Using a Model of Scheduler Runtime to Improve the Effectiveness of Scheduling Embedded in Execution

Abstract

Scheduling often interacts with execution. When the scheduler is developing a schedule, real time (execution) proceeds. Usually a scheduler cannot modify portions of the schedule expected to start execution prior to the scheduler's expected completion. In deployed systems, often little effort is spent on predicting scheduler runtime and instead an extremely conservative, simple model is used, resulting in loss of performance as less of the schedule can be updated.

We develop predictive model(s) of scheduler runtime and use these models to improve scheduler and execution performance. We present several models of scheduler runtime based on a scheduler being deployed onboard NASA's next Mars rover, the M2020 rover Perseverance. The models consider algorithmic complexity, characteristics of the input plan, and prior runtime data. First, we show how these still relatively unsophisticated models can more accurately predict scheduler runtime compared to the static conservative baseline being used for the actual M2020 onboard scheduler. Second, we show how the more accurate scheduler runtime models' tighter (shorter) runtime predictions enable better scheduler performance as measured by makespan and percentage of activities executed. Finally, we discuss a number of future steps to further advance this line of work.

Background

The M2020 onboard scheduler (Rabideau and Benowitz 2017) is intended to handle execution variations throughout a sol (Martian day) by changing the plan as needed to react to deviations from the predicted conditions. Because the rover has extremely limited CPU resources [1], the scheduling and execution algorithms are relatively simple. However (re)scheduling takes a significant amount of time; it is conservatively allocated 60 seconds onboard the rover. As the scheduler is expected to be invoked approximately 15 times per sol this amount of time can significantly impact rover efficiency. A typical martian sol has about 5-7 hours of productive time between receiving inputs from the ground and end of execution, of which half might be spent sleeping, so usable time is close to 183 minutes. Thus, 15 minutes spent rescheduling would use a non-negligible 8% of productive time of a multi billion dollar space mission. Therefore mitigating the effect of the rescheduling time on rover productivity is important to the mission. Previous work has described the M2020 onboard scheduling algorithms (Rabideau and Benowitz 2017; Chi, Chien, and Agrawal 2019; Agrawal et al. 2019) as well as the integration of scheduling into execution (Chi et al. 2018). We build upon this prior work to study gains from a more informed model of scheduler runtime.

Rover Schedules

We adopt the schedule representation previously described in (Rabideau and Benowitz 2017). In this formulation the scheduler is presented with a list of activities $A_1, ...A_n$ ordered by priority. Activity A_i has a predicted duration and a set of constraints including unit resources, dependencies (activities that must complete before A_i begins), and execution time windows. There are two unusual aspects of this scheduling problem. First, activities may require thermal activities (preheats and maintenance) to be scheduled before and concurrently with the activity. Second, the scheduler must manage the wake/sleep schedule for the rover where most activities require the rover to be awake during their execution, a small number require that the rover must be asleep during their execution, and that the rover manages energy by trying to sleep as much as possible to conserve energy (Chi, S.Chien, and Agrawal 2020).

However the research in this paper applies to a wide range of schedulers and scheduling problems. Specifically we address the situation where: (1) the time required to reschedule is significant compared to the overall productive schedule time; (2) it is possible to (with some accuracy) predict the scheduler runtime from data available prior to invocation; (3) imperfect runtime predictions can be handled; and (4) more precise runtime predictions can be leveraged by the scheduler to produce better schedules.

In this paper we use two different types of plans: synthetic plans and sol types. Both are valid inputs to the scheduler, but synthetic plans are generated to demonstrate a specific

Copyright © 2020, All rights reserved.

[1] The RAD750 processor used by the Mars 2020 rover has measured performance in the 200-300 MIPS range. In comparison a 2016 Intel Core i7 measured over 300,000 MIPS or over 1000 times faster). Furthermore, the onboard scheduler is only allocated a portion of the computing cycles onboard the RAD750. Therefore a typical laptop has several thousand times more compute power than allocated to the M2020 onboard scheduler

behavior, while sol types are designed to approximate plans the M2020 project will use during operations. Synthetic plans' activities have unit resource constraints so that activities cannot occur at the same time, but activities have no other constraints unless explicitly mentioned. Meanwhile, sol types are more complex and follow certain patterns; they are also designed such that all activities should execute if starting at a high state of charge (SOC) energy level.

Schedule Execution

We presume that both flexible execution and rescheduling are used to address activities taking more or less time than their expected duration to complete. If an activity completes earlier that the scheduled completion time by greater than a certain threshold, it is said that an *event* has occurred, and rescheduling is triggered. Likewise if an activity completes later than its scheduled completion time, an *event* is also declared and rescheduling is triggered. [2]

An important design decision is what to execute while the scheduler is rescheduling (and the corresponding decision of what the scheduler should assume executed while it is rescheduling). For this we use the concept of a *commit window* (discussed later).

When there are smaller variations in execution than warrant rescheduling, we assume some form of *flexible execution* (FE) can modify the schedule (Agrawal, Chi, and Chien 2019). FE uses a directed acyclic graph of activities within a predefined temporal window to capture their relative ordering. Then if an activity ends early or late, FE can pull or push activities, respectively, within this window. FE respects the execution windows of activities and required states but does not fully model more complex resources (such as energy) and presumes that minor variations in timing will not significantly affect such resources.

