Real-Time Syst (2012) 48:3–33
DOI 10.1007/s11241-011-9126-9

Partitioned EDF scheduling for multiprocessors using a $C = D$ task splitting scheme

A. Burns · R.I. Davis · P. Wang · F. Zhang

Published online: 7 May 2011

Abstract An EDF-based task-splitting scheme for scheduling multiprocessor systems is presented 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. Different heuristics for choosing the task to split are derived and evaluated. Issues pertaining to the implementation of the $C = D$ scheme on Linux or via the Ada programming language are also discussed.

Keywords Scheduling · Multiprocessors · Multi-core · Task-splitting

1 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

A. Burns (✉) · R.I. Davis · P. Wang · F. Zhang
Department of Computer Science, University of York, York, UK
e-mail: burns@cs.york.ac.uk

the fully partitioned to the totally global. It is unlikely that a single scheme will meet the needs of all applications (Davis and Burns 2011).

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 difficulties 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 may 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.

1.1 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 ($\tau_1 \ldots \tau_n$) 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 (Burns and Wellings 2009). 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 (Liu and Layland 1973). In this analysis we assume tasks do not experience release jitter. We are concerned with analysis that is necessary, sufficient and sustainable (Baruah 2006).

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) (Baruah et al. 1990, 1993). 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 floor function capped below by 0 (i.e. 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 L on the values of t that need to be checked. 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* (Ripoll et al. 1996; Spuri 1996). 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 iterating 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 (Hoang et al. 2006):

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

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 (Zhang and Burns 2008a, 2008b). This test, known as QPA (Quick convergence Processor-demand Analysis), starts from time L and iterates backwards towards time 0 checking a small subset of time points. These points are proved (Zhang and Burns 2008a, 2008b 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 (e.g. 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 declared schedulable.

1.2 Previous related research

A number of papers have been published on partitioning and, specifically, task splitting. Andersson and Tovar introduced in 2006 (Andersson and Tovar 2006) 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 $50 + \delta\%$ can be accommodated on m processors (for arbitrary small δ).

Andersson et al. later extended this approach to task sets with arbitrary deadlines (Andersson et al. 2008). 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' (Bletsas and Andersson 2009). 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 (Kato and Yamasaki 2007). 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 (Kato and Yamasaki 2008a), 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 (2008b) 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 (Kato and Yamasaki 2009). DM-PM strictly dominates fully partitioned fixed task priority approaches, as tasks are only permitted to migrate if they will not 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 (Kato et al. 2009) (EDF with Window constrained Migration).

For fixed priority scheduling Lakshmanan et al. (2009) 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. (2010) 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 (2011) from which the above discussion is distilled.

1.3 Contribution and organisation

In this paper[1] we motivate, describe and evaluate the 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 (with no unusual RTOS functions required, only a standard timer); CPU time monitoring is not necessary. The scheme has low overheads that can easily be accommodated into the analysis. It also utilises an off-line analysis-based procedure that exploits some key properties of EDF scheduling (for single processors). Note it has some resemblance to the fixed priority scheme of Lakshmanan et al. (2009) 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; although some of the 'bin-packing' schemes such as largest utilisation first do deal with heavy tasks first. 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 Sect. 6.

[1] A preliminary version of this paper was presented at RTNS 2010 (Burns et al. 2010).

2 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.

2.1 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 $U = 1$) (Liu and Layland 1973); it is not necessary to employ QPA. However, using an extension of QPA (Zhang 2010) 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[2]; hence QPA is employed. For this task set, each task can have its deadline reduced to the minimum value of 2 (i.e. $D = C$) 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 the work of Balbastre et al. (2006).

A less constrained example is given in Table 1. 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 (units of execution time) in 5, but the 6th tick takes until time 26.

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 (Bini and

Table 1 Example task set

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

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

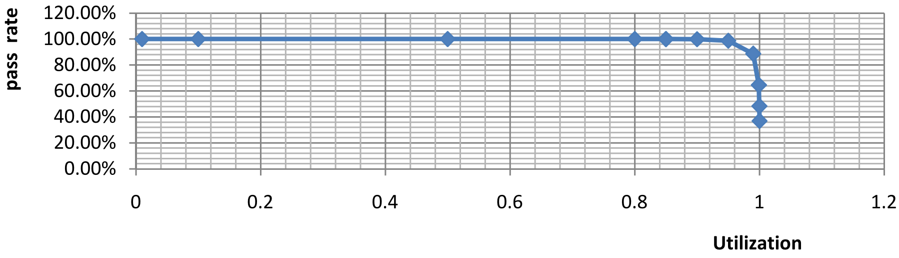


Fig. 1 Task sets with one $C = D$ task

Buttazzo 2005) was employed to generate 10,000 task sets per experiment, each with implicit deadlines, i.e. $D = T$. 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.

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.

2.2 The $C = D$ scheme

The task splitting scheme introduced in this paper involves two distinct activities: the mapping of complete tasks to the available processors, and the choice of which tasks to split and their allocation. These two activities give rise to a number of strategies/heuristics for employing the basic $C = D$ scheme:

- the allocation of complete tasks to processors, referred to as the 'bin-packing' phase can take a number of forms including *first-fit* and *best-fit*, and can be based on different task parameters, for example decreasing density or decreasing utilisation;
- the choice over which task to split can also be based on different task parameters including increasing deadline or increasing computation time;
- finally the phasing of the allocation and splitting activities—this could be undertaken in an integrated way or be undertaken serially with the allocation of most tasks occurring first followed by the splitting of the few tasks that were reserved for this activity.

These strategies are considered further in Sect. 3. And an evaluation of a number of these strategies is given in Sect. 4. In the following, a simple example strategy is defined which uses an integrated approach with 'first-fit' bin packing for tasks that can be assigned to one processor and 'next-fit' for tasks to be split:

- Each processor (p) is 'filled' with tasks until no further task can be added without leading to unschedulability of the processor.
- A single non-allocated 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 $(5, 30, 30)$—i.e. 5 units of execution every 30 with a deadline of 30. After splitting, its first part may be restricted 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.

In the required schedulability analysis of the processor with the second part of the task, the worst-case critical instant of all tasks being released together is assumed. If the processor is schedulable when all task are released at the same time then it will remain schedulable if there is a phased release between any two tasks, as there might be between the second part of one split task and the first part of another split task.

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, see Sect. 5.2). 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 migration occurs after a period of 'real' (elapse) 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. At run-time there is actually only one thread per split task that moves/migrates between the two processors. It also follows that there are at most $m - 1$ split tasks.

