Embedding a Scheduler in Execution for a Planetary Rover

Wayne Chi, Steve Chien, Jagriti Agrawal, Gregg Rabideau, Edward Benowitz,
Daniel Gaines, Elyse Fosse, Stephen Kuhn, James Biehl
Jet Propulsion Laboratory
California Institute of Technology
4800 Oak Grove Drive
Pasadena, CA 91109
{firstname.lastname}@jpl.nasa.gov

Abstract

Scheduling often takes place in the context of execution. This reality drives several key design decisions: (1) when to invoke (re) scheduling, (2) what to do when the scheduler is running, and (3) how to use the schedule to execute scheduled activities. We define these design decisions *theoretically* in the context of the embedded scheduler and *practically* in the context of the design of an embedded scheduler for a planetary rover. We use the concept of a *commit window* to enable execution to use the previously generated schedule while (re) scheduling. We define the concepts of *fixed cadence*, *event driven*, and *hybrid* scheduling to control invocation of (re) scheduling. We define the concept of *flexible execution* to enable execution of the generated schedule to be adaptive within the response cycle of the scheduler. We present empirical results from both synthetic and planetary rover scheduling and execution model data that documents the effectiveness of these techniques at enabling the scheduler to take advantage of execution opportunities to complete activities earlier.

Scheduling in the Context of Execution

Scheduling is often performed in the context of execution. As such, for many realistic problems, scheduling and execution for a specific problem type are inextricably intertwined - aspects of execution affect scheduling, and aspects of scheduling affect execution. For example, in many scheduling problems, the activity model is imperfect and unable to predict execution aspects such as activity duration, resource usage, and activity success/failure. Therefore, rescheduling to incorporate execution feedback is a key part of the integrated scheduling and execution solution. This paper addresses the problem of embedding a scheduler in a specific class of execution environments. First we define a framework for analyzing trade-offs in scheduling and execution. Second, we discuss the resolution of these design choices in the context of a specific scheduler and execution environment under development for NASA's next planetary rover, the Mars 2020 (M2020) rover (Jet Propulsion Laboratory 2017a).

- *When to reschedule?* Because execution often differs from predicted scheduling models, the scheduler must reschedule to deal with activities completing early, completing late, or even failing. We explore fixed cadence and event-driven scheduling in which (re) scheduling occurs when actual execution differs from predicted execution by a prespecified threshold (e.g. an activity ends early by more than T seconds). Hybrid approaches combine these two in an effort to gain from the strengths of each approach.
- *Non-zero scheduling time.* Scheduling a set of activities on an actual CPU requires a potentially significant amount of time during which execution may or may not continue. This raises two issues: (a) what to execute in the time period before a new schedule is available; and (b) what potential is there to perform additional schedule manipulations within the time cycle of the scheduler (e.g. flexible execution or limited rescheduling).

To illustrate these concepts, consider the following example where activities A and B are scheduled consecutively due to use of a shared resource.

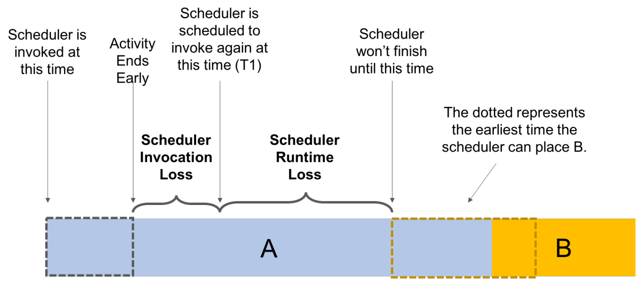

Scheduler is invoked at this time
Activity Ends Early
Scheduler is scheduled to invoke again at this time (T1)
Scheduler won't finish until this time
The dotted represents the earliest time the scheduler can place B.
Scheduler Invocation Loss
Scheduler Runtime Loss
A
B

Figure 1

Activity A ends early, and we wish to recoup this time within the schedule by executing B earlier. However, with a simple periodic scheduler invocation, the scheduler is not invoked until time T1 and then it requires some amount of time T_{sc} to actually reschedule. [1] This results in lost time that the system cannot recoup, which results in overall reduced efficiency. In this case, the lost time is the sum of *scheduler invocation loss* and *scheduler runtime loss*.

[1]Because of the difficulty in predicting scheduler runtime practically this is a conservative bound. See Discussion.

In general, in the remainder of this paper we discuss a number of techniques to address these challenges

- To mitigate *scheduler invocation loss* from fixed cadence scheduling, *event-driven scheduler invocation* invokes the scheduler in response to schedule expectation violations.
- To mitigate *scheduler runtime loss*, *flexible execution* adjusts activities to run earlier or later based on both explicitly defined precedence constraints and inferred precedences based on shared resource incompatibilities.
- To mitigate potential inconsistencies from *flexible execution* and *coarse resource modeling*, *hybrid event-driven scheduler invocation* invokes the scheduler for both scheduler expectation violations and fixed cadence offset prior invocations.

In the remainder of the paper we first define terminology to enable analysis of the general problem of embedding scheduling in execution. Then we describe a specific set of design assumptions relevant to our planetary rover application and analyze the effects of these decisions using both synthetic and actual rover pre-operational data.

