Optimal Priority Ordering for Imprecise Computation - A Utility-Based Approach

Yanching Chu and Alan Burns

Real-time Systems Research Group
Department of Computer Science, University of York, UK
{ching,burns}@cs.york.ac.uk

Abstract

The deadline monotonic priority ordering is not optimal for schedulability when offsets are allowed in the task model. In addition, once schedulability is satisfied the maximization of system utility by scheduling tasks with higher values becomes important. This motivates the development of an optimal priority ordering algorithm, called UBPO, based on task utility by which higher priorities can be associated with more important tasks. The algorithm is optimal in the sense that it can find any feasible ordering whenever one exists, plus that its lexicographical distance relative to the optimal ordering is minimized, when priorities are ordered from the lowest to the highest priorities. Simulations show that the algorithm is much more efficient than another optimal algorithm in general, where although the measures being optimized are different, most of the time the final priority orderings are found to be the same. Further simulations demonstrate that the tractable schedulability test can be used with very small penalty on the priority orderings obtained. Together with previous works conducted in imprecise computation based on fixed priority scheduling, the package provides a more comprehensive set of scheduling theories and algorithms for supporting real-time AI systems.

1 Introduction

In fixed priority preemptive scheduling, task priority is static and assumed to be unchanged throughout a system’s lifespan. The assignment of task priority in a hard real-time system has only one objective, which is to guarantee that all the tasks in the system are schedulable with their assigned priorities, thereby meeting their respective deadlines. However, real-time AI applications require a more adaptive resource management in the system [8, 10]. This means that only satisfying system schedulability will be inadequate in supporting such applications.

For example, in the context of Dual Priority Scheduling for imprecise computation (iDPS) [9, 4, 6], a higher priority prologue task will finish its execution earlier (suffering less possible interference from higher priority tasks) and a higher priority epilogue task will have a later promotion time, which leads to a larger interval in which an optional component can be scheduled. Therefore, it will be more reasonable
to assign higher priority to imprecise tasks whose optional components have higher values to the system.

Although assigning priorities based on tasks’ deadlines is usually a general rule for optimal scheduling, Audsley [2] showed that if arbitrary offsets are allowed, neither rate monotonic priority ordering (RMPO) nor deadline monotonic priority ordering (DMPO) is optimal. He devised an optimal priority ordering algorithm which integrates with a feasibility test in determining task priority. The algorithm is guaranteed to find a feasible ordering whenever one exists. Although it is applicable to our task model (allowing offsets), however, the algorithm only concerns schedulability while the maximization of system utility based on the importance of tasks has not been considered.

Aguilar-Soto and Bernat [1] presented an optimal priority assignment algorithm which finds a priority ordering for a task set maximizing a given quality of service (QoS) measure that can be associated with task’s priority. The algorithm, unfortunately, assumes the optimality of DMPO in determining priorities and is associated with an inefficient schedulability testing procedure, which are both unsuitable for the task model concerned.

These previous works therefore motivate the development of an efficient algorithm that can find optimal priority orderings based on task’s utility. The rest of this report is organized as follows: we first revisit the priority ordering algorithm of Audsley in the next section before looking at the algorithm by Aguilar-Soto and Bernat. We discuss the details of their algorithms and the reason why they are not suitable for our scheduling framework. A new utility-based priority ordering algorithm is then presented, with proofs of correctness and optimality. This is followed by a set of simulations examining the performance of the proposed algorithms. The use of a tractable schedulability test in the UBPO algorithm is also studied and evaluated before we conclude the work.

2 Audsley’s Algorithm

The priority ordering algorithm devised by Audsley [2] addressed the problem of assigning priorities to tasks with arbitrary start time (offsets). When offset is allowed, neither RMPO nor DMPO is optimal in assigning task’s priorities. That is, there could be other priority orderings which are feasible but that RMPO and DMPO are infeasible. In such situation, in the worst-case, one has to examine all the $N!$ priority orderings for a system with $N$ tasks. Audsley’s algorithm reduces such complexity by proving the following theorem:

**Theorem 1.** Audsley [3]. If process $p$ is assigned the lowest priority and is feasible, then, if a feasible priority ordering exists for the complete task set, an ordering exists with process $p$ assigned the lowest priority.

