is somewhat different from the original formulation of bin-packing where the objective is to minimize the number of bins to be used. We prefer our formulation in the current context where the number of processing cores is likely to be fixed on a given platform.

schedulability test is invoked for the individual core to ensure that all its allocated tasks are schedulable under deadline-monotonic scheduling.

We utilize a routine called $HPTaskSplit$, which is invoked, whenever the schedulability test on a particular core fails on trying to assign task τ_f to processor P_j. We define the following data structures and operations:

- $Unallocated\_Queue$: Data structure to hold unallocated tasks in the order required by the bin-packing scheme.
- $Allocated\_Queue_j$: Data structure that holds the tasks allocated to processor P_j in priority order.
- $Enqueue(Unallocated\_Queue, \tau)$: Enqueues the task τ back into the unallocated task queue.
- $\tau \leftarrow Dequeue(Unallocated\_Queue)$: Dequeues a task τ from the unallocated task queue.
- $IsEmpty(Unallocated\_Queue)$: Returns TRUE, whenever the unallocated task queue is empty, signaling that there are no more tasks to be scheduled.
- $EnqueueHP(Allocated\_Queue_j, \tau)$: Enqueues task τ in priority order to allocated task queue of processor P_j.
- $\tau \leftarrow DequeueHP(Allocated\_Queue_j)$: Dequeues the highest-priority task τ from allocated task queue of P_j.
- $IsSchedulable(Allocated\_Queue_j)$: Performs the exact schedulability test on the tasks allocated to processor P_j under deadline-monotonic scheduling.
- $(\tau', \tau'') \leftarrow MaximalSplit(Allocated\_Queue(j), \tau)$: Splits task $\tau : (C, T, D)$ into $\tau' : (C', T, D)$ and $\tau'' : (C - C', T, D - C')$, so that processor P_j is maximally utilized when task τ' is added to it.
- $Available_j$: A boolean flag used to indicate whether processor P_j is available for scheduling.

Algorithm 1 $HPTaskSplit$

Require: $Unallocated\_Queue, Allocated\_Queue_j, \tau_f$
  $Removed\_Size \leftarrow 0$
  $Old\_Unallocated\_Queue \leftarrow Unallocated\_Queue$
  $Old\_Allocated\_Queue_j \leftarrow Allocated\_Queue_j$
  $EnqueueHP(Allocated\_Queue_j, \tau_f)$
  while $IsSchedulable(Allocated\_Queue_j) \neq TRUE$ **do**
    $\tau \leftarrow DequeueHP(Allocated\_Queue_j)$
    $Removed\_Size \leftarrow Removed\_Size + S(\tau)$
    if $IsSchedulable(Allocated\_Queue_j) \neq TRUE$ **then**
      $Enqueue(Unallocated\_Queue, \tau)$
    end if
  end while
  $(\tau', \tau'') = MaximalSplit(Allocated\_Queue_j, \tau)$
  if $(Removed\_Size - S(\tau')) \geq S(\tau_f)$ **then**
    $Unallocated\_Queue \leftarrow Old\_Unallocated\_Queue$
    $Allocated\_Queue_j \leftarrow Old\_Allocated\_Queue_j$
    $Available_j \leftarrow FALSE$
    return
  end if
  $EnqueueHP(Allocated\_Queue_j, \tau')$
  $Enqueue(Unallocated\_Queue, \tau'')$
  $Available_j \leftarrow FALSE$
  return

The $IsSchedulable$ algorithm is readily derived from the exact schedulability tests for fixed-priority preemptive scheduling. Trivial utilization bound tests can be used to determine `MaximalSplit`. Although such tests are faster to perform, they will significantly reduce the average-case performance. A better implementation of $MaximalSplit(Allocated\_Queue_j, \tau)$ could perform a binary search to find the maximum computation time (C') of the task τ, at which the $IsSchedulable(Allocated\_Queue_j)$ test is exactly satisfied.

A more sophisticated version of $MaximalSplit$ uses a variation of the exact schedulability test for fixed-priority preemptive scheduling. In order to compute the maximum-computation time C' sustainable for task τ', we consider each task τ_k in $Allocated\_Queue_j$ and compute the maximum value C' of τ' at which τ_k still meets its deadlines[4].The minimum value of C' over all tasks τ_k in $Allocated\_Queue_j$ gives the maximum sustainable computation time (C') of τ. We can then create τ' to be $\tau' : (C', T, D)$ and $\tau'' : (C - C', T, D - C')$ from the original task-parameters of $\tau : (C, T, D)$.

A. Analysis of HPTS

We now provide an analysis of HPTS when the deadlines of given task-set $\tau : \{\tau_1, \tau_2, .., \tau_n\}$ are such that $\forall i,\ D_i = T_i$. These task-sets are referred to as implicit-deadline task-sets.

When a task τ is split, it results in sub-tasks with $D \leq T$, which are referred to as constrained-deadline task-sets.

A given task-set $\{\tau\}$ is *schedulable* by a scheduling algorithm A, if all the jobs released by all the tasks in $\{\tau\}$ meet their deadlines. This is denoted as $Schedulable(A, \{\tau\})$.

Let $NS(ID, A)$ represent the collection of all finite sets of implicit-deadline (ID) tasks that are not schedulable by algorithm A.

Let $NS(CD, A)$ represent the collection of all finite sets of constrained-deadline (CD) tasks that are not schedulable by algorithm A.