For our defined scheduling problem (Rabideau and Benowitz 2017), the scheduler is given

- a list of activities $A_1 ... A_n$,
- where each activity can use or not use any number of unit resources (up to project limitations - 128 for M2020) $R_1 ... R_m$, and
- each activity can also use two consumable resources (a) energy and (b) data volume, and
- each activity is optionally constrained to a start time window $T_{i\_start} ... T_{i\_end}$, and
- also has a list of dependency constraints from $A_j \rightarrow A_k$ [2]

The charter of the scheduler is to produce a grounded time schedule that satisfies all of the above constraints.

We also make the following assumptions

1. Schedule activities form an approximately single serial path (e.g. critical path (Kelley Jr and Walker 1959))[3].
2. The prior schedule is executed while the scheduler is running (see below).
3. All activities fit in the initial schedule (the schedule is not oversubscribed).
4. Activities do not fail.
5. No preemption (activities are only preempted as a major failure case for M2020).

The goal is to schedule activities such that the schedule has the shortest possible makespan. We chose this because a shorter makespan implies that resources such as energy, data volume, and unit resources (time) are freed (see Future Work).

[2] $A_j \rightarrow A_k$ means the scheduled end time of A_k must be before the scheduled start time of A_j.

[3] Activities are rarely in parallel except for non interfering side activities.

What to Do During Scheduler Execution: The Commit Window

One challenge of a non-zero runtime scheduler is determining what to execute while the scheduler is running. A number of obvious alternatives are:

- Execute nothing - this has the downside of not achieving anything during the scheduler runtime. In addition, if the scheduler runtime is equal to the cadence of rescheduling then nothing will ever be done.
- Execute some default policy of activities - but how to define a useful default?
- Execute the previously generated schedule - we establish ”committed activities” that are committed to execution.

In the remainder of this paper we adopt the option of executing the previously generated schedule extending the use of a *commit window* approach (Chien et al. 2000; Knight et al. 2001) which was used to operate the Earth Observing One (Chien et al. 2005) and IPEX (Chien et al. 2016) spacecraft. However these prior usages of a commit window presumed an iterative repair, satisficing scheduler using a fixed offset of time close to execution as ”committed” whether the schedule is consistent (e.g. not in conflict) or not. The M2020 application requires a consistent schedule at the completion of each scheduler invocation and uses a prediction of scheduler runtime to predict scheduler completion time - thereby implying a commit window that ends up being a variable amount in the future. Additionally, the committed activities may be in conflict with higher value activities that, if we had not committed an activity (especially a long duration one), we now cannot switch to.

Property 1: Any activity whose scheduled start time is after the start of the commit window and before the end of the commit window must be committed to execution and cannot be rescheduled by the scheduler.

Property 2: The scheduler cannot schedule activities to start in the commit window.

For simplicity, assume that the scheduler runtime, T_{sc}, is predictable and constant, and fix the end of the commit window to now + T_{sc} when the scheduler is invoked.

T_{sc} via the commit window limits rescheduling in three ways.

First, the scheduler must wait until it can be run. This is the *scheduler runtime loss*.

Second, it is challenging for the scheduler to incorporate execution updates while running. Therefore typically the scheduler is invoked with the execution state at invocation time and the execution status is not updated during scheduling. Therefore its data is stale by up to T_{sc}.

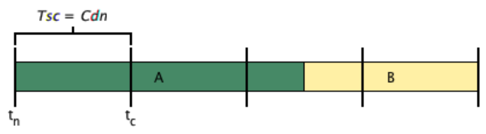
Third, the scheduler does not schedule activities in the commit window - inherently this limits the earliest start time of the scheduler control.

These restrictions limit scheduler possible gain.

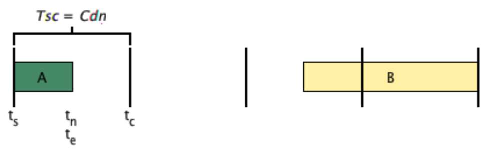
Fixed Cadence Scheduling

A simple approach to rescheduling is Fixed Cadence Scheduling where the scheduler is invoked at a cadence of every Cdn.

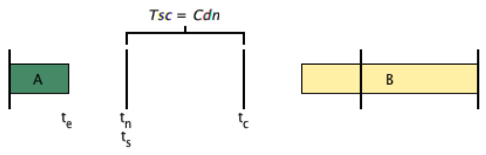
Typically, $Cdn \geq T_{sc}$ otherwise we would not have activities from the prior invocation for the current scheduler invocation. To maximize responsiveness in Fixed Cadence Scheduling, $Cdn = T_{sc}$. Fixed Cadence Scheduling is simple to implement and makes invoking the scheduler predictable.



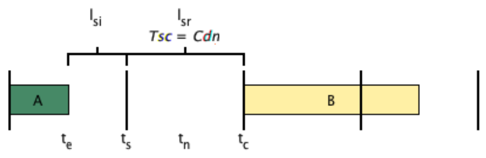
(a) The scheduler is invoked at now (t_n) and predicted to end at t_c.



(b) A finishes early at t_e, but the scheduler doesn't ingest this information since it's already scheduling.



(c) The scheduler is reinvoked at t_s and the commit window shifts.



(d) The scheduler moves B up to t_c, but is unable to move it further. There is a scheduler invocation loss (l_{si}) and a scheduler runtime loss (l_{sr}).