The algorithm assigns priority using a bottom-up approach from the lowest priority to the highest, since once a task has been chosen for the lowest priority level, the theorem still holds for the rest of the
task set - the same procedure can therefore be applied iteratively to a smaller and smaller set of tasks. This priority ordering algorithm is optimal with respect to any scheduling test as long as task worst-case response time are monotonically non-increasing with higher task priority; hence it can also be used with the schedulability tests derived in [6] for imprecise computation of P-O-E task structure.

However, the original algorithm only assumes arbitrary ordering of tasks before running. That is, there is no attempt to associate higher priority with more important tasks. Observe that more than one feasible priority ordering may exist, to maximize the utility of scheduling optional components within the framework of iDPS, the objective is therefore to find a priority ordering which, in maintaining the schedulability of a system, maximizes the interval between the prologue and epilogue tasks.

In terms of fixed priority scheduling, it can be achieved by assigning higher priorities to imprecise tasks whose optional component has higher value than others, since the higher the priority the earlier prologue tasks can finish their execution and the later the promotion time can be set for epilogue tasks; the time interval for potential optional task execution is thus enlarged.

3 Aguilar-Soto and Bernat’s Algorithm

Aguilar-Soto and Bernat [1] presented an optimal algorithm, named “D&I”, which finds a priority ordering from a task set maximizing a QoS measure associated with task priority. In particular, the algorithm finds a schedulable priority ordering which has the minimum lexicographical distance from the most desired ordering, which is determined by a task importance function.

The algorithm employs a branch and bound mechanism, where priority ordering is modelled as a tree searching problem. From highest to lowest priority, at each iteration it tries to assign a priority to a task (an entry point leading to a subtree of possible orderings), where lower priorities tasks are ordered in the DMPO fashion. If any task becomes unschedulable that subtree is abandoned and another task is selected for the concerned priority.

Although it also takes $O((N^2 + N)/2)$ steps to find the ordering like Audsley’s algorithm, in each step where a priority level is tested, each lower priority task has to be verified to be schedulable. In contrast, each step of Audsley’s algorithm only involves testing the response time of the task concerned.

In addition, the algorithm assumes that when testing a priority for a task, all the lower priority tasks are ordered according to DMPO. However, as mentioned DMPO is not optimal for tasks with offsets. In other words, if the lower priority tasks ordered in DMPO is not schedulable, there may be other orderings which can make the task set schedulable. This renders the branch and bound mechanism used in the algorithm non-optimal and backtracking is now required. That is, if $M$ is the number of lower priority tasks whose priorities have not been determined in each iteration, in the worst-case it is necessary to examine all $M!$ extra orderings rather than the original $M$ orderings, which is clearly inefficient.
4 Utility-Based Priority Ordering Algorithm

Inspired by the bottom-up approach of Audsley’s algorithm, we modify it for assigning priorities to tasks by taking into account task utility. We prove here that the UBPO algorithm (Figure 1) also minimizes lexicographical distance like the D&I algorithm of Aguilar-Soto and Bernat when a different measure is used. In particular, the algorithm tries to assign the lowest possible priority to a task with low important at any given time, where an optimal priority ordering sequence is one whose tasks are ordered from lowest priority to the highest priority, left to right, where lower priority is assigned to tasks with lower importance based on a given task preference function.

We give a definition for our version of lexicographical distance of priority ordering sequence below and use a similar notations as defined in [1]. In particular, a task system is denoted by $\tau = \{\tau_1, \tau_2, ..., \tau_N\}$, representing all tasks in the system, where $N$ includes the total number of non-imprecise and imprecise tasks each represented by a pair of prologue and epilogue tasks.

If an order relation $\prec$ (“precede to”) is defined over the task set, a sequence $S = \langle \tau_1, \tau_2, ..., \tau_N \rangle$ can be obtained such that for any task position $j < k$ in the sequence $\tau_j \prec \tau_k$. The ordering relation is generic and any such relation that produces a total ordering over the task set can be employed as desired; in this context it may be referred to as a “less important than” relation. The set $\hat{S}$ represents the set of all possible $N!$ sequences for a system with $N$ tasks. Further we denote $S^D$ as the input sequence which is most desired by an order relation, and $S^F$ referring to the final priority ordering sequence found by an algorithm.

The ordering defines priority preference also: if $\tau_j \prec \tau_k$, we will try to assign lower priority to $\tau_j$. For simplicity, priorities are assigned from left to right in a priority ordering sequence - clearly if such sequence is equivalent to the sequence defined by the ordering we have obtained the most desirable priority ordering for all tasks.

