Non-preemptive scheduling to maximize the minimum intercompletion time

Carlos C. Amaro
Thomas J. Marlowe
Sanjoy K. Baruah
Alexander D. Stoyen*

Abstract

Temporal load-balancing – "spreading out" the executions of tasks over time — is desirable in many applications. A form of temporal load-balancing is introduced: scheduling to maximize minimum inter-completion time (MICT-scheduling). It is shown that MICT-scheduling is, in general, NP-hard. A number of restricted classes of task systems are identified, which can be efficiently MICT-scheduled.

Keywords: non-preemptive scheduling, independent tasks, single and multiple processors, load-balancing.

*Amaro, Marlowe, and Stoyen are with the Real-Time Computing Laboratory, Department of Computer & Information Science, New Jersey Institute of Technology, University Heights, Newark, NJ 07102 USA. Marlowe is also with the Department of Mathematics and Computer Science, Seton Hall University, South Orange, NJ 07079. Baruah is with the Department of Computer Science, The University of Vermont, Burlington, VT 05405. E-Mail/Phone: *amaro@homer.njit.edu/973-596-2989, sanjoy@cs.uvm.edu, marlowe@cs.rutgers.edu/-3397, alex@rtlab.njit.edu/-5675.*

JCMCC 34 (2000), pp. 33-50

1 Introduction

The issue of shared resources is of prime importance in computer systems in general, and parallel and distributed applications in particular. Resources that need to be shared can range from processing elements (PE's) on parallel machines, to communication links in networks, to critical sections of code to which processes need exclusive access.

Given one or several resources that need to be shared by a set of tasks, **scheduling** is the process of determining which task gets access to which resource, and when. Depending upon the characteristics of the application system under consideration, this process of schedule generation aims to optimize specified objectives. In hard real-time systems, the critical objective is that all tasks complete by their deadlines.

Our attention in this research is restricted to the *non-preemptive* scheduling of independent real-time tasks. In our model, each task T_i is characterized by three parameters – a *release time* r_i, an *execution requirement* e_i and a *deadline* d_i, with the interpretation that task T_i becomes ready for execution at time r_i, and needs to be executed non-preemptively for e_i units of time over the interval $[r_i, d_i)$. Given a set $\tau = \{T_1, \ldots, T_n\}$ of n such tasks to be scheduled on m identical processors, the primary goal is to generate a schedule in which each task completes execution by its deadline.

Assuming that this primary goal can be met by several different schedules, secondary objectives may play a role in determining scheduling strategy. The focus of this research is one such secondary objective — that of maximizing inter-completion time.

Inter-completion time. A schedule for τ on m processors is completely defined by specifying, for each $T_i \in \tau$, the processor p_i on which T_i is to execute, and the *start time* s_i at which it begins execution[1]. The time instant $c_i \stackrel{\text{def}}{=} s_i + e_i$ is called the *completion time* of task T_i in this schedule. For a given schedule, the *minimum inter-completion time on processor* p is defined to be the smallest difference between the completion-times of successive tasks that execute on processor p (if there is only one task that executes on a processor, then its minimum inter-completion time is defined to be ∞.). The *minimum inter-completion time (MICT) of a schedule* is defined to be the minimum, over all processors p, of the minimum inter-completion time of processor p. *MICT-scheduling* is the process of generating a schedule with the largest possible minimum inter-completion time.

[1] Of course, a valid schedule would require that each task execute within its release time and deadline (i.e., $s_i \geq r_i$ and $s_i + e_i \leq d_i$), and that no processor is assigned to more than one task at any given instant in time (i.e., if $p_i = p_j$, and $s_i \leq s_j$, then $s_i + e_i \leq s_j$).

Example 1 Consider a set of tasks $\tau = \{T_1 = (0,4,5), T_2 = (0,3,7), T_3 = (3,2,10), T_4 = (0,10,12), T_5 = (3,1,10), T_6 = (9,1,12)\}$. Figure 1 shows a schedule for this system on three processors. The minimum inter-completion time on Processor $P1$ is 6; since only once task completes on $P2$, its minimum inter-completion time is set equal to ∞; the minimum inter-completion time on processor $P3$ is between the completions of tasks T_4 and T_3, and is equal to 4. The minimum inter-completion time for the schedule is therefore $\min(6, \infty, 4)$, which is 4.

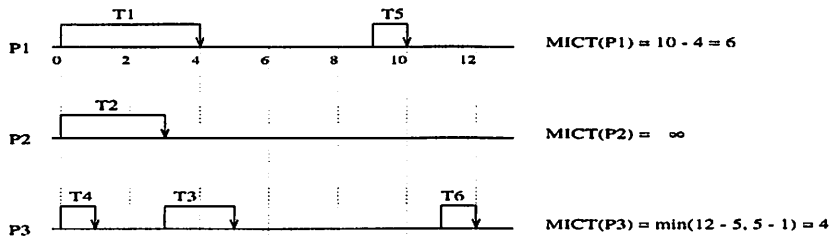


Figure 1: Schedule for task system of Example 1

■

In traditional load-balancing, the aim is to distribute the given set of tasks as evenly as possible among the available PE's. We view MICT-scheduling as an extension to this view of load balancing, in that we are attempting to "balance" the load *temporally* as well as spatially (i.e., over the PE's). This perspective on load-balancing is particularly useful in situations where some additional work needs to be done whenever a task completes execution, and we therefore wish to spread out these events as much as possible. For example, a real-time architecture may have two caches associated with each PE – while the currently executing process T_i uses one cache, the memory subsystem (e.g., the DMA processor) is loading the other for the next scheduled process T_j. When T_i completes execution, its cache is flushed and loaded for the process T_k that is scheduled to follow process T_j (of course, T_k can only begin execution after T_j completes). By spreading out the completion times, we render it less likely that the memory subsystem gets overloaded.

