Partitioned EDF Scheduling for Multiprocessors using a C=D Scheme

Alan Burns, Robert Davis, P. Wang, Fengxiang Zhang

Partitioned EDF Scheduling for Multiprocessors using a C=D Scheme

A. Burns, R.I. Davis, P. Wang and F. Zhang
Department of Computer Science,
University of York, UK.

***Abstract*—An EDF-based task-splitting scheme for scheduling multiprocessor systems is introduced in this paper. For m processors at most $m-1$ tasks are split. The first part of a split task is constrained to have a deadline equal to its computation time. The second part of the task then has the maximum time available to complete its execution on a different processor. The advantage of this scheme is that no special run-time mechanisms are required and the overheads are kept to a minimum. Analysis is developed that allows the parameters of the split tasks to be derived. This analysis is integrated into the QPA algorithm for testing the schedulability of any task set executing on a single processor under EDF. Evaluation of the C=D scheme is provided via a comparison with a fully partitioned scheme and the theoretical maximum processor utilisation.**

I. Introduction

Multiprocessor and multi-core platforms are currently the focus of considerable research effort. There are a number of open theoretical questions and many practical problems, all of which need to be addressed if effective and efficient real-time systems are to be hosted on these emerging platforms. One of key issues, that does not exist with single processor systems, is the allocation of application tasks to the available processors. Many different schemes have been advocated and evaluated, from the fully partitioned to the totally global. It is unlikely that a single scheme will meet the needs of all applications [13].

In this paper we consider a *task-splitting* approach in which most tasks are statically partitioned, but a few (at most one per processor) are allowed to migrate from one processor to another during execution. For each execution of one of these tasks it is initially (statically) allocated to one processor, after a period of execution the task moves to a (predefined) second processor where it completes its execution. When it is next released it returns to the first processor. The motivation for this task-splitting scheme is that it can benefit from most of the advantages of the fully partitioned scheme, but can gain enhanced performance from its minimally dynamic behaviour. A number of researchers have considered task-splitting (they are reviewed below). In this paper we use an approach that utilises the effectiveness of EDF scheduling for single processors whilst not requiring any particular run-time facilities from the multi-core platform. It is low on overheads and hence it can form the basis for a practical approach to scheduling real-time applications with near optimal use of the processing resources.

Fully partitioned systems have the advantage that each processor is scheduled separately and hence standard single processor theory is applicable. The disadvantage comes from the necessary 'bin packing' problem that must efficiently allocate tasks to processors without overloading any of the processors – an overload would lead to a deadline being missed at runtime. Globally scheduled systems do not suffer from this bin packing problem, but they do have other problems to consider. At the theoretical level it seems that no simple dispatching scheme with low overheads can optimally schedule all task sets (in particular those that include sporadic tasks and arbitrary deadlines). At the practical level there are cache coherence problems that add significantly to the overheads of task migration. An approach that involves a minimum amount of migration but allows a small number of tasks to be 'split', so that the processor bins are better filled, clearly has many attractions.

A. System Model and EDF Analysis

We use a standard system model in this paper, incorporating the preemptive scheduling of periodic and sporadic task systems. A real-time system, $\mathcal{A}$, is assumed to consist of n tasks (τ_1 .. τ_n) each of which gives rise to a series of jobs. Each task τ_i is characterized by the following profile (C_i, D_i, T_i):

- A *period* or *minimum inter-arrival time* T_i; for *periodic* tasks, this defines the exact temporal separation between successive job arrivals, while for *sporadic* tasks this defines the minimum temporal separation between successive job arrivals.
- A *worst-case execution time* C_i, representing the maximum amount of time for which each job generated by τ_i may need to execute. The worst-case utilization (U_i) of τ_i is C_i/T_i. All tasks must have $U_i \leq 1$. The total system utilisation, U, is simply the sum of all these individual task utilisations.
- A *relative deadline* parameter D_i, with the interpretation that each job of τ_i must complete its execution within D_i time units of its arrival. The *absolute deadline* of a job from τ_i that arrives at time t is $t + D_i$. In general, deadlines are *arbitrary*: they can be less than, greater than or equal to the period values. We use the term *implicit deadline* for tasks with $D_i = T_i$ and *constrained deadline* for tasks with $D_i \leq T_i$.

Once released, a job does not suspend itself. We also assume in the analysis, for ease of presentation, that tasks are independent of each other and hence there is no blocking factor to be incorporated into the scheduling analysis. General system overheads are ignored in this treatment. Their inclusion would not impact on the structure of the results presented, but would complicate the presentation of these results. In practice, these overheads must of course not be ignored [10]. We do however consider the extra overheads introduced by the task-splitting scheme.

There are no restrictions on the relative release times of tasks (other than the minimum separation of jobs from the same task). Hence we assume all tasks start at the same instant in time – such a time-instant is called a *critical instant* for the task system [22]. In this analysis we assume tasks do not experience release jitter. We are concerned with analysis that is necessary, sufficient and sustainable [5].

The hardware platform consists of m identical processors. On each processor the allocated tasks (including any that might be split) are scheduled by EDF. Therefore with implicit deadlines there is a potential utilisation bound of m.

Exact analysis for EDF scheduled tasks on a single processor usually involves the use of Processor-Demand Analysis (PDA) [7], [6]. This test takes the following form (the system start-up is assumed to be at time 0):

$$\forall t > 0 : \quad h(t) \ \leq \ t \tag{1}$$

where $h(t)$ is the total load/demand on the system (all jobs that have started since time 0 and which have a deadline no greater than t). A simple formulae for $h(t)$ is therefore:

$$h(t) \ = \ \sum_{j=1}^{n} \left\lfloor \frac{t + T_j - D_j}{T_j} \right\rfloor_0 C_j \tag{2}$$

n here is the number of tasks on this single processor, and $\lfloor\rfloor_0$ is the usual floor function capped below by 0 (ie. minimum value it can furnish is 0).

