# A Comparison of Combinatorial Optimization and Dispatch Rules for Online Scheduling

David Montana

BBN Technologies

10 Moulton Street, Cambridge, MA 02138

EMail: dmontana@bbn.com

## Abstract

Online, or dynamic, scheduling refers to when the tasks to perform are not known prior to the execution of the schedule but rather are continually revealed during the execution process. The real-time aspect of online scheduling potentially changes the preferred properties of the scheduling algorithm. In particular, the time required to generate a schedule becomes a more important factor, while the quality of the schedules produced becomes less important. The relative importance of these two properties, schedule creation time and schedule quality, differs greatly depending on the characteristics of the scheduling problem. Here, we describe an empirical investigation of how the tradeoff between them varies with the scheduling problem. We do this by comparing the performance of an optimizing scheduler, which creates better schedules, and a dispatch-rule scheduler, which provides faster turnaround, on a set of different online scheduling problems. Our experiments identify some key factors in determining which type of scheduler is better: (i) the predictability with which schedules are executed and (ii) the time scale with which the scheduling problem changes as compared to the time required to optimize a schedule.

## 1 Introduction

For online scheduling, also referred to as dynamic scheduling, schedule creation and schedule execution are not separate processes, but rather are inextricably intertwined. The tasks are not known before starting the execution process but rather incrementally arrive in the scheduler's task "bin" during execution [6]. Hence, schedules need to be dynamically created and updated in real time.

Online scheduling places different requirements on a scheduling algorithm. One such requirement is that schedules should be created and updated quickly enough to react to changes dynamically. The consequence is that often a different type of algorithm is required for online scheduling than "standard" static scheduling. For example, to schedule processes in a computer operating system, it does not make sense to use combinatorial optimization, because any decision needs to be made too quickly. Instead, a simple dispatch rule for when to switch processes and which process to run next is better.

However, not all online scheduling problems are like this. For example, field service scheduling (i.e., scheduling repairpeople to service calls) is an online scheduling problem, but an optimizing scheduler will work well for this problem [9]. The reasons for the difference, as we will discuss, are a larger time scale and schedule execution that is relatively predictable.

In this paper, we carry out a systematic experimental exploration of the tradeoff in importance between schedule creation time and schedule quality as the properties of the online scheduling problem vary. To do this, we define a sample online scheduling problem that is simple enough that it can be easily understood and analyzed but complex enough that scheduling performance can potentially benefit from combinatorial optimization. We implement the dynamic aspect of the problem as a simulation in pseudo-real time. This allows explicit control over the time for schedule creation relative to the actual time of the process scheduled.

Parameters control various aspects of the online scheduling problem, including a variety of statistical distributions, such as the arrival time of tasks, the time to complete tasks, and the requirements on when to complete tasks. Also determined by parameters are precedence relationships, delays, transition times, and priorities. Varying these parameters changes the nature of the scheduling problem.

We use two different schedulers. One performs combinatorial optimization; since the problem is sufficiently small, it finds the optimal schedule according to our defined criterion. A second performs dispatch-rule scheduling, optimizing the same criterion but only in a greedy manner as resources become free. This allows

comparison between the two types of scheduling as we vary the nature of the scheduling problem as well as the schedule creation times.

The rest of the paper is structured as follows. Section 2 discusses previous work on online scheduling and related topics. Section 3 defines our sample scheduling problem in detail. Section 4 describes the two different scheduling algorithms used. Section 5 discusses experiments and results.

## 2 Previous Work

While it is still prevalent to treat scheduling as a static optimization problem, there has been a variety of work on the dynamic aspects of scheduling. We now examine different threads of research on dynamic scheduling. [Note that we only cite representative examples; more comprehensive surveys of the field are available, such as [15]. Further note that the vocabulary in the field is not standardized, with researchers often using terms differently. We attempt to be as consistent as possible with previous usage.]

One general approach, used by [4] and [14] among others, divides the scheduling process into planning and execution phases. During the planning phase, a *predictive scheduler* performs combinatorial optimization to create a nominal schedule. During the execution phase, a *reactive scheduler* continually modifies or repairs the nominal schedule to account for unanticipated circumstances. One goal of the reactive scheduler is to keep the schedule as close as possible to the original nominal schedule. This approach makes sense when all the scheduling data (including what tasks to schedule and the state of the resources) is known for a certain period of time into the future, and deviations that arise during execution are likely to be minor.