Our attention was drawn to maximizing the MICT as a scheduling objective by the following situation. A given set of software modules, each with an *a priori* known release time, deadline, and (worst-case) execution requirement, needs to be scheduled for non-preemptive execution on a homogeneous multiprocessor with m PE's, where each PE consists of a powerful primary processor, and a much less powerful co-processor. The software modules need to execute on the primary processors; when each

	FIXED			Processors	
	release time	execution req.	deadline	Uniproc	Multiproc
1	N	N	N	NP-hard (Theorem 1)	NP-hard (Theorem 2)
2	N	N	Y	$O(n \log n \log d)$ (Section 6.1.1)	NP-hard (Theorem 6)
3	N	Y	N	$O(n^2 \log n \log d)$ (Section 6.1.3)	open (Section 6.2)
4	Y	N	N	$O(n^2 \log d)$ (Section 6.1.2)	NP-hard (Theorem 7)
5	N	Y	Y	$O(n \log n)$ (Section 5.1)	$O(n \log n)$ (Section 5.1)
6	Y	N	Y	$O(n \log n)$ (Section5.2)	NP-hard (Theorem 4)
7	Y	Y	N	$O(n \log n)$ (Section 5.1)	$O(n \log n)$ (Section 5.1)
8	Y	Y	Y	$O(n)$ (Section 4)	$O(n)$ (Section 4)

Table 1: Summary of results in this paper ($n \stackrel{\text{def}}{=}$ number of tasks; $\hat{d} \stackrel{\text{def}}{=}$ the largest deadline; the smallest release time is assumed to be 0)

module completes execution, it spawns certain other jobs (including: *bookkeeping*, *checkpointing*, and *communicating* the results generated by the module to the other PEs), which are performed on the co-processor. It is desirable that the co-processor complete executing all jobs spawned by one module before the subsequent module spawns its set of jobs — this is more likely to be achieved in schedules with large inter-completion times.

This research. Each task may be considered to enjoy three "degrees of freedom" — one for each of its parameters. In Section 2 (Theorems 1 and 2), we show that we are unlikely to be able to obtain efficient MICT-scheduling algorithms that can schedule arbitrary sets of tasks, even on a single processor. We therefore investigate the issue of designing optimal MICT-scheduling algorithms when one or more of the degrees of freedom are curtailed. In Section 4, we consider sets of tasks in which all tasks are identical – i.e., each task has zero degrees of freedom. In Section 5, we consider task sets in which each task has one degree of freedom. That is, we separately consider the cases where all tasks (i) have the same release time and execution requirement, but may have different deadlines, (ii) have the same execution requirement and deadline, but may have different release times, and (iii) have the same release time and deadline, but may have different execution requirements. In Section 6, we consider task sets where each task has two degrees of freedom — once again, we have three different possibilities, which are individually analyzed. For each of the cases listed above, we consider both uniprocessor and multiprocessor MICT-scheduling. Our results are summarized in Table 1. In addition, we have established a relationship (Section 3) between the MICT-scheduling problem and the well-studied problem of non-preemptive scheduling when inter-completion time is not a consideration: this relationship permits us to apply a wide variety of results — heuristics, approximation algorithms, etc — that exist for non-preemptive scheduling to the MICT-scheduling problem.

2 MICT-scheduling is hard

In this section, we prove that the general problem of obtaining a schedule with large minimum inter-completion time for an arbitrary task system is intractable. We start with some definitions.

A *task system* is specified by an ordered pair

$$\langle \tau = \cup_{i=1}^{n} \{T_i = (r_i, e_i, d_i)\}, m \rangle$$

and represents a set of n tasks $T_1, T_2, \ldots, T_n$, to be scheduled on m identical processors, where task T_i is released at time r_i, has a deadline of d_i, and an execution requirement of e_i.

$\mathsf{mict}(\langle \tau, m \rangle) = \Delta$ indicates that there is a schedule for task system $\langle \tau, m \rangle$ with a minimum inter-completion time at least Δ. (Thus, asserting $\mathsf{mict}(\langle \tau, m \rangle) = 0$ is equivalent to stating that τ is feasible on m processors.)

Lemma 1 Given an arbitrary set of tasks τ and an arbitrary integer $\Delta > 0$, the problem of determining whether $\mathsf{mict}(\langle \tau, 1 \rangle) = \Delta$ is NP-complete in the strong sense.

Proof Sketch: Transformation from *Sequencing with release times and deadlines* [2, page 236]. ■

As a direct consequence of Lemma 1, we obtain the following theorem:

Theorem 1 Given an arbitrary set of tasks τ, it is NP-hard to schedule τ on one processor such that the minimum inter-completion time is maximized.

■

Theorem 2 immediately follows.

Theorem 2 Given an arbitrary set of tasks τ and m processors it is NP-hard to schedule τ on the m processors such that the minimum inter-completion time is maximized.