The need to check all values of t is reduced by noting that only values of t that correspond to job deadlines have to be assessed. Also there is a bound on t. An unschedulable system is forced to fail inequality (1) before the bound L. A number of values for L have been proposed in the literature. A large value is obtained from the LCM of the task periods. A tighter value comes from the *synchronous busy period* [24], [25]. This is usually denoted by L_B and is calculated by forming a recurrence relationship:

$$s^{q+1} \ = \ \sum_{i=1}^{n} \left\lceil \frac{s^q}{T_i} \right\rceil C_i \tag{3}$$

The recurrence stops when $s^{q+1} = s^q$, and then $L_B = s^q$. Note that the recurrence cycle is guaranteed to terminate if $U \leq 1$ for an appropriate start value such as $s^0 \ = \ \sum_{i=1}^{n} C_i$.

If U is strictly less than 1 then a simpler formulae for L is possible [15]:

$$L_A \ = \ Max \left\{ D_1, ..., D_n \frac{\sum_{j=1}^{n} (T_j - D_j) U_j}{1 - U} \right\} \tag{4}$$

With all available estimates for L there may well be a very large number of deadline values that need to be checked using inequality (1) and equation (2). This level of computation has been a serious disincentive to the adoption of EDF scheduling in practice. Fortunately, a new much less intensive test has recently been formulated [27], [28]. This test, known as QPA (Quick Processor-demand Analysis), starts from time L and iterates backwards towards time 0 checking a small subset of time points. These points are proved [27], [28] to be adequate to provide a necessary and sufficient test.

A version of the QPA algorithm optimised for efficient implementation is encoded in the following pseudo code in which `D_min` is the smallest relative deadline in the system, and `Gap` is the common divisor of the computation times and deadlines. The value of `Gap` is such that no two significant points in time (eg interval between two adjacent deadlines) is less than `Gap`. For example, if all task parameters are given as arbitrary integers then `Gap` will have the value 1.

```
t := L - Gap
while h(t) <= t and t >= D_min loop
  t := h(t) - Gap
end loop
if t < D_min then
  -- task set is schedulable
else
  -- task set is not schedulable
end if;
```

In each iteration of the loop a new value of `t` is computed. If this new value is less than the computed load at that point, the task set is unschedulable. Otherwise the value of `t` is reduced during each iteration and eventually it must become smaller than the first deadline in the system and hence the system is schedulable.

B. Previous Related Research

A number of papers have been published on partitioning and, specifically, task splitting. Andersson and Tovar introduced in 2006 [2] an approach to scheduling periodic task sets with implicit deadlines, based on partitioned scheduling, but splitting some tasks into two components that execute at different times on different processors. They derived a utilisation bound depending on a parameter k, used to control the division of tasks into groups of heavy and light tasks. A heavy task has a high utilisation. These tasks cause particular difficulties for partitioned systems. Indeed they lead to a utilisation bound of just 50% as only m tasks each with a utilisation of 50+δ% can be accommodated on m processors (for arbitrary small δ).

Andersson et al. later extended this approach to task sets with arbitrary deadlines [1]. They showed that first-fit and next-fit were not good allocation strategies when task splitting is employed. Instead, they ordered tasks by decreasing relative deadline and tried to fit all tasks on the first processor before then choosing the remaining task with the shortest relative

deadline to be split. At run-time, the split tasks are scheduled at the start and end of fixed duration time slots. The disadvantage of this approach is that the capacity required for the split tasks is inflated if these slots are long, while the number of preemptions is increased if the time slots are short.

Bletsas and Andersson developed an alternative approach in 2009 based on the concept of notional processors[9]. With this method, tasks are first allocated to physical processors (heavy tasks first) until a task is encountered that cannot be assigned. Then the workload assigned to each processor is restricted to periodic reserves and the spare time slots between these reserves organised to form notional processors.

A distinct series of developments lead to the introduction of the Ehd2-SIP algorithm [17]. Ehd2-SIP is predominantly a partitioning algorithm, with each processor scheduled according to an algorithm based on EDF; however, Ehd2-SIP splits at most m-1 tasks into two portions to be executed on two separate processors. EhD2-SIP has a utilisation bound of 50%. Kato and Yamasaki presented a further semi-partitioning algorithm called EDDP [19], also based on EDF, that splits at most m-1 tasks across two processors. The two portions of each split task are prevented from executing simultaneously by EDDP, which instead defers execution of the portion of the task on the lower numbered processor, while the portion on the higher numbered processor executes. During the partitioning phase, EDDP places each heavy task with utilisation greater than 65% on its own processor. The light tasks are then allocated to the remaining processors, with at most m-1 tasks split into two portions. They showed that EDDP has a utilisation bound of 65% for periodic task sets with implicit deadlines, and performs well in terms of the typical number of context switches required which is less than that of EDF due to the placement strategy for heavy tasks. Subsequently, Kato and Yamasaki [18] also extended this approach to fixed task priority scheduling, presenting an algorithm with a utilisation bound of 50%.

Kato et al. then developed a semi-partitioning algorithm called DM-PM (Deadline-Monotonic with Priority Migration); applicable to sporadic task sets, and using fixed priority scheduling [20]. DM-PM strictly dominates fully partitioned fixed task priority approaches, as tasks are only permitted to migrate if they wont fit on any single processor. Tasks chosen for migration are assigned the highest priority, with portions of their execution time assigned to processors, effectively filling up the available capacity of each processor in turn. At run-time, the execution of a migrating task is staggered across a number of processors, with execution beginning on the next processor once the portion assigned to the previous processor completes. Thus no job of a migrating task returns to a processor it has previously executed on. They showed that DM-PM has a utilisation bound of 50% for task sets with implicit deadlines. Subsequently, they extended the same basic approach to EDF scheduling; forming the EDF-WM algorithm (EDF with Window constrained Migration).

For fixed priority scheduling Lakshmanan et al. [21] also developed a semi-partitioning method for sporadic task sets with implicit or constrained deadlines. This method called PDMS-HPTS splits only a single task on each processor; the task with the highest priority. Note that a split task may be chosen again for splitting if it has the highest priority on another processor. PDMS-HPTS takes advantage of the fact that under fixed priority preemptive scheduling, the response time of the highest priority task on a processor is the same as its worst-case execution time; leaving the maximum amount of the original task deadline for the part of the task split on to another processor to execute. They showed that for any task allocation, PDMS-HPTS has a utilisation bound of at least 60% for task sets with implicit deadlines; however, if tasks are allocated to processors in order of decreasing density (PDMS-HPTS-DS), then this bound increases to 65%. Further, PDMS-HPTS-DS has a utilisation bound of 69.3% if the maximum utilisation of any individual task is no greater than 0.41. Notably, this is the same as the Liu and Layland bound for single processor systems without the restriction on individual task utilisation. Subsequently, Guan et al. [14] developed the SPA2 partitioning / task-splitting algorithm which has the Liu and Layland utilisation bound, assuming only that each task has a maximum utilisation of 1.

