A Priority-Based Preemption Algorithm for Incremental Scheduling with Cumulative Resources

Qu Zhou and Stephen F. Smith

The Robotics Institute
Carnegie Mellon University
5000 Forbes Avenue, Pittsburgh, PA 15213
{quzhou,sfs}@cs.cmu.edu

CMU-RI-TR-02-19

July 2002

ABSTRACT

When scheduling in dynamic continuous environments, it is necessary to integrate new tasks in a manner that reflects their intrinsic importance while at the same time minimizing solution change. A scheduling strategy that re-computes a schedule from scratch each time a new task arrives will tend to be quite disruptive. Alternative a purely non-disruptive scheduling strategy favors tasks that are already in the schedule over new ones, regardless of respective priorities. In this paper, we consider algorithms that attempt to strike a middle ground. Like a basic non-disruptive strategy, the algorithm we propose emphasizes *incremental* extension/revision of an existing schedule, rather than regeneration of a new schedule from scratch. However, by allowing selective *preemption* of currently scheduled tasks, our algorithm also gives attention to the relative importance of new tasks. We consider a specific class of scheduling problems involving the allocation of cumulative (or multi-capacity) resources. We develop an approach to preemption based on the concept of freeing up a *resource area* (i.e., time and capacity rectangle) comparable to the resource requirement of the new task to be scheduled. Through experimental analysis performed with a previously developed system for air combat operations scheduling, we demonstrate that our priority-based preemption algorithm is capable of producing results comparable in solution quality to those obtained by regenerating a new schedule from scratch with significantly less disruption to the current schedule.

1. INTRODUCTION

As a scheduling policy, preemption has wide applications in many areas (e.g. CPU scheduling, bandwidth allocation, manufacturing scheduling). Most basically, preemption can be seen as a process that removes (un-schedules, suspends or aborts) one or more previously scheduled activities according to certain criteria and re-allocates freed resource capacity to a new activity. A preemption policy is normally used for scheduling high priority activities when a capacity shortage appears.

Preemption has been investigated fairly extensively relative to scheduling single-capacity resources. CPU scheduling, which is central to operating system design, is a representative example, The CPU is single-capacity resource, which can be time-shared to accommodate multiple tasks by algorithms (such as round robin) that repeatedly allocate time slices to competing tasks. Here, a preemptive scheduling policy provides a means for reallocating time slices as new, more important jobs arrive for processing.

Preemptive scheduling is much more complex in the context of cumulative or multi-capacity resources, and this problem has received much less attention in the literature. The principal complication concerns the selection of which activity (or activities) to preempt. In the case of multi-capacity resources, the number of candidate sets of activities increases exponentially with resource capacity size, while only a single activity must be identified in the single-capacity case.

In this paper, we consider the problem of preemptive scheduling of multi-capacity resources. Our broad interest is to define scheduling mechanisms for *continuous* scheduling environments. By continuous scheduling, we refer to an ongoing planning and execution process, where the scheduler receives new (previously unknown) batches of tasks over time, and these new tasks must be accomplished together with currently executing and previously scheduled tasks. Customer order scheduling and air operations scheduling are examples of continuous scheduling environments. Since, a key concern in such environments is to attempt to minimize change (even when responding to the receive of new, higher priority tasks), we focus on the design of *incremental* scheduling techniques. Incremental scheduling techniques emphasize revision and extension of an existing schedule rather than periodic regeneration of a new schedule from scratch. We propose an incremental, priority-based preemption algorithm for use in continuous, multi-capacity resource scheduling problems, and evaluate its performance in a complex air campaign scheduling domain where aircraft and munitions capacity must be allocated to various planned air missions.

2. RESEARCH MOTIVATIONS

Generally there are two extreme strategies for scheduling in continuous environments as new tasks become known. The first is to simply discard the current schedule, and regenerate a new schedule that incorporates both previously known (but not yet executed) tasks and new tasks. The advantage of this type of strategy is that all tasks can be scheduled at the same time, using the same scheduling criteria. For example, more important tasks (new or old) can always be given priority to receive the best

resources and time durations. Its disadvantage is obviously continuing disruption to the existing schedule.

The second extreme strategy is to schedule new tasks "on top" of the existing schedule, treating existing resource reservations as additional constraints and allocating only excess available resource capacity to newly received tasks. The benefit of using this strategy is that there is no disruption to the existing schedule. However, some high priority tasks may fail to be scheduled due to the lack of free capacity.

The preemption approach proposed here aims at providing some middle ground. By using this approach, a new task can preempt previously scheduled (but lower priority) tasks if there is not enough free capacity to accommodate the new task within its time constraints. At the same time,, by being careful to preempt only those tasks necessary to allow scheduling of new higher priority tasks, the approach attempts to minimise the amount of disruption caused to the current schedule,. Our research work is directly motivated by the problem of air campaign scheduling, which we are addressing within the DARPA "JFACC After Next" (Joint Forces Air Component Commander) research program [Myers and Smith 1999]. The preemption approach is needed to schedule new important tasks (missions) as they arise during an air campaign.

Note that although the field of reactive scheduling has proposed many strategies to deal with resource or time conflicts (e.g., [Sadeh et al., 1993, Smith, 1994, El Sakkout et al., 1998 and 2000]), most use a "shifting" strategy to move scheduled tasks in order to reduce resource and temporal contentions caused by scheduling a new task until the schedule consistency is restored. There are obviously limitations of these strategies.

- No matter how important it is, a new task will never be scheduled if the resource capacity is fully allocated and "shifting" fails to completely solve the time/resource conflicts.

- In reality, unscheduling one task can sometimes be a better solution than disturbing many previously scheduled tasks, though minimal temporal disruption can be achieved by means of certain algorithms [El Sakkout et al., 1998 and 2000].

3. PREVIOUS RESEARCH

Bar-Noy et al. [1999] proposed a preemption approach for bandwidth allocation in the design of networks where bandwidth must be reserved for connections in advance. Network bandwidth is a multi-capacity resource. Each task, called a *call*, requires one or more units of capacity of the network bandwidth for some specific duration. Requests for *calls* arrive one by one as time proceeds, and all requests must either be serviced immediately or rejected. To reduce overheads for this on-line scheduling system, two simple algorithms were presented. The optimal goal of each is to maximize the throughput of completed *calls*, where throughput is measured as the sum of the duration times the bandwidth (capacity) requirement.

The first algorithm, called left-right (*LR*) algorithm, implements a compromise between the need to hold on to jobs that have been running for the longest time and the need to hold on to jobs that will run for the longest time in the future. The algorithm gives half of bandwidth capacity to each of these two classes of jobs. In response to the algorithm two sets of *calls, L* and *R,* are created. The *L* set holds a sequence of jobs that are sorted by *increasing* order of start time. The *R* set holds a sequence of jobs that are sorted by *decreasing* order of ending time. A preemption or reject decision will be made to the *calls* that cannot stay in the sequence of the *L* or the *R* because of the sequence's capacity limit.