■

3 Reducing MICT-scheduling to feasibility

While the issue of non-preemptive scheduling to maximize minimum inter-completion time has not been widely studied, there does exist a vast amount of literature devoted to feasibility analysis for non-preemptive scheduling. These include intractability results [2, Section A5], approximation algorithms [3, 4, 5], optimal algorithms for special cases [1, 6], etc. In this

section, we attempt to exploit this wide body of research by establishing a relationship between MICT-scheduling and general non-preemptive scheduling.

The following theorem reduces the problem of determining schedules with specified minimum inter-completion times to the problem of determining feasibility of sets of tasks.

Theorem 3 Let $\tau \stackrel{\text{def}}{=} \cup_{i=1}^{n}\{T_i = (r_i, e_i, d_i)\}$ be a set of tasks. Let $\Delta \geq 0$. For each i, $1 \leq i \leq n$, define $\delta_i \stackrel{\text{def}}{=} \max(0, \Delta - e_i)$. Define $r(\tau, \Delta) = \cup_{i=1}^{n}\{T_i' = (r_i - \delta_i, e_i + \delta_i, d_i)\}$.

$$\text{mict}(\langle \tau, m \rangle) = \Delta \quad \textbf{iff} \quad r(\tau, \Delta) \text{ is feasible on } m \text{ processors.}$$

Example 2 Consider a set of tasks $\tau = \{T_1 = (0, 3, 6), T_2 = (0, 1, 7), T_3 = (4, 6, 12)\}$. We wish to determine whether τ can be scheduled on one processor such that the minimum inter-completion time is at least five ($\Delta = 5$). Since $\delta_1 = 2, \delta_2 = 4, \delta_3 = 0$, Theorem 3 claims that this is equivalent to determining whether $r(\tau, 5) = \{T_1' = (-2, 5, 6), T_2' = (-4, 5, 7), T_3' = (4, 6, 12)\}$ can be scheduled on one processor:

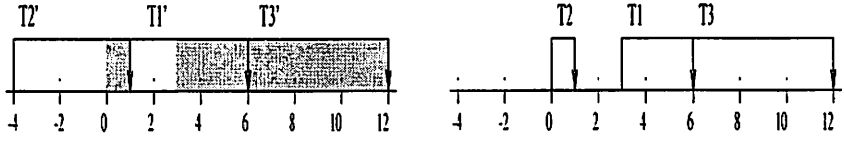


Figure 2: Schedule for $r(\tau, 5)$, and schedule for τ with MICT 5

Proof of Theorem 3: In this proof, let τ' denote the task set $r(\tau, \Delta)$.
LHS ⇒ RHS. $\text{mict}(\langle \tau, m \rangle) = \Delta \Rightarrow \langle \tau', m \rangle$ is feasible:
Suppose first that $\text{mict}(\langle \tau, m \rangle) = \Delta$, and let S_o be an m-processor schedule for τ with a minimum inter-completion time $\geq \Delta$. We describe how to obtain a schedule S_1 for τ' on m processors.

For each i, $1 \leq i \leq n$, let $[t_s, t_s + e_i)$ denote the interval during which T_i was executed in schedule S_o. T_i' is executed on the same processor in S_1; its execution interval is determined as follows:

- If $\delta_i = 0$, then the execution requirements of T_i' and T_i are the same, and S_1 schedules T_i' over the interval $[t_s, t_s + e_i)$.
- If $\delta_i > 0$, observe that (i) the processor on which T_i is executed in S_o is idle over the interval $[t_s - \delta_i, t_s)$ (this follows from the fact that the minimum inter-completion time of S_o is at least Δ), and (ii) since $t_s \geq r_i$, it must be the case that $t_s - \delta_i \geq r_i - \delta_i$. Schedule S_1 therefore executes T_i' over the interval $[t_s - \delta_i, t_s + e_i)$.

LHS $\Leftarrow$ RHS. $\langle \tau', m \rangle$ is feasible $\Rightarrow$ mict$(\langle \tau, m \rangle) = \Delta$:
Suppose now that τ' is feasible on m processors, and let S_2 be an m-processor schedule for τ'. We describe below how to obtain a schedule S_3 for τ on m processors, which has a minimum inter-completion time of (at least) Δ.

For each i, $1 \leq i \leq n$, let $[t_s, t_s + e_i + \delta_i)$ denote the interval during which T_i' is scheduled in S_2. Then T_i is executed on the same processor in S_3; its execution interval is determined as follows:

- If $\delta_i = 0$, then the execution requirements of T_i' and T_i are the same, and S_3 schedules T_i over the interval $[t_s, t_s + e_i)$ as well. Since $e_i \geq \Delta$, the separation between the completion time of T_i and the task (if any) that was executed prior to it on the same processor is at least Δ.
- If $\delta_i > 0$, observe that $t_s \geq r_i - \delta_i$. S_3 assigns the processor to T_i over the interval $[t_s + \delta_i, t_s + e_i + \delta_i)$. Observe that (i) this interval is of size e_i, (ii) since $t_s \geq r_i - \delta_i$, this interval starts no earlier than r_i, and (iii) since $e_i + \delta_i = \Delta$, the separation between the completion time of T_i and the task (if any) that was executed prior to it on the same processor is exactly Δ.

■

Remark 1 The proof of Theorem 3 is *constructive* — given a schedule for $r(\tau, \Delta)$ on m processors, we can use the reduction defined in Case 2 of the proof to construct a schedule for τ on m processors with a minimum inter-completion time $\geq \Delta$. Furthermore, such a reduction can be performed in $\mathcal{O}(n)$ time.

