Improved Task Management Techniques for Enforcing EDF Scheduling on Recurring Tasks

Michael Short
Electronics & Control Group,
Teeside University,
Middlesbrough, UK
M.Short@tees.ac.uk

Abstract—The management of tasks is an essential requirement in most real-time and embedded systems, but invariably leads to unwanted CPU overheads. This paper is concerned with task management in real-time and embedded systems employing the Earliest Deadline First (EDF) scheduling algorithm. Currently, the best known techniques to manage EDF scheduling lead to overheads with complexity $O(\log n)$, where $n$ is the number of recurring (periodic/sporadic) tasks. In this paper it will be shown that if both the ready and waiting queues are represented by either i) timing and indexed deadline wheels or ii) digital search trees, then all scheduling decisions may be made in time proportional to the logarithm of the largest time representation required by the system, $p_m$. In cases where $p_m$ is relatively small, for example in some embedded systems, extremely efficient task management may then be achieved. Experimental results are then presented, and it is shown that on an ARM7 microcontroller, when the number of tasks is comparatively large for such a platform (> 250), the worst-case scheduling overheads remain effectively constant and below 20 µs. The results indicate that the techniques provide some improved performance over previous methods, and also seem to indicate that there is little discernable difference between the overheads incurred between employing a fixed- or dynamic-priority scheduler in a given system.

Keywords-Hashed Wheels; Timers; Deadline Scheduling; Implementation Models; Task Management; Digital Search Trees.

I. INTRODUCTION

The effective management of tasks is an essential requirement in many real-time and embedded systems. If these overheads – CPU time reserved for making task scheduling decisions - are not carefully managed, system performance can be negatively affected leading to reduced real-time response and increased power consumption. This paper is concerned with the efficient management of such overheads for recurring (periodic / sporadic) task models, with applications to single-processor, real-time and embedded systems. In this context, the two main aspects (requirements) of task management can be stated as follows:

Maintaining a waiting queue: Many types of events are either periodic or quasi-periodic in nature; for example the signaling of repeated (recurring) events, or the enforcement of temporal isolation between system responses to sporadically occurring (external) events. In these situations, a waiting queue – in which a specific event is inserted, to be activated at some future point in time - is a typical means to achieve the desired behavior.

Figure 1. Aspects of task management.

Maintaining a ready queue: Many systems are required to respond to different internal or external events and perform specific processing in a timely fashion; when multiple events occur simultaneously, then some form of scheduling algorithm is normally required to process the events in an appropriate order. A priority queue – into which an active event is inserted along with some notion of priority (e.g. deadline) is inserted - is a typical means to achieve the desired behavior.

These two main aspects are illustrated in Fig. 1. The performance of algorithms to implement task management is an issue worthy of study - not only for resource-constrained and embedded systems - but also wider applications such as general-purpose operating system implementations, network and discrete simulation environments, and also process planning applications [1][2][3][4]. This paper is concerned with systems implementing the Earliest Deadline First (EDF) scheduling algorithm; specifically, it will suggest new task management techniques to reduce the overheads in such systems. The main motivating factors for the current work are as follows. It is known that when preemption is allowed, EDF allows the full utilization of the CPU and is optimal on a single processor under a wide variety of different operating constraints ([3][5][6]). Although EDF generally allows greater utilization of the CPU over alternate fixed-priority schemes (and - arguably - has many other benefits [7]), it is generally seen as being more cumbersome, complex and harder to implement than fixed-priority schemes [7][8]. When preemption is not allowed, the scheduling problem (in general)
becomes NP-Complete [9]. However if attention is restricted to ‘efficient’ schedulers – i.e. those that do not allow the use of inserted idle-time – then non-preemptive EDF (npEDF) is known to be optimal for many types of task sets amongst this sub-class of schedulers [4][10][11]. Although this paper is primarily concerned with preemptive EDF, the results easily generalize to npEDF. The remainder of this paper is organized as follows. Section II describes the assumed task model, and reviews previous work in this area. Section III presents the proposed techniques for EDF task management. Section IV presents a small computational study designed to provide initial performance measures for the proposed techniques, evaluated using an ARM7 microcontroller. Section V concludes the paper.

II. PREVIOUS WORK

Several previous works have considered the problem of task management overheads, for both fixed-priority and deadline-driven systems; this Section will briefly review this material, along with related work on the management of timers and time representation in real-time and embedded systems. Prior to this, the assumed task model will first be discussed.

A. Task Model

This paper is concerned with implementing the recurring task model on a single processor. Such a system may be represented by a set $\tau$ of n tasks, where each task $t_i \in \tau$ is represented by a tuple:

$$t_i = (p_i, c_i, d_i)$$

(1)

In which $p_i$ is the task period (minimum inter-arrival), $c_i$ 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 [6] and since has been widely adopted – see, for example, [2][3][8][11]. This paper is primarily concerned with periodic tasks; however, extensions to sporadic task management are also described in Section III. Attention is restricted in this paper to implicit and constrained deadline tasks, i.e. those in which $d_i \leq p_i$; such tasks are the most widely discussed in the literature (and used in practice).

