Preprint of a paper to be published in the
European Journal of Operations Research, 2009

A note on 'Efficient scheduling of periodic information monitoring requests'

Michael Short[1]

Embedded Systems Laboratory, University of Leicester,
University Road, Leicester LE1 7RH, UK

Abstract

A recently published paper by Zeng et al. ['Efficient scheduling of periodic information monitoring requests', EJOR 173, pp 583-599] (Zeng et al. 2006) considers the non-preemptive scheduling of periodic server information requests for mission-critical monitoring applications such as policing and homeland security. In their paper, it was claimed that the decision version of the considered Periodic Monitoring (PM) problem was NP-Complete, and several greedy heuristics were developed to 'efficiently' solve the problem. The standard argument of polynomial-time solution verification was employed in their complexity proof. However, the present note points out that that the PM problem is actually Σ_2^p – Complete, and verification of a PM solution is coNP-Complete, in the strong sense. A consequence of these results is that the greedy heuristics proposed by Zeng et al. are all strongly coNP-Hard, invalidating the authors' claims of efficiency; since their algorithms are implemented on-line in a mission-critical application, this clearly needs to be taken into account. The final contributions of the present note are the description of an efficient algorithm for the underlying peak server load problem, and showing that equivalence classes in request start times can be efficiently detected and pruned prior to searching. These former elements - if incorporated into the original heuristics - can potentially improve stability and efficiency by large orders of magnitude.

Keywords: Non-preemptive Scheduling, Greedy Heuristics, Periodic Queries, Complexity.

[1] Corresponding author:
Tel: +44 (0)116 252 5052
Email: mjs61@le.ac.uk (M Short)

1. Introduction

A recently published paper by Zeng et al. ['Efficient scheduling of periodic information monitoring requests', EJOR 173, pp 583-599] (Zeng et al. 2006) considers the scheduling of periodic server information requests for mission-critical monitoring applications such as homeland security. In their paper, this problem (the periodic monitoring or PM problem) was formulated as an optimization problem, with the goal of assigning integer start times to each periodic request such that the peak server load in any time unit is minimized. Formally, in a PM instance we are given a set N of n periodic database requests, each characterised by a period $p_i \in \aleph^+$ and a server demand $d_i \in \Re^+$ to be scheduled. Each request for information places a load d_i on the server every time it is serviced, and each request must be serviced *exactly* p_i time units apart; as soon as a request is ready, it is immediately serviced (in a non-preemptive fashion) within the same time unit. Given a candidate solution to PM – i.e. integer start times s_i for each request – to determine the peak server load (the objective function) it suffices to consider only a length of the schedule known as the hyperperiod h, given as follows:

$$h = lcm(p_1, \ldots, p_n) \tag{1}$$

Where *lcm* is the least common multiple of the request periods. The similarity been this problem and other related scheduling problems (primarily in real-time systems) was identified in the previous paper; in fact it is an identical scheduling model to the non-preemptive version of offset-free scheduling (Goosens 2003), using Pont's 'TTC' scheduler (Pont 2001). Theorem 1 in the previous paper claims that the decision version of the PM problem is strongly NP-Complete, and several polynomial-time greedy heuristics were subsequently developed by the authors to 'efficiently' solve the problem. The standard argument of polynomial-time verification was claimed – without any kind of analysis or support - as part of their complexity proof to show membership of the PM problem in NP. However, in this note it is argued there is an inconsistency in their complexity results; it will be shown in Section 2 that PM is actually complete for Σ_2^p, and verification of a PM solution is strongly coNP-Complete. A consequence of this result is that the heuristics proposed by Zeng. et al. are in fact strongly coNP-Hard. In order to help ameliorate this problem, Section 3 of the present note formulates an efficient algorithm (the Largest Congruent Subset or LCS algorithm) to help overcome this verification problem; for any *fixed* number of requests it runs in polynomial-time. This Section also shows how symmetries – in terms of equivalence classes in the request starting times - can be efficiently pruned from the search. In

combination, these two elements would seem to make the application of the heuristics proposed by Zeng et al. tractable for low / medium sized problem instances.