4 Task systems with no degrees of freedom

We start out by considering the very simple problem of MICT-scheduling a set of n identical tasks $\tau = \{T_1, \ldots, T_n\}$, where $T_i = (0, E, D)$, on m processors.

Consider any schedule for $\langle \tau, m \rangle$. Since there are n tasks to be scheduled on m processors, some processor will be assigned at least $\lceil n/m \rceil$ tasks. The first task on this processor completes at (or after) time E; the interval $[E, D)$ is to be partitioned into at least $(\lceil n/m \rceil - 1)$ inter-completion times. Therefore, the minimum inter-completion time, obtained by partitioning $[E, d)$ as evenly as possible subject to integer boundaries, is

$$\Delta_{max} \stackrel{\text{def}}{=} \lfloor \frac{D - E}{\lceil n/m \rceil - 1} \rfloor$$

An algorithm for generating a schedule with a minimum inter-completion time of $\Delta_{\max}$ is given in Figure 3; since its correctness is quite obvious, a

```
Δ ← ⌊(D−E)/(⌈n/m⌉−1)⌋;
If Δ < E return "not feasible";
t ← 0; /* time */
p ← 1; /* processor */
for i ← 1 to n do {
    if t + E > D {
        p ← p + 1;
        t ← 0;
    }
    Schedule T_i on processor p over the interval [t, t + E);
    t ← t + Δ
}
```

Figure 3: Algorithm for MICT-scheduling a set of identical tasks

formal proof of correctness is omitted. Observe that its run-time complexity is $\mathcal{O}(n)$, where n is the number of tasks.

5 Task systems with one degree of freedom

The case when the task system is allowed one degree of freedom is more interesting, and not quite as trivial as in the previous section. The results of this section are summarized in rows 5–7 of Table 1: observe that 5 of the 6 cases here are efficiently solvable while the sixth, rather surprisingly, is intractable. (We also point out here that the $\mathcal{O}(n \log n)$ complexity of each of the tractable problems is due to the complexity of sorting n numbers; if the tasks are available in sorted order according to their non-fixed parameter, each of these problems can be solved in $\mathcal{O}(n)$ time.)

5.1 Equal release times and execution requirements

We first consider task systems where all tasks (i) are released at the same instant, and (ii) have the same execution requirement. Without loss of generality, we assume that the common release time is 0, and let E denote the execution time of each task. The deadlines of different tasks may be different. (Since we are concerned with the off-line versions of the problem, in which all task parameters are known beforehand, the results here, by symmetry, apply also to the case when individual release times may differ, but all execution times are equal and all tasks have the same deadline.)

One processor. First, we consider the case when set of tasks $\tau = \{T_1, T_2, \ldots, T_n\}$, with $T_i = (0, E, d_i)$, are to be scheduled on a single

processor. Assume that the tasks are sorted by deadline (i.e., $d_i \leq d_{i+1}$ for all i) — given a set of n tasks, this can be achieved in $O(n \log n)$ time. Algorithm SCHEDULEPROC (Figure 4) generates a schedule for τ with the maximum possible minimum inter-completion time:

```
delmin ← ∞
for ℓ ← 2 to n do
    if (⌊(d_ℓ − E)/(ℓ − 1)⌋ ≤ delmin) then delmin ← ⌊(d_ℓ − E)/(ℓ − 1)⌋
    if (delmin < E) then return "not feasible"
for i ← 1 to n do
    schedule task T_i over interval [(i − 1) · delmin, (i − 1) · delmin + E)
```

Figure 4: Algorithm SCHEDULEPROC

Lemma 2 Let Δ be the largest number such that $\mathsf{mict}(\langle \tau, 1 \rangle) = \Delta$. Algorithm SCHEDULEPROC generates a schedule for τ on one processor with a minimum inter-completion time equal to Δ.

Proof: Let $d_1, d_2, \ldots, d_n$ denote the deadlines of the tasks, arranged in order of non-decreasing deadline. Observe first that Algorithm SCHEDULEPROC generates a schedule with a minimum inter-completion time equal to $\min_{\ell=2}^{n} \{\lfloor (d_\ell - E)/(\ell - 1) \rfloor\}$

Consider now any schedule that schedules all the tasks. Since the first completion time is $\geq E$, and the k'th is $\leq d_k$, it follows that the minimum inter-completion time on this schedule is no smaller than $\lfloor (d_k - E)/(k-1) \rfloor$ for each integer k, $2 \leq k \leq n$. ■

Observe that the run-time complexity of Algorithm SCHEDULEPROCS is $\mathcal{O}(n)$ if the tasks are already sorted by deadline. If the tasks are not already sorted, they can be sorted in $\mathcal{O}(n \log n)$ time.

Multiple processors. We now consider the case when $\tau = \bigcup_{i=1}^{n} \{T_i = (0, E, d_i)\}$ are to be scheduled on m processors, $m > 1$. Given such a system, Algorithm MULTIPROC (Figure 5) generates a schedule for τ with the maximum possible minimum inter-completion time.

Lemma 3 Let Δ be the largest number such that $\mathsf{mict}(\langle \tau, m \rangle) = \Delta$. Algorithm MULTIPROC generates a schedule for $\langle \tau, m \rangle$ with a minimum inter-completion time equal to Δ.

Before proving this Lemma, we need some auxiliary results. Let the deadlines of the n tasks, arranged in non-decreasing order, be $d_1, d_2, \ldots, d_n$.

