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. Specifically, we focus on the partitioned (static binding) approach, which statically allocates each task to one processing core. We also assume fixed-priority preemptive scheduling on each core. The previously established 50% bound for partitioned multiprocessor scheduling is overcome by task-splitting (TS), 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, when only the highest priority task on each processing core is allowed to be split (HPTS). A specific instance of PDMS_HPTS, where tasks are allocated in the decreasing order of size PDMS_HPTS_DS, is shown to have a utilization bound of 65%. The PDMS_HPTS_DS algorithm also achieves a utilization bound of 69% on lightweight tasksets where no single task utilization exceeds 41.4%. The average-case behavior of PDMS_HPTS_DS is studied using randomly generated tasksets, and it is seen to have an average schedulable utilization of 88%. We also characterize the practical 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. Commercial chipmakers including AMD, IBM, Intel and Motorola have already introduced processors with more than one computational core. Massively multi-core processors are also prominently featured in the future product roadmaps of many industry leaders. Thermal and power considerations are forcing the shift of focus from increasing the clock speed to adding more cores per chip. Memory and ILP (Instruction Level Parallelism) bottlenecks are further aiding this trend towards multi-core processors. A major issue with the multi-core technology however is in developing parallel software to effectively utilize the available processing power. In this paper, we consider the problem of scheduling periodic real-time tasks on multi-core processors to better utilize their parallel processing capability.

Multi-core processors largely resemble symmetric multiprocessors (SMPs), therefore existing software experience with SMPs may prove useful. In multi-core processors the computation cores are co-located on the same chip, thus significantly reducing the communication and co-ordination latencies. Inter-core data-sharing is also facilitated in many modern processors through shared levels in the cache hierarchy. Software developed for multi-core processors should exploit these architectural features to achieve better performance.

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 the partitioned approach, each task is statically allocated to a processor and always executes on that processor. A separate ready queue is therefore used on each processor. In the global approach, there is a single global ready queue for the entire system, and the highest priority ready tasks are always executed on the system processors. 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 arising from task splitting. Per-processor run-queues are still used to schedule tasks, thus retaining 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%. A specific instance of this class of algorithms, in which tasks are allocated in the decreasing order of sizes (PDMS_HPTS_DS), can achieve a per-processor utilization bound of 65%. The behavior of PDMS_HPTS_DS on lightweight tasks is better, and a worst-case schedulable utilization of 69% is achieved when the individual task-utilizations are less than 41.4%. When exact schedulability

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

tests are used on the individual processors, PDMS_HPTS_DS is seen to achieve an average schedulable utilization of 88%, which is much higher than its theoretical worst-case performance. In order to characterize the practical overhead of task-splitting, we present a case-study using the Intel Core 2 Duo processor.

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 translate the notion of dividing objects into splitting tasks, and compute the penalty incurred by splitting the highest priority task (HPTS). Utilization bounds are then developed for the class of PDMS algorithms, when the highest priority task is allowed to be split. The PDMS_HPTS_DS algorithm is then presented, its worst-case utilization bounds analyzed and its average-case performance evaluated. Finally, we measure the overheads of task-splitting on the Intel Core 2 Duo processor, and present our concluding remarks.