When there are larger variations in execution, then rescheduling is required to generate a new schedule to account for the changed context.

In order to model uncertain actual activity execution durations, we use data from the Mars Science Laboratory rover operations (Gaines et al. 2016a; 2016b) to vary the durations of activities according to normal distributions with a mean equal to approximately 70% of the activity's predicted duration and variance in order to achieve a pre-designated proportion of activities taking longer than their predicted duration.

Scheduler and Execution Performance Metrics

We evaluate schedule performance using two metrics - the number of activities executed and schedule makespan. The number of activities executed is simply the number of the activities specified in the problem that are successfully scheduled and then executed within their constraints (state, execution window, etc.).

The second schedule execution metric is makespan. The makespan of a schedule is the difference between the start of the earliest activity in the schedule and the end of the latest activity. We slightly modify this calculation to ignore certain activities that are fixed in time, such as communications passes. We use makespan as a proxy for efficiency of the schedule (a shorter schedule usually means less awake time and therefore less energy consumption). But more importantly a shorter schedule means that schedule activities could be lengthened (r.g. rover drive activity) or more activities can be added (e.g. science observations).

Makespan gain is the difference between the makespan of the original schedule and the makespan of the final executed schedule. Because in rover operations conservative (e.g. longer activity duration) models are used for scheduling, execution typically results in a shorter (smaller) makespan. Therefore we define larger makespan gain as a larger desired shortening.

[2]This is described as *event-driven* rescheduling, as compared to rescheduling that occurs at a fixed frequency (Chi et al. 2018).

Commit window

The purpose of a commit window is to (a) determine what activities to execute while the scheduler is rescheduling and (b) to specify the context (e.g. activities will be executed) prior to the new scheduled activities that the scheduler returns. One common policy is to set the commit window to span from the time of scheduler invocation to a conservative estimate of when the scheduler will complete. This means that (a) any activities in the prior schedule with scheduled start times in this interval are eligible for execution when their start times occur; (b) these activities scheduled to start within the commit window are the context (e.g. to determine state and resource values) for generation of the new schedule. Additionally, any activities with start times beyond the end of the commit window will be considered for rescheduling and the scheduler may only reschedule activities to start later than the end of the commit window. While such a conservative commit window prevents conflicts between the existing schedule and the newly generated schedule while the scheduler is running, it often also results in loss of performance as it limits which activities can be rescheduled, as shown in the empirical results section.

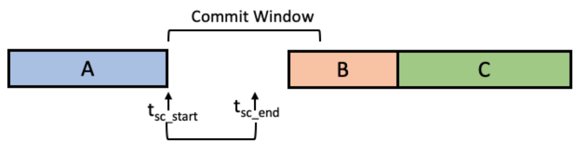


Figure 1: The scheduler is invoked at time t_{sc_start} because Activity A ends early. The scheduler completes at time t_{sc_end} but it cannot reschedule Activity B to start earlier because the start time of B is within the commit window, so it will execute as per the prior schedule. The earliest the scheduler can reschedule Activity C to start is at the end of the commit window.

Prior work has noted that under the assumptions that the actual scheduler runtime is the predicted scheduler runtime and the commit window is set to this time, a shorter scheduler runtime and commit window resulted in higher makespan gain for a range of rescheduling invocation and flexible execution methods (Figure 2) (Chi et al. 2018).

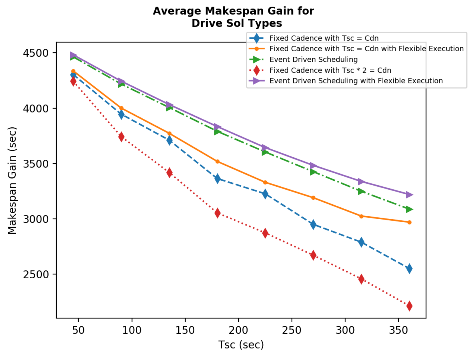


Figure 2: For all the methods explored in (Agrawal, Chi, and Chien 2019), a lower predicted scheduler runtime (and thus smaller commit window) resulted in more makespan gain.

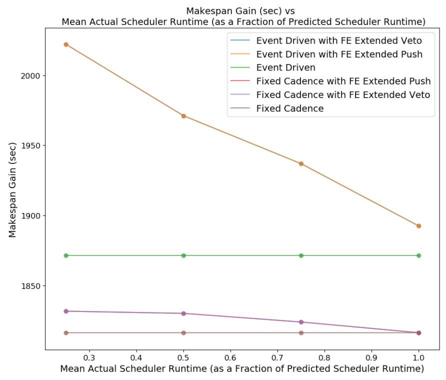


Figure 3: If we assume the scheduler finishes earlier than the commit window, FE can recoup makespan gain during the rest of the commit window. FE Extended Push and Extended Veto are variations of the FE algorithm but they behave the same when activities only end early, resulting in overlapping results and only 4 lines in the plot.

This is not surprising as longer scheduler runtimes prevent the scheduler from pulling activities forward when preceding activities complete early.