For a broader review of research appertaining to multiprocessor scheduling the reader is referred to the survey paper by Davis and Burns [13] from which the above discussion is distilled.

C. Contribution and Organisation

In this paper we motivate, describe and evaluate the average behaviour of an EDF-based C=D scheme in which a maximum of m-1 tasks are split (for m processors). What is distinctive about the developed C=D scheme is that it is straightforward to implement (no unusual RTOS functions required, only a standard timer; CPU time monitoring is not necessary) and has low overheads that can easily be accommodated into the analysis. The scheme utilises an off-line analysis-based procedure that exploits some key properties of EDF scheduling (for single processors). It has some resemblance to the fixed priority scheme of Lakshmanan et al. [21] described above, in that the split task occupies its first processor for the minimum elapsed time but maximum execution time; it effectively executes non-preemptively on its first processor

We leave for future work the development of a utilisation bound. We also leave open the question as to the best 'bin-packing' algorithm to use in conjunction with the scheme. We make no attempt to deal with 'heavy' tasks differently from 'light' ones. Again this might lead to further improvements. Rather our motivation here is to illustrate the usefulness of a very basic and straightforward approach. For this reason the current paper does not include a detailed comparison with other task splitting schemes.

The remainder of the paper contains two main sections. The first describes the C=D scheme, the other provides an evaluation. Conclusions are presented in Section IV.

II. The C=D Partitioning Scheme

In this section we develop the partitioning scheme by first noting some useful properties of EDF scheduling of single processors. These properties can be exploited by employing the QPA scheme to explore the characteristics of any particular task set's parameters.

A. Motivational Characteristic of EDF Systems

Consider a task set with $D = T$ for all tasks and a total utilisation of 1. For example a simple system of 5 identical tasks with $C = 2$ and $T = D = 10$. This task set is clearly deemed schedulable on a single processor by use of the standard utilisation test (as $U = 1$); it is not necessary to employ QPA. However, using an extension of QPA [29], [30] for sensitivity analysis it is possible to ask the question: 'For each task (separately), what is the minimum value of D that will still deliver a schedulable system?' As now $D < T$ for one task, a utilisation based test is not applicable[1]; hence QPA is employed. For this task set, each task can have its deadline reduced to the minimum value of 2 (ie. $D = C$) and the system remains schedulable. The intuition here is that a single task (τ_i) with $D_i = C_i$ can be accommodated if there is sufficient slack within the other tasks (for example, if no other task (τ_j) has $T_j - C_j < C_i$). The optimal behaviour of EDF can accommodate one task with an extreme requirement of $D = C$. These observations are also supported by Balbastre et al [4].

A less constrained example is given in Table I. Here again the total utilisation is 1 and all tasks have $D = T$. There are seven tasks and a range of periods from 10 to 48. Sensitivity analysis again shows that if each task is individually assessed to see what its minimum deadline could be then all but one of the tasks can have its deadline reduced to its computation time without jeopardising schedulability. The one that cannot, has the longest period (and deadline). It can actually get 5 ticks in 5, but the 6th tick takes until time 26.

Task	T	D	C	Min D
τ_1	10	10	1	1
τ_2	12	12	3	3
τ_3	15	15	3	3
τ_4	16	16	2	2
τ_5	20	20	3	3
τ_6	40	40	2	2
τ_7	48	48	6	26

TABLE I
EXAMPLE TASK SET

The behaviour shown by this example is typical. To consider *how typical* a number of random task sets were generated and evaluated. The UUniFast algorithm [8] was employed to generate 10,000 task sets per experiment, each with implicit deadlines, ie. $D = T$. If the utilisation of each task set is fixed at 1, 60.23% of these task sets had at least one task that could have its deadline reduced to its execution time (the number of tasks in these experiments was between 20 and 100, and the ratio of longest to shortest period was 2). For utilisation of .95 the percentage rose to 99.10%, and for utilisation of .9 the result was over 99.999%. Figure 1 shows the percentage of task sets with at least one $C = D$ task for various utilisation levels and 10 tasks.

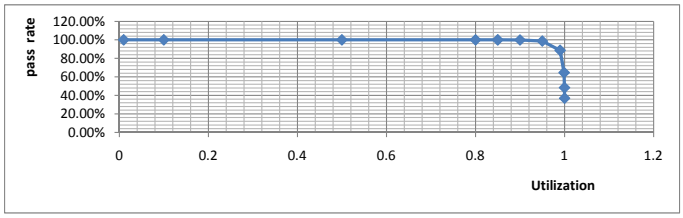

Fig. 1. Task sets with one $C = D$ task

These results imply that, in a multiprocessor system, a high level of utilisation can be achieved by allowing C to equal D for the first part of any split task. To try and fit more than this value on to the first processor would require a much longer deadline – leaving a much shorter interval for the second part of the task, which in turn would constrain the scheduling of the processor that is assigned the second part of the task.

B. The C=D Scheme

The task splitting scheme introduced in this paper is defined as follows. First the tasks are ordered by some 'bin packing' scheme based on, for example, utility or density[2].

- Each processor (p) is 'filled' with tasks until no further task can be added without leading to unschedulability of the processor.
- The next task, τ_s with profile (C_s, D_s, T_s) is then split so that the first part is retained on processor p and the task load on that processor is schedulable.
- The first part of the split task (τ_s^1) has the constraint that its deadline is reduced to be equal to the maximum computation time that can be accommodated. Its profile is therefore (C_s^1, $D_s^1 = C_s^1$, T_s).
- The second part of the split task (τ_s^2) has the derived profile ($C_s^2 = C_s - C_s^1$, $D_s^2 = D_s - D_s^1$, T_s). It is allocated to processor $p + 1$.
- The scheme continues by allocating further tasks to processor $p + 1$ until that processor can no longer accommodate a further complete task. Another task is then chosen to be split between processor $p + 1$ and $p + 2$.

