

Article

Exact and Heuristic Algorithms for Thrift Cyclic Scheduling

Michael J. Short

Embedded Systems Laboratory, University of Leicester, Leicester, UK; E-Mail: mjs61@le.ac.uk;
Tel.: +44(0)116 252 5052; Fax: +44(0)116 252 2619.

Received: 7 October 2009 in revised form: 17 November 2009 / Accepted: 18 November 2009 /
Published: 26 November 2009

Abstract: Non-preemptive schedulers, despite their many discussed drawbacks, remain a very popular choice for practitioners of real-time and embedded systems. The non-preemptive ‘thrift’ cyclic scheduler—variations of which can be found in other application areas—has recently received considerable attention for the implementation of such embedded systems. A thrift scheduler provides a flexible and compact implementation model for periodic task sets with comparatively small overheads; additionally, it can overcome several of the problems associated with traditional ‘cyclic executives’. However, severe computational difficulties can still arise when designing schedules for non-trivial task sets. This paper is concerned with an optimization version of the offset-assignment problem, in which the objective is to assign task offsets such that the required CPU clock speed is minimized whilst ensuring that task overruns do not occur; it is known that the decision version of this problem is complete for Σ_2^P . The paper first considers the problem of candidate solution verification—itself strongly coNP-Complete—and a fast, exact algorithm for this problem is proposed; it is shown that for any *fixed* number of tasks, its execution time is polynomial. The paper then proposes two heuristic algorithms of pseudo-polynomial complexity for solving the offset-assignment problem, and considers how redundant choices of offset combinations can be eliminated to help speed up the search. The performance of these algorithms is then experimentally evaluated, before conclusions are drawn.

Keywords: non-preemptive scheduling; offset-assignment; heuristics; complexity

1. Introduction

In many real-time embedded systems, some form of scheduler is generally used instead of a full “real-time operating system” to keep the software environment as simple as possible. A scheduler typically consists of a small amount of code, and a minimal hardware footprint (e.g., a timer and interrupt controller). In this context, a scheduler is a basic component of an embedded system architecture that is primarily employed to assign (or ‘dispatch’) software tasks to CPU’s at run-time. In a typical situation, at any given time there will be more tasks than CPU’s available; the role of the scheduler is to assign tasks to CPU’s such that some design goal is met. This paper is concerned with single-processor scheduling, in which there are several periodic tasks to be dispatched on a single CPU, and the goal is to schedule the tasks such that deadlines are not missed. An example of such a situation would be a processor employed in a critical application such as automotive/avionic control and robotics. In such a situation, ‘late’ calculations—*i.e.*, deadline misses—may lead to system instability, resulting in damage to property or the environment, and even loss of life [1].

In general, a scheduler for such a system may be designed around several basic paradigms: time-triggered or event-triggered, and preemptive or co-operative (non-preemptive) [1,2,4,5]. This paper is primarily concerned with time-triggered, co-operative (TTC) schedulers. TTC architectures have been found to be a good match for a wide range of low-cost, resource-constrained applications. These architectures also demonstrate comparatively low levels of task jitter, CPU overheads, memory resource requirements and power consumption [2-5]. Additionally, exploratory studies seem to indicate better transient error recovery properties in TTC systems over their preemptive counterparts; however this issue requires much further investigation [6,7].

However, non-preemptive schedulers also have certain well discussed drawbacks over their preemptive counterparts; a presumed lack of flexibility, an inability to cope with long task execution times and long/non-harmonic periods, and fragility/complexity problems when generating a schedule being the main perceived obstacles [3,5,8,10]. In practice several techniques now exist to help overcome some of these problems. For example, co-operative designs based on well-understood patterns and state machines [9] coupled with pre-runtime scheduling techniques [4] provide good working solutions to the ‘long task’ problem.

The simplest form of practical TT scheduler is a “cyclic executive” [2,3,8]. Cyclic executives achieve their behavior by storing a (repeating) look-up table which controls the execution of jobs at any instant of time. The lookup table is divided into a number of minor cycles, which make up a major cycle; the current minor cycle advances with each timer ‘tick’. Such a model is generally effective when all task periods are harmonically related, resulting in a short major cycle length. It is known that the off-line problem of assigning task instances to minor cycles (*i.e.*, creating the lookup table), such that periodicity constraints of the task are respected, is NP-Hard [8]. However, several effective techniques—such as those adapted from bin packing and simulated annealing—have been developed for most small instances of the problem [8].

A clear limitation that can arise in such architectures is as follows; the major cycle length is proportional to the least common multiple of the task set periods. Thus, when arbitrary - as opposed to harmonic - task periods are considered, no sub-exponential bound can be placed on the length of the required major cycle. In these cases the amount of memory required for storing the lookup table, and

the resulting computational resources that are needed to design the schedule, become hugely impractical. Therefore heavy restrictions have to be placed on the choice of task set periods.

To overcome this limitation, the non-preemptive ‘thrift’ scheduler (or ‘TTC’ scheduler) has been proposed [9]. This scheduling algorithm does not allow the use of inserted idle-time, and essentially maintains a system of congruences to replace the lookup table, and is presented in pseudo code in Figure 1. As can be seen, when tasks are released by the scheduler, they are immediately dispatched on a first-come, first-served basis (FCFS). In general, generating a feasible schedule for a thrift scheduler reduces to assigning offsets (delays) to each task [9,10], resulting in an asynchronous schedule. However, previous investigations into the complexity of deciding the feasibility of asynchronous schedules have shown it to be a strongly coNP-Complete problem, at least in the case of the preemptive schedulers [11]. Moreover, as Goossens [12] has commented, when attempting to assign offsets to preemptively schedule a task set, that there seems to be no way to avoid enumerating and searching an exponential amount of offset combinations; in fact the decision version of this particular problem is known to be complete for Σ_2^P , even when preemption is allowed [11]. Unsurprisingly, similar results have been demonstrated for the non-preemptive thrift cyclic scheduler, in which certain task parameters are allowed to be either integer or fractional [20,21].

Figure 1. Thrift scheduling algorithm, in pseudo-code.

```

1  START
2  tick := 0;
3  DO(FOREVER)
4      Timer_Wait();          // Wait for timer signal
5      FOR i := 1 TO n DO
6          IF(((tick - oi) mod(pi)) = 0)
7              Run(ti);      // Immediately run a task, if due
8          END FOR
9          tick := tick + 1;   // Advance the time
10 END DO
11 EXIT

```

The thrift scheduling algorithm has also been suggested for use in domains other than that of embedded systems, for example the processing of information requests in mission critical security applications [22], and the underlying model has also proved to be identical (in most aspects) to scheduling problems described in other application areas, for example the periodic vehicle delivery problem [23]. The underlying concept in these formulations is that there exists a periodic (indefinitely repeating) system of actions to be performed (tasks to execute, information to be serviced, items to deliver) in which the temporal separation of actions must *exactly* correspond to the actions’ periodicity requirements; the optimization goal is to assign individual start times (‘offsets’) to each action, such that the number of actions to be simultaneously performed over the system lifetime is minimized.

The application of heuristics for this assignment of offsets in such problems—for example using greedy approaches such as the ‘First-Fit Decreasing’ (FFD) algorithm—is an attractive option; a severe drawback of recent works has been the limitation that in each step of the FFD procedure, it is required to evaluate the quality of a partial candidate solution; which is itself strongly coNP-Hard [20]. In most cases, previous studies have severely limited the choice of task periods to attempt to circumvent this problem. For example, in [10] Gendy & Pont describe a heuristic optimization

procedure to generate feasible thrift schedules. Although the technique works reasonably well in practice, it is required to limit the ratio of minimum to maximum periods in the task set to 1:10. A similar restriction on periods is enforced by Goossens [11], who investigates heuristic methods for preemptive schedules. In other domains, similar restrictions are placed on the various parameters in question [22,23]. For many applications, this restriction is clearly unrealistic. Although it has been argued that in many cases, a task set with either coprime or non-harmonic periods is not representative of real-world applications (e.g., [10,14])—this is true only for closely related transactions, such as filtering and updating a single control loop. In reality, for economic, topological and efficiency reasons, a single processor may well be performing multiple—not physically related—functions. As such, the periods are dictated by many factors and there is likely to be little, if any, harmonic coupling between transactions [5,8].