Note also that if the commit window is longer than the expected scheduler runtime, or if the expected scheduler runtime is longer than the actual scheduler runtime, flexible execution can take advantage of some of the time gained from early scheduler completion by pulling eligible activities earlier. In previous research, this area was explored by simulating ”actual scheduler runtime” as a normal distribution where the mean was some fraction of a fixed commit window (Agrawal, Chi, and Chien 2019). FE was somewhat effective at handling these cases where the scheduler finished early. When we simulated a smaller actual scheduler runtime, FE recouped more makespan gain than with a larger actual scheduler runtime (Figure 3).

In this paper, we consider the scheduler runtime and commit window in more depth. We can model scheduler runtime in the context of execution more accurately than in previous work. Once we have a better idea of its actual distribution, we can use this model to make a smarter commit window. Furthermore, we can show that in certain cases, a smarter commit window yields better results in execution.

Modeling Scheduler Runtime

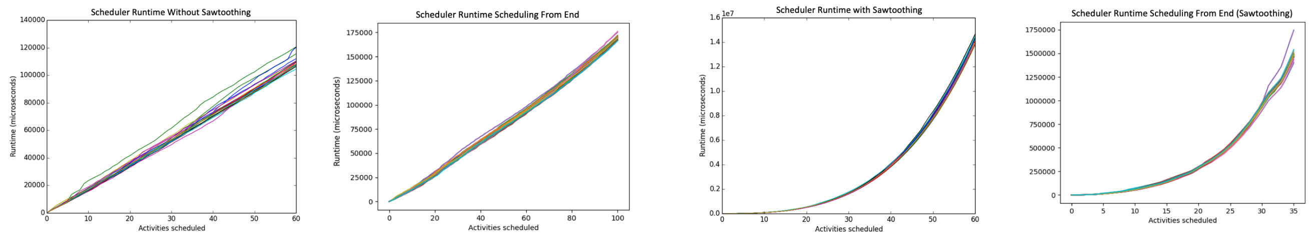
In the previous research described above, the actual scheduler runtime was modeled as a normal distribution with a mean as a fraction of the predicted scheduler runtime (and the commit window). We build on this prior work to develop an explicit predictive model of scheduler runtime based on empirical analysis. We develop models of the scheduler runtime through previous research on the scheduling algorithm, observations from synthetic plans, and tests on sol types.

Inspection of the scheduling algorithm and its implementation can suggest a model of the algorithm’s runtime. This scheduler has been analyzed independent of the context of execution, both theoretically and empirically. (Chi, Chien, and Agrawal 2019) found the worst-case runtime of the algorithm to be $\mathcal{O}(n^3)$, where n is the number of activities in the input plan. The paper also notes that the problem is easier when the incoming SOC is higher. The worst case performance is when the incoming SOC is lower and the scheduler has to repeatedly schedule a group of activities, which then enforces that the rover will sleep to generate power for another group of activities, then schedule the next group of activities. As the scheduler schedules each activity group, it must project forward the energy timelines, the most computationally demanding resource timelines, repeatedly. The final energy timelines have a repeated pattern of (a) a group of activities use energy leaving the rover with low energy, (b) the rover sleeps sleeps to regain a small amount of energy, (c) another small group of activities leaving the rover with no energy, (d) rover sleeps to regain energy, repeatedly. This pattern is described as *sawtoothing* due to the shape of the energy timeline.

Using Synthetic Plans to Observe Scheduler Runtime

While the worst-case runtime may be $\mathcal{O}(n^3)$ as described by (Chi, Chien, and Agrawal 2019), measuring the runtime of the scheduler on synthetic plans shows how different inputs can drastically change the runtime complexity. By noting which synthetic cases yield what complexity, we justify our expectations of scheduler runtime on mission-realistic cases with sol types.

Starting with a simple case, we adjust the plan structure and incoming SOC to cause the runtime complexity to increase. We show the runtime complexity by measuring the amount of time to finish scheduling each activity given that the previous activities are already in the schedule. Observe the following four cases in Figure 4. When the scheduler ran on a synthetic plan with 60 nonparallel activities, at a high incoming SOC the runtime is linear in n, the number of activities scheduled (Figure 4a). When the activities were



(a) Serial Activities + High Incoming SOC (b) Preferred Time at End of Plan + High Incoming SOC (c) Serial Activities + Low Incoming SOC (d) Preferred Time at End of Plan + Low Incoming SOC

Figure 4: Runtime complexity is linear or polynomial depending on problem characteristics.

given preferred times at the end of the plan, so that activities scheduled in backwards temporal order, the runtime exhibits a small quadratic component as shown in Figure 4b. When the incoming SOC was below the minimum required SOC, the problem did indeed become harder, as predicted by (Chi, Chien, and Agrawal 2019). The plans exhibited *sawtoothing*, where each activity had its own separate wakeup, awake, and shutdown, causing a sawtooth-like pattern of energy consumption as energy increases while the rover sleeps and decreases while it is awake (Figure 5). For a plan with sawtoothing, the third graph shows the runtime conforms to a function quadratic in n activities (Figure 4c). In the worst case, shown in the fourth graph, scarce power and required preheats exhibit a higher order polynomial form (Figure 4d).