The *utilization bound* UB of a scheduling algorithm A is defined as the greatest lower bound (*glb*) of utilizations of *all* the task-sets not schedulable by A. Formally, the utilization bound of RMS for implicit-deadline (ID) tasks is given by:

$$UB(ID, RMS) = glb\{ \sum_{\forall \tau_i \in \{\tau\}} U(\tau_i) | \forall \{\tau\} \in NS(ID, RMS)\}$$

Also, $\forall \{\tau\} \in ID$,

$$\sum_{\forall \tau_i \in \{\tau\}} U(\tau_i) < UB(ID, RMS) \Longrightarrow Schedulable(RMS, \{\tau\}) \quad (1)$$

The size bound SB of a scheduling algorithm A is defined as the greatest lower bound (*glb*) of sizes of *all* the task-sets not schedulable by A.

The size bound of DMS for implicit-deadline (ID) tasks is given by:

$$SB(ID, DMS) = glb\{ \sum_{\forall \tau_i \in \{\tau\}} S(\tau_i) | \forall \{\tau\} \in NS(ID, DMS)\}$$

[4] C' can be obtained by computing the maximum computation time of τ at which the worst-case response-time of τ_k coincides with a feasible scheduling point (as defined in [17]).

Also, $\forall\{\tau\} \in ID$,

$$\sum_{\forall\tau_i\in\{\tau\}} S(\tau_i) < SB(ID, DMS) \Longrightarrow Schedulable(DMS, \{\tau\}) \quad (2)$$

The size bound of DMS for constrained-deadline (CD) tasks is given by:

$$SB(CD, DMS) = glb\{\sum_{\forall\tau_i\in\{\tau\}} S(\tau_i) | \forall\{\tau\} \in NS(CD, DMS)\}$$

Also, $\forall\{\tau\} \in CD$,

$$\sum_{\forall\tau_i\in\{\tau\}} S(\tau_i) < SB(CD, DMS) \Longrightarrow Schedulable(DMS, \{\tau\}) \quad (3)$$

The following lemmas describe the relationship between the size bound and the utilization bound of each processor.

Theorem 1: *The size bound for uniprocessor deadline-monotonic scheduling (DMS) on constrained-deadline (CD) task-sets is equal to the utilization bound of rate-monotonic scheduling (RMS) on implicit-deadline (ID) task-sets.*

$$(SB(CD, DMS) = UB(ID, RMS))$$

Proof: Considering implicit-deadline tasks, Rate-Monotonic Scheduling is identical to Deadline-Monotonic Scheduling, since task periods equal their deadlines. It follows that $UB(ID, RMS) = SB(ID, DMS)$, since implicit-deadline tasks have a size equal to their utilization.

For implicit-deadline task-sets,

$$\sum_{i=1}^{n} \frac{C_i}{T_i} < UB(ID, RMS) = SB(ID, DMS) \Longrightarrow Schedulable(RMS, \{\tau_1, \tau_2, ..., \tau_n\}) \text{ (from (1))}$$

Recall that implicit-deadline task-sets (I) are a subset of constrained-deadline task-sets (C). Therefore,

$$SB(CD, DMS) \leq SB(ID, DMS) = UB(ID, RMS) \quad (4)$$

For constrained-deadline task-sets,

$$\sum_{i=1}^{n} \frac{C_i}{D_i} < SB(CD, DMS) \Longrightarrow Schedulable(DMS, \{\tau_1, \tau_2, ..., \tau_n\}) \text{ (from (3))}$$

Now, consider if there exists a constrained deadline task-set $\{\tau_1, \tau_2, ..., \tau_n\}$ that is *not* schedulable under DMS but satisfies

$$\sum_{i=1}^{n} \frac{C_i}{D_i} < UB(ID, RMS)$$

Each constrained-deadline task τ in $\{\tau_1, \tau_2, ..., \tau_n\}$ can be transformed into a pessimistic implicit-deadline task $\tau*$, by artificially shortening the task period T to equal its deadline D. Under this transformation (C, T, D) to (C, D, D), the task size remains unchanged as the deadlines are not modified. Rate-monotonic priority assignments of $\{\tau*\}$ correspond to the deadline-monotonic priority assignments of $\{\tau\}$. If the transformed task-set $\{\tau*\}$ has a total utilization less than UB(ID,RMS), it is schedulable under RMS. However, the nature of the transformation also implies that the original task-set $\{\tau\}$ should be schedulable under DMS. This contradicts the assumption that $\{\tau\}$ is not schedulable, therefore

$$SB(CD, DMS) \geq UB(ID, RMS) = SB(ID, DMS) \quad (5)$$

From Inequalities 4 and 5, it follows that

$$SB(CD, DMS) = UB(ID, RMS) (= SB(ID, DMS))$$ ■

We now provide a brief description of the `HPTaskSplit` algorithm. `HPTaskSplit` is invoked with τ_f, only when the schedulability test on a particular processor P_j fails on assigning task τ_f. `HPTaskSplit` forcefully enqueues task τ_f to the allocated queue of P_j. While P_j remains unschedulable, `HPTaskSplit` continuously removes tasks from the allocated queue of P_j in the order of decreasing priorities, till P_j becomes schedulable. `MaximalSplit` is then invoked on the task τ, which was the last to be removed from P_j. If the total size removed from P_j to allocate τ_f is greater than the size of τ_f itself, it is not beneficial to add τ_f to P_j. `HPTaskSplit` therefore restore the old queues under such a condition, and decides not to invoke `MaximalSplit`.

Lemma 2: *The total size on each processing core P_j determined to be unavailable for scheduling by $HPTaskSplit$ is greater than or equal to $SB(CD, DMS) - \epsilon$ where* [5] $\epsilon \to 0$.

$$(Available_j = FALSE) \Longrightarrow \sum_{\tau_i\in P_j} S(\tau_i) \geq SB(CD, DMS) - \epsilon, \text{ where } (\epsilon \to 0)$$

Proof: Two cases must be considered.