B. Fixed Priority Task Management

When task indices and priorities are fixed – as opposed to dynamic – the problem of task management seems to be conceptually much simpler [7]. The ready queue may be represented as a simple priority queue; one simple approach to its implementation is to represent the ready tasks as a bit-field. In this approach, it is first assumed that the tasks are ordered such that for any two tasks $t_i$ and $t_j$, $i < j$ iff $f_i > f_j$, where $f_j$ is the (fixed) priority assigned to task $t_i$, and the goal is to always schedule the ready task with the highest priority (lowest index). Each task is then assigned a corresponding bit in the bit-field, the bit index corresponding to the task index; the individual bit is set to ‘1’ if the task is ready, and ‘0’ if it is not. The advantage of such an approach is that if the number of tasks is less than the bit-width of the underlying CPU, the ready queue can be represented as an unsigned integer; determining the task with the highest priority (i.e. lowest index) can be performed by simply finding the least set bit in this integer. This can be performed with a single machine instruction if supported by the underlying hardware (e.g. on x86-based CPUs) or with 4 machine instructions if not directly supported (e.g. on an ARM or other RISC CPU), using a technique based on the deBruijn sequence [12].

When the number of priority levels is fixed (e.g. 1024), then such an approach – in conjunction with a timing wheel (to be discussed in Section 2C) - may be used to create an $O(1)$ solution [4]. Typically, an index can be placed over an array of FIFO queues, and used to indicate which of the queues contain one or more elements; finding the task with the highest priority can effectively be performed in $O(1)$. An approach similar to this is employed in the so-called ‘O(1) scheduler’ used in the Linux kernel [13], primarily aimed at x86 systems. However when optimal priority assignments such as such as rate or deadline monotonic are employed [6][8], it is easy to observe that the worst-case number of task priorities required will be directly proportional to the largest task period $p_n$ as $n$ increases, leading to solutions that are effectively $O(\log p_n)$. However, since priorities are dynamic in the case of the EDF scheduler, this solution does not seem to directly translate, and alternate schemes are required.
C. EDF Task Management

In the case of EDF scheduling, task priorities are dynamically assigned inversely proportional to their absolute deadlines. A basic solution is to sort the tasks – with every new task arrival – in order of increasing deadlines, giving a solution of complexity $O(n^2)$, $O(n \log n)$ or $O(n)$ depending on the implementation. At first glance, it seems a simple way to reduce this complexity to $O(\log n)$ would be to employ a priority queue based around a binary heap [14]. A binary heap consists of a number of nodes, one for each task, where each node has a maximum of two children. A heap represents a partially sorted list; the partial sorting is achieved via the basic heap operations, which reorder nodes (through swapping) when performing inserting or deleting operations in order to respect the heap invariant. In a min-heap, the heap invariant is simply that the priority of a node must be either identical to or lower than the priority of both its children.

Accordingly, the root of the heap contains the task with the minimum priority; in the case of EDF scheduling, absolute deadlines can be used as keys for the heap invariant, and the root node then becomes the task with the earliest deadline. Although basic heap operations can be performed in time proportional to $O(\log n)$, if a heap is also used to represent the waiting queue problems may arise when several tasks with identical release times must be inserted. As shown by Mok [15], in the case of EDF scheduling there are several conditions which cause heap management to degenerate to $O(n)$. For example, consider the heap of waiting tasks depicted in Fig. 3. Note that insertion of an element into a heap - which already contains one or more elements with identical keys – does not merge the data into a single equivalent element. At $t = 20$, all 5 tasks shown in Fig. 3 will need transferring to the ready queue; when crossing the hyperperiod boundary for periodic tasks, the entire heap contents will need to transferred to the run queue, an $O(n)$ operation.

A solution to this problem, as suggested by Mok [15], is twofold; firstly the waiting queue may be represented as a balanced binary search tree (such as a RB, 3/4 or AVL tree [14]), where the data contained in each node represents the root of a heap. Secondly, the ready queue can be represented as a ‘Heap of Heaps’ (HoH) data structure; this is essentially a heap whose elements can be individual nodes, or heaps themselves; the root node of a sub-heap is used as its effective priority when performing basic operations on the ‘master’ heap. This overcomes the problems outlined above as follows. When a task runs to completion, its next release time is updated and this is used as the key for insertion into the balanced tree. Because the tree is balanced, it supports lookup, insertion and deletion operations in time proportional to $O(\log n)$; if a node with an identical release time exists within the tree, then a heap is created at this node and the tasks relative deadline – which remains static – is used as the key to push the task down the heap. It is trivial to observe that when multiple tasks are required to be released simultaneously, then all that is required is to locate the root of the heap corresponding to the release time, and push this heap onto the HoH ready queue; again an $O(\log n)$ operation.