Before giving the required analysis, and reporting on an evaluation of this $C = D$ scheme, two useful properties of the scheme are worth emphasising. These properties are only strictly true if the overhead of splitting a task is ignored—they are however indicative of the benefits that would be manifest when realistic overheads are included. Again consider a completely allocated task set in which task τ_s is split between processors p and $p + 1$. If it were not to be split (i.e. 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 (Baruah 2006) 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 potential 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 (if overheads are ignored).

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. This informal proof of dominance applies to the particular case where the starting point for task splitting is the fully allocated scheme delivered by a non-splitting partitioning approach. Where splitting and allocation are integrated it does not follow that the result is inevitable better than just using allocation; it is possible that a poor choice for splitting may cause allocation problems later in the process.

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 a potential 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. The gain is only 'potential' as the deadline for the task has been reduced.

Note that although the $C = D$ task splitting scheme dominates any partitioning method, the best overall scheme may not result from this approach—see evaluation section (Sect. 4).

2.3 $C = D$ 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.

Assume a schedulable processor p becomes unschedulable when task τ_s is added. This task must therefore be split with only the first part, with parameters $(C_s^1, D_s^1 = C_s^1, T_s)$, being allocated to p. The following steps are undertaken to derive C_s^1.

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 Sect. 1.1).
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; if no failure then the current value of C_s^1 is the optimum one.

Assuming there is a failure at time t, i.e. $h(t) > t$. 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:

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

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

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

releases of τ_s.

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

$$C_s^1 = (t - Oth(t)) \Big/ \left\lfloor \frac{t + T_s - D_s^1}{T_s} \right\rfloor \tag{5}$$

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

$$C_s^1 \leftarrow (t - Oth(t)) \Big/ \left\lfloor \frac{t + T_i - C_s^1}{T_i} \right\rfloor \tag{6}$$

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:

$$C_s^1(r+1) \leftarrow (t - Oth(t)) \Big/ \left\lfloor \frac{t + T_i - C_s^1(r)}{T_i} \right\rfloor \tag{7}$$

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 (7) will provide a solution; that is, there exists a v such that $C_s^1(v+1) = C_s^1(v) = C_s^1$. Note the sequence $C_s^1(1)$, $C_s^1(2), \ldots$, 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 (7) delivers the optimal (i.e. largest) value for C_s^1 is obtained from the following observations. If $C_s^1(1)$ is a solution to (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 (within step 7) that if there is any recomputed values then the algorithm must return to step 3 and 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 (Zhang 2010) 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 (for Rational-valued parameters) 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:

Table 2 Two processor task set

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

- A starting value of $U = 1$ which 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, $U = 0.9999$ is used. Now L_A (4) can be employed and a reasonable starting value can be computed. An inspection of (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 $U = 0.9999$, 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 0.9999, could be deemed accepted or the process repeated with $U = 0.99995$ etc. In practice a utilisation of exactly 1 would never be used as some level of tolerance would be expected.

2.4 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 2 contains the details of the split task.

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.

An alternative model for the overhead is that the migration takes 1 unit of time so that τ_2^2 is released at time 35 with a computation time of 32 and a relative deadline of 65. But note that requiring 33 units of computation in 66 must imply that 32 are provided in 65—hence schedulability of the first formulation implies schedulability of the second.

For another example consider again the task set given in Table 1. This has a total utilisation of 1. In Table 3 the computation times of the tasks are increased to give a

Table 3 Three processor example

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 4 Three processor example

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

total utilisation of approximately 2.9. In this example the cost of migrating the two split tasks is ignored.

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 4 has the derived parameters for this task set.

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

[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).

is to 1 the better the scheme. We leave to future work the derivation of a utilisation lower bound for the scheme.

3 Strategies for selecting tasks to split

In this section, we present two strategies aimed at choosing which tasks to split. Both of these strategies build on an underlying partitioning algorithm which uses First-Fit allocation, and the exact EDF schedulability test, QPA, described in Sect. 1.1 to determine the schedulability of the tasks assigned to each processor.

In the following, we use the term packing order to refer to the order in which tasks are selected for allocation (of the whole task) to a processor, and splitting order to refer to the order in which tasks are selected for splitting. Different heuristics can be used to define these two orders, for example decreasing density is an effective packing order, while minimum (i.e. increasing) deadline is an effective splitting order.

The two strategies are as follows:

1) *Continuous*: the choice of which tasks to split occurs during task allocation, and so is affected by both the packing order and the splitting order.
2) *Pre-selection*: a subset of the tasks is pre-selected for splitting. Tasks that are not preselected are not split and are all allocated to processors prior to the subsequent allocation and splitting of the preselected tasks. With this strategy, the tasks that are split are chosen solely according to the splitting order.

The continuous strategy requires only a single pass over the set of tasks, and is described by the pseudo code shown in Algorithm 1 (in which all the tasks are initially 'unassigned').

```
P = 0          //P = 1 is the first processor
while(unassigned tasks) {
  P=P+1 //next processor
  for each unassigned task T in packing order {
    if (task T is schedulable on processor P){
      assign task T to processor P
    }
  }
  if(unassigned tasks) {
    S = first unassigned task in splitting order
    determine the max C=D values for the first
      part (S1) of task S on processor P
    allocate S1 to processor P
    the second part (S2) is now an unassigned
      task
   }
 }
```

Algorithm 1: Continuous strategy

The pre-selection strategy requires multiple passes over the set of tasks. The idea here is to use the minimum number of pre-selected tasks, such that after the remaining tasks have been allocated (without splitting), the pre-selected (splitable) tasks can utilise spare capacity on all of the processors used. However, prior to completing task allocation, it is not possible to determine what this minimum number of pre-selected tasks should be. To address this problem requires iteration over increasing numbers of preselected tasks (starting from just one) until the number pre-selected becomes just enough to utilise capacity on all of the processors. The pseudo code for the pre-selection strategy is given in Algorithm 2.

```
repeat = true
K = 0   //number of tasks that can be split
while(repeat) {
  repeat = false
  K = K + 1
  P = 0      // P = 1 is the first processor
  pre-sel tasks = the first K tasks in splitting order
  unassigned tasks = the tasks not pre-selected
  while(unassigned tasks) {
   P=P+1       //next processor
   for each unassigned task T in packing order {
     if (task T is schedulable on processor P){
      allocate task T to processor P
     }
   }
  }
  usedP = P   // number of processors used
  P = 1       // P = 1 is the first processor
  unassigned tasks = pre-sel tasks
  while(unassigned tasks) {
    S = first unassigned task in splitting order
    if (task S is schedulable on processor P){
     allocate task S to processor P
    }
    else {
      split task S into two parts S1 and S2
      determine the max C=D values for the first
      part (S1) of task S on processor P
      allocate S1 to processor P
      second part (S2) is now an unassigned task
      P = P + 1       // next processor
    }
  }
  if(P < usedP) {
  repeat = true
  }
}
```

Algorithm 2: Pre-selection strategy

Both the continuous and pre-selection strategies can be used with a variety of packing orders and splitting order heuristics. Note that some splitting heuristics will actually lead to the same task being split more than once. It will have a first phase (with $D = C$) a second, third or more phases with $D = C$ and a final phase with the remaining computation time being delivered within the remaining deadline. The analysis of this variant of the algorithm is identical to that presented for the normal two phase model. At most there will be m splits (for m processors). This can range from m tasks split once to a single task split m times (or any position between these extremes).