However, since such sequence may not be schedulable, the objective is then to find a sequence that is as close to the desired sequence as possible. Aguilar-Soto and Bernat [1] defined the comparison of any two sequences in $\hat{S}$ as follows: if $\alpha, \beta \in \hat{S}$, $\alpha$ is said to be more important than $\beta$ if comparing each element $\alpha[i]$ with $\beta[i]$, where $i = 1...N$ starting from the leftmost to rightmost of two sequences, the first difference is in the $k^{th}$ task and $\alpha[k] \prec \beta[k]$. The more important sequence is said to be lexicographically smaller. However, since the order relation we use is “less important than” rather than “more important than”, the less important sequence will be lexicographically smaller. With the ordering relation, a function $I : \hat{S} \rightarrow [0, 1, ..., N! - 1]$ can be defined mapping all possible priority orderings to a unique number, where $I(S) = 0$ if $S$ is the most desired sequence defined by the relation. The lexicographical distance of two sequences $\alpha, \beta \in \hat{S}$ can therefore be obtained by $I(\alpha) - I(\beta)$. 
int Optimal_Priority_Ordering(Task[] TaskSet) {  
    int Schedulable;  
    for (int K=N; K>1; K--) {  
        Schedulable = 0;  
        for (int J=K-1; J>0; J--) {  
            if (Schedulability_Test(TaskSet, K)) {  
                Schedulable = 1;  
                break;  
            }  
            Swap(TaskSet, K, J);  
        }  
        if (!Schedulable)  
            return(UNSCHEDULABLE);  
    }  
    return(SCHEDULABLE);  
}

Figure 1. The utility-based priority ordering (UBPO) algorithm.

For example, let \( \tau = \{a, b, c\} \) and the desired ordering \( S^D = \langle bac \rangle \) (b the lowest importance), the lexicographical distance of the sequences are listed below:

\[
I(\langle bac \rangle) = 0 \\
I(\langle bca \rangle) = 1 \\
I(\langle abc \rangle) = 2 \\
I(\langle acb \rangle) = 3 \\
I(\langle cba \rangle) = 4 \\
I(\langle cab \rangle) = 5
\]

Figure 1 shows the utility-based priority ordering (UBPO) algorithm, in which the original bottom-up approach by Audsley is adopted. Given a desired task set sequence ordered based on task’s utility (a task importance ordering) as input, where high priority is assigned to task with higher utility, the algorithm finds a priority ordering that is both schedulable and lexicographically closest to the desired input ordering, if a feasible ordering does exist.

The outer loop in \( \text{Optimal\_Priority\_Ordering()} \) serves to fix the next lowest priority \( K \) for a task (from left to right of a priority sequence as \( K \) decreases), where the inner loop iteratively swaps the next lowest priority task candidate to priority \( K \) and test its schedulability. In particular, if a task is not schedulable with the assigned priority, the inner loop swaps its position with the next candidate task indexed by \( J \), until a schedulable task can be found.

After a task is found to be schedulable with an assigned priority, the iteration goes on to find the task for the next lowest priority until when \( K = 1 \) where there is only one task remaining and it is assigned
the highest priority. Note that for simplicity purpose of the algorithm, without loss of generality, it is assumed that any task is schedulable when it is assigned the highest priority in the system so that there is no need to perform schedulability test on that task.

Theorem 2. In fixed priority preemptive scheduling, the UBPO algorithm, using an exact feasibility test for determining task schedulability, is optimal in the sense that if there exists a feasible priority ordering it is guaranteed that the algorithm will find it.

Proof. This optimality is inherited from Audsley’s algorithm, based on the properties of fixed priority scheduling and response time test (Theorem 1) [2]. The original algorithm assumes the selection order of tasks in fixing priority can be arbitrary, the only difference between the two is that the UBPO algorithm makes use of a particular input ordering and task selection order.

Note that whether or not the algorithm can find a feasible ordering whenever one exists depends on the schedulability test employed. If a sufficient but not exact test is used, some orderings may be feasible but can not be found by the algorithm because it is a decision of the schedulability test. Since the schedulability test derived in [6] for imprecise computation is exact, the UBPO algorithm is also optimal in the sense that if there exists a priority ordering that makes a system schedulable it will find one.