An example split task would be one that originally had the profile (5,30,30) – ie. 5 units of execution every 30 with a deadline of 30. After splitting, its first part may be restricted

[1]Strictly, a sufficient density test could be used but this would lead to a result of unschedulable.

[2]Note the actual algorithm used for allocating tasks to processors is not of paramount importance to the scheme, in the following we assume a *first-fit* process, but any one of the possible *best-fit* approaches could also be applied.

to (2,2,30) and hence its second part would be (3,28,30). The second part being released 2 units of time after the first part.

The implementation of this C=D scheme is straightforward and requires no special features of the RTOS (other than support for EDF scheduling, task affinities and the identification of deadlines). The scheduling analysis, and the means by which the C_s^1 values are found, is considered in the next section. In terms of implementation, the following aspects are pertinent.

- The split task (τ_s), when released for execution at time t on processor p, will execute on p until time $t + D_s^1$. Computation time need not be measured by the RTOS, the task switch occurs after a period of 'real' time – only a standard timer is needed. In effect the first part of the split task will execute non-preemptively as it has $D = C$ on its release.
- At time $t + D_s^1$ the task's affinity is changed from p to $p + 1$. The task will now execute (preemptively) on processor $p + 1$ with a deadline of $t + D_s^1 + D_s^2$ which is equivalent to $t + D_s$, the original task deadline.

It follows directly from this implementation scheme that the two parts of the task can never execute concurrently. No further run-time action is needed to ensure that this necessary constraint is satisfied. It also follows that there are at most $m - 1$ split tasks.

To briefly illustrate how easy it is to implement this scheme, the following Ada code (in the forthcoming Ada 2012 version of the language) represents the 'handler' that is executed at the time for the task move.

```
procedure Handler(TM :in out Timer) is
  New_Deadline : Deadline;
begin
  New_Deadline := Get_Deadline(Client);
  -- obtains the deadline of the
  -- Client task to be moved
  Set_Deadline(New_Deadline + Extra, Client);
  -- extends deadline by fixed amount
  Set_CPU(Q+1,Client);
  -- moves task to processor Q+1
end Handler;
```

The periodic client task, which executed first on processor Q, has a simple behaviour:

```
Set_CPU(Q,Client);
loop
  Set_Handler(Next+First_Deadline,Handler'Access);

  -- code of application

  Next := Next + Period;
  -- compute the next release time
  Set_Deadline(Next+First_Deadline);
  Set_CPU(Q,Client);
  delay Until Next;
end loop;
```

The details of this example are given elsewhere [11] – where the task switching algorithm is used to illustrate the usefulness of some new language features for Ada.

Before giving the required analysis, and reporting on an evaluation of this C=D scheme, two useful properties of the scheme are worth emphasising. Again consider task τ_s to be split between processors p and $p+1$. If it were not to be split (ie. a fully partitioned scheme was being employed) then all of τ_s would need to be allocated to processor $p + 1$.

Property 1: With the C=D scheme, processor p is making a positive contribution to scheduling the task set.

This is clearly true as p has all of the 'un-split' load plus some further work. Its overall utilisation is going to be closer to 1.

Property 2: With the C=D scheme, processor $p + 1$ is not making a negative contribution to scheduling the task set.

This follows from the sustainability [5] of single processor EDF scheduling. In the fully partitioned scheme processor $p + 1$ must accommodate all of τ_s; it guarantees C_s within D_s. Now if C_s is guaranteed within D_s then $C_s - X$ must have been accommodated within $D_s - X$ for all positive X ($X < C_s$). For example, if 5 ticks are to occur within an interval of 20 ticks then 4 ticks must be completed within 19. It follows that accommodating τ_s^2 cannot be harder than accommodating τ_s in a schedulability sense. However as the required computation time is reduced then the utilisation load on processor $p + 1$ is also reduced, thereby increasing the capacity of the processor to accommodate extra work.

This latter property is an important one as it implies that *any* partitioning scheme can be used with this C=D scheme. Any 'bin packing' algorithm that gives an effective mapping of tasks to processors can form the starting point for the scheme. Of course if, for some task set, a partitioning scheme delivers 100% utilisation then task splitting cannot improve the allocation. But if there is spare capacity then the splitting of some task that was previously allocated to just one processor may improve the mapping, but cannot make it worse.

More formally, it follows that the C=D task splitting scheme *dominates* any partitioning scheme. Assume a system has been developed using a specific partitioning approach. All n tasks are therefore allocated to the m processors. Order the processors (from 1 to m). Start with the first processor and attempt to bring to this processor a part of any task from the second processor. If no task on the second processor can be split then the first processor is unchanged. But if some initial part of any task can be 'brought forward' on to the first processor then the utilisation of the first processor is increased, the schedulability of the second processor is unaffected, but its total utilisation is reduced. Repeat this process for the second and subsequent processors (up to processor $m - 1$). No anomalies are possible, the system remains schedulable but the number of processors needed may be reduced or the utilisation of the final processor may be reduced. Either way the task splitting scheme performs as well or better than any partitioning scheme.

When overheads are taken into account (for example, cost of migration and any penalty for disturbing the cache) then if the run-time cost of splitting the task is δ there is an overall gain if $\delta < C_s^1$. This follows from the observation that the resulting execution time of the second part of the task will be $C_s^2 = C_s - C_s^1 + \delta$ which will be less than the original

computation time when the additional overhead is so bounded.

C. C=D Sensitivity Analysis

To employ the C=D scheme, an effective means of computing the value of C_s^1 must be provided. This is developed in this section, first a basic approach is given, then means of making the scheme more efficient are considered.

Assuming processor p is deemed unschedulable when task τ_s is added. The following steps are undertaken.

1) Choose (initially) C_s^1 ($< C_s$) so that the utilisation of processor p is 1.
2) Set $D_s^1 \leftarrow C_s^1$.
3) Compute L, the maximum 'test' interval (using L_B if $U = 1$ – see Section I-A).
4) Start from L working backward with the QPA scheme.
5) If there is a failure, recompute a reduced C_s^1 (and hence D_s^1) – see below.
6) If the newly computed value of C_s^1 is 0 then exit – no portion of the task can be accommodated on processor p.
7) Continue working backward towards time D_{min} (the shortest deadline of any task on that processor) then repeat from step 3 if there has been a failure, it no failure then the current value of C_s^1 is the optimum one.