4 Empirical investigation

In this section, we present an empirical investigation, examining the effectiveness of our approach to task splitting and the two strategies described in the previous section. Please note the graphs are best viewed online in colour.

4.1 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 (Davis and Burns 2009, 2010), 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 10 ms 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 $[C_i, T_i]$.

4.2 Algorithms investigated

We investigated the performance of five algorithms all of which are based on First-Fit partitioning (Lopez et al. 2000) and use Decreasing Density (DD) as the packing order as it is widely recognised as an effective packing heuristic (Johnson 1974; Yao 1980; Davis and Burns 2011).

1) "EDF Partition (DD)": This is the baseline algorithm with no task splitting, just First-Fit Decreasing Density partitioning.
2) "EDF Split (DD-MaxD)": This is Algorithm 1, with Decreasing Density as the packing order and Maximum (Decreasing) Deadline as the splitting order.
3) "EDF Split (DD-MaxDen)": This is Algorithm 1, with Decreasing Density as both the packing order and the splitting order.
4) "EDF Split (DD-MinD)": This is Algorithm 1, with Decreasing Density as the packing order and Minimum (Increasing) Deadline as the splitting order.
5) "EDF Split then Pack(DD-MinD)": This is Algorithm 2 using the pre-selection strategy, with Decreasing Density as the packing order and Minimum (Increasing) Deadline as the splitting order.

4.3 Experiment 1

In this experiment, we generated 1000 task sets with cardinalities of 6, 8, 12, 20, and 36, and a total utilisation of 4 (or cardinalities of 16, 20, 28, 44 and 76 and a utilisation of 8). 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. Figure 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, potentially achievable, upper bound of 1. For large numbers of tasks this

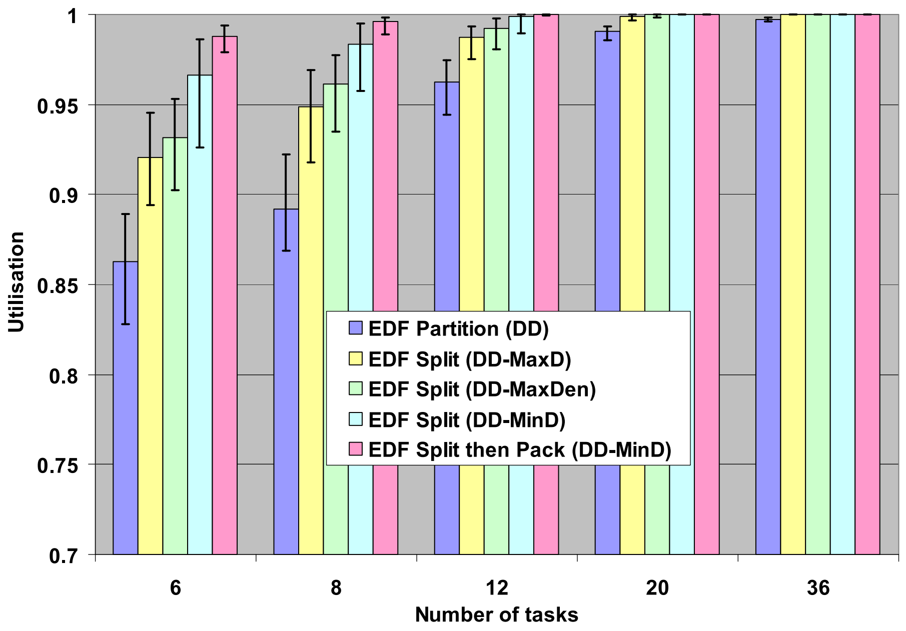


Fig. 2 Performance with $U = 4$ and $D = T$

bound is approached for the $C = D$ task splitting schemes. It is also approached, though more slowly, by the baseline partitioning scheme. With smaller numbers of tasks there is a significant improvement demonstrated by the task splitting schemes using Minimum Deadline as the splitting order. For example, for 8 tasks with an average utilisation of 0.5 each, the average utilisation of the 4 'full' processors is approximately 98% using Algorithm 1, and over 99% using the pre-selection strategy (Algorithm 2). Note also that the 25 and 75 percentile bounds are much closer together for Algorithm 2.

For task sets with cardinalities of 16, 20, 28, 44, and 76, and a total utilisation of 8, the results are shown in Fig. 3. The results are broadly similar to those for a utilisation of 4. We note that as the number of tasks increases the probability of achieving a very high utilisation for the first few processors increases, and so does the average utilisation achieved by all of the algorithms.

We experimented with a range of different heuristics for the task splitting order, including minimum execution time and minimum period, as well as maximum execution time and maximum period. We found that these minimum (maximum) heuristics gave very similar results to those for minimum (maximum) deadline. This is because there is a strong correlation between the values of these parameters; if a task has a large period, then it is likely that it will also have a long deadline and large execution time.

We also repeated this experiment for constrained deadline task sets. Here the 'achievable' upper bound is not straightforward to compute, but the effectiveness of the $C = D$ task splitting schemes is clear to see in Figs. 4 and 5.