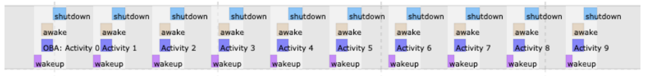


Figure 5: *Sawtoothing*, where each activity has its own wakeup, awake, and shutdown, occurs at low incoming SOC.

The most realistic case is to have activities schedule in mostly temporal order. Incoming SOC may vary. Therefore, the linear case and the quadratic case with sawtoothing are most likely to appear in sol types.

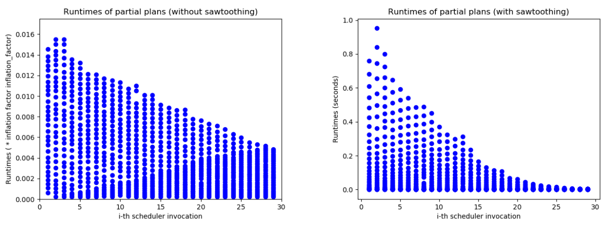


Figure 6: Runtimes while creating a schedule with and without sawtoothing

During execution, if every activity finishes early or late beyond a certain threshold, causing an event to occur as described in the Schedule Execution section above, the scheduler will reschedule n times: once at the beginning with n activities, then with $n-1$, $n-2$, etc. activities. Activities from earlier in the plan still have to be placed in the schedule, but this takes less time than scheduling, and does not affect the time to schedule following activities. The runtimes will decline linearly (without sawtoothing) and quadratically (with sawtoothing) (Figure 6). While the runtime does not usually exhibit the cubic complexity we would expect from inspection of the algorithm, using a normal distribution as a runtime model during execution is still quite unrealistic. If we must choose a commit window greater than or equal to the longest expected runtime, the commit window could be orders of magnitude greater than the shortest and average runtime during the execution of a non-sawtoothing plan.

Modeling Runtimes for Sol Types

Even a linear or quadratic model, however, is not perfect for modeling the scheduler runtime of actual rover sol types. Additional constraints such as heating thermal zones, execution time windows, and dependencies can either increase or decrease the time it takes to schedule each activity. Sawtoothing can still occur, but because of the additional constraints, scheduling at a low enough SOC to cause sawtoothing may also cause many activities to fail, resulting in a non-quadratic runtime. In addition, different plans have different constraints and degrees of parallelism, so finding a model that works consistently across all sol types is difficult.

We first describe the Linear model, which uses linear regression to predict the scheduler runtime, with inputs being the number of committed, scheduled, and failed activities (predicted based on what happened in the most recently generated plan), the incoming SOC, the maximum allowed SOC, the number of activities which must occur at a fixed time such as communication passes. The results were reasonably accurate for the sol types. Moreover, the coefficients are all close to the same order of magnitude [3], showing that each feature chosen for the linear regression was indeed correlated to the runtime. However, the data does have a somewhat quadratic shape in certain cases as runtime increases, and the Linear model could not capture this trend.

A different approach to modeling scheduler runtime during execution is to use the actual runtime of the immedi-

[3]The coefficients were approximately $5.49\text{e}{-3}$, $6.56\text{e}{-4}$, $-1.11\text{e}{-3}$, $1.95\text{e}{-3}$, $2.15\text{e}{-3}$, $-2.52\text{e}{-3}$.

Algorithm 1 Linear Runtime Prediction

```
Input:
    $A$: List of activities in the input plan
    $O$: Last output plan
    $e$: Energy at current time
    $t$: current time
    $inflation\_factor$: scalar describing how much longer the actual scheduler is expected to take than the computer on which the code is being run
    $c_0 \ldots c_4$: coefficients found by linear regression
Output:
    $r$: Predicted runtime of the scheduler
1: $committed \leftarrow a \in A$ if $a$ was committed in $O$ or starts before $t$
2: $failed \leftarrow a \in A$ if $a$ failed in $O$
3: $pinned \leftarrow a \in A$ if $earliest\_start\_time(a) == latest\_start\_time(a)$
4: $schedulable \leftarrow a \in A$ if $a \in A$ and $a \notin committed, failed, pinned$
5: $r \leftarrow (c_0 + c_1 * |committed\_activities| + c_2 * |failed| + c_3 * |schedulable| + c_4 * |pinned| + c_5 * e) * inflation\_factor$
```

ately previous scheduler invocation as a prediction for the current scheduler invocation runtime, which we call the Previous model. This is useful because characteristics such as number of activities to be scheduled, incoming SOC, heating activities, and constraints are often consistent from one invocation to the next. This method is more likely to overestimate than underestimate the runtime (because generally speaking some activities have executed since the last rescheduling, resulting in fewer activities remaining to schedule), but that is acceptable because it is preferrable to overestimate scheduler runtime (see (Agrawal, Chi, and Chien 2019)). It is possible for the scheduler runtime to increase over successive invocations; when this happens it is usually because activities ran long, causing the SOC to be lower than predicted by earlier invocations, so more activities sawtooth than before.

	fixed	linear	prev
Mean error	1.2e5	−2.2e3	2.1e3
Variance	8.2e8	5.1e8	1.3e9

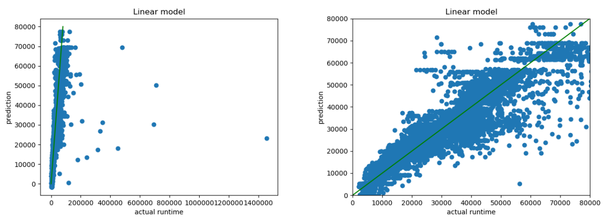


Figure 7: Linear Model is fairly accurate for many cases, but misses some outliers—greatly underestimating several. The graph on the right is the result of zooming in on the graph on the left.