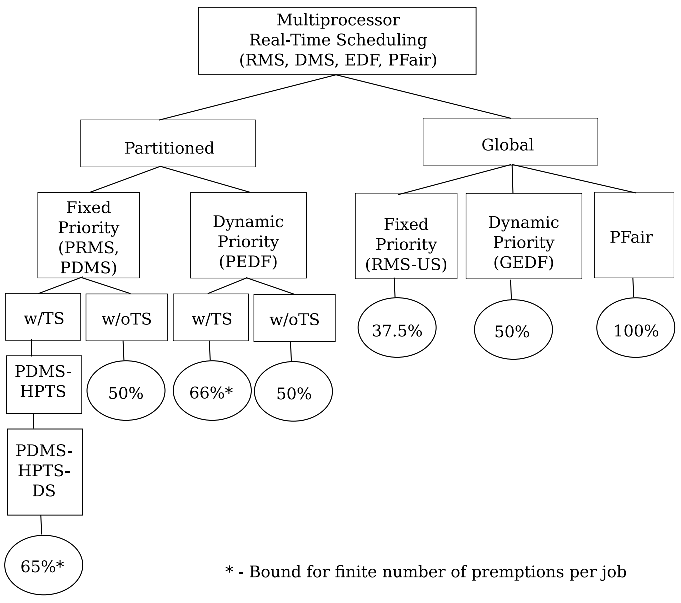

Fig. 1. Design space of Multi-Processor Real-Time Scheduling

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 have their own advantages and disadvantages [2]. Global scheduling schemes can better utilize the available processors, as best illustrated by PFair algorithms [3]. These schemes appear to be best suited for applications with small working set sizes. Their weak processor affinity and preemption overheads need to be managed to fully exploit their benefits[4]. On the other part of the design space, partitioned approaches are severely limited the low worst-case utilization bounds associated with bin-packing problems. The advantage of these schemes is that they have stronger processor affinity, and can 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 heavyweight (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 taskset into a heavyweight 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 massively multicore processors, 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 multicore 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 much 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 splitting the highest priority task (HPTS) and show that this can lead to a utilization bound of 60% for this category of algorithms. A specific instance of this class, where tasks are allocated in the decreasing order of sizes 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 our PDMS_HPTS_DS algorithm:

1) higher worst-case utilization than conventional partitioned multiprocessor scheduling schemes,
2) a bound on the number of tasks straddling across 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 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

A. Notation

We shall use the following notation throughout this paper.

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

We use the standard (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.

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

The *size* 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**: Taskset has to be divided into m sub-sets.
2) **Uniprocessor Schedulability**: The sub-problem of ensuring that each subset is schedulable.

We briefly describe the underlying 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 objects to bins, subject to the following constraint:

$$\sum_{O_i \in B_j} S_i \leq 1$$

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 least upper bound on the average size AS of all instances of the bin-packing problem that are solvable. Therefore, given a problem instance with average size AS less than or equal to SB, there exists a solution to the bin-packing problem.[2]

[2]This formulation is somewhat different from the classical 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.

It is obvious that a set of $m + 1$ objects of size $0.5 + \epsilon$ for any $\epsilon > 0$ cannot be packed in m bins, since at most one of these objects can fit in a bin. Consequently, $SB < 0.5 + \frac{\epsilon}{m}$ for all $\epsilon > 0$ and m. It follows that $SB \leq 0.5$. Conversely, consider any task set with $AS \leq 0.5$, and suppose that it cannot be packed into the m bins. Order the objects in decreasing order, and pack them arbitrarily. Suppose the kth object, O_k, is the first that cannot be packed, that is, if O_k is added to each bin, it will exceed the bin capacity. Note that $S_k < 0.5$, else $\sum_{i=1}^{k} S_i > 0.5m$ contrary to assumption. It follows that $\sum_{i=1}^{k-1} S_i + m \cdot S_k > m$. Consequently $\sum_{i=1}^{n} S_i + (m-1)S_k > m$ or $AS + \frac{m-1}{m} S_k > 1$. Since $S_k < 0.5$, this contradicts the assumption that $AS < 0.5$. Therefore SB = 0.5, and the bin-packing problem becomes the bottleneck for partitioned real-time scheduling schemes. It restricts the worst-case schedulable utilization to 50% per processor, although the uniprocessor scheduling problem is known to have a much higher utilization bound (69% for RMS, 100% for EDF).

Bin-packing is a simple problem when we introduce the possibility that 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 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 object. 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 and the processor can become unschedulable. Since a task τ also has three parameters (C, T, D), we need to define an appropriate mechanism for splitting tasks. And 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 size of the unsplit object.

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 cannot execute at the same time. The first piece τ' must execute first, and only then can τ'' execute. Furthermore, both τ' and τ'' must complete within the same relative deadline. 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 having parameters (C, T, D) will be split into τ' and τ'', such that:

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

$$(D' \leq D, R' \leq D')$$

In this scheme, R' represents the worst-case response time of subtask τ' on its allocated processor. This value may be obtained through an analysis of its critical instant, 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 to execute on its 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 on a processor under fixed-priority scheduling has its worst-case response time R'_h equal to its worst-case computation time C'. 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) when used with 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 schedulability test is invoked for the individual core to ensure that all its allocated tasks are schedulable under deadline-monotonic scheduling.

We utilize the $HPTaskSplit$ routine 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, signalling 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 the 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. A simple 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'_k of τ at which τ_k still meets its deadlines[3] . The minimum value of C'_k 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 first introduce some definitions and notation.

A given taskset is *schedulable* by a scheduling algorithm A, if all the jobs released by all the tasks meet their deadlines.