If thrift scheduling is therefore to be employed in more realistic and representative situations, then fast and effective algorithms to both decide feasibility and assign task offsets are required. This forms the main motivation for the current research. The remainder of the paper is organized as follows. Section 2 presents the task model, defines the problem and reviews its complexity. Section 3 describes the Largest Congruent Subset (LCS) algorithm, which significantly improves upon previous algorithms for the feasibility problem. In Section 4, exact and approximate methods for solving the offset assignment problem are described, along with a discussion of how offset symmetries can be efficiently broken. Section 5 presents detailed results of computational experiments to investigate the performance of the new algorithms. Section 6 concludes the paper and suggests areas for further research.

2. Problem Formulation and Complexity Review

2.1. Task Model and Problem Description

Consider a set τ of n real-time tasks, where each task $t \in \tau$ is represented by a four-tuple:

$$t_i = (p_i, c_i, d_i, o_i) \quad (1)$$

In which p_i is the task period, c_i is the worst case computation time of the task, d_i is the task deadline and o_i is the task offset (release time), represented as integers in suitable processor time units, for example microseconds. For the purposes of this paper it is assumed that the worst case computation time c of each task has been determined using suitable analysis techniques (see, for example [15]) and that an appropriate ‘factor of safety’ has been added to this estimate. Even when rigorous techniques such as static analysis are used in conjunction with standard testing, it is normal to overestimate these values by a factor of around 5% [10,15]. In this paper it also assumed that tasks are not subject to precedence constraints, and task I/O processing is small enough to be neglected. The paper restricts attention to instances having $n \leq 30$ tasks. Since no restriction is placed on the choice of task periods, this is representative of most real-world task sets. The objective of any feasibility analysis or test is to determine if the task set will always meet its deadlines for a given scheduling algorithm. In the case of the non-preemptive thrift scheduler, this surmounts to verifying that a ‘task overrun’ – an overload of

the processor in any given time quantum - does not occur. The time quantum is normally referred to as the tick interval t_{Tick} and is chosen to be as large as possible given the task periods, using (2):

$$t_{Tick} = \text{gcd}(p_1, \dots, p_n) \tag{2}$$

where gcd represents greatest common divisor. Without loss of generality, it can be assumed that the offsets for each task are specified as an integer multiple of the tick interval, according to the following condition [10-12]:

$$\forall i; 0 < i < n; o_i = k \cdot t_{Tick}; 0 \leq k < \frac{P_i}{t_{Tick}} \tag{3}$$

Since it is desired to prevent task overruns, a constraint is placed in thrift schedulers that the effective deadline of *all* tasks released at the start of any tick interval is the start of the following tick interval [2,3,9]. As such, a specific deadline is often omitted when describing task sets for thrift schedulers. Under these conditions, a sufficient and necessary schedulability test for synchronous task sets can be formulated to be as follows [21]:

$$\sum_{i=1}^{i=n} c_i \leq t_{Tick} \tag{4}$$

Although this also provides a sufficient test for offset-free schedules, it is clearly not necessary, as illustrated in Figure 2. Similarly, two necessary (but not sufficient) related tests for offset-free thrift schedules are as follows:

$$\forall i, 0 < i < n; c_i < t_{Tick} \tag{5}$$

$$\sum_{i=1}^{i=n} \frac{c_i}{p_i} \leq 1 \tag{6}$$

That is, the execution time of each task must be less than t_{Tick} , and the total CPU utilization u must be less than unity. A necessary and sufficient test for schedulability can only be determined algorithmically; existing approaches generally perform this through a simulation of the task schedule over a fixed time horizon, known as the schedule hyperperiod—this algorithmic process is generally referred to as “hyperperiod simulation” [1,9-12]. In the case of the thrift scheduler, given the deadline restrictions the hyperperiod has a length h given by [20,22,23]:

$$h = \text{lcm}(p_1, \dots, p_n) \tag{7}$$

where lcm is the least common multiple, in this case of the task periods. At each search point, the computational load η placed on the processor by the schedule in the given time bound of the current search point is computed. From (3), it can be seen that tasks may only be released at the start of any given tick interval. Thus, if the time interval is divided up in segments such that $\eta[t_1, t_2]$ defines a single tick interval, as follows:

$$\forall k, 0 \leq k < h; t_1 = k \cdot t_{Tick}, t_2 = t_1 + t_{Tick} \tag{8}$$

Then it follows that an asynchronous thrift schedule is schedulable iff:

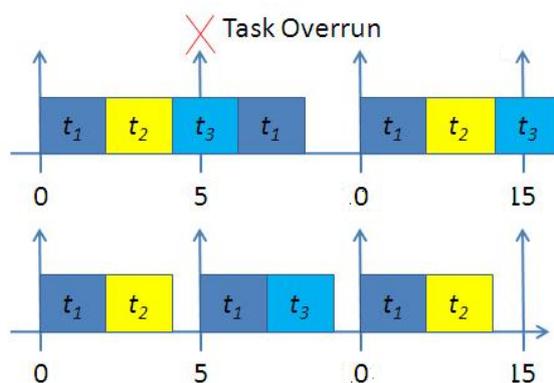
$$\forall k, 0 \leq k < h; \eta[k] \leq t_{Tick} \tag{9}$$

Deciding this problem (9) will be referred to as the Thrift Schedule Asynchronous Feasibility Problem (TSAFP). The search version of this problem can be used to determine c_{Max} , the worst-case demand for processor time over the entire hyperperiod for a given asynchronous schedule. It can be seen (e.g., Figure 2) that by assigning appropriate task offsets, this value of c_{Max} can be considerably reduced over the synchronous case; ideally we wish to assign offsets such that (9) is satisfied. In addition, since t_{Tick} is fixed by the task periods which, in turn, are dictated by the environment, it can also be seen that with knowledge of c_{Max} any task set satisfying (6) can effectively be scheduled by increasing the clock speed of the CPU by a factor α , defined as follows:

$$\alpha = \frac{c_{Max}}{t_{Tick}} \tag{10}$$

Since the task execution times are largely (inversely) proportional to the CPU clock speed, this clearly follows from (9). However, there are a multitude of reasons – for example power consumption, logical circuit stability, EMC issues and transient rejection – why system designers would wish to minimize α and hence the clock speed [8,9].

Figure 2. Effect of task release times on feasibility, showing (top): a synchronous infeasible task set with $\tau = \{(5,2),(10,2),(10,2)\}$; (bottom): an asynchronous (feasible) version of this task set with release time of 1 tick – 5 time units - added to t_3 .



In this paper, we therefore consider the following minimax problem: Given a task set τ , assign a non-negative offset to each task such that c_{Max} is minimized. This problem will be referred to as the Thrift Schedule Offset Assignment Problem (TSOAP). Consideration of such a problem has several advantages over previous (decision) formulations of similar problems (such as those of [8], [10] and [12], for example); the algorithms to be described in the current paper will always find a schedule that is feasible. Although the value of α for the found schedule may not be the optimal value, it is hoped that it will be relatively close to it. Additionally, many modern low-cost microcontrollers (such as the ARM-7 and its variants) feature the ability to dynamically change the CPU clock speed—over a wide range of possible values—by manipulating the appropriate on-chip PLL registers. In such cases, the fragility problems (discussed in [8] and [10] for example) are greatly reduced, as a change to the schedule may also simply require a single line of code to be changed to adjust the clock speed. An additional benefit of such an approach is that the algorithms to be presented in this paper may be adapted for use in other domains – for example the fractional periodic delivery problem described by

Campell & Hardin [22]. It can also be noted that conceptually, this problem is somewhat similar to the makespan minimization problem of multiprocessor scheduling – the dual of Bin Packing [13,16,17]. In the following section, the complexity of this problem is briefly reviewed.

2.2. Complexity of the Problem

This Section first considers the decision version of the TSAFP. Formally, we are given a set of n tasks, each with period p_i , offset o_i , computational cost c_i , and a positive integer b . The question that is asked is as follows: *is the maximum demand for processor time over the hyperperiod (c_{Max}) less than or equal to b ?* From Figure 1, it can be seen that TSAFP is in coNP, since the existence of a counterexample (task overrun) in the j^{th} tick can be verified in polynomial-time by setting $\text{tick} := j$ and executing lines 5,6 and 7 of the TTC algorithm, summing the execution times of all tasks that are run.