Both the Linear and Previous models (Figure 7 and Figure 8) predict scheduler runtime during execution much better than a fixed number, because inputs like number of activities to be scheduled and incoming SOC change over the course of execution. [4]Overall, the Linear method predicts scheduler runtime more accurately than the Previous method. However, the Linear model's accuracy relies on the assumption that the input plan and other parameters will be similar to the conditions from which we derived the coefficients by using linear regression. Therefore, for work beyond the scope of the next section, the Previous model is still more accurate than using a fixed prediction, and it may even be better than Linear in some cases.

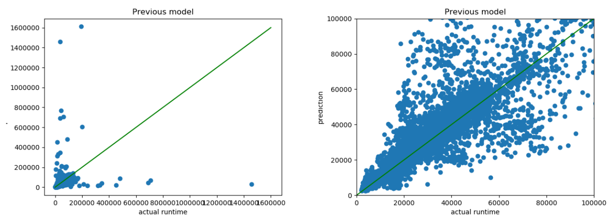


Figure 8: Previous Model adapts to different plans well, but it does not account for variation within the plan. The graph on the right is the result of zooming in on the graph on the left.

Variable Commit Window

An improved model of scheduler runtime enables a more accurate commit window. Compared to a constant, conservative estimate, this should result in smaller commit windows whose sizes decrease over the course of execution as more activities are executed successfully so that there are fewer activities remaining to schedule.

We show results of setting a variable commit window using both Previous and Linear models. Because our execution algorithm does not have a mechanism to address the scheduler running longer than estimated, when the scheduler does run long, we act as if it completed at the end of the commit window. The predictions from the Previous Linear models are each multiplied by a scale factor so that only 5% of predictions are expected to be underestimates (also making the comparison between Previous and Linear more comparable).[5]

While the variable commit window method (VCW) offers improvements over a fixed, longer commit window, flexible execution is often good enough at regaining time from overly long commit windows. We observe that variable commit windows are especially useful with certain constraints

[4]Both the Linear and Previous methods require information from the last invocation of the scheduler; during our simulation there is always an initial scheduling of the plan before any activities have started executing, which provides us with a schedule for Linear and a previous scheduler runtime for Previous. Since the first invocation does not interact with execution, we do not need to predict its runtime or set a commit window for it. This will not be the case on the actual M2020 onboard planner, but we defer a base-case runtime prediction to future work.

[5]The scale factor is based on the results from the data depicted in Figures 7 and 8. When the Linear predictions were each multiplied by 2.7, only 5% of the Linear predictions were underestimates. When the Previous predictions were multiplied by 1.3, only 5% of the Previous predictions were underestimates.

that FE cannot consider such as execution time ranges, setup activities, and energy constraints.

Empirical Results with Sol Types

We test the three methods for determining commit windows in sol types described in the ”Modeling Scheduler Runtime” section earlier: using a fixed commit window based on a conservative maximum scheduler runtime, a variable commit window based on the previous runtime invocation, and a variable commit window based on a linear estimate from characteristics of the plan and execution. We test execution over six different sol types, four random seeds (which determine the activity durations for execution), five inflation factors, and four levels of incoming SOC. We use an inflation factor to account for the fact that the scheduling runs are being done on a much faster laptop to generate the empirical results. This laptop takes a fraction of a second to generate a schedule, unlike the M2020 onboard planner which will is much slower. Each sol type contains between 20 and 50 activities. To make a proper comparison, commit windows and inflation factors scale together: an inflation factor of 300, for instance, means that the fixed commit window is 60 seconds, the Linear VCW runtime prediction[6]is scaled by 300, and the actual runtimes are scaled by 300; proportionately, an inflation factor of 2400 means the fixed commit window is 480 seconds and predictions and actual runtimes are multiplied by 2400.

These methods can be used with or without FE. Because execution without FE does not handle activities running long, to compare between runs with and without FE, we truncate the distribution from which ”actual” durations are drawn so that activities do not exceed predicted durations. The results demonstrate that more accurate (tighter) commit windows do improve makespan gain. When FE is operating, the effect of shorter commit windows is lessened, but still there (Figure 9).

The VCW methods result in a larger fraction of activities executing successfully at low starting SOC than FE does (Figure 10). In fact, in a few cases using FE in addition to VCW results in worse performance due to scheduler suboptimality.