A second approach to dynamic scheduling is *dispatch rules*. In practice, the use of dispatch rules is very common, although the choice to use dispatch rules for real-world scheduling often has more to do with ease of implementation than scheduling performance. There is a large variety of different rules that have been used, many of them chronicled in summaries such as [11]. When simple heuristic rules, such as earliest-due-date (EDD) and first-in-first-out (FIFO), were applied to a range of different scheduling problems, it was found that which rule was best varied with the problem [10]. There is also the potential for using a set of rules, rather than a single rule, in an expert system [7]. The advantage of using dispatch rules is that it enables quick reaction to changes. The disadvantage is that it creates suboptimal schedules because it does not plan into future. This approach for online scheduling is best when there is online

scheduling (i.e., tasks being revealed during execution), the fundamental time scale of the problem (a concept we will examine below) is short, and the future is very uncertain.

A third approach to dynamic scheduling is continuous combinatorial optimization. Periodically, a scheduling algorithm essentially recomputes the entire schedule to reflect the changed situation. A simple repair algorithm changes the schedule minimally as required to ensure that no hard constraints are violated due to unexpected deviations from the schedule between scheduler updates. In practice, the results can be similar to the predictive/reactive approach. This is particularly true when the optimization criterion includes a term that rewards schedule stability, i.e. keeping the schedule as similar to the previous schedules as possible, usually for the sake of humans either being scheduled or assisting with the schedules. (Examples of the use of a schedule stability terms are given in [12] and [13].) The fundamental distinction between this approach and the predictive-reactive approach is that the latter has two distinct scheduling algorithms (one for planning and one for execution) while the former uses a single scheduling algorithm (with no distinction between planning and execution). A real-world example of this approach applied to field service scheduling, i.e. scheduling of repairpeople, is described in [9]. This is a good approach when there is online scheduling, the future is relatively predictable (at least in the near term), and the time scale is long enough that an optimized schedule can be generated within a time that the problem is not likely to change significantly.

The fourth approach is stochastic optimization. Algorithms of this variety perform a statistical analysis of possible future scenarios (usually via a Monte Carlo method) to compare expected payoffs of different scheduling decisions. Examples of this approach are [5], [2] and [3]. This approach works well when there is online scheduling and the future is unpredictable in a deterministic sense but is predictable in a statistical sense. Except in those cases where strategies can be precomputed, this approach (like combinatorial optimization) requires that the fundamental time scale of the scheduling problem not be too small.

As we have pointed out, these different approaches would be expected to perform well under different conditions. Certain extreme situations are clearly suited to one particular approach. (For example, the very short time scale of operating system scheduling means that dispatch rules are best suited.) However, often the situation is in some middle ground, and it is not clear which approach is best. In this paper we examine how to quantify the tradeoffs to help decide on the best approach.

# 3 Test Problem Definition

In order to evaluate the performance of different online scheduling algorithms, we need one or more test problems. We have invented a test problem with the following desirable properties:

- It involves online scheduling.
- There is a clear potential advantage to planning into future, but also a clear potential advantage to fast schedule turnaround (hence establishing the fundamental tradeoff).
- Altering the parameters specifying the statistical properties of the data changes the nature of the problem (essentially allowing the creation of different scheduling problems) in a well controlled way.

We now describe the details of this test problem.

## 3.1 Jobs, Tasks and Resources

A *task* is an atomic unit of work to perform. A *job* is a set of one or more interrelated tasks. Each task is part of exactly one job. The tasks within a job form a simple sequence, with the first task in the sequence needing to finish before the second begins, and so on. There is potentially a non-zero delay between tasks in a sequence, where a task must wait a specified amount of time after its preceeding task completes before being allowed to execute.

Jobs enter the system continually. Each job has associated with it three times:

- arrival time: when the system/scheduler first has knowledge of the job
- release time: the earliest that the job can start executing
- deadline: the time by which the job must complete or else be considered late

Each job is a member of a *job class*. All the jobs in a particular job class share the following properties:

- (fixed) sequence of tasks with the associated delays
- (fixed) utility, which is the cost of not finishing the job within the deadline (with the overall performance metric being the sum over all dropped jobs of the job's utility)

All the jobs within a job class are generated probabilistically from the following statistical distributions:

- jobs arrive following an exponential distribution, specified by the mean time between arrivals
- each job's holding duration (time between arrival and release) is generated from a truncated normal distribution, specified by the mean, standard deviation, and minimum value
- each job's execution duration (time between release and deadline) is generated from a truncated normal distribution

A *resource* can perform at most one task at a time. Each resource can perform only tasks of a certain *task type*, where a task's type is defined by its job's class and its position in the sequence of the job. The time required to execute a task, which depends on the task type, is generally non-deterministic. Between tasks of different types, there are potentially non-zero resource transition times, during which the resource is not available to execute a task. Resources fail intermittently and are not available to execute tasks until they are repaired.