Theorem 1: TSAFP is strongly coNP-Complete.

Proof: Short (2009) [20], Transformation from the complement of Simultaneous Congruences Problem (SCP) which is known to be strongly NP-Complete [11].

It should be noted at this stage that it was previously (incorrectly) assumed by some authors, that a particular version of the TSAFP problem was actually in P [21,23]. Considering next the decision version of TSOAP, again we are given a set of n tasks each with period p_i , computational cost c_i , and a positive integer b . The question that is asked is as follows: *is it possible to assign a non-negative offset to each task such that the maximum demand for processor time over the hyperperiod (c_{Max}) is less than or equal to b ?* Given the previous result, this problem is clearly coNP-Hard; however under the assumption that $P \neq NP$, it is worthwhile investigating where the problem lies on the so-called “polynomial hierarchy” [16]. From (3), it can be seen that the task offsets for ‘Yes’ instances of this problem can be encoded in a number of bits that is less than or equal to the task periods, and hence the size of the problem instance. Given the previous Theorem, the resulting asynchronous schedule is verifiable in polynomial time by a Turing machine with an oracle for the feasibility problem; the problem resides in Σ_2^P . The lower bound that shows the problem is complete for this complexity class is achieved by a transformation from the Periodic Maintenance Scheduling Problem (PMSP); this problem is known to be Σ_2^P -Complete [11,19].

Theorem 2: TSOAP is Σ_2^P -Complete.

Proof: Short (2009) Transformation from PMSP [20].

Given these results, it can be seen that there are two (closely related) issues to be addressed in order to efficiently solve this problem, even approximately. In order to begin to address these complexity issues, the coNP-Hard problem of determining c_{Max} for a given asynchronous schedule will first be addressed. From (7), it can be seen that the length of the hyperperiod that needs to be examined is

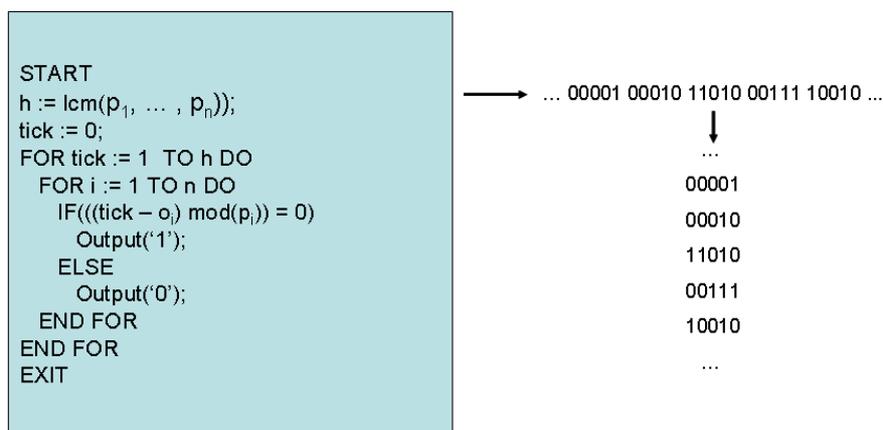
potentially proportional to the product of the task periods, a largely undesirable situation. An algorithm to circumvent this problem is described in the following Section.

3. Algorithm for TSAFP

3.1. Basic Idea

The basis for a new algorithm starts from a single requirement; given the thrift scheduling algorithm (as shown in Figure 1) and an asynchronous task set τ , determine - in the fewest possible steps—the correct value of c_{Max} . In order to find this value in the fewest possible steps—in a manner that is robust to the task set periods—the possibility of examining task phasings is explored. Since the scheduler controls the release of each task, and tasks can only be released at the start of each tick interval, the scheduler can be viewed purely as a generator of an equivalent binary sequence, as follows. Given the task parameters τ , the scheduler produces h n -bit long binary sequences, one for every tick interval; where a ‘1’ in each sequence indicates that the corresponding task is released at this tick, and a ‘0’ represents otherwise. This abstraction is illustrated in Figure 3, for a 5-task system.

Figure 3. Thrift scheduling as binary sequence generation.



In such an abstraction, the demand for processor time η at each tick interval relates to this binary sequence as follows. The processor demand at the k^{th} tick interval is determined by simply weighting the i^{th} binary digit in the k^{th} n -bit sequence by the i^{th} task’s execution time, c_i , then summing the weighted totals. To observe the worst-case behavior of a given schedule, it follows from Theorem 1 and Equation (7) that h n -bit long binary sequences must be examined. In general, from (7), it can be observed that from the numerical properties of least common multiple, in the worst-case h is proportional to the product of the task set periods (this situation occurs when all periods are pairwise coprime). If the maximum of the task set periods is represented by the parameter p_{max} , it follows that the running time of a hyperperiod simulation algorithm is given by $O(n p_{max}^n)$. It can also be observed that in the best case—for instance when all task periods are restricted to be integer powers

of 2 (2, 4, 8, 16, 32 ... etc.)— h is simply proportional to p_{max} ; the best one can hope for is that the algorithm running time is pseudo-polynomial in the input parameters.

It may seem that a useful restriction to consider would be to place an upper bound on the choice of periods—say in the interval [1,50]—but this can still result in a length of h that can approach 10^{38} . In most cases, a more realistic restriction would be to suppose time is represented in milliseconds, and tasks may require periods in the interval [1, 1,000]. It is trivial to create instances in which number of tasks is small (e.g., ≤ 30), and the length of h exceeds 10^{100} with this more realistic restriction. It can be observed then, that the actual time complexity of deciding a given PSD problem instance is incredibly fragile, and is highly dependent on the task periodicity requirements.

However, consider a parameterization of the input representation, which primarily considers the number of tasks n and the largest task period p_{max} (the remaining task parameters may be neglected from this analysis as they do not affect algorithm run times). Now, in a system with n tasks, there is a maximum of 2^n possible binary combinations of tasks, the actual patterns which will be generated by the scheduler at run time depend on the choice of offsets; for almost all instances of this problem, $2^n \ll h$. For example, consider a system of 10 tasks with periods in the interval [1, 1,000]. Whilst h may approach 4×10^{20} in such a situation, 2^n is significantly smaller—1,024—in comparison. In most real-world instances of the problem, n will in fact be relatively small (e.g., ≤ 30). This observation forms the basis for the LCS algorithm; if an algorithm can be developed such that the running time is exponential only in the number of tasks n , and polynomial in the representation of p_{max} , this will be a marked improvement in efficiency in most cases; it will in fact constitute a fixed-parameter tractable (FPT) algorithm in the sense described by Niedermeier [25]. In order to explore this possibility further, the nature of congruencies in a periodic system of tasks will be explored.

3.2. Congruence of Periodic Tasks

Consider two tasks t_1 and t_2 represented purely as binary sequences. This is shown in Figure 4 for the case where $p_1 = 5$ and $p_2 = 3$, with offsets of $o_1 = 1$ and $o_2 = 2$. The coincident task releases (logical AND) are also indicated, where a ‘1’ indicates that the two tasks are released (phased) together.