[3] C'_k can be computed 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])

The *utilization bound* UB of a scheduling algorithm A is defined as the maximum utilization value such that *all* tasksets with total utilization less than or equal to UB are schedulable by A. Formally,

$$\forall\{\tau\}, \sum_{\forall\tau_i\in\{\tau\}} U(\tau_i) \leq UB(A) \Longrightarrow Schedulable(A, \{\tau\}) \tag{1}$$

The size bound SB of a scheduling algorithm A is defined as the maximum size value, such that *all* tasksets with total size less than or equal to SB are schedulable by A. Formally,

$$\forall\{\tau\}, \sum_{\forall\tau_i\in\{\tau\}} S(\tau_i) \leq SB(A) \Longrightarrow Schedulable(A, \{\tau\}) \tag{2}$$

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

Lemma 1: *When all tasks in the considered taskset have deadlines less than or equal to their periods, the size bound for uniprocessor deadline-monotonic scheduling(DMS) is bounded below by the utilization bound of rate-monotonic scheduling(RMS) for this taskset.*

$$\begin{aligned}(\forall\tau_i : (C_i, T_i, D_i) \in \{\tau\}, D_i \leq T_i)\\ \Longrightarrow (SB(DMS) \geq UB(RMS))\end{aligned}$$

Proof: Recall that the Deadline Monotonic scheduling algorithm is the optimal fixed priority scheduling algorithm, that is any task set that can be scheduled by the Rate Monotonic scheduling algorithm can also be scheduled by the Deadline Monotonic scheduling algorithm. It follows that $UB(RMS) \leq UB(DMS)$. Now, recall that task deadlines are less than task periods, hence every task's size is at least as large as the corresponding task utilization. It follows that the size bound must be larger than the utilization bound, i.e. $UB(DMS) \leq SB(DMS)$. ∎

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 $SB(DMS)$.*

$$(Available_j = FALSE) \Longrightarrow \sum_{\tau_i\in P_j} S(\tau_i) \geq SB(DMS)$$

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 (2), the total size(MS_{tot}) of tasks allocated to the maximally utilized processor P_j must be greater than or equal to $SB(DMS)$, since otherwise the size of τ' can be increased.

$$MS_{tot} = \textstyle\sum_{\tau_i^{case\,1}\in P_j} S(\tau_i) \geq SB(DMS)$$

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}).

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

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

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

∎

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 taskset.

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

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

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

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

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

∎

Lemma 4: *When a task $\tau : (C, T, D)$ is split into sub-tasks $\tau' : (C', T, D)$ and $\tau'' : (C-C', T, D-C')$, $S(\tau')+S(\tau'') \leq 2(1 - \sqrt{1 - S(\tau)})$*

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

$$S'' = \frac{C - C'}{D - C'} = \frac{\frac{C}{D} - \frac{C'}{D}}{\frac{D}{D} - \frac{C'}{D}} = \frac{S - x}{1 - x}, \tag{3}$$

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

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:

$$\delta = S(\tau') + S(\tau'') - S(\tau) \leq 2(1 - \sqrt{1 - S(\tau)}) - S(\tau) \tag{4}$$

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 of a multi-processor scheduling algorithm (A) is defined to be the maximum utilization value ($UB_m(A)$), such that all tasksets with individual task-utilizations less than or equal to 1, and total utilization less than or equal to $m * UB_m(A)$ are guaranteed to be schedulable on m processors.

$$\forall\{\tau\}, \forall\tau_i \in \{\tau\},$$

$$U(\tau_i) \leq 1 \wedge \sum_{\forall\tau_i\in\{\tau\}} U(\tau_i) \leq m * UB_m(A) \Longrightarrow Schedulable(A, \{\tau\}) \quad (5)$$

The size bound of a multi-processor scheduling algorithm (A) is defined to be the maximum size value ($SB_m(A)$), such that all tasksets with individual task-sizes less than or equal to 1, and total size less than or equal to $m * SB_m(A)$ are guaranteed to be schedulable on m processors. That is,

$$\forall\{\tau\}, \forall\tau_i \in \{\tau\},$$

$$S(\tau_i) \leq 1 \wedge \sum_{\forall\tau_i\in\{\tau\}} S(\tau_i) \leq m * SB_m(A) \Longrightarrow Schedulable(A, \{\tau\}\}) \quad (6)$$

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

Let us consider any taskset 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 processing core, except the last). Therefore, in the worst case, it increases the total size of the taskset by $(m-1)*\delta$. The cumulative task size of the taskset after task-splitting is:

$$CS^{TS} = S_{tot} + (m-1) * \delta = \textstyle\sum_{i=1}^{n} S(\tau_i) + (m-1) * \delta.$$

For the entire taskset 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 $SB(DMS)$, therefore

$$\textstyle\sum_{i=1}^{n} S(\tau_i) + (m-1) * \delta \geq m * SB(DMS).$$

Invoking Lemma 1 and re-writing, we obtain

$$\textstyle\sum_{i=1}^{n} S(\tau_i) \geq m * UB(RMS) - (m-1) * \delta,$$

therefore the size-bound $SB_m(PDMS\_HPTS)$ is at least:

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

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

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

The utilization bound for RMS is 0.6931 ([1]), therefore

$$UB_m(PDMS\_HPTS) > 0.6931 - \delta + \tfrac{\delta}{m}.$$

As $m \rightarrow \infty$, 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 tasksets.

