The Case For Non-preemptive, Deadline-driven Scheduling In Real-time Embedded Systems

Michael Short[1]

***Abstract*—Non-preemptive schedulers remain a very popular choice for practitioners of resource constrained real-time embedded systems. This paper is concerned with the non-preemptive version of the Earliest Deadline First algorithm (npEDF). Although several key results indicate that npEDF should be considered a viable choice for use in resource-constrained real-time systems, these systems have traditionally been implemented using static, table-driven approaches such as the 'cyclic executive'. This is perhaps due to several popular misconceptions regarding the basic operation, optimality and robustness of the npEDF algorithm, leading to a general lack of coverage in the wider academic community. This paper will attempt to redress this balance by showing that the supposed 'problems' attributed to npEDF either simply do not hold, or can be easily overcome by adopting an appropriate implementation. Examples are given to highlight the fact that npEDF generally outperforms other non-preemptive software architectures when scheduling periodic and sporadic tasks. The paper concludes with the observation that npEDF should in fact be considered as the algorithm of choice for such systems.**

***Index Terms*— Deadline Scheduling, Embedded Systems, Non-Preemptive Scheduling, Real-Time Systems.**

I. INTRODUCTION

This paper is concerned with the non-preemptive scheduling of recurring (periodic / sporadic) task models, with applications to resource-constrained, single-processor real-time and embedded systems. In particular, the paper is concerned with scheduler architectures for use with such systems, consisting of a small amount of hardware (typically a timer / interrupt controller) and software. In this context, the two main aspects (requirements) of a scheduler can be stated as follows:

Task activation: this is the process of deciding at which points in time a task becomes ready for execution (is activated). Periodic tasks are normally activated via a timer; event driven (sporadic) tasks can be either directly activated by interrupts or by polling an interrupt status flag.

Task dispatching: Real-time systems are required to perform specific processing in a timely fashion; when multiple tasks are simultaneously active, then some form of scheduling algorithm is normally required to process the events in an appropriate order.

These two main aspects of scheduling are illustrated in Fig. 1. The performance of scheduling algorithms and techniques is an area worthy of study; the seminal paper of Liu & Layland [1], published in 1973, spawned a multitude of research and a significant body of results can now be found in the literature. Liu & Layland were the first to discuss deadline-driven scheduling techniques.

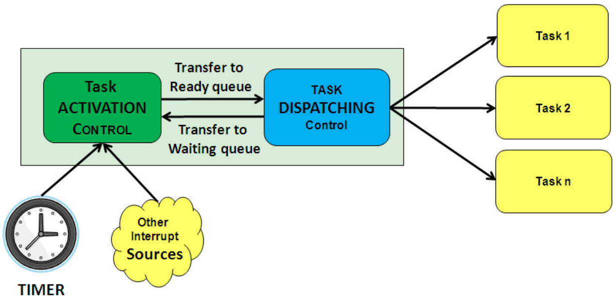


Fig. 1: Aspects of real-time embedded scheduling.

It is known that when task preemption is allowed, this technique – also known as Earliest Deadline First (EDF) - allows the full utilization of the CPU, and is optimal on a single processor under a wide variety of different operating constraints ([1][2][3]). However, for developers of systems with severe resource constraints, preemptive scheduling techniques may not be viable; the study of non-preemptive alternatives is justified for the following (non-exhaustive) list of reasons [4][5][6][7]:

- Non-preemptive scheduling algorithms are easier to implement than their preemptive counterparts, and can exhibit dramatically lower runtime overheads;
- Non-preemptive scheduling naturally guarantees exclusive access to resources, eliminating the need for complex resource access protocols;
- Preemptive systems require individual task stacks whereas non-preemptive tasks can share a common stack, leading to vastly reduced memory requirements;
- Exploratory studies seem to indicate that preemptive systems are more susceptible to transient errors such as electromagnetic disturbances than their non-preemptive counterparts.

Despite these advantages, non-preemptive scheduling is also known to have several associated problems; task response times will be (in general) longer, event-driven (sporadic) task executions are not as well supported (if at all), and when preemption is not allowed, in general scheduling problems become NP-Complete or NP-Hard [8]. This paper is concerned with systems implementing the non-preemptive version of EDF (npEDF). The main motivating factors for the current work are as follows. Although the treatment of npEDF has been (comparatively) small in the literature,

Manuscript received 1st February, 2010, accepted 21st March 2010.

