Pre-Scheduling: Balancing between Static and Dynamic Schedulers *

Weirong Wang Aloysius K. Mok

Department of Computer Sciences
University of Texas at Austin
Austin, Texas 78712-1188
{weirongw, mok}@cs.utexas.edu

Abstract

Compared with dynamic schedulers, static schedulers require lower on-line scheduling costs but higher demand on resources in capacity planning, especially if there are sporadic tasks whose deadlines are relatively short compared with their mean interval of arrival. A hybrid scheduling approach is proposed in this paper to strike a balance between static and dynamic schedulers for real-time systems with both periodic and sporadic tasks. The hybrid approach is based on the technique we call pre-scheduling. Like a static schedule, a pre-schedule of periodic tasks is produced off-line but unlike a static schedule, a pre-schedule has built in slack for the periodic tasks so that their start-times are not fixed. Pre-schedules are used on-line with two on-line schedulers. A dynamic on-line scheduler schedules sporadic tasks at high priority, and a static on-line scheduler schedules the periodic tasks according to the pre-schedule at low priority. The challenge of this approach is that the arrival times of sporadic tasks are not known *a priori*, yet a pre-schedule must be produced off-line to make sure that a valid schedule is always produced by the on-line schedulers. In this paper, this challenge is answered with an optimal and computationally practical pre-scheduler. The pre-scheduling approach augments and extends the capability of architectures such as TTA (Time Triggered Architecture) and is important for the design of mechanically certificable safety-critical embedded systems that have both time-driven and event-driven tasks with short deadlines.

*This work is supported in part by a grant from the US Office of Naval Research under grant number N00014-99-1-0402 and N00014-98-1-0704, and by a research contract from SRI International under a grant from the NEST program of DARPA

1 Introduction

1.1 Motivation

Both static schedulers (in the form of cyclic executives) and dynamic schedulers (mostly fixed priority schedulers) are widely used in real-time systems. During run-time, a static scheduler dispatches tasks by following a static schedule which is computed off-line and is stored as a list of entries in main memory. Each entry in the list defines a start time and duration allocated to some task, and a cursor points to the "current" entry in the schedule. After a task is dispatched according to the current entry, the cursor moves to the next entry. When the end of the static schedule is reached, the cursor is reset to the beginning of the list. Dynamic schedulers, on the contrary, do not depend on a precomputed off-line schedule and instead make all scheduling decisions at run-time based on the states and timing attributes of the ready tasks.

Both static schedulers and dynamic schedulers have their own advantages and disadvantages. Static schedulers usually have lower on-line scheduling costs since the only scheduler overhead is in moving the cursor down the static schedule. It should also be noted that the on-line scheduling costs include not only computation costs but also information distribution costs and other considerations such as fault tolerance. For instance, the fixed priority scheduling policy assumes a global ordering of priorities but in practice tasks are frequently assigned to processors in a fixed assignment to avoid paying the cost of global priority comparisons, even though a fixed assignment of tasks to processors may not be consistent with global task priorities. On the other hand, static schedulers may impose a higher demand on processor capacity because enough time slots must be reserved for "sporadic" tasks in a static schedule so that a sporadic task can be guaranteed CPU time regardless of when it requests for service. For sporadic tasks that demand immediate attention but request for CPU time only rarely, the static scheduler must "sample" frequently enough to meet their deadlines. Therefore, there is a trade-off between demand on resource capacity and the costs of on-line scheduling. We illustrate this trade-off with the following simple real-time system design example.

In this paper, we shall investigate a hybrid approach that strikes a balance between static and dynamic schedulers in a real-time system with both periodic and sporadic tasks. The hybrid approach is based on a technique we call pre-scheduling. Like a static schedule, a pre-schedule of periodic tasks is produced off-line but unlike a static schedule, a pre-schedule has built-in slack for the periodic tasks so that their start-times are not fixed. Pre-schedules are used on-line with two on-line schedulers: a dynamic on-line scheduler schedules sporadic tasks at higher priority, and a static on-line scheduler schedules the periodic tasks according to the pre-schedule at lower priority.

Example 1 There are three nodes in a system and each nodes may send a stream of requests. Each request needs 1 unit of time of a resource, say, the CPU. Stream 0 and 1 are synchronous periodic streams, which means the first request of each stream arrives at time 0, and subsequent requests arrive at a regular time interval called "period". The period of Stream 0 is 3, and the period of Stream 1 is 4. A request from each of the streams must be processed before the arrival of the next request from that stream. Stream 2 is an aperiodic stream; the exact arrival time of each request is unknown *a priori*, but it is known that at least 5 time units must separate consecutive arrivals. Each request from Stream 2 must be processed as soon as it arrives. Assume that arrival times of requests are integers. ■

Figure 1 illustrates three different scheduler designs for the streams in this example. The first design requires two copies of the resource and uses a static scheduler to schedule each resource. One resource is dedicated to Stream 2. This is because there must be one resource ready to be scheduled to Stream 2 for every time unit. Stream 0 and Stream 1 share the other resource according to the static schedule shown in the figure. A static schedule is given by a tuple of two elements: length of a time frame and a list of entries. The schedule iterates once in every time frame. Each entry defines which stream obtains the resource at which time unit (starting from 0) in the frame. Stream 0 uses the resource at time 0 of each frame, Stream 1 uses the resource at time 1 of each frame, and so forth.

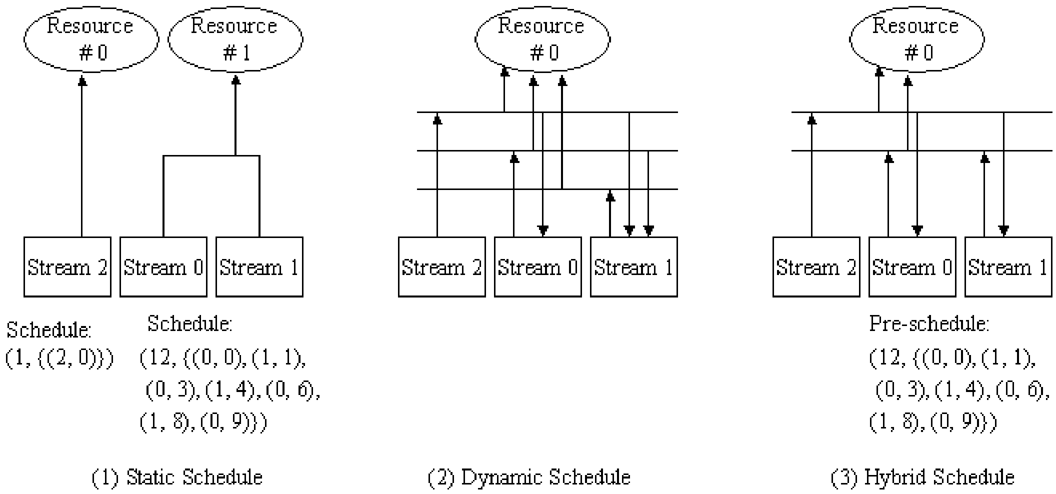

Figure 1: Comparison of Schedulers

The second design in Figure 1 uses a fixed priority scheduler. There is only one copy of the resource and there are 3 fixed task priorities. Stream 2 has the highest task priority, which means that if Stream 2 issues a request, it obtains the resource immediately, and Stream 0 and Stream 1 which are informed of the request, will be blocked until the current request from Stream 2 is completed. Likewise, Stream 1 has a higher priority than Stream 0. Notice that if the requests from the streams have to travel a considerable distance to the resource scheduler, then the static scheduler of the first design may be preferable since each stream may consult its own copy of the static schedule and claims the resource only when it is its turn, and no priority comparison is necessary.

A hybrid design, shown as the third choice in Figure 1, requires fewer copies of the resource than the static scheduler and also fewer levels of priorities than the dynamic scheduler. In this design, there are two priorities: Stream 2 has the higher priority and Stream 0 and Stream 1 share the lower priority. A pre-schedule for Stream 0 and Stream 1 is computed off-line. There is a difference between a schedule and a pre-schedule: the time in each tuple of a schedule is the exact time a stream obtains a resource, but the time in a pre-schedule is the earliest time that a stream can obtain the resource. The exact schedule also depends on the arrival times of the dynamic requests. For example, if no request arrives in the time interval $[0, 12)$, then the schedule in this time interval is as follows: $(0,0),(1,1),(0,3),(1,4),(0,6),(1,8),(0,9)$. This schedule is illustrated by Figure 2

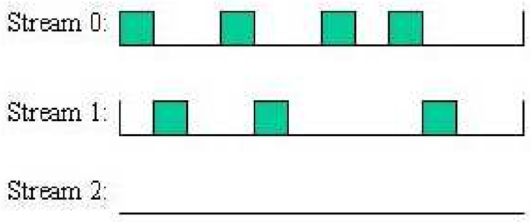

Figure 2: Pre-schedule 1 for the Hybrid Scheduler

If two requests of Stream 2 arrive at time 3 and time 8, then the hybrid scheduler will produce the schedule between time=0 and time=12 as follows: $(0,0),(1,1),(2,3),(0,4),(1,5),(0,6),(2,8),(1,9),(0,10)$; here the first element in each tuple is a stream number, and the second element is the time the stream occupies the resource. This schedule is illustrated by Figure 3.

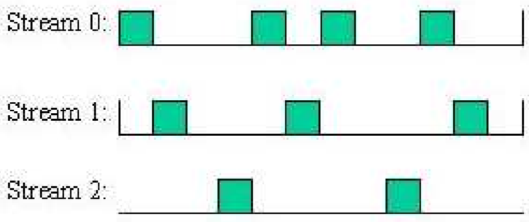

Figure 3: Pre-schedule 2 for the Hybrid Scheduler

Notice that Stream 2 needs only to notify Stream 0 and Stream 1 of its request and proceeds to use the resource; and Stream 0 and Stream 1 may access the resource according to the pre-schedule as long as there is no request from Stream 2. In general, the two-level priorities of the hybrid approach provides a balance between purely static and purely dynamic approaches. This tradeoff is illustrated by the shadowed area in Figure 4 which delimits the feasible design space, bounded by the maximal on-line scheduling costs and the demand on resource capacity of the application. It might be the case that only hybrid schedulers fall inside the feasible design space for a given availability of the resource.

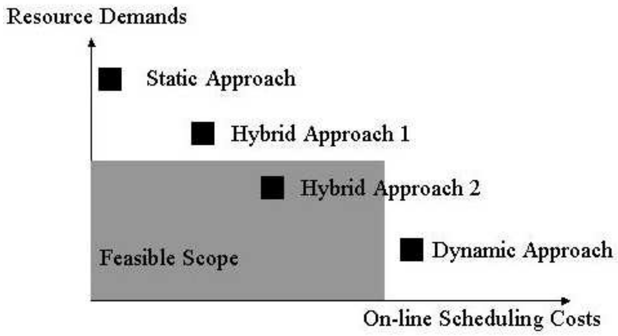

Figure 4: Balance between Resource Requirements and On-line Scheduling Costs

The pre-schedule plays critical role in the hybrid approach. and we call this approach to scheduler design the pre-scheduling approach. More importantly, the pre-scheduling approach represents a way to augment and extend the capability of architectures such as TTA (Time Triggered Architecture). Because of its static nature, TTA has mechanically certifiable fault-tolerance as well as timeliness properties that are required by highly safety-critical applications, but its cyclic-executive-like architecture has its limitations [15] in flexibility when compared with dynamic schedulers. The pre-scheduling approach is amenable to mechanical certification and also provides more flexibility than the cyclic executive.