Theorem 3. In fixed priority preemptive scheduling, the UBPO algorithm, using an exact feasibility test for determining task schedulability, is optimal in the sense that if a schedulable priority ordering is found, the lexicographical distance to the desired input ordering will be minimized among the set of all feasible orderings.

Proof. We first show that, in each iteration, the swapping operation does not affect the importance ordering relation in the subsequence representing tasks whose priorities have not been determined. Then it is shown that fixing a priority of a task with such swapping method will produce a task sequence having a smaller lexicographical distance from the optimal ordering among other feasible sequences. Since such property still holds true when the procedure is performed on smaller and smaller task subsets, the theorem is proved.

Intuitively, for the whole task set to be schedulable, each task has to be assigned a distinct priority with which it can meet its deadline. Starting from lowest priority, the algorithm tries to find the task with lowest importance that can be assigned the lowest priority. Since there must be one task to be assigned with a given priority for satisfying overall system schedulability, when a task is found schedulable with a priority it is the best task to be at that priority. The procedure can be performed iteratively on the remaining tasks in the same way. The effect is that the final priority ordering, when found, if any, has the minimum lexicographical order among all other schedulable orderings, relative to the desired task ordering.
More formally, consider a priority ordering sequence \( S = \langle \pi \tau_2 ... \tau_N \rangle \) with \( N \) tasks, which can be written as \( S = \langle \psi w \phi \rangle \), where \( \phi \) is a subsequence denoting tasks whose priorities have not been determined and less than \( K \), \( w \) the task under schedulability testing for determining priority (priority level \( K \) in the UBPO algorithm), and \( \psi \) the tasks with their priorities assigned. A desired ordering relation \( D \) is assumed to be given over \( \tau \) so that \( S^D \) is the optimal ordering sequence.

For example, let a sequence \( S \) be \( \langle abcd \rangle \) and \( S^D = \langle bade \rangle \). After \( b \) is found schedulable with the lowest priority and the schedulability of \( a \) in \( \langle bade \rangle \) is being checked, the whole sequence can be represented by \( \langle \psi w \phi \rangle \), where \( \langle \psi \rangle = \langle b \rangle \), \( w = a \) and \( \langle \phi \rangle = \langle dc \rangle \).

**Lemma 1.** The position swapping operation of a task in the subsequence \( \langle \phi \rangle \) with \( w \) retains the input task ordering relation in \( \langle \phi \rangle \).

**Proof.** Consider fixing a priority at level \( K \), where size of \( \langle \phi \rangle \) is \( K - 1 \), \( w \) is the task concerned. There are two cases to consider:

Case 1) \( w \) is schedulable at priority level \( K \). If \( w \) is schedulable, no swapping has been performed and \( \langle \phi \rangle \) has not been changed. Since the original input sequence is ordered by their importance, the importance ordering relation is retained.

Case 2) \( w \) is not schedulable with priority level \( K \). If \( w \) is not schedulable, the first task to be swapped to the position \( K \) is \( S[K-1] \) (note the sequence is indexed by \( K \) decreasingly from left to right), let’s call it \( \tau_s \). If \( \tau_s \) is schedulable at \( K \), \( \langle \phi \rangle \) is clearly in accordance with the ordering relation since task \( S[K-1] \) certainly has a lower utility than the rest in \( \langle \phi \rangle \).

If \( \tau_s \) is not schedulable at priority level \( K \), observe how it is swapped with \( S[K-2] \) as indexed by \( J \) now (as in the algorithm). The task \( S[K-1] \) and \( S[K-2] \) are in the same ordering as in the original sequence where they were at \( S[K] \) and \( S[K-1] \) - only the task that is of interest is swapped to position \( K \) while maintaining the ordering relation in \( \langle \phi \rangle \).

When a task’s priority is fixed, considering the next task \( w \) indexed by \( K \) is just another instance of the same decision and swapping procedure mentioned above (Case 1 and 2). Since in each iteration the swapping operation does not concern the tasks in \( \langle \psi \rangle \) whose priorities have been fixed, it is easy to see that the lemma holds true as the algorithm carries on fixing priorities from level \( K - 1 \) to 2, after which the algorithm stops when the size of \( \langle \phi \rangle \) equals 1.

**Lemma 2.** After a task \( w \) is found feasible and its priority fixed, the resulting subsequence \( \langle w \phi \rangle \), ignoring \( \langle \psi \rangle \), has the smallest lexicographical ordering among all other feasible orderings of \( \langle w \phi \rangle \).