The second algorithm, called effective time (EFT) algorithm, implements a different compromise, in this case between banking on past profit and ensuring future profit. Rather than dividing the bandwidth between the two classes of jobs, this algorithm attaches a time-value, called the *effective time*, to each single *call*. *Calls* with later *effective times* are preempted first. The *effective time* of a call, $call_i$, is equal to the *call'*s arrival time minus its duration.

$$Effective\_time(call_i) = arrival\_time(call_i) - duration(call_i)$$

The idea behind the simple equation is that the work that has been done is worth twice as much as the work to be done.

The comparison of both algorithms shows that the LR algorithm is more effective for *calls* with small capacity requirement, while EFT is a better choice when *calls* require large amounts of capacity. One interesting feature of these algorithms is that neither algorithm uses the *call's* bandwidth (capacity) requirement as an evaluation criterion.

El Sakkout et al. [2000, 1998] combined constraint and linear programming techniques and proposed a unimodular probing backtrack search algorithm for minimizing temporal disruptions in reactive scheduling. This work also considered multi-capacity resources, under the assumption that all tasks require only a single unit of capacity. The constraint programming technique was used to restore schedule consistency, while the linear program was used to minimize temporal disruption from the original schedule. Temporal disruption is measured by the total time shift, i.e. the total change to the start and end times of disrupted tasks. The disruption function is defined to be the sum, over all temporal variables, of absolute time changes.

This algorithm can basically be divided into two phases: a *resource feasibility phase* and a *temporal optimisation phase*. First, potential resource conflicts caused by scheduling a new task are dealt with by posting the temporal overlays between tasks in the *resource feasibility phase*. Then, in the *temporal optimisation phase,* the values of the temporal variable are fixed to values that are consistent and optimal according to the minimal disruption function. Like most of papers on reactive scheduling, a preemption approach is not used in [El Sakkout et al., 1998 and 2000]. Its disruption function is also limited to the sum of temporal variable changes.

Ow and Smith et al. [1988, 1995] proposed a set of alternative modification actions for reactive scheduling and corresponding guidelines for action selection. Preemption, also called *bumping*, was introduced as one of the possible actions. When scheduling a new task, the scheduling search algorithm not only looks for currently free capacity but also considers capacity allocated to lower priority jobs as available capacity. However only single capacity resources and single capacity tasks were considered. To some extent, this paper can be considered as an extension of this work to allow solution of more complicated problems.

4. DESCRIPTION OF THE PROBLEM

Aiming at giving a formal description of the preemption problem, this section presents a preemption model that includes four types of representations: resource capacity profile and capacity intervals, scheduled and new operations[1], available capacity time blocks and preemption solution.

4.1 Resource capacity profile and capacity intervals

Figure 1 shows the capacity profile of a multi-capacity resource *r*. The total capacity (number of resource units) of the resource is denoted by $total\_res\_cap_r$, which is a positive integer. $total\_res\_cap_r$ can usually be divided into two parts: $free\_res\_cap_r(t)$ and $alloc\_res\_cap_r(t)$. $free\_res\_cap_r(t)$ is the capacity that has not yet been allocated to any operation; $alloc\_res\_cap_r(t)$ represents the capacity that has been allocated to operations. Both of them are functions of time, *t*. There is a relation between the three capacity representations (see Figure 1).

$$total\_res\_cap_r = free\_res\_cap_r(t) + alloc\_res\_cap_r(t), \text{ where } t \geq 0$$

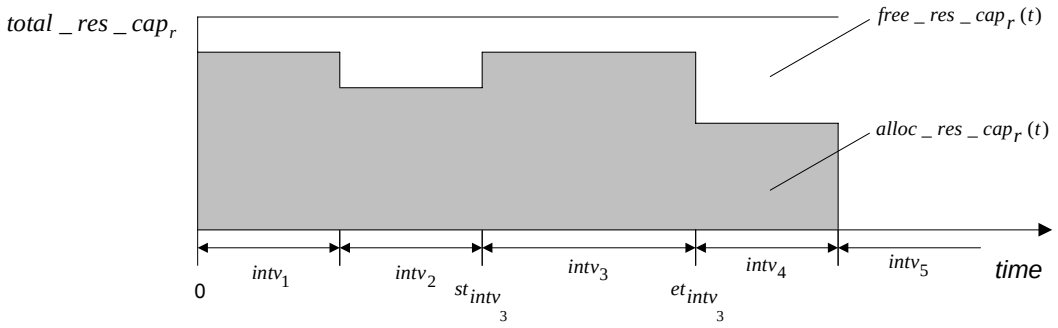

Figure 1 Resource capacity profile

[1] Operation: operations represent tasks in a schedule. A scheduled task has at least one operation to fullfil the task.

The capacity profile of resource *r*, CP_r, consists of a list of capacity intervals. For example, in Figure 1, $CP_r = \{intv_1, intv_2, ..., intv_5\}$. Each capacity interval, $intv_i$, is a period of time and has its start and end times, st_{intv_i} and et_{intv_i}. A given capacity interval is generated when a new operation is scheduled on the resource and the operation's start or end times are not equal to any capacity interval's start or end time. In this case an existing capacity interval should be split into two (or three) intervals. Figure 2 shows the generation of a new capacity interval after we schedule a new operation (see Figure 2).

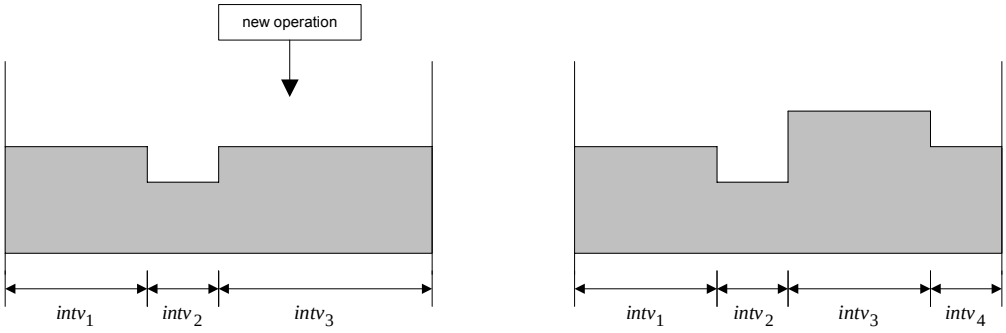

Figure 2 The generation of capacity intervals

It is not difficult to see from Figure 2 that $free\_res\_cap_r(t)$ and $alloc\_res\_cap_r(t)$ are constants within every capacity interval. That is

$\forall intv_i \in CP_r : free\_res\_cap_r(t_1) = free\_res\_cap_r(t_2)$, and

$$alloc\_res\_cap_r(t_1) = alloc\_res\_cap_r(t_2),$$

where $t_1 \neq t_2$, $st_{intv_i} \leq t_1 < et_{intv_i}$, $st_{intv_i} \leq t_2 < et_{intv_i}$.