1.2 Framework of the Pre-Scheduling Approach

The pre-scheduling approach, as illustrated in Figure 5, can be summarized as follows. We assume that the system has both periodic tasks and sporadic tasks. A pre-schedule for periodic tasks is produced off-line. The pre-scheduler consists of two algorithms: the Planner and the Weight Assigner. The Planner produces a "plan" which defines a search space for finding a pre-schedule. The Weight Assigner produces a pre-schedule according to the plan. The on-line scheduler consists of a dynamic scheduler, which schedules all sporadic tasks at higher priority, and a static scheduler, which schedules all periodic tasks at lower priority. The challenge of this approach is that the arrival times of sporadic tasks are not known *a priori*, yet a pre-schedule must be produced off-line to make sure that a feasible schedule is always produced by the on-line schedulers. In this paper, this challenge is answered with an optimal and computationally practical pre-scheduler.

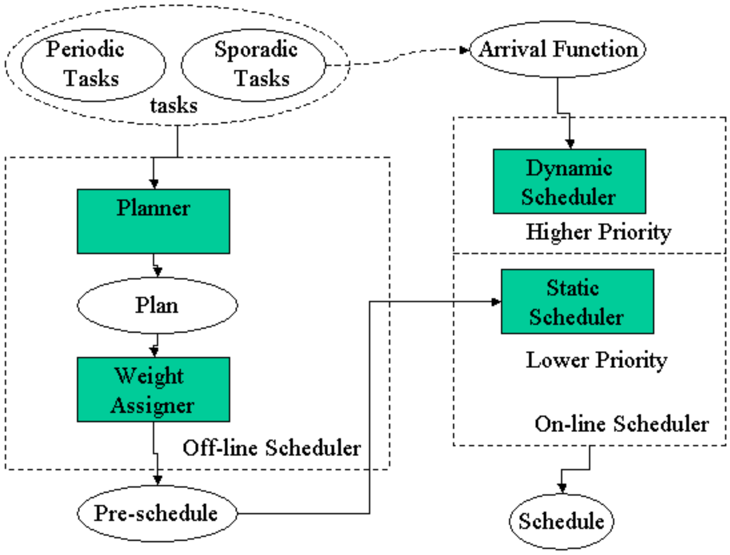

Figure 5: The Pre-scheduling Framework

1.3 Paper Organization

The rest of the paper is organized as follows. Section 2 provides references to related previous works. Section 3 defines the main elements in the framework. The algorithms in the pre-scheduler, Planner and Weight Assigner are defined and analyzed in Section 4 and Section 5 respectively. Section 6 illustrates the working of the pre-scheduler with an example. Section 7 concludes the paper and proposes future work.

1.4 Miscellaneous

- Owing to the page limitation, the full proofs of many lemmas, theorems and properties are not presented in this paper. The details are available in a technical report accessible at http://www.cs.utexas.edu/users/mok/RTS/pubs.html.
- The following two types of notations are frequently used in this paper.
  - Given that X identifies an object and Y is one of the attributes of the object, we use the notation $X.Y$ to denote the attribute Y of X. For instance, if (i, j) identifies a sporadic job, then $(i, j).d$ denotes the deadline of job (i, j).
  - Suppose L is a list of enumerable objects such as a list of jobs, and X and Y are elements in the list, and X is not after Y. We use $[X, Y]$ to denote the sub-list of L between and including X and Y; the list (X, Y) is the sub-list of L between and excluding X and Y; the list $[X, Y)$ is

the sub-list of Li between X and Y, including X and excluding Y; the list $(X, Y]$ is the sub-list of Li between X and Y, excluding X but including Y.

2 Previous Work

In this paper, we combine some previous techniques and propose new techniques to solve the pre-scheduling problem. We summarize the related previous work below.

- **Worst-Case Response Time/Time-Demand Analysis** The constraint analysis in this paper benefits from previous works on worst-case response time, time-demand analysis, and slack stealing analysis. The feasibility analysis for a mixture of periodic and sporadic tasks under the dynamic scheduling regime is proved by [5] to be in co-NP. Pseudo-polynomial feasibility tests based on the same assumptions are provided by [5] and [18]. Feasibility test techniques on static priority scheduling under a variety of assumptions on deadline, release jitter, offset, blocking, etc. are discussed in [1] and [2]. A schedulability test for rate monotonic scheduling is given in [11]. a schedulability test with arbitrary relative deadlines (especially, the case where deadlines may be longer than the period) is discussed in [12]. Techniques on static priority scheduling is reviewed in [3].

  When a set of aperiodic tasks are scheduled together with a set of periodic tasks, slack stealing can be used to reduce the response times of aperiodic jobs without making periodic tasks miss their deadlines. A locally optimal slack stealing method for static priority scheduling is described in [13]. An on-line acceptance test for aperiodic tasks with hard-deadline is further defined in [14], and it shows that there is no optimal priority assignment for hard aperiodic tasks. The non-existence of globally optimal slack stealing algorithm for fixed-priority preemptive scheduling is shown in [23]. A slack stealing algorithm is described in [22] which is based on dynamic priority (EDF) scheduler with preemption for periodic tasks.

- **Decompositional Approaches** The plan construction described in this paper is similar to the decompositional approaches for non-preemptive scheduling described in [8] and [25]. These previous papers follow a line of investigation aiming at reducing the number of job sequences to be searched in finding a non-preemptive scheduler. However, sporadic tasks are not considered in these models. We adopt a similar approach for our model.

- **Parametric Scheduling** A parametric scheduling method to solve non-preemptive periodic task scheduling is described in [9] and [7]. This model assumes that the execution times of jobs of a task may vary at run-time. The scheduler consists of an off-line component and an on-line component. The off-line component computes a total order of the tasks and two functions to determine the lower and upper bounds of the start time on-line, and the on-line component generates the schedule according to the total order, the bounds and the actual execution times of the jobs. Sporadic tasks are not considered in [9] and [7]. Our analysis methods are quite different for those used in [9] and [7] and deal with a combination of periodic and sporadic tasks.

- **Scheduling a Combination of Periodic and Aperiodic Tasks** Besides the papers [5, 18] we mentioned earlier, previous works in this area also include Deferrable Server [10], Sporadic Server [20], and the algorithms described in [19]. Major differences between these previous works and the pre-scheduling approach include the follows:
  - These previous works focus on improving the responsiveness of aperiodic tasks while not violating the deadlines of periodic tasks. The focus of pre-scheduling approach is to reduce the on-line scheduling costs while meeting the deadlines of all periodic and sporadic tasks.
  - These previous schedulers are dynamic schedulers. The pre-scheduling approach is a hybrid of dynamic and static schedulers.
  - The pre-scheduling approach proposed in this paper involves a set of solution techniques that are different from previous approaches.
- **Resource Partitioning and Hierarchical Schedulers** In the pre-scheduling approach, tasks are divided into a periodic subset and a sporadic subset, and each set is scheduled on-line by their dedicated scheduler. In this respect, the pre-scheduling approach has some similarities to the ideas of resource partitioning and hierarchical schedulers, as exemplified by the papers [16, 17]. The primary goal of resource partitioning and hierarchical schedulers is to make individual application or schedulers work as if each one of them owns a dedicated resource, while the goal of pre-scheduling approach is to balance the on-line scheduling costs and the demand on resource capacity.

3 The Pre-scheduling Framework

In this section, we describe and analyze the main elements in the pre-scheduling framework other than the off-line pre-scheduler.

3.1 Tasks

First, we define some concepts related to the tasking model.

Time has as its domain the non-negative real numbers. As usual, the time interval between time b and time e is denoted by (b, e). For scheduling purposes, we shall partition the time line into an infinite number of *frames*. All frames have the same length F. The jth frame is the time interval $(j \cdot F, (j+1) \cdot F)$, for $j \geqslant 0$.

A *resource* is an object to be allocated to tasks. It could be a CPU, a bus, or a packet switch, etc. For this paper, we consider the case of a single resource which can be shared by the tasks with preemption allowed. We assume that context switching takes no time (in practice, this assumption can be taken care of by adding a context switch cost to the basic time quantum for resource allocation but advanced optimization techniques can often minimize this cost in static schedulers).

A *job* is defined by a tuple of three attributes (c, r, d) which are non-negative real numbers:

- c is the *execution time* of a job, which defines the amount of time that must be allocated to the job.
- r is the *ready time* or *arrival time* of the job, which is the earliest time the job can be scheduled.
- d is the *deadline* of the job, which is the latest time by which the job must be completed.

A *task* is an infinite sequence of jobs. Each task is identified by a unique ID i. A task is either periodic or sporadic.

The set of periodic tasks in a system is represented by T_p. A *periodic task* is denoted by $(i, (c, r, d))$, where i identifies the task, and tuple (c, r, d) defines the attributes of its jobs, as follows. There is one and only one job arrival in each frame for each periodic task. The job of periodic task i in the jth frame is called the jth job of i, denoted by job (i, j). The attributes in the definition of a periodic task, c, r and d, are non-negative real numbers:

- c is the *execution time* of a task, which defines the amount of time that must be allocated to each job of the task.
- r is the *ready time* or *arrival time* of the task, which is the earliest time the task can be scheduled in a frame. For every periodic task, $0 \leqslant r < F$.
- d is the *deadline* of the task, which is the latest time by which the task must be completed after its arrival. For every periodic task, $c \leqslant d$ and $r + d < F$.

If periodic task i is defined by (c, r, d), job (i, j) is defined by $(c, j \cdot F + r, j \cdot F + r + d)$.

We define some relations between a pair of periodic tasks as follows. Given a pair of periodic tasks i and i',

- i is *parallel to* i' if and only if $i.r = i'.r$ and $i.d = i'.d$. Without losing generality, we shall assume in the rest of this paper that no periodic task is parallel to another periodic task, since for scheduling purposes, we can always treat a pair of parallel periodic tasks as equivalent to one periodic task whose execution time is the sum of the execution times of the pair.

- periodic task i is *before* periodic task i' if and only if at least one of the following conditions is true:
  - $i.r < i'.r$ and $i.r + i.d \leqslant i'.r + i'.d$, or
  - $i.r \leqslant i'.r$ and $i.r + i.d < i'.r + i'.d$.
- periodic task i *contains* periodic task i' if and only if $i.r < i'.r < i'.r + i'.d < i.r + i.d$;

With the above definitions, we say that periodic task i is *after* periodic task i' if and only if periodic task i' is before periodic task i. Periodic task i is *contained by* periodic task i' if and only if periodic task i' contains periodic task i;

A task in T_p is a *top* task if and only if it does not contain any other task in T_p.

The set of sporadic tasks in a system is represented by T_s. A *sporadic task* is denoted by a tuple $(i, (c, p, d))$, where i identifies the task, and (c, p, d) defines the attributes of its jobs, as follows. The jth job of sporadic task i is identified as job (i, j), $j \geqslant 0$. The arrival times of jobs of a sporadic task are not known *a priori*, and are determined at run time by an *arrival function* A that maps each job of a sporadic task to its arrival time for the particular run:

$A :: T_s \times \mathbf{N} \rightarrow \mathbf{R}$, where $\mathbf{N}$ is the set of natural numbers and $\mathbf{R}$ is the set of real numbers.

$A(i, j) = t$ if the job (i, j) arrives at time t;

$A(i, j) = \perp$ if the job (i, j) never arrivals.