Case 1: `HPTaskSplit` decides to perform `MaximalSplit` on task τ, allocates the first piece τ' to processing core P_j and adds the second piece τ'' back to the $Unallocated\_Queue$.

Since `MaximalSplit` was called, the processing core P_j is maximally utilized. This means that increasing the computation time of sub-task τ' any further would render the tasks allocated to P_j unschedulable. Therefore, from Equation 3, the total size MS_{tot} of tasks allocated to the maximally utilized processor P_j must be greater than or equal to $SB(CD, DMS) - \epsilon$ (where $\epsilon \to 0$).

$$MS_{tot} = \sum_{\tau_i^{case\,1}\in P_j} S(\tau_i) \geq SB(CD, DMS) - \epsilon \text{ where } \epsilon \to 0$$

Case 2: `HPTaskSplit` decides not to invoke `MaximalSplit`, and instead restores the old allocated and unallocated queues.

This scenario occurs when the total size S_{tot} of the currently allocated tasks to P_j is greater than or equal to the total size obtainable through `MaximalSplit` (MS_{tot}).

$$S_{tot} = \sum_{\tau_i^{case\,2}\in P_j} S(\tau_i) \geq MS_{tot}$$

Using the analysis from previous case for MS_{tot},

$$S_{tot} = \sum_{\tau_i^{case\,2}\in P_j} S(\tau_i) \geq MS_{tot} \geq SB(CD, DMS) - \epsilon \text{ where } \epsilon \to 0$$

■

We now prove some useful properties about splitting the highest-priority task. First, the next Lemma proves that whenever task-splitting is performed, the operation does not increase the size of the unallocated task-set.

Lemma 3: *When a task $\tau : (C, T, D)$ is split into sub-tasks $\tau' : (C', T, D)$ and $\tau'' : (C - C', T, D - C')$, the sizes of*

[5] To improve readability, we will assume $\epsilon = 0$ in our subsequent usage of size bound.

both τ' and τ'' are less than or equal to that of τ. That is, $S(\tau') \leq S(\tau)$ and $S(\tau'') \leq S(\tau)$.

Proof: ($\forall C', 0 \leq C' \leq C$ and $\forall C, C \leq D$)

We have, $S(\tau) = \frac{C}{D}$,

$S(\tau') = \frac{C'}{D} \leq \frac{C}{D} = S(\tau)$ and $S(\tau'') = \frac{C-C'}{D-C'} \leq \frac{C}{D} = S(\tau)$ ■

Lemma 4: *When a task $\tau : (C, T, D)$ is split into sub-tasks $\tau' : (C', T, D)$ and $\tau'' : (C - C', T, D - C')$,*

1) $S(\tau'') = \frac{S(\tau)-S(\tau')}{1-S(\tau')}$
2) $S(\tau') + S(\tau'') \leq 2(1 - \sqrt{1 - S(\tau)})$

Proof: For simplicity of notation, we let $S(\tau) = S$, $S(\tau') = S' = x$, and $S(\tau'') = S''$

$$S'' = \frac{C - C'}{D - C'} = \frac{\frac{C}{D} - \frac{C'}{D}}{\frac{D}{D} - \frac{C'}{D}} = \frac{S - x}{1 - x}, \quad (6)$$

$$S' + S'' = x + \frac{S-x}{1-x}$$

(hence claim (1) is established)

Maximizing w.r.t. x subject to $0 \leq x \leq S$ yields,

$$S' = S'' = 1 - \sqrt{1 - S},$$

hence, $S(\tau') + S(\tau'') \leq 2(1 - \sqrt{1 - S(\tau)})$. ■

The penalty due to task-splitting is the increase in the size of the split task, which is quantified as:

$$\delta = S(\tau') + S(\tau'') - S(\tau) \leq 2(1 - \sqrt{1 - S(\tau)}) - S(\tau) \quad (7)$$

The incurred penalty δ is an increasing function of the task size $S = S(\tau)$. Specifically this shows that the penalty of task splitting is as low as 2% when the maximum size of any task in the system is less than or equal to 25%.

B. Utilization Bound

The *utilization bound* $UB_m(A)$ of a multi-processor scheduling algorithm (A) is defined as the greatest lower bound (*glb*) of per-processor utilization of *all* the task-sets with individual task-utilizations less than or equal to 1, which are not schedulable by A.

Considering implicit-deadline task-sets (ID) scheduled on m processors,

$$UB_m(ID, A) = glb\{\frac{\sum_{\forall \tau_i \in \{\tau\}} U(\tau_i)}{m} | \forall \{\tau\} \in NS(ID, A)\}$$

Also, $\forall \{\tau\} \in \{ID, m\}, \forall \tau_i \in \{\tau\}$,

$$U(\tau_i) \leq 1 \wedge \sum_{\forall \tau_i \in \{\tau\}} U(\tau_i) < m * UB_m(ID, A) \Longrightarrow Schedulable(A, \{\tau\}) \quad (8)$$

The *size bound* $SB_m(A)$ of a multi-processor scheduling algorithm (A) is defined as the greatest lower bound (*glb*) of per-processor size of *all* the task-sets with individual task-sizes less than or equal to 1, which are not schedulable by A.