Claim 3.1 Each k, $m < k \leq n$, imposes the restriction that

$$\Delta \leq \lfloor \frac{d_k - E}{\lceil k/m \rceil - 1} \rfloor \tag{1}$$

Proof: Observe that there are k tasks with deadline $\leq d_k$. By the pigeonhole principle, there is one processor which is assigned at least $\lceil k/m \rceil$ of these tasks. Since the first completion time on this processor is $\geq E$, and the completion time for each task with deadline no more than d_k is $\leq d_k$, it follows that the minimum inter-completion time is no more than $\lfloor (d_k - E)/(\lceil k/m \rceil - 1) \rfloor$. ■

We are now ready to prove Lemma 3.

Proof of Lemma 3: Suppose that Algorithm MULTIPROC generates a schedule with minimum inter-completion time $\Delta_{\min}$. Suppose that this minimum inter-completion time occurs on processor j, and is due to the assigning of the i'th-largest deadlined task[2]. That is,

$$\Delta_{\min} = \lfloor \frac{d_i - E}{n_{i,j} - 1} \rfloor$$

where $n_{i,j}$ denotes the number of tasks with deadline $\leq d_i$ that have been assigned to processor j in Step 2 of Algorithm MULTIPROC. Since the tasks are assigned to the processors in round-robin order, it is clear that exactly $\lceil i/m \rceil$ of the first i tasks are assigned to processor j; i.e., $n_{i,j} = \lceil i/m \rceil$. Therefore,

$$\Delta_{\min} = \lfloor \frac{d_i - E}{\lceil i/m \rceil - 1} \rfloor \tag{2}$$

By setting k in Equation 1 to i, it follows that no schedule can obtain a larger minimum inter-completion time. ■

Run-time complexity. Step 1 takes $\mathcal{O}(n \log n)$ time. Step 2 takes $\mathcal{O}(n)$ time. Let n_j denote the number of tasks allocated to processor j, $1 \leq j \leq m$, in Step 2. Step 3 requires calls to Algorithm SCHEDULEPROC on sets of tasks that are already sorted by deadline. The total complexity of this step is therefore $\sum_{j=1}^{m} \mathcal{O}(n_j)$, which is equal to $\mathcal{O}(n)$. The dominant step is therefore Step 1, and the total complexity is $\mathcal{O}(n \log n)$.

5.2 Equal release times and deadlines

When all the release times and execution requirements are equal (or, by symmetry, when all the deadlines and execution requirements are equal),

[2]It can be shown that j is necessarily 1; however, this fact does not concern us here.

1. Sort the tasks by deadline.
2. Assign the tasks, considered in deadline order, to the processors in a round-robin fashion. That is, let $T_1, T_2, \ldots, T_n$ denote the tasks sorted by deadline. Assign the tasks to the m processors, as follows:

   $j \leftarrow 1$
   for $i \leftarrow 1$ to n do
   assign T_i to the j'th processor
   if $(j < m)$ then $j \leftarrow j + 1$ else $j \leftarrow 1$

3. Schedule each processor individually, by Algorithm SCHEDULEPROC.

Figure 5: Algorithm MULTIPROC

we have seen that the problem of scheduling to maximize minimum intercompletion can be very efficiently solved on any number of processors. We will now see that, when execution requirements may vary while release times and deadlines are fixed, the situation is not quite the same.

Theorem 4 Let $\tau = \cup_{i=1}^{n} \{T_i = (0, e_i, D)\}$ be a task system. The problem of MICT-scheduling τ on m processors is NP-hard, for arbitrary m.

Proof Sketch: We prove this theorem by showing that there is a polynomial transformation from the NP-hard problem of multiprocessor scheduling to MICT-scheduling with equal release times and deadlines.

The **multiprocessor scheduling problem** is defined as follows [2, page 238]:

> INSTANCE: Set T of tasks, number $m \in Z^+$ of processors, length $\ell(t) \in Z^+$ for each $t \in T$, and a deadline $D \in Z^+$.
> QUESTION: Is there an m-processor schedule for T that meets the overall deadline D, i.e., a function $\sigma : T \rightarrow Z_0^+$ such that, for all $u \geq 0$, the number of tasks $t \in T$ for which $\sigma(t) \leq u < \sigma(t) + \ell(t)$ is no more than m and such that, for all $t \in T$, $\sigma(t) + \ell(t) \leq D$?

Given an arbitrary instance of the multiprocessor scheduling problem, we obtain an instance of the problem of MICT-scheduling with equal release times and deadlines, by the following mechanism: for each task t in the multiprocessor scheduling problem instance, we define an MICT task with release-time 0, deadline D, and execution requirement $\ell(t)$. It is relatively straightforward to observe that this system of MICT tasks can be scheduled with an intercompletion time ≥ 0 if and only if the multiprocessor scheduling problem instance has a solution. ■