2. Complexity of the PM problem

Consider first the verification of a solution to PM, which will be termed the peak server demand (PSD) problem. Formally, in the decision version of PSD we have an instance of PM along with an associated start time $s_i \in \aleph_0^+$ for each request, and a bound on the peak server demand $b \in \Re^+$. W.l.o.g., the start times can be assumed to satisfy the following (Goosens 2003; Zeng et al. 2006):

$$\forall i \in N; 0 \leq s_i < p_i \tag{2}$$

The question that is asked is as follows: is the peak demand on the server less than or equal to b? In Appendix 1 of the Zeng et al. paper, it is claimed that "Using the standard polynomial-time verification argument, we can easily prove that D-PM is in NP", implying that an instance of PSD can be solved in polynomial-time. To refute this claim, PSD will now be shown to be coNP-Complete. Membership of the problem in coNP can be seen as follows: each request in an instance of PSD can be represented as a congruence of the form:

$$x \equiv s_i \bmod(p_i) \tag{3}$$

In which x is one of the (multiple) possible solutions to the congruence. A 'No' answer for an instance of PSD results in counter example, i.e. a time interval j in which the server demand is greater than b. The demand in the j^{th} time instant can be determined in time proportional to n by summing the demands for each periodic request that is congruent to j, satisfying $x = j$ in (3). If PSD is polynomially verifiable, then a 'Yes' answer would seemingly require a similar certificate (Garey & Johnson 1979). However, since there seems to be no sub-exponential bound on the length of h given by (1), this would seem to be unlikely. To show that PSD is in fact coNP-Complete, the Simultaneous Congruences Problem (SCP) is now introduced. SCP is known to be NP-Complete, in the strong sense (Baruah et al. 1990).

SIMULTANEOUS CONGRUENCES PROBLEM (SCP)

Instance: A set A of ordered integer pairs $\{(x_1, y_1) \ldots (x_n, y_n)\}$ and a positive integer[2] $2 < k \leq n$.

Question: Is there a subset $A' \subseteq A$ of k ordered pairs, and a positive integer z, such that for all $(x_i, y_i) \in A'$, $z \equiv x_i \bmod(y_i)$?

Theorem 1: PSD is strongly coNP-Complete.

Proof: Transformation from the compliment of SCP.

Let $\phi = < (x_1, y_1) \ldots (x_n, y_n), k >$ denote an arbitrary instance of SCP. From this we create a set N of n requests for an instance of PSD, with a bound b equal to k-1, as follows:

$$\begin{aligned} p_i &= x_i \\ d_i &= 1 \\ s_i &= y_i \end{aligned} \tag{4}$$

This transformation can be performed in time proportional to n and is hence polynomial. Next, it is argued that a positive solution to ϕ exists iff there is a negative answer to PSD. If PSD is negative, then it implies that during at (at least) one time interval j, the server demand is greater than b. Since b is 1 less than k, at least k requests must be simultaneously active; this gives a solution to ϕ with a certificate j. Conversely, if the answer to SCP is positive, then the peak server demand is $\leq b$ and a time interval in which k or more requests are simultaneously active does not exist; implying a negative answer to ϕ. It can also be observed that the largest integer resulting from this transformation is no larger than the largest integer in the original instance; and since SCP is strongly NP-Complete, the Theorem is proved. □

Given this result, it would seem that a proof of PSD's polynomial solvability as proposed by Zeng et al. would constitute a proof that P = coNP, and seems unlikely. Since there also seems to be an exponential number of possible start times for a PM instance, under the assumption that P ≠ NP it is also worthwhile investigating exactly where the PM problem lies on the so-called 'polynomial hierarchy' (Garey & Johnson 1979). From (2), it can be seen that the request start times for 'Yes' instances of this problem can be encoded in a number of bits that is less than or equal to the request periods, and hence the size of the problem instance. Given the previous Theorem, the resulting

[2] In the case when $k \leq 2$, the problem can be solved in polynomial time.

request schedule is verifiable in polynomial time by a Turing machine with an oracle for the PSD problem; the problem resides in Σ_2^p. To show that the problem is complete for this complexity class, the Periodic Maintenance Scheduling Problem (PMSP) is now introduced. This problem is known to be Σ_2^p – Complete (Mok et al. 1989; Baruah et al. 1990).

PERIODIC MAINTENANCE SCHEDULING PROBLEM (PMSP)

Instance: A set C of ordered pairs $\{(p_1, c_1) \ldots (p_n, c_n)\}$, with each c_i representing a maintenance activity having an integer period p_i, positive integer[3] $1 < k \leq n$.