**Proof.** By Lemma 1 we know that for any task selected from \( \langle \phi \rangle \) which is schedulable at priority level \( K \), \( \langle \phi \rangle \) retains the order relation according to the given importance function.
Since the swapping operation only swaps a task with the next lowest utility when one is found not schedulable, task $w$ must be the first feasible task with the lowest utility and $\langle w\phi \rangle$ must have the lexicographically smallest order among all possible ones.

Assume one final feasible sequence is found by the algorithm, let’s call it $S^F$. If we traverse from the final step of the algorithm to the initial one, the last subsequence of $\phi$ contains the highest assigned priority task at $S^F[N]$ (with priority 1, the highest in the algorithm), it is guaranteed by the above lemmas that by assigning the task $S^F[N - 1]$ (the task $w$) with its priority the lexicographical distance of $\langle w\phi \rangle$ will be minimized. Similarly, when we enlarge the sequence $\phi$ to include the next task $S^F[N - 2]$, it is also guaranteed by the above lemmas that by assigning the task $S^F[N - 2]$ (the new $w$) its priority the lexicographical distance of the sequence $\langle w\phi \rangle$ will be minimized. The same argument goes on and the lemmas still hold true for the remaining tasks when the lexicographical comparison considers the next task to the left, until all tasks are included.

As mentioned, similar to the original algorithm, the algorithm takes $O((N^2 + N)/2)$ steps where each step only involves performing a schedulability test for one single task, whereas the D&I algorithm requires testing the schedulability of all the lower priority tasks plus the task concerned. The basic (not exact for our task model, but sufficient) response time test [5] for all tasks requires pseudo-polynomial time which depends on the number of tasks in the system. The exact test developed for the P-O-E task model has a complexity of $O(2^N)$ time [6] while the tractable sufficient test [7] requires $O(2N)$ for a single task.

5 Performance Evaluation

In this section we first compare the performance of the D&I and UBPO algorithms before examining the utility gain, via simulations, from adopting a utility-based approach for assigning priorities to imprecise tasks with higher values.

The primary metric for measuring performance is the total number of exact response time tests, defined as Equation (5) and (6) in [6], for a particular task the algorithms perform in finding the optimal priority ordering. Note that a test is also performed for the task with the highest priority in the system (assumed to be automatically schedulable in the previous section). Further, we denote $T_{UBPO}$ as the number of response time tests (a busy period calculation) performed by the UBPO algorithm, and, $T_{D&I}$ as the number of tests used in the D&I algorithm.

As mentioned, the D&I algorithm is not optimal for tasks with offsets because when a priority is tested for a task, the lower priority tasks of which can not be ordered in DMPO with absolute certainty that it is optimal. In other words, if any of the lower priority tasks is not schedulable, there may be other priority
orderings for them that are schedulable. Without this optimality, the algorithm will interpret that the priority being tested for the current task is not feasible, and switch to the next task with highest utility among the lower priority tasks.

However, since DMPO is generally a good scheduling heuristic and that we would like to compare the performance of the two algorithms, it is only when the final priority orderings found by the two are equal are their performance being compared. This evaluation will be valuable because if task offsets are not allowed then the D&I algorithm is still optimal.

Given the utilization of a task system $u$, a task $\tau_i$ (including all prologue, epilogue and normal tasks) is randomly generated with period values spanning across a few orders of magnitude (4 - 7 digit figures) in a roughly uniform way where the value of each digit is chosen randomly (1 - 9 for the most significant digit and 0 - 9 for the rest). Deadlines are chosen to be at about 90% of the period value with small variations such that $C_i \leq D_i \leq T_i$. Figure 7 in the appendix shows the task preference function used in the simulations for comparing two tasks (the type of a task as the primary factor, and then deadlines), so that an optimal priority ordering can be obtained.