If the scheduler were given lower CPU priority, it would have a much longer runtime (indeed in the original rover flight software design the scheduler was expected to have a runtime of 3.5 minutes, well more than the final 1 minute estimate). The higher inflation factors simulate this case, scaling up both the fixed and variable commit windows. At these higher inflation factors, the VCW methods have an even more positive effect on both the makespan and activities executed, shown in Figure 11 and more explicitly in Figure 12 with an inflation factor of 300 .

There are some oddities in these graphs due to outlier cases. Sometimes a longer commit window is better because an ”event,” as described in the introduction, means that an activity has finished early or late by *more than the commit window*. If an activity A finishes early by x seconds, and the commit window is $2x$ seconds long, rescheduling will not be triggered; instead, FE might be able to pull the next activity B all the way up to when Activity A ended. But if the commit window were $\frac{x}{2}$ seconds, and the scheduler runtime took the entire commit window, Activity B could only be pulled up to $end(A) + \frac{x}{2}$. However, the data points with extremely large inflation factors still show that if the maximum scheduler runtime is even longer than some of the activities' execution windows, VCW can do more than FE to improve the schedule throughout execution.

[6]The Linear VCW runtime prediction is already being multiplied by a scaling factor of 2.7 as described at the beginning of the VCW section; the 300 here is an additional inflation factor.

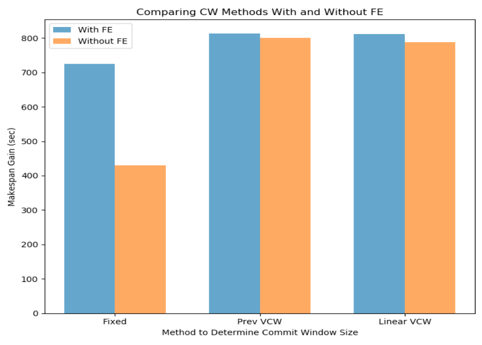


Figure 9: Tighter, more accurate commit windows enable more makespan gain, with and without FE.

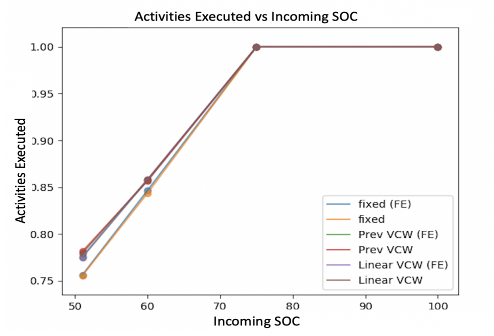


Figure 10: At low SOC, VCW helps more activities be executed more than FE does.

Synthetic Cases Highlighting Shorter Commit Window

While FE is able to recoup much of the time lost due to a conservative commit window, it is still myopic while VCW allows for deliberation and informed activity scheduling, which is especially beneficial in certain cases.

We construct problems to highlight where shorter commit windows are beneficial. In the following examples, the fixed commit window is 60 seconds. We use the Previous model to predict scheduler runtime for the variable commit window, and we multiply the actual scheduler runtime by an inflation factor of 300. We do not use the Linear model with synthetic plans because the Linear Model coefficients were derived using plan and execution characteristics of the sol

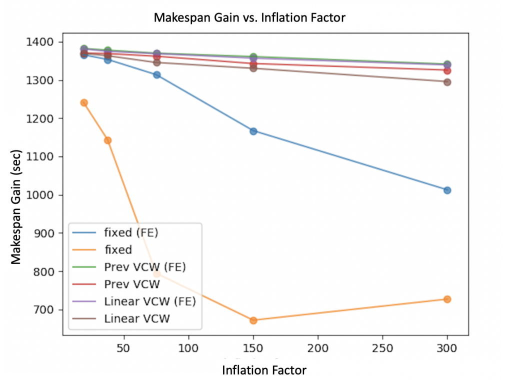


Figure 11: As the scaling factor for both commit windows and inflation factors grows, VCW and FE benefit increases.

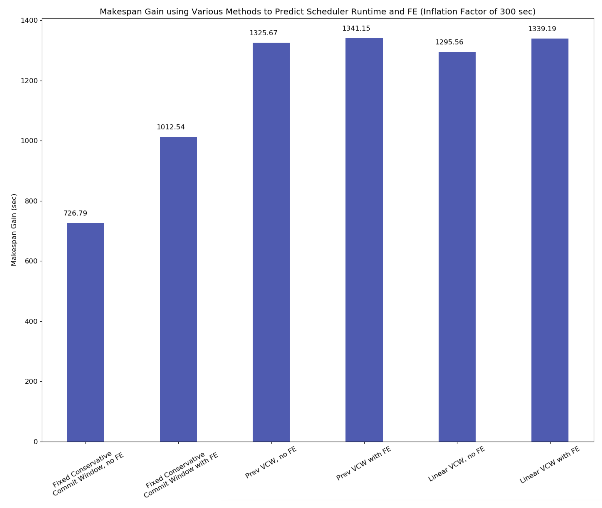


Figure 12: Makespan gain using different methods to predict commit window with and without FE at an infaction factor of 300

types. Deriving a model that generalizes to synthetic problem sets is an area of future work. Synthetic plan characteristics are selected to highlight the weaknesses of FE and the strengths of VCW:

- *Tight execution windows.* FE can only pull and push activities within their continuous execution time windows, while rescheduling can move activities among different windows. VCW is then better if having a shorter commit window means an activity can be moved into an earlier execution window (Figure 13). In some cases, activities may even end up switching order, which cannot happen with FE. When constraints are very tight, an activity running long often causes the next activity to be displaced from the schedule, but with VCW, if a future activity ends early, the unscheduled activity can be scheduled back in the plan right then (Figure 14).