[1] M. Short is with the Electronics & Control Group, Teesside University, Middlesbrough, UK (phone: +44(0)16423422528; e-mail: m.short@tees.ac.uk).

several key results exist that indicate npEDF can overcome most (perhaps not all) of the problems associated with non-preemption; as such it should be considered as a viable choice for use in resource-constrained real-time and embedded systems. However, such systems have traditionally been implemented using static, table-driven approaches such as the 'cyclic executive' and its variants (see, for example, [4][9][10][11]). This is perhaps due to several popular misconceptions[1] with respect to the basic operation, implementation complexity, optimality and robustness of the npEDF algorithm, leading to a general lack of coverage in the wider academic (especially engineering) community.

This paper will attempt to redress this balance by arguing the case for npEDF, and showing that the supposed 'problems' commonly attributed to it either simply do not hold, or can easily be overcome by adopting an appropriate implementation and by applying simple off-line analysis techniques. The paper is organized as follows. Section II considers exactly why npEDF seems to be 'missing' from most major texts on real-time systems. Section III presents the assumed task model, gives a basic description of npEDF and identifies a list of its common criticisms. Section IV then addresses each of these criticisms in turn, and establishes whether or not the claims actually hold; it is shown that in each case, the claims are baseless. Section V concludes the paper, with the observation that npEDF should be considered as the algorithm of choice for scheduling resource constrained real-time embedded systems.

II. NPEDF: A MISSING ALGORITHM

In most of the major texts in the field of real-time systems, npEDF does not get more than a passing mention. For example, analysis of non-preemptive scheduling is typically restricted to the use of 'cyclic executives' or 'timeline schedulers'. In almost all cases, after problems have been identified with such scheduling models, attention is then focused directly on Priority-Driven Preemptive (PDP) approaches as a 'cure for all ills'. For example, Buttazzo [5] discusses timeline scheduling in C4 of his (generally) well-respected book on hard real-time computing systems, concluding with a list of problems associated with this type of scheduling. On p78 - immediately before moving onto descriptions of PDP algorithms – it is stated that:

> *"The problems outlined above of timeline scheduling can be solved by using priority-based [preemptive] algorithms."*

Liu takes a similar approach in what is perhaps the most widely-acclaimed book in this area (Real-Time Systems) [12]. Cyclic scheduling is discussed in C5 of her book, ending with a list of associated problems on p122. In each case, it is stated that a PDP system can overcome the problem. This type of argument is by no means limited to reference texts. Burns et al. [9] describe (in-depth) some techniques that can be used for generating feasible cyclic or timeline schedules, followed by a discussion of the problems associated with this type of scheduling, directly followed by a final section (p160) discussing:

> *"Priority [-based preemptive] scheduling as an alternative to cyclic scheduling"*

Whilst it is clearly untrue to say these statements are false, as stated above PDP scheduling is not without its own problems; the next Section will examine the basics of npEDF, and examine why it seems to have been overlooked.

III. TASK MODEL AND NPEDF PRELIMINARIES

A. Recurring Task Model

This paper is concerned with the implementation of recurring / repeated computations on a single processor, such as those that may be required in signal processing and control applications. Such a system may be represented by a set τ of n tasks, where each task $t_i \in \tau$ is represented by a tuple:

$$t_i = \left(p_i, c_i, d_i\right) \tag{1}$$

In which p_i is the task period (minimum inter-arrival time), c_i is the (worst-case) computation requirement of the task and d_i is the task (relative) deadline. A similar model was introduced in this context by Liu & Layland [1] and has since been widely adopted – see, for example, [2-7]. Note that it can be assumed w.l.o.g. that time is discrete, and all task parameters can be assumed to be integer [13]. Attention is primarily restricted in this paper to implicit deadline tasks, i.e. those in which $d_i = p_i$; such tasks are the most widely discussed in the literature (and employed in practice). Such a task will simply be described by two parameters p_i and c_i. Note that a periodic task may additionally be described by an addition parameter, its initial release time (or relative phasing) r_i.

B. npEDF Basic Operation

The npEDF algorithm may be described, in simple terms, as follows:

1. When selecting a task for execution, the task with the earliest deadline is selected first (and then run to completion).
2. Ties between tasks with identical deadlines are broken by selecting the task with the lowest index.
3. Unless the processor is idle, scheduling decisions are only made at task boundaries.
4. When the scheduler is idle, the first task to be invoked is immediately executed (if multiple tasks are simultaneously invoked, the task with the earliest deadline is selected).