The attributes in the definition of a sporadic task, c, p and d, are non-negative real numbers:

- c is defined the same as in a periodic task;
- There is a *minimum* interval p between the arrival times of consecutive jobs; in terms of the function A, if $A(i, (j + 1))$ is defined, then $A(i, (j + 1)) - A(i, j) \geqslant p$;
- d is the *deadline* of the task, which is the latest time by which the task must be completed after its arrival.

The arrival time of a sporadic job is unknown until it arrives on-line, i.e., for a sporadic task $(i, (c, p, d))$, job (i, j) is defined as $(c, A(i, j), A(i, j) + d)$. The on-line scheduler has no knowledge of the job (i, j) before time=$A(i, j)$.

A task set T is the union of the periodic and sporadic tasks: $T = T_p \cup T_s$.

3.2 Schedule

The function S maps each job to a set of time intervals:

$S :: T \times \mathbf{N} \to \{(\mathbf{R}, \mathbf{R})\}$, where T is the task set, $\mathbf{N}$ and $\mathbf{R}$ represent the set of natural numbers and the set of real numbers respectively.

$S(i, j) = \{(b_{i,j,k}, e_{i,j,k}) | 0 \leqslant k < h\}$, where k and h are natural numbers.

We say that the function S is a *schedule* of T if and only if all of the following constraints are satisfied:

- **Constraint 1**: for every job (i, j), every time interval assigned to it in the schedule must have positive length, i.e., for all $(b, e) \in S(i, j)$, $e - b \geqslant 0$.
- **Constraint 2**: the resource is allocated to at most one job at a time, i.e., time intervals do not overlap: For every $(b_{i,j,k}, e_{i,j,k}) \in S(i, j)$ and for every $(b_{i',j',k'}, e_{i',j',k'}) \in S(i', j')$, one of the following cases must be true:
  - $e_{i,j,k} \leqslant b_{i',j',k'}$; or
  - $e_{i',j',k'} \leqslant b_{i,j,k}$; or
  - $i = i'$, $j = j'$ and $k = k'$.
- **Constraint 3**: a job must be scheduled between its ready time and deadline: for every $(b, e) \in S(i, j)$,

$$(i, j).r \leqslant b < e \leqslant (i, j).d$$

- **Constraint 4**: for every job (i, j), the total length of all time intervals in $S(i, j)$ is sufficient for executing the job, i.e.,

$$\sum_{(b,e) \in S(i,j)} (e - b) \geqslant (i, j).c$$

Given a time t, if there exists a time interval (b, e) in $S(i, j)$ such that $b \leqslant t < e$, then we say that job (i, j) is *scheduled at* time t, and task i is *scheduled at* time t. Suppose S_A is a schedule for sporadic task T_s for the arrival function A, and schedule S_p is a schedule for the periodic task set T_p. Then $S = S_A \cup S_p$ is a schedule for T, such that for any sporadic job (i, j), $S(i, j) = S_A(i, j)$, and for any periodic job (i, j), $S(i, j) = S_p(i, j)$. $S_A \cup S_p$ is meaningful only when they do not conflict, i.e., for any sporadic job (i, j) and for any periodic job (i', j'), for any $(b, e) \in S_A(i, j)$ and for any $(b', e') \in S_p(i', j')$, either $e \leqslant b'$ or $e' \leqslant b$.

If a job can be scheduled into multiple seperate time intervals, the job is *preemptive*; otherwise, it is *non-preemptive*. A task is either preemptive, in which case all jobs of it are preemptive, or non-preemptive, in which case all jobs of it are non-preemptive. Given a non-preemptive task i, schedule S is valid if and only if the following extra constraint besides the constraints given in the definition of schedule are all satisfied: for all j, $S(i, j) = I$, and there is one and only one element in I. Non-preemptive scheduling is NP-hard in the strong sense. In this paper, we assume that periodic tasks are preemptive.

3.3 The On-line Dynamic Scheduler and S_A

The sporadic tasks are scheduled by an on-line dynamic scheduler. Given an arrival function A of T_s, S_A is the schedule produced by the dynamic scheduler. For clarity of explanation, we shall represent the on-line dynamic scheduler by a state transition diagram with two states: *idle* and *busy*. The initial state is *idle*. When the dynamic scheduler is in *idle* and a sporadic job arrives, the dynamic scheduler enters *busy* state. When the dynamic scheduler is *busy* and the accumulated time (from time=0) that the dynamic scheduler is in *busy* state equals the total execution times of all sporadic jobs that have arrived up to that point, the dynamic scheduler enters *idle*. We also make the following assumptions about the on-line dynamic scheduler and schedule S_A.

- The dynamic scheduler is *busy* at time t if and only if there exists $(b, e) \in S_A(i, j)$ such that $t \in (b, e)$.
- The dynamic scheduler does not idle unnecessarily: Suppose S_A is a schedule of T_s. If $(b, e) \in S_A(i, j)$, then for any $t \in (A(i, j), e)$, there exists $(b', e') \in S_A(i', j')$ such that $t \in (b', e')$ for some i', j'. A *non-idling schedule* is a schedule produced by a scheduler that does not idle unnecessarily.
- For any sporadic job (i, j), $\sum_{(x,y) \in S_A(i,j)} (y - x) = i.c$

Many dynamic scheduling algorithms, such as the fixed-priority and earliest-deadline-first policies satisfy these assumptions.

3.4 Pre-schedule

A pre-schedule is similar to a static schedule, but more flexible. A static schedule defines the exact time intervals allocated to every task. While a pre-schedule also defines an order of time "blocks" to be scheduled, these time blocks are preemptible and can be postponed to a certain extent in order to accomodate the sporadic tasks that are scheduled at a higher priority.

A pre-schedule is defined as shown in Figure 6. An *executive* is defined by a tuple of (i, c), where $i \in T_p$, and c is the execution time for this executive. A *list of executives* L is a total order of executives. A list of executives L is a *pre-schedule* of T if and only if for every non-idling schedule S_A of T_s, there exists a schedule S_p of T_p and a function M such that all of the following constraints are satisfied:

- S_p and S_A do not conflict. For any sporadic job (i,j) and for any periodic job (i',j'), suppose that $(b,e) \in S_A(i,j)$ and $(b',e') \in S_p(i',j')$. Then either $e \leqslant b'$ or $e' \leqslant b$.

- If executive k is before executive k' in L, $M(k,j) = (b,e)$ and $M(k',j) = (b',e')$, then $e \leqslant b'$.

- Only one periodic job $(k.i,j)$ is scheduled between $M(k.i,j) = (b,e)$: if $(x,y) \in S_p(i',j')$ and $b \leqslant x \leqslant y \leqslant e$, then $i' = k.i$ and $j' = j$.

- If $(x,y) \in S_p(i,j)$, then there exists executive k, such that $k.i = i$, $M(k,j) = (b,e)$, and $b \leqslant x \leqslant y \leqslant e$.

- For every executive k, the accumulated time allocated to task $k.i$ in time interval $M(k,j)$ is equal to $k.c$:

$$k.c = \sum_{(x,y)\in S(k.i,j) \text{ and } b\leqslant x<y\leqslant e} max(0,(min(e,y)-max(b,x))) = \sum_{(x,y)\in S(k.i,j) \text{ and } b\leqslant x<y\leqslant e} (y-x)$$

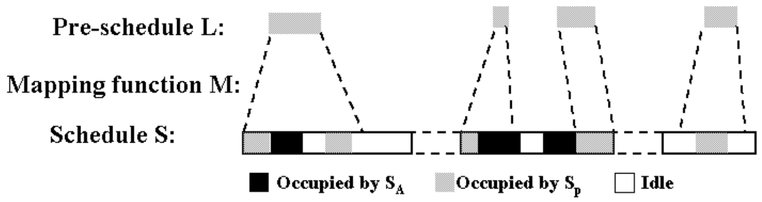

Figure 6: Definition of Pre-schedule

We say that the schedule S_p satisfying the conditions above is *compliant to* S_A and the pre-schedule L; and $S = S_p \cup S_A$ is compliant to L. Given a task set T, it is *pre-schedulable* if and only if there exists a pre-schedule L of T.

3.5 On-line Static Scheduler and Schedule S_p

For clarity of presentation, we represent the on-line static scheduler by a state transition diagram which outputs the schedule S_p. It is shown in Figure 7.

The static scheduler refers to a number of global variables during its execution. In the following, the current time is denoted by the variable t. initialized to 0. The current time frame is the jth time frame, where $j = \lfloor \frac{t}{F} \rfloor$. the list of executives is denoted by L. There is a pointer to the *current* executive L, which is initialized to 0, the first executive in L. The variables b and $start$ are used to denote the start times of the current time intervals of S_p and M, respectively.

The state transition diagram has five states: *idle*, *busy*, *transient*, *preempted*, and *failed*. State *idle* is the initial state. When a static scheduler is in *busy*, the resource is allocated to the job of the

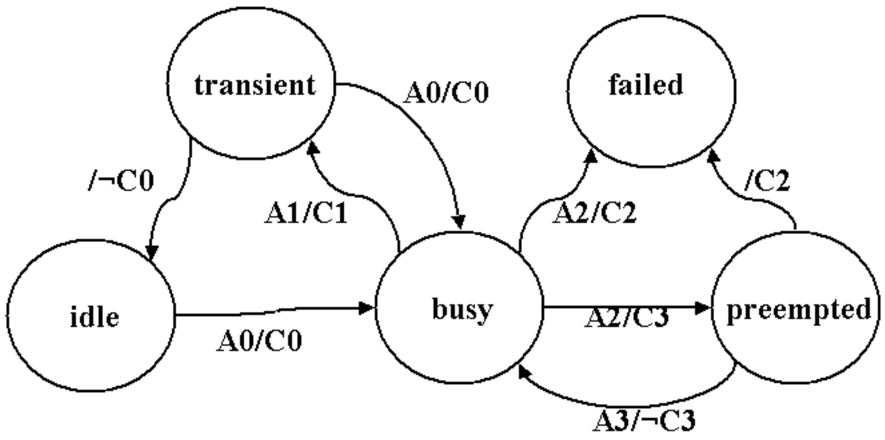

Figure 7: On-line Static Scheduler

current executive, and we also say that current time is allocated to the current executive. When a static scheduler is in any other states, the resource is not allocated to any periodic job. When a static schduler is in the $failed$ state, the on-line scheduler fails. When the dynamic scheduler is $busy$, the static scheduler is $preempted$. When the static scheduler finishes scheduling the current executive, it moves to the $transient$ state and decides if it will enter the $idle$ state to wait for the ready time of the job of the next executive or re-enters $busy$ to run the next executive. Notice that the static scheduler never stays in $transient$ state for any time interval with non-zero length.

The conditions and actions for the state transition diagram are defined as follows.

- **C0** : the job of current executive is ready: $(current.i, j).r \leqslant t$.
- **C1** : The accumulated time allocated to the current executive is equal to the execution time of the executive:
$$t - b + \sum_{(x,y) \in S_p(current.i,j) \text{ and } start \leqslant x} (y - x) = current.c$$
- **C2** : the deadline of the job of current executive is missed: $(current.i, j).d \leqslant t$.
- **C3** : The on-line dynamic scheduler is $busy$.

When the static scheduler is in $busy$ state and two or three conditions become true at the same time, the static scheduler takes the transition and action triggered by the condition with the lowest index number. When the static scheduler is in $preempted$ state, if $C2$ becomes true and $C3$ becomes false at the same time, the static scheduler takes the transition toward state $failed$.