  The effect of VCW on makespan gain can be seen in many tightly constrained plans. A synthetic problem generator was constructed that gives each activity a few randomly placed tight execution windows. With four different plans, each run on four different seeds, makespan gain was higher for the VCW methods than with a fixed, longer commit window, and FE had *no effect* on makespan for any method because of the tight constraints (Figure 15).

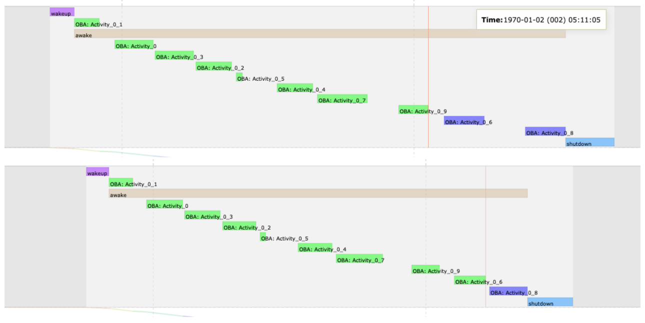


Figure 13: Activity 8 has execution windows near adjacent but disjoint, so FE cannot *pull* Activity 8 into the earlier window, but VCW can *reschedule* it into the earlier window.

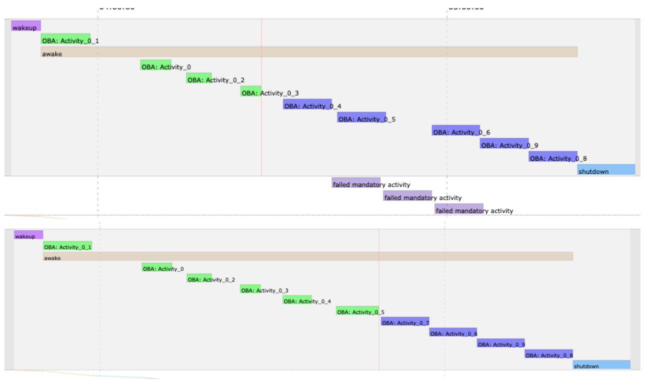


Figure 14: At first, activity 6 is excluded because none of its execution windows fit into empty space in the schedule. But when activity 5 ends early, VCW enables 6 to fit.

- *Resource Constraints, e.g. Low SOC.* FE cannot affect sleep state other than pulling forward shutdowns when activities end early, while rescheduling can move and change the durations of awakes. Also, FE does not consider energy, while the scheduler does. This becomes significant at low SOC, with sawtoothing, when an activity ends early right before a shutdown. VCW can schedule an earlier shutdown and take advantage of this extra energy to improve the plan. The scheduler can improve the plan by scheduling an additional activity later (Figure 16), or by scheduling the next activity (with its associated wakeup, awake, and shutdown) to occur at an earlier time. FE can pull the shutdown, but by then the scheduler will have finished running with the shutdown occurring after the commit window, still assuming that there won't be enough energy to improve the plan, and FE cannot add an awake when there is extra energy.

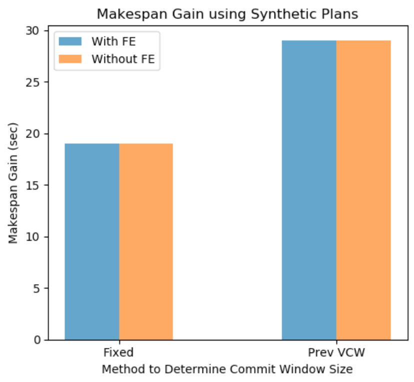


Figure 15: Using VCW improved makespan gain across several problems with constrained synthetic plans while FE had no effect on makespan gain.

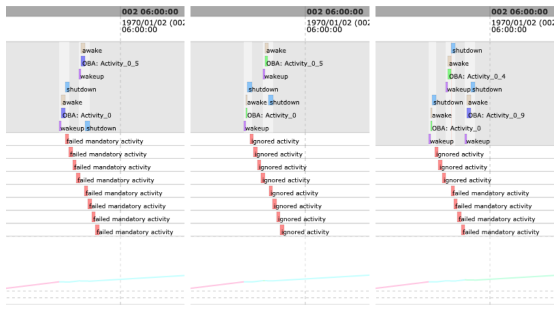


Figure 16: When an activity ends early, FE can pull a shutdown forward (middle image), but it cannot use the extra energy to schedule another activity, as VCW can (3rd image).

- *Preferred times.* When activities have preferred times specified during the sol, they are no longer placed at the earliest possible times, but rather as close to their preferred times as possible. Here, activities in a synthetic plan were left unconstrained but given random preferred times within a small fraction of the sol and VCW showed improvement over the Fixed method both in terms of achieving a higher makespan gain and scheduling activities closer to their preferred time (Figure 17) .