Question: Is there a mapping of the activities in C to positive integer time slots such that successive occurrences of each c_i are exactly p_i time slots apart, and no more than k activities ever collide in a single slot?

Theorem 2: PM is Σ_2^p - Complete.

Proof: Transformation from PMSP.

Let $C = < ((c_1, p_1) \ldots (c_n, p_n), k >$ denote an arbitrary instance of PMSP. From this a set N of n requests to be scheduled by PM are created, and the bound b is set equal to k:

$$\begin{aligned} p_i &= p_i \\ d_i &= 1 \end{aligned} \tag{5}$$

Again this transformation can be performed in polynomial time. Next, it is argued that a solution to M exists iff N can be scheduled by PM. If the answer to this instance of PM is 'Yes', this implies that start times can be assigned to each request in N such that the peak demand is $\leq b$, implying that a maintenance schedule for C - in which no more than k activities occur simultaneously - also exists with a certificate $(s_1 \ldots s_n)$. Conversely, if the answer to PM is 'No', then a schedule in which the peak demand is $\leq b$ does not exist for any combination of request start times, implying that a maintenance schedule for C - in which no more than k activities occur simultaneously - does not exist. Since PMSP is Σ_2^p - Complete, the Theorem is proved. □

[3] For the special case when $k = 1$, a PMSP solution becomes polynomially verifiable but the problem remains strongly NP – Complete.

The consequences of these results are as follows; the greedy heuristics proposed in the Zeng et al. paper all require the solution of PSD instances multiple times during their operation (Page 590, Step 2.2). Since this is a coNP-Hard problem, this bound also applies to the proposed heuristics. As the authors' report that the greedy scheduling policies have been implemented on-line – as part of a mission critical system serving a major US police force – this clearly must be taken into account.

At this point it is worth mentioning that there are special cases in which PSD *can* be efficiently determined. Consider an instance in which all request periods are restricted to be integer powers of 2; PSD for such an instance can be solved in time proportional to the maximum request period, i.e. in pseudo-polynomial time. However even with such a period restriction, the PM problem remains strongly NP-Hard. Another potential restriction that may be considered is to place an upper bound on the choice of periods – say in the interval [1, 50] – however this still results in a length of h that can potentially approach 10^{38}.

It can be observed then, that the actual time complexity of deciding a given PSD problem instance is incredibly fragile, being highly susceptible to the choice of request periods; this point is illustrated further in Section 4. It is worth noting that every problem instance described in the computational study performed by Zeng et al. was restricted to be either a 'power two' or 'restricted period' instance, in *all* cases resulting in a maximum length of h of 64 time units; this perhaps leads to the erroneous conclusion that the PM problem resides in NP. Unfortunately, given the nature of the problem, this is wholly unrepresentative of practical situations. As noted by Zeng et al. (2006), instances of the PM problem are likely to consist of thousands of client requests – perhaps distributed over several states or time zones – with vastly differing period requirements ranging from one or more hours, to perhaps several days (or even weeks). By way of example, suppose time is represented in hours and users may request information with periods in the interval [1, 1000]. It is trivial to create instances in which number of requests is < 50, and the length of h exceeds 10^{100}. Should such an instance be input to the system, it will surely fail.

An alternative approach to brute-force hyperperiod simulation for PSD is to look for a more efficient formulation to determine the peak server load, and to avoid – as much as possible – the calls to this procedure. This is the subject of the following Section.

3. Improved algorithms for PM

3.1 Tackling the PSD problem

It can be seen from (1) that the length of the hyperperiod that needs to be examined is potentially proportional to the product of the request periods, a largely undesirable situation. The basis for a new algorithm starts from a single observation; as mentioned in Section 1, when requests become due they are immediately serviced in a non-preemptive fashion. Therefore, in a system with n requests, there is a maximum of 2^n possible combinations of request phasings that may occur *in any single time unit*; over the hyperperiod, the actual combinations that will appear clearly depends on the request start times. For almost all instances of the unrestricted PSD problem, it is clear that $2^n << h$; this forms the basis for the LCS algorithm. According to the *linear congruence theorem*, two requests will occur simultaneously in some time interval in h iff:

$$(s_i - s_j) \mid \gcd(p_i, p_j) \tag{6}$$

That is, if the relative start times of the requests divides the greatest common divisor (*gcd*) of the request periods. Suppose now that we wish to consider the congruence of a set M of m requests. According to the *generalised Chinese remainder theorem* (Knuth 1973), (6) can be extended such that a set of m requests will occur simultaneously in some time interval in h iff:

$$\forall i, j \in M; i \neq j; (s_i - s_j) \mid \gcd(p_i, p_j) \tag{7}$$

That is, if each pair wise combination of requests in the set M satisfies (6) simultaneously. Equations (6) and (7) form the basis for the LCS algorithm. A *congruent subset* is defined as a set $T \subseteq N$ s.t. equation (7) holds. Let $|T|$ be the cardinality of the subset, i.e. the number of requests it contains, and let (T) refer to the magnitude of the subset:

$$(T) = \sum_{i \in T} d_i \tag{8}$$

The purpose of the LCS algorithm is to search for the congruent subset of requests with the largest magnitude, given an instance of PSD. Central to the proposed algorithm is the notion of a *phase*

matrix ϕ. This is an *n*-by-*n* matrix that contains all pairwise request 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 requests *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 (7) to each pair wise combination of requests in the set. Let each row of this matrix be referred to as a 'phase code', and represented as an *n*-bit long binary string; the phase code P^j corresponds to the j^{th} row of the phase matrix ϕ.

The LCS algorithm operates as a depth-first search of all possible request phasings, employing simple bounding heuristics to help prune the search. The inputs to the algorithm are a set of *n* periodic requests with start times, and the output of the algorithm is the determined peak demand. The first step is the generation of the phase matrix. The initial incumbent solution value *b* is set to the maximum of the largest single or pair-wise congruent request demand. The algorithm then begins to recursively search for congruent subsets of requests; the main elements of the algorithm are shown in Figure 1.

Each node of the search tree represents a subset *T* of congruent requests; 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} request 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 (7) in one single operation, and the only bits that will subsequently be set to '1' in *P* thus correspond to requests that are properly congruent with all requests 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.

```
01 procedure LCS(N, n)
02 {
03 b := max(d_i), i ∈ N;    // Set initial incumbent
04 FOR i := 1 TO n-1 DO    // Generate each row of the phase matrix
05    P^i = {0};
06    FOR j := (i + 1) TO n DO
07       IF (s_i – s_j) mod (gcd(p_i, p_j)) = '0'   // Test for congruence
08          P^i_j = '1';
09          b := max(b, (d_i + d_j));            // Update incumbent
10       END IF
11    END FOR
12 END FOR
13 FOR i: = 1 TO (n - 2) DO
14    Expand(i, 0, P^i);    // Expand each request
```

```
15 END FOR
16 RETURN(b);       // Return incumbent
17 }

18 procedure Expand(i, T', P')
19 {
20 P := P' & P^i;     // Generate new phase code
21 j := i + 1;       // Set initial request to check
22 T := T' + i;      // Add request i to the set
23 b := max(b, (T));  // Update incumbent
24 WHILE P ≠ {0} AND ((T) + b_up) > b DO
25    {
26    IF (P_j = '1') // Test for request j's congruence with T
27       {
28       Expand(j, T, P);   // Expand the new child node
29       P_j := '0'; // Mark the node as visited
30       }
31    j:= j + 1;    // Try the next request
32    }
33 END WHILE
34 }
```

Figure 1. LCS Algorithm

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} request 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:

$$b_{up} = \sum_{i=j}^{n} d_i \tag{9}$$

This bound expresses the fact that the best possible situation - in which *all* requests 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. It can be seen that with a set of n requests, there are exactly 0.5 n x n-1 combinations of request pairings. To generate the matrix, each pair requires an application of the *gcd* calculation and application of (7). If the periods in question can be encoded in n bits, the *gcd* algorithm has quadratic time complexity $O(n^2)$ (Knuth 1973). Thus the computation of the phase matrix runs in time polynomial. 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 requests n, with complexity $O(2^n)$. However, running time of the algorithm is clearly independent of the choice of periods; for any *fixed* n, $|2^n|$ is constant; LCS therefore constitutes a polynomial-time algorithm for the search version of PSD for fixed n. As will

be demonstrated, for values of $n \leq 60$, the algorithm is extremely fast, and also predicable; a key requirement for a mission-critical, on-line setting. As LCS is intended to be embedded in a PM search algorithm, certain elements (such as the pairwise request *gcd*'s) may be memoized to further speed execution. Since the depth of recursion is limited by n, the space requirement is polynomial in this input, and LCS can be implemented using $O(n^2)$ bits. Attention is now turned to the problem of symmetry in the choice of start times.