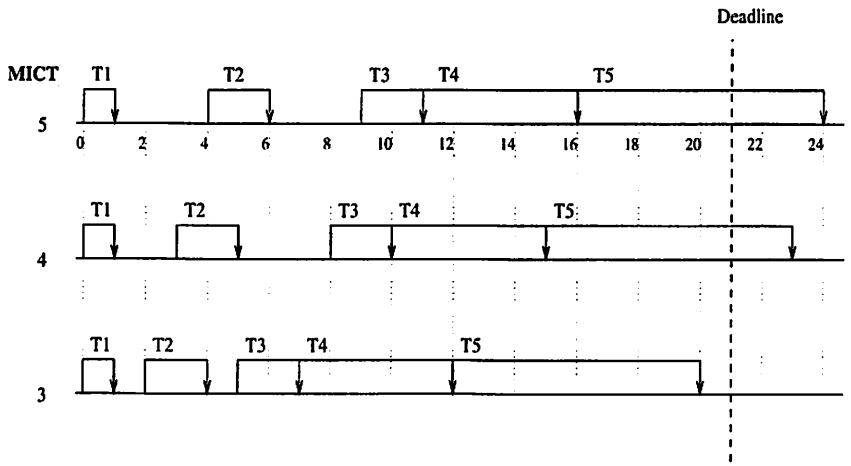
Algorithm EQUALEXUNIPROC
  $delmin \leftarrow \infty$
  for $\ell \leftarrow n$ down to 2 do
    if $(\lfloor (D - e_1 - \sum_{j=\ell+1}^{n} e_j)/(\ell - 1) \rfloor \leq \texttt{delmin})$
      then $\texttt{delmin} \leftarrow \lfloor (D - e_1 - \sum_{j=\ell+1}^{n} e_j)/(\ell - 1) \rfloor$
  schedule task T_1 over $[0, e_1)$
  $c \leftarrow e_1$ /* last completion time */
  for $i \leftarrow 2$ to n do
    if $(e_i \leq \texttt{delmin})$ {
      schedule task T_i over the interval $[c + \texttt{delmin} - e_i, c + \texttt{delmin})$
      $c \leftarrow c + \texttt{delmin}$ }
    else {
      schedule task T_i over the interval $[c, c + e_i)$
      $c \leftarrow c + e_i$ }

Figure 6: Algorithm EQUALEXUNIPROC

However, the situation is not quite as bleak on a single processor. Assume that the tasks are sorted by execution requirement (i.e., $e_i \leq e_{i+1}$ for all i) — given a set of n tasks, this can be achieved in $\mathcal{O}(n \log n)$ time. Observe that an MICT-schedule would have a task with smallest execution requirement (without loss of generality, T_1) scheduled over the interval $[0, e_1)$, and that this would leave the interval $[e_1, D)$ to be partitioned into $n - 1$ inter-completion intervals. We want $[e_1, D)$ to be partitioned as evenly as possible, subject to the constraint that each $e_i, i > 1$, has to "fit" within an interval. Algorithm EQUALEXUNIPROC (Figure 6) generates such a schedule, in which the order of task-execution is $T_1, T_2, \ldots, T_n$. (The first for-loop accounts for the possibility that some of the later tasks have very large execution requirements, thus forcing the remaining tasks closer together.)

Example 3 Consider a set τ of 5 tasks, with $r_i = 0$ for all tasks, $d_i = 21$ for all tasks (i.e., $D = 21$),, and $e_1 = 1$, $e_2 = 2$, $e_3 = 2$, $e_4 = 5$, and $e_5 = 8$. We trace below the execution of Algorithm EQUALEXUNIPROC on τ: for each iteration of the first for loop, we indicate how the value of `delmin` gets updated. (The figure shows that $\texttt{delmin} = 5$ or 4 are unacceptable, illustrating the need to loop through $\ell = 5, 4, 3, 2$ to determine the optimal `delmin`.)

ℓ	$\lfloor (D - e_1 - \sum_{j=\ell+1}^{n} e_j)/(\ell - 1) \rfloor$	delmin
		∞
5	$\lfloor (21 - 1 - 0)/(5 - 1) \rfloor = 5$	5
4	$\lfloor (21 - 1 - 8)/(4 - 1) \rfloor = 4$	4
3	$\lfloor (21 - 1 - 13)/(3 - 1) \rfloor = 3$	3
2	$\lfloor (21 - 1 - 15)/(2 - 1) \rfloor = 5$	3



■

Theorem 5 Let Δ be the largest number such that $\text{mict}(\langle \tau, 1 \rangle) = \Delta$. Algorithm EQUALEXUNIPROC generates a schedule for $\langle \tau, 1 \rangle$ with a minimum inter-completion time equal to Δ.

Proof Sketch: Similar to the proof of Lemma 2. ■

If the tasks are already sorted by execution requirement, the run-time complexity of Algorithm EQUALEXUNIPROC is $\mathcal{O}(n)$. Since sorting can be done in $\mathcal{O}(n \log n)$ time, the total complexity of MICT-scheduling a set of tasks with equal release times and deadlines on one processor is $\mathcal{O}(n \log n)$.

6 Task systems with two degrees of freedom

In Sections 4 and 5, we considered task systems with zero and one degrees of freedom. All of these were relatively straightforward to analyze and, with the exception of Theorem 4, could be solved from first principles. Task systems with *two* degrees of freedom — the subject of this section — are a lot more challenging. For the uniprocessor problems, we make use of the reduction defined in Section 3 to transform MICT-scheduling to

(general) non-preemptive scheduling, and then design efficient solutions to the resulting scheduling problems. Each of the three cases (Sections 6.1.1 - 6.1.3) require a fresh approach that differs significantly from the ones employed in the other two. Two of the three multiprocessor problems turn out to be intractable; the complexity of the third remains unresolved.

6.1 One Processor

Let $\tau = \bigcup_{i=1}^{n} \{T_i = (r_i, e_i, d_i)\}$ be a set of tasks to be scheduled on a single processor. Let $\hat{d} \stackrel{\text{def}}{=} \max_{i=1}^{n} \{d_i\}$, and assume without loss of generality that $\min_{i=1}^{n} \{r_i\} = 0$. Let $\Delta_{\max}$ denote the largest integer Δ for which $\text{mict}(\langle \tau, 1 \rangle) = \Delta$. Observe that $\lfloor \hat{d}/(n-1) \rfloor$ is a (loose) upper bound on the value of $\Delta_{\max}$. The aim in MICT-scheduling is to generate a schedule with a minimum inter-completion time equal to $\Delta_{\max}$.