This simple (but deceptively effective) algorithm may be implemented using only a single hardware timer. Ideally this timer will be free-running, with a single interrupt-on-match register if the system is required to enter idle or power-down mode when no further tasks are pending. Clearly the algorithm differs from the static table-driven approaches in that the schedule is effectively produced on-line, and there is therefore no concept of a fixed time 'frame' or 'tick'; an

---

[1] The key results for npEDF - and their implications - are comparatively more difficult to interpret that for other types of scheduling; for example, many previous works assume the reader possesses an in-depth understanding of formal topics in computer science, such as computational complexity.

example (feasible) schedule for the set of synchronous tasks $\tau = [(4,1),(6,2),(12,3)]$ is shown in Fig. 2 below.

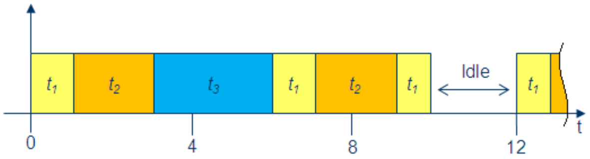


Fig. 2: npEDF schedule.

C. *npEDF Common Criticisms*

As mentioned in the introduction, generally due to misconceptions (or misinterpretations) of its operation and use, npEDF is generally seen to be too problematic for use in real systems. The main criticisms that can be found in the literature are listed below:

1. npEDF is not an optimal non-preemptive scheduling algorithm. Optimal in this sense refers to its ability to build a valid (feasible) schedule, if such a schedule exists;
2. npEDF is difficult to analyze, and no efficient feasibility test exists;
3. npEDF is not 'robust' to changes in the task set parameters; in particular, reductions in the run-time execution requirement of one (or more) tasks can lead to deadline misses in an otherwise feasible task set;
4. Timer rollover can lead to anomalies and deadline misses in an otherwise feasible task set;
5. The use of npEDF leads to increased overheads (and power consumption) compared to other non-preemptive scheduling techniques.

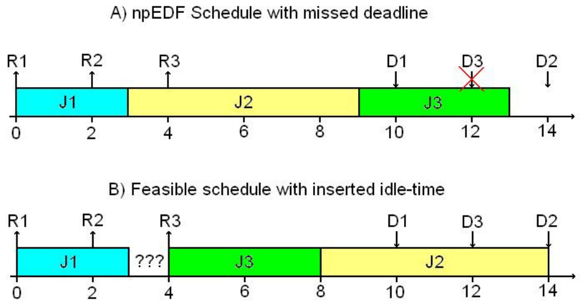
Please note that this list of criticisms is specific to npEDF, and therefore does not include the so-called 'long-task' problem which is endemic to all non-preemptive schedulers. This specific problem arises when one or more tasks have a deadline that is less than the execution time of another task. In this situation, effective solutions are known to include code-refactoring at the task level, employing state-machines, or alternately adopting the use of hybrid designs [4][8][14]. Such solution techniques easily generalize to npEDF, and are not discussed in any further depth in this paper.

IV. npEDF CRITICISMS: ARE THEY JUSTIFIED?

If all of the criticisms given in the previous Section are based in fact, then npEDF does not seem a wise choice for system implementation; in fact the contrary would be true. This Section will examine each point in greater detail, to investigate if, in fact, each specific claim actually holds.

A. *npEDF is not Optimal.*

As mentioned, optimal in this sense refers to the ability of a scheduling algorithm to build a valid (feasible) schedule for an arbitrary set of tasks, if such a feasible schedule exists. Each (and every) proof that npEDF is sub-optimal relies on a counter-example of the form shown in Fig. 3 (taken from Liu [12] – a similar example appears in Buttazzo [5]). It can be seen that despite the existence of a feasible schedule, obtained via the use of a scheduler which inserts idle-time between $t = 3$ and $t = 4$ (indicated by the question marks in the figure), the schedule produced by npEDF misses a deadline at $t = 12$.


A) npEDF Schedule with missed deadline

B) Feasible schedule with inserted idle-time

Fig. 3: npEDF misses a deadline, yet a feasible schedule exists.