4.2 Scheduled and new operations

We use O_r to represent all scheduled operations on resource *r* and op_i to represent a scheduled operation on resource *r*. Hence, $op_i \in O_r$. Let st_{op_i} and et_{op_i} denote the start and end times of op_i, and $priority_{op_i}$ and $capacity_{op_i}$ represents op_i's priority and booked capacity respectively. To represent the resource capacity consumed by op_i, we define the *resource area*, $res\_area_{op_i}$, of op_i. It can be computed by the following equation.

$$res\_area_{op_i} = (et_{op_i} - st_{op_i}) * capacity_{op_i}$$

For the convenience of the following discussion, we also represent the capacity of op_i as $cap_{op_i}(t)$, a function of time.

$$cap_{op_i}(t) = \begin{cases} capacity_{op_i}, & \text{if } st_{op_i} \le t < et_{op_i} \\ 0, & \text{if } t < st_{op_i} \text{ or } t \ge et_{op_i} \end{cases}$$

Likely, we use O_r' to represent a set of new operations that are required to be scheduled on resource *r*. op_k ($op_k \in O_r'$) represents a new operation which is required to be scheduled. Similar to op_i, $priority_{op_k}$ and $capacity_{op_k}$ are used to represent op_k's priority and its required capacity. Let est_{op_k}, let_{op_k} and d_{op_k} denote the earliest start time, latest end time and time duration of op_k respectively. Then we have

$$0 < d_{op_k} \le let_{op_k} - est_{op_k} \text{ and } res\_area_{op_k} = d_{op_k} * capacity_{op_k}$$

4.3 Available capacity time blocks

To schedule a new operation, op_k, on resource, *r*, using normal search mode (i.e., non-disruptively without preemption), we scan the resource's capacity profile to identify a set of *time blocks*, TB_r with sufficient available capacity. Each time block, tb_i ($tb_i \in TB_r$), should cover or partially cover a list of contiguous capacity intervals, $\{intv_{m_i}, intv_{m_i+1}, ..., intv_{m_i+n_i}\}$, where $m_i \ge 1, n_i \ge 0$. Let st_{tb_i} and et_{tb_i} denote tb_i's start and end times, then we have

$$st_{tb_i} = \max(st_{intv_{m_i}}, est_{op_k}), \text{ and}$$

$$et_{tb_i} = \min(et_{intv_{m_i+n_i}}, let_{op_k})$$

The reason for using *max* and *min* in the equations is because the first and last intervals ($intv_{m_i}$ and $intv_{m_i+n_i}$) can be partially coved by tb_i.

Obviously there are both time and capacity constraints on tb_i.

- tb_i's time constraint:

$$et_{tb_i} - st_{tb_i} \ge d_{op_k} \quad (1)$$

- tb_i's capacity constraint:

$$free\_res\_cap_r(t) \geq capacity_{op_k}, \text{ where } st_{tb_i} \leq t < et_{tb_i} \quad (2)$$

If no time block can be found because of constraint (2), preemption may be used to obtain additional resource capacity. We refer to this portion of capacity as the required preemption resource capacity, $req\_pre\_res\_cap_r(t)$ (see Figure 3). It can be provided by the operations currently scheduled within (or partially within) tb_i, which have lower priority than op_k has. Let $req\_pre\_res\_area_{tb_i}$ denotes the required preemption resource area, then we have :

$$req\_pre\_res\_area_{tb_i} = \int_{st_{tb_i}}^{et_{tb_i}} req\_pre\_res\_cap_r(t)dt .$$

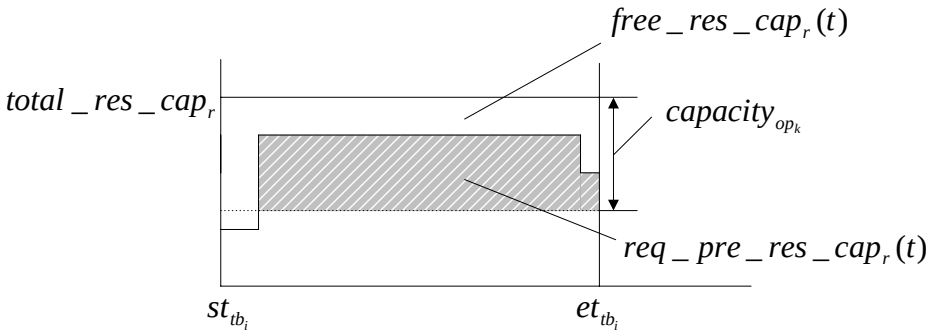

Figure 3 Required preemption area

If we define all lower priority operations covered (or partially covered) by tb_i as OPS_{tb_i}, and, furthermore, define all lower priority operations on resource *r* as O_{r,op_k}, then we have

$$OPS_{tb_i} = \{op_j : op_j \in O_r \text{ and } priority_{op_j} < priority_{op_k} \text{ and } (st_{tb_i} \leq st_{op_j} < et_{tb_i} \text{ or } st_{tb_i} < et_{op_j} \leq et_{tb_i})\}$$

$$O_{r,op_k} = \bigcup_{tb_i \in TB_r} OPS_{tb_i}$$

The new capacity constraint on tb_i can be formulated as the follows when preemption is applied:

- tb_i's capacity constraint in preemption mode:

  $$\sum_{op_j \in OPS_{tb_i}} cap_{op_j}(t) \geq req\_pre\_res\_cap_r(t) \geq capacity_{op_k} - free\_res\_cap_r(t)$$

  where $st_{tb_i} \leq t < et_{tb_i}$ (2')

After finding all time blocks based on this constraint, we can certainly preempt all lower operations in (or partially in) any time block to get enough capacity and time duration for op_k. However, this can often result in unnecessary disruption, since the resulting resource area that is generated is larger than required. The ideal approach is to select an optimal subset of operations from O_{r,op_k} and try to minimize disruptions to the existing schedule. We also call such a subset of operations a *preemption solution*. The number of preempted operations in the subset, their average priority and total resource area provide three basic criteria for measuring the extent of the disruption to the current schedule.

4.4 Preemption solution

Suppose we can find all subsets of O_{r,op_k}, which, together with existing free capacity, can provide both sufficient time duration and capacity to schedule op_k. Let S denote all these subsets, $S = \{S_1, S_2, ...S_n\}$. Then the preemption solution S^*_{r,op_k} ($S^*_{r,op_k} \subseteq O_{r,op_k} \subseteq O_r$) can be theoretically represented as the following.

$$S^*_{r,op_k} = \min_{S_i \in S}(w_1 * |S_i| + w_2 * \sum_{op_j \in S_i} priority_{op_j} / |S_i| + w_3 * \sum_{op_j \in S_i} res\_area_{op_j}) \quad (3)$$

As mentioned in the last section, three factors are used in the equation for the selection of S^*_{r,op_k}. The first is the number of preempted operations. The second is the average priority of these operations. The last is the sum of resource area of the operations. Different weights (w_i) can be assigned to the three factors. Note we have made an assumption that the operations in O_r cannot be partially preempted.

From the above preemption problem description, it can be seen that the crux of the multi-capacity preemption problem is theoretically the optimal selection of a subset of operations S^*_{r,op_k} from all lower priority operations O_{r,op_k} according to some criteria. It is obviously a combinatorial optimisation problem. In order to get S^*_{r,op_k} a search algorithm may search all the subsets of O_{r,op_k}, except the empty subset. The size of search space is $(2^{|O_{r,op_k}|} - 1)$. When $|O_{r,op_k}|$ is large, searching the space becomes impractical by using classic search methods (e.g. breadth-first, depth-first and best-first). Aiming at avoiding searching the exponential space, the next section proposes two approximate algorithms for efficiently determining a preemption solution (S_{r,op_k}).