An alternative to the HoH data structure of [15] is to represent the ready queue (and also the elements of the waiting queue) as binomial heaps [16]. With such an approach, transferring multiple tasks from the waiting queue to the ready queue can be achieved by merging the two binomial heaps in question; since the merging of two binomial heaps is an $O(\log n)$ operation, this leads to a solution with identical worst-case complexity, and is the technique adopted in the LITMUS\textsuperscript{RT} multiprocessor kernel [16]. This particular kernel also has an option to represent the waiting queue using an alternate technique to a balanced tree, known as a timing wheel. This, along with other work in time representation for real-time systems, is the subject of the next Section.

D. Timer Management and Time Representation

It can be assumed that in all embedded and real-time systems, time can be represented by unsigned integers (see for example [17] for justification). As shown by Varghese & Lauck [1], if sufficient memory on the target hardware is available then a timing wheel may be employed to represent a waiting queue. A timing wheel is simply a sequential array of records, whose combined size corresponds to the maximum size of timer required by a system (in the case of task management, this corresponds to the largest task period divided by the required timer granularity, henceforth denoted as $p_m$).

If the records in question correspond to priority queues (or references to priority queues), such a scheme can be operated as follows in a real-time system. When a task – with absolute release time $r_i$ - is inserted into the waiting queue, it is inserted into the $r_i \mod p_m^{th}$ element (queue) in the timing wheel, assuming that time is incremented mod($p_m$) with each timer tick. Since time is represented in integer – and using integers to index into an array is a trivial operation – insert and delete operations take place in constant time. This creates a highly effective solution that is also much simpler than implementing a balanced tree, and is shown graphically in Fig. 4, with elements that are indexed from 0. However, a drawback is that large arrays will be needed for large values of $p_m$; it is easily seen that such a scheme is best suited to embedded systems in which $p_m$ is bounded to be some small power of 2, e.g. 1024.

The assumption that time is represented as integer – and in small embedded systems, normally with a fixed number of bits (e.g. 16) may also lead to timer rollover problems. Deadlines will naturally ‘wrap around’ due to this modular
representation of time, and since the normal laws of arithmetic no longer hold it cannot be guaranteed that \( d_i \mod(2^b) < d_j \mod(2^b) \) when \( d_i < d_j \) and time is represented by \( b \)-bit unsigned integers. In such cases, assuming that the inequality \( p_m < 2^b / 2 \) – i.e. the maximum period is less than half the life time of the underlying linear timer – holds for a given task set, then this problem may be efficiently overcome by using Carlini & Buttazzo’s ICTOH algorithm [18]. This algorithm – which has a very simple code implementation - exploits the fact that the modular distance between any two events (deadlines) \( x \) 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. The ICTOH algorithm has been integrated into the ERIKA Enterprise kernel [23], and is easily extended such that it may be integrated with either regular or binomial heaps for comparing deadlines, to maintain the heap invariants.

![Figure 4. Employing a timing wheel to represent waiting tasks.](image)

III. PROPOSED EDF TASK MANAGEMENT TECHNIQUES

Given the discussion of the previous Section, it can be seen that the current ‘state-of-the-art’ solutions manage tasks in an EDF implementation in \( O(\log n) \) time. This Section will present two solutions that can potentially improve upon this situation by performing task management in time proportional to \( O(\log p_m) \), or equivalently linear in \( b \). In some systems, e.g., small embedded systems, \( p_m \) is likely to be small – for example 1024 – and may lead to more efficient scheduler implementations in such situations. Please note that proposed solutions assume that only feasible task sets are to be scheduled on a single processor; transient or permanent task overloads may lead to timeline breaks, are not considered further in this paper. The first solution employs an extension of the timing wheel, and is described below.

A. The Indexed Deadline Wheel Approach

Suppose that \( p_m \) is small enough to warrant the use of a timing wheel in a given design; as mentioned this will normally be the case for kernels in small embedded systems. Suppose also that tasks are indexed by the scheduler such that for any two tasks \( t_i \) and \( t_j \), \( i < j \) iff \( d_i <= d_j \); with such an approach, a priority queue of tasks – sorted according to minimum relative deadline – may be implemented using a (hierarchical) bit-field representation, as discussed in Section 2B. The main idea of the first solution is to extend the use of a timing wheel such that it can also represent a ready queue; such a queue will be termed a deadline wheel. The basic idea behind this approach can be summarized as follows:

1. Upon power-on, all tasks are inserted into the waiting queue according to their first release time; any tasks that hash to an identical release time are inserted into a bit-field priority queue according to their index (relative deadline).
2. When a timer interrupt occurs, after saving the context of the current task the priority queue \( P_t \) of tasks to be released (if any) at the current time \( t \) is extracted from the timing wheel.
3. The index of the task with the smallest relative deadline \( b \) is peeked from \( P_t \), and its absolute deadline \( d_b \) is determined.
4. \( P_t \) is inserted into the deadline wheel using the key \( d_b \mod(p_m) \); if another queue exists at this location, a linked list of queues is formed (ties can be broken arbitrarily under EDF).
5. Starting from index \( i = t + 1 \), the deadline wheel is searched by incrementing \( i \mod(p_m) \) to find the first non-empty slot; a reference to priority queue \( P_b \) (possibly at the head of a linked list of queues) residing in this slot is extracted; if no valid entry is found and \( i = t \), the idle task index is selected.
6. The task with the smallest relative deadline \( b \) is peeked from \( P_b \) and switched into context.
7. When task \( b \) runs to completion, it is deleted from \( P_b \) before having its next release time and absolute deadline updated; \( b \) is then inserted into the waiting queue (timing wheel).
8. If \( P_b \) is not empty, the index of the task with the smallest relative deadline is peeked from \( P_b \), and its absolute deadline \( d_b \) is determined; it is then inserted into the timing wheel using the key \( d_b \mod(p_m) \).
9. Repeat from step 5 until the next timer interrupt (step 2).

Although this solution will essentially run in constant time for a given (fixed) \( p_m \), the size of the fixed constant degenerates to \( p_m \) iterations in the worst case, which is unacceptable in most situations. The size of this fixed constant can be reduced to \( O(\log p_m) \) by introducing a hierarchical bit-field index over the deadline wheel; such an index is illustrated for a \( p_m \) with 13 slots in Fig. 5. When slot \( i \) contains a valid record, index bit \( i \) is set to a '1', and '0' otherwise. At a given time \( t \), finding the nearest non-zero element can be performed by finding the least set bit after first left barrel-shifting the index by \( t + 1 \), and then adding \( t + 1 \) to the returned value (if the index is non-zero). Note that in a feasible task set, no task should have a deadline at \( t \); a valid record in slot \( t \) must have a deadline at \( t + p_m \mod(p_m) = t \).
When the hierarchical bit-field index requires several levels, for example to cover a larger sized deadline wheel, although its implementation is slightly more complicated there are still only a small number of operations required to determine the nearest valid record. Even for large deadline wheels, this reduces the value of the fixed constant to the time taken to execute a relatively small amount of integer operations, as will be shown in Section IV.

![Figure 5. Indexing the deadline wheel.](image)

At the expense of increased memory use, a simpler solution than performing barrel shifting of indexes would be as follows. If enough memory can be allocated such that the deadline wheel is made up of 2\(p_m\) records, then two (separate) indices of size \(p_m\) – a lower index LI and an upper index UI – may be placed over each half of the deadline wheel. Deadlines may then be inserted into the wheel and time incremented \(\text{mod}(2\cdot p_m)\); task activations can still be inserted into (and retrieved from) a timing wheel of size \(p_m\). When the nearest non-zero record is sought from the deadline wheel, then only two cases need to be considered. If the current time variable \(t\) is \(< p_m\) then the lowest indexed non-zero element is first sought from the LI; if no record is found then the lowest element is then sought from the UI. If the current time variable \(t\) is \(\geq p_m\) then the lowest indexed non-zero element is first sought from the UI; if no record is found then the lowest indexed non-zero element is then sought from the LI. This approach is illustrated in Fig. 6; given that no task has a deadline greater than \(t + p_m\), searching the wheel in this manner will always return the next non-zero record, and no barrel-shifting of the indices is required.

**Theorem 1:** The use of a timing wheel, deadline wheel and bit-field priority queues solve the EDF task management problem in time proportional to \(\log p_m\) per scheduling event.

**Proof:** It suffices to show that all constituent operations in this solution can be performed in \(O(\log p_m)\) or better. Firstly, it follows from [1] that insertion, peek and removal operations on a sequential array of records such as a deadline or timing wheel can be performed in \(O(1)\). Secondly – under the assumption of implicit and constrained deadline tasks – the number of relative deadlines in the task set is upper-bounded by \(p_m\); the operations insert, peek min and delete on a bit-field priority queue (or indexed array of FIFO queues) of size \(p_m\) can be performed in \(O(\log p_m)\) [7]. Finally, searching the deadline wheel for the nearest non-empty slot using the techniques described requires at most \(\log p_m\) iterations. Since all constituent operations can be performed in constant time, the theorem is proved.

For cases when the use of timing and deadline wheels is impractical, the next Section describes a somewhat more generic approach employing digital search trees.