Now, since the use of inserted idle-time can clearly have a beneficial effect with respect to meeting deadlines, this clearly begs the question – how complex is a scheduler that uses inserted idle time – will such a scheduler be of practical use for a real system? The answer, unfortunately, is a resounding no - the following two results were formally shown by Howell & Venkatro [15]:

- There cannot be an optimal on-line scheduling algorithm using inserted idle-time for sporadic tasks; only non-idling scheduling strategies can be optimal;
- An on-line scheduling strategy that makes use of inserted idle-time to schedule periodic tasks cannot be efficiently implemented unless P = NP.

It can thus be seen that inserted idle-time is not beneficial when scheduling sporadic tasks, and if efficiency is taken into account, then attention must be restricted to non-idling strategies when scheduling periodic tasks. Efficiency in this sense refers to the amount of time taken by the scheduler to make scheduling decisions; only schedulers that take time proportional to some polynomial in the task set parameters can be considered efficient (a scheduler which takes 50 years to decide the optimal strategy for the next 10 ms is not much practical use). What is known about the non-idling scheduling strategies? These include, for example, npEDF, TTC scheduling [4][14] and non-preemptive Rate Monotonic (npRM) scheduling [16]. npEDF is known to be optimal among this class of algorithms for scheduling recurring tasks; results in this area were known as early as 1955 [17]. The proof was demonstrated in the real-time context by Jeffay et al. [6] for the implicit deadline case, and extended by George et al. [18] to the constrained deadline case. Thus, the overall claim status: *npEDF is sub-optimal for periodic tasks if and only if P = NP, and is optimal for sporadic tasks regardless of the equivalence (or otherwise) of these complexity classes.*

B. *No Efficient Feasibility Test Exists for npEDF.*

Consider again the example shown in Fig. 3, in which the npEDF algorithm misses a deadline. Why is the deadline missed? At $t = 3$, only J2 is active and, since the scheduler is non-idling, it immediately begins execution of this task. Subsequently at $t = 4$, J3 is released and has an earlier deadline – but it is blocked (due to non-preemption) until J2 has run to completion at $t = 9$. This is known as a 'priority inversion' as the scheduler cannot change its mind, once committed. This is highlighted further in Fig. 4 below.

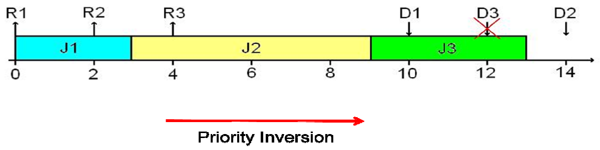


Fig. 4: npEDF priority inversion.

How complex is it to analyze the schedule that will be produced by npEDF for a given (arbitrary) set of tasks, to predict the effect of such priority inversions? This turns out to be not as hard as it may first appear - for the implicit deadline case, the following result was formally shown by Jeffay et al. [6]: A periodic (sporadic) set of *n* tasks, indexed in order of increasing period, is feasible (for any set of release times) under npEDF if and only if the following conditions are true:

$$\sum_{i\in\tau}\frac{c_i}{p_i}\le 1.0 \tag{2}$$

$$\forall i, 1<i\le n; \forall t, p_1<t<p_i;$$

$$c_i+\sum_{j<i}\left\lfloor\frac{t-1}{p_j}\right\rfloor\cdot c_j\le t \tag{3}$$

Informally, the condition of Equation (2) states that the processor should not be overloaded, and condition (3) expresses the worst-case penalty for non-preemption under npEDF, which is illustrated in Fig. 5. That is, each task (apart from the task with the smallest period) is assumed to lead a worst-case priority inversion, and each deadline lying in the interval (p_1, p_i) is checked. If these deadlines are met under these worst-case priority inversions, the task set is feasible.

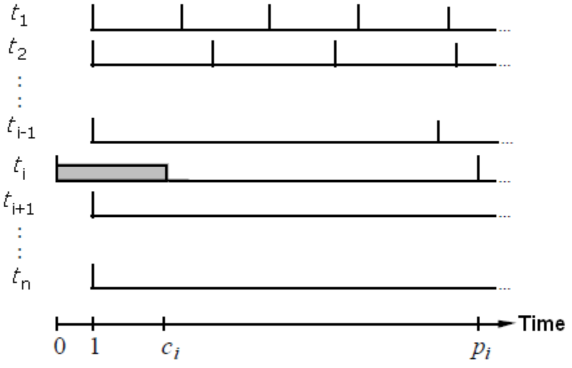


Fig. 5: npEDF critical instants: worst case blocking induced by task *i*.