5. PREEMPTION ALGORITHM

Two models will be discussed for preemption in this Section. The first model is a simple model in which a new operation does not have est_{op_k} and let_{op_k} values, but only st_{op_k} and et_{op_k}. It can be seen as a simplification of the general problem where

$$st_{op_k} = est_{op_k}, \quad et_{op_k} = let_{op_k}, \quad d_{op_k} = et_{op_k} - st_{op_k}$$

Resource allocation problems, in which an operation's start and end times have been fixed by previous processes, is an example of this model. Machine break down which can be seen as an operation with fixed start time and end time (or time infinity) also belongs to this model. A simple algorithm will be presented for this model in Section 5.1.

The second model considers the complete problem, in which est_{op_k} and let_{op_k} form a time window within which st_{op_k} and et_{op_k} can be established. An algorithm is also created for this model in Section 5.2. It basically reformulates the model into the first model to which the simple algorithm can be applied.

5.1 Preemption algorithm for operations with fixed start and end times

It is a simplified case to schedule a new operation that has fixed start and end times in the preemption mode. Only one time block can be found on resource *r* at most, i.e. $|TB_r| \le 1$. Figure 4 describes the preemption algorithm for the model.

```
Preemption Algorithm1
begin preemption_schedule1(op_id)
      tb1 <- scan_resource(resource_id,op_id,op_priority,op_est,op_let,op_cap)
      return generate_preemption_solution(tb1)
end
```

Figure 4 preemption algorithm1

First of all, the algorithm uses the ***scan_resource*** function to find the time block, *tb1* (tb_1), which has to satisfy both time and capacity constraints (constraint (1) and constraint (2') discussed in Section 4.3) on a time block. Then the algorithm calls a function called ***generate_preemption_solution***, which actually provides the core preemption algorithm to find the approximate preemption solution S_{r,op_k} from the time block specified. Figure 5 shows the core preemption algorithm.

```
Core Preemption Algorithm
begin generate_preemption_solution(tb)
      if tb≠NIL
        then ops <- get_candidate_operations(tb)
             req_area <- get_required_preemption_area(tb)
             sorted_ops <- sort_candidate_operations(ops)
             preempted_area <- 0
             candidate_sol <- NIL
             while complete_coverage(preempted_area,req_area)=NIL
                do op <- pop(sorted_ops)
                   area <- increase_preempted_area(op,preempted_area)
                   if more_coverage(area,preempted_area,req_area)
                     then preempted_area <- area
                          push(op, candidate_sol)
             final_sol <- candidate_sol
             while candidate_sol≠NIL
                do op <- pop(candidate_sol)
                   area <- decrease_preempted_area(op,preempted_area)
                   if complete_coverage(area,req_area)
                     then preempted_area <- area
                          delete(op, final_sol)
             return final_sol
      else
             return NIL
end
```

Figure5 Core preemption algorithm

For the better understanding of the algorithm, we can divide the whole preemption procedure into four steps. Figure 6 gives an example of algorithm and shows the four steps.

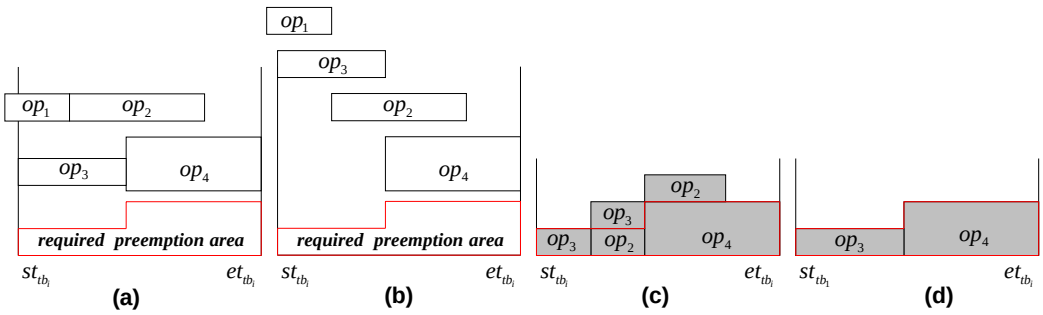

Figure 6 A preemption example of the core preemption algorithm. (a) generate a list of lower priority operations in tb_i, $OPS_{tb_i} = (op_1, op_2, op_3, op_4)$ ***and required preemption area (*** $req\_pre\_res\_area_{tb_i}$ ***). (b) sort operations in*** OPS_{tb_i} ***and get*** $SORTED\_OPS_{tb_i} = (op_4, op_2, op_3, op_1)$. ***(c) generate a preemption solution candidate list,*** $S'_{r,op_k} = (op_3, op_2, op_4)$, ***by popping the best operation from*** $SORTED\_OPS_{tb_i}$ ***and push it into*** S'_{r,op_k} ***one by one until the whole required preemption area can be coved. (d) refine*** S'_{r,op_k} ***by deleting redundant operations and generate an approximate preemption solution*** $S_{r,op_k} = \{op_3, op_4\}$.

Step 1: Generate the lower priority operations list OPS_{tb_i} and $req\_pre\_res\_area_{tb_i}$ for tb_i

According to the provided time block *tb* (tb_i), the core algorithm can get *ops* (OPS_{tb_i}) and *req_area* ($req\_pre\_res\_area_{tb_i}$) from *tb* by calling two functions, ***get_candidate_operations*** and ***get_required_preemption_area***.

Step 2: Sort operations in OPS_{tb_i}

The function ***sort_candidate_operations*** is responsible for sorting operations in OPS_{tb_i} by using the following operation evaluation function. Let $SORTED\_OPS_{tb_i}$ (*sorted_ops* in the algorithm) denotes a list of sorted elements in OPS_{tb_i} and define $SORTED\_OPS_{tb_i} = (op_{s_1}, ...op_{s_j}, op_{s_{j+1}}, ...op_{s_n})$, then we have $eval\_op(op_{s_j}, tb_i) \le eval\_op(op_{s_{j+1}}, tb_i)$, where $1 \le j \le n-1$ and $n = |OPS_{tb_i}|$. This means that operations with lower evaluation values are more likely to be selected for preemption than operations with higher values.

The following three criteria are used to formulate the evaluation function $eval\_op(op_j, tb_i)$:

1. Priority ($priority_{op_i}$), represents the inherent importance of a given operation. Operations with lower priority values are less important. They are more likely to be preempted than higher value operations.

2. Effective resource area ($eff\_res\_area_{op_i}$), measures the resource area contribution of each operation to the required preemption area (see Figure 7). In order to reduce the total number of preempted operations, operations with bigger effective resource area are more likely to be preempted than the operations with smaller effective resource area.

3. Ineffective resource area ($ineff\_res\_area_{op_i}$), is the remaining resource area of an operation (see Figure 7). It represents the unnecessary resource area if the operation is selected for preemption. Although the over-preempted resource area could be reused by some new operations, some of the area can never be reused. So we prefer that operations with smaller ineffective operation is more likely to be selected for preemption.