Figure 4. Congruence relationship of two periodic tasks.

t_1	010000100001000010000100001000010000
t_2	001001001001001001001001001001001001
$t_1 \wedge t_2$	00000000000010000000000000001000000000

Given the thrift scheduling algorithm, each task can be represented as a linear congruence of the form:

$$x \equiv r_i \pmod{p_i} \tag{11}$$

where x is one of the (multiple) possible solutions to the congruence. If the two tasks are phased together (denoted by $\phi(t_1, t_2)$) then x must be a valid integer solution to both congruences simultaneously. According to the *linear congruence theorem* this occurs iff:

$$\varphi(t_1, t_2) \mapsto (o_2 - o_1) \mid \text{gcd}(p_1, p_2) \tag{12}$$

That is, if the relative offsets of the tasks divides the *gcd* of the periods. Suppose now that we wish to consider the phasing of a set of m tasks. Clearly, x must now provide a valid solution to each of the m simultaneous congruences for the tasks to all be in phase. According to the *Chinese remainder theorem* [12,18], (12) can be extended such that a valid integer solution to such a set of m congruences exists iff:

$$\varphi(t_1, \dots, t_m) \mapsto \forall i, \forall j; i \neq j; (o_j - o_i) \mid \text{gcd}(p_i, p_j) \tag{13}$$

That is, if each pair wise combination of tasks in the set m satisfies (12) simultaneously. In such a case that each pair wise combination of periods in m are coprime then (13) is guaranteed to be true. Equations (12) and (13) form the basis for the LCS algorithm. A *congruent subset* is defined as a set $T \subseteq \tau$ s.t. $\forall i, j \in T$, equation (13) holds. Let $|T|$ be the cardinality of the subset, *i.e.*, the number of tasks it contains, and let (T) refer to the magnitude of the subset:

$$(T) = \sum_{i \in T} c_i \tag{14}$$

The purpose of the LCS algorithm is to search for a congruent subset of tasks with the largest magnitude, given an asynchronous task schedule. Central to the proposed algorithm is the notion of a *phase matrix* ϕ . This is an n -by- n matrix that contains all pairwise task phasing information, encoded in binary format. For all elements $i, j, j > i$, a ‘1’ is placed in the ij^{th} element of the matrix indicating a phasing of tasks i and j , and a ‘0’ indicating otherwise. The matrix is ‘0’ both on and below the diagonal, and can be generated by repeated application of the *gcd* algorithm and Equation (12) to each pair wise combination of tasks in the set, and effectively defines the powerset of all possible task phasings. Each row of this matrix is referred to as a ‘phase code’, and represented as an n -bit long binary string; in this paper, each phase code is represented as an unsigned integer. The phase code P^j corresponds to the j^{th} row of the phase matrix ϕ .

3.3. Algorithm Description

The LCS algorithm (Please note that an earlier for the LCS algorithm was previously described by the current author in [21]) operates as a depth-first search of all possible task phasings, employing simple bounding heuristics to help prune the search. The inputs to the algorithm are a task set τ , and the output of the algorithm is the determined value of c_{Max} . The first step is the calculation of the phase matrix. The initial incumbent solution value b is set to the maximum of the largest task execution times, and summed execution times of any pairwise phased tasks during this process; c_{Max} cannot be any less than this. The algorithm then begins to recursively search for congruent subsets of tasks; the main elements of the algorithm are shown in Figure 5.

Each node of the search tree represents a subset T of congruent tasks; the depth of recursion corresponds to $|T|$. The incumbent is updated whenever a subset with $(T) > b$ is found. Each node has associated with it its own phase code P . Since only proper congruent subsets can be expanded as child nodes in the search, the i^{th} task can only be added into the current subset T to form a new child node with subset T' if the i^{th} bit of P is set to a ‘1’. To generate the new phase code in the child node, a

logical AND is performed between P' and P^i , the i^{th} row of the phase matrix. This effectively applies equation (13) in one single operation, and the only bits that will subsequently be set to '1' in P thus correspond to tasks that are properly congruent with all tasks in T . When an empty phase code is encountered, the algorithm begins to backtrack since either a leaf node has been reached or - since bits are cleared by the algorithm after the corresponding child node has been explored - no further child nodes exist.

Figure 5. LCS algorithm, in pseudo-code.

```

01 procedure LCS( $\tau$ , n)
02 {
03    $b := \max(c_i), i \in \tau$ ; // Set initial incumbent
04    $\phi := \text{Create\_Phase\_Matrix}(\tau)$ ;
05   FOR  $i := 1$  TO  $n$  DO
06     Expand( $i, 0, P^i$ ); // Expand each task
07   END FOR
08   RETURN( $b$ ); // Return incumbent
09 }

10 procedure Expand( $i, T', P'$ )
11 {
12    $P := P' \& P^i$ ; // Generate new phase code
13    $T := T' + i$ ; // Add task  $i$  to the set
14   IF ( $T$ ) >  $b$ 
15      $b := (T)$ ; // Update incumbent
16   END IF
17   WHILE  $P \neq \{0\}$  AND ( $T$ ) +  $b_{up} > b$  DO
18     {
19      $j := \text{ssb}(P)$ ; // Locate the next set bit in  $P$ 
20     {
21     Expand( $j, T, P$ ); // Expand the new child node
22      $P_j := '0'$ ; // Mark the node as visited
23     }
24     }
25   END WHILE
26 }

```

The call to sub-function *ssb* on line 19 locates the next lowest set bit in the current phase code; since these codes are represented as bit-strings, this can be performed in small fixed constant time, e.g. using techniques based around a deBruijn sequence [24]. The pruning rule that is implemented simply prevents the expansion of child nodes which cannot possibly lead to a value greater than the current incumbent. This simple procedure works as follows; suppose the current node represents a subset T , and that a proper child node corresponding to the j^{th} task is about to be expanded. An upper bound b_{up} on the best possible value that can be achieved by such an expansion is as follows:

$$b_{up} = \sum_{i=j}^n c_i \quad (15)$$

This bound expresses the fact that the best possible situation - in which *all* tasks corresponding to unexplored branches - are congruent to T . If this value is not greater than b , then the node need not be expanded. Moreover, since the search progresses in a depth-first, left-to-right search, the current node can be considered fathomed; this bound can only be non-increasing for all child nodes greater than j .

This fathoming rule, along with the fact that the algorithm can be implemented entirely using (fast) integer arithmetic, leads to a very efficient formulation. After first showing that the algorithm completely solves the TSAFP, its time complexity is analysed.

Theorem 3: Algorithm LCS exactly solves the search version of TSAFP.

Proof: By inspection. It can be seen that LCS will recursively enumerate only congruent subsets of tasks as the search progresses, as follows; line 12 ensures that only subsets of tasks that conform to (14) are considered as valid child nodes in the search, and the IF statement of line 20 ensures that only these nodes are actually expanded. The WHILE statement on line 18 ensures that the algorithm will explore every possible child node, backtracking iff either no unexplored child nodes remain or the current node has been fathomed according to (15). The conditional assignment on lines 15 and 16 ensure that the incumbent is only updated when the magnitude of the proper congruent subset (T) corresponding to the current node exceeds the current incumbent value b . Upon termination, the incumbent b clearly corresponds to the correct value of c_{Max} .

3.4. Complexity

The analysis starts with the generation of the phase matrix. It can be seen that with a set of n tasks, there are exactly $0.5 n \times n + 1$ combinations of task pairs. To generate the matrix, each pair requires an application of the gcd calculation and application of (15). If the periods in question can be encoded in $\log_2(p_{max})$ bits, generation of this matrix has time complexity $O(n^2 \lceil \log_2(2p_{max}) \rceil)$ [18]. Thus the computation of the phase matrix runs in time polynomial, and need only be calculated once and memoized. Moving now onto the main body of the algorithm, it can be seen from its formulation that in the worst case it is exponential in the number of tasks n , with complexity $O(2^n)$. However, the running time of the algorithm is clearly independent of the choice of task periods and p_{max} ; for any fixed n , $|2^n|$ is constant; LCS therefore constitutes a polynomial-time algorithm for the search version of TSAFP for fixed n . If n is less than or equal to the binary bit-width of the solving machine, then as mentioned previously the application of equation (13) can be performed with a single binary conjunction instruction; the running time of the algorithm can be described as follows:

$$f(n) \cdot O(1) \tag{16}$$

where $f(n)$ is equal to 2^n . LCS thus constitutes a FPT algorithm for TSAFP, in the sense described by Niedermeier [25]. In the more general case, it can be observed that LCS may also be applied to solve the underlying SIMULTANEOUS CONGRUENCES problem; when n is larger than the bit width of the solving machine, the running time will be proportional to $f(n) \cdot n$, which is also FPT for a given n . As will be subsequently demonstrated, for values of $n \leq 30$ —such as for the real-world task sets considered in this paper—the LCS algorithm provides an exponential running time improvement over hyperperiod simulation. The adoption of LCS has allowed for deeper investigations into the offset assignment (TSOAP) problem; attention is now focused on this problem in the following Section.