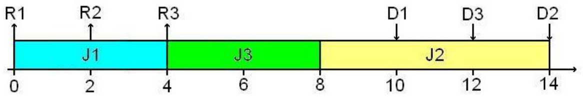
It should be noted that the time complexity of an algorithm to decide (2) and (3) is pseudo-polynomial and hence highly efficient, taking time proportional to the number of tasks multiplied by the largest period or $O(np_{max})$ – the non-preemptive scheduling problem, in this formulation, turns out to be only *weakly* coNP-Complete. In the case when one or more task has a constrained deadline, George et al. developed similar feasibility conditions, with the same complexity [18]. When compared to feasibility tests for other non-preemptive scheduling disciplines, this is significantly better. For example, it is known that deciding if a set of periodic process can be scheduled by a cyclic executive or timeline scheduler is *strongly* NP-Hard [8][9]; it is also known that deciding if a set of periodic process can be scheduled by a TTC scheduler is *strongly* coNP-Hard[2] [19]. Note that *strong* and *weak* complexity results have a precise technical meaning; specifically, amongst other things the former rules out the prospect of a pseudo-polynomial time algorithm unless P = NP, whereas the latter does not.

Thus, although a very efficient algorithm may be formulated to exactly test for Equations (2) and (3), it is thought that no exact algorithm can ever be designed to efficiently test feasibility for these alternate scheduling policies. Please note that for 'liquid' task sets - i.e. those with execution times significantly shorter than their periods – it is known that the 'penalty for non-preemption' – i.e. condition (3) - evaporates, and we are simply left the same feasibility test as the preemptive case, i.e. condition (2). Overall claim status: *npEDF admits an efficient feasibility test for periodic (sporadic) tasks that ensures even worst-case priority inversions do not lead to deadline misses.*

C. *npEDF is not Robust under reduced system load.*

With respect to this complaint, Jane Liu presents some convincing evidence on p.73 of her book Real-Time Systems [12], and cites the seminal paper by Graham [20] investigating timing anomalies. There are two principal problems here. The paper by Graham deals only with the multiprocessor case; specifically, it investigates the effects of reduced (aperiodic) task execution times on the makespan produced by the LPT heuristic scheduling technique. As do the examples on p.73 of Liu's book, although it is not made explicitly clear. This paper is concerned with single-processor scheduling, and these examples simply do not apply. The only single processor timing anomaly referred to in the Liu text is shown in Fig. 6; at first glance, it seems that a run-time reduction in the execution requirement of job C1 does lead to a deadline miss of J3:

A) npEDF Schedule with C1 = 4



B) npEDF Schedule with C1 = 3

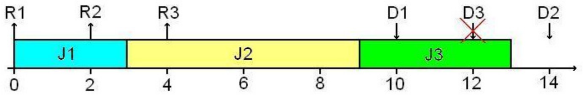


Fig. 6: A Run-time reduction in task execution times leading to deadline misses: a valid example?

However upon closer inspection, this example can be seen to be *almost* identical to the example given in Fig. 3, with the execution of J1 between t = 3and t = 4 effectively serving the same purpose as the inserted idle-time in Fig. 3. In order for this example to hold up, it must logically follow that the schedule must be provably feasible when the tasks

---

[2] In fact, this situation is known to considerably worse than this. The problem is actually known to be NP^{NP}-Complete [19]. Under the assumption that P ≠ NP, this means that the feasibility test requires an exponential number of calls to a decision procedure which is itself strongly coNP-Complete.

have nominal parameters given by A); applying Equations (2) and (3) to these tasks, it can be quickly determined that the task set is *not* deemed to be feasible, since the formulation of Jeffay's feasibility test takes worst-case priority inversion into account. This example is misleading w.r.t. npEDF – since the task set simply fails the basic feasibility test, Liu's argument of *'an otherwise feasible task set'* becomes a non-starter. This again highlights the fact that misconceptions regarding robustness and priority inversions have principally arisen from one simple fact; as shown in the previous Section, the worst case behavior of a task set – its critical instants - under non-preemptive scheduling is *not* the same as under preemptive scheduling. Overall claim status: *If appropriate (off-line) analysis is performed to confirm the feasibility of a task set, this task set will remain feasible under npEDF even under conditions of reduced system load.*