Suppose now that we had an algorithm that, given τ and a positive integer Δ, determines whether τ can be scheduled with a minimum inter-completion time of at least Δ on a single processor (i.e., whether $\text{mict}(\langle \tau, 1 \rangle) = \Delta$); if so, it generates a schedule with minimum inter-completion time at least Δ. Then an MICT-schedule for τ — i.e., a schedule with minimum inter-completion time equal to $\Delta_{\max}$ — can be obtained by making $\mathcal{O}(\log \lfloor \hat{d}/(n-1) \rfloor)$ calls to this algorithm, by essentially performing "binary search" between the values 0 and $\lfloor \hat{d}/(n-1) \rfloor$. Since $\mathcal{O}(\log \lfloor \hat{d}/(n-1) \rfloor) = \mathcal{O}(\log \hat{d})$, the complexity of MICT-scheduling τ is therefore $\mathcal{O}(\log \hat{d})$ times the complexity of generating a schedule with a specified minimum inter-completion time Δ, if it exists[3].

In the remainder of Section 6.1, we consider separately the three cases when τ is restricted in one of its degrees of freedom – fixed deadlines (Section 6.1.1), fixed release times (Section 6.1.2), and fixed execution requirements (Section 6.1.3). For each, we make use of the reduction defined in Theorem 3 to design an efficient algorithm which accepts as input a constrained task system τ and a positive integer Δ and, if $\text{mict}(\langle \tau, 1 \rangle) = \Delta$, generates a schedule for τ with minimum inter-completion time of at least Δ.

6.1.1 Fixed deadlines

When all the deadlines are equal, we may make use of Theorem 3 to reduce MICT-scheduling on a single processor to a tractable problem in (regular) scheduling.

[3]Observe that $\log \hat{d}$ is polynomial in the size of the binary representation of τ; MICT-scheduling is therefore a polynomial-time operation, provided the problem of generating a schedule with specified minimum inter-completion time is in PTIME.

Let $\tau = \cup_{i=1}^{n}\{(T_i = (r_i, e_i, D)\}$ be a task system in which all tasks have the same deadline D, and let Δ be a given positive integer. We apply the reduction r defined in Theorem 3 to τ, yielding the taskset $r(\tau, \Delta) \stackrel{\text{def}}{=} \cup_{i=1}^{n}\{(T_i' = (r_i - \delta_i, e_i + \delta_i, D)\}$, where $\delta_i \stackrel{\text{def}}{=} \max(0, \Delta - e_i)$. By Theorem 3, τ has a schedule with minimum intercompletion time Δ if and only if $r(\tau, \Delta)$ is feasible.

Since each task in $r(\tau, \Delta)$ has the same deadline D, it is trivial to determine whether $r(\tau, \Delta)$ is feasible, and to generate a schedule if the answer is yes: simply schedule the tasks according to earliest release times (ties broken arbitrarily), and report success if they all complete by time D, and failure otherwise. The run-time complexity is $\mathcal{O}(n \log n)$, with the dominant cost being the cost of sorting the tasks by order of non-decreasing release times. The overall complexity of determining $\Delta_{\max}$ is therefore $\mathcal{O}(n \log n \log \hat{d})$.

6.1.2 Fixed release times

When all the release times are equal, a technique similar to the one used in Section 6.1.1 may be used.

Let $\tau = \cup_{i=1}^{n}\{(T_i = (0, e_i, d_i)\}$ be a task system in which all tasks have the same release time (without loss of generality, we have assumed that this release time is 0). Let Δ be a given positive integer. We once again apply the reduction r defined in Theorem 3 to τ, yielding the taskset $r(\tau, \Delta) \stackrel{\text{def}}{=} \cup_{i=1}^{n}\{(T_i' = (-\delta_i, e_i + \delta_i, d_i)\}$, where $\delta_i \stackrel{\text{def}}{=} \max(0, \Delta - e_i)$. By Theorem 3, τ has a schedule with minimum intercompletion time Δ if and only if $r(\tau, \Delta)$ is feasible.

Observe that each task in $r(\tau, \Delta)$ may have a different release time, execution requirement, and deadline. Scheduling such systems is, in general, NP-hard in the strong sense (*Sequencing with release times and deadlines* [2, page 236]). Fortunately, $r(\tau, \Delta)$ is not quite general – notice that the interval between the release time $-\delta_i$, and the instant zero, is no larger than the execution requirement $e_i + \delta_i$, for every task T_i. We may therefore conclude that at most one task executes before time-instant zero in any schedule for $r(\tau, \Delta)$. In the pseudocode below, each iteration of the for loop "guesses" a different candidate T_ℓ for this first task. The rest of the tasks are all available by the time T_ℓ completes execution, and may therefore be executed in deadline order. Since some task T_j must execute first in a schedule for $r(\tau, \Delta)$, this algorithm will discover the schedule during the j'th iteration.