Note that the resolution of a simulation tick is one micro-second ($\mu$s), thus the tasks generated represent a reasonable period distribution of a real system. All mandatory tasks are executed up to their WCET where they do not hold any resources (no blocking). Once a period is generated, the WCET $C$ of the task is then calculated to provide the desired system utilization. Further, one of the normal tasks will be generated with the skew parameter $s$ as:

$$C = s \ast u,$$  \hspace{1cm} (1)
Figure 3. Tractable test performance against varying system utilization.

where the WCET of remaining tasks are defined as:

\[ C = (1 - s) \times u. \]  

(2)

Except the varying parameter in each experiment, all the other parameters are held at default values as follows:

- system utilization \( u = 0.5 \);
- number of imprecise tasks \( n = 5 \);
- number of normal tasks \( m = 10 \);
- utilization skew \( s = 1/(n + m \times 2) \) such that all tasks equally share \( u \);

Figure 2 shows the values of \( T_{UBPO} \) and \( T_{D\&I} \) against varying number of system utilization. For statistical significance, each data point in the graph represents the average \( T_{UBPO} \) and \( T_{D\&I} \) over 100 randomly generated task sets with the same set of parameters. A reference line is provided indicating the average number of swapping operations actually performed per task set.

It can be seen that the the number of response time test performed by UBPO, \( T_{UBPO} \), is relatively small and hardly increases as the system becomes more utilized. The value is larger than the number of swapping because at least one testing needs to be done to confirm a valid priority for a task. In fact a few of the task sets generated do not require any priority swapping at all; they are all schedulable with the desired priority ordering.

The same task sets used for UBPO were also used by D\&I for producing its performance metric. The number of tests performed increases very steadily upward with higher system utilization, due to the
increased likelihood a task may become unschedulable (more interference from other tasks). It can also be seen that there is a big difference in performance between the two algorithms.

Figure 3 shows the values of $T_{UBPO}$ and $T_{D&I}$ against varying number of imprecise tasks. The line of $T_{D&I}$ shows a more rapid decline in performance (note the scale of y-axis) than in the previous experiment as more imprecise tasks are added into the system. This can be understood by realizing that even in the best case scenario for a system of 20 tasks, UBPO only requires $N = 20$ tests to be performed while D&I will require $(N^2 + N)/2 = 210$ tests - a decisive difference. As more tasks are present in the system, it becomes harder to schedule tasks in their desirable positions. This results in more swapping operations required to be performed (note the slight increase in the reference line) and correspondingly the performance deteriorates very quickly with the D&I approach. This is further compounded by the fact that since the exact test runs in exponential time with regard to the number of imprecise tasks, only a handful number of imprecise tasks can be tested (also see next section).

Note that although the two algorithms minimize lexicographical distance in different directions, it is hard to justify which is better. This is because even being lexicographically smaller, the average difference of task position to their relative optimal position may actually be higher. There is no simple means in differentiating which ordering direction is better. That is, deciding if trying to assign higher priorities to more important tasks is better than trying to assign lower priorities to less important tasks. Note again that only UBPO can work with tasks with offsets, which are required in the task model of iDPS.

In addition, only in 3 out of the 903 generated task sets (thus that 3 task sets are not used) did we see the orderings obtained from the two algorithms being different, which is a very small ratio (0.003%). It is also interesting to note that, if the two orderings obtained by the two algorithms are equal, the total number of task priority swappings that are required in finding the optimal ordering by the two algorithms are exactly the same - even though the tasks and the order in which that they are swapped are totally different.

6 Employment of Tractable Schedulability Test

The exact test [6] for imprecise computation is computationally prohibitive when there are large number of imprecise tasks present in the system. This motivates a more efficient approach for verifying system schedulability. Based on Tindell’s analysis, a tractable test for the task model concerned has been derived [7].

Although the test performs extremely well in the response time difference produced compared to that of the exact test, nevertheless, the UBPO algorithm tries to assign task priorities based on task importance. Since DMPO is no longer employed, tasks are likely to suffer longer interference from other
Figure 4. Exact and tractable UBPO performance against varying system utilization.

...tasks due to the fact that a more urgent task (with shorter deadline) may have to wait for other tasks which have higher utility.

If this happens when the UBPO algorithm uses the tractable test for checking schedulability, the resulting priority ordering may be different from that produced using the exact test. This is because a task may be deemed unschedulable with an assigned priority while it actually is schedulable.

It will be interesting to see how the tractable test affects the results when different parameters are used. The Manhattan distance as a measure is defined as the sum of the absolute distance of each task to its relative position in the optimal priority ordering sequence. More formally, the Manhattan distance function is a function $M : \hat{S} \rightarrow \{0, 1, 2, \ldots\}$ such that:

$$M(S) = \sum_{i \in \tau} |Pri(S, i) - Pri(S^D, i)|,$$

where $Pri(S, i)$ returns the priority position of a particular task $\tau_i$ in the priority sequence $S$ so that $M(S^D) = 0$.