Considering constrained-deadline task-sets (CD) scheduled on m processors,

$$SB_m(CD, A) = glb\{\frac{\sum_{\forall \tau_i \in \{\tau\}} S(\tau_i)}{m} | \forall \{\tau\} \in NS(CD, A)\}$$

Also, $\forall \{\tau\} \in \{CD, m\}, \forall \tau_i \in \{\tau\}$,

$$S(\tau_i) \leq 1 \wedge \sum_{\forall \tau_i \in \{\tau\}} S(\tau_i) < m * SB_m(CD, A) \Longrightarrow Schedulable(A, \{\tau\}\}) \quad (9)$$

We now restrict our discussion to task-sets with task periods equal to their deadlines. We are interested in finding the utilization bound $UB_m(ID, PDMS\_HPTS)$ (equal to $SB_m(ID, PDMS\_HPTS)$ since period = deadline), below which all given task-sets are schedulable by any multiprocessor algorithm in PDMS_HPTS.

Let us consider any task-set which is not schedulable by $PDMS\_HPTS$. The total size (S_{tot}) is given by:

$$S_{tot} = \sum_{i=1}^{n} S(\tau_i)$$

Task-splitting adds a maximum of $(m-1)$ new tasks (one per each core, except the last). Therefore, in the worst-case, it increases the total size of the task-set by $(m-1) * \delta$. The cumulative task size of the task-set after task-splitting is: $CS^{TS} = S_{tot} + (m-1) * \delta = \sum_{i=1}^{n} S(\tau_i) + (m-1) * \delta$.

For the entire task-set to be unschedulable, the $Available_j$ flag must be $FALSE$ for all m processing cores. Invoking Lemma 2 however, we see that each processing core must have a total size at least equal to $SB(CD, DMS)$, therefore $\sum_{i=1}^{n} S(\tau_i) + (m-1) * \delta \geq m * SB(CD, DMS)$.

Invoking Theorem 1 and re-writing, we obtain $\sum_{i=1}^{n} S(\tau_i) \geq m * UB(ID, RMS) - (m-1) * \delta$,

therefore $SB_m(CD, PDMS\_HPTS)$ is at least:

$$SB_m(CD, PDMS\_HPTS) \geq \frac{\sum_{i=1}^{n} S(\tau_i)}{m} \geq UB(ID, RMS) - \delta + \frac{\delta}{m}.$$

Recall that the collection of finite implicit-deadline task-sets (I) is a subset of the collection of finite constrained-deadline tasks (C), therefore

$$SB_m(ID, PDMS\_HPTS) \geq \frac{\sum_{i=1}^{n} S(\tau_i)}{m} \geq UB(ID, RMS) - \delta + \frac{\delta}{m}.$$

When all tasks have periods equal to their deadlines, the size-bound is the same as the utilization bound; therefore,

$$UB_m(ID, PDMS\_HPTS) \geq UB(ID, RMS) - \delta + \frac{\delta}{m}.$$

The utilization bound for RMS on implicit-deadline tasks is 0.6931 [1], therefore

$$UB_m(ID, PDMS\_HPTS) \geq 0.6931 - \delta + \frac{\delta}{m}.$$

As $m \to \infty$, we get the utilization bound to be $0.6931 - \delta$. As we have shown earlier, when the individual task sizes are less than or equal to 25%, the size penalty (δ) of task splitting is less than 2%, leading to an utilization bound of 67.31% for such task-sets.

C. General Case

Lemma 5: *Given a task-set consisting of $r+1$ tasks, one of which has size S and all of which have deadlines less than or equal to their periods (CD), then the corresponding size bound for Deadline Monotonic Scheduling, $SB(CD, DMS|S|r)$ is*

$$SB(CD, DMS|S|r) = r((\tfrac{2}{1+S})^{\frac{1}{r}} - 1) + S.$$

Letting $r \to \infty$, the asymptotic bound is given by

$$SB(CD, DMS|S) = \ln(\tfrac{2}{1+S}) + S.$$

Proof: Consider a pessimistic task-set $\{\tau*\}$ obtained by artificially shortening the task period of each task to equal its deadline. The utilization of the $\tau*$ task-set is the same as the size of the original τ task-set. Furthermore, the Rate Monotonic priorities for the $\tau*$ task-set are the same as the Deadline Monotonic priorities for the original τ task-set. One can use the results of [16] who analyzed the behavior of the polling server to find a utilization bound for the Rate Monotonic scheduling of the $\tau*$ task-set. Specifically, for $r+1$ tasks, one of which has size S (utilization in the $\tau*$ task-set), the bound is given by

$$SB(CD, DMS|S, r) = UB(ID, RMS|S, r) = r((\frac{2}{1+S})^{\frac{1}{r}} - 1) + S. \quad (10)$$

Letting $r \to \infty$, we find the asymptotic bound

$$SB(CD, DMS|S) = UB(ID, RMS|S) = \ln(\tfrac{2}{1+S}) + S.$$

■

Theorem 6: *The utilization bound of the PDMS_HPTS class of algorithms for task-sets with task deadlines equal to task periods is at least 60%. That is,*

$$UB_m(ID, PDMS\_HPTS) \geq 60\%$$

Proof: Since task deadlines equal periods, the size bound and the utilization bound for $PDMS\_HPTS$ are the same.

Tasks with individual sizes greater than or equal to 60% can be allocated to their own processors as they have a size greater than or equal to the claimed size bound. These tasks can then be ignored for analyzing the worst-case size bound, since they can only increase the overall utilization.

We now need to consider the two-cases of `HPTaskSplit` .

Case 1: `HPTaskSplit` does not call `MaximalSplit` and returns the original allocated and unallocated task-queues.

In this scenario, from Lemma 2, the total size S_{tot} allocated to P_j is at least $SB(CD, DMS)$. From Theorem 1,

$$S_{tot} \geq SB(CD, DMS) = UB(ID, RMS) = 0.6931$$

These processing cores are allocated total sizes greater than or equal to 60% and also can be ignored.

Case 2: `HPTaskSplit` decides to call `MaximalSplit` .