4. Algorithms for TSOAP

A simple, exact algorithm for the TSOAP problem would be as follows: enumerate and search all possible offset combinations, using the LCS algorithm to determine the optimal value of c_{Max} . This can be improved upon by generating an initial heuristic solution of reasonable quality, and using this solution to guide a simple back stepping search, similar to the procedure outlined in [15]. Further improvements can be gleaned by omitting redundant configurations of offsets from the search; this is discussed in the next Section.

4.1. Breaking Symmetry

Symmetry, in the current context, occurs when two sets of task offsets O and O' result in identical periodic behavior over the hyperperiod h , the only difference being the relative start point of the schedules at $t = 0$. In such cases, the value of c_{Max} for both schedules is identical. A simple example can illustrate the concept of symmetry: suppose we have two tasks with periods $\{3,5\}$. Although offsets in the range 0–2 and 0–4 can seemingly be considered, the periods are coprime; all choices of offset are in fact identical and result in the same periodic behavior, shifted only at the origin. Obviously, to reduce the search space it is wished to only consider one set of offsets in situations where symmetry exists.

Previous work in the area of offset-free preemptive scheduling by Goossens [11] has shown that in most periodic task system in which one is free to assign offsets, many classes of equivalent offset combinations exist. It is clear from the proofs developed in this previous work (specifically Theorem 9) that this can be also be shown to hold for the case of the offset-free non-preemptive thrift scheduler. [11] also outlined a method to break symmetries as part of a backstepping search algorithm; this can be adapted to develop a simple polynomial-time symmetry breaking algorithm, which is called once (and once only) before a search begins. Also, this method does not require the manipulation of any large integers—which in practice causes integer overflow even for moderately low n —and is also somewhat faster in practice.

Let p' be the *phase capacity* of a task, which is used to place an upper bound on the choice of its offset, such that if offsets for task i are selected in the range $0 \leq o_i < p'$ then all (and only) redundant configurations of offsets will be removed from the search. Since the evaluation of each offset configuration in the search space requires the solution of a coNP-Hard problem, this clearly has many benefits. For the first task, p' is always zero; this follows from Theorem 6 in [11]. For all $1 < i \leq n$, p' can be calculated as the *lcm* of the pairwise *gcd*'s between task i 's period, and all other periods less than i :

$$p_i' = lcm(\gcd(p_i, p_1) \dots \gcd(p_i, p_j)) \forall j, j < i \quad (17)$$

Theorem 4: Considering task offsets upper bounded by p' only removes all (and only) redundant offset configurations from a search.

Proof: Trivially follows from Theorem 10 in [11], the definition of p' given by (17) and the basic numerical properties of gcd and lcm .

Given the complexity results outlined in Section 2, it is unlikely that even small instances of the offset assignment problem may be solved exactly in a tractable time. In order to obtain relatively quick 'good' solutions to the problem, heuristics can be employed. Given the similarity of the considered problem to the multiprocessor scheduling problem, it follows that algorithms that have been useful in this area may also be applied in the current context. Before describing two such algorithms, the notion of symmetry will be extended further. In previous papers, the potential effects of symmetry on the hyperperiod of the task set have not been considered. From (12) and (13), it is clear that only the relative offsets of individual tasks define the subsets of tasks that are congruent. Thus, let the *harmonic period* h' of a task i be defined as follows:

$$h_i = lcm(gcd(p_i, p_1) \dots gcd(p_i, p_j)) \forall j, j \neq i \quad (18)$$

This is a simple extension of the notion of phase capacity. Suppose we now consider a *reduced task set* τ' , in which the tasks are identical to those in τ , with the exception that each and every period p is replaced by h . Such a reduced task set would then have a *reduced hyperperiod* h_r defined as follows:

$$h_r = lcm(h_1, \dots, h_n) \quad (19)$$

Equation (18) defines a hyperperiod that contains no symmetries; since $h_r \leq h$ (and in most cases $h_r \ll h$) an alternative formulation to decide TSAFP would be to perform hyperperiod simulation using τ' . To show that such an algorithm is proper, from (12) and (13) it would need to be shown that τ' contains all (and only) those task congruences that are present in τ .

Theorem 5: τ' contains all (and only) those task congruences that are present in τ .

Proof: By showing that $gcd(p_1, p_2) = gcd(h_1, h_2)$. Let $x = gcd(p_1, p_2)$ and $y = gcd(h_1, h_2)$. From (18) and by the definition of lcm , we have $h_1 = ax$ and $h_2 = bx$, where a and b are some positive (non-zero) co-prime integers. Then we have $y = gcd(ax, bx)$; since a and b are co-prime, by definition $gcd(ax, bx) = x$. Since we have that in fact $y = x$, the Theorem is proved.

This shows that all task congruences are preserved by using such a method, and no further congruences are introduced. It can be noted that this also gives a fast and effective way to detect instances which can be decided rapidly using traditional means, for example equivalent synchronous systems; *i.e.*, those in which all periods are co-prime, or task sets with high degrees of harmonic coupling. The LCS algorithm can be modified to detect such instances as follows: if $nh' \ll 2^n$ then hyperperiod simulation is used; else the procedure defined in Section 3 is used. However, early experience with LCS indicates that in the average case, a more suitable test can be $nh' < np_{Max}$, where p_{Max} is the largest of the task periods.

4.2. The MULTIFIT Algorithm

In [10], the authors suggest a basic heuristic for offset assignment, achieved by sorting the list by non-decreasing period before application of a basic First-Fit (FF) algorithm under the assumption that each successive task is ‘easier’ to fit around the previous tasks. Given the analysis of the previous Section, it is clear that a better option would be to order the tasks by non-decreasing harmonic period; this leads to the First-Fit-Phase (FFP) algorithm. FFP first performs such a sort, then each task is assigned—one by one—the lowest indexed offset that will schedule it, until either all task are scheduled (returning TRUE) or one or more tasks couldn’t be scheduled (returning FALSE). The LCS algorithm is used for the feasibility test, and to reduce the search space only offsets bounded by each tasks phase capacity p' are considered. The MULTIFIT algorithm proposed in this paper uses FFP as a key component in a binary search; MULTIFIT for the classical multiprocessor scheduling problem was originally proposed in [13], and it has been proven that the resulting makespan can be no further than 22% away from the optimal makespan. The modified algorithm considered in this paper is very similar; a binary search is performed to find the minimum value of α such that applying FFP schedules the task set. Since FFP (and hence LCS) are called multiple times, the gcd 's of each pair of tasks are computed once only and stored in an array, and sorting is performed once only. The initial upper bound is set to the sum of tasks execution times, and the lower to the largest execution time in the task set. The MULTIFIT algorithm proposed in this paper is shown in Figure 6.

Figure 6. MULTIFIT algorithm, in pseudo-code.

```

01 procedure MULTIFIT( $\tau$ ,  $n$ )
02 {
03    $u := \text{sum}(c_i), i \in \tau;$       // Set upper bound
04    $l := \text{max}(c_i), i \in \tau;$   // Set lower bound
05   Sort( $\tau$ );                  // Sort by phase capacity
06   WHILE  $u \geq l$               // Binary search loop
07      $c := (u + l) / 2;$         // Take the midpoint
08      $t := \text{FFP}(\tau, c);$     // Attempt FFP at this value
09     IF  $t = \text{TRUE}$ 
10        $u := c;$               // Set new upper bound
11        $O = o_i, i \in \tau;$     // Store solution
12     ELSE
13        $l := c + 1;$           // Raise lower bound
14   END IF
15 END WHILE
16 Return( $u, O$ )

```

As mentioned, for fixed n LCS runs in polynomial time; hence for fixed n , MULTIFIT has complexity $O(np'_{max})$, i.e., pseudo-polynomial. The second algorithm to be described is based upon a local-search version of the classical LPT algorithm described by Graham [17].