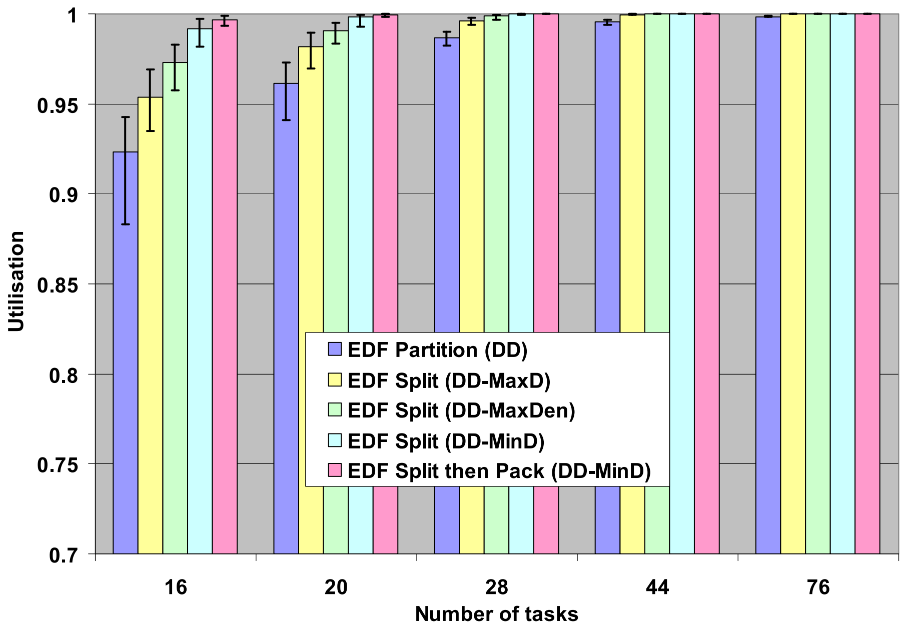


Fig. 3 Performance with $U = 8$ and $D = T$

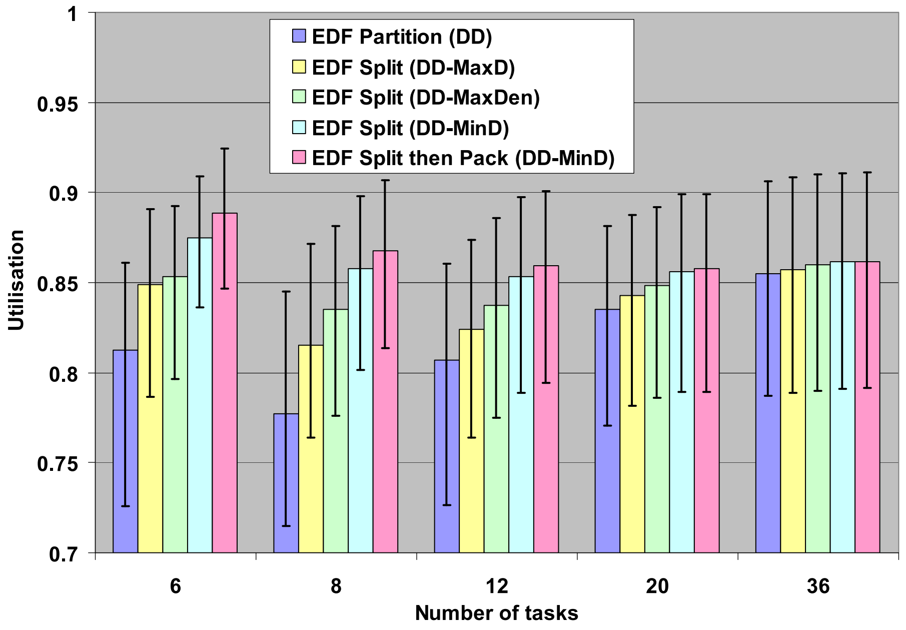


Fig. 4 Performance with $U = 4$ and $D \leq T$

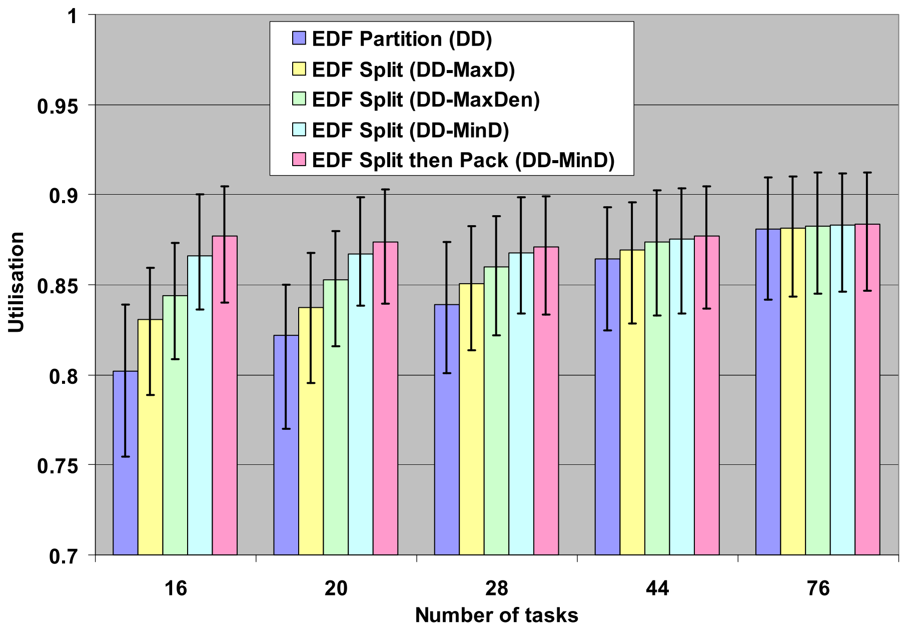


Fig. 5 Performance with $U = 8$ and $D \leq T$

4.4 Experiment 2

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 five algorithms, assuming the fixed number of processors studied. The graphs plot the percentage of task sets generated that were deemed schedulable in each case. Note the lines on all of the graphs appear in the order given in the legend.

Figures 6 and 7 illustrate the performance of the algorithms for 4 and 8 processor systems respectively, with implicit deadline tasksets, and two tasks per processor (i.e. 8 tasks in the 4 processor case, and 16 tasks in the 8 processor case). In Fig. 6, just under 90% of the tasksets generated where schedulable using the baseline partitioning algorithm 'EDF Partition (DD)' at a utilisation of $0.89m$, where m is the number of processors, whereas, with the pre-selection task splitting algorithm 'EDF Split then Pack(DD-MinD)', the same proportion of tasksets were schedulable at a utilisation of $0.99m$. Figure 7 shows a similar level of performance improvement in the 8 processor case.