Consider a scenario in which a processor, P_j, overflows. Let the task being split be denoted by τ. The split piece of this task remaining on this processor is denoted by τ' and the remaining piece of the task is denoted by τ''. The algorithm ensures through task-splitting that the individual processor is maximally utilized, therefore the total size of the tasks allocated to the processor must be greater than or equal to the size-bound for DMS.

P_j has a given task τ' of size $S(\tau')$. Hence, from Lemmas 2 and 5, the total size S_{tot} of tasks allocated to the P_j satisfies

$$S_{tot} \geq SB(CD, DMS|S(\tau')) = \ln(\frac{2}{1+S(\tau')}) + S(\tau'). \quad (11)$$

Task-splitting increases the size of the task-set by δ, therefore subtracting (Equation 7) $(S(\tau') + S(\tau'') - S(\tau))$ from (Equation 11) gives the total size of the original task-set allocated per-processor (S_{orig}),

$$S_{orig} \geq \ln(\frac{2}{1+S(\tau')}) + S(\tau) - S(\tau''). \quad (12)$$

Using (claim 1) of Lemma 4, we obtain

$$S_{orig} \geq \ln(\tfrac{2}{1+S(\tau')}) + S(\tau) - \tfrac{S(\tau)-S(\tau')}{1-S(\tau')}.$$

For simplicity, let $S = S(\tau)$ represent the size of task τ and $x = S(\tau')$ represent the size of task τ'. Then,

$$SB_m(CD, PDMS\_HPTS) = \ln(\frac{2}{1+x}) + S - \frac{S-x}{1-x} \quad (13)$$

Observe that (13) is a non-increasing function of $S = S(\tau)$. The minimum lower bound of $SB_m(CD, PDMS\_HPTS)$, therefore, occurs at the maximum value of $S(\tau)$. We are allocating individual processors to all tasks with size greater than or equal to $SB_m(CD, PDMS\_HPTS)$, therefore the maximum value of $S(\tau)$ is $SB_m(CD, PDMS\_HPTS)$. Therefore, using $SB_m(CD, PDMS\_HPTS) = S$

$$\ln(\frac{2}{1+x}) - \frac{S-x}{1-x} = 0. \quad (14)$$

Rewriting S as a function of x gives

$$S = (1-x)\ln(\frac{2}{1+x}) + x. \quad (15)$$

To find the minimum value of S, differentiate S w.r.t. x and equate it to 0, to find the minimizing value.

$$\ln(\frac{2}{1+x}) - \frac{2x}{1+x} = 0. \quad (16)$$

Substituting $t = \frac{2}{1+x}$,

$$\ln(t) - (2-t) = 0 \Longrightarrow \ln(t) + t = 2 \Longrightarrow te^t = e^2$$

Solving this equation gives[6] $t = LambertW(e^2)$

Re-substituting $x = \frac{2-t}{t}$, we get

$$x = S(\tau') = \frac{2 - LambertW(e^2)}{LambertW(e^2)} = 0.2837, \quad (17)$$

The second derivative of (15) is positive, therefore the value of x given above corresponds to the minimum value of S.

Corresponding value of $S = S(\tau) = SB_m(CD, PDMS\_HPTS)$ is 60.03%. The initially given task-set is assumed to have deadlines equal to their period, the size bound for such task-sets is the same as the utilization bound. We have accommodated the penalties due to task-splitting on each processor, and therefore we conclude that the utilization bound of PDMS_HPTS on task-sets with implicit deadlines is at least 60.03% (with the exception of the last processor, which can achieve up to 69.31%).

$$UB_m(ID, PDMS\_HPTS) = 0.6003 + \tfrac{0.0928}{m}.$$

As $m \to \infty$, the $UB_m(ID, PDMS\_HPTS) \to 60.03\%$.

■

We have shown above that for the general class of PDMS algorithms, HPTS can achieve an utilization of at least 60.03%. Depending on the specific scheme used for bin-packing, higher utilization bounds may be achievable.

The usefulness of this result is the flexibility in choosing the bin-packing scheme. The task allocation phase can therefore be guided by multiple heuristics like (i) minimizing communication across cores, (ii) co-locating cache-collaborative

[6] $LambertW(t)$ is the inverse of the function $f(t) = te^t$.

tasks, and (iii) allocating replicas of the same task on different processing cores. The HPTS technique ensures that even when any of these different heuristics are used, the system can achieve a utilization bound of 60%. Task-splitting may displace previously allocated tasks under some bin-packing schemes; this is a trade-off to achieve higher system utilization. The bin-packing algorithm could decide to not use HPTS on certain processors and settle for a lower utilization value, whenever the already allocated tasks should not be displaced. We believe that this flexibility would be very useful in practical applications with various constraints on allocating tasks.

D. Recent Work on Task-Splitting

A brief comparison of PDMS_HPTS with recent work on task-splitting [19], [20], [21] follows:

1) Each processing core uses purely fixed-priority preemptive scheduling, in contrast to mixed-priority schemes considered elsewhere. Our approach is therefore readily usable on fixed-priority reservation-based OSs like Linux/RK [22].

2) Under HPTS, task-splitting represents purely sequential migration across processors. For split tasks, when the reserve allocated on a particular processor completes, it is subsequently released on the next processor. In our current implementation for Linux/RK, when the task reserve is exhausted, the OS scheduler explicitly uses the already existing `sched_set_affinity()` function to migrate the task. No other support for inter-processor synchronization is required to ensure that the same task is not scheduled in parallel.