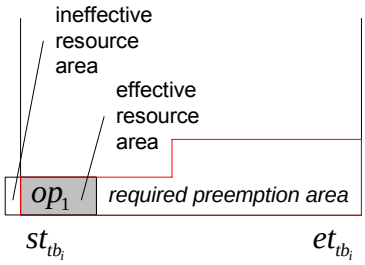

Figure 7 An operation's effective and ineffective resource area

Applying the effective and ineffective resource area criteria together can actually find the 'best fit' operation to fill the required preemption resource area in order to reduce the total number of bumped operations and their total resource area, which are two basic ideas of the core algorithm.

Given the above criteria, we define *eval_ op(op_j,tb_i)* as follows:

$$\begin{aligned} &\forall op_j \in OPS_{tb_i}: \\ &eval\_op(op_j, tb_i) = w_1 * priority_{op_j} / priority_{\max} - \\ &\qquad w_2 * eff\_res\_area_{op_j} / req\_pre\_res\_area_{tb_i} + \\ &\qquad w_3 * ineff\_res\_area_{op_j} / res\_area_{op_j} \end{aligned} \tag{4}$$

where $priority_{\max}$ is the maximal priority number used in the system.

The three weight coefficients (w_1, w_2 and w_3) indicate the importance of each criterion to the overall evaluation function. For example, one assignment we used for the experiments introduced later is $w_1 = 0.5$, $w_2 = 0.3$ and $w_3 = 0.2$. Data normalization is used in the evaluation function by means of computing the ratio of each criterion instance's value to the criterion's maximal value.

Step 3: Generate a preemption solution candidate list, S'_{r,op_k} ,

After obtaining a sorted list of operations, $SORTED\_OPS_{tb_i}$, the algorithm in Figure 5 iterates to generate a candidate solution S'_{r,op_k} . This candidate solution provides enough (and sometimes more than enough) preempted area to cover the required preemption resource area. Three functions are introduced in the loop.

1. ***complete_coverage***: This function compares two resource area objects (*preempted_area* and *req_area*) to see whether the *preempted_area* can completely cover the *req_area*. More operations should be added to form the candidate solution S'_{r,op_k} if the function returns 'NIL'.

2. ***increase_preempted_area***: This function increases the size of existing preempted area (*preempted_area*) by including a new operation (*op*). It, then, returns the size-increased resource area object to a temporal resource area object (*area*).

3. ***more_coverage***: This function takes two resource area objects (*area* and *preempted_area*) and compares them with the *req_area* object to see whether the *area* object has more coverage on the *req_area* than the *preempted_area* object. If the function returns 'T' (true), the *preempted_area* object will be replaced by the *area* object. The operation (*op*), then, will be added to the *candidate_sol* set that will finally form the S'_{r,op_k}), using the *push* operator.

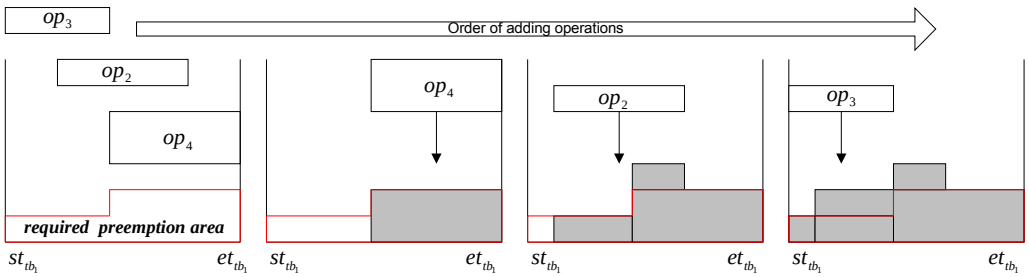

Figure 8 Generation of the candidate solution

As a simple example, Figure 8 shows the generation procedure of a candidate solution. To cover the whole required preemption area completely, we have $S'_{r,op_k} = (op_3, op_2, op_4)$.

Step 4: Refine S'_{r,op_k} and generate the approximate preemption solution S_{r,op_k}.

The last *while* loop in the algorithm in Figure 5 refines the candidate solution S'_{r,op_k} generated in the previous loop and forms the final preemption solution S_{r,op_k}. The refining process simply tries to take every operation from the S'_{r,op_k}, using ***decrease-preempted-area*** to check whether the remaining operations can still provide enough resource area. This action is performed in the reverse order of the order in which operations were added, because the better operations were added earlier in the generation of S'_{r,op_k}. For example Figure 8 shows the operation adding order, $op_4 \rightarrow op_2 \rightarrow op_3$. So the operation refining order should be $op_3 \rightarrow op_2 \rightarrow op_4$. After the loop, we can get the final solution set $S_{r,op_k} = \{op_3, op_4\}$, since op_4 and op_3 have provided enough resource area, and op_2 is redundant.

In summary, the algorithm introduced in the section provides a basic preemption approach. It avoids searching an exponential solution space. It is also very flexible in terms of the selection of the evaluation criteria and their weight coefficients.

5.2 Preemption algorithm for operations with no-fixed start and end times

In this section we discuss an extended model, which deals with the problem of scheduling new operations with non-fixed start and end times. Compared to the model introduced in Section 5.1, the model is different in two aspects in terms of algorithm design.

- The time blocks found can be bigger than what the new operation needs.
- More time blocks can be found on the same resource. That means $|TB_r| \geq 1$.

Since the time blocks found can be bigger than what the new operation requires, it is not necessary to preempt the entire time block's duration. Only a portion of it should be enough for the new operation. We define this portion as a *minimal duration time block*, which does not cover any unnecessary capacity intervals for the purpose of providing the new operation enough time duration. We also call it a *sub-time-block*, since it is always within a time block. The key of the preemption algorithm for operations with non-fixed start and end times is finding the best minimal duration time block to apply the core preemption algorithm introduced in Section 5.1.

5.2.1 Minimal duration time block

If we use stb_j to represent a minimal duration time block (sub-time-block) in tb_i, st_{stb_j} and et_{stb_j} to represent its start and end time, $INTV_{stb_j} = \{intv_{m_j}, intv_{m_j+1}, ..., intv_{m_j+n_j}\}$ $(m_j \geq m_i \geq 1, n_j \geq 0, m_j + n_j \leq m_i + n_i)$ to represent the list of contiguous capacity intervals it covers, then we have

$$st_{stb_j} = \max(st_{intv_{m_j}}, est_{op_k}) \geq st_{tb_i}, \text{ and}$$

$$et_{stb_j} = \min(et_{intv_{m_j+n_j}}, let_{op_k}) \leq et_{tb_i}$$

$$req\_pre\_res\_area_{stb_j} = \int_{st_{stb_j}}^{et_{stb_j}} req\_pre\_res\_cap_r(t)dt$$

Here are the constraints on stb_{ij}, when preemption is applied.