Figure 2: The challenge with Fixed Cadence Scheduling

As an example, Figure 2a shows two back to back activities, A and B, that cannot run in parallel due to shared resources. The vertical lines represent when the scheduler is scheduled to run. Activity A is currently executing. In Figure 2b, activity A finishes early. Our goal is to gain back the time between the time A finishes and B starts. Unfortunately, the scheduler must wait until the next time it has been predetermined to invoke (2c). In addition, once the scheduler is reinvoked it can only schedule activities to start no earlier than the end of the commit window (2d). The losses from such restrictions are represented as follows.

$$l_{total} = l_{si} + l_{sr} \tag{1}$$

l_{total} is the total loss, l_{si} is the loss from waiting for the scheduler to be invoked (*scheduler invocation loss*), and l_{sr} is the loss from not being able to schedule activities inside the commit window (*scheduler runtime loss*).

Another perspective is to observe the specific points in time that constitute l_{si} and l_{sr}. l_{si} can be redefined as the delay between an event and the start of the next scheduling cycle, t_s.

$$l_{si} = t_s - t_e \tag{2}$$

l_{sr} is the loss from t_s to the end of the commit window, t_c.

$$l_{sr} = t_c - t_s \tag{3}$$

Knowing that $l_{si} = t_s - t_e$ and $l_{sr} = t_c - t_s$, we can derive the following.

$$l_{total} = (t_s - t_e) + (t_c - t_s) \tag{4}$$

Here, l is the loss, t_s is the time where the scheduler is reinvoked, t_e is the time of an event (scheduler expectation violation) is the end of commit window.

(4) can be further simplified to

$$l_{total} = (t_c - t_e) \tag{5}$$

This loss can be significant as it can occur every time there is an event where the scheduler can recoup resources.

The overall gain of the Fixed Cadence Scheduling assuming serial activities and that activities do not run late is

$$G_{fc} = \sum_{i=1}^{n} f(activity_i) \tag{6}$$

$$f(activity_i) = \begin{cases} (s_i - a_i) - l_{total} & \text{if } (s_i - a_i) - l_{total} > 0 \\ 0 & \text{otherwise} \end{cases} \tag{7}$$

where s_i and a_i are the scheduled end time and actual end time of $activity_i$ respectively.

A drawback of Fixed Cadence scheduling is that to maximize responsiveness, $Cdn = T_{sc}$ which can consume significant CPU resources (critical on a planetary rover).

Event Driven Scheduling

Rather than invoking the scheduler at a fixed cadence, Event Driven Scheduling invokes the scheduler immediately when a change or "event" occurs. This provides two benefits.

First, if an event occurs when the scheduler is idle, because the scheduler can be invoked immediately, l_{si} is eliminated when an activity ends early.

$$l_{total} = l_{sr} \tag{8}$$

Second, Event Driven Scheduling may result in fewer scheduler invocations than Fixed Cadence Scheduling, freeing valuable CPU time. Furthermore if the scheduler runs less often, it can potentially run at a higher task priority thereby reducing T_{sc}.

Using Event Driven Scheduling with serial activities and no late completing activities, the makespan gain is

$$G_{ed} = \sum_{i=1}^{n} g(activity_i) \tag{9}$$

$$g(activity_i) = \begin{cases} (s_i - a_i) - l_{sr} & \text{if } activity_i \text{ triggers an event} \\ f(activity_i) & \text{otherwise} \end{cases} \tag{10}$$

where s_i and a_i are the scheduled end time and actual end time of $activity_i$ respectively, and $g(activity_i)$ is the gain from Event Driven Scheduling.

With serial activities, and infrequent events, Event Driven Scheduling performs at least as well as Fixed Cadence Scheduling and can potentially outperform it.

$$l_{si} \in R_{\geq 0} \tag{11}$$

we can deduce from (1) that

$$l_{sr} \leq l \tag{12}$$

Thus,

$$G_{ed} \geq G_{fc} \tag{13}$$

The Challenge of Events During Scheduling

During an event triggered scheduling, another event can occur while the scheduler is running (see Figure 3).

In this situation, the scheduler has three possible choices.

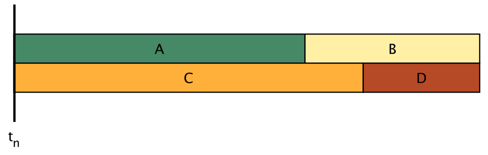
- Scheduler ignores the event. (Downside - lose the gain from early completion)
- The scheduler does not stop to reinvoke, but immediately reinvokes once the current scheduling cycle completes. (Downsides: (a) the remainder of this cycle T_{sc} is Scheduler invocation loss for this event which is not handled until the next scheduling cycle and (b) two invocations mean more CPU time spent scheduling).
- The scheduler stops and reinvokes. If we have an anytime scheduler we can use the partial schedule at the cost of suboptimality. If not the expended T_{sc} becomes invocation loss for the first event and time spent scheduling is wasted.

If the scheduler does not stop to reinvoke and ignores the event then the scheduler will not gain value from the event until the next scheduler invocation (potential large loss).

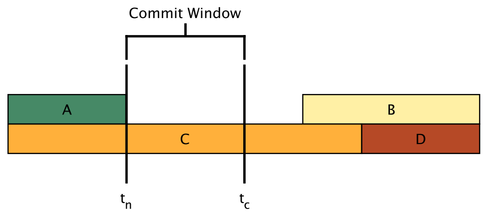
Decision-theoretic methods can address the above trades (Horvitz, Breese, and Henrion 1988) but often accurate marginal utility information is not available.