Assuming there is a failure at time t, ie. $h(t) > t$. The value of C_s^1 must be reduced. The recomputed value of C_s^1 follows directly from the demand function at the point of the failure. In the interval from 0 to t the amount of time that all the other tasks require, $Oth(t)$, is given by:

$$Oth(t) \;=\; \sum_{\substack{\tau_j \in p \\ \tau_j \neq \tau_s}} \lfloor \frac{t + T_j - D_j}{T_j} \rfloor C_j$$

where the summation is over all the other tasks assigned to processor p (not including τ_s).

In the remaining time (t minus this value) there will be

$$\lfloor \frac{t + T_s - D_s^1}{T_s} \rfloor$$

releases of τ_s.

This implies that each release must have a maximum computation time given by:

$$C_s^1 = (t - Oth(t)) \;/\; \lfloor \frac{t + T_s - D_s^1}{T_s} \rfloor \qquad (5)$$

The D_s^1 term must then be replaced by the C_s^1 term:

$$C_s^1 \leftarrow (t - Oth(t)) \;/\; \lfloor \frac{t + T_i - C_s^1}{T_i} \rfloor \qquad (6)$$

giving a formulation in which the unknown parameter, C_s^1, is on both sides of the equation. To compute C_s^1 requires a recurrence solution:

$$C_s^1(r+1) \leftarrow (t - Oth(t)) \;/\; \lfloor \frac{t + T_i - C_s^1(r)}{T_i} \rfloor \qquad (7)$$

The starting value, $C_s^1(1)$, is that computed in step 1 when U=1. If processor p is schedulable without τ_s (which is the assumption) then equation (7) will provide a solution. Note the sequence $C_s^1(1)$, $C_s^1(2)$, ..., is monotonically non-increasing. In exceptional circumstances (when the processor cannot accommodate any further load) the value of C_s^1 will be zero.

The fact that the repeated application of equation (7) delivers the optimal (ie. largest) value for C_s^1 is obtained from the following observations. If $C_s^1(1)$ is a solution to equation (7) then it must be optimal as U equals 1. Otherwise, for each iteration of the equation, a value $C_s^1(r+1)$ is computed which is the maximum computation time for C_s^1 that is achievable with a deadline of D_s^1 = $C_s^1(r) \geq C_s^1(r+1)$. The deadline is then reduced and the new maximum value of C_s^1 computed. When $C_s^1(r+1)$ = $C_s^1(r)$ the deadline is equal to the computation time and computation time is at its largest value.

The double reduction of C_s^1 and D_s^1 is the reason why the approach requires (step 3) that if there is any recomputed values then the algorithm must check from L again. If only a task's computation time is being reduced then only a single pass of the QPA algorithm is needed. Having reduced C_s^1 to remove the failure at time t then it has been proved [29], [30] that all values greater then t will remain safe (no deadline failures). Unfortunately when D_s^1 is also reduced it is possible (though unlikely) that a new failure point (f) may arise with $t < f < L$.

It is possible to compute a new starting value of L that would be smaller than the original value; but this optimisation is not explored further here. Rather a simple scheme is used that returns to the original L if there has been any failure identified. Only when there has been no failure does the algorithm terminate and the resulting C_s^1 is then the largest possible computation for the first phase of the split task compatible with the C=D constraint. Note that termination is assured as each iteration must reduce the value of C_s^1.

The above scheme, whilst straightforward in its form, suffers from a potentially exponential growth in execution time due to:

- A staring value of $U = 1$ than means that L_B must be used, and
- When $U = 1$, L_B is equal to the LCM of the periods of the tasks assigned to the processor, and that may be very large.

In the evaluation section (below) task sets are generated randomly. With $U = 1$ and $D = T$, for twenty or more tasks the LCM (and hence L_B) could indeed be prohibitively large. To counter this an alternative scheme is possible. Rather than start with $U = 1$ a value of, say, $U = 0.99$ is used. Now L_A (Eqn 4) can be employed and a reasonable starting value can be computed. An inspection of equation (4) shows that for most tasks (in our evaluations) $T = D$ and hence there are only two terms (from the two split tasks) in the formulation for each processor (indeed for the first and last processor there is only one).

If, when starting from $U = 0.99$, a failure is found (and therefore C_s^1 and D_s^1 are reduced) then the scheme will deliver the optimal value of C_s^1. If no failure is found then either a suboptimal result, but with a processor utilisation of 0.99, could be deemed accepted or the process repeated with $U = 0.999$ etc. In practice a utilisation of exactly 1 would never be used as some level of tolerance would be expected.

At a practical level it would always be necessary to bound the minimum size of the initial phase of a split task to be greater then the extra overheads introduced into the system by the required task migration.

D. Illustrative Examples

For a very simple first example consider three tasks each with $C = 66$ and $D = T = 100$. Clearly the utilisation of this task set is almost 2 (actually 1.98). A fully partitioned approach would require three processors (one per task). The C=D scheme delivers a two processor system (even when a migration overhead of 1 is assumed). Table II contains the details of the split task.

Task	T	D	C	p
τ_1	100	100	66	1
τ_2^1	100	34	34	1
τ_2^2	100	66	33	2
τ_3	100	100	66	2

TABLE II
TWO PROCESSOR TASK SET

The second task (τ_2) executes first for 34 ticks on processor 1 with task τ_1. It has a computation time equal to deadline equal to 34. Processor 1 is schedulable. The second part of τ_2 is released at time 34, its has a computation time of 33 (66-34+1) and a deadline of 66. Processor 2 containing all of τ_3 and this second part (τ_2^2) is also schedulable.

For another example consider again the task set given in Table I. This has a total utilisation of 1. In Table III the computation times of the tasks are increased to give a total utilisation of approximately 2.9. In this example the cost of migrating the two split tasks is ignored.