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, then the correspond size bound for Deadline Monotonic Scheduling, $SB(DMS|S)$ is given by*

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

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

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

Proof:

Consider a pessimistic taskset $\{\tau*\}$ 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 taskset. 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 $\tau*$ taskset), the bound is given by

$$SB(DMS|S, r) >= UB(RMS|S, r) = r((\frac{2}{1+S})^{\frac{1}{r}} - 1) + S. \quad (7)$$

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

$$SB(DMS|S) >= UB(RMS|S) = \log(\tfrac{2}{1+S}) + S.$$

■

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

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

Proof: Since task deadlines equal task 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 processor P_j is at least $SB(DMS)$. From Lemma 1 we have,

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

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.

Processor 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 processor satisfies

$$S_{tot} \geq SB(DMS|S(\tau')) = \log(\frac{2}{1+S(\tau')}) + S(\tau'). \quad (8)$$

Task-splitting increases the size of the taskset by δ, therefore subtracting (Equation 4) $(S(\tau') + S(\tau'') - S(\tau))$ from (Equation 8) gives the total size of the original taskset allocated per-processor (S_{orig}),

$$S_{orig} \geq \log(\frac{2}{1+S(\tau')}) + S(\tau) - S(\tau''). \quad (9)$$

Using (Equation 3) of Lemma 4, we obtain

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

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

$$SB_m(PDMS\_HPTS) = \log(\frac{2}{1+x}) + S - \frac{S-x}{1-x} \quad (10)$$

Observe that (10) is a non-increasing function of $S = S(\tau)$. The minimum lower bound of $SB_m(PDMS\_HPTS)$, therefore, occurs at the maximum value of $S(\tau)$. We are allocating individual processors to all tasks with size greater than or equal to $SB_m(PDMS\_HPTS)$, therefore the maximum value of $S(\tau)$ is $SB_m(PDMS\_HPTS)$. Therefore, using $SB_m(PDMS\_HPTS) = S$ in (Equation 10), we obtain

$$\log(\frac{2}{1+x}) - \frac{s-x}{1-x} = 0. \quad (11)$$

Rewriting S as a function of x gives

$$s = (1-x)\log(\frac{2}{1+x}) + x. \quad (12)$$

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

$$\log(\frac{2}{1+x}) - \frac{2x}{1+x} = 0. \quad (13)$$

Solving this equation gives[4]

$$x = S(\tau') = \frac{2 - LambertW(e^2)}{LambertW(e^2)} = 0.2837, \quad (14)$$

The second derivative of (12) is positive, therefore the value of x given above corresponds to the minimum value of S.

Corresponding value of $s = S(\tau) = SB_m(PDMS\_HPTS)$ is 60.03%. The initially given taskset is assumed to have deadlines equal to their period, the size bound for such tasksets 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 tasksets with deadline equal to the period is at least 60.03% (with the exception of the last processor, which can achieve up to 69.31%).

$$\begin{aligned} UB_m(PDMS\_HPTS) &= \tfrac{0.6003*(m-1)+0.6931}{m} \\ &= 0.6003 + \tfrac{0.0928}{m}. \end{aligned}$$

As $m \to \infty$, the utilization bound of PDMS_HPTS is 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 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.

In the following section, we will describe a specific instance of PDMS_HPTS that achieves an utilization bound of 65% on generic tasksets.

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. The $Enqueue$ and $Dequeue$ functions are modified as follows to reflect the decreasing size order of task allocation:

- $EnqueueDS(Unallocated\_Queue, \tau)$: Enqueues the task τ in decreasing size order to the unallocated task queue.

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

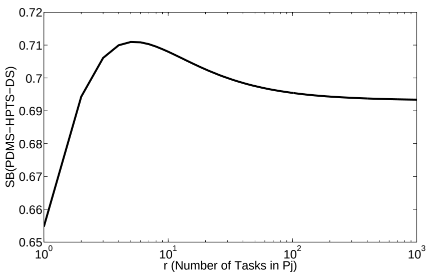

Fig. 2. Behavior of SB(PDMS_HPTS_DS) as a function of r

- $\tau \leftarrow DequeueDS(Unallocated\_Queue)$: Dequeues the task τ with the largest size ($\frac{C}{D}$) from the 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. Worst-Case Behavior

The following theorem shows that a higher worst case utilization bound hold for the PDMS_HPTS_DS algorithm.