Event Threshold

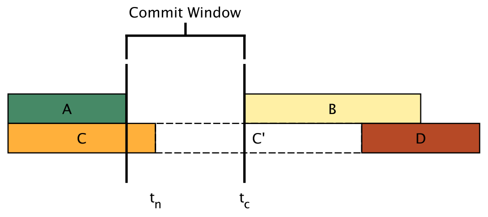
An event threshold can lower the frequency of events by suppressing less important changes. Because we focus on makespan we consider events where an activity completes earlier than its scheduled end time by greater than a fixed threshold. If the seriality assumption holds, generally speaking an activity completing early enables makespan reduction. Generally, any resource under-run (including time) or over-run (activities not completing, energy, or data volume) are all valid events (see Discussion).



(a) A shares a resource with B. C shares a resource with D.



(b) A finishes early and triggers Event Driven Scheduling.



(c) C finishes early, but the scheduler is already running and does not ingest the information. It perceives C + C' and can't move D earlier until the next scheduler invocation.

Figure 3: The challenge of events during scheduling

Hybrid Fixed Cadence - Event Driven Scheduling

Event Driven Scheduling improves on Fixed Cadence Scheduling by eliminating scheduler invocation loss. However, Event Driven Scheduling has the dual drawbacks of (a) frequent events triggering scheduler invocations can starve lower priority tasks and (b) infrequent events can cause potentially lengthy gaps between scheduler invocations causing schedule disconnect from resource state. Hybrid scheduling uses events, t_e, but adds a minimum and maximum cadence to prevent these potential issues. Hybrid scheduling also can help to keep the scheduler synchronized with flexible execution as described below.

Flexible Execution: Responding to execution updates within T_{sc}

All of the above methods for rescheduling operate at a responsiveness limited by T_{sc}. Flexible Execution (FE) solves this problem by providing the ability to modify execution of activities in the time interval from Now to Now + T_{sc}.

FE is a process separate from the scheduler and runs at a higher cadence with the execution system. It accomplishes this by only considering a subset of the scheduling constraints and options considered by the scheduling algorithm.

We first define a predecessor-successor relationship between activities.

$$((sharedResource(A,B) \wedge\ A_{start} < B_{start}) \\ \vee\ dependent(A,B)) \rightarrow predecessor(A,B) \tag{14}$$

$$predecessor(A,B) \rightarrow successor(B,A) \tag{15}$$

The predecessor-successor relationship is one of relative ordering between activities in the schedule that share the same unit resources or are dependent on/establish precedence between one another. [4]

FE allows execution according to a directed acyclic graph based on predecessor-successor relationships of committed activities. Whenever an activity has no uncompleted predecessors in the graph, FE allows execution of the activity.

Algorithm 1 Flexible Execution

```
Input:
    $G$: Directed acyclic graph of committed activities up to the
    current time (empty if at the start of the plan).
    $A$: List of all activities sorted by start time
Output:
    $L$: Set of activities to be started [5]
 1: $L \leftarrow \emptyset$
 2: repeat
 3:     for each $a \in A$ do
 4:         if $a$ is committed and $a \notin G$ then
 5:             add $a$ to $G$
 6:             for each $n \in G$ do
 7:                 if $n \in a.predecessors$ then
 8:                     add $n$ to $a.children$
 9:                 end if
10:             end for
11:             if $a.children = \emptyset$ then
12:                 add $a$ to $L$
13:             end if
14:         end if
15:         if $a.finished$ then
16:             $G.pop(a)$
17:             for each $n \in G$ do
18:                 if $n.children = \emptyset$ then
19:                     add $a$ to $L$
20:                 end if
21:             end for
22:         end if
23:     end for
24: until $A$ is empty
```

FE therefore enables activities to execute at times different from when scheduled, but in the same relative ordering in the schedule for activities using shared unit resources. When an activity finishes early, its successors can be executed at an earlier time than when they were scheduled. When an activity runs later than expected its successors are postponed. Instead of rescheduling, FE executes successor activities whenever their predecessor activities finish.

Flexible Execution and Scheduling

If FE only knows about activities that are committed, it does not need to keep track of scheduled start times since it considers executing an activity when all of its predecessors are finished. However, the scheduler still needs to know about scheduled start times to predict when activities will end and schedule around them. Rather than updating scheduled start times every second, FE need only update scheduled start times when the scheduler is invoked.

While we know the exact time an activity ends early, we cannot predict when an activity, A, that is running late will finish and cannot predict when the next activity, B, can start. However, the scheduler needs a scheduled start time for all committed activities to predict activity end times. A has not finished executing, but the old scheduled start time for B is no longer accurate. Thus, B's scheduled start time must be updated to the current time for accuracy.

When updating start times, there might be a chain of multiple committed activities. It is important to note that if an activity in the chain gets delayed or finishes early, the whole chain should be "ripple-updated" so that all activities in the chain are updated. If not, then the scheduler may have incorrect information on predicted activity start times.

Caveats and Drawbacks of Flexible Execution

Although Flexible Execution provides value in incorporating execution feedback within the timescale of the scheduler runtime T_{sc}, there are several drawbacks to FE as proposed.