3) No special dispatching support or explicit suspensions are required, thus tasks may be even split across critical sections. However, the migration overhead increases the duration of the critical section, and hence it adversely affects the other tasks waiting on the critical section. After migration, the critical section becomes a global critical section [23] and hence remote-blocking effects need to be accounted for. In practice, critical sections are small and splitting may therefore be avoided at the cost of a slight loss in utilization.

4) Our task-splitting involves a single split per-processor thus trivially bounding the additional number of total preemptions. In this respect, our approach is similar to that of [19], although we do not provide tunable parameters to adjust the number of preemptions.

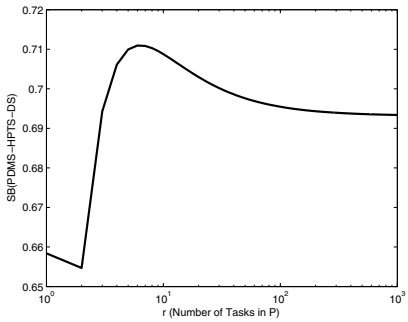
V. PDMS_HPTS_DS

In this section, we present a specific instance of the $PDMS\_HPTS$ class of algorithms in which tasks are allocated in decreasing order of size ($PDMS\_HPTS\_DS$). In describing this algorithm, we will use the same data-structures and notation used in the previous section to analyze PDMS_HPTS. $Enqueue$ and $Dequeue$ are modified as follows to reflect the decreasing size order of task allocation:

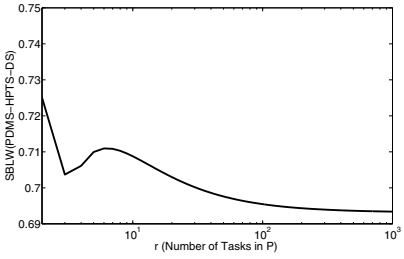
- $EnqueueDS(Unallocated\_Queue, \tau)$: Enqueues task τ in decreasing size order to unallocated task queue.
- $\tau \leftarrow DequeueDS(Unallocated\_Queue)$: Dequeues task τ with largest size ($\frac{C}{D}$) from unallocated task queue.

Algorithm 2 PDMS_HPTS_DS

Require: $Unallocated\_Queue$
  $j \leftarrow 1$
  $Allocated\_Queue_j \leftarrow EMPTY$
  while $IsEmpty(Unallocated\_Queue) \neq TRUE$ **do**
    $\tau \leftarrow DequeueDS(Unallocated\_Queue)$
    $Old\_Allocated\_Queue_j \leftarrow Allocated\_Queue_j$
    $EnqueueHP(Allocated\_Queue_j, \tau)$
    if $IsSchedulable(Allocated\_Queue_j) \neq TRUE$ **then**
      $Allocated\_Queue_j \leftarrow Old\_Allocated\_Queue_j$
      $HPTaskSplit(Unallocated\_Queue, Allocated\_Queue_j, \tau)$
      $j \leftarrow j + 1$
    end if
  end while



(a) SB(CD,PDMS_HPTS_DS) as a function of r



(b) SBLW(CD,PDMS_HPTS_DS,$\sqrt{2}-1$) as a function of r

A. Worst-Case Behavior

The following theorem shows that a larger utilization bound holds for the PDMS_HPTS_DS algorithm.

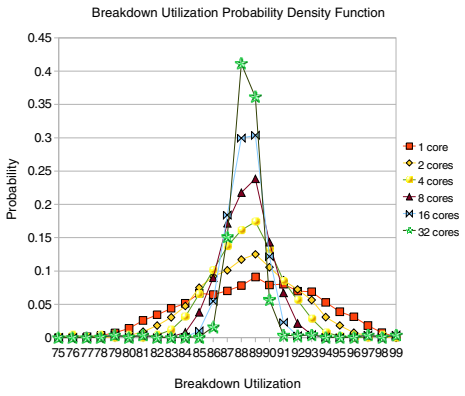
Theorem 7: *The utilization bound of PDMS_HPTS_DS for task-sets with implicit deadlines is at least 65.47%.*

Proof: The proof can be found in [25]. ■

Let 'r' represent the number of tasks allocated to a processor core P. The minimum values of $SB(CD, PDMS\_HPTS\_DS)$ as a function of r are shown in the logarithmic plot Fig. 2(a).

The minimum value of SB(CD,PDMS_HPTS_DS), 0.6547, occurs at $r = 2$. The value of $SB(CD, PDMS\_HPTS\_DS)$ converges to 0.693 as $r \rightarrow \infty$. The size bound of PDMS_HPTS_DS is therefore 65.47%. For all tasksets with implicit deadlines, the utilization bound of PDMS_HPTS_DS is the same as the size bound, and hence at least 65.47%.

The asymptotic behavior suggests that PDMS_HPTS_DS achieves a utilization bound of 0.693 when the task-set is



(c) Breakdown Utilization PDF (PDMS_HPTS_DS)

composed of sufficiently lightweight tasks. Space limitations prevent a presentation of the detailed analysis, but it can be easily derived that the size-bound of PDMS_HPTS_DS, for lightweight tasks with maximum utilization α, is given by:

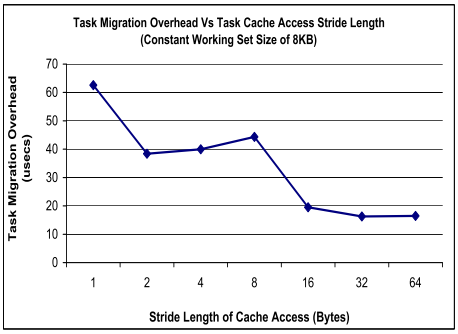
$$SBLW(CD, PDMS\_HPTS\_DS, \alpha)$$
$$= r((\tfrac{2}{1+x})^{\frac{1}{r}} - 1) + (1 - MIN(\alpha, (\tfrac{2}{1+x})^{\frac{1}{r}} - 1))\tfrac{x}{1-x}$$

The behavior of this function, when $\alpha = \sqrt{2} - 1 = 0.414$ is given in Fig. 2(b). The minimum value of $SBLW(CD, PDMS\_HPTS\_DS, \sqrt{2} - 1)$ is 0.693. The choice of $\alpha = 0.414$ implies that each processing core is forced to have more than 2 tasks, which can be guaranteed on a significant number of practical task-sets. Thus, the PDMS_HPTS_DS algorithm achieves the classical Rate-Monotonic Scheduling utilization bound of 0.6931, for sufficiently lightweight tasks (less that 41.4%).