Each resource is a member of a *resource class*. All the resources in a resource class have the same properties, which include:

- which task types are handled
- (fixed) transition times between tasks of different types as a function of the two task types
- truncated normal distributions of task execution times, with a different distribution per task type, each determined by a mean and standard deviation with a minimum value of 1
- exponential distributions of time between failures and time for repairs, determined by the mean values

Unlike jobs, all resources are present at the start of the simulation and persist throughout. Each resource class specifies:

- the number of resources of that class

## 3.2 Pseudo-Real-Time Execution

In the previous subsection, we examined the static aspects of the scheduling problem, and now we discuss the dynamic aspects. To perform online scheduling, we need a dynamic process with a sense of time progressing. In this case, we create a simulated dynamic process with a simulation clock, which provides us with easy control over time-related issues.

There are three basic processes occuring in the simulation:

1. Events (associated with new job arrivals, resource failures, and resource repairs) are arriving into the system.
2. The scheduling algorithm is creating schedules by assigning tasks to resources at specified future times.
3. Tasks are being executed by resources as specified by the current schedule.

**Event Processing -** The simulation maintains a list of jobs with their arrival times invisible to the scheduling and execution processes. When the simulation time advances to the arrival time of a job, the simulation injects the job into the system by revealing it to the scheduling and execution processes. Resource failure events and re-

source repair events are treated similarly. Note that the simulation time is quantized so that every event must occur at a time that is an integral number of simulated seconds into the run.

**Scheduling -** The scheduling algorithm is invoked to create a new schedule under three possible conditions:

- a new event (job arrival, resource failure/repair) occurs
- a specified number of seconds has passed since the last invocation of the scheduler
- a reactive (dispatch-rule) scheduling algorithm is being used **and** a resource becomes free

However, if the scheduler is already in the simulated process of creating a schedule when cued to create a new schedule, the new schedule creation process must wait to begin until the previous schedule is completed.

The scheduling algorithm requires a potentially non-zero amount of simulation time to make its decisions and create a new schedule. The number of seconds required for the scheduler to run is a parameter of the simulation. (Note that the simulation time required to generate a schedule has no relation to the actual time that the algorithm runs.) During the simulated time that the scheduler is creating a schedule, it ignores any changes to the underlying data that occur. These changes include not only new events but also deviations from expected task execution times. Furthermore, the schedule that was last created stays in effect for the duration of the simulation time required to create a new schedule. Note that this schedule creation duration can cause degraded utilization of the resources because: (i) there is a delay in reacting to changes and (ii) newly created schedules are based on outdated information.

**Task Execution -** One important aspect of task execution is the decision which task to execute with which resource at what time. If everything goes according to the schedule, then each resource executes each task assigned to it at the time when it is assigned. However, due to the stochastic nature of the problem, generally schedules do not proceed exactly as planned, and some form of schedule repair is required. The execution process does the simplest repair to ensure that the schedule does not violate hard constraints (leaving any more sophisticated adjustments to the schedule to the scheduling process). Each resource maintains the same set of assigned tasks in the same order that they are scheduled, but ignores the absolute times of these assignments. The resource executes the next task at the earliest time that does not violate hard constraints (including task precedence relationships, earliest release time, and resource failures and transition periods). Note that this means a task can be executed earlier than it was scheduled (if the preceeding task in its sequence or the preceeding task on its resource finishes earlier than anticipated) as well as later than it

was scheduled.

A second aspect of task execution to consider is how to handle resource transition periods, i.e. the time required for a resource to reconfigure to handle a task of a different type than the previous one. We specify that a resource should start the transition process at the earliest possible time that it appears a transition will be needed and the resource is idle. It should not wait until the next task is ready to execute, because that will cause an unnecessary delay in the tasks's execution waiting for the transition to occur. There will be times, namely when the schedule changes which task to execute next, when this policy can potentially cause extra delay, but overall it leads to less time with resources idle.

A third aspect of task execution is what happens to tasks and transition periods that are in the middle of executing when a resource fails. We specify that any such interruption of a task or transition has the effect that not only does the task or transition stops executing but also that the state is as if the task or transition had never started executing. Furthermore, such a task is taken off the schedule until the scheduling process puts it back on the schedule.