- Non unit resources such as power/energy, thermal, and data volume are ignored. For our applications this is an acceptable compromise in order to avoid the more expensive modeling of these resources as these resources are total sum not peak value. However, this may not be true for other applications.
- If FE is allowed to run concurrently while the scheduler is scheduling, it can allow execution to become inconsistent with the schedulers knowledge (e.g. activities start and end times may differ in the commit window) resulting in the scheduler generating a suboptimal or even invalid schedule due to stale data (although flexible execution can mitigate these inconsistencies).
- FE tries to optimize activities locally, but does not consider the full scope of the plan. As a result, it may modify the plan in a way that would be suboptimal compared to the scheduler.
- In order to maximally improve performance, FE may need to modify uncommitted activities. If an activity ends early by more than T_{sc}, FE cannot begin execution of successor activities. Even after event driven rescheduling, T_{sc} time will be lost. An alternate is to allow uncommitted

[4]Prior work exploits similar relationships (Muscettola 2002; Policella et al. 2004; 2009). We do not explore these approaches due to M2020 project concerns on complexity for V&V and strict timing requirements (1 Hz response).

[5]If no constraints (e.g. execution time constraints) are violated.

activities which are scheduled to start further in the future to begin execution when their predecessors complete. However, this would prevent the rescheduling of alternate activities in conflict with the activity FE has brought forward. In short, FE only considers activities already in the schedule, in the same relative order. The scheduler can consider different activities and orders, which may be preferable.

- Flexible activity chains may push activities out of the commit window. In updating the start times of activities before running the scheduler, activities can be pushed later due to late running activities. In the extreme case an activity can be pushed beyond the commit window, enabling the scheduler to potentially not schedule it in the current scheduler invocation.

Empirical Evaluation

In order to test our model of scheduling and execution we have implemented a simulation of execution and rescheduling used to test the accuracy of our model using both synthetic data and data from *sol types*. *Sol types* are the currently best available data on expected Mars 2020 rover operations (Jet Propulsion Laboratory 2017a). In order to feed this simulation, we use the *M2020 surrogate scheduler* - an implementation of the same algorithm as the M2020 onboard scheduler (Rabideau and Benowitz 2017), but implemented for a linux workstation environment - as such, it is expected to produce the same schedules as the operational scheduler but runs much faster in a workstation environment. Indeed, current plans are to use the surrogate scheduler both to assist in validating the flight scheduler implementation and also in ground operations for the mission.

The Mars 2020 scheduler algorithm is a greedy, priority-first, non-backtracking scheduler. It loops through requested activities in priority first order, using timeline-based modeling common in space applications (e.g. (Chien et al. 2012)) to compute valid activity start times considering activity state and resource needs. The major complication is that each activity placement may require (a) preheat scheduling (to warm up mechanisms on the rover prior to usage) and (b) power management (managing the waking and sleeping of the rover to to avoid battery undercharge (endangering the vehicle) or overcharge (detrimental to battery performance). Because each iteration for activity placement leaves the schedule in a consistent state, it is theoretically possible to use the scheduler as an anytime algorithm with some additional algorithmic complexity to checkpoint copies of the schedule. Also the M2020 processor (Rad 750) is 100 times slower than a laptop and the scheduler only receives a fraction of the processor.

We have developed a ground-based model that simulates rover operations for a single ground - flight operations cycle (usually one Martian day or a "Sol"). Each input plan contains typically 40 activities. We utilize a probabilistic execution model based on operations data from the Mars Science Laboratory Mission (Jet Propulsion Laboratory 2017b; Gaines et al. 2016a; 2016b) in order to plausibly simulate activities completing early. As described earlier in assumptions, this model assumes that the scheduling model activity durations given are conservative, that the original requested activities are feasible, and that the overall quality metric is to minimize makespan during execution. Note that we do not explicitly change activity consumables as as energy and data volume are generally modelled as rates so activities completing early decreases energy and data volume usage.

We measured the effectiveness of rescheduling by calculating the reduction in makespan, which is the difference between the makespan for the initial schedule (using all of the conservative scheduling model activity durations) and the makespan for the executed schedule. This *rescheduling enabled* schedule can be compared against the theoretical minimum makespan schedule (simulation with T_{sc} = 0).

We compare combinations of the scheduling methods in our framework - the case where $Cdn = 2 * T_{sc}$ and with and without and Event Driven Scheduling and Flexible Execution. For runs using Event Driven Scheduling the scheduler is triggered when an activity ends early by at least T_{sc}.

Because the Mars 2020 Mission Sol Types (a) are not fully serial and (b) contain other constraints such as execution time constraints, we first evaluate synthetic schedules based on the Sol Types. Each synthetic schedule is a serial path of X activities that share the same unit resource and no other constraints. The predicted (conservative) duration of the activities is generated from a normal distribution with $\mu = 1125$ seconds and $\sigma = 1852$ seconds based on the durations from activities in the M2020 Sol Types.

Synthetic data enforces seriality and avoids dependency and time window constraints. Therefore, achieved makespan gain + model calculated Scheduler Invocation Loss + Scheduler Runtime Loss = theoretical max makespan gain.

In contrast Figures (5a and 5b) use sol type inputs derived from schedules that are expected to run on the M2020 mission. The average duration of activities throughout all schedules is 15 minutes and 8 seconds and the standard deviation is 27 minutes and 27 seconds. The probabilistic model used to generate realistic activity durations compared to the original conservative ones resulted in activities ending on average about 32 percent early. In these runs the model predicted loss only loosely correlates with actual loss due to schedule parallelism, execution time constraints, and activity dependencies/setups.