We repeated this experiment for constrained deadline task sets. Figure 8 shows the percentage of constrained-deadline task sets that were deemed schedulable by each algorithm on a 4 processor system, with Fig. 9 showing similar results for an 8 processor system. These results indicate a significant improvement in schedulability over the baseline partitioning algorithm. For example, in the 4 processor case (Fig. 8) with EDF Partition (DD), approximately 50% of the tasksets generated with a utilisation of $0.8m$ were unschedulable. Whereas, with the $C = D$ task splitting scheme using

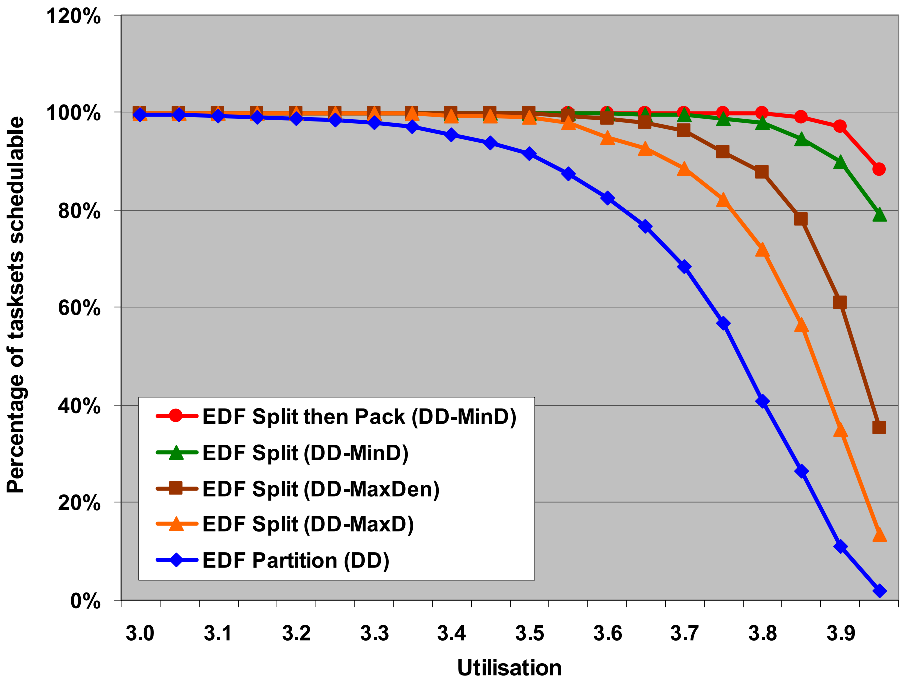


Fig. 6 Performance with 4 processors, 8 tasks and $D = T$

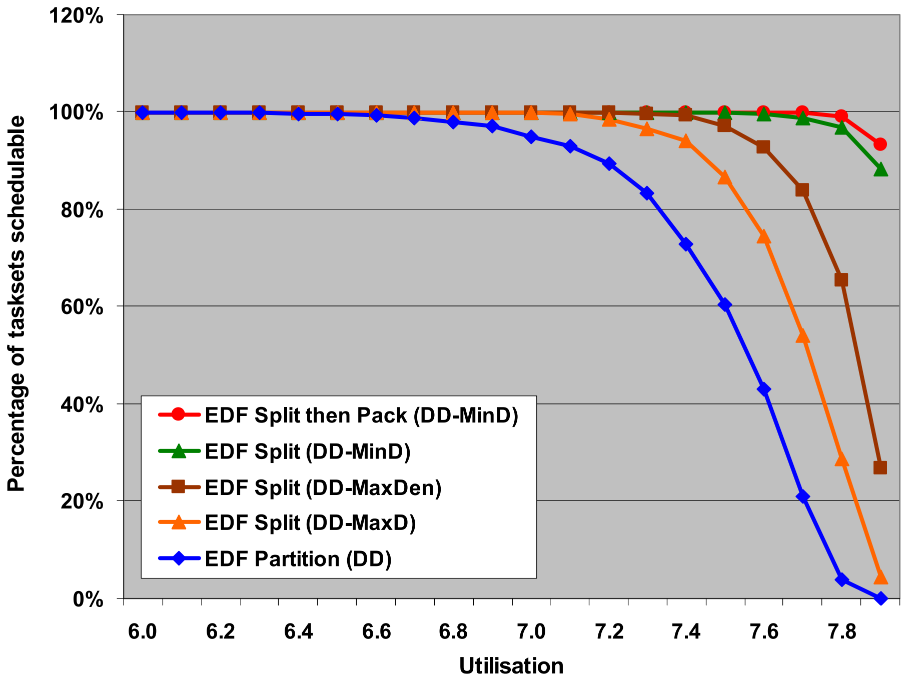


Fig. 7 Performance with 8 processors, 16 tasks and $D = T$

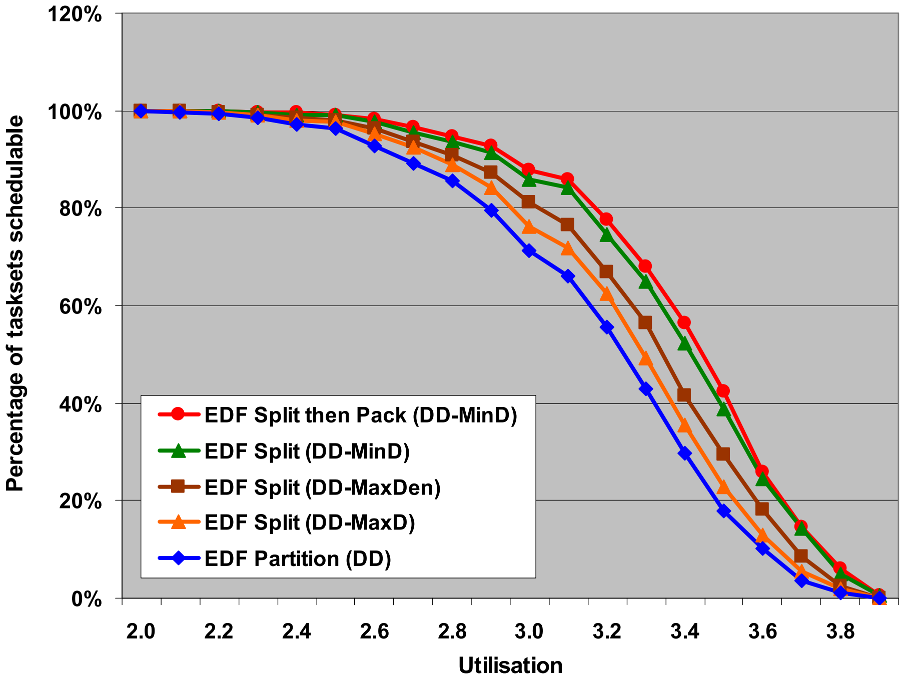


Fig. 8 Performance with 4 processors, 8 tasks and $D \leq T$

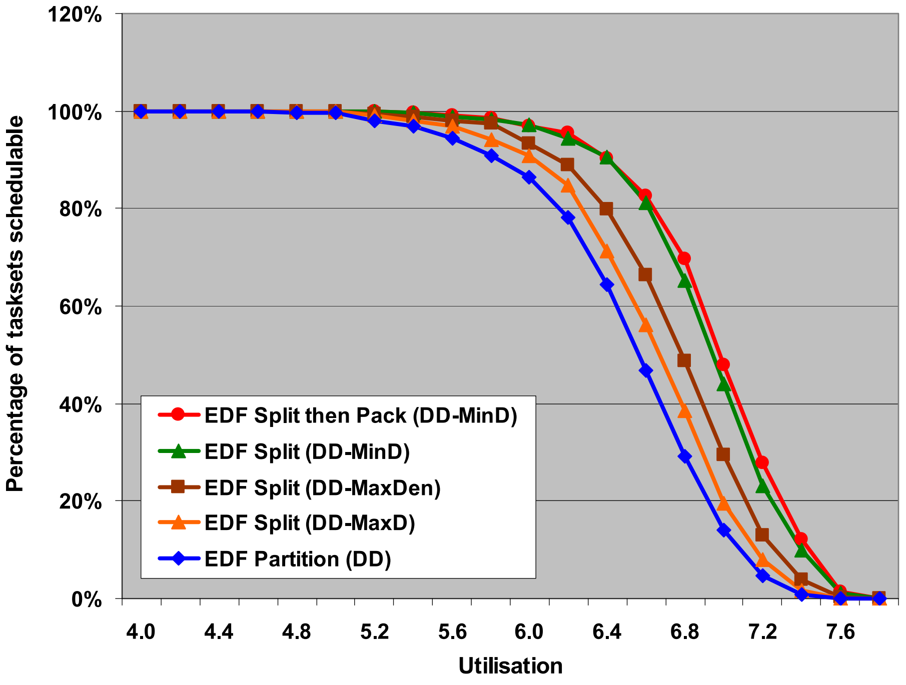


Fig. 9 Performance with 8 processors, 16 tasks and $D \leq T$

pre-selection (“EDF Split then Pack(DD-MinD)”), 50% of the generated tasksets did not become unschedulable until a total utilisation of $0.86m$. In the 8 processor case, the equivalent figures are $0.825m$ for the baseline algorithm, and $0.875m$ for the pre-selection task-splitting approach. This equates to an improvement of 6% in the usable processor capacity.

5 Implementation considerations

One of the motivations for the $C = D$ scheme is that it is of practical utility. The previous section has shown how it compares favourable with fully partitioned approaches. In this section we consider how it might be implemented on a real multi-core platform where overheads and *ease of use* are important issues. Although full experimental evaluation has yet to be carried out (it forms part of future work) it is possible to consider the programmability aspects of the scheme. Two implementation technologies are evaluated, first a concurrent programming language (Ada 2012), and second the Linux operating system.

5.1 Ada approach

First a representation of the multiprocessor platform is required. In Ada a simple integer type is used to represent the range of CPUs (`CPU_Range`); the following package is defined:

```
package System.Multiprocessors is
  pragma Preelaborate(Multiprocessors);
  type CPU_Range is range 0 .. <implementation-defined>;
  Not_A_Specific_CPU : constant CPU_Range := 0;
  subtype CPU is CPU_Range range 1 .. CPU_Range'Last;
  function Number_Of_CPUs return CPU;
end System.Multiprocessors;
%\end{alltt}
```

A task that is partitioned to execute on just a single processor (say, CPU 2) can be have its CPU fixed at the time declaration:

```
task Normal is
  pragma Relative_Deadline(Milliseconds(50));
  pragma CPU(2);
end Normal.
```

This task has also been given a static relative deadline of 50 ms. The Ada program can be instructed to use EDF scheduling (on all processors) by use of the following pragma:

```
pragma Task_Dispatching_Policy(EDF_Across_Priorities);
```

EDF scheduling is supported by the following package:

```
package Ada.Dispatching.EDF is
  subtype Deadline is Real_Time.Time;
  Default_Deadline : constant Deadline :=
     Real_Time.Time_Last;
  procedure Set_Deadline(D : Deadline;
     T : Task_Identification.Task_ID :=
     Task_Identification.Current_Task);
  procedure Delay_Until_And_Set_Deadline(