Theorem 7: *The utilization bound of PDMS_HPTS_DS for tasksets with task deadlines equal to task periods is at least 65.47%.*

Proof: The size-bound of PDMS_HPTS_DS ($\beta = SB(PDMS\_HPTS\_TS)$) is determined by

$$\beta = (r - \frac{x}{1-x})((\frac{2}{1+x})^{\frac{1}{r}} - 1) + x. \tag{15}$$

The detailed derivation of this size-bound is a bit involved, the interested reader may refer to the Appendix for a more complete proof. ■

Using (15), the minimum values of $SB(PDMS\_HPTS\_DS)$ as a function of r are shown in the logarithmic plot Fig. 2.

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

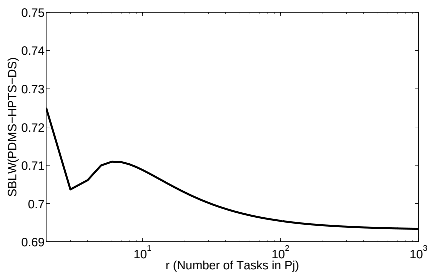

Fig. 3. Behavior of SBLW(PDMS_HPTS_DS,$\sqrt{2}-1$) as a function of r

The asymptotic behavior suggests that PDMS_HPTS_DS achieves a utilization bound of 0.693, when the taskset is composed of sufficiently lightweight tasks. Space limitations prevent a presentation of the detailed analysis, but it can easily derived that the size-bound of PDMS_HPTS_DS, for lightweight tasks with maximum utilization α is given by:

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

The behavior of this function, when $\alpha = 0.414$ is given in Fig. 3. It can be seen that the minimum value of $SBLW(PDMS\_HPTS\_DS, \sqrt{2} - 1)$ 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 tasksets. The PDMS_HPTS_DS algorithm is optimal for lightweight tasksets in that no fixed priority scheduling algorithm can guarantee a utilization bound greater than 0.6931 for multiprocessor scheduling (since the $m = 1$ scenario will become the worst-case due to the uniprocessor scheduling problem).

B. Average-Case Behavior

We have thus far analyzed the worst-case performance of the PDMS_HPTS_DS algorithm and obtained utilization bounds on its adversarial tasksets. 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 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 worst-case utilization bound of 65%.

We studied the average-case performance of the PDMS_HPTS_DS algorithm on randomly generated tasksets. The C values and T values for the tasksets were chosen at random, and the break-down utilization values were computed as described in [17]. The average-case utilization as a function of the number of processing cores is given in Fig.4. It can be seen that there is a slight decrease in the average utilization with the increasing number of processors, but the value seems to converge at around 88%. The probability density function of the taskset breakdown utilizations is given in Fig.5. As we can see, with an increasing number of processors the variance in the breakdown utilization decreases, indicating that more and more tasksets reach the average-case utilization.

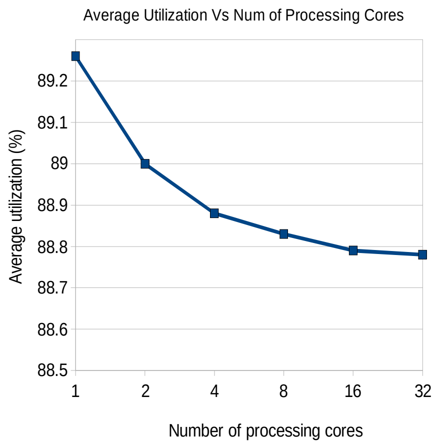

Fig. 4. Average-Utilization of PDMS_HPTS_DS with increasing no. of cores

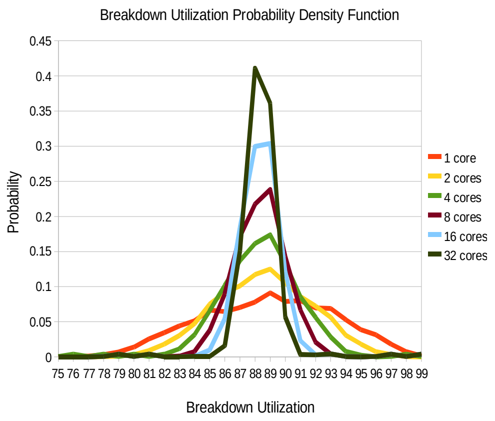

Fig. 5. PDF of Breakdown Utilizations under PDMS_HPTS_DS

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 Intel Core 2 Duo processor has 2 cores with 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 32KB) and stride-lengths (1 to 64bytes). The performance of these workloads is shown in Figure 6.