- **A0** : $start := t$; $b := t$.
- **A1** :
  - $S_p(current.i, j) := S_p(current.i, j) \cup \{(b, t)\}$;
  - $M(current, j) := (start, t)$;
  - If the *current* executive is the last executive in L, set *current* as the first executive in L; otherwise, move *current* to the next executive in L.
- **A2** : $S_p(current.i, j) := S_p(current.i, j) \cup \{(b, t)\}$.
- **A3** : $b := t$.

The following lemmas and theorems show that the on-line static pre-scheduler as defined above is optimal.

Lemma 1 *Suppose L a pre-schedule of T. For every non-idling schedule S_A of T_s, the on-line static scheduler never enters $failed$ state.*

Proof: First prove that the on-line static scheduler never enters *failed* state. Assume the opposite: the static scheduler enters *failed* state at time e which is in time frame $j = \lfloor \frac{e}{F} \rfloor$. Let b be the latest time when the static scheduler is *idle* before time e, or the start of time frame j, whichever is later.

Assume that executive k_b and k_e are the current executives in L at time b and e. Assume X_p the accumulated time scheduled between (b, e) by S_p, and X_A the accumulated time scheduled between (b, e) by S_A:

$$X_A = \sum_{(x,y) \in S_A(i,j)} max(0, min(y, e) - max(x, b))$$

$$X_p = \sum_{(x,y) \in S_p(i,j)} max(0, min(y, e) - max(x, b))$$

Claim 1:

$$X_p + X_A = e - b$$

This follows the assumptions on time b and e.

Claim 2:

$$X_p < \sum_{k \in [k_b, k_e]} k.c$$

This follows the definition of on-line static scheduler and assumptions on k_b, k_e, and e. Notice that because the execution time of executive k_e is not fully scheduled before e, operator $<$ instead of operator $\leqslant$ is used.

Claim 3: Job $(k_b.i, j)$ is ready at or after time b.
Otherwise, on-line static scheduler enters $busy$ state earlier, contradicts to the assumption on b and k_b.

Claim 4: The deadline of job $(k_e.i, j)$ is time e.
This is because on-line static scheduler enters $failed$ at time e.

Let X'_p represents the accumulated time scheduled to periodic tasks by S' in time interval (b, e). Because of Claim 3 and Claim 4, For an arbitrary schedule S' compliant to L and S_A, every executive $k \in [k_b, k_e]$ is scheduled between time interval (b, e). Then

$$X'_p \geqslant \sum_{k \in [k_b, k_e]} k.c$$

Because of Claim 1 and 2,

$$X'_p + X_A > e - b$$

which means the accumulated time scheduled to all tasks between (b, e) by S' is greater than the length of the time interval. Contradict to Constraint 2 in the definition of schedule. ■

Lemma 2 *If the static scheduler does not enter $failed$ state, $S = S_A \cup S_p$ satisfies Constraint 1, 2 and 3 in the definition of schedule.*

Proof: Time interval (b, e) is a *busy interval* if and only if for every time $t \in (b, e)$, on-line static scheduler is $busy$. A busy interval (b, e) is a maximal busy interval if and only if for every time interval (b', e') such that either $b' \leqslant b$ and $e < e'$, or $b' < b$ and $e \leqslant e'$, (b', e') is not a busy interval.

By the definition of on-line static scheduler, b is recorded when the on-line static scheduler enters $busy$ state by action A0, and the following maximal busy interval (b, t) is scheduled to the job of current executive. Therefore, the following two claims are true: Claim 1: For every maximal busy interval (b, e), there exists one and only one periodic job (i, j) such that $(b, e) \in S_p(i, j)$.
Claim 2: If $(b, e) \in S_p(i, j)$, then (b, e) must be a maximal busy interval.

We show that all constraints in the definnition of schedule are satisfied.

Because the length of busy interval is non-negative, Constraint 1 is satisfied because of Claim 2.

Because of Claim 1, time intervals scheduled to periodic jobs does not overlap. Also, when on-line dynamic scheduler is $busy$, on-line static scheduler is either *idle* or *preempted*. Then because of Claim 2, any time interval scheduled to a periodic job does not overlap with any time interval scheduled to a sporadic job. Because we assume that S_A is a valid schedule, then time intervals scheduled to sporadic jobs do not overlap. Therefore Constraint 2 is satisfied.

Because of condition C0 and C2 in the definition, when the on-line static scheduler is in *busy* state, the time must be between the ready time and the deadline of the job of the current executive. Then by Claim 2, Constraint 2 is satisfied. ■

Lemma 3 *Suppose L a pre-schedule of T and S_A a schedule of T_s. If the static scheduler does not enter failed state, $S = S_p \cup S_A$ is a schedule compliant to L.*

Proof: Because of condition C1 in the definition, The cursor to the "current" executive is moved to the next if and only if the accumulated time scheduled to the executive is equal to the execution time of the executive. Also, the definition of pre-schedule implies that the accumulated execution time of executives in a pre-schedule for a task is equal to the execution time of the task. Then Constraint 4 is satisfied. By Lemma 2, all constraints in the definition of schedule are satisfied. S is a schedule of T. The mapping function M is also defined by the static scheduler, L, S and M satisfy all constraints in the definition of a pre-schedule. Therefore, S is a schedule compliant to L. ■

Theorem 1 *If a list of executives L is a pre-schedule, then for any non-idling schedule S_A of T_s, the on-line static scheduler produces a schedule S_p of T_p that is compliant to L.*

Proof: Follows from Lemma 1, 3. ■

Theorem 2 *Given a list of executives L and any non-idling schedule S_A for T_s, if the on-line static scheduler produces a schedule S_p of T_p that is compliant to L and S_A, then L is a pre-schedule for T.*

Proof Sketch: Let schedule S_p and function M be produced by the on-line static scheduler. It is straightforward to check that L, S_p, S_A and M satisfy all the constraints required in the definition of the pre-schedule. ■

4 The Planner

A *plan* P is a list of special executives with undefined execution time. The undefined execution time is denoted by $\perp$. A *weight assignment* W to a plan P assigns a non-negative real number to the execution time of each executive in P, and we denote the resulting assignment by $W(P)$. Two desirable properties of a plan are:

- A plan should be *relevant*: Suppose task set T is pre-schedulable, A plan P is a *relevant* plan of T if and only if there exists a weight assignment W such that $W(P)$ is a pre-schedule of T.
- A plan should be as *instrumental* as possible: A plan P is more instrumental than another plan P' if P has fewer executives than P'.

This section describes the algorithm Planner and show that the plan produced by Planner is relevant and also quite instrumental.

4.1 The Algorithm

Planner produces a plan of T_p by two steps.

In the first step, Planner produces a sorted list of executives of the top tasks (recall that a top task is one that does not contain any other task) according to Constraints 1 and 2:

- **Constraint 1** For each top task i in T_p, there is one and only one executive $(x, (i, \perp))$ in P. This executive is called the executive of task i.
- **Constraint 2** Given top tasks i and i', executive $(x, (i, \perp))$ is before executive $(y, (i', \perp))$ if and only if and task i is before task i'.

In the second step, Planner inserts executives of non-top tasks between the top task executives according to Constraint 3 and sort them according to Constraint 4. To facilitate the description, we introduce the concept of a *gap*. For every pair of consecutive executives of top tasks $(b, (i_b, \perp))$ and $(e, (i_e, \perp))$ in P, all the executives between and excluding executives b and e are said to belong to the gap identified as gap (b, e).

- **Constraint 3** For any gap (b, e) in P and any task i in T_p, if task i contains either task $b.i$ or task $e.i$, then there exists one and only one executive $(x, (i, \perp))$ in gap (b, e); otherwise, no executive of task i exists in gap (b, e).
- **Constraint 4** If executives x and y are in the same gap (b, e), then executive x is before executive y if and only if one of the following conditions holds:
  - task $x.i$ is before task $y.i$;
  - task $x.i$ contains task $y.i$, and task $y.i$ does not contain task $b.i$;
  - task $x.i$ is contained by task $y.i$, and task $x.i$ contains task $b.i$.

4.2 Analysis

Theorem 3 *The plan produced by Planner is relevant.*

Proof: The proof consists of the following steps.

- Define an algorithm $ExSort$ to transform L_g to L_p;
- Prove that algorithm $ExSort$ terminates;
- Prove that L_p is a pre-schedule of T;
- Prove that there exists a weight assignment W such that $W(P) = L_p$.

■

Definition of $ExSort$:

$ExSort$ defines a sequence atomic transformations to transform L_g to L_p. A transformation is either a merging of adjacent executives, or a swapping of executives of two tasks when the executives of this two tasks are not in order. Steps and rules are established to guide the sequence of transformations from L_g to L_p.

$ExSort$ consists of the following steps:

- Step 0: Sort T_p, such that for every pair of task i and i' in T_p, the position of i in T_p is before the position of i' in T_p if and only if task i is either before or contained by task i'.
- Step 1:

(1) $k := 0$;
(2) **while** $k < |T_p|$
(3) consider the kth (start from 0) task in T_p as task x refered in Rule 2 and Rule 3:
(4) apply transformations defined by Rule 1, 2 and 3 until the predicts of Rule 1, 2, 3 are false;
(5) $k := k + 1$;

- Step 2:
For every gap (b, e) in L, apply the fransformations defined by Rule 1 and Rule 4 until the predicts of Rule 1 and Rule 4 are false.

The four rules are described below:

- **Rule 1**
Predict:
There exist a pair of executives e and e', such that the following are all true:
  - executive e is before executive e'; and

- executive e and e' are adjacent in L (in other words, there is no executive between e and e' in L); and
- $e.i = e'.i$.

Transformation:
$Merge(e, e')$

- **Rule 2**
  Predict:
  There exists a periodic task y in T_p, such that

  - task x is before task y; and
  - given the executives b the first executive of y and e the last executive of x in L, b is before e in L.

  Transformation:
  $Swap(x, y, b, e)$

- **Rule 3**
  Predict:
  There exist periodic tasks y and z in T_p, such that

  - task y contains task x;
  - task z is either task x, or is contained by task x;
  - given executives f the first executive of z and e last executive of x, there exists at least an executive of y between f and e, and let b the first executive of y between f and e.

  Transformation:
  $Swap(x, y, b, e)$

- **Rule 4**
  Predict:
  There exists a pair of executives x and y in the gap (b, e), such that

  - executive y is before executive x,
  - one of the following conditions is true:
    * task $x.i$ is before task $y.i$; or
    * task $x.i$ contains task $y.i$, and task $y.i$ does not contain task $b.i$; or
    * task $x.i$ is contained by task $y.i$, and task $x.i$ contains task $b.i$.

Transformation:

$Swap(x.i, y.i, b, e)$

The two transformations are defined as follows.

Function $Merge(x, y)$ merges executive x and executive y to executive z:

- define executive z as $(x.i, x.c + y.c)$;
- replace executive x with executive z in L, and delete y from L.

Function $Swap(x, y, b, e)$ is defined as follows:

X represents accumulated execution time of all executives of task x between $[b, e]$.

Step 1: Find the last executive m, such that

$$\sum_{k \text{ is between } [b,m) \text{ and } (k.i=x \text{ or } k.i=y)} k.c \leqslant X$$

Step 2: For each executive k between $[b, m)$, if $k.i = y$, then $k.i := x$; For each executive k between $(m, e]$, if $k.i = x$, then $k.i := y$.