3.2 Breaking Symmetries

Symmetry, in the current context, occurs when two sets of request start times S and S' results in identical periodic behaviour in a schedule. In such cases, the peak demand value for both schedules is identical. Obviously, to reduce the search space it is wished to only consider one set of start times. Previous work in the area of offset-free scheduling by Goossens (2003) has shown that in most periodic task systems in which one is free to assign start times, many classes of equivalent offset combinations exist. It is clear from the proofs developed in this previous work (specifically Theorem 9) that this can also be shown to hold for the PM problem. The main results of Goosens (2003) can be adapted to the current context as follows: let p' be the *phase capacity* of a request, which is used to place an upper bound on the choice of its start time, such that if start times for request i are selected in the range $0 \leq s_i < p'$ then all (and only) redundant configurations of start times will be removed from the search. Since the evaluation of each start time configuration in the search space requires the solution of a coNP-Hard problem, this clearly has many benefits. For the first request, w.l.o.g. the start time for request 1 can be fixed at zero, i.e. $p_1' = 1$; this follows from Theorem 6 in (Goosens, 2003). For all remaining requests, $1 < i \leq n$, p_i' can be calculated as the *lcm* of the pairwise *gcd*'s between request i's period, and all request periods j less than i:

$$p_i' = lcm(\gcd(p_i, p_1) \ldots \gcd(p_i, p_j)) \forall j < i \tag{10}$$

The correctness of (10) trivially follows from Theorem 10 in Goosens (2003), and the basic numerical properties of *gcd* and *lcm*. Application of (10) can clearly be performed in polynomial time before searching begins.

4. Computational study and summary

This Section describes a series of computational studies that were performed to illustrate the fragility and complexity problems outlined in Section 2, and gauge the efficiency of the proposed LCS algorithm. 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.

The first results to be reported consider the effects of period selection on running times for both approaches to determine PSD. In order to keep the value of h tractable, the number of requests restricted to 10 with periods selected in the interval [1, 100]. In total 1,000 different representative request sets were generated as follows; periods were randomly generated, and request start times were then randomly assigned according to (10). Demands were then assigned in the interval [10, 1000]. The resulting sets of requests were then sorted in order of non-increasing d (to improve the efficiency of the bounding rule) and used as input to both the LCS and hyperperiod simulation algorithms. The resulting computation times for each are shown in Figures 2 and 3. A summary of the results is given in Table I.

Table I. Comparative results (s)

	LCS	Hyperperiod
Min	0.0000023	0.0000218
Max	0.0000385	1007.2945080
Ave	0.0000066	188.4035393