The overall overhead was acceptably low (less than 65 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 2GHz, to avoid side-effects due to frequency scaling.

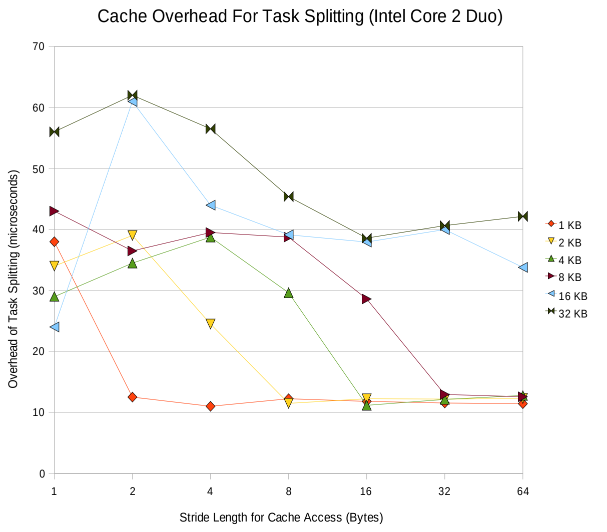

Fig. 6. Cache overhead due to Task-Splitting on the Intel Core 2 Duo

The results show that the overheads are generally lower at smaller working set sizes, as expected. The variation with respect to the cache access stride-lengths, is quite interesting. At lower stride-lengths, the performance of a hot-cache (without task-splitting) is much better than that of a cold-cache (with task-splitting) and hence the cache overhead is quite high. When the stride-length is really large (like 64 bytes), fetching a cache block is of no real use, as the loaded data will not be accessed. The performance under a cold-cache is affected adversely by this effect, since the first access to all data elements will result in a cache miss at high stride lengths. The cache overhead, therefore, slightly increases at very high stride lengths.

Optimal stride-lengths mostly appear to be mid-range values and are dependent on the working set-size. This analysis was done with read-write access patterns; however, similar behavior was also observed with read-only access patterns.

The above analysis shows that applications that access the cache in a spatially sparse manner are the least affected by task migration. As expected, applications with smaller working sets are also less affected by task splitting. Although not shown here, applications with very low temporal locality will also have negligible 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. 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 the codec being used and a wmv3 file was being played. The core computational loop in the video player is comprised solely of calls to $decode\_frame$ in addition to minor accounting updates. Without task splitting, the average time taken by a single call to $decode\_frame$ was 7.6ms. After performing task splitting, the average time taken for a $decode\_frame$ call was increased

Application	**Time Taken (ms) Without Task Splitting**	**Time Taken (ms) With Task Splitting**
MPlayer	7.6	7.8
GL Gears	4.5	4.5

TABLE I
TASK SPLITTING OVERHEAD ON REAL-WORLD APPLICATIONS

by approximately 0.2ms.

The reason for the overhead of about 200 microseconds in the case of *MPlayer* is probably the fact that it incurs heavy overheads, due to both the instruction cache and the data cache. The video was being played at approximately 25 frames/second (a period of 40ms), and this cache overhead translates to 0.5% in terms of the *decoder* task utilization.

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 processor 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%. A specific instance of this class of algorithms PDMS_HPTS_DS, has been shown to achieve a utilization bound of 65% on generic tasksets. PDMS_HPTS_DS is also shown to achieve a higher utilization bound of 69% on lightweight tasksets, where the individual task utilizations are restricted to a maximum of 41.4%. The average case analysis of PDMS_HPTS_DS using randomly generated tasksets shows that it achieves a very high utilization of 88% on the average. Extensions to the basic PDMS_HPTS_DS scheme that exploit relationships between the task periods may achieve a much higher utilization bound on the average. Although we currently do not provide details of these modifications, it is an important future work to evaluate this possibility.

The cache overhead due to task-splitting has been evaluated on the Intel Core 2 Duo, using both synthetic cache-workloads and real-world applications. The overhead of task migration is seen to be very low on this platform.

REFERENCES

[1] C. L. Liu, and J. W. Layland, *"Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment"*, Journal of the ACM (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"*, In processdings of the 10th Euromicro Workshop on Real-Time Systems, 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 Parallel and Distributed Processing Symposium, 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 8th IEEE Real-Time and Embedded Technology and Applications Symposium (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, and J. Jonsson, *"The utilization bounds of partitioned and pfair static-priority scheduling on multiprocessors are 50%"*, In Proceedings of the 15th Euromicro conference on Real-Time Systems (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 the 25th IEEE Real-Time Systems Symposium (RTSS), 2004.