     Delay_Until_Time : Real_Time.Time;
     TS : Real_Time.Time_Span);
  function Get_Deadline(T : ask_Identification.Task_ID :=
     Task_Identification.Current_Task) return Deadline;
end Ada.Dispatching.EDF;
```

and hence the code pattern for this normal task is as follows:

```
task body Normal is
  Next : Time;
  Period : Time_Span := Milliseconds(50);
    -- equal to full deadline
begin
  Next := Ada.Real_Time.Clock;
  loop
    -- code of application
    Next := Next + Period;
    Delay_Until_And_Set_Deadline(Next, Next+Period);
    -- this suspends the task until time of next release
    -- and fixes the new absolute deadline at that time
    -- to be release time plus 50 ms
  end loop;
end Normal;
```

For multiprocessor scheduling a further language-defined package is required to represent the collection of CPUs on which a task can execute. Part of this packages definition is

```
with Ada.Real_Time;
with System; use System;
package Multiprocessors.Dispatching_Domains is
   Dispatching_Domain_Error : exception;
   type Dispatching_Domain (<>) is limited private;
   System_Dispatching_Domain : constant Dispatching_Domain;
     ...
   procedure Set_CPU(P : CPU_Range;
      T : Task_Id := Current_Task);
   function Get_CPU(T : Task_Id := Current_Task)
      return CPU_Range;
   procedure Delay_Until_And_Set_CPU(Delay_Until_Time
      : Ada.Real_Time.Time; P : CPU_Range);
end Multiprocessors.Dispatching_Domains;
```

The type `Dispatching_Domain` represents a series of processors on which a task may execute. Each processor is contained within exactly one `Dispatching_Domain`. To implement the $C = D$ scheme only a single `Dispatching_Domain` is employed and all non-split tasks are statically allocated to exactly one CPU. Any split task however must migrate at time D after being released. To effect this migration a 'timer' is defined. In Ada a timer is code (in the form of a handler procedure) that is executed when the system clock reaches a specific time. In the following example the handler is programmed to extend the deadline of the task (by the amount `Extra`) and to move the task to CPU 1. In this code `Client` is an abstract pointer to the task; it is a task ID.

```
  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(1,Client);
    -- moves task to processor 1
  end Handler;
```

To illustrate the code required to program a split task, consider an example of a periodic task with period 20 ms, an initial phase with deadline 5 ms (i.e. $C = D = 5$), and hence an `Extra` value (as used above) of 15 ms. The task (called `Split`), which executes first on CPU 0, has a simple form:

```
task Split is
  pragma Relative_Deadline(Milliseconds(20));
end Split.

task body Split is
  Id : Task_ID := Current_Task;
  Next : Time;
  Period : Time_Span := Milliseconds(20);
  First_Deadline : Time_Span := Milliseconds(5);
begin
  Set_CPU(0);
  Next := Ada.Real_Time.Clock;
  loop
    Switch.Set_Handler(First_Phase,Switcher.Handler'Access);
    -- set up handler as defined above

    -- code of application

    Next := Next + Period;
    Set_CPU(0); -- return task to initial CPU
    Delay_Until_And_Set_Deadline(Next, Next+First_Deadline);
  end loop;
end Split;
```

Further details of this example are to be found elsewhere (Burns and Wellings 2010)—where the task switching algorithm is used to illustrate the usefulness of some new language features for Ada.