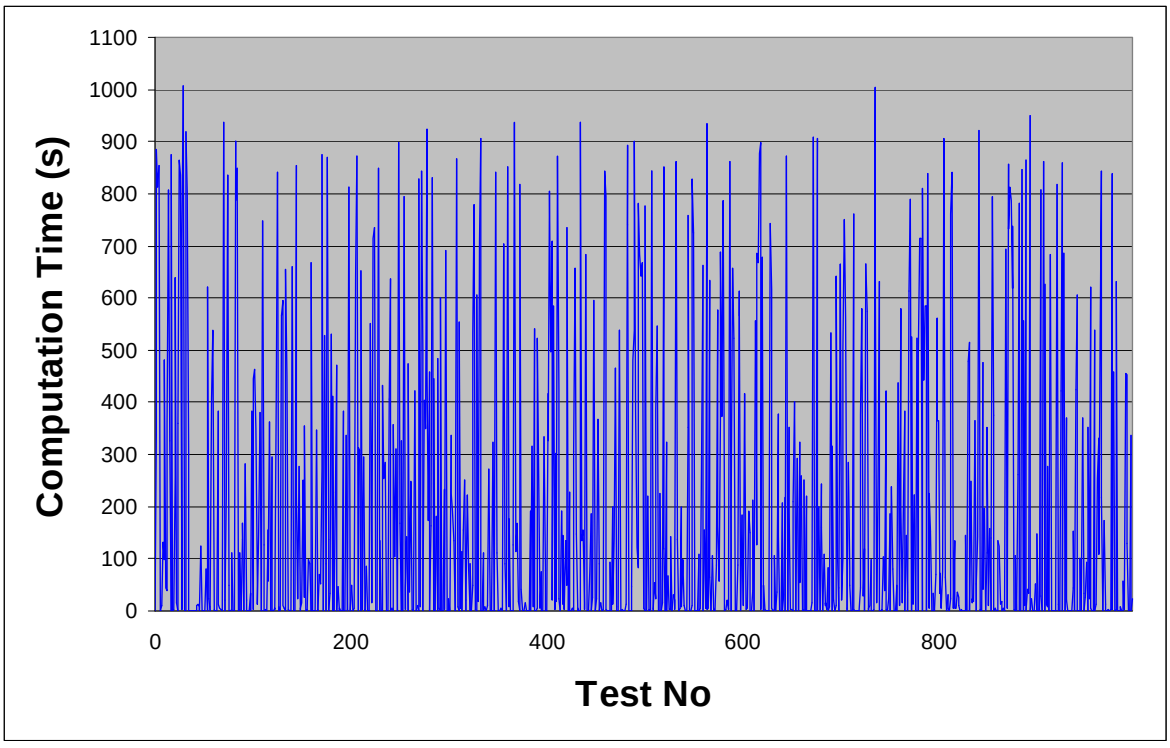

Figure 2. Hyperperiod simulation computation times

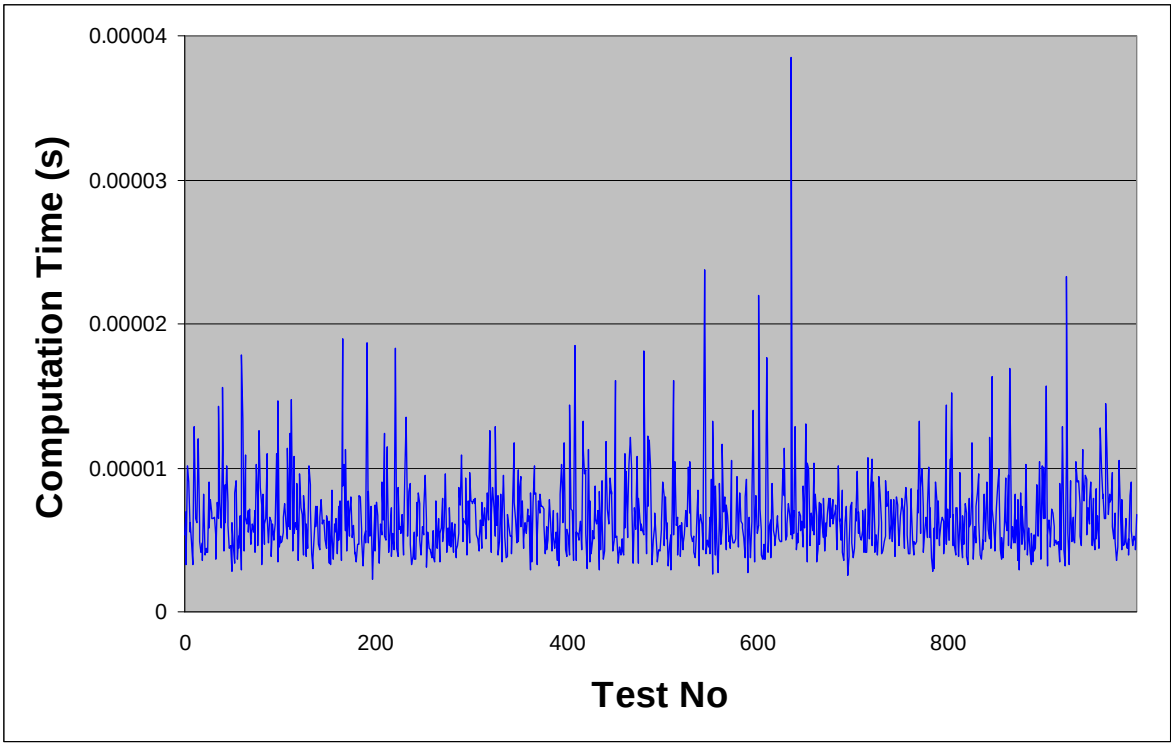

Figure 3. LCS algorithm computation times

It can be seen from Table I that in all cases, the LCS method was significantly faster than basic hyperperiod simulation for solving PSD. As can also be seen from the results, LCS is extremely robust in terms of the request parameters; in comparison, hyperperiod simulation sees huge fluctuations in computation time as these parameters vary, in some cases taking in excess of 16 minutes to solve even these vastly restricted instances. In order to gauge the effects symmetry on the heuristics proposed by Zeng et al. (2006), the relative size of the heuristic search spaces (r) with and without symmetry breaking was calculated as given by (11) and recorded for each instance.