[12] B. Andersson, and E. Tovar, *"Multiprocessor Scheduling with Few Preemptions"*, In Proceedings of the 12th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA), pp. 322-334, 2006.

[13] S. Kato, and N. Yamasaki, *"Real-Time Scheduling with Task Splitting on Multiprocessors"*, In Proceedings of the 13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (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 the 12th IEEE Real-Time and Embedded Technology and Applications Symposium (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 the Real-Time Systems Symposium, pp. 166-171, 1989.

[18] Dionisio de Niz, and Raj Rajkumar, *Partitioning Bin-Packing Algorithms for Distributed Real-Time Systems*, International Journal of Embedded Systems. Special Issue on: Design and Verification of Real-Time Embedded Software", vol. 2, pp. 196-208, 2006.

APPENDIX

Utilization bound of PDMS_HPTS_DS is 65.47%

Proof: As in the proof of Theorem 6, tasks with individual sizes greater than or equal to 65.47% are allocated to their own processors as they have a size greater than or equal to the size bound. These tasks do not affect the per-processor allocated sizes, and can be ignored. Now, consider the two cases of `HPTaskSplit`.

Case 1. `HPTaskSplit` decides to not perform `MaximalSplit`.

Lemmas 1 and 2 show that the total size S_{tot} allocated to processor P_j is at least $SB(DMS)$, that is

$S_{tot} \geq SB(DMS) \geq UB(RMS) = 0.6931.$

These processing cores are allocated cumulative sizes greater than or equal to 65.47%; therefore, they can be ignored in our analysis for the worst-case per-processor utilization.

Case 2: `HPTaskSplit` decides to perform `MaximalSplit` on processor P_j.

In this case, the processor overflows on allocating some task, say τ_f; therefore, the highest priority task τ in the processor P_j is split. Let the split pieces be denoted by τ' and τ''. Let the total number of tasks allocated to P_j be $(r+1)$ including τ'. Task τ' is obtained through `MaximalSplit`; therefore, by definition the processor is maximally utilized w.r.t. τ', i.e. any increase in the computation time of τ' renders the processor to be unschedulable.

Let the cumulative sum of the task-sizes excluding τ and τ_f be CS_r, so

$$CS_r + S(\tau_f) + S(\tau') > CS_r + S(\tau), \qquad (16)$$

where the left-hand size of the above expression gives the cumulative sum of task sizes after allocating τ_f, and the right-hand side gives a lower-bound on the cumulative task-sizes before allocation τ_f. If the inequality (16) does not hold good, `HPTaskSplit` would have decided to not invoke `MaximalSplit`.

Simplifying and re-writing (16) gives

$$S(\tau) - S(\tau\prime) < S(\tau_f). \qquad (17)$$

Let the size of the task τ be denoted by $S = S(\tau)$ and the size of sub-task τ' be denoted by $x = S' = S(\tau')$.

The task has deadlines D less than or equal to the period T, and the processor is maximally utilized under deadline-monotonic scheduling (after adding τ' with size x). Using Lemma 2 and Lemma 5, the total allocated size on the processor is greater than or equal to:

$$SB(DMS|x|r) = r((\frac{2}{1+x})^{\frac{1}{r}} - 1) + x. \qquad (18)$$

First, let us consider the case when $r = 1$. Re-writing (18) gives

$$SB(DMS|x|1) = \tfrac{1-x}{1+x} + x.$$

Accommodating the penalty of task-splitting from (4), we get the size bound of PDMS_HPTS_DS to be:

$$SB(PDMS\_HPTS\_DS) = \tfrac{1-x}{1+x} + S - \tfrac{S-x}{1-x}.$$

Let $\beta = SB(PDMS\_HPTS\_DS)$.

β is a non-increasing function in S ($\forall\ 0 \leq x \leq 1$), therefore we want to use the maximum value of S to find the minimum β. The maximum possible value of S in our case is 0.6547, since tasks greater than this value are allocated their own processing cores and task-splitting does not increase the size of tasks (from Lemma 3). We therefore assert that,

$$\beta = \tfrac{1-x}{1+x} + 0.6547 - \tfrac{0.6547-x}{1-x}.$$

Minimizing this function w.r.t. x, we find that the minimum occurs at $x = 0.4129$.