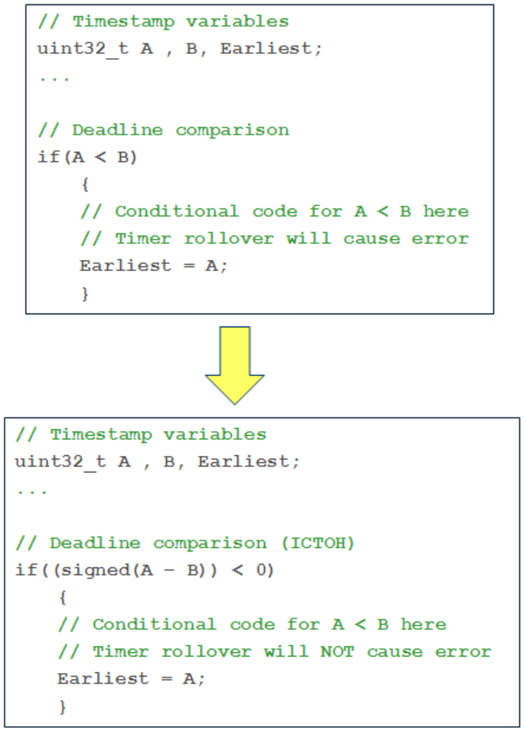
D. *Timer Rollover can Lead to Timing Anomalies.*

With respect to this complaint, this can in fact be shown to hold, but is easily solved. The assumption that time is represented as integer – and in embedded systems, normally with a fixed number of bits (e.g. 16) – eventually leads to timer rollover problems; deadlines will naturally 'wrap around' due to the modular representation of time. Since the normal laws of arithmetic no longer hold, it cannot be guaranteed that $d_i \bmod(2^b) < d_j \bmod(2^b)$ when $d_i < d_j$ and time is represented by b-bit unsigned integers. There are several efficient techniques that may be used to overcome this problem, perhaps the most efficient is as follows. Assuming that the inequality $p_m < 2^b / 2$ holds over a given task set, i.e. the maximum period is less than half the linear life time of the underlying timer, then the rollover problem may be efficiently overcome by using Carlini & Buttazzo's Implicit Circular Timer Overflow Handler (ICTOH) algorithm [21]. The algorithm has a very simple code implementation, and is show as C code in Fig. 7.

The algorithm's operation exploits the fact that the modular distance between any two events (e.g. deadlines or activation times) x and y, encoded by b-bit unsigned integers, may be determined by performing a subtraction modulo 2^b between x and y, with the result interpreted as a signed integer. Overall claim status: *rollover is easily handled by employing algorithms such as ICTOH.*

E. *Use of npEDF Leads to Increased Overheads.*

In order to shed more light on this issue, let us consider the required number of 'scheduling events' over the hyperperiod[3] of a given periodic task set, and also the complexity – the required CPU iterations, as a function of the task parameters – of each such event. Specifically, let us consider these scheduling events as required for task sets under both npEDF and TTC scheduling. TTC scheduling is considered as the baseline case in this respect, as it has previously been argued that a TTC scheduler provides a software architecture with minimal overheads and resource requirements [4][7][14]. Given the definition of npEDF, one scheduling event is required for every task execution. The scheduler enters idle mode when all pending tasks are executed; it is woken by an interrupt set to match the earliest time at which a new task will be invoked. The TTC algorithm is designed to perform a scheduling event at regular intervals, in response to periodic timer interrupts; the period of these interrupts is normally set to be the greatest common divisor of the task periods [4][14].



```
// Timestamp variables
uint32_t A , B, Earliest;
...

// Deadline comparison
if(A < B)
    {
    // Conditional code for A < B here
    // Timer rollover will cause error
    Earliest = A;
    }
```

```
// Timestamp variables
uint32_t A , B, Earliest;
...

// Deadline comparison (ICTOH)
if((signed(A - B)) < 0)
    {
    // Conditional code for A < B here
    // Timer rollover will NOT cause error
    Earliest = A;
    }
```

Fig. 7: Testing the temporal ordering of events, assuming an absolute (top) versus a modular (bottom) representation of time.