$$r = \frac{\sum_{i \in N} p_i'}{\sum_{i \in N} p_i} \cdot 100 \tag{11}$$

It was found that the average relative size of the search space was 39.6 %, with the worst at 100.0 % (indicating a fully harmonic instance). These figures indicate the general effectiveness of the symmetry breaking technique[4]. A number of tests were then performed to gauge the increase in execution time of the LCS algorithm as the number of requests in the input set was increased. The

[4] It should be noted that when symmetry breaking is applied before exhaustive search, these figures are exponentially smaller, since the sum of terms in (11) can be replaced by product of terms.

restriction on request periods was lifted for these tests, being generated in the interval [1, 1000]. Starting from $n = 5$, the number of requests was increased in steps of 5 up to and including $n = 60$ requests. At each step, 1000 sets of requests were randomly generated and used as input to LCS. The resulting computation times (c) and relative sizes (r) are given in Table II, with computation times smoothed and shown graphically in Figure 4.

Table II. Effects of increasing request numbers (s)

n	Worst *c*	Ave *c*	Ave *r*	Worst *r*
5	0.0000017	0.0000013	5.0	98.4
10	0.0000391	0.0000063	13.6	84.4
15	0.0002108	0.0000111	25.2	84.8
20	0.0004312	0.0000336	31.9	87.5
25	0.0009067	0.0001115	37.9	85.2
30	0.0024535	0.0002995	42.9	86.0
35	0.0109658	0.0015746	47.2	85.2
40	0.0255350	0.0042588	50.7	85.1
45	0.0522661	0.0086182	53.8	88.0
50	0.0814180	0.0168891	56.5	91.5
55	0.1119380	0.0318693	58.9	92.0
60	0.1407680	0.0533591	61.0	91.8