Task	T	D	C	U
τ_1	10	10	5	.5
τ_2	12	12	6	.5
τ_3	15	15	6	.4
τ_4	16	16	6	.375
τ_5	20	20	9	.45
τ_6	40	40	14	.35
τ_7	48	48	16	.333

TABLE III
THREE PROCESSOR EXAMPLE

A simple first fit allocation scheme is used based on smallest utilisation first[3] (so the order of tasks is τ_7, τ_6, τ_4, τ_3, τ_5, τ_2 and τ_1). For processor 1, τ_7 and τ_6 can be fully allocated; τ_4 is then split – processor 1 can accommodate 5 units of computation time (with a deadline of 5) leaving 1 to be provided on processor 2. On processor 2, τ_4^2 has a deadline of 11 (16-5). Also on to this processor can be allocated τ_3 and τ_5, leaving τ_2 or τ_1 to be split (they have the same utilisation). Splitting τ_2 leaves the final processor with most of τ_2 and all of the final task, τ_1. Table IV has the derived parameters for this task set.

Task	T	D	C	p
τ_1	10	10	5	3
τ_2^2	12	11	5	3
τ_2^1	12	1	1	2
τ_5	20	20	9	2
τ_3	15	15	6	2
τ_4^2	16	11	1	2
τ_4^1	16	5	5	1
τ_6	40	40	14	1
τ_7	48	48	16	1

TABLE IV
THREE PROCESSOR EXAMPLE

The computed utilisations of the three processors are 0.9958, 0.9958 and 0.9545 (actually the value of τ_4^1 has been rounded down slightly to 5, if the actual value is used the utilisation of the first processor is 1). Obviously the utilisation of the final processor in any allocation is arbitrary – it depends on the task set's total utilisation. But the utilisation of the first 2 processors, in this example, give an indication of the effectiveness of the scheme. In the next section we will evaluate the C=D scheme over a large set of randomly generated task sets. The evaluation criteria for the first experiment will be the average utilisation of the first $m - 1$ processors – the closer this is to 1 the better the scheme. We leave to future work the derivation of a utilisation lower bound for the scheme – a preliminary result is however included in the Appendix.

III. EVALUATION

In this section, we present an initial empirical investigation, examining the effectiveness of task splitting using the approach described in this paper. The emphasis of these experiments is to show the average performance of the scheme and to illustrate that it can be used with a variety of 'bin packing' methods. The experiments should not be viewed as evidence of the optimal performance of the $C = D$ approach. Future work will look to develop the best compatible 'bin packing' method. Please note the graphs are best viewed online in colour.

[3]This scheme is chosen here to illustrate the versatility of the $C = D$ approach – in the evaluation section below the opposite and more effective ordering is used (decreasing utilisation/density).

A. Task set parameter generation

The task set parameters used in our experiments were randomly generated as follows:

- Task utilisations were generated using the UUnifast-Discard algorithm [12], giving an unbiased distribution of utilisation values.
- Task periods were generated according to a log-uniform distribution with a factor of 100 difference between the minimum and maximum possible task period. This represents a spread of task periods from 10ms to 1 second, as found in many hard real-time applications.
- Task execution times were set based on the utilisation and period selected: $C_i = U_i/T_i$.
- To generate constrained deadline task sets (for the second experiment), task deadlines were assigned according to a uniform random distribution, in the range $[C_i,T_i]$.

B. Algorithms investigated

We investigated the performance of four algorithms all of which were based on First-Fit partitioning [23]:

1) "EDF Partition (DD)": Allocates tasks to processors using First-Fit in Decreasing Density order[4], and uses an exact EDF schedulability test (QPA)[27] to determine the schedulability of tasks allocated to each processor.
2) "EDF Split (DD)": Allocates tasks to processors using First-Fit Decreasing Density order, and determine the schedulability of tasks allocated to each processor. Once no further tasks can be allocated to the first processor, the remaining task with the largest density is split, then task allocation continues for the next processor in Decreasing Density order and so on[5]. The approach adopted follows the 'more efficient' scheme defined in Section II-C, the upper bound on each processor's utilisation is fixed at 0.9999 (rather then 1).
3) "EDF Partition (rDM)": Similar to "EDF Partition (DD)" however the tasks are allocated in reverse Deadline Monotonic order; that is longest relative deadline first.
4) "EDF Split (rDM)": Similar to "EDF Split (DD)"; however the tasks are allocated / split in reverse deadline monotonic order.

Algorithms 1) and 3) are pure partitioning schemes and are included as benchmarks for the other schemes. Algorithms 2) and 4) present the $C = D$ scheme for two different First-Fit methods.

C. Experiment 1

[4]Decreasing Density combined with a First Fit approach has been shown to give good average performance[16], [26], [13].

[5]The scheme used in the evaluation is slightly different from that described in Section II-B – depending on the bin packing scheme to be employed, it is possible for the second phase of a split task to be chosen as the task to be split on its second processor. As a result, the task is split again. The basic approach is however maintained, there are at most m-1 migrations between processors, all but the last phase of any task has $D = C$, and different phases of the same task can never execute concurrently.

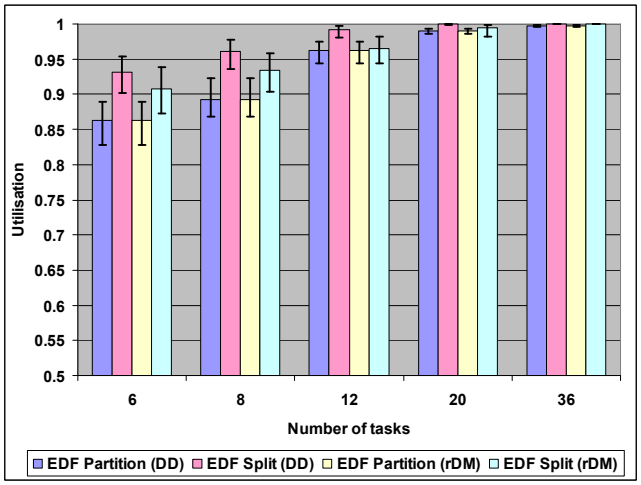

Fig. 2. Performance with $U = 4$ and $D = T$

In this experiment, we generated 1000 task sets with cardinalities of 6, 8, 12, 20, and 36, and a total utilisation of 4. We allowed each algorithm as many processors as it required to schedule each task set. For each algorithm, and each task set, we determined the average utilisation of the fully occupied processors; that is the processors which were allocated tasks with the exception of the highest indexed (partially used) processor.