For example, with $\tau = \{a, b, c\}$ and a desired ordering $S^D = \langle bac \rangle$, the Manhattan distance of all the possible sequences are listed below:

- $M(\langle bac \rangle) = 0$
- $M(\langle bca \rangle) = 2$
- $M(\langle abc \rangle) = 2$
- $M(\langle acb \rangle) = 3$
- $M(\langle cba \rangle) = 4$
- $M(\langle cab \rangle) = 4$
Figure 5. Exact and tractable UBPO performance against varying number of imprecise tasks.

The metric for measuring the performance of the algorithms is the average Manhattan distance per task in a task set defined as:

$$AM = M/N,$$  \hspace{1cm} (4)

where $N$ is the number of tasks in the system. Further we denote $AM_e$ to be the average Manhattan distance per task of the UBPO algorithm performed with the exact test, while $AM_t$ one produced with the tractable test. The parameter settings are similar to the first set of experiments documented in the last section.

Figure 4 shows the values of $AM_e$ and $AM_t$ against varying number of system utilization. For statistical significance, each data point in the graph is averaged over 100 randomly generated task sets with the same set of parameters.

It can be observed that as the utilization increases the average Manhattan distance per task also increases, which implies it is harder to obtain an ordering which is as close to the optimal ordering as possible as the system load increases. It can also be seen that the tractable test is tracking the performance of the exact test rather closely.

Figure 5 shows the values of $AM_e$ and $AM_t$ against varying number of imprecise tasks. The figure shows that there seems to be no direct correspondence between the performance and the control parameter, since the performance gets worse and then better twice alternatively as more imprecise tasks are present in the system. However, the performance of the tractable test shows great promise (note the scale of the y-axis) in both set of simulations, with only 0.15 difference in the average Manhattan distance per task set on average. We therefore conclude that tractable test can be safely employed with only little penalty in performance in most cases.
7 Conclusion

The deadline monotonic priority ordering is not optimal for schedulability when offsets are allowed in the task model. Further, when schedulability is satisfied the maximization of system utility by scheduling more important optional components becomes important. This motivates a priority ordering algorithm based on utility, called UBPO. The algorithm is optimal in the sense that it can find any feasible ordering whenever one exists, plus that the lexicographical distance to the optimal ordering (lowest to highest, left to right) is minimized.

Simulations show that the UBPO algorithm is in general much more efficient than the algorithms previously available, where most of the time the final priority orderings found are the same as found by the other optimal algorithm. Where there is a large number of imprecise tasks the tractable schedulability test may have to be introduced in determining task priorities. Experimental data shows that the average Manhattan distance penalty for a large number of randomly generated task sets is only 0.15 on average, proving the usability of the tractable test.

Together with previous works, imprecise computation based on fixed priority scheduling provides an efficient means for supporting real-time AI systems with a set of scheduling theories and algorithms. These are expected to be incorporated into existing real-time operating systems in the future, which hopefully will promote a more general use of imprecise computing.

Acknowledgement

It is a pleasure to acknowledge Rob Davis for his helpful comments and discussions on the earlier drafts of this report.

References


```c
Task Task_Preference(Task a, Task b) {
    /* if 'a' is an imprecise task */
    if (a.type == IMPRECISE) {
        /* give preference to imprecise task */
        if (b.type == NON_IMPRECISE)
            return a;
        else {
            /* give preference to higher utility */
            if (a.utility > b.utility)
                return a;
            else if (a.utility == b.utility) {
                /* if 'a' has smaller deadline */
                if (a.deadline < b.deadline)
                    return a;
                else if (a.deadline == b.deadline) {
                    /* give preference to epilogue over prologue */
                    if (a.IS_EPILOGUE && b.IS_PROLOGUE)
                        return a;
                    else if (a.IS_PROLOGUE && b.IS_EPILOGUE)
                        return b;
                    /* if all are equal return 'a' */
                    else
                        return a;
                }
                /* if 'b' has has smaller deadline */
                else
                    return b;
            }
            /* if 'b' has higher utility */
            else
                return b;
        }
        /* if 'a' is a non-imprecise task */
    } else {
        /* give preference to imprecise task */
        if (b.type == IMPRECISE)
            return b;
        else {
            /* preference to smaller deadline, if same return 'a' */
            if (a.deadline <= b.deadline)
                return a;
            else
                return b;
        }
    }
}
```