- stb_j's time constraint :

$$et_{stb_j} - st_{stb_j} \geq d_{op_k},$$
$$\sum_{intv_l \in (INTV_{stb_j} - \{intv_{m_j}\})} (\min(et_{intv_l}, et_{stb_j}) - st_{intv_l}) < d_{op_k}, \text{ and}$$
$$\sum_{intv_l \in (INTV_{stb_j} - \{intv_{m_j+n_j}\})} (et_{intv_l} - \max(st_{intv_l}, st_{stb_j})) < d_{op_k} \quad (5)$$

- stb_{ij} 's capacity constraint in preemption scheduling mode:

$$\sum_{op_j \in OPS_{stb_j}} cap_{op_j}(t) \geq req\_pre\_res\_cap_r(t) \geq capacity_{op_k} - free\_res\_cap_r(t),$$

where $st_{stb_j} \leq t < et_{stb_j}$,

$$OPS_{stb_j} = \{op_l : op_l \in O_{r,op_k} \text{ and } (st_{stb_j} \leq st_{op_l} < et_{stb_j} \text{ or } st_{stb_j} < et_{op_l} \leq et_{stb_j})\} \quad (6)$$

It can be seen that stb_j 's capacity constraint is the same as tb_i 's. However, as claimed by its name, the minimal duration time block stb_j 's time constraint is tighter. As an example Figure 9 shows a time block tb_1 and all its sub-time-blocks.

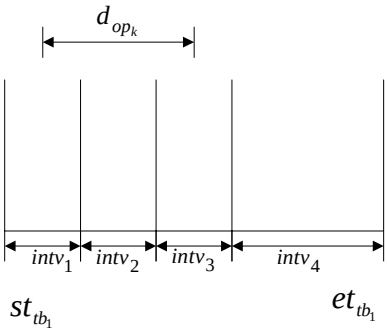

Figure 9 Capacity intervals in a time block

tb_1 consists of four capacity intervals. $INTV_{tb_1} = \{intv_1, intv_2, intv_3, intv_4\}$. Three sub-time-blocks, stb_1, stb_2 and stb_3 can be found in it. Their capacity interval profiles are as follows:

$INTV_{stb_1} = \{intv_1, intv_2\}$,

$INTV_{stb_2} = \{intv_2, intv_3\}$

$INTV_{stb_3} = \{intv_4\}$.

Note that $\{intv_3, intv_4\}$ is not a qualified sub-time-block because of the time constraint (4). $intv_4$ provides enough duration for the new operation and $intv_3$ is redundant.

5.2.2 Algorithm for finding a good minimal duration time block

Let STB_r denotes all minimal duration time blocks found for operation op_k on resource r, we can see that the idea of using the core preemption algorithm repeatedly

to obtain a preemption solution is $|STB_r|$ times expensive than the case of operations with fixed start and end times. This idea is not desirable in some cases, although the time it needs is not exponentially expensive. Here we try to design an algorithm to find a good minimal duration time directly.

We decide how good a sub-time-block is by considering two factors.

- What is the size of required preemption area in this sub-time-block?
- How good is the best operation for preemption in the sub-time-block?

Figure 10 describes the algorithm.

```
Preemption Algorithm2
begin preemption_schedule2(op_id)
        TB <- scan_resource(resource_id,op_id,op_priority,op_est,op_let,op_cap)
        STB <- NIL
        SORTED_STB<-NIL
        ops <- NIL
        for each time block tb in TB
            do push(generate_sub_time_blocks(tb), STB)
                push(get_candidate_operations(tb), ops)
        SORTED_STB <- sort_sub_time_blocks(STB,ops)
        best_stb <- pop(SORTED_STB)
        return generate_preemption_solution(best_stb)
end
```

Figure 10 Preemption algorithm2

Similar to the preemption algorithm for the first model, we first use ***scan_resource*** to get time blocks. However a set of time blocks, $TB(TB_r)$, should be returned in this model. Then there is a *for* loop in the algorithm. The main function in the loop is ***generate_sub_time_blocks***. It catches all the sub-time-blocks, $STB(STB_r)$, which satisfy constraints (5) and (6) from all time blocks in TB_r. All the candidate operations in TB_r are also caught and stored in a list, $ops(O_{r,op_k})$, using ***get_candidate_operations*** which simply picks up all candidate operations from tb_i.

After the loop there is a major function called ***evaluate_sub_time_blocks,*** which evaluates every sub-time-block in STB_r and sorts them. Let $SORTED\_STB_r$ ($SORTED\_STB$ in the algorithm) denote a sorted list of all elements in STB_r and define $SORTED\_STB_r = (stb_{s_1},...stb_{s_j}, stb_{s_{j+1}},...stb_{s_n})$, then we have $eval\_stb(stb_{s_j}) \le eval\_stb(stb_{s_{j+1}})$, where $1 \le j \le n-1$ and $n = |STB_r|$.

To evaluate a given sub-time-block, we consider two criteria:

1. Required preemption resource area ($req\_pre\_res\_area_{stb_j}$), represents the preemption resource area required by the new operation op_k if stb_j is selected for preemption. A sub-time-block requiring smaller preemption areas is more likely to be selected for preemption.

2. The operation most preferred for preemption in stb_j indicates how good a resource area fit the fittest operation is, if stb_j is selected for preemption. The evaluation function defined for this purpose previously in Equation (4) of Section 5.1 -- $eval\_op$ -- is used here again.

Given these considerations, the evaluation function used to rank sub-time-blocks is as follows:

$$\forall stb_j \in STB_r :$$

$$eval\_stb(stb_j) = w_1 * req\_pre\_res\_area_{stb_j} / res\_area_{op_k} + w_2 * \min(eval\_op(op_l, stb_j) : op_l \in OPS_{stb_j})$$

Two weight coefficients (w_1 and w_2) determine the relative importance of each criterion to the evaluation function. For example, one assignment we used for the experiments introduced later is $w_1 = 0.6$ and $w_2 = 0.4$. Data normalization is also used in the evaluation function.

Finally we select the best sub-time-block, *best_stb* , by popping the first sub-time-block in $SORTED\_STB$. We then use the core preemption algorithm by calling the ***generate_preemption_solution*** function with the parameter *best_stb*, and generate the final solution S_{r,op_k} for the new operation op_k on the given resource *r*.

In summary, the preemption algorithm introduced in this section for scheduling new operations with non-fixed start and times can be seen as an extension version of the algorithm proposed in Section 5.1 for scheduling new operations with fixed start and end times. It is less expensive than applying Section 5.1's algorithm repeatedly to every sub-time-block, which could be an alternative approach.

Note that a preempted operation $op_j (op_j \in S_{r,op_k})$ can be rescheduled either by using the preemption algorithms again or by using the normal scheduling algorithm (in which case there is no further preemption and disruption).

6. EXPERIMENTS AND APPLICATIONS

In order to test the preemption algorithms, a set of experiments were designed based on the JFACC Scheduler.

The core scheduling algorithm of the JFACC Scheduler is designed as a depth first search algorithm (Figure 11). When attempting to schedule a given mission[2] request (e.g. hitting a target) the scheduler first selects the most appropriate weaponeering solution (WS) for the target. A WS designates an <aircraft-type munitions-type> pair. According to the selected WS, the most suitable base wing (BW) for flying the mission is then chosen. A given BW consists of some number of a particular type of aircraft (e.g., an F15 fighter) and some numbers of associated munitions. There may be several candidate base wings that can provide a specific WS, each positioned in a different location. The closest BW is normally explored first. If that base wing does not have enough free capacity for the operation request, the second base wing will also be explored.