Fig 2 shows, for implicit deadline task sets, the median (50 percentile) of the average utilisation of fully occupied processors for each algorithm and value of task set cardinality. The error bars indicate the 25 and 75 percentiles. For implicit deadline task sets there is a clear, potential achievable, upper bound of 1. For large numbers of tasks this bound is approached for both of the $C = D$ schemes. It is also approached, though more slowly, by the partitioned EDF schemes. With smaller numbers of tasks there is a significant improvement demonstrated by the EDF (DD) splitting approach. For example, with 8 tasks of average utilisation of 0.5 each, the average utilisation of the 4 'full' processors is over 0.95.

D. Experiment 2

Fig 3 shows the results of the same experiment performed on constrained deadline task sets. Here the 'achievable' upper bound is not straightforward to compute, but the effectiveness of the $C = D$ scheme when used with First-Fit Decreasing Density is clear. However, the alternative packing approach (rDM) is nowhere near as effective, implying that for constrained deadline task sets the optimal performance of the scheme is dependent on the 'bin packing' approach – an issue that will be taken up in further work.

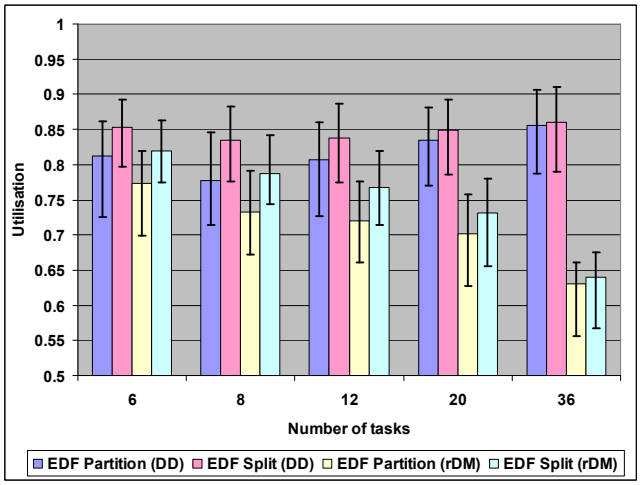

Fig. 3. Performance with $U = 4$ and $D \leq T$

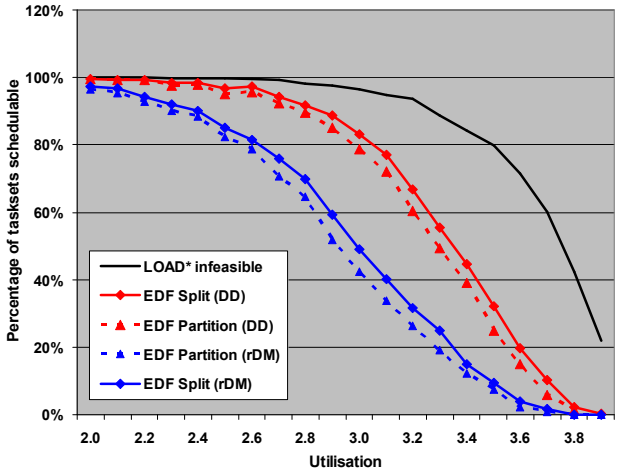

Fig. 4. Performance with 4 processors, 12 tasks and $D \leq T$

E. Experiment 3

In this experiment, the task set utilisation was varied from 0.025 to 0.975 times the number of processors in steps of 0.025. For each utilisation value, 1000 task sets were generated and the schedulability of those task sets determined using the various algorithms, assuming the fixed number of processors studied. The graphs plot the percentage of task sets generated that were deemed schedulable in each case, see Fig 4. Note the lines on all of the graphs appear in the order given in the legend. The algorithms were also compared to the LOAD* infeasibility test of Baker and Cirinei [3]. This line indicates the percentage of task sets that are not known to be infeasible according to the test at each utilisation level. It represents the currently best know upper bound on achievable schedulability.

Fig 4 shows the percentage of constrained-deadline task sets that were deemed schedulable by each algorithm on a 4 processor system.

The results clearly indicate that DD is better than rDM, and that the splitting schemes make a small but significant improvement over their fully partitioned equivalent. This improvement should be interpreted as the minimum that the scheme can achieve. With a better packing scheme more task sets will be deemed schedulable. One potential means of improving the results is the use a different criteria for identifying the 'task to be split' from the next 'task to pack'. An exploration of these issues will be made as part of further work.

IV. Conclusions

This paper has introduced a task splitting scheme for EDF scheduled identical multiprocessors. The motivation for the scheme is ease of implementation and low overheads. For m processors at most m-1 tasks are split. Each processor runs a standard EDF policy, with the split tasks changing their affinities after a fixed period of time after their releases. The first part of any split task is constrained to have its deadline equal to its computation time. It therefore runs (in effect) non-preemptively on its initial processor. This provides the maximum time possible, on the subsequent processor, for the task to complete the remainder of its computation time. Analysis is provided by which the optimal parameters of the split task can be obtained.

Evaluation over a wide range of randomly generated task sets is provided and these results indicate that the scheme does indeed have promising performance. Nevertheless a number of issues for further study are immediately apparent.

- An evaluation is needed for task sets with arbitrary deadlines.
- An evaluation is needed of the run-time performance of the scheme when compared with other EDF-based task-splitting approaches.
- Other approaches to task allocation need to be considered, including other first fit methods such as largest D-C first, and largest T/C first – also various possible best fit algorithms.
- The development of utilisation-based bounds for this C=D scheme.
- The incorporation of possible pre-allocation schemes for heavy (high utilisation) tasks – this has proved to be a useful approach with other partitioning schemes.

The overall conclusion of this study, confirming the views expressed in a number of papers on similar approaches, is that minimal task splitting seems to be a practically useful means

of scheduling multiprocessor systems. Most of the advantages of the purely partitioned approach are maintained, but higher levels of processor utilisation can be delivered. At the same time few of the disadvantages of the purely global approach are encountered.

ACKNOWLEDGEMENTS

The work reported in this paper is supported, in part, by the EPSRC project TEMPO (EP/G055548/1).