Step 3:

$$X' := \sum_{k \text{ is between } [b,m) \text{ and } k.i=x} k.c$$

Step 4: If $X = X'$, $m.i = y$; otherwise split executive m into two executives m_x and m_y; m_x is before m_y, and for every executive z before m in L, z is before m_x in L'; for every executive z after m in L, z is after m_x in L'. Executive m_x is defined as $(x, X - X')$, and executive m'_x is defined as $(y, m.c - (X - X'))$. ■

Lemma 4 *Step 1 of algorithm ExSort terminates.*

Proof: We only need to prove that Rule 1, 2 and 3 can only be applied for a finite number of times in each iteration of the loop in Step 1.

Assume that the number of executives before the last executive of task x in L is n, then n is a finite integer at the beginning of the loop. Every time task x is swapped with other task in this loop, n decreases because x is always swapped toward the front of L. every time Rule 1 is applied, n remains the same or decreases. Therefore, the number of times that Rule 2 or Rule 3 are applied is finite. The number of executives in L decrements every time Rule 1 is applied; and the number increases at most by 1 every time Rule 2 or Rule 3 is applied. Therefore the number of times Rule 1 is applied is also finite. ■

Lemma 5 *Assume that $L_{g'}$ is the list of executives after Step 1 of ExSort, $L_{g'}$ is a pre-schedule.*

Proof: Given arbitrary transformation $Trans$ in Step 1, assume that the list of executives right before the transformation is L, and the list of executives right after the transformation is L'. It is sufficient to prove that if L is a pre-schedule, L' is also a pre-schedule.

Assume L is a pre-schedule, then given arbitrary schedule S_A of T_s, there exists a schedule S_p of T_p comply to L and S_A by mapping function M. We prove that then there exists a a schedule S'_p comply to L' and S_A by mapping function M'.

Case 1: Assume that the transformation is to merge two adjacent executives x and y in L to executive x' in L'. Let schedule $S'_p = S_p$. Mapping function M' is defined as follows: for every $j \in \mathbf{N}$,

- for every executive k earlier than x' in L', $M'(k,j) := M(k,j)$;
- assume that $M(x,j) = (b_x, e_x)$ and $M(y,j) = (b_y, e_y)$, then $M'(x',j) := (b_x, e_y)$;
- for every executive k after x' in L', $M'(k,j) := M(k+1,j)$.

Case 2: Assume that the transformation is $Swap(x,y,b,e)$. Assume that $M(b,j) = (b_b, e_b)$, $M(e,j) = (b_e, e_e)$, In the definition of function $Swap$, there is a "middle" executive m in L, which may be split into two executive m_x and m_y in L'. Assume $M(m,j) = (b_m, e_m)$. Now we distinguish two sub-cases:

Case 2.1: m is not split in swap. Then S'_p is defined as follows:

- step 1: $S'_p := S$
- step 2: for all $(b,e) \in S'_p(y,j)$ and $b_b \leqslant b < e \leqslant b_m$,
  - $S'_p(y,j) := S'_p(y,j) \setminus \{(b,e)\}$;
  - $S'_p(x,j) := S'_p(x,j) \cup \{(b,e)\}$;
- step 3: for all $(b,e) \in S'_p(x,j)$ and $b_m \leqslant b < e \leqslant e_e$,
  - $S'_p(x,j) := S'_p(x,j) \setminus \{(b,e)\}$;
  - $S'_p(y,j) := S'_p(y,j) \cup \{(b,e)\}$.

Definition of function M':
$M' := M$.

Case 2.2: Assume that executive m in L is split into two executive m_x and m_y in L'. Define time t_m, such that the accumulated time scheduled to task x or y between (b_m, t_m) by S_p is $m_x.c$. In other words,

$$\sum_{(b,e)\in(S_p(x,j)\cup S_p(y,j))} (min(t_m, e) - max(b_m, b)) = m_x.c$$

Then S'_p is defined as follows:

- step 1: $S'_p := S$
- step 2: for all $(b, e) \in S'_p(y, j)$ and $b_b \leqslant b < e \leqslant t_m$,
  - $S'_p(y, j) := S'_p(y, j) \setminus \{(b, e)\}$;
  - $S'_p(x, j) := S'_p(x, j) \cup \{(b, e)\}$;
- step 3: for all $(b, e) \in S'_p(x, j)$ and $t_m \leqslant b < e \leqslant e_e$,
  - $S'_p(x, j) := S'_p(x, j) \setminus \{(b, e)\}$;
  - $S'_p(y, j) := S'_p(y, j) \cup \{(b, e)\}$;
- step 4: if there exists $(b, e) \in (S_p(x, j) \cup S_p(y, j))$ and $b \leqslant t_m \leqslant e$;
  - $S'_p(x, j) := S'_p(x, j) \setminus \{(b, e)\}$;
  - $S'_p(y, j) := S'_p(y, j) \setminus \{(b, e)\}$;
  - $S'_p(x, j) := S'_p(x, j) \cup \{(b, t_m)\}$;
  - $S'_p(y, j) := S'_p(y, j) \cup \{(t_m, e)\}$;

Definition of M':

- For every executive k before m, $M'(k, j) = M(k, j)$;
- $M'(m, j) = (M(m, j).b, t_m)$;
- $M'(m + 1, j) = (t_m, M(m, j).e)$;
- For every executive k after $m + 1$, $M'(k, j) = M(k - 1, j)$.

It can be verified that S'_p and M' as defined in this proof, togather with L' as defined by transformation, and S_A, satisfy all constraints in the definition of pre-schedule. ■

Lemma 6 *Given an arbitrary pair of tasks u and v in T_p, neither Rule 2 nor Rule 3 defined in ExSort can apply to $L_{g'}$ on tasks u and v.*

Proof: Assume that task u is the kth task in T_p after Step 0 of $ExSort$. Assume L_k is the pre-schedule by the end of the kth iteration in Step 1 of algorithm $ExSort$. Assume task v an arbitrary task after or containing task u, Neither Rule 2 nor Rule 3 can apply to L_k for task u and v, otherwise the kth iteration continues.

Assume that transformation $Trans$ is the first transformation after the end the kth iteration, such that the following items are all true:

- L is the pre-schedule right before the transformation, and there exists no task v, such that either Rule 2 or Rule 3 may apply to L on task u and v;
- L' is the pre-schedule right after the transformation, and there exists a task v, such that either Rule 2 or Rule 3 may apply to L' on task u and v;

Observation 1: Merging does not change the relative positions of executives from different tasks, therefore $Trans$ must be a swap.

Observation 2: Because $Trans$ happens after the end of the kth iteration, task u can not involve in any swap after iteration k. Therefore, $Trans$ must be a swap between task v and some task w.

Observation 3: Either task w is after task u or task w contains task u.

Assume that Rule 2 applies to L' by $Trans$ to tasks u and v, some executives of task v is swapped forward to the front of executive e, the last executive of task u. Then task w has some executives before executive e in L. If task w is after task u, then Rule 2 applies to L on tasks u and w, contradiction to the assumption made on L. Therefore task w contains task u. By $Trans$, task w is swapped toward the end of the pre-schedule and task v is swapped toward the beginning, therefore task w is not before task v, therefore task w contains task v. Therefore transformation $Trans$ is triggered by Rule 3 on tasks v and w. Then there must exist a executive f of task z, which is a task contained by task v, and there exist executives of task w between f and executives of task u. Task z is not after task u, because otherwise Rule 2 can apply on L on task v and z, which is a contradiction. Therefore, Task z is contained by task u. Then Rule 3 applies to L on task u and w, contradiction.

If Rule 3 applies to L' to tasks u and v, then some executives of task v is swapped with some executives of task w between some executive (f, e), where f is the first executive of task u or a task contained by task u, and e is the last executive of task u. If task w is after task u, Rule 2 applies to L on task u and w, contradiction. If task w contains task u, Rule 3 applies to L on task u and w, contradiction.
■

Lemma 7 *$L_{g'}$ has the following properties.*

- *Property 1: for each top task i, there is one and only one executive x in $L_{g'}$, such that $x.i = i$.*
- *Property 2: For every pair of executives x and y, if task $x.i$ and task $y.i$ are both top tasks and task $x.i$ is before task $y.i$, then executive x is before executive y in $L_{g'}$ $(x < y)$.*

Proof: Property 1: Assume the opposite: there are multiple executives of top task i in $L_{g'}$, assume that the first and the last executives of task i are b and e. Case 1: If there exists no executives of other tasks between b and e, then Rule 1 should be applied to executives between b and e to merge them into one executive. Then $ExSort$ won't stop before applying this merge before it terminates. Contradiction. Case 2: If there exists an executive m between b and e and $m.i \neq i$, then by definition of top task, task $m.i$ is not contained by task i. Then if task $m.i$ is before task i, Rule 2 applies to $L_{g'}$ on tasks $m.i$ and i, contradicts to Lemma 6; if task $m.i$ is after task i, Rule 2 applies to $L_{g'}$ on tasks i and $m.i$, contradicts to Lemma 6; if task i is contained by task $m.i$, Rule 3 applies to $L_{g'}$ on tasks i and $m.i$, contradicts to Lemma 6.

Property 2: otherwise, Rule 2 applies to $L_{g'}$ on tasks $x.i$ and $y.i$, contradicts to Lemma 6. ■

Lemma 8 *Step 2 in algorithm $ExSort$ terminates.*

Proof:
There exists no circle on the order of executives in a gap. Therefore this step terminates. ■

Lemma 9

L_p is a pre-schdule.

Proof: The transformations in Step 2 of algorithm $ExSort$ are merge and swap, which are the same as in Step 1 of algorithm $ExSort$. In the proof of Lemma 5, we have shown that each merge or swap transforms a pre-schedule to another pre-schedule. Therefore, as long as the input of Step 2 is a pre-schedule, the output of Step 2 is a pre-schedule. Togather with Lemma 5, L_p is a pre-schedule. ■

Lemma 10 *Pre-schedule L_p still holds Property 1 and 2 of $L_{g'}$, and plus the following properties:*

Property 3: For every gap (b, e) in $L_{g'}$, and every non-top task i in T_p, if task i contains either task $b.i$ or task $e.i$, then there exists at most one executive x in gap (b, e), or $b < x < e$, such that $x.i = i$; otherwise, there does not exist such an executive in this gap.

Property 4: if executives x and y are in the same gap (b, e), then executive x is before executive y (or $x < y$) iff one of the following conditions are true:

- *task $x.i$ is before task $y.i$;*

- *task $x.i$ contains task $y.i$, and task $y.i$ does not contain task $b.i$;*
- *task $x.i$ is contained by task $y.i$, and task $x.i$ contains task $b.i$.*

Proof: Step 2 limits the transformations within the scope of gaps, therefore executives for all top tasks, both their orders and their weights remain un-changed in this step. Therefore Property 1 and 2 of $L_{g'}$ are also the first two properties of L_p.