![Figure 6. Using a LI and UI on a deadline wheel of size 2\(p_m\).](image)

### B. The Digital Search Tree Approach

A Digital Search Tree (DST) is a simple data structure that was first described by Coffman [19]. In many respects a DST resembles a balanced binary search tree, but has two major advantages; firstly, component routines for inserting, deleting and searching for keys are extremely simple - the tree does not require complicated rebalancing procedures [14]. Secondly, when keys are unsigned integers, for a key length of \(b\) bits and a radix \(r\), these component routines have a worst-case time complexity of \(O(b/r)\) [14][19][20]. These two advantages make a (slightly modified) DST a good candidate for EDF task management. Each node in a DST has \(2^r\) child links (a binary DST has a radix \(r = 1\)). Inserting a key into a DST involves simply traversing nodes until a NULL pointer is found. At this point, a new node is created from the supplied data and attached to this NULL pointer. When a node is deleted, the node with the required key is first located (if it exists), and swapped with any one of its descendant leaf nodes. The procedure that controls how nodes in the tree are traversed is simple; branches are taken according to the next \(r\) most significant bits of the key, from MSB to LSB. At each node, the supplied key is compared to the node key; if not equal, the appropriate child node is simply taken from the index of \(2^r\) possible child links. Insertion of the 4-bit key '1010' into a partially built DST with \(r = 1\) is shown in Fig. 7 below.

It is trivial to observe that a DST is a good candidate for implementing a waiting queue; for example, if time is represented by 16-bit integers, with a radix \(r = 4\) each node will have 16 possible child links, and insertion and deletion into the DST will only require - at most - 4 nodes to be traversed, and 4 keys inspected, regardless of the number of
nodes in the DST. Since a DST does not keep the nodes in a sorted order, it may seem a poor candidate to implement a priority queue; however, a DST certainly has structure, and given the node traversal procedure the following can be observed; the node with the smallest key lies on a path that is traversed by starting at the root node and taking the left-most (non-NULL) child link at each node, until a leaf is encountered. Thus, the node with the smallest key may be found by traversing no more than \( O(b/r) \) nodes, and taking the smallest key of those examined. For example, in Fig. 6 the path 0101 → 0010 → 0011 is traversed, with the smallest key located being 0010. For nodes with \( r > 1 \), a bit-wise index may be optionally employed at each node, to indicate non-NULL child links and speed up this process.

![Figure 7. DST node insertion.](image)

Thus, when mission times are such that timer rollover is not an issue, the DST may be used to solve the EDF task management problem as follows. The same procedure as outlined in Section A may be employed, with both the waiting and ready queues represented by DST’s. Each node in these DST’s uses a key of either a task release time or absolute deadline respectively, and has as data a reference to a priority queue implemented as a DST. The priority queue DST’s use a task’s relative deadline (encoded in \( b \) bits) as a key, and the task index as data. Alternatively, since relative deadlines are fixed, then bit-field priority queues may be employed instead of DST’s. However, in the case when timer rollover is an issue that must be dealt with, perhaps the simplest solution is as follows. The root node of the ready queue DST can be made such that it cannot contain a key (or data); it has two child links, and is simply used to branch on the MSB of the inserted key. Clearly, any nodes in the left sub-tree have an MSB of ‘0’ and in the right, an MSB of ‘1’. Now suppose that the ICTOH invariant - discussed in Section 2D - holds over the task period relationships, i.e. \( p_m < 2^b/2 \). When the key with the earliest deadline is sought from the ready queue, the MSB of the current time variable \( t \) is used to control which branch is first taken from the root node. If the MSB of \( t \) is ‘0’, then we simply search for the smallest key as detailed above. If the MSB of \( t \) is ‘1’, however, at the root node we first attempt to take the right branch – if non-NULL – and search for the smallest key in this sub-tree as before. If the right branch is NULL, then the left branch is first taken from the root and the smallest key sought as before. It is clear that given the period restrictions described above, any tasks with deadlines that have rolled over will not be processed until there are no keys - at any particular time \( t > 2^b/2 \) - with a deadline value \( d \) lying in the interval \([t, 2^b]\). For best results, then, it is suggested to determine the number of bits \( b \) to use when encoding time for a particular system according to the following simple relationship:

\[
b = \lceil \log_2 p_m \rceil + 1
\]

(2)

**Theorem 2:** The use of DST’s solves the EDF task management problem in constant time per scheduling event for fixed \( b \) and \( r \), leading to a \( O(\log p_m) \) solution.

**Proof:** For fixed \( b \) and \( r \), all DST operations – including finding the minimum element – run in constant time, i.e. \( O(1) \) [14][20]. Since \( b \) can be selected using (2), the proof trivially follows from the discussion above.