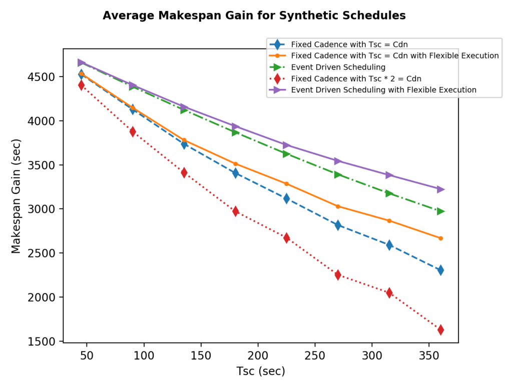
By studying makespan gain as a function of the commit window for various scheduling methods on both synthetic and sol type data we draw several conclusions.

Event driven rescheduling has the greatest impact on performance by reducing scheduler invocation loss.

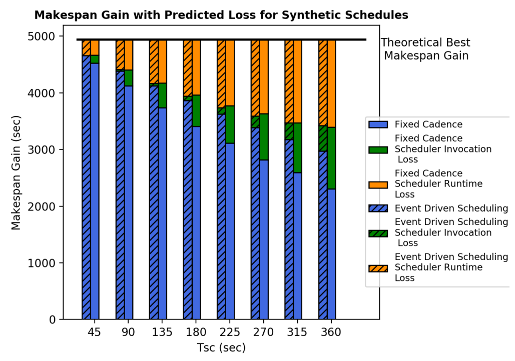
As T_{sc} approaches zero (to the left of the graphs) the scheduler runtime loss decreases and the makespan gain increases as a direct consequence.

Flexible Execution is more effective when T_{sc} is larger. When T_{sc} is smaller, scheduling can recover the time from early activity completion, losing less time to scheduler runtime loss. When T_{sc} is larger, FE is needed to recover this time. This is true for both Event driven and fixed cadence rescheduling, but in the case of fixed cadence FE can also mitigate the scheduler invocation loss.

Individually, Event Driven Scheduling is more effec-



(a) Makespan gain for varying scheduler T_{sc} and methods averaged across synthetic schedules.



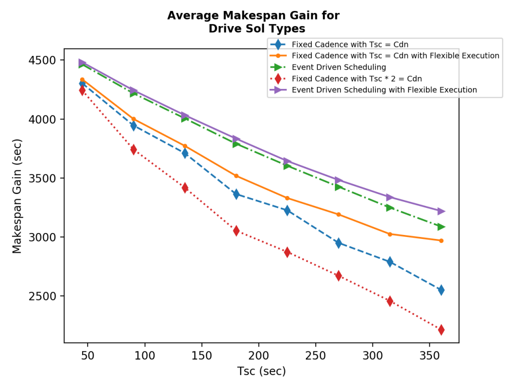
(b) Synthetic schedules are entirely serial: Makespan gain + predicted loss = theoretical best makespan gain.

Figure 4: Synthetic Data: Makespan gain over 30 runs for varying scheduler T_{sc} and methods. Predicted Scheduler Invocation Loss and Scheduler Runtime Loss are stacked to show predicted total loss compared to theoretical max gain.

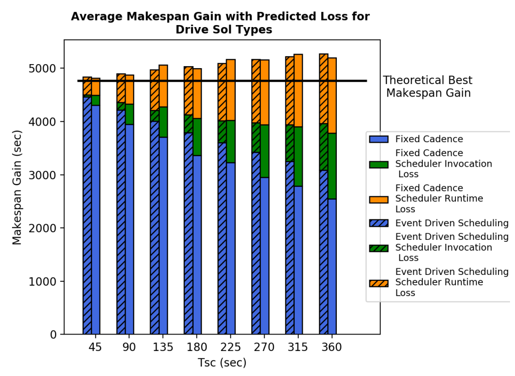
tive than Flexible Execution for our observed performance regime. Although both Flexible Execution and Event Driven Scheduling consistently result in a higher makespan gain than just Fixed Cadence Scheduling, the makespan gain with Event Driven scheduling is larger than that with Flexible Execution. This is because most activities decreased by an amount greater than T_{sc}. This allows Event Driven Scheduling to trigger events and reschedule activities as early as possible and limits Flexible Execution since activities infrequently end early while the scheduler is running [6].

The computational model scheduler invocation loss and scheduler runtime loss is somewhat inaccurate on the M2020 sol type data for several reasons. First, the sol types have some parallelism causing the computational model to double count loss or count loss that does not affect makespan and therefore overestimate. Second, in some cases an activity completes early, but a following activity cannot be moved forward because it requires a setup or preheat activ-

[6]This only holds true because the scheduler runtime is equal to the commit window size.



(a) Makespan gain for varying scheduler T_{sc} and methods averaged across range of M2020 Sol Types.



(b) Makespan gain + predicted loss $\neq$ theoretical best makespan gain (if we had an instantaneous scheduler). Real schedules are not all perfectly serial and have other execution constraints (e.g. preheats).

Figure 5: Sol Type Data: Makespan gain over 30 runs for varying scheduler T_{sc} and methods. Predicted Scheduler Invocation Loss and Scheduler Runtime Loss are stacked to show predicted total loss compared to theoretical max gain.

ity which has already started and cannot move earlier (even optimal scheduler cannot recoup). Third, the computational loss model does not account for activities that cannot be brought forward even when earlier activities complete early due to ”no earlier than” constraints. In these cases the computational loss model calculates a loss that even the optimal rescheduler cannot recoup. [7]

Related Work

The Remote Agent Experiment (Muscettola et al. 1998; Pell et al. 1997) used a batch scheduler onboard the Deep Space One Spacecraft in 1999 for approximately 48 hours. The scheduler was invoked on a fixed periodic cadence and generated a flexible temporal schedule.