Assuming a given set of tasks are synchronous with periods expressed in a minimal form, let $h = lcm(p_1, p_2, \ldots p_n)$. The number of scheduling events over h for the TTC scheduler – SE_{TTC} - is then directly equal to h. The number of scheduling events for the npEDF scheduler over h – SE_{EDF} - is given by:

$$SE_{EDF} = \sum_{i\in\tau} \frac{h}{p_i} \tag{4}$$

Clearly, $SE_{EDF} \leq SE_{TTC}$, and in most cases the former will be significantly smaller. By way of example, the number of scheduling events required for both scheduling disciplines is shown (over the initial portion of the schedule) for the task set $\tau = [(90,5),(100,5)]$ in Fig. 8 (scheduling events are indicated by the presence of up-arrows on the timeline).

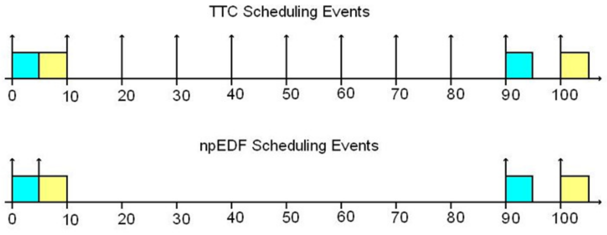


Fig. 8: Density of scheduling events in both TTC and npEDF scheduling.

---

[3] The hyperperiod of a task set is the duration of time taken for the schedule to repeat. For synchronous tasks, this corresponds to the least common multiple of the task periods [1]; for asynchronous tasks the duration is related to the *lcm* but is somewhat longer, see [13] for further details.

As mentioned, also of interest are the time complexities of each scheduling event; this will now be considered, and expressed as a function of the number of tasks, *n*. Given the design of the TTC scheduler, it is clear to observe from its basic design and implementation (see, for example, [4][14]) that its complexity is fixed to be linear in the number of tasks, in other words *O(n)*. However, task management in the npEDF scheduler significantly improves upon this situation; it is known that the algorithm can be implemented with complexity *O(log n)* by employing a data structure known as a heap-of-heaps [22]. Additionally, recent work by the current author has also shown that this can be improved further still; by employing simple data structures known as timing and deadline wheels, npEDF can be implemented with small constant overhead, i.e. independent of the number of tasks and with complexity *O(1)* [23].

To further illustrate this final point, Fig. 9 shows a comparison of the overheads incurred per scheduling event as the number of tasks was increased on a 72-Mhz ARM7-TDMI microcontroller. Overheads execution times were extracted using the technique described in [23]; please note that the horizontal scale is logarithmic. This graph clearly shows the advantages of the npEDF technique, when $n > 8$, the TTC overheads are significantly greater the than npEDF; when $n > 32$, they become an order of magnitude larger. Overall claim status: *With an appropriate implementation, the density of npEDF scheduling events is no worse that (and in most cases significantly better) than competing methods; the CPU overheads incurred at each such event are also significantly lower.*

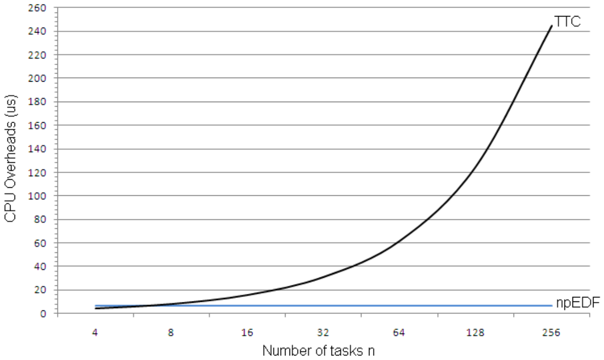


Fig. 9: CPU overheads vs. number of tasks *n*.

V. CONCLUSION

This paper has considered the non-preemptive version of the Earliest Deadline First algorithm. Specifically, it has first considered - and subsequently refuted - many of the supposed ‘problems’ that have been attributed to this type of scheduling technique. Where appropriate, examples and analysis have been given to highlight that not only are many of these claims simply baseless, in fact npEDF outperforms other non-preemptive software architectures – oftentimes significantly so - when scheduling periodic and sporadic tasks. On the merits of these arguments, it the conclusion of the current author that npEDF should actually be considered as the de-facto algorithm of choice for implementing resource-constrained real-time embedded systems.

REFERENCES

[1] J. Liu and J. Layland, “Scheduling algorithms for multiprogramming in a hard real-time environment,” Journal of the ACM, Vol. 20, No. 1, pp. 46-61, 1973.