5.2 Linux approach

To obtain a similar level of programmability when implementing on the Linux OS it must be possible to get the necessary behaviour from the use of only user level

APIs. Here we step through the required functionality using the IRMOS implementation (Checconi et al. 2009)[4] which implements an EDF scheduling class for the Linux kernel (Faggioli et al. 2009). Note the basic approach of Linux on a multicore platform is a partitioned one. Each processor has its own run queue, and a thread can only reside on a single queue. To aid more global schemes the Linux kernel supports PUSH and PULL primitives to move a thread from one run queue to another. At the user level these primitives can be accessed by altering a thread's execution *mask*.

First, the basic $C = D$ scheme requires support for partitioned multiprocessor scheduling. A thread (with thread id *tid*) would be fixed to processor 0, say, by manipulating its mask:

```
cpu_set_t mask;  // declare a mask

CPU_ZERO(&mask);  // set~all entries to zero
CPU_SET(0, &mask);    // set pin (bit) on CPU 0 only
sched_setaffinity(tid, sizeof(mask), &mask);
```

Assume that the thread to be split is a periodic thread that is released by the action of an absolute timer (TIMER_ABSTIME) using an accurate clock (CLOCK_MONOTONIC with nanosecond precision). Another timer is then used (with the same clock) to fire when the migration should occur. The easiest way to react to this timer is to define another thread that wakes up (using `clock_nanosleep()`) at 'migration time'. This high priority 'migrator' thread then changes the deadline and the affinity of the 'client' thread (identified by `tid`). Assuming the thread should move to cpu1, first an appropriate mask for that processor is declared:

```
cpu_set_t mask1;  // declare a mask

CPU_ZERO(&mask1);  // set~all entries to zero
CPU_SET(1, &mask1);    // set pin on CPU 1
```

the code to be executed by the migrator thread is simply:

```
sched_setaffinity(tid, sizeof(mask1), &mask1);
```

The result of the `setaffinity()` call is that the thread disappears from the run queue of CPU 0, and it appears on the run queue of CPU 1.

In the IRMOS implementation a thread has a budget and a deadline. If the budget is exhausted the thread is postponed. In the $C = D$ scheme it is not necessary to monitor execution times, and hence the budget values are not actively used. The migrating task is given a reservation on both cpus. A reservation being a pair of values: (C, D)—computation time and relative deadline. Assume it has a $C = D$ value of 4 ms and an overall deadline of 30 ms (and a total computed WCET of 10 ms, including runtime overheads—see following discussion). Its reservation on cpu0 will be (5, 4) and on cpu1 it will be (6, 26). The value 5 is chosen to be higher than the deadline value of 4 (so its budget is never exhausted—note kernel-level admission control must be disabled to enable this to be done). As the thread moves cpus its reservation on cpu0 goes idle (ready to be reset when the task returns to cpu0) whilst its reservation on

[4]The IRMOS scheduler code is available from: http://sourceforge.net/aquosa/linux-irmos.

cpu1 becomes active. The reservation on cpu1 is also used by the migrator thread when it is released.

At the end of the thread's execution (for this job) the thread will sleep awaiting the timer signal that will release the migrator thread which will first reset its client thread's affinity to cpu0 and then release the thread for its next job's execution.

5.3 Overheads

Any real implementation must adapt the 'theoretical' scheduling analysis equations to include parameters for the overheads that inevitable arise with whatever operating system the multi-tasking application is executing on. For single processor analysis this is achieved by adding extra execution time to the tasks (to capture context switch times) and by adding extra (high priority) tasks to represent the overheads of interrupts and OS functions such as delay queue manipulation (Burns and Wellings 2009). For multiprocessor systems there is a further source of overhead to consider, that caused by migrating a task from one processor to another.

The cost of this migration is platform dependent. However, if there is cache coherence between the processors then there are examples in which the cost of migration is of a similar order to that of the normal context switch (Bastoni et al. 2010). These findings, by Bastoni, Brandenburg and Anderson, are based on real measurements of an extensive number of experiments on actual multiprocessor hardware (i.e. not simulations).

Any split task must therefore have two extra 'context switch times' added to its execution time. The first phase of the task cannot have its execution time extended (without jeopardising schedulability), and hence its final phase requirement must be extended—this was illustrated by the first example in Sect. 2.4.

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 than the extra overheads introduced into the system by the required task migration.

In a real application tasks will often take less computation time than their worst-case. In some situations a job may actually complete before its migration time, in which case migration would not occur and the task would remain on its initial processor.

6 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 elapse 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. For example, the average utilisation of the 'full' processors for experiments with 8 tasks with a total utilisation of 4 rose from less than 0.89 for strict partitioning to over 0.99 for a $C = D$ strategies. 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 effectiveness 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 and next 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) and EU funded NoE ArtistDesign. The authors would like to thank Tommaso Cucinotta of ReTiS Lab, Scuola Superiore Sant'Anna, Pisa for useful input on the issues involved in the implementation of the $C = D$ scheme on multicore Linux. Also thanks for useful discussion with Pierre Courbin, Laurent George and Shinpei Kato.