REFERENCES

[1] B. Andersson, K. Bletsas, and S.K. Baruah. Scheduling arbitrary-deadline sporadic task systems on multiprocessors. In *IEEE Real-Time Systems Symposium*, pages 385–394, 2008.

[2] B. Andersson and E. Tovar. Multiprocessor scheduling with few preemptions. In *RTCSA*, pages 322–334, 2006.

[3] T.P. Baker and M. Cirinei. A necessary and sometimes sufficient condition for the feasibility of sets of sporadic hard-deadline tasks. In *Work-In-Progress (WIP), RTSS*, 2006.

[4] P. Balbastre, I. Ripoll, and A. Crespo. Pptimal deadline assignment for periodic real-time tasks in dynamic priority systems. In *Euromicro Conference on Real-Time Systems (ECRTS)*, 2006.

[5] S.K. Baruah and A. Burns. Sustainable schedulability analysis. In *IEEE Real-Time Systems Symposium (RTSS)*, pages 159–168, 2006.

[6] S.K. Baruah, R.R. Howell, and L.E. Rosier. Feasibility problems for recurring tasks on one processor. *Theorectical Computer Science*, 118:3–20, 1993.

[7] S.K. Baruah, A.K. Mok, and L.E. Rosier. Preemptive scheduling of hard real-time sporadic tasks on one processor. In *IEEE Real-Time Systems Symposium (RTSS)*, pages 182–190, 1990.

[8] E. Bini and G.C. Buttazzo. Measuring the performance of schedulability tests. *Real-Time Syst.*, 30(1-2):129–154, 2005.

[9] K. Bletsas and B. Andersson. Notional processors: An approach for multiprocessor scheduling. In *IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 3–12, 2009.

[10] A. Burns and A. J. Wellings. *Real-Time Systems and Programming Languages*. Addison Wesley Longman, 4th edition, 2009.

[11] A. Burns and A.J. Wellings. Dispatching domains for multiprocessor platforms and their representation in ada. In J. Real and T. Vardanega, editors, *Proceedings of Reliable Software Technologies - Ada-Europe 2010*, volume LNCS 6106, pages 41–53. Springer, 2010.

[12] R.I. Davis and A. Burns. Priority assignment for global fixed priority pre-emptive scheduling in multiprocessor real-time systems. In *Proceedings of RTSS*, pages 398–409, 2009.

[13] R.I. Davis and A. Burns. A survey of hard real-time scheduling algorithms for multiprocessor systems. *Accepted for publication in ACM Computing Surveys*, 2010.

[14] N. Guan, M. Stigge, W. Yi, and Ge Yu. Fixed priority mulitprocessor scheduling with liu and layland utilization bound. In *Proceedings of the IEEE Real-Time Technology and Applications Symposium (RTAS)*. IEEE, April 2010.

[15] H. Hoang, G.C. Buttazzo, M. Jonsson, and S. Karlsson. Computing the minimum EDF feasible deadline in periodic systems. In *RTCSA*, pages 125–134, 2006.

[16] D.S. Johnson. *Near-Optimal Bin- Packing Algorithms*. PhD thesis, Department of Mathematics, MIT, 1974.

[17] S. Kato and N. Yamasaki. Real-time scheduling with task splitting on multiprocessors. In *RTCSA*, pages 441–450, 2007.

[18] S. Kato and N. Yamasaki. Portioned EDF-based scheduling on multiprocessors. In *EMSOFT*, pages 139–148, 2008.

[19] S. Kato and N. Yamasaki. Portioned static-priority scheduling on multiprocessors. In *IPDPS*, pages 1–12, 2008.

[20] S. Kato and N. Yamasaki. Semi-partitioned fixed-priority scheduling on multiprocessors. In *IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 23–32, 2009.

[21] K. Lakshmanan, R. Rajkumar, and J. Lehoczky. Partitioned fixed-priority preemptive scheduling for multi-core processors. In *ECRTS '09: Proceedings of the 2009 21st Euromicro Conference on Real-Time Systems*, pages 239–248, 2009.

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

[23] J.M. Lopez, M. Garcia, J.L. Diaz, and D.F. Garcia. Worst-case utilization bound for EDF scheduling on real-time multiprocessor systems. In *Proceedings of ECRTS*, pages 25–33, 2000.

[24] I. Ripoll and A.K. Mok. Improvement in feasibilty testing for real-time tasks. *Journal of Real-Time Systems*, 11(1):19–39, 1996.

[25] M. Spuri. Analysis of deadline schedule real-time systems. Technical Report 2772, INRIA, France, 1996.

[26] A.C. Yao. New algorithms for bin packing. *Journal of the ACM*, 27(2), 1980.

[27] F. Zhang and A. Burns. Schedulability analysis for real-time systems with EDF scheduling. Technical Report YCS 426, University of York, 2008.

[28] F. Zhang and A. Burns. Schedulability analysis for real-time systems with EDF scheduling. *IEEE Transaction on Computers*, 58(9):1250–1258, 2008.

[29] F. Zhang, A. Burns, and S. Baruah. Sensitivity analysis for real-time systems. Technical Report YCS 438, University of York, Computer Science Dept., 2009.

[30] F. Zhang, A. Burns, and S. Baruah. Sensitivity analysis for EDF scheduled arbitrary deadline real-time systems. In *RTCSA (to appear)*, 2010.

APPENDIX - TOWARDS A UTILISATION BOUND

As indicated above, the derivation of a utilisation bound for the $C = D$ scheme for implicit deadline tasks forms part of further work. However, in this section we give a two-task example that indicates that the bound can be no higher than 0.833. Here we are concerned with a single processor in which one task has its deadline set to its computation time and all the other tasks have deadline equal to period. Consider the simple task set defined by the parameters given in Table V.

Task	T	D	C
τ_1	2	2	1
τ_2	3	3	1

TABLE V
TASK SET WITH LOW UTILISATION

This task set can have D_1 reduced to 1 and remain schedulable. But if either tasks' computation time is increased by an infinitesimal small amount then it loses this property. Hence this task set is at the boundary of retaining the one-task $C = D$ property. The utilisation of the task set is 1/2 + 1/3 = 5/6 = 0.833.

In all the experiments reported in Section II-A no failing example with utilisation less than 0.833 was found. The bound for the task splitting scheme is, of course, also dependent on the 'bin packing' approach.