The corresponding value of $SB(PDMS\_HPTS\_DS)$ is given by

$$\beta = 0.6584 \geq 0.6547.$$

When $r = 1$, we see that $SB(PDMS\_HPTS\_DS)$ is greater than or equal to 0.6547, which is the proposed size-bound for PDMS_HPTS_DS. We therefore need to prove that the bound also holds good for $r \geq 2$.

The PDMS_HPTS_DS algorithm always allocates tasks in the decreasing order of size; therefore, the size of the last-task considered for allocation, τ_f, must be less than or equal to the size of all the other tasks allocated to P_j (except possibly τ' which was split from task τ). The size bound of the processor P_j can now be refined:

1) When $S(\tau_f) < ((\frac{2}{1+x})^{\frac{1}{r}} - 1)$, apply (18) to obtain

$$SB^{case1}(DMS|x|r) = r((\frac{2}{1+x})^{\frac{1}{r}} - 1) + x. \qquad (19)$$

Adjusting the overhead of task-splitting from (4), we obtain the size bound of PDMS_HPTS_DS as:

$$\beta = r((\tfrac{2}{1+x})^{\frac{1}{r}} - 1) + S - S(\tau'').$$

Using (3) and re-writing gives,

$$\beta = r((\tfrac{2}{1+x})^{\frac{1}{r}} - 1) + (1-S)\tfrac{x}{1-x}.$$

We see that β is a non-increasing function of S ($\forall\ 0 \leq x \leq 1$), therefore we want to maximize S in order to find the minimum value of β. Using (17):

$$S < x + S(\tau_f).$$

Currently we are considering the case
$S(\tau_f) < ((\frac{2}{1+x})^{\frac{1}{r}} - 1)$, so we can minimize β as:

$$\beta = r((\tfrac{2}{1+x})^{\frac{1}{r}} - 1) + (1 - x - ((\tfrac{2}{1+x})^{\frac{1}{r}} - 1))\tfrac{x}{1-x}.$$

Re-writing gives,

$$\beta = (r - \tfrac{x}{1-x})((\tfrac{2}{1+x})^{\frac{1}{r}} - 1) + x.$$

2) When $S(\tau_f) \geq (\frac{2}{1+x})^{\frac{1}{r}} - 1)$, all the r tasks (other than τ') must have sizes greater than or equal to the size of the last, and the sub-task τ' has a size x, therefore:

$$SB^{case2}(DMS|x|r) = r(S(\tau_f)) + x. \qquad (20)$$

Adjusting the overhead of task-splitting from (4), we get the size bound of PDMS_HPTS_DS to be:

$$\beta = r(S(\tau_f)) + S - S(\tau'')$$

Using (3) and re-writing gives,

$$\beta = r(S(\tau_f)) + (1-S)\tfrac{x}{1-x}.$$

We see that β is a non-increasing function of S ($0 \leq x \leq 1$); therefore, we want to maximize S in order to find the minimum value of β. Using (17) we get

$$S < x + S(\tau_f).$$

Currently we are considering the case
$S(\tau_f) \geq ((\frac{2}{1+x})^{\frac{1}{r}} - 1)$, or equivalently
$S(\tau_f) = ((\frac{2}{1+x})^{\frac{1}{r}} - 1 + \delta)($ for some $\delta \geq 0)$;
Therefore, we can minimize β as:

$$\beta = r((\tfrac{2}{1+x})^{\frac{1}{r}} - 1 + \delta) + (1 - x - ((\tfrac{2}{1+x})^{\frac{1}{r}} - 1 + \delta))\tfrac{x}{1-x}.$$

Re-writing this, we get

$$\beta = r((\tfrac{2}{1+x})^{\frac{1}{r}} - 1) + (1 - x - ((\tfrac{2}{1+x})^{\frac{1}{r}} - 1))\tfrac{x}{1-x} \\ + \delta(r - \tfrac{x}{1-x})$$

This is an increasing function of δ for $r \geq 2$ (since $x \leq 0.6547$); therefore, the minimum value of β occurs at $\delta = 0$, so we obtain

$$\beta = r((\tfrac{2}{1+x})^{\frac{1}{r}} - 1) + (1 - x - ((\tfrac{2}{1+x})^{\frac{1}{r}} - 1))\tfrac{x}{1-x}.$$

Rewriting gives,

$$\beta = (r - \tfrac{x}{1-x})((\tfrac{2}{1+x})^{\frac{1}{r}} - 1) + x,$$

which is the same as the previous case. Therefore, the size-bound of PDMS_HPTS_DS ($\beta = SB(PDMS\_HPTS\_TS)$) is determined by

$$\beta = (r - \frac{x}{1-x})((\frac{2}{1+x})^{\frac{1}{r}} - 1) + x. \qquad (21)$$

■