IDEA (Gregory et al. 2002) uses a hierarchy of time-bounded hybrid execution/planning agents to meet both de-

[7]More information can be found in Additional Materials (Chi et al. 2018).

liberative and execution needs. As an architecture, it does not dictate scheduler execution interaction as these would be implemented within a single IDEA agent.

CASPER (Chien et al. 2000; Knight et al. 2001) is a continuous, iterative repair scheduler that takes very small CPU timeslices (seconds to minutes) but does not guarantee leaving the schedule in a consistent state. Any inconsistencies within a static duration commit window are the responsibility of execution. CASPER flew onboard the Earth Observing One spacecraft, controlling it for over a dozen years (Chien et al. 2005; Tran et al. 2004). CASPER also flew on and controlled the IPEX mission for over 1 year (Chien et al. 2016). OASIS is a research prototype for rover autonomy using CASPER for planning and scheduling and TDL for execution with a similar fixed commit window strategy as described above (Castano et al. 2007).

Woods et al. 2009 (Woods et al. 2009) describe a research prototype rover with replanning (TVCR) that uses a plan fragment based repair approach but does not include details on integration of execution with (re) scheduling such as what to execute during rescheduling. The general problem of embedding a scheduler within execution shares aspects of continuous planning and execution (Myers 1999).

Discussion and Future Work

This work represents an exploration into the overlap between scheduling and execution within the context of batch rescheduling and execution with a focus on a planetary rover applications. There are many topics for future study - we describe several below.

Further exploration of flexible execution concepts and their relationship to scheduling and more complex resources (energy, time, and data volume for our problem) is of interest. Extending this work to more concurrent schedules would increase usefulness of techniques and analysis.

Our formulation presumes that the schedule objective function is to minimize makespan. In reality, the goal is to maximize some utility function over executed activities. For our planetary rover scheduling problem this means accepting an oversubscribed list of activity requests with the charter of rescheduling to get more high value activities executed as resources become available due to activities completing early and freed energy and data volume. In some cases there is a range of activities in a preference order and the ideal scheduling and execution system would execute the best possible within resources. For example, the scientists might request minimally a 2x2 mosaic of images, but prefer a 3x3 or 3x4 if time, energy, and data volume permit.

Additionally, we presume that earlier is better for activity execution. More realistically, activities might have a preferred start time or offset from other activities and preserving/optimizing these relationships might be more important than executing earlier or even executing more. Finally, conserving some resources might also be an optimization metric (e.g. conserving power or data volume). Representing this criteria in scheduling and execution would be ideal.

In our formulation, scheduling is a batch process initiated with inputs available at the time of initiation. Any execution information that arrives after the start of the scheduling process is not incorporated into the ongoing schedule generation. How to incorporate execution information acquired during the scheduler runtime into the scheduling algorithm could be very valuable and improve the relevance and quality of the generated schedule. Scheduler Runtime: In reality scheduler runtime is not easily predictable. Ongoing work is building a model of scheduler runtime from inputs (e.g. number of activity requests). We run the scheduler over 50 full Sol simulations to profile the scheduler's runtime against varying input activities and scale surrogate runtime for an estimate of the actual onboard planner (assuming full onboard CPU utilization) (See Figure 6).

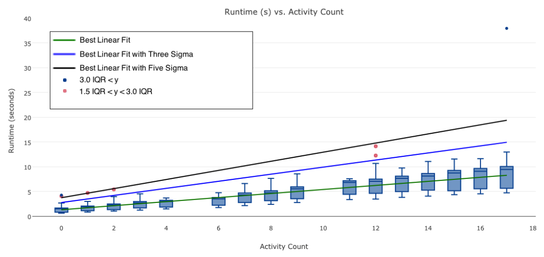


Figure 6: Scheduler Runtime vs Activity Count. The green, blue, and black lines represent the linear fit, the three sigma linear fit, and the five sigma linear fit respectively. The blue outliers are over 3.0 IQR (the difference between the 75th and 25th percentile a.k.a. interquartile range) away from the closest whisker and the pink outliers are over 1.5 IQR but less than 3.0 IQR away from the closest whisker.

As this data indicates - predicting scheduler runtime is not trivial. Furthermore, correctly handling when the scheduler completes early (and Flexible Execution can shift activities forward) or, more challenging, when the scheduler completes late is an area of future work.

Conclusion

We have presented a theoretical framework for embedding a scheduler within the context of execution - highlighting several key design decisions required to embed the scheduler in execution.

When to invoke rescheduling - we have discussed the pros and cons of several approaches to invoking scheduling - specifically fixed cadence rescheduling, event-driven rescheduling and hybrid rescheduling.

What to execute while rescheduling - we have described three options for what to execute while (re) scheduling - nothing, a default policy, and the previous schedule.

How to structure execution to allow for limited response within the T_{sc} scheduler response time - we describe flexible execution to adjust execution for earlier and later based on precedence analysis.

We have then shown both analytical and empirical analysis of both synthetic and planetary rover model data to quantify the effects of these design options on schedule execution utility as measured by schedule makespan.

Acknowledgments

This work was performed at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration.