4.3. The SWAPFIT Algorithm

The LPT algorithm for multiprocessor scheduling first sorts the tasks in order of decreasing execution time, and assigns the tasks one by one onto the first available processor; the latter operation is known as list processing. It has been proven that the makespan resulting from LPT is no further than 33% from optimal. The rationale behind SWAPFIT is simple; there is a permutation of the tasks such that application of list processing produces an optimal makespan, and in [17], it is shown that the list produced by sorting for non-increasing execution times produces a permutation that is very close to optimal. Starting from a good, initial solution, an attempt is made to sort the list into a better order, guided by resulting changes in the value of the objective function. In the SWAPFIT algorithm, list processing is modified as follows; each task in the list is assigned, one by one, an offset (bounded by p') that minimizes the worst-case release of the task. LPT is initially employed to obtain a starting solution. Next, every pairwise set of tasks in the list is reversed, one by one; if such a reverse causes the objective function to decrease when the tasks are list processed, it is accepted. If not, the change is rejected, and the tasks swapped back. The SWAPFIT algorithm is shown in Figure 7.

Figure 7. SWAPFIT algorithm, in pseudo-code.

```

01 procedure SWAPFIT( $\tau$ , n)
02 {
03   r := TRUE;           // Set repeat flag
04   k := 0;
05   Sort( $\tau$ );           // Sort
06   b := List( $\tau$ );      // Set initial incumbent
07   WHILE r = TRUE AND k < n DO // Loop until limit
08     r := FALSE;        // Reset flag
09     FOR i := 1 TO n-1 DO
10       FOR j := i TO n DO
11         Swap(i, j);    // Pairwise swap
12         b' := List( $\tau$ ); // Check for improvement
13         IF b' < b
14           b := b';     // Improved solution
15           O = oi, i ∈  $\tau$ ; // Store solution
16           r := TRUE;   // Repeat again
17         ELSE
18           Swap(i, j);  // Go back to previous list
19         END IF
20       END FOR
21     END FOR
22     k := k + 1;      // Increment loop counter
23   END WHILE
24   Return(b, O)

```

For SWAPFIT, it can be seen that the inner loop of pairwise reversals performs approximately $0.5n^2$ reverses and list processes, and this process continues until either no more beneficial changes can be made, or an upper bound on allowed iterations is reached. As with most local search techniques, no sub-exponential bound can be placed on the number of iterations of this outer loop; in practice, this seems proportional to (but significantly less than) n . By setting an upper bound as n and simply returning the best solution found if this bound is exceeded, the algorithm again runs in pseudo-

polynomial time for fixed n , with complexity $O(\max(n^3, np'_{max}))$. Since LCS is called multiple times, the gcd 's of each pair of tasks are again computed once only and stored in an array.

Before moving onto describing the computational experiments, it is worthwhile discussing performance bounds for the two algorithms. The quoted performance bounds for the original versions of both MULTIFIT and LPT are known to be extremely tight for multiprocessor scheduling [13,17]. However, it is clear that the performance bounds do *not* hold for the current algorithms. When periods are evenly divisible, it can be seen that FFP results in a packing that is identical to First Fit (FF); however for general problem instances, no bounds are currently known. That said, it can be conjectured that both heuristics will produce a 'good' schedule; exactly how good this schedule is likely to be for general instances of the problem is investigated in Section 5. A general lower bound (z_l) for the optimal value of general problem instances can be obtained through (20):

$$z_l = \max(u \cdot t_{Tick}, (\max(c_i) i \in \tau)) \quad (20)$$

where u is the task set utilization. However it can be observed that this bound can be arbitrarily far from the true value, simply by considering subsets of tasks with co-prime periods; it follows from the Chinese remainder theorem that no choice of task offsets can avoid them being congruent. In many cases a tighter bound can be obtained by employing LCS to determine the largest co-prime task phasing. This observation leads to an alternate formulation for exact solutions to the problem; the Iteratively Tightening Lower Bound (ITLB) algorithm.

4.4. The ITLB Algorithm

The ITLB algorithm builds from the fact that in any schedule, there is a task 'bottleneck' that defines the optimal value achievable for the problem instance. This bottleneck is most likely to be between tasks whose pairwise gcd 's are low; for example, a subset of tasks with co-prime periods. The ITLB algorithm works as follows; an initial incumbent solution b is found using SWAPFIT, and the tasks are then sorted according to non-decreasing harmonic period; tasks with low harmonic period are the most difficult to separate. Then, starting with low i , the first i tasks are solved to optimality, with backstepping guided by b . The optimal value l found by this process constitutes a lower bound on z ; as the number of tasks is increased above i , it is clear to see that l cannot possibly decrease. Given the bound l , and a set of offsets O' for the first i tasks, the algorithm then attempts to schedule the remaining $n-i$ tasks using SWAPFIT, such that the optimal value does not increase. If this can be achieved, then a provably optimal schedule has been found; if it cannot, i is incremented to deepen the search, and b and l are updated if a better solution or tighter lower bound have been found, and the process repeated. The ITLB algorithm is shown in Figure 8.

The call to Solve() on line 8 finds the best solution to the first i tasks; while the call to Fathom() on line 9 applies the heuristic to schedule the remaining tasks. It can be seen that with every iteration the lower bound is non-decreasing, and the incumbent is potentially improved; the iterations repeat until optimality is proved. In the worst case, every possible offset combination is enumerated and tested, and it is clear to see that the problem is solved exactly. However, it is also clear that the algorithm may be terminated at any point, and the best know solution returned; in addition, the deviation from this solution from the best known lower bound can also be returned. Thus, unlike regular backstepping

algorithms for this and similar problems, ITLB constitutes a *complete anytime* algorithm for TSOAP. The following Section presents the performance data for the described algorithms.

Figure 8. ITLB algorithm, in pseudo-code.

```

01 procedure ITLB( $\tau$ , n)
02 {
03 {b, O} := SWAPFIT( $\tau$ , n); // Get a good initial solution
04 l := max( $c_i$ ),  $i \in \tau$ ; // Get initial lower bound
05 i := 2; // Set initial search depth
06 Sort( $\tau$ ); // Sort by harmonic period
07 WHILE b > l DO
08 {l, O'} := Solve( $\tau$ , i, b);
09 {b', S} := Fathom( $\tau$ , O', i); // Attempt to fathom problem
10 IF b' < b
11 {b, O} := {b', S}; // Store better solution
12 END IF
13 i := i + 1; // Increment search depth
14 END WHILE
15 Return(l, O); // Return the solution
16 }

```

5. Performance Analysis

This Section describes a series of computational studies that were performed to gauge both the execution time and performance of the proposed algorithms. In each of the tests described in this section, a ‘typical’ desktop PC setup was employed to perform the assessments. The hardware used was standard off-the-shelf office computing equipment. The PC was based around an Intel® Core-2 Duo® processor running at 2.13 GHz, with 1 GB of RAM. The operating system employed was Windows® XP, and all software was written in C++ using the Borland® C++ Builder package, and compiled to favor speed. Timing measurements were taken using the Pentium® performance counter.

5.1. TSAFP Algorithm Testing

The LCS algorithm was implemented and first tested in comparison to direct hyperperiod simulation. In order to keep the value of h tractable task periods were limited to the interval [1, 100] ms, and the number of tasks restricted to 10; this ensured each individual test could be performed in less than 20 minutes on the target hardware. In total 1,000 different representative task sets were generated as follows; periods were randomly generated, and assigned to tasks. Following this, (3) was used to determine the tick interval, and (17) applied to calculate p' . Offsets were then randomly assigned within this bound. Task computation times were then assigned in the interval $[0.1 t_{Tick}, t_{Tick}]$ μ s. The resulting task sets were then used as input to both LCS and a hyperperiod simulation algorithm, and the resulting computation times for each are shown in Figures 9 and 10. A summary of the results is given in Table 1, with units of seconds.

Following this, a number of tests were performed to gauge the increase in complexity of the proposed algorithm as the number of tasks in the input set was increased. The restriction on task

periods was lifted for these tests, and task periods were generated in the interval [1, 1,000] ms. Starting from $n = 5$, the number of tasks was increased in steps of 5 up to and including $n = 30$ tasks. At each step, 100,000 task sets were randomly generated and used as input to LCS. The results of these tests are given in Table 2, with units of seconds, and are shown graphically in Figure 11.

Figure 9. Hyperperiod simulation performance.