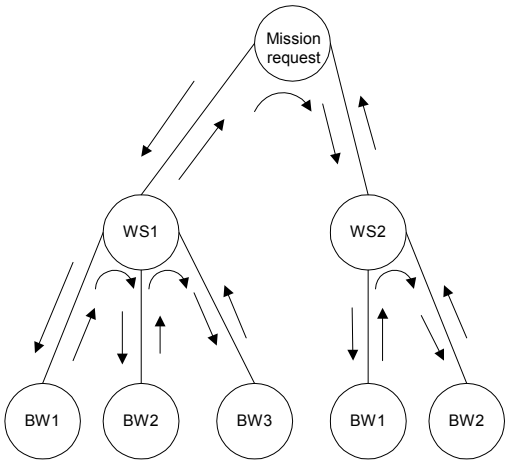

Figure 11 Scheduling search mechanism for JFACC scheduler

Using this search algorithm, a base plan (a set of operation requests) was first scheduled to provide a backdrop for the experiments. Note that a large percentage of capacity of various base wings is consumed after scheduling the base plan. Next a new batch of new operation requests (referred to as a new plan *assertion*) arrives. As introduced in Section 2, two extreme approaches can be used at this stage. One approach, called *regeneration*, involves unscheduling the operations in the existing base plan, mixing them with the operations in the new assertion and then generating a new schedule from scratch. Another approach, called *non-disruption*, involves simply scheduling the assertion on the top of the base plan, using only excess base wing capacity. Our experiment is designed to compare our preemption approaches to these two approaches. Based on the preemption algorithm2 introduced in Section 5.2, we further designed two preemption approaches: *preemption-1* and *preemption-2*.

[2] mission: we use the term `mission' instead of `task' in this application domain.

Preemption-1 only finds the first preemption solution. It searches the set of base wings one by one until an S_{r,op_k} is found. Preemption-2 is similar to the preemption-1 except that it does not stop when the first S_{r,op_k} is found. Instead, it tries to find all preemption solutions from all possible resources, R, and then selects the best. Let S_{op_k} denote the best preemption solution that can be found on R, then we have $S_{op_k} = \min(eval\_pre\_sol(S_{r,op_k} : r \in R))$. Here the evaluation function, $eval\_pre\_sol$, is defined as follows. It is similar to the theoretical equation (Equation (3)), except data normalization is used here.

$$eval\_pre\_sol(S_{r,op_k}) = w_1 * \left|S_{r,op_k}\right| / \max\{\left|S_{l,op_k}\right| : l \in R\} +$$
$$w_2 * (\sum_{op_j \in S_{r,op_k}} priority_{op_j} / \left|S_{r,op_k}\right|) / priority_{\max}\} +$$
$$w_3 * \sum_{op_j \in S_{r,op_k}} res\_area_{op_j} / \max\{\sum_{op_j \in S_{l,op_k}} res\_area_{op_j} : l \in R\}$$

where $R = \{\text{all base wings that can support } op_k\}$

Three criteria are considered in the evaluation function. They are the total preempted resource area, average priority of preempted operations and the number of preempted operations. A weight coefficient assignment we used for the experiment is $w_1 = 0.5$, $w_2 = 0.3$ and $w_3 = 0.2$.

Table 1 shows the experiment data we used. There are a base-plan that has 253 missions and 3 assertion-plans (new sets of missions) that have 8, 25 and 86 new missions respectively. Combining the base-plan with the three assertion-plans yields three sets of mission data for the experiment. At the same time we prepared 11 sets of resource data, which have decreasing levels of resource capacity labelled from 100% to 50%. Using both resource data and mission data we formulated 33 (3*11) experiments. The four scheduling approaches were applied to every experiment to get comparison results. The four comparison criteria are listed in Table 2.

Table 1 Experiment data

Mission Data	***Resource Data***
Base-plan (253 missions) + Assertion-plan-1 (8 missions)	100% Resource profile
Base-plan (253 missions) + Assertion-plan-2 (25 missions)	95% Resource profile
Base-plan (253 missions) + Assertion-plan-3 (86 missions)	90% Resource profile
	85% Resource profile
	80% Resource profile
	75% Resource profile
	70% Resource profile
	65% Resource profile
	60% Resource profile
	55% Resource profile
	50% Resource profile

Table 2 Comparison criteria

	Criteria	***Description***
1	Running time	The running time needed for scheduling the assertion.
2	Percentage of non-satisfied missions	The ratio of the number of missions that cannot be scheduled to the total number of missions
3	Average priority of the non-satisfied missions	The average priority of all the missions that cannot be scheduled.
4	Percentage of disrupted mission operations	The ratio of the number of scheduled mission operations in the base-plan that are disrupted because of scheduling the assertion to the total number mission operations in the base-plan schedule

Figure 12 shows the comparison result of computation time. The regeneration approach needs more time than preemption approach when the assertion plan is small. However the preemption approach (especially the preemption-2) becomes more time expensive when the assertion plan is big enough, although the preemption only needs to schedule the assertion plan while the regeneration has to reschedule the whole plan (including the base plan). There are two reasons for the increase. Firstly the preemption algorithm tries to find best solution with least disruptions to the existing

schedule, while the regeneration approach only picks up the first available time block. Secondly in order to avoid making higher priority missions unschedulable, the preemption approach uses 'cascading' preemption. That is, a preempted mission can preempt other missions if its priority is higher than theirs. The non-disruption approach always uses less time than the others. It is also very clear that preemption-2 is more time expensive than the preemption-1, since it does more search.

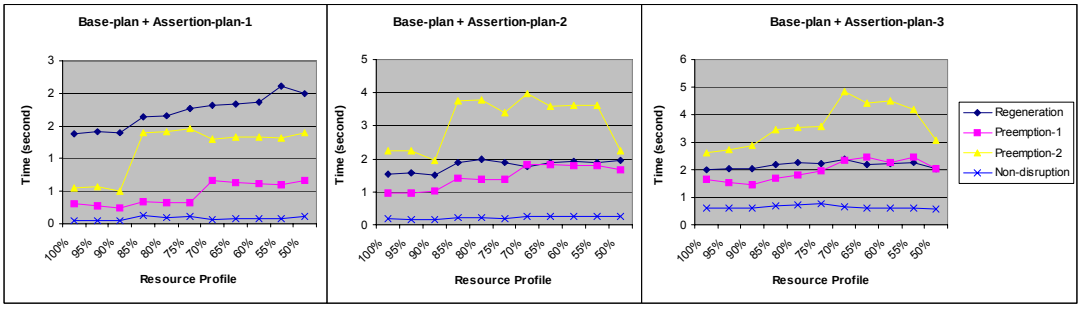

Figure 12 Run time comparison

Figure 13 shows the percentage of non-satisfied missions comparison. There is no clear difference between the four approaches, since random priorities are used in the experiments. Each mission in the experiment was assigned a priority without any considerations on the mission size (resource requirement). It can be expected that the percentage of non-satisfied missions can be reduced if small size missions have priority in scheduling.