References

Castano, R.; Estlin, T.; Anderson, R. C.; Gaines, D. M.; Castano, A.; Bornstein, B.; Chouinard, C.; and Judd, M. 2007. Oasis: Onboard autonomous science investigation system for opportunistic rover science. *Journal of Field Robotics* 24(5):379–397.

Chi, W.; Chien, S.; Agrawal, J.; Rabideau, G.; Benowitz, E.; Gaines, D.; Fosse, E.; Kuhn, S.; and Biehl, J. 2018. Embedding a scheduler in execution for a planetary rover: Additional materials. Technical report, Technical Report D-101730, Jet Propulsion Laboratory.

Chien, S. A.; Knight, R.; Stechert, A.; Sherwood, R.; and Rabideau, G. 2000. Using iterative repair to improve the responsiveness of planning and scheduling. In *Artificial Intelligence Planning and Scheduling*, 300–307.

Chien, S. A.; Sherwood, R.; Tran, D.; Cichy, B.; Rabideau, G.; Castano, R.; Davies, A.; Mandl, D.; Trout, B.; Shulman, S.; et al. 2005. Using autonomy flight software to improve science return on earth observing one. *Journal of Aerospace Computing Information and Communication* 2(4):196–216.

Chien, S.; Johnston, M.; Frank, J.; Giuliano, M.; Kavelaars, A.; Lenzen, C.; Policella, N.; and Verfailie, G. 2012. A generalized timeline representation, services, and interface for automating space mission operations. In *12th International Conference on Space Operations (SpaceOps 2012), Stockholm, Sweden*, 11–15.

Chien, S.; Doubleday, J.; Thompson, D. R.; Wagstaff, K. L.; Bellardo, J.; Francis, C.; Baumgarten, E.; Williams, A.; Yee, E.; Stanton, E.; et al. 2016. Onboard autonomy on the intelligent payload experiment cubesat mission. *Journal of Aerospace Information Systems*.

Gaines, D.; Anderson, R.; Doran, G.; Huffman, W.; Justice, H.; Mackey, R.; Rabideau, G.; Vasavada, A.; Verma, V.; Estlin, T.; et al. 2016a. Productivity challenges for mars rover operations. In *Proceedings of 4th Workshop on Planning and Robotics (PlanRob)*, 115–125. London, UK.

Gaines, D.; Doran, G.; Justice, H.; Rabideau, G.; Schaffer, S.; Verma, V.; Wagstaff, K.; Vasavada, A.; Huffman, W.; Anderson, R.; et al. 2016b. Productivity challenges for mars rover operations: A case study of mars science laboratory operations. Technical report, Technical Report D-97908, Jet Propulsion Laboratory.

Gregory, N. M.; Dorais, G. A.; Fry, C.; Levinson, R.; and Plaunt, C. 2002. Idea: Planning at the core of autonomous reactive agents. In *Proceedings of the 3rd International Workshop on Planning and Scheduling for Space*. Citeseer.

Horvitz, E. J.; Breese, J. S.; and Henrion, M. 1988. Decision theory in expert systems and artificial intelligence. *International journal of approximate reasoning* 2(3):247–302.

Jet Propulsion Laboratory. 2017a. Mars 2020 rover mission https://mars.nasa.gov/mars2020/ retrieved 2017-11-13.

Jet Propulsion Laboratory. 2017b. Mars science laboratory mission https://mars.nasa.gov/msl/ 2017-11-13.

Kelley Jr, J. E., and Walker, M. R. 1959. Critical-path planning and scheduling. In *Papers presented at the December 1-3, 1959, eastern joint IRE-AIEE-ACM computer conference*, 160–173. ACM.

Knight, S.; Rabideau, G.; Chien, S.; Engelhardt, B.; and Sherwood, R. 2001. Casper: Space exploration through continuous planning. *IEEE Intelligent Systems* 16(5):70–75.

Muscettola, N.; Nayak, P. P.; Pell, B.; and Williams, B. C. 1998. Remote agent: To boldly go where no ai system has gone before. *Artificial Intelligence* 103(1-2):5–47.

Muscettola, N. 2002. Computing the envelope for stepwise-constant resource allocations. In *International Conference on Principles and Practice of Constraint Programming*, 139–154. Springer.

Myers, K. L. 1999. Cpef: A continuous planning and execution framework. *AI Magazine* 20(4):63.

Pell, B.; Gat, E.; Keesing, R.; Muscettola, N.; and Smith, B. 1997. Robust periodic planning and execution for autonomous spacecraft. In *International Joint Conference on Artificial Intelligence*, 1234–1239.

Policella, N.; Smith, S. F.; Cesta, A.; and Oddi, A. 2004. Generating robust schedules through temporal flexibility. In *ICAPS*, volume 4, 209–218.

Policella, N.; Cesta, A.; Oddi, A.; and Smith, S. F. 2009. Solve-and-robustify. *Journal of Scheduling* 12(3):299.

Rabideau, G., and Benowitz, E. 2017. Prototyping an onboard scheduler for the mars 2020 rover. In *International Workshop on Planning and Scheduling for Space*.

Tran, D.; Chien, S.; Rabideau, G.; and Cichy, B. 2004. Flight software issues in onboard automated planning: Lessons learned on eo-1. In *International Workshop on Planning and Scheduling for Space*.

Woods, M.; Shaw, A.; Barnes, D.; Price, D.; Long, D.; and Pullan, D. 2009. Autonomous science for an exomars rover–like mission. *Journal of Field Robotics* 26(4):358–390.