[2] E. Coffman, Jnr, “Introduction to Deterministic Scheduling Theory”, in Computer and Job-Shop Scheduling Theory, Wiley, New York, 1976.

[3] M.L. Dertouzos, “Control robotics: the procedural control of physical processes”, Information Processing, Vol. 74, 1974.

[4] M. Pont, Patterns for time-triggered embedded systems, ACM Press / Addison-Wesley Education, 2001.

[5] G.C. Buttazzo, Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications”, Spinger-Verlag, New York, 2005.

[6] K. Jeffay, D. Stanat and C. Martel, “On Non-Preemptive Scheduling of Periodic and Sporadic Tasks”, Proc. of the IEEE Real-Time Systems Symposium, 1991.

[7] Short, M., Pont, M.J. and Fang, J, “Exploring the impact of pre-emption on dependability in time-triggered embedded systems: A pilot study”, In: Proceedings of the 20th Euromicro conference on real-time systems (ECRTS 2008), Prague, Czech Republic, pp. 83-91, 2008.

[8] M.R. Garey and D.S. Johnson, Computers and Intractability: A guide to the Theory of NP-Completeness, W.H. Freeman & Co Ltd, April 1979.

[9] Burns, A., Hayes, N. and Richardson. M, “Generating feasible cyclic schedules”, Control Engineering Practice, Vol. 3, No. 2, pp. 151-162, 1994.

[10] Baker, T. P. and Shaw, A. The cyclic executive model and Ada. Real-Time Systems, Vol. 1, No. 1, pp. 7-25, 1989.

[11] Locke, C. D. Software architecture for hard real-time applications: Cyclic executives vs. fixed priority executives. Real-Time Systems, 4(1): 37-52, 1992.

[12] Liu, J.W.S, “Real-Time Systems”, Prentice-Hall, New Jersey, 2000.

[13] S. Baruah, L. Rosier and R. Howell, “Algorithms and Complexity Concerning the Preemptive Scheduling of Periodic, Real-Time Tasks on One Processor”, Real-Time Systems, Vol. 2, No. 4, pp. 301-324, 1991.

[14] Gendy, A.K. and Pont, M.J., “Automatically configuring time-triggered schedulers for use with resource-constrained, single-processor embedded systems”, IEEE Transactions on Industrial Informatics, Vol. 4, No. 1, pp. 37-45, 2008.

[15] Howell, R. and Venkatro, M, “On Non-Preemptive Scheduling of Recurring Tasks Using Inserted Idle Times”, Information and Computation, Vol. 117, 1995.

[16] Park, M, “Non-preemptive Fixed Priority Scheduling of Hard Real-Time Periodic Tasks”, Lecture Notes in Computer Science, Vol. 4990, pp. 881-888, 2007.

[17] J.R. Jackson, “Scheduling a Production Line to Minimize Maximum Tardiness”, Research Report 43, Management Science Research Project, University of California, Los Angeles, USA, 1955.

[18] L. George, N. Rivierre and M. Supri, “Preemptive and Non-Preemptive Real-Time Uni-Processor Scheduling”, Research Report RR-2966, INRIA, Le Chesnay Cedex, France, 1996.

[19] Short, M, “Some complexity results concerning the non-preemptive ‘thrift’ cyclic scheduler”. Proceedings of the 6th International Conference on Informatics in Control, Robotics and Automation (ICINCO 2009), Milan, Italy, pp. 347-350, July 2009.

[20] Graham, R.L, “Bounds on multiprocessing timing anomalies”, SIAM J. Appl. Math., Vol. 17, pp. 416-429, 1969.

[21] A. Carlini and G.C. Buttazzo, “An Efficient Time Representation for Real-Time embedded Systems,” in Proc. Of the ACM Symp. On Applied Computing (SAC 2003), Florida, USA, pp. 705-712, March 2003.

[22] A. Mok, “Task Management Techniques for Enforcing ED Scheduling on a Periodic Task Set”, Proceedings of the Fifth IEEE Workshop on Real-Time Software and Operating Systems, pp. 42-46, Washington, D.C., May 12-13, 1988.

[23] Short, M. “Improved task management techniques for enforcing EDF scheduling on recurring task sets”, In: Proc. of the 16th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2010), Stockholm, Sweden, pp. 56-65, April 2010.