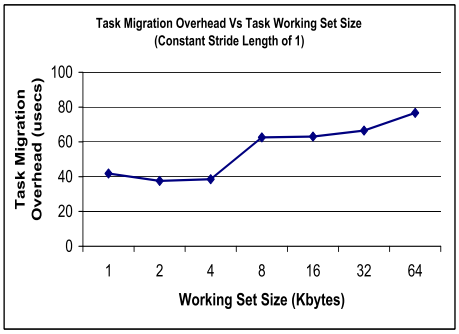
B. Average-Case Behavior

We have so far analyzed the worst-case performance of the PDMS_HPTS_DS algorithm and obtained utilization bounds on its adversarial task-sets. The worst-case performance occurs when all the task periods are related in a non-harmonic fashion and the task-splitting happens to divide the tasks into the worst possible pieces. These conditions represent extreme situations, and therefore the average-case performance of PDMS_HPTS_DS is expected to be far better than its utilization bound of 65%. In this section, we determine the probability density function of task-set breakdown utilizations, when the task-periods are chosen in an uniformly random fashion. This methodology is consistent with classical average-case analysis, in which the mean-performance of an algorithm is evaluated for a randomly chosen input. Earlier sections have shown the worst-case analysis. The best case is achieved when task periods are harmonic and each core is completely filled.

We studied the average-case performance of the PDMS_HPTS_DS algorithm on randomly generated task-sets and the break down utilization values were computed as given in [17]. Task period T for each task was chosen in an uniformly random fashion from the interval $[100, 5000]$. Computation time C for each task was chosen using a uniform distribution from the interval $[0, 0.4T]$. Tasks were generated till the total utilization exceeded $m * 100\%$ (where m is the number of cores). These computation times were then proportionally scaled down to compute



(d) Migration overheads *vs.* Cache access stride lengths



(e) Migration overheads *vs.* Working-set sizes

the break down utilization (for more details see [17]). Although a slight decrease in the average utilization was seen with increasing number of processors, the value converges around 88%. The probability density function of the task-set breakdown utilizations is given in Fig.2(c). As we can see, with an increasing number of processors, the variance in the breakdown utilization decreases, indicating that more and more task-sets reach the average-case utilization. This behavior is due to the fact that as the number of cores increases, there is more flexibility to achieve good packings.

VI. Case Study of Task-Splitting

In this section, we present a case study of task migration on the Intel Core 2 Duo processor to characterize the practical overheads of task-splitting. The processor has 2 cores with a private L1-caches and a shared L2-cache. The L1-cache (64KB) is a split cache with both Instruction and Data caches having a size of 32KB each. The L2-cache is a unified cache of size 4MB. Both L1 and L2 are on-chip cache resources, and the cache line size is 64 bytes. There are 512 cache lines in the individual L1 caches (32KB). The access time for L1 cache is about 3 cycles, while the access time for the L2 cache is anywhere from 11 to 14 cycles. The data bus between the L1 and L2 cache has a width of 256 bits.

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 1KB to 64KB) and stride-lengths (1 to 64bytes). Performance of these workloads is shown in Figures 2(d) and 2(e).

The overall overhead was acceptably low (less than 80 microseconds) for these cache-workloads. These workloads exercised only the data cache, and the instruction cache effects

Application	*Time Taken (ms) w/o Task Splitting*	*Time Taken (ms) w/ Task Splitting*
MPlayer	5.79	5.84
GL Gears	4.5	4.5

TABLE I
TASK SPLITTING OVERHEAD ON REAL-WORLD APPLICATIONS

were neglected in these experiments. The timing measurements were done at a fixed processor frequency of 2GHz.

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. 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 some real-world applications and benchmarks, we looked at the media player application called *MPlayer* and the standard OpenGL *Gears* benchmark in isolation. The results of these experiments are shown in Table 1. The overhead due to task-splitting was negligible in both the cases.

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 was used and a wmv3 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 time taken by a single call to $decode\_frame$ was 5.79ms. After performing task splitting, the average time taken for a $decode\_frame$ call was increased by approximately 0.05ms.

The reason for the overhead of about 50μs in the case of *MPlayer* is probably the fact that it incurs heavy cache overheads. The video was being played at approximately 25 frames/second (a period of 40ms), and this cache overhead translates to 0.125% in terms of the *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.

There was no real overhead seen with the *Gears* application. This is due to the fact that *Gears* has a very low working set size. Most of the time is spent in rendering the image in the video buffer and the cache is not utilized very frequently.

The evaluation of *MPlayer* and *Gears* shows that the cache overheads due to task-splitting can be expected to be negligible in multi-core platforms. Hence task-splitting is a practical mechanism of improving the overall system utilization in partitioned real-time multi-core scheduling.