Sol types tend to have tight execution windows for activities; they should be able to perform well at low SOC, although they are designed with the hope that they start at higher SOC; and they do not currently have preferred times. Given these comparisons to the synthetic plans and other considerations, sol types are expected to benefit from VCW in some places, but in general FE should be sufficient to handle execution changes during long commit windows.

Related and Future Work

Deep Space One's New Millenium Remote Agent treated replanning as its own activity that would be scheduled into the plan. Its timing and resource estimates were conservative but activities were given flexibility to handle duration

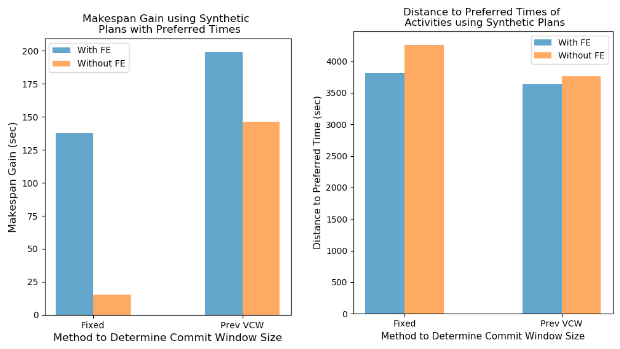


Figure 17: VCW is better equipped to balance the demands of preferred time with makespan gain.

uncertainty (Pell et al. 1997; Muscettola et al. 1998).

The CASPER system (Chien et al. 2000) handles execution and replanning by iteratively repairing the schedule as it becomes inconsistent with execution, an anytime approach. However, the resultant schedule is not guaranteed to be conflict-free. CASPER flew onboard the Earth Observing One (Chien et al. 2005) for over a dozen years and flew on the IPEX cubesat for one year (Chien et al. 2016).

Another way to handle time-consuming problem-solving within execution is to use the ”design-to-time” approach which uses meta-level deliberation to select a problem-solving approach based on how much time is available (Garvey and Lesser 1995; 1996). Considerable prior work in AI has studied this problem of ”how much/long” to think (e.g. (Zilberstein and Russell 1995; Horvitz 2011)), this paper has focused on the dual problem, namely how to deal with uncertainty in ”thinking” runtime in the context of execution in the context of an applied problem.

We have investigated linear runtime models based on few parameters; future work could investigate more powerful analytical techniques and additional parameters. Also, we have focused on conservative runtime models; a more comprehensive approach would be to understand the cost of under- and overestimating runtime and to incorporate this in biasing the use of the estimated scheduler runtime in setting the commit window.

Conclusions

Schedulers typically must operate in the context of execution. In these cases, lost productivity due to inefficient execution during rescheduling can be significant.

We have described work to model scheduler runtime to improve scheduler runtime prediction accuracy, thereby enabling a reduction in lost productivity during rescheduling. We presented two models to predict scheduler runtime: (1) based on current scheduler inputs and (2) based on scheduler runtime from the immediately prior scheduler invocation.

Use of these scheduler runtime prediction models resulted in modest execution productivity on actual rover sol types and more significant improvements on synthetic data designed to highlight potential improvements.

References

Agrawal, J.; Chi, W.; Chien, S.; Rabideau, G.; Kuhn, S.; and Gaines, D. 2019. Enabling limited resource-bounded disjunction in scheduling. In *11th International Workshop on Planning and Scheduling for Space (IWPSS 2019)*, 7–15.

Agrawal, J.; Chi, W.; and Chien, S. 2019. Extended abstract-using rescheduling and flexible execution to address uncertainty in execution duration for a planetary rover. In *11th International Workshop on Planning and Scheduling for Space (IWPSS 2019)*, 4–6.

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. In *International Conference on Automated Planning and Scheduling (ICAPS 2018)*.

Chi, W.; Chien, S.; and Agrawal, J. 2019. Scheduling with complex consumptive resources for a planetary rover. In *International Workshop for Planning and Scheduling for Space (IWPSS 2019)*, 25–33.

Chi, W.; S.Chien; and Agrawal, J. 2020. Scheduling with complex consumptive resources for a planetary rover. In *International Conference on Automated Planning and Scheduling (ICAPS 2020)*.

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.; Sherwood, R.; Tran, D.; Cichy, B.; Rabideau, G.; Castano, R.; Davis, A.; Mandl, D.; Frye, S.; Trout, B.; 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.; 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* 307–315.

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.

Garvey, A., and Lesser, V. 1995. Design-to-time scheduling with uncertainty. Technical report, UMass Computer Science Technical Report.

Garvey, A., and Lesser, V. 1996. Design-to-time scheduling and anytime algorithms. *ACM SIGART Bulletin* 7(2):16–19.

Horvitz, E. 2011. *Metareasoning: Thinking about thinking*. MIT Press.

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.

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.

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

Zilberstein, S., and Russell, S. 1995. Approximate reasoning using anytime algorithms. In *Imprecise and approximate computation*. Springer. 43–62.