It can be estimated that actual performance of a DST is largely dependent on the choice of radix \( r \); for larger \( r \), fewer nodes need to be traversed (and hence keys compared). However, since each node requires 2\( r \) child links, for larger \( r \) the required storage space can increase dramatically. This is illustrated in Table I below, which shows the effect of increasing radix on computation time (worst-case number of key comparisons per insertion/lookup/deletion), and required node storage space (memory locations) for a system using 16-bit timestamps (keys). Note that when \( r = 16 \), the solution essentially mimics an (inefficient) timing wheel, and child links should be replaced by a fixed array of records. It can be observed from this that the DST approach is perhaps best suited to systems in which there is a relatively large amount of system memory available, and the number of tasks is potentially very large (for example in a real-time x86-based, embedded PC system).

| Table I. Effect of increasing DST radix on memory and basic operation overheads (required number of key comparisons). |
|---|---|---|---|---|---|
| **Radix** | 1 | 2 | 4 | 8 | 16 |
| **Memory Loc. / Node** | 5 | 7 | 19 | 259 | 65539 |
| **Comparisons Req’d** | 18 | 8 | 4 | 2 | 1 |

---

In the special case when the number of child nodes is made equal to \( p_m \), the DST essentially acts as a timing wheel; it can be noted that in general a DST resembles the *hierarchical* timing wheel described in [1], but gives improved performance in terms of lookup, insertion and deletion.
C. Extensions to Sporadic Tasks

Sporadic tasks were introduced by Mok [21] to model event-driven, reactive real-time systems. Arrival times of sporadic tasks, as opposed to periodic tasks, are not explicitly known beforehand; that is, successive arrivals of a task $i$ are not necessarily separated by exactly $p_i$ time units as with periodic tasks. Instead, all that can be guaranteed is that successive arrivals of a sporadic task will be separated by a minimum of $p_i$ time units. Typically, when an event (interrupt) linked to a sporadic task occurs at time $t_i$ if the interrupt is enabled the interrupt handler will simply update the absolute deadline of the task to $t_i + d_i$ and insert it into the ready queue. The advantages of such a simple technique are numerous; the implementation is straightforward, each interrupt can be assigned the same (or similar) priority, and on systems which make use of a multiplexed interrupt controller – e.g. x86 and ARM-based devices - only a single, generic interrupt handler that reads back from an interrupt status register needs to be coded (and hence verified) [22].

Most often it will be wished to enforce temporal isolation between successive occurrences of a task. This may be achieved by temporarily disabling the corresponding interrupt source, upon activation. Most devices have multiple interrupt channels that can be enabled or disabled by writing an appropriate code to an Interrupt Enable Register (IER). After updating the task deadline and insertion into the ready queue, the interrupt activation time – the earliest time at which the task can next be activated, i.e. $t + p_i$ – is also updated and an event inserted into a distinct interrupt waiting queue. This queue may be implemented by either a timing wheel or DST. The data fields of the interrupt queue are integers of the same width as the IER; when an event to re-enable the $i^{th}$ bit of the IER is inserted, the corresponding $i^{th}$ bit of the record is set to a ‘1’. With each timer interrupt, if a valid record is retrieved from the interrupt waiting queue, its data contents are ORed with the IER and the record immediately deleted. Since the number of interrupts is fixed (and in many cases equal to the word-width of the CPU) in any system implementation, such a solution requires very little computation and has a constant worst-case performance.

IV. EXPERIMENTAL ANALYSIS AND DISCUSSION

Although the task management techniques described in the previous Section have $O(\log p_m)$ complexity, it is clear that there are several factors that may influence actual computation times and required storage space on a given platform. This Section will describe a series of computational studies were undertaken to begin to quantify this behavior in real situations.

A. Target Hardware and Experimental Procedures

A total of five experiments were carried out in order to explore and compare the performance of the proposed techniques - in terms of time taken to perform scheduling events - as the number of tasks increases. These experiments were carried out on what is currently considered to be a typical hardware platform for the implementation of deeply-embedded real-time systems, the 32-bit ARM7TDMI-S core [23]. The LPC2387 microcontroller from NXP semiconductors was employed in this study, running with a CPU clock speed of 72 MHz. The device has 512 Kb of on-chip flash, and 98 Kb of on-chip RAM. The following five cases were considered in the experiments (note that a timing wheel with 1024 elements was used to represent the waiting queue for cases 1-4):

1. A single-indexed, 1024-element deadline wheel to represent the ready queue (SDW);
2. A dual-indexed, 2048-element deadline wheel to represent the ready queue (DDW);
3. A HoH to represent the ready queue, with the ICTOH algorithm employed as the heap invariant (HoH);
4. A single-indexed, 1024 element fixed-priority FIFO arrangement to represent the ready queue (FPP).
5. A basic $O(n)$ EDF implementation, which indexes through the task array using ICTOH to determine the best task to execute with each timer tick, and with each end task (LIN).