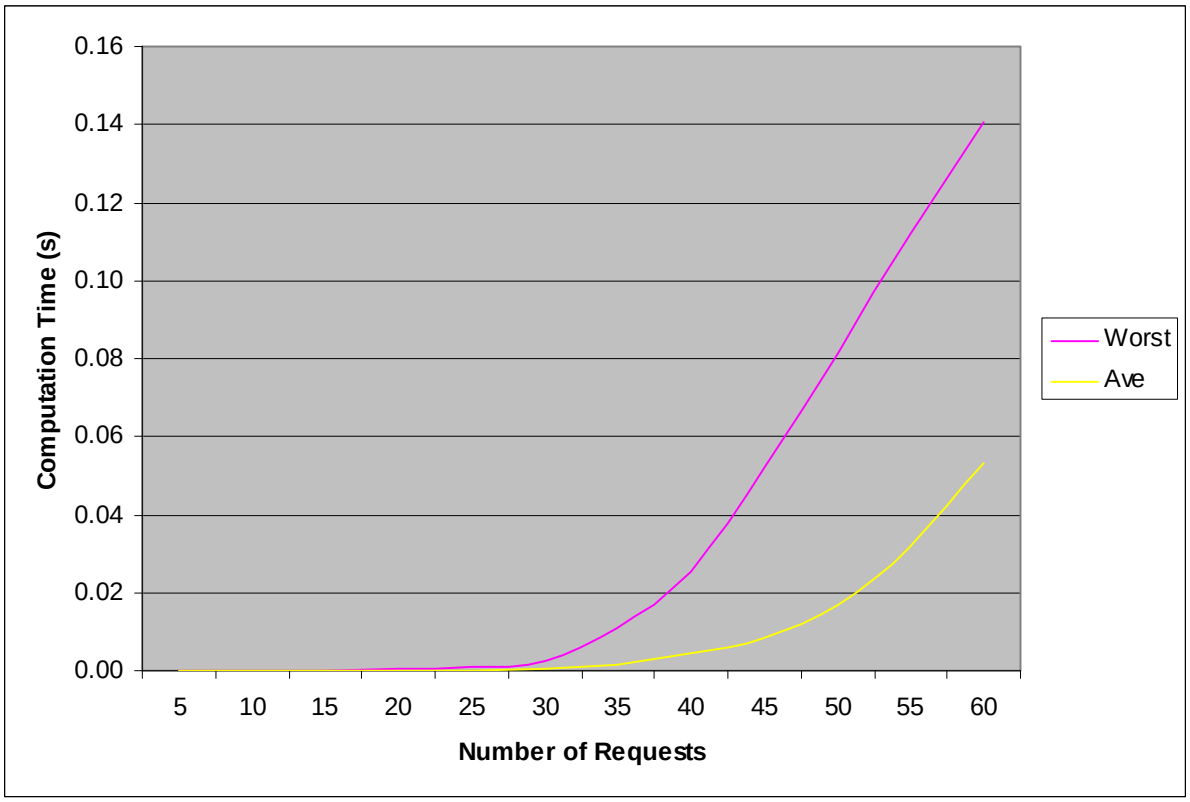

Figure 4. Effects of increasing request numbers

It can be seen from Table II that as the number of requests increases, the worst case computation time reflects the exponential in n; however, even given this complexity rise the worst case recorded

for $n = 60$ was still well under a second, providing evidence of the efficiency of the algorithm formulation and the effectiveness of the pruning rule. The average relative search space size increases with n, however significant average-case reductions are still evident. From these results, LCS would seem to be an effective subroutine to assist in quickly obtaining (heuristic) solutions to low / medium sized PM problem instances, with symmetry breaking further increasing the efficiency. In comparison, it can be extrapolated from the results shown in Table I and the calculated lengths of h that in the average case, hyperperiod simulation is completely intractable for periods in the interval [1, 1000] for most instances with $n \geq 15$.

The results and analysis presented in this short paper indicate the need for validating any assumptions – regardless of how 'obvious' they may seem – regarding the membership (or otherwise) of problems in particular complexity classes. In this particular case, it seems that a possibly incorrect assumption of the membership of a problem in NP could directly lead to unpredictable behaviour, possibly even run-time failures, of a mission critical information server application. Although techniques have been introduced in this note that may somewhat alleviate the problems for low to medium n, given the complexity of the problem it is unlikely that it can be solved - even heuristically - for arbitrarily large instances.

As a final note, the author notes that in such cases the *earliest deadline first* algorithm may be an attractive option; it is known to be optimal among the non-idling, non-preemptive scheduling algorithms such as those considered in this note (Jeffay et al. 1990). If the peak server demand is related to some form of required 'efficiency' setting for the server (required CPU speed, for example), then under non-preemptive EDF the optimum settings for an arbitrary PM instance may be determined in pseudo polynomial-time. Such a procedure could be achieved through binary search for a minimal efficiency setting to maintain feasibility of the request set, using the sufficient condition for feasibility derived by Jeffey et al. (1990). Given the optimality of EDF, it is easily seen that such a solution is likely to be no worse (and in many cases significantly better) than an optimal solution to the current formulation of the PM problem.

References

Baruah, S.K., Rosier, L.E. and Howell, R.R., 1990. Algorithms and Complexity concerning the preemptive scheduling of periodic tasks on one processor, Real-Time Systems, Vol. 2, No. 4, pp. 301-324.

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

Goosens, J., 2003. Scheduling of Offset Free Systems, Real-Time Systems, Vol. 24, No. 2, pp. 239-258.

Jeffay, K., Stanat, D.F. and Martel, C.U., (1991). On non-preemptive scheduling of periodic and sporadic tasks, In Proceedings of the 12th IEEE Symposium on Real-Time Systems, pp. 129-139.

Knuth, D.E., 1973. The Art of Computer Programming, Vol. 2: Semi numerical Algorithms, Addison-Wesley, Reading, Mass., 2nd edition, 1973.

Mok, A., Rosier, L., Tulchinsky, I. and Varvel, D., 1989. Algorithms and complexity of the periodic maintenance problem, Microprocessing and Microprogramming, Vol. 27, No. 1-5, pp. 657-664.

Pont, M.J., 2001. Patterns For Time Triggered Embedded Systems, ACM Press / Addison Wesley, 2001.

Zeng, D.D., Dror, M. and Chen, H., 2006. Efficient scheduling of periodic information monitoring requests. European Journal of Operational Research, 173, pp. 583-599.