References

Andersson B, Tovar E (2006) Multiprocessor scheduling with few preemptions. In: RTCSA, pp 322–334

Andersson B, Bletsas K, Baruah SK (2008) Scheduling arbitrary-deadline sporadic task systems on multiprocessors. In: IEEE real-time systems symposium, pp 385–394

Balbastre P, Ripoll I, Crespo A (2006) Optimal deadline assignment for periodic real-time tasks in dynamic priority systems. In: Euromicro conference on real-time systems (ECRTS)

Baruah SK, Burns A (2006) Sustainable schedulability analysis. In: Proceedings of IEEE real-time systems symposium (RTSS), pp 159–168

Baruah SK, Mok AK, Rosier LE (1990) Preemptive scheduling of hard real-time sporadic tasks on one processor. In: Proceedings of IEEE real-time systems symposium (RTSS), pp 182–190

Baruah SK, Howell RR, Rosier LE (1993) Feasibility problems for recurring tasks on one processor. Theor Comput Sci 118:3–20

Bastoni A, Brandenburg B, Anderson J (2010) Cache-related preemption and migration delays: empirical approximation and impact on schedulability. In: Proceedings of sixth international workshop on operating systems platforms for embedded real-time applications, pp 33–44

Bini E, Buttazzo GC (2005) Measuring the performance of schedulability tests. Real-Time Syst 30(1–2):129–154

Bletsas K, Andersson B (2009) Notional processors: an approach for multiprocessor scheduling. In: IEEE real-time and embedded technology and applications symposium, pp 3–12

Burns A, Wellings AJ (2009) Real-time systems and programming languages, 4th edn. Addison-Wesley/Longman, Reading/Harlow

Burns A, Wellings AJ (2010) Dispatching domains for multiprocessor platforms and their representation in Ada. In: Real J, Vardanega T (eds) Proceedings of reliable software technologies—Ada-Europe 2010. LNCS, vol 6106. Springer, Berlin, pp 41–53

Burns A, Davis RI, Wang P, Zhang F (2010) Partitioned edf scheduling for multiprocessors using a $C = D$ scheme. In: Proceedings of 18th international conference on real-time and network systems (RTNS), pp 169–178

Checconi F, Cucinotta T, Faggioli D, Lipari G. (2009) Hierarchical multiprocessor cpu reservations for the Linux kernel. In: Proceedings of 5th international workshop on operating systems platforms for embedded real-time applications (OSPERT, 2009)

Davis RI, Burns A (2009) Priority assignment for global fixed priority pre-emptive scheduling in multiprocessor real-time systems. In: Proceedings of IEEE real-time systems symposium (RTSS), pp 398–409

Davis R, Burns A (2010) Improved priority assignment for global fixed priority pre-emptive scheduling in multiprocessor real-time systems. Real-Time Syst J 1–40

Davis RI, Burns A (2011) A survey of hard real-time scheduling algorithms for multiprocessor systems. ACM Comput Surv (accepted)

Faggioli D, Checconi F, Trimarchi M, Scordino C (2009) An edf scheduling class for the Linux kernel. In: Proceedings of 11th real-time Linux workshop (RTLWS)

Guan N, Stigge M, Yi W, Yu G (2010) Fixed priority multiprocessor scheduling with Liu and Layland utilization bound. In: Proceedings of the IEEE real-time technology and applications symposium (RTAS), April 2010. IEEE Press, New York,

Hoang H, Buttazzo GC, Jonsson M, Karlsson S (2006) Computing the minimum EDF feasible deadline in periodic systems. In: RTCSA, pp 125–134

Johnson DS (1974) Near-optimal bin-packing algorithms. PhD thesis, Department of Mathematics, MIT

Kato S, Yamasaki N (2007) Real-time scheduling with task splitting on multiprocessors. In: RTCSA, pp 441–450

Kato S, Yamasaki N (2008a) Portioned static-priority scheduling on multiprocessors. In: IPDPS, pp 1–12

Kato S, Yamasaki N (2008b) Portioned EDF-based scheduling on multiprocessors. In: EMSOFT, pp 139–148

Kato S, Yamasaki N (2009) Semi-partitioned fixed-priority scheduling on multiprocessors. In: IEEE real-time and embedded technology and applications symposium, pp 23–32

Kato S, Yamasaki N, Ishikawa Y (2009) Semi-partitioned scheduling of sporadic task systems on multiprocessors. In: ECRTS'09: proceedings of the 2009 21st euromicro conference on real-time systems, pp 249–258

Lakshmanan K, Rajkumar R, Lehoczky J (2009) Partitioned fixed-priority preemptive scheduling for multi-core processors. In: ECRTS'09: proceedings of the 2009 21st euromicro conference on real-time systems, pp 239–248

Liu CL, Layland JW (1973) Scheduling algorithms for multiprogramming in a hard real-time environment. JACM 20(1):46–61

Lopez JM, Garcia M, Diaz JL, Garcia DF (2000) Worst-case utilization bound for EDF scheduling on real-time multiprocessor systems. In: Proceedings of ECRTS, pp 25–33

Ripoll I, Crespo A, Mok AK (1996) Improvement in feasibilty testing for real-time tasks. Real-Time Syst 11(1):19–39

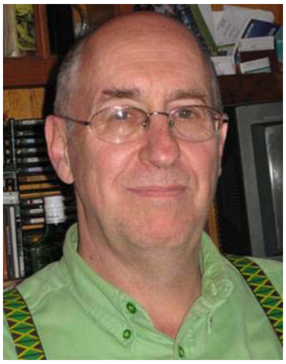
Spuri M (1996) Analysis of deadline schedule real-time systems. Technical Report 2772, INRIA, France

Yao AC (1980) New algorithms for bin packing. Journal of the ACM 27(2)

Zhang F, Burns A (2008a) Schedulability analysis for real-time systems with EDF scheduling. Technical Report YCS 426, University of York

Zhang F, Burns A (2008b) Schedulability analysis for real-time systems with EDF scheduling. IEEE Trans Comput 58(9):1250–1258

Zhang F, Burns A, Baruah S (2010) Sensitivity analysis for EDF scheduled arbitrary deadline real-time systems. In: Proceedings of 16th IEEE conference on embedded and real-time computing systems and applications (RTCSA), pp 61–70



A. Burns co-leads the Real-Time Systems Research Group at the University of York. His research interests cover a number of aspects of real-time systems including the assessment of languages for use in the real-time domain, distributed operating systems, the formal specification of scheduling algorithms and implementation strategies, and the design of dependable user interfaces to real-time applications. Professor Burns is a member of the ARTIST Network of Excellence.

R.I. Davis is a Senior Research Fellow in the Real-Time Systems Research Group at the University of York, and a Director of Rapita Systems Ltd. He received his DPhil in Computer Science from the University of York in 1995. Since then he has founded three start-up companies, all of which have succeeded in transferring real-time systems research into commercial products. Robert's research interests include scheduling algorithms and schedulability analysis for real-time systems.

P. Wang is currently studying for an MPhil in Advanced Computer Science at Cambridge University, UK, specializing in theoretical computer science. His main interests are category theory and its applications, especially verifying and reasoning about concurrent computations. He received a first class degree from the University of York in 2009. His tutor was Professor Alan Burns.

F. Zhang received the Ph.D. degree in Computer Science from the University of York (UK) in 2009. He is currently an associate professor in the College of Computer and Information Science, Southwest University, China. His main research interests are in the areas of scheduling analysis of real-time and embedded systems.