These particular cases were chosen in order to compare the proposed techniques (case 1 and 2) with both the current state-of-the-art (case 3), and also baseline cases including the fixed priority case (case 4) and a basic EDF implementation (case 5). The deadline monotonic priority assignment was employed in case 4. Please note that for case 3, an implicit $n$-element array was employed as the ‘master’ heap, each element of which contains a reference to a sub-heap of tasks consisting of an indexed bit-field. In each experiment, dummy task sets were created. In order to induce the worst-case overhead behavior in these task sets, the following parameters were employed for any particular task $i$, with the total number of tasks being $n$ (in time units of ms):

$$c_i = 1.1$$
$$p_i = 4n$$
$$d_i = 4n - 2(n - i)$$
$$r_i = n - i$$

A timer of granularity 1 ms was employed in each case. This choice of task parameters ensured a relatively low CPU utilization ($\approx 27.5\%$) and that at time $t = n-1$, task $1$ is transferred to the ready queue – which already contains the remaining $n-1$ tasks - and becomes the task with the earliest deadline (highest priority in case 4). This choice also ensured that the tasks were indexed by the scheduler such that the index $i$ was inversely proportional to the relative deadline $d$ (or fixed priority in case 4). Similarly, prior to crossing the hyperperiod boundary (at $t = 2n + n - 1$ [3][17]), task $n$ is required to be inserted into the waiting queue - which contains all remaining $n$-tasks, each with a different activation time. The overheads in each case were measured as the number of tasks was increased, starting from $n = 4$, and doubled in each successive case. A maximum of 256 tasks was used, as above
this point the internal memory space of the device became exceeded; there seems to be little option but to implement task data structures with anything less than $O(n)$ storage. Timestamps from a free-running timer with an accuracy of 0.1 µs were used to calculate the execution times of both the timer ‘tick’ overheads and ‘end task’ overheads. The GNUARM C compiler was employed (within the Keil µVision IDE) to generate code for the target; code optimization was disabled to prevent out-of-order instruction execution impacting the results.

B. Experimental Results

The results to be presented are in units of µs, and include only the cost of updating task data structures, TCBs, and the ready and waiting queues. The cost of task preemption was fixed for each implementation, and therefore omitted. It was found that the cost of saving a task context was 0.36 µs, and restoring a task context 0.39 µs; heavy use was made of the ARM7 Fast-Interrupt-reQuest (FIQ) scratch registers and the store/load multiple register instructions for these routines. Results for each of the experiments detailed in the previous Section are shown in tabular form in Table II, and graphically in Figs. 8 and 9, which show the timer tick and end task overheads on a logarithmic scale representing $n$. Please note that the SDW results were omitted from these figures to improve clarity, due to their similarity to the DDW results.

<table>
<thead>
<tr>
<th>Measurement</th>
<th>4</th>
<th>8</th>
<th>16</th>
<th>32</th>
<th>64</th>
<th>128</th>
<th>256</th>
</tr>
</thead>
<tbody>
<tr>
<td>H0H (tick)</td>
<td>10.2</td>
<td>13.4</td>
<td>16.7</td>
<td>19.8</td>
<td>23.1</td>
<td>26.3</td>
<td>29.6</td>
</tr>
<tr>
<td>H0H (end task)</td>
<td>21.2</td>
<td>27.1</td>
<td>32.9</td>
<td>38.7</td>
<td>44.7</td>
<td>50.3</td>
<td>56.3</td>
</tr>
<tr>
<td>SDW (tick)</td>
<td>6.9</td>
<td>6.9</td>
<td>6.9</td>
<td>6.9</td>
<td>6.9</td>
<td>6.9</td>
<td>6.9</td>
</tr>
<tr>
<td>SDW (end task)</td>
<td>16.4</td>
<td>16.4</td>
<td>16.4</td>
<td>16.4</td>
<td>16.4</td>
<td>16.4</td>
<td>16.4</td>
</tr>
<tr>
<td>DDW (tick)</td>
<td>6.3</td>
<td>6.3</td>
<td>6.3</td>
<td>6.3</td>
<td>6.3</td>
<td>6.3</td>
<td>6.3</td>
</tr>
<tr>
<td>DDW (end task)</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>FPP (tick)</td>
<td>5.7</td>
<td>5.7</td>
<td>5.7</td>
<td>5.7</td>
<td>5.7</td>
<td>5.7</td>
<td>5.7</td>
</tr>
<tr>
<td>FPP (end task)</td>
<td>14.4</td>
<td>14.4</td>
<td>14.4</td>
<td>14.4</td>
<td>14.4</td>
<td>14.4</td>
<td>14.4</td>
</tr>
<tr>
<td>LIN (tick)</td>
<td>5.7</td>
<td>5.8</td>
<td>18.1</td>
<td>34.7</td>
<td>67.7</td>
<td>137.5</td>
<td>255.4</td>
</tr>
<tr>
<td>LIN (end task)</td>
<td>3.3</td>
<td>5.9</td>
<td>11.1</td>
<td>21.4</td>
<td>42.1</td>
<td>85.2</td>
<td>165.5</td>
</tr>
</tbody>
</table>