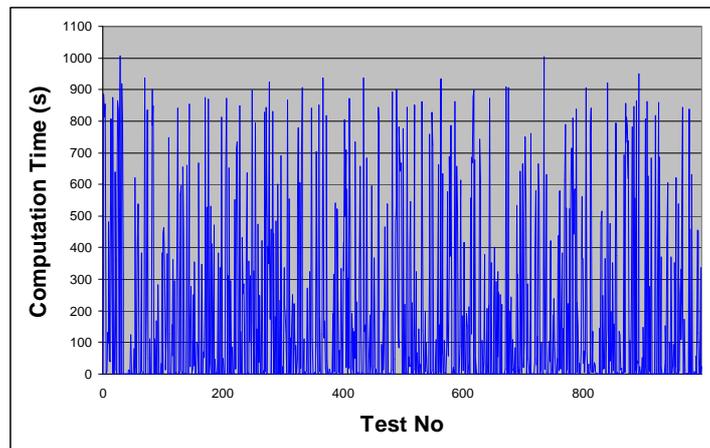


Figure 10. LCS algorithm performance.

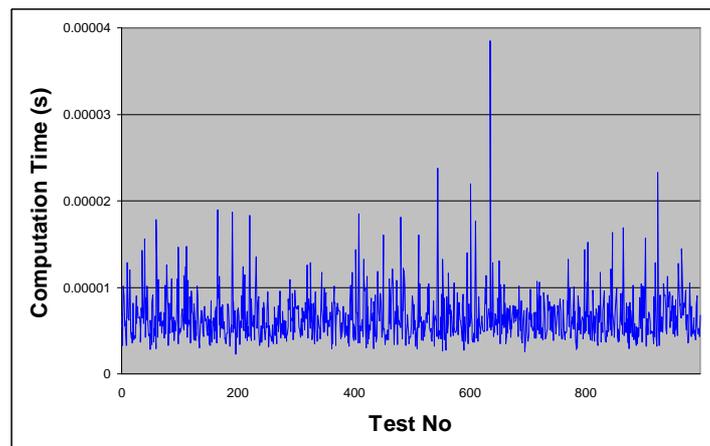


Table 1. Comparative results of computation times.

	LCS	Hyperperiod
Min	0.0000023	0.0000218
Max	0.0000385	1,007.2945080
Ave	0.0000066	188.4035393

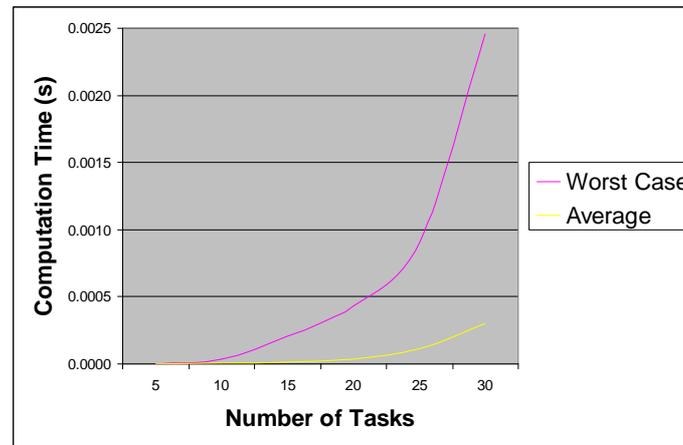
It can be seen from Table 1 that in all cases, the LCS method was significantly faster than basic hyperperiod simulation for solving TSASP. As can be seen from the results, LCS is extremely robust in terms of the task parameters; in comparison, hyperperiod simulation sees huge fluctuations in computation time as the task parameters vary, in some cases taking in excess of 16 minutes to solve these vastly restricted instances. Considering now the results of Table 2, it can be seen that as the number of tasks increases, the worst case computation time reflects the exponential in n ; however,

even given this complexity rise the worst case recorded for $n = 30$ was under 400 μ s. It should be noted that reduced hyperperiod simulation (as outlined in Section 4.1) was employed on average less than 1% of the time. Despite this exponential rise, the LCS algorithm in its current formulation is still extremely efficient at this level, and gives a huge gain in efficiency over basic hyperperiod simulation. For example, with $n = 30$ and periods in the range 1–1,000 ms, out of 100 randomly generated instances, not a single one could be solved in under 1 hrs computation time. Extrapolating the results from Table 1—using the calculated length of h —indicated that most would take around 10^{70} Centuries to solve.

Table 2. Effect of increasing task numbers.

N	Max	Average
5	0.00000166	0.00000126
10	0.00003905	0.00000628
15	0.00021083	0.0000111
20	0.00043122	0.00003358
25	0.00090671	0.00011151
30	0.00245345	0.00029952

Figure 11. Running time increase.



5.2. TSOAP Algorithm Testing

The MULTIFIT and SWAPFIT algorithms were tested both in terms of their computation time and the quality of the schedule they produced. First, task sets were generated as per the previous Section. For each value of n 1,000 task sets were considered, and the computation time required for both algorithms on each instance was recorded. A summary of the results is given in Table 3 and Figures 12 and 13, with units of seconds. It was observed that in every instance, the value of the objective function found by SWAPFIT (z_s) was \leq that found by MULTIFIT (z_m), and in most cases, significantly lower.

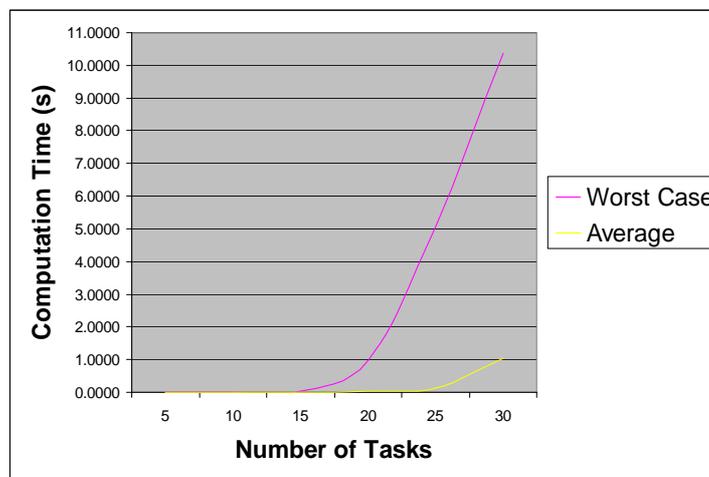
Following this, in order to gauge the performance of each algorithm the instances were solved exactly using the ITLB algorithm outlined in Section 4. It should be noted that even with broken symmetries and the application of the ITLB algorithm, computation times varied considerably. A 30

minute computation time limit was placed on each instance. Every instance with $n \leq 15$ was solved exactly within this bound; in most cases taking considerably less time, a significant achievement given the complexity of the problem. For instances with $n > 15$, approximately 30% of instances could not be solved exactly within this time bound; however in each of these cases a solution that was probably within 5% of optimality was returned. The best know lower bound was used for the performance comparison. The results allowed the performance of each of the heuristic algorithms to be further analyzed. Table 4 shows the average and maximum deviation from optimality for both algorithms (as a percentage) in these selected instances.

Table 3. Summary of algorithm running times (s).

N	SWAPFIT		MULTIFIT	
	Max	Ave	Max	Ave
5	0.00068	0.000009	0.000766	0.000018
10	0.01792	0.000836	0.005827	0.000243
15	0.152797	0.011035	0.042646	0.001635
20	0.84614	0.058131	0.988282	0.016758
25	4.81073	0.231637	5.02838	0.120799
30	8.99549	0.780521	10.3763	1.04288

Figure 12. MULTIFIT running times.



It can be seen from these Tables and Figures that the running time for SWAPFIT was competitive with MULTIFIT, having a marginally better average and worst case running time in most instances. However, whilst SWAPFIT consistently found optimal schedules, and was never further than 4.68% from optimality, with an average deviation of 0.11%, the same could not be said for MULTIFIT. The average deviation was 22.36% for the latter algorithm, with deviations as large as 63.63% being observed. In summary, the SWAPFIT algorithm seems to be far more consistent than MULTIFIT both in terms of its running time and the resulting solution quality; the algorithm seems to capture the essence of the problem much better. Although SWAPFIT can only be considered a local search algorithm, it can be observed that a single pair-wise reversal of tasks in a list has the potential to radically change the entire resulting schedule; this ability to widely explore the search space whilst

honing in on a good solution may explain its excellent performance. As a final point, it can also be observed that since task computation times are often over-estimated by as much as 5%, the reported bounds for SWAPFIT and the errors for unsolved ITLB instances can effectively be neglected in most cases, as they are a far smaller source of potential error.