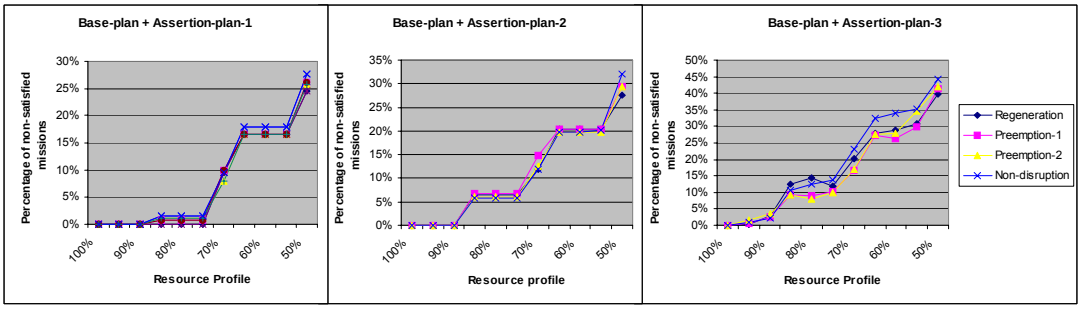

Figure 13 Percentage of non-satisfied missions comparison

However there is a clear difference in terms of the average priority of the non-satisfied missions (Figure 14). The regeneration approach always produces the lowest average priority, while the non-disruption approach always produces the highest. Both preemption approaches do pretty well and get very close to the regeneration approach regarding the average priority.

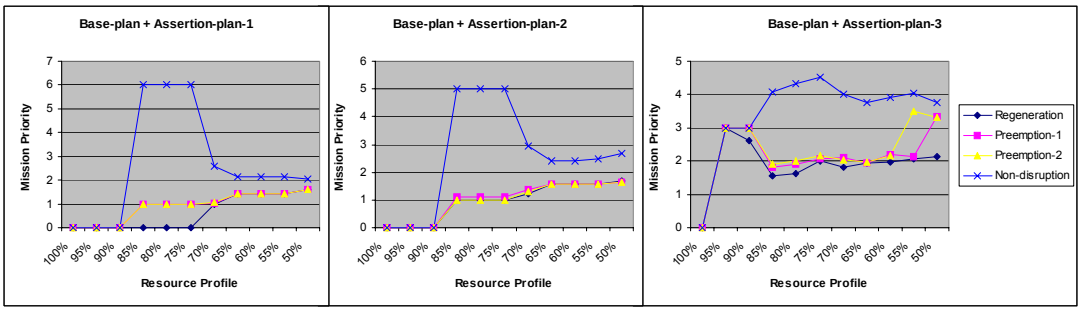

Figure 14 Average priority of non-satisfied mission comparison

The final evaluation metric used is disruption (see Figure 15). The result, as we expected, shows that the regeneration causes the biggest disruption, while the non-disruption approach, as its name indicates, causes no disruption. The important results on disruption come from the preemption approaches. The preemption approaches cause some disruption, but it is less than that caused by the regeneration approach. Particularly the preemption-2 approach causes much less disruption than the preemption-1 approach. Comparing Figure 12 and 15 we can see clearly the trade-off between time and disruption for both preemption approaches.

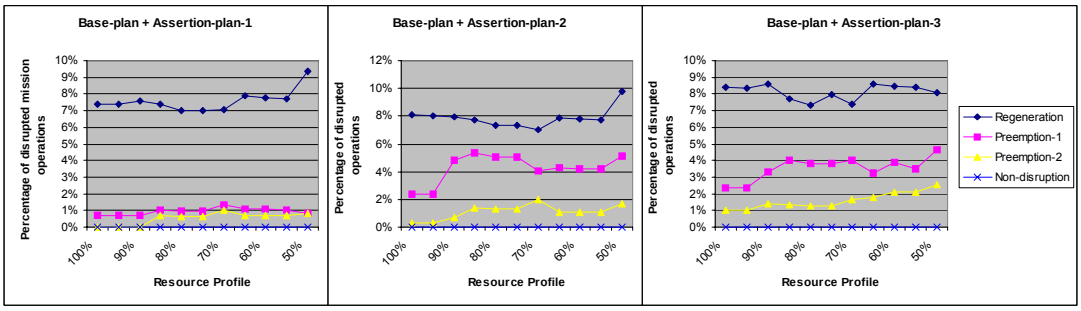

Figure 15 Percentage of disrupted mission operations comparison

In summary, the preemption algorithm proposed in the paper provides a third way for continuous scheduling. It can not only schedule new higher priority missions when there is resource contention, but also minimize disruptions on the existing schedule. Furthermore, the preemption algorithm can be used in other occasions, although its research motivation is incremental scheduling. For example the maintenance / break-down operations can also be scheduled by using the preemption algorithm. The preemption can even be modified and used as a major scheduling mechanism, in which case every operation tries to preempt lower priority scheduled operations and get its best resource and time duration even if there is enough free resource capacity for it. The advantage of the preemption mechanism over most of current scheduling mechanisms is that every operation is guaranteed to get its best resource and best time duration according to its priority, regardless of the sequence of scheduling.

However a shortcoming of the preemption algorithm has also been identified. Small time gaps (e.g. 1-2 minutes time gaps) may appear after preemption and become useless afterwards. So the algorithm should be used carefully for resource based scheduling, in which resource utilisation is the main optimisation goal for scheduling. One approach to overcoming this shortcoming might be to couple the use of a compaction strategy to squeeze out time gaps where possible.

ACKNOWLEDGEMENTS

The research reported in this paper was supported in part by the Department of Defense Advanced Research Projects Agency (DARPA) under contract F30602-97-2-0066 and the CMU Robotics Institute.

REFERENCE

Bar-Noy A., Canetti R., Kutten S., Mansour Y. and Schieber B., 1999, Bandwidth allocation with preemption, *SIAM journal on computing*, Vol.28, No.5.

Davenport A.J., Managing uncertainty in scheduling: a survey, University of Toronto Technical Report, 1999.

El Sakkout H. and Wallace M.G., 2000, Probe Backtrack Search for Minimal Perturbation in Dynamic Scheduling, in *Constraints: Special Issue on Industrial Constraint-Directed Scheduling*, Vol5 No4.

El Sakkout H., Richards T. and Wallace M., 1998, Minimal perturbation in dynamic scheduling, in *Proceedings of the 13th European conference on artificial intelligence (ECAI-98)*, Brighton, UK.

Myers, K.A. and S.F. Smith, "Issues in Integrating Planning and Scheduling for Enterprise Control", *DARPA Research Symposium on Advances in Enterprise Control*, San Diego, CA, Nov., 1999

Ow P.S., Smith S.F. and Thiriez A., 1988, Reactive plan revision, *Proceedings Seventh National conference on Artificial Intelligence (AAAI-88)*, St. Paul, MN, August.

Sadeh N., Otsuka S., and Schnelbach R., 1993, Predictive and reactive scheduling with the MicroBoss production scheduling and control system, in *IJCAI-93 workshop on knowledge-based production planning, scheduling and control*. Chambray, France.

Smith S.F., 1995, Reactive Scheduling Systems, in *Intelligent Scheduling Systems*, (eds. D. Brown and W. Scherer), Kluwer Publishing Co.