Figure 8. Graphical representation of increasing task numbers on scheduler ‘tick’ overheads.
C. Discussion

It can be observed from these data and figures that when the number of tasks $n$ is relatively small (e.g. $\leq 8$), then the overheads in all cases are comparable to the overheads reported for other basic $O(n)$ solutions (with a small number of tasks) that have been previously reported for similar targets [23][24]. Indeed in such a situation, the basic LIN approach as implemented in this study is comparable with (or outperforms) competing methods; since the code implementation and required data structures are also comparatively simpler, prospective system designers should be aware this would appear to be the implementation of choice for small $n$. As the number of tasks increases, the logarithmic rise in overheads of the HoH approach can be observed, whilst the FPP, SDW and DDW overheads remain effectively constant. The drawback of the LIN approach also becomes apparent; as may be expected, the ‘tick’ overheads are not competitive with the DDW, SDW and FPP for $n > 6$, and with the HoH method for $n > 16$. Although the ‘end task’ overheads perform somewhat better, they are still not competitive with the alternatives for $n > 24$ and $n > 64$ respectively. Although the logarithmic rise in overheads in the HoH case is clearly acceptable for many systems, these results would indicate a small – but perhaps significant in some cases - improvement in the SDW and DDW techniques over the HoH technique in this instance. Since it known that overheads can lead to increased task response times (e.g. see [23][24]), the adoption of the proposed techniques may prove beneficial to certain classes of real-time system. Although the DDW overheads are marginally less than the SDW (and are almost indistinguishable from the FPP case), as mentioned this is at the expense of increased memory used to implement the larger deadline wheel. It may be observed that the reported levels of overhead - in all cases - may be improved by allowing code optimizations (either at the compiler level or by hand-assembly), however these benefits would be effectively uniform and would serve little purpose in a comparative study. Informal experimentation seems to suggest these optimizations have a greater effect on the ‘end task’ overheads on the current platform.

It is interesting to observe the level of overheads between the proposed techniques and the fixed-priority case. Considering Fig. 5, in the fixed-priority case each record would consist of a FIFO queue containing tasks with the same priority. To determine the best task, they may be searched for the first non-empty record in a fixed order from record 0 (highest priority) to 11 (lowest priority). The only differences between the dynamic- and fixed-priority cases in this situation are that i) deadlines (and not fixed priorities) are used for insertion into a particular FIFO record and ii) the FIFO records are searched not in a fixed order (highest to lowest) but in an order that depends on the current time $t$. As demonstrated by the results obtained, the run-time differences between the two are almost – but not exactly - negligible.

A final point for discussion is the amount of memory space that is required for each implementation. It is possible to calculate prior to run-time the number of bit field priority queues that are required. In the worst case, a safe upper bound to employ would be $\min(n, p_m)$ queues, although – as is the also the case with stack sharing [8][23] - harmonically related periodic tasks may sometimes share a single queue. Memory space may be allocated statically at compile time for these
queues, and single-level pointer indirection used to build the records to be inserted into the deadline and timing wheels. These latter two points are obviously beneficial with respect to coding guidelines and standards such as MISRA, which forbids the use of dynamic memory allocation and places strict limits on the use of pointer indirection [25].

V. CONCLUSIONS AND FURTHER WORK

This paper has considered techniques for the enforcement of EDF scheduling on periodic and sporadic task sets on a single-processor. Previous work in this area has been described, and several different solution techniques have been suggested. These solution techniques are primarily intended to reduce the overheads to a relatively small value given by $log(p_m)$, and essentially independent of the number of tasks. Preliminary results have been presented that indicate the potential usefulness of the techniques; however more exploration in this area is required. Specifically, further work will aim to assess and compare the potential memory / execution time tradeoffs further, and explore the techniques based around DST’s. Finally, perhaps the most interesting observation of this paper - which would seem to be supported by the computational study - is that when compared to the fixed-priority case, the proposed EDF scheduling techniques not only have the same analytical worst-case overheads, but the observable levels of overheads in both cases seems to be almost indistinguishable in practice.

ACKNOWLEDGMENT

The author wishes to express thanks to Prof. Aloysius K. Mok for kindly supplying a copy of reference [15] during the preparation of the paper, which was out of print in the UK at the time. The author also wishes to express thanks to Dr. Nathan W. Fisher for his guidance when preparing the revised manuscript.

REFERENCES