Property 3: Assume the opposite: there are multiple executives of task i in L_p between gap (b, e), the first and the last executives of task i in gap (b, e) are b' and e'. Case 1: If there exists no executives of other tasks between b' and e', then Rule 1 should be applied to executives between b' and e' to merge them into one executive. Contradiction. Case 2: there exists an executive m between b' and e' and $m.i \neq i$. If task $m.i$ is before task i, executive b' and m are not in order and Rule 4 applies, contradiction; If task $m.i$ is after task i, executive m and e' are not in order and Rule 4 applies, contradiction; if task i is contained by task $m.i$ and task i contains $b.i$, m and e' are not in order, contradiction; if task i is contained by task $m.i$ and task i does not contain $b.i$, b' and m are not in order, contradiction; if task i containes task $m.i$ and task $m.i$ contains $b.i$, b' and m are not in order, contradiction; if task i contains task $m.i$ and task $m.i$ does not contain $b.i$, m and e' are not in order, contradiction. All combinations are covered. Property 3 must be true when Step 2 terminates.

Property 4 directly follows Rule 4. ■

Lemma 11 *There exists a weight assignment $W(P) = L_p$.*

Proof: Weight assignment W is defined as follows:

- For every top task i, there is one and only one executive e_i in P corresponding to it, and there is one and only one executive e'_i in L_p corresponding to it. $W(e_i) = e'_i.c$;
- Given any non-top task i and any pair of adjacent top tasks b and e (b is before e), assume that executives k_b and k_e are corresponding to task b and e in P, there is one and only one executive k_i of task i in P in gap (k_b, k_e), if and only if task i contains either task b or task e or both. Assume that executives k'_b and k'_e are corresponding to task b and e in L_p, If task i contains either task b or task e or both, there might exists at most one executive k'_i in L_p in gap (k'_b, k'_e). If so, $W(k_i) = k'_i.c$; otherwise, $W(k_i) = 0$, or equivalently, delete k_i from P.

So W assigns an execution time to every executives in P. According to properties of L_p described in Lemma 10 and constraints on P set in algorithm $Planner$, $W(P) = L_p$. ■

Theorem 4 *The number of executives in a plan produced by Planner is upper bounded by $O(n_p^2)$, where n_p is the number of tasks in T_p.*

Proof: The number of gaps of the plan is bounded by $O(n_p)$, and for each periodic task, at most one executive of it exists in each gap. Therefore, the number of executives for each task is upper bounded by $O(n_p)$. ■

Theorem 5 *The computational complexity of Planner is bounded by $O(n_p^2 log(n_p))$, where n_p is the number of periodic tasks.*

Proof: The computational complexity for sorting a list of length N is $O(Nlog(N))$. The length of a plan is bounded by $(O(n_p^2))$, so the complexity of sorting the executives in a plan is bounded by $O((n_p^2)log(n_p^2)) = O(n_p^2 log(n_p))$. ■

5 The Weight Assigner

Given a task set T and a plan P generated by Planner, the Weight Assigner finds a weight assignment $W(P)$ which is a pre-schedule of T. We observe below that a list of executives is a pre-schedule if and only if a finite number of linear constraints can be satisfied. Therefore the weight assignment problem can be solved as a linear programming problem off-line.

5.1 Maximal Sporadic Task Occupation

The function MSO is defined to denote the maximal accumulated time that can be allocated to sporadic tasks by the on-line dynamic scheduler during any interval of a given length. Given T_s and length of time l, function $MSO(l)$ is computed by the following Saturated Test algorithm.

- We define a saturated arrival function A_s as follows. The first job of every task in T_s arrives at time 0 and all subsequent jobs arrive at the maximum rate, after the minimum interval from the previous job arrival. Formally, for all i in T_s and all j in $\mathbf{N}$, $A_s(i,j) = j \cdot (i.p)$.
- Suppose schedule S_{A_s} is an arbitrary non-idling schedule for T_s under A_s (recall that a non-idling schedule is one in which the resource never idles as long as there is an unfinished job). The function $MSO(l)$ returns the accumulated length of all the allocated time in S_{A_s} between $(0, l)$:

$$MSO(l) = \sum_{i \in T_s \text{ and } j \in \mathbf{N}} \left(\sum_{(x,y) \in S_{A_s}(i,j)} max(0, min(l,y) - x) \right)$$

The function MSO has the following properties.

- **Property 0** $MSO(l) \leqslant l$.
- **Property 1** Given any time interval (b, e), there exists an arrival function A such that the accumulated amount of time allocated between (b, e) by S_A is $MSO(e - b)$.

  Proof: Define the arrival function A as follows. The first job of every sporadic task arrives at time b, and the following jobs arrive at the minimum interval afterwards. The accumulated amount of time scheduled to sporadic tasks between (b, e) by S_A is the same as that value in Algorithm $Saturated\_Test$ between $(0, l)$, which is equal to $MSO(l)$. ■

- **Property 2** Given any non-idling schedule S_A of T_s, $MSO(l)$ is the maximal amount of time that can be allocated in S_A to sporadic tasks in any time interval of length of l.

  Proof:

  For convenience of proof, we define a function SO as follows: Given an arrival function A of T_s, and an interval of time (b, e), function SO is the accumulated amount of time scheduled to the sporadic tasks by an arbitrary work-conserving schedule S_A during (b, e).

  $$SO(A, (b, e)) = \sum_{i \in T_s \text{ and } j \in \mathbf{N}} (\sum_{(x,y) \in S_A(i,j)} max(0, min(e, y) - max(b, x)))$$

  To prove Property 2 of function MSO, just need to prove the following: given an arbitrary arrival function A of T_s and a length of time l, for any a time interval (b, e) such that $e - b = l$, $SO(A, (b, e)) \leqslant MSO(l)$.

  Case 0: $MSO(l) = l$. The property is obviously true.

  Case 1: $MSO(l) < l$, and all sporadic jobs scheduled between (b, e) arrives at or after time b in A.

  Assume that (x, y) is the latest idle time interval between $(0, l)$ by S_{A_s}, where A_s is the arrival function in algorithm Saturated Test.

  Assume that the accumulated execution time of all sporadic jobs arrived between $(0, y)$ by A_s is E_s, and the accumulated time of all sporadic jobs arrived between $(b, b + y)$ in A is E. Because of the definition of A_s, $E \leqslant E_s$.

  Assume that the accumulated time scheduled to sporadic jobs by S_{A_s} between $(0, y)$ is O_s. Because S_{A_s} is work-conserving and because S_{A_s} is idle between (x, y), all arrived sporadic jobs must be completely scheduled by time x. Then $O_s = E_s$.

  The accumulated time scheduled to sporadic jobs between $(b, b+y)$ by S_A is $SO(A, (b, b+y))$. Because of the case assumption,

  $$SO(A, (b, b + y)) \leqslant E \leqslant E_s = O_s$$

Because of the assumption made on y, S_{A_s} is never idle between (y, l). Therefore, $MSO(l) = O_s + (l - y)$. Then,

$$(SO(A, (b, e)) = SO(A, (b, b + y)) + SO(A, (b + y, e)) \leqslant O_s + (l - y) = MSO(l)$$

Case 2: $MSO(l) < l$, and there exist sporadic job (i, j), such that $A(i, j) < b$ and there exists time t between (b, e), such that t is scheduled to job (i, j) by S_A.

Schedule S_A is work-conserving, so S_A is never idle between $(A(i, j), t)$, then

$$SO(A, (b, e)) \leqslant SO(A, (A(i, j), A(i, j) + l))$$

Then we only need to prove $SO(A, (A(i, j), A(i, j) + l)) \leqslant MSO(l)$. Repeat this transformation until Case 1 is true for an interval with length l. ■

- **Property 3** For all non-negative real numbers x and y, $MSO(x + y) \leqslant MSO(x) + MSO(y)$.

  Proof: According to the definitions of function MSO and function SO,

$$MSO(x + y) = SO(A_s, (0, x)) + SO(A_s, (x, x + y))$$

  According to Property 2 of function MSO,

$$SO(A_s, (0, x)) \leqslant MSO(x)$$

$$SO(A_s, (x, x + y)) \leqslant MSO(y)$$

  Therefore,

$$MSO(x + y) \leqslant MSO(y) + MSO(y)$$

  ■

5.2 Minimal Periodic Task Occupation

Given a list of executives L, the minimal periodic task occupation represents the accumulated time that must be allocated to periodic tasks in a time interval in any schedule compliant to L. The minimal periodic task occupation is quantitatively defined by the function PO as follows. Given a time interval (b, e), let $b' = b \bmod F$ and $e' = e \bmod F$ and let f be the first executive in L satisfying the condition $b' \leqslant (f.i).r$; and let executive l be the last executive in L satisfying the condition $(l.i).r + (l.i).d \leqslant e'$. The Function $PO(b, e)$ is defined by two cases:

- Case 1: if b and e are in the same frame, i.e., $\lfloor \frac{b}{F} \rfloor = \lfloor \frac{e}{F} \rfloor$, then $PO(b, e)$ is the summation of the execution times of all the executives between and including executive f and executive l in L.

$$PO(b, e) = \sum_{k \in [f, l]} k.c$$

- Case 2: otherwise, $PO(b, e) = A + B + C$, where A, B and C are defined as follows:
  - A is the summation of the execution times of all the executives including and after executive f in L:

$$A = \sum_{k \in [f, |L|)} k.c$$

  - B is the summation of execution times of all the executives in L multiplied by the number of whole frames between (b, e):

$$B = (\lfloor \frac{e}{F} \rfloor - \lceil \frac{b}{F} \rceil) \cdot \sum_{k \in [0, |L|)} k.c$$

  - C is the summation of the execution times of all executives before and including executive l in L:

$$C = \sum_{k \in [0, l]} k.c$$

■

The function PO has the following properties.

- **Property 1** For any $j \in \mathbf{N}$,

$$PO(b + j \cdot F, e + j \cdot F) = PO(b, e)$$

This property directly follows the definition of function PO. ■

- **Property 2** Suppose L is a list of executives and S is a schedule compliant to L. Then the accumulated execution time allocated to periodic tasks between any time interval (b, e) in S must be greater than or equal to $PO(b, e)$:

$$PO(b, e) \leqslant \sum_{i \in T_p \text{ and } j \in \mathbf{N}} (\sum_{(x,y) \in S(i,j)} max(0, (min(y, e) - max(x, b))))$$

Proof:

Case 1: time b and time e are in the same frame($j \cdot F \leqslant b < e \leqslant (j + 1) \cdot F$).

Assume executive f and l are defined the same as in the definition of function PO. Assume that (b_f, e_f) is the first time interval scheduled in the jth frame to executive f, and (b_l, e_l) is the last time interval scheduled in the jth frame to executive l. By definition of pre-schedule and function PO, $b \leqslant b_f$, $e_l \leqslant e$, and all executives between $[f, l]$ are scheduled between $[b_f, e_l]$, then between $[b, e]$. Then Property 2 is true.