1 Assume that the tasks are available in order of non-decreasing deadlines
2 for $\ell \leftarrow 1$ to n do{

```
    /*Task T_ℓ is executed first */
3   execute task T_ℓ over the interval [−δ_ℓ, e_ℓ)
4   execute the remaining tasks in EDF order
5   if all tasks meet their deadlines return "success"
  }
```

The run-time complexity may be computed as follows: It costs $\mathcal{O}(n \log n)$ to sort the tasks by deadline (line 1). Each iteration of the for loop (lines 2–5) takes $\mathcal{O}(n)$ time, and there could be up to n iterations, for a total complexity of $\mathcal{O}(n \log n + n^2)$, which equals $\mathcal{O}(n^2)$. The overall complexity of determining $\Delta_{\max}$ is therefore $\mathcal{O}(n^2 \log \hat{d})$.

6.1.3 Equal execution requirements

Let $\tau = \cup_{i=1}^{n} \{(T_i = (r_i, E, d_i)\}$ be a task system in which all tasks have the same execution requirement E, which we wish to schedule on a single processor. Let Δ be a given positive integer. We make use of the following result from [6] in determining whether $\text{mict}(\langle \tau, 1 \rangle) = \Delta$:

Result 1 (Simons (1978)) Let τ be a set of n tasks, in which all tasks have the same execution requirement. Simons presented an $\mathcal{O}(n^2 \log n)$ algorithm to determine if τ can be non-preemptively scheduled on a single processor, and to generate such a schedule if it exists. We will refer to this algorithm as *Simons' Algorithm.*

We apply the reduction r defined in Theorem 3 to τ, yielding the taskset $r(\tau, \Delta) \stackrel{\text{def}}{=} \cup_{i=1}^{n} \{(T_i' = (r_i - \delta, E + \delta, d_i)\}$, where $\delta \stackrel{\text{def}}{=} \max(0, \Delta - E)$. By Theorem 3, τ has a schedule with minimum intercompletion time Δ if and only if $r(\tau, \Delta)$ is feasible.

The crucial observation is that the execution requirements of all tasks in $r(\tau, \Delta)$ are equal. We can therefore use Simons' Algorithm to determine in $\mathcal{O}(n^2 \log n)$ time if $r(\tau, \Delta)$ is feasible, and to generate a schedule if so. The total complexity of determining $\Delta_{\max}$ is therefore $\mathcal{O}(n^2 \log n \log \hat{d})$.

6.2 Multiple processors

While all the problems studied above are seen to have efficient solutions, two of the corresponding problems on multiple processors turn out to be intractable. The complexity of the third remains open.

Theorem 6 Let $\tau = \cup_{i=1}^{n} \{T_i = (r_i, e_i, D)\}$. The problem of MICT-scheduling τ on m processors in NP-hard, for arbitrary m.

Proof: Directly follows from Theorem 4. ■

Theorem 7 Let $\tau = \cup_{i=1}^{n} \{T_i = (0, e_i, d_i)\}$. The problem of MICT-scheduling τ on m processors in NP-hard, for arbitrary m.

Proof: Directly follows from Theorem 4. ■

7 Conclusions

While scheduling application systems, it is sometimes advantageous to spread out over time the instants at which different tasks complete execution. We have formalized this idea into the concept of *scheduling to maximize inter-completion time* — MICT-scheduling. We have shown that the MICT-scheduling problem is, in general, NP-hard, and have studied a wide variety of special cases of task systems, where each special case is distinguished by being restricted along some of its degrees of freedom. For a large number of these special cases, we present very efficient scheduling algorithms; others we prove NP-hard. In addition, we have identified (Theorem 3) a very useful relationship between MICT-scheduling and the well-understood problem of scheduling to meet deadlines. Our current research efforts include using this relationship to obtain MICT-scheduling algorithms for other problems which have tractable corresponding feasibility problems.

In this research, we have chosen to focus on maximizing the minimum inter-completion time *on each processor* — in the Introduction, we explained why we were directed to this metric by the application systems under consideration. An alternative approach could be to maximize the minimum inter-completion time over *all* processors. That is, we could have attempted to maximize the minimum difference between the completion times of successive tasks to complete, irrespective of the processor each task executes on. This metric would be more appropriate in a situation where, for example, each completion is followed by some interprocessor communication, and the idea is to balance out the network load over time. We are currently exploring issues relating to this "global" minimum inter-completion time metric — it appears that techniques fundamentally different from the ones used in this paper are needed.

8 Acknowledgements

We are indebted to generous support partially provided for this work under the U.S. ONR Grants N00014-92-J-1367 and N00014-93-1-1047, and NSF grants CCR-9596282 and CCR-9402827. Useful, constructive and valuable criticism and ideas have been provided by members of the Real-Time Computing Laboratory.

References

[1] M. Garey and D. Johnson. Two-processor scheduling with start-times and deadlines. *SIAM Journal of Computing*, 6:416–426, 1977.

[2] M. Garey and D. Johnson. *Computers and Intractability : a Guide to the Theory of NP-Completeness.* W. H. Freeman and company, NY, 1979.

[3] R. Graham. Bounds for certain multiprocessing anomalies. *Bell System Technical Journal*, 45:1563–1581, 1966.

[4] R. Graham. Bounds on multiprocessor timing anomalies. *SIAM Journal on Applied Mathematics*, 17:416–429, 1969.

[5] D. S. Hochbaum and D. B. Shmoys. A unified approach to approximation algorithms for bottleneck problems. *Journal of the ACM*, 33:533–550, 1986.

[6] B. Simons. A fast algorithm for single processor scheduling. In *Proceedings of the Ninteenth Annual Symposium on Foundations of Computer Science*, 1978.