VII. CONCLUSIONS AND FUTURE WORK

We have introduced the mechanism of Highest-Priority Task-Splitting (HPTS) and used it to improve the utilization bound of the class of partitioned deadline-monotonic scheduling algorithms (PDMS) from 50% to 60% on implicit-deadline task-sets. A specific instance of this class of algorithms PDMS_HPTS_DS has been shown to achieve a utilization bound of 65% for implicit-deadline task-sets. PDMS_HPTS_DS is also shown to achieve a higher utilization bound of 69% on lightweight implicit-deadline task-sets, where the individual task utilizations are restricted to a maximum of 41.4%. The average-case analysis of PDMS_HPTS_DS using randomly generated task-sets shows that it achieves a very high utilization of 88% on the average. Future work involves extensions to the basic PDMS_HPTS_DS scheme that exploit relationships between the task periods may achieve a higher utilization bound. The cache overhead of task-splitting has been evaluated on the Intel Core 2 Duo. Task migration overheads were seen to be very low on this platform.

ACKNOWLEDGMENTS

We thank Bjorn Andersson and some insightful anonymous reviewers who helped improve earlier versions of this paper.

REFERENCES

[1] C. L. Liu, and J. W. Layland, *"Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment"*, JACM, vol. 20, pp. 46-61, 1973.
[2] S. Lauzac, R. Melhem, and D. Mosse, *"Comparison of global and partitioning schemes for scheduling rate monotonic tasks on a multiprocessor"*, ECRTS, 1998.
[3] S. K. Baruah, Shun-Shii Lin, *"Pfair scheduling of generalized pinwheel task systems"*, Transactions on Computers, 1998.
[4] A. Srinivasan, P. Holman, J. H. Anderson, and S. K. Baruah, *"The case for fair multiprocessor scheduling"*, In Proceedings of the PDPS, 2003.
[5] S. K. Dhall, and C. L. Liu, *"On a Real-Time Scheduling Problem"*, Operations Research, Vol. 26, No. 1, Scheduling(Jan.-Feb.,1978), pp. 127-140.
[6] B. Andersson, S. K. Baruah, and J. Jonsson, *"Static priority scheduling on multiprocessors"*, In Proceedings of RTSS, 2001.
[7] L. Lundberg, *"Analyzing fixed-priority global multiprocessor scheduling"*, In Proceedings of the RTAS, 2002.
[8] T. P. Baker, *"An analysis of EDF schedulability on a multiprocessor"* Parallel and Distributed Systems, IEEE Transactions on , vol.16, no.8, pp. 760-768, Aug. 2005.
[9] S. K. Baruah, N. K. Cohen, C. G. Plaxton, and D. A. Varvel, *"Proportionate progress: A notion of fairness in resource allocation"*, Algorithmica, vol. 15, 1996.
[10] B. Andersson, J. Jonsson, *"The utilization bounds of partitioned and pfair static-priority scheduling on multiprocessors are 50%"*, Proceedings of ECRTS, 2003.
[11] J. M. Lopez, J. L. Diaz, and D. F. Garcia, *"Utilization Bounds for EDF Scheduling on Real-Time Multiprocessor Systems"*, In Proceedings of RTSS, 2004.
[12] B. Andersson, and E. Tovar, *"Multiprocessor Scheduling with Few Preemptions"*, In Proceedings of RTCSA, pp. 322-334, 2006.
[13] S. Kato, and N. Yamasaki, *"Real-Time Scheduling with Task Splitting on Multiprocessors"*, In Proceedings of RTCSA, pp. 441-450, 21-24 Aug. 2007.
[14] J. H. Anderson, J. M. Calandrino, and U. C. Devi, *"Real-Time Scheduling on Multicore Platforms"*, In Proceedings of RTAS, 2006.
[15] R. Rajkumar, *"Task Synchronization In Real-Time Systems"*, Kluwer Academic Publishers, Boston, 1991.
[16] J. K. Strosnider, J. P. Lehoczky, and L. Sha, *"The deferrable server algorithm for enhanced aperiodic responsiveness in hard real-time environments"*, IEEE Transactions on Computers, vol. 44, no. 1, pp. 73-91, Jan. 1995.
[17] J. Lehoczky, L. Sha, and Y. Ding, *"The rate monotonic scheduling algorithm: exact characterization and average case behavior"*, In Proceedings of RTSS, 1989.
[18] Dionisio de Niz, and Raj Rajkumar, *Partitioning Bin-Packing Algorithms for Distributed Real-Time Systems*, Intl. Jnl. of Embedded Systems", 2006.
[19] Bjorn Andersson, Konstantinos Bletsas, "Sporadic Multiprocessor Scheduling with Few Preemptions," In Proceedings of ECRTS, pp. 243-252, 2008.
[20] Bjorn Andersson, Konstantinos Bletsas, Sanjoy Baruah, "Scheduling Arbitrary-Deadline Sporadic Tasks on Multiprocessors," In Proceedings of RTSS, 2008.
[21] S. Kato, N. Yamasaki, "Semi-Partitioned Fixed-Priority Scheduling on Multiprocessors", In Proceedings of RTAS, 2009.
[22] S. Oikawa, R. Rajkumar, "Portable RK: A Portable Resource Kernel for Guaranteed and Enforced Timing Behavior," In Proceedings of RTAS, pp.111, 1999.
[23] R. Rajkumar, L. Sha, J. P. Lehoczky, "Real-time synchronization protocols for multiprocessors, In Proceedings of RTSS, pp.259-269, 1988.
[24] H. Cho, B. Ravindran, E. D. Jensen, "An Optimal Real-Time Scheduling Algorithm for Multiprocessors," In Proceedings of RTSS, 2006.
[25] K. Lakshmanan, R. Rajkumar, "Partitioned Fixed-Priority Preemptive Scheduling for Multi-Core Processors", TR., Department of ECE, CMU, 2008.