Case 2: if the condition in Case 1 does not hold, then function PO is defined as follows:

$$PO(b, e) = A + B + C$$

Because the arrival time and deadline of every periodic job is in the same frame,

$$\sum_{i \in T_p} (\sum_{(x,y) \in S(i,j)} max(0, (min(y,e) - max(x,b)))) = A' + B' + C'$$

where

$$A' = \sum_{i \in T_p} (\sum_{(x,y) \in S(i,j)} max(0, (min(y, \lceil \frac{b}{H} \rceil) - max(x,b))))$$

$$B' = (\lfloor \frac{e}{H} \rfloor - \lceil \frac{b}{H} \rceil) \cdot \sum_{i \in T_p} (\sum_{(x,y) \in S(i,j)} max(0, (min(y,H) - max(x,0))))$$

$$C' = \sum_{i \in T_p} (\sum_{(x,y) \in S(i,j)} max(0, (min(y,e) - max(x, \lfloor \frac{e}{H} \rfloor))))$$

According to Case 1 in this proof, $A \leqslant A'$, $B \leqslant B'$, and $C \leqslant C'$. Therefore, Property 2 is true. ■

5.3 Constraints on the Pre-Schedule

The constraints on a pre-schedule can be categorized into two sets: *minimal constraints* and *maximal constraints.*

The minimal constraints refer to the following constraints. Given L a pre-schedule of T and task x in T_p, the accumulated execution time of all the executives in L for task x must be greater than or equal to the execution time of task x. In other words, for any $i \in T_p$ and all executive $k \in L$,

$$\sum_{k.i=x} k.c \geqslant x.c$$

The maximal constraints refer to the following constraints. For every time interval (b,e),

$$MSO(e-b) + PO(b,e) \leqslant e - b$$

Theorem 6 *Given a task set T and a list of executives L, L is a pre-schedule of T if and only if L satisfies all minimal and maximal constraints.*

Proof: First prove that if L is a pre-schedule, then all minimal and maximal constraints are satisfied.

If L is a pre-schedule, then for any S_A, there exists schedule S_p of T_p comply to L and S_A. Given arbitrary periodic job (i,j), assume X represents the accumulated time scheduled to periodic job (i,j) by S_p, Y represents the accumulated execution time of all executives of task i in L. According to the definition of pre-schedule, $X = Y$. According to Constraint 4 in the definition of schedule, $X \geqslant i.c$. Therefore, $Y \geqslant i.c$, which means the minimal constraint on i is satisfied.

Given a time interval (b, e), according to Property 2 of function MSO and Property 2 of function PO, for every interval (b, e), there exists a work-conserving schedule S_A such that

$$MSO(e-b) \leqslant \sum_{i \in T_s \text{ and } j \in \mathbf{N}} \Big(\sum_{(x,y) \in S(i,j)} max(0, min(y,e) - max(x,b)) \Big)$$

and

$$PO(b,e) \leqslant \sum_{i \in T_p \text{ and } j \in \mathbf{N}} \Big(\sum_{(x,y) \in S(i,j)} max(0, (min(y,e) - max(x,b))) \Big)$$

Therefore,

$$MSO(e-b) + PO(b,e) \leqslant \sum_{i \in T \text{ and } j \in \mathbf{N}} \Big(\sum_{(x,y) \in S(i,j)} max(0, min(y,e) - max(x,b)) \Big)$$

Because Constraint 2 of a schedule,

$$\sum_{i \in T \text{ and } j \in \mathbf{N}} \Big(\sum_{(x,y) \in S(i,j)} max(0, min(y,e) - max(x,b)) \Big) \leqslant e - b$$

Therefore,

$$MSO(e-b) + PO(b,e) \leqslant e - b$$

which is the maximal constraint on time interval (b, e).

Second, assume that all minimal and maximal constraints are satisfied by L, prove that L is a pre-schedule. According to Theorem 2, only need to prove that the on-line static scheduler produces a schedule S_p comply to L and S_A.

Claim 1: on-line static scheduler does not fail. Assume that on-line static scheduler enters $failed$ state at time e and time b is the latest time that the on-line static scheduler is *idle*. Then at any time $t \in (b, e)$, resource is scheduled to either a sporadic or a periodic task. Assume that k_b and k_e are the current executives at time b and e respectively. Then because static scheduler is *idle* at time b, the $(k_b.i, \lfloor \frac{b}{F} \rfloor).r \geqslant b$. Because on-line static scheduler fails at time e, $(k_e.i, \lfloor \frac{e}{F} \rfloor).d \leqslant e$. Then by definition of function PO, $\sum_{k \in [k_b, k_e]} k.c \leqslant PO(b, e)$. Then because maximal constraint is satisfied on time interval (b, e), then $(\sum_{k \in [k_b, k_e]} k.c) + SMO(e-b) \leqslant e-b$. Assume that the accumulated time scheduled to periodic tasks and sporadic tasks between time (b, e) is X_p and X_A respectively. Because the executive k_e is not fully scheduled (because on-line static scheduler is $failed$ at time e), $X_p < \sum_{k \in [k_b, k_e]} k.c$. By definition of function SMO, $X_A \leqslant SMO(e-b)$, therefore, $X_A + X_p < e - b$, which means the accumulated time scheduled to periodic and sporadic tasks between time (b, e) is less than the length of the time interval, which contradicts with the assumptions made on e and b.

Claim 2: Assume that on-line static scheduler produces S_p, then S_p is a schedule of T_p.

Because of Lemma 2 and Claim 1, Constraint 1, 2, and 3 in definition of schedule are satisfied.

Because of condition C1 in the definition of static scheduler, The cursor to the "current" executive is moved to the next if and only if the accumulated time scheduled to the executive in the current frame is equal to the execution time of the executive. Therefore the accumulated time scheduled to each periodic job (i, j) by S_p is equal to the accumulated execution time of all executives of task i. Because minimal constraint on every task is satisfied, the accumulated execution time of all executives of task i is greater than or equal to the execution time of task i, therefore the accumulated execution time scheduled to job (i, j) by S_p is no less than the execution time of task i. then Constraint 4 in definition of schedule is satisfied. All constraints in the definition of schedule are satisfied, then S is a schedule of T. ■

5.4 Critical Maximal Constraints for Pre-scheduler

The set of maximal constraints defined in the last sub-section consists of an infinite number of constraints. In this sub-section, we define a finite set of *critical* time intervals, and prove that as long as the maximal constraints are satisfied on these critical time intervals, all maximal constraints are satisfied.

Lemma 12 *If for any time x and time y satisfying $0 \leqslant x \leqslant y \leqslant F$, $MSO(y-x) + PO(x, y) \leqslant y - x$, then for any time interval (b, e), $MSO(e-b) + PO(b, e) \leqslant e - b$.*

Proof: Given arbitrary time interval (b, e), one of the following two cases must be true:

Case 1: Time b and e are in the same frame. In other words, $\lfloor \frac{b}{F} \rfloor = \lfloor \frac{e}{F} \rfloor$.

Assume that $b' = b \bmod F$, $e' = e \bmod F$. Because of the case assumption, $0 \leqslant b' < e' \leqslant F$. Because of the assumption in the lemma, $MSO(e' - b') + PO(b', e') \leqslant e' - b'$.

According to Property 1 of function PO, $PO(b, e) = PO(b', e')$. Because $e' - b' = e - b$, $MSO(e-b) = MSO(e' - b')$. Therefore, $MSO(e-b) + PO(b, e) = MSO(e' - b') + PO(b', e') \leqslant e' - b' = e - b$.

Case 2: Time e and time b are not in the same frame. In other words, $\lfloor \frac{b}{F} \rfloor < \lfloor \frac{e}{F} \rfloor$.

Define the following variables:

$b' = b \bmod F$,

$e' = e \bmod F$,

$n = \lfloor \frac{e}{F} \rfloor - \lceil \frac{b}{F} \rceil$.

According to the definitions,

$$e - b = (F - b') + n \cdot F + e'$$

Then according to Property 3 of function MSO,

$$MSO(e-b) \leqslant MSO(F-b') + n \cdot MSO(F) + MSO(e')$$

According to the definition of function PO,

$$PO(b,e) = PO(b',F) + n \cdot PO(0,F) + PO(0,e')$$

By the assumption of the lemma,

$MSO(F-b') + PO(b',F) \leqslant (F-b')$,

$MSO(F) + PO(0,F) \leqslant F$,

$MSO(e') + PO(0,e') \leqslant e'$.

Therefore,

$$MSO(e-b) + PO(b,e) \leqslant (F-b') + n \cdot F + e' = e-b$$

■

The time interval (b,e) is *critical* if and only if there exists a pair of periodic tasks i and i' in T_p, such that $i.r = b$ and $i'.r + i'.d = e$.

Lemma 13 *If for any critical time interval (x,y), $MSO(y-x) + PO(x,y) \leqslant y-x$, then for any b and e satisfying $0 \leqslant b \leqslant e \leqslant F$, $MSO(e-b) + PO(b,e) \leqslant e-b$.*

Proof: Suppose executive f is the first executive in L such that $b \leqslant (f.i).r$, and executive l is the last executive in L such that $(l.i).r + (l.i).d \leqslant e$. Suppose further that $b' = (f.i).r$ and $e' = (l.i).r + (l.i).d$. Then the following are all true: (i) $PO(b,e) = PO(b',e')$; and (ii) $b \leqslant b' \leqslant e' \leqslant e$; and (iii) (b',e') is a critical interval.

Then $MSO(e'-b') + PO(b',e') \leqslant e'-b'$,

By Property 0 and 3 of function MSO,

$$MSO(e-b) \leqslant MSO(b'-b) + MSO(e'-b') + MSO(e-e') \leqslant MSO(e'-b') + (e-b) - (e'-b')$$

Then we have

$$MSO(e-b) + PO(b,e) \leqslant (MSO(e'-b') + PO(b',e')) + ((e-b) - (e'-b')) \leqslant e-b$$

■

Theorem 7 *Given a task set T and a list of executives L, L is a pre-schedule of T if and only if L satisfies all minimal constraints and maximal constraints on all critical time intervals.*

Proof: This follows from Theorem 6, Lemma 12 and 13. ■

5.5 The Algorithm

Given plan $P = \{(i_0, \perp)..(i_{|L|-1}, \perp)\}$, suppose that a pre-schedule $L = W(P)$ is $\{(i_0, c_0)..(i_{|L|-1}, c_{|L|-1})\}$. Then the minimal constraint for a task x can be represented in the following form:

$$c_{x_0} + .. + c_{x_{k-1}} \geqslant x.c$$

where k is the number of executives in P corresponding to task x. A maximal constraint on a critical time interval (b, e) can be re-written into the following form:

$$\sum_{k \in [f,l]} c_k \leqslant (e - b) - MSO(e - b)$$

where f and l are defined in the definition of function PO.

For a given task set T, the execution time of tasks, critical intervals, and the values of function MSO on all lengths of critical intervals are all fixed. Therefore, if the execution times of the executives are considered as variables, all minimal and maximal constraints are linear constraints. Therefore, the Weight Assigner can be defined by two steps: Step 1 generates the set of minimal constraints and maximal constraints based on T and P; Step 2 runs a linear programming solver to find an assignment to the execution times of the executives satisfying all constraints.

Theorem 8 *Given task set T and a relevant plan P, the Weight Assigner produces a pre-schedule L.*

Proof: Linear programming problems can be solved optimally. Therefore this theorem directly follows from Theroem 7. ■

Theorem 9 *The time complexity of the Weight Assigner is $O(C(n_p^2) + n \cdot F)$, where C is the complexity of the linear programming solver, n_p and n are respectively the number of periodic tasks and all tasks, and F is the size of a frame.*

Proof: The number of critical time intervals is $O(n_p^2)$, The number of minimal constraints is $O(n_p)$, and so the number of constraints in the linear programming problem is $O(n_p^2)$. According to Theorem 4, the number of executives in a plan is $O(n_p^2)$. Therefore, the problem size of the linear programming problem is $O(n_p^2) + O(n_p^2) = O(n_p^2)$, and its time complexity is $O(C(n_p^2))$. The Saturated Test is executed once to compute MSO values on all critical time intervals, and so the complexity is $O(n \cdot F)$. Therefore the total complexity is $O(C(n_p^2) + n \cdot F)$. ■

6 Example

We illustrate the generation of a pre-scheduler with the following example.