Figure 13. SWAPFIT running times.

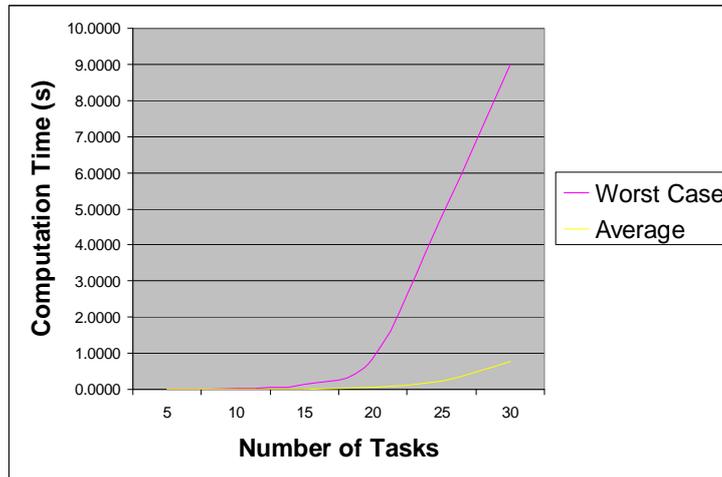


Table 4. Summary of algorithm performance (% deviation from best know lower bound).

N	SWAPFIT		MULTIFIT	
	Max	Ave	Max	Ave
5	0.00	0.00	41.27	1.66
10	3.64	0.04	48.66	13.56
15	2.07	0.04	58.20	23.88
20	3.75	0.10	54.16	27.06
25	3.84	0.16	59.22	33.24
30	4.68	0.31	63.63	36.36

6. Conclusions

In this paper, the problem of assigning offsets to tasks in order to minimize worst-case congruent release of tasks in a thrift scheduler has been investigated, and the complexity of the problem has been shown. Several new approaches to solving the problem have been formulated; in practice both the time complexity and the performance of these algorithms seems acceptable for real-world task instances. In particular, the LCS algorithm, in conjunction with the SWAPFIT offset assignment algorithm, very quickly produces a schedule that seems to be either optimal or extremely close to it. It can also be noted that the ITLB algorithm also performs well, and may be used by system developers who wish to obtain a provably good solution within a certain time limit; initial experience indicates that even for larger instances, a solution that is within 10% of optimal can be found within 5 minutes. However, although the MULTIFIT algorithm performs well in its original problem domain, the same cannot be said for the modified version employed in this paper. An additional benefit of the algorithms developed in this paper is that they are guaranteed to find a schedule; many current tools to support

thrift cyclic scheduling (and its close relatives) provide no such guarantee. Further work in this area will include deeper investigations into the performance of SWAPFIT, with further experiments employed to classify its performance. Improvements to the ITLB algorithm will also be investigated, along with the possibility of formulating an exact algorithm in which the number of offsets to be considered is proportional to $O(2^n)$. Further work will also consider algorithms for problem instances in which $n > 30$.

In conclusion, it has been shown that for most real-world instances of the described problem, it can be solved to near-optimality in a very acceptable time. The algorithms are relatively simple, allowing integration into existing development tools. Further work will investigate alternate approaches to solving the offset assignment problem, by using ILP techniques, and will further investigate the performance of the SWAPFIT algorithm, to explore its performance in other problem domains.

References

1. Buttazo, G. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*; Springer Verlag: New York, NY, USA, 2005.
2. Baker, T.P.; Shaw, A. The cyclic executive model and Ada. *Real-Time Syst.* **1989**, *1*, 7–25.
3. Locke, C.D. Software architecture for hard real-time applications: Cyclic executives vs. fixed priority executives. *Real-Time Syst.* **1992**, *4*, 37–52.
4. Xu, J.; Parnas, D.L. Fixed priority scheduling versus pre-run-time scheduling. *Real-Time Syst.* **2000**, *18*, 7–23.
5. Bate, I.J. *Scheduling and Timing Analysis for Safety Critical Real-Time Systems*. PhD. Dissertation. Department of Computer Science, University of York, Heslington, York, UK, 1998.
6. Short, M.; Pont, M.J.; 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, July 2–4, 2008; pp. 83–91.
7. Short, M.; Pont, M.J.; Fang, J. Assessment of performance and dependability in embedded control systems: methodology and case study. *Contr. Eng. Pract.* **2008**, *16*, 1293–1307.
8. Burns, A.; Hayes, N.; Richardson, M.F. Generating feasible cyclic schedules. *Contr. Eng. Pract.* **1995**, *3*, 151–162.
9. Pont, M.J. *Patterns For Time Triggered Embedded Systems*; ACM Press(Addison Wesley): New York, NY, USA, 2001.
10. Gendy, A.K.; Pont, M.J. Automatically configuring time-triggered schedulers for use with resource-constrained, single-processor embedded systems. *IEEE Trans. Ind. Inf.* **2008**, *4*, 37–45.
11. Baruah, S.K.; Rosier, L.E.; Howell, R.R. Algorithms and complexity concerning the preemptive scheduling of periodic tasks on one processor. *Real-Time Syst.* **1990**, *2*, 301–324.
12. Goossens, J. Scheduling of offset free systems. *Real-Time Syst.* **2003**, *24*, 239–258.
13. Coffman, E.G.; Garey, M.R.; Johnson, D.S. An application of Bin-Packing to multiprocessor scheduling. *SIAM J. Comput.* **1978**, *7*, 1–17.
14. Gerber, R.; Hong, S.; Saksena, M. Guaranteeing real-time requirements with resource-based calibration of periodic processes. *IEEE Trans. Softw. Eng.* **1995**, *21*, 579–592.

15. Engblom, J.A.; Ermedahlx, A.; Sjodinxz, M.; Gustafsson, J.; Hanssonx, H. Worst-case execution-time analysis for embedded real-time systems. *J. Softw. Tools Technol. Transf.* **2001**, *4*, 437–455.
16. Garey, M.R.; Johnson, D.S. *Computers and Intractability: A guide to the Theory of NP-Completeness*; W.H. Freeman: New York, NY, USA, April, 1979.
17. Graham, R.L. Bounds on multiprocessing timing anomalies. *SIAM J. Appl. Math.* **1969**, *17*, 416–429.
18. Knuth, D.E. Semi numerical algorithms. In *The Art of Computer Programming*, 2nd ed.; Addison-Wesley: Reading, MA, USA, 1973; Vol. 2.
19. Mok, A.; Rosier, L.; Tulchinsky, I.; Varvel, D. Algorithms and complexity of the periodic maintenance problem. *Microprocess. Microprogram.* **1989**, *27*, 657–664.
20. Short, M. Some complexity results concerning the non-preemptive ‘thrift’ cyclic scheduler. In *Proceedings of the 6th International Conference on Informatics in Control, Robotics and Automation (ICINCO 2009)*, Milan, Italy, July 2009; pp. 347–350.
21. Short, M. A note on the efficient scheduling of periodic information monitoring requests. *Eur. J. Oper. Res.* **2010**, *201*, 329–335.
22. Campbell, A.M.; Hardin, J.R. Vehicle minimization for periodic deliveries. *Eur. J. Oper. Res.* **2004**, *165*, 668–684.
23. Zeng, D.D.; Dror, M.; Chen, H. Efficient scheduling of periodic information monitoring requests. *Eur. J. Oper. Res.* **2006**, *173*, 583–599.
24. Leiserson, C.; Prokop H.; Randall, K.H. *Using de Bruijn Sequences to Index a 1 in a Computer Word*; MIT Technical Report; Available online: <http://supertech.csail.mit.edu/papers/debruijn.pdf> (accessed August 9, 2009).
25. Niedermeier, D. *Invitation to fixed-parameter algorithms*; Oxford University: Oxford, UK, 2006.

© 2009 by the authors; licensee Molecular Diversity Preservation International, Basel, Switzerland. This article is an open-access article distributed under the terms and conditions of the Creative Commons Attribution license (<http://creativecommons.org/licenses/by/3.0/>).