Example 2 The length of a frame is 24, and periodic task set T_p consists of four streams of periodic tasks sets defined as follows:
$(0,(1,0,3))$, $(2,(1,3,3))$, $(3,(1,6,3))$, $(6,(1,9,3))$, $(8,(1,12,3))$, $(9,(1,15,3))$, $(11,(1,18,3))$, $(13,(1,21,3))$;
$(1,(1,0,5))$, $(4,(1,5,5))$, $(7,(1,10,5))$, $(10,(1,15,5))$;
$(5,(1,0,10))$, $(12,(1,11,10))$;
$(14,(2,0,24))$.
In Figure 8, each periodic task is represented by a slot with two vertical bars at both ends. The position of the first vertical bar corresponds to the arrival time r of the task, and the length of the slot corresponds to the relative deadline d. The top tasks are drawn with solid bottom lines, while non-top tasks are drawn with dashed bottom lines. The sporadic task set T_s is defined as $\{(1,4,4)\}$.

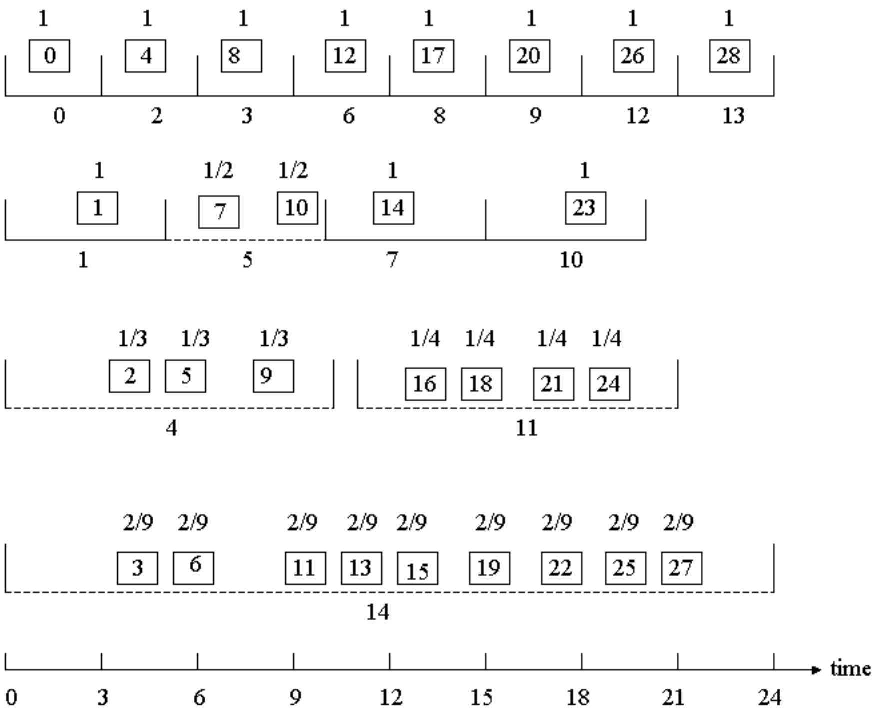

Figure 8: Producing a Pre-schedule

The Planner generates a list of executives. Each box represents an executive and the number inside the box indicates its position in the list. If a box is in the slot of a task, the corresponding executive is an executive of the task. For example, the 14th executive in the list corresponds to task 7. It is easy to verify that all constraints in the definition of Planner are satisfied. For example, task 3 and task 6 are two adjacent top tasks, executive interval $(8, 12)$ is a gap. Task 4, 5, and 14 contain task 3, and therefore there is one and only one executive for each of the task between gap $(8, 12)$. Therefore Constraint 3 is satisfied in this case. Because task 4 is before task 5, the executive of task 4 in this gap (executive 9) is before the executive of task 5 (executive 10) in this gap. Because task 14 is after task 4 and contains task 5, its executive in this gap is after the executives of task 4 and 5. Therefore Constraint 4 is satisfied in this gap.

The Weight Assigner assigns an execution time for each executive, which is shown on the top of each box. All minimal and maximal constraints are satisfied by this assignment. It is obvious that for every task, the accumulated execution time of all the executive of the task is equal to the execution time of the task, and therefore the minimal constraints are satisfied. All maximal constraints are also satisfied; for example, given a critical time interval $(5, 10)$, $MSO(5) = 2$, $PO(5, 10) = \sum_{7 \leqslant k \leqslant 10} k.c = 2\frac{1}{3}$ and hence $MSO(10 - 5) + PO(5, 10) = 4\frac{1}{3} \leqslant 10 - 5 = 5$, so the maximal constraint on this critical time interval is satisfied. Therefore, Figure 8 represents a pre-schedule of 29 executives: $\{(0, 1), (1, 1), (2, \frac{1}{3}), ..., (14, \frac{2}{9}), (13, 1)\}$.

7 Conclusion

This paper proposes a pre-scheduling approach to combine the advantages of static schedulers with those of dynamic schedulers. The technical challenge is how to produce pre-schedules for any given set of periodic and sporadic tasks. The challenge is answered in this paper by an optimal and efficient pre-scheduler.

In terms of practical impact, the pre-scheduling approach is a natural way to extend the flexibility of real-time system architectures such as the Time Triggered Architecture (TTA) without sacrificing important requirements such as mechanical certification and ease of support for fault tolerance protocols. For embedded applications that are safety-critical but may not require fully dynamic schedulers, the pre-scheduling approach opens up new possibilities for making trade-offs in the design space. This is especially so for embedded systems that have both time-driven (periodic) and event-driven (sporadic) tasks with short deadlines.

For future work, we propose further generalizations on this approach, especially in the following directions. First, this paper investigates the pre-scheduling problem where tasks are independent. The problem is NP-hard for the general case where tasks may have arbitrary dependencies. However, sub-optimal but practically effective algorithms to pre-schedule task sets with dependencies may be possible in cases of practical interest. Second, this paper uses a very simple on-line scheduling strategy - sporadic tasks always have higher priorities over periodic tasks. It is certainly possible to replace the two-priority scheme with other strategies, such as embedding a sporadic server or a weighted fair-queueing scheduler in a static scheduler by the pre-scheduling approach. Third, even though the integer programming problem is

NP-hard in general, there are techniques that we can exploit to optimally produce pre-schedule on integer problems in practice. we shall pursue these problems in our follow-up work.

References

[1] N. Audsley, A. Burns, M. Richardson, K. Tindell, A.J. Wellings. Software Engineering Journal, 8(5): 284-292, 1993.

[2] N. Audsley, K. Tindell, A. Burns. The End of The Line for Static Cyclic Scheduling? Proceedings of 5th Euromicro Workshop on Real-Time Systems, Oulu, Finland, pp.36-41, 1993.

[3] N. C. Audsley, A. Burns, R.I. Davis, K. W. Tindell, A. J. Wellings. Fixed Priority Pre-emptive Scheduling: An Historical Perspective, Real-Time Systems, Vol. No.8, pp. 173-198, 1995.

[4] T. P. Baker, A. Shaw. The cyclic executive model and Ada, Proceedings of IEEE Real-Time Systems Symposium, pp.120-129, December 1988.

[5] S. Baruah, A. Mok, L.Rosier. Preemptively Scheduling Hard-Real-Time Sporadic Tasks on One Processor, Proceedings of the Elventh Real-Time Systems Symposium, pp. 182-190, Dec 1990.

[6] A. Burns. Fixed Priority Scheduling with Deadlines Prior to Completion, YCS 212, 1993.

[7] S. Choi and Ashok K. Agrawala. Dynamic Dispatching of Cyclic Real-Time Tasks with Relative Timing Constraints, Real-Time Systems, Vol. 19, No.1, pp. 5-40, July 2000.

[8] J. Erschler, F. Fontan, C. Merce, F. Roubellat. A New Dominance Concept in Scheduling n Jobs on a Single Machine with Ready Times and Due Dates, Operations Research, 31:114-127.

[9] R. Gerber, W. Pugh, and M. Saksena. Parametric Dispatching of Hard Real-Time Tasks, IEEE Trans. on Computers, Vol.44, No.3, pp. 471-479, Mar 1995.

[10] J. Lehoczky, L. Sha and J. K. Strosnider. Enhanced Aperiodic Responsiveness in Hard Real-Time Environments, Real-Time Systems Symposium. pp. 261-270, 1987.

[11] J. Lehoczky, L. Sha and Y. Ding. The Rate Monotonic Scheduling Algorithm-Exact Characterization and Average Case Behavior, Real-Time Systems Symposium. pp. 166-171, Dec 1989.

[12] John P. Lehoczky. Fixed Priority Scheduling of Periodic Tasks Sets with Arbitrary Deadlines, Real-Time Systems Symposium, pp. 201-209, Dec. 1990.

[13] John P. Lehoczky. An Optimal Algorithm for Scheduling Soft-Aperiodic Tasks in Fixed-Priority Preemptive Systems, Real-Time Systems Symposium, pp. 110-123, 1992.

[14] John P. Lehoczky. Algorithms for Scheduling Hard Aperiodic Tasks in Fixed-Priority Systems using Slack Stealing, Real-Time Systems Symposium, pp. 22-33, 1994.

[15] C. D. Locke, Software Architecture for Hard Real-Time Applications: Cyclic Executives vs. Fixed Priority Executives, Real-Time Systems, Vol. 4, No. 1, pp. 37-53, 1992.

[16] A. K. Mok, X. Feng. Towards Compositionality in Real-Time Resource Partitioning Based on Regularity Bounds. Real-Time Systems Symposium, pp. 129-138, 2001.

[17] J. Regehr, J. A. Stankovic. HLS: A Framework for Composing Soft Real-Time Schedulers. Real-Time Systems Symposium, pp. 3-14, 2001.

[18] I. Ripoll, A. Crespo, A. K. Mok. Improvement in Feasibility Testing for Real-Time Tasks, Real-Time Systems, Vol. 11, No. 1, pp. 19-39, 1996.

[19] M. Spuri, G. Buttazzo. Scheduling Aperiodic Tasks in Dynamic Priority Systems, Real-Time Systems, Vol. 10, No.2, pp. 179-210, 1996.

[20] B. Sprunt, L. Sha and J. Lehoczky. Aperiodic Task Scheduling for Hard Real-Time Systems, Real-Time Systems, Vol. 1, No.1, pp.27-60, 1989.

[21] Manas Chandra Saksena. Parametric Scheduling for Hard Real-Time Systems. PhD thesis, Department of Computer Science, University of Maryland, 1994.

[22] T.S. Tia. Utilizing Slack Time for Aperiodic and Sporadic Requests Scheduling in Real-time Systems, Technical Report, UIUCDCS-R-95-1906.

[23] T.S. Tia, J. W.S. Liu, M. Shankar, Algorithms and Optimality of Scheduling Soft Aperiodic Requests in Fixed-Priority Preemptive Systems, Real-Time Systems, Vol. 10, pp. 23-43, 1996.

[24] W. Wang, A. K. Mok, Pre-Scheduling:Balancing Between Static and Dynamic Schedulers, UTCS Technical Report RTS-TR-02-01, 2002, http://www.cs.utexas.edu/users/mok/RTS/pubs.html.

[25] X. Yuan, M.C. Saksena, A.K. Agrawala, A Decomposition Approach to Non-Preemptive Real-Time Scheduling, Real-Time Systems, Vol. 6, No. 1, pp. 7-35, 1994.