# 4 Scheduling Algorithms

The way we have defined our scheduling problem, there will generally be clear distinctions between well-performing and poorly-performing schedulers. The good schedulers will align tasks so as to minimize the transition and idle times of the resources. They will also make good decisions, when overwhelmed with tasks, about which jobs to drop and will make those decisions early in the jobs' task sequence to avoid wasted execution time. This will result in more jobs of higher utility finishing by their deadline.

We have implemented two scheduling algorithms, whose performance we will compare empirically. One algorithm uses combinatorial optimization, while the other employs a dispatch rule. We do not claim that these particular algorithms are the best of their respectives classes in terms of performance, but they are representative. Hence, we can draw at least preliminary conclusions about the relative merits of the two classes of scheduling algorithms.

We now describe each of these two algorithms.

## 4.1 Optimizing Scheduler

We have created the optimizing scheduler using Vishnu [8, 1]. Vishnu is a scheduling application we developed that is what we call a *reconfigurable scheduler*, which means that it can be configured for a wide variety of dif-

ferent scheduling problems without modifying the software. For the purposes of this work, we configured it to implement the hard constraints described above and the optimization criterion given below. It will search for, and generally find (given enough time), the optimal schedule that satisfies the hard constraints. (The actual time required to perform the optimization does not affect the simulation results, since simulation time is completely separate from wall-clock time.)

The optimization criterion for the scheduler should consider not just how a schedule as constituted satisfies the criterion of completing jobs on time, but also what happens beyond its time horizon and how future events could disrupt the schedule. We define the optimization criterion as

$$\sum_{j \in J} \rho(j) \cdot P_f(j)$$

where $J$ is the set of all outstanding jobs (i.e., jobs that are neither completed nor overdue), $\rho(j)$ is the utility of job $j$, and $P_f(j)$ is the probability that $j$ does not complete on time given the proposed schedule.

The value of $P_f(j)$ must be estimated, even for jobs whose deadlines are within the scheduling window, because of the stochastic nature of the task execution process. (Note that the scheduling window is the finite interval of time into the future into which the scheduler can place tasks. For a job whose deadline lies within this window, we know whether or not the job is scheduled to finish in time, but can only estimate whether it will actually finish in time.) We do not use a formal approach since there is no analytic solution and a Monte Carlo approach would take too long given all the jobs in all the schedules we need to handle. Instead, we just use a reasonable, quick estimate (in a similar fashion to how chess-playing algorithms use a quick board evaluator to evaluate one of many potential paths into the future).

The estimate of $P_f(j)$ for job $j$ is computed as follows. Let $t_s(j)$ denote the last task in the sequence of tasks for $j$ that is scheduled if such a task exists. (It will not exist if no tasks of $j$ are currently scheduled.) Let $t_n(j)$ be the first task in the sequence of tasks for $j$ that is not scheduled if such a task exists. (It will not exist if all tasks of $j$ are currently scheduled.) Either $t_s(j)$ or $t_n(j)$ must exist, and if they both exist then they are consecutive. If $t_s(j)$ exists, compute the quantity

$$\text{Prob}\{E_{req}(j) < \text{rand}(E_{sch}(t_s(j)) + T_{foll}(t_s(j)),$$
$$\sigma_{foll}(t_s(j)) + \sigma_{exec}(t_s(j)))\} \quad (1)$$

where $E_{req}(j)$ is the required end time (i.e., deadline) for $j$, $E_{sch}(t_s(j))$ is the scheduled end time for $t_s(j)$, $T_{foll}(t_s(j))$ is the expected time to complete job $j$ after finishing task $t_s(j)$, $\sigma_{foll}(j)$ is the standard deviation in

the time to complete $j$ after finishing $t_s(j)$, $\sigma_{exec}(j)$ is the standard deviation in the task execution time of $t_s(j)$, and $\text{rand}(m, \sigma)$ is a random number selected from a normal distribution with mean $m$ and deviation $\sigma$. If $t_n(j)$ exists, compute the quantity

$$\text{Prob}\{E_{req}(j) < \text{rand}(E_{win} + T_{incl}(t_n(j)),$$
$$\sigma_{incl}(t_n(j)))\} \quad (2)$$

where $E_{win}$ is the end time of the scheduling window, $T_{incl}(t_n(j))$ is the expected time to complete $t_n(j)$ and any subsequent tasks in $j$ (with the caveat that once the window is shifted forward in time $t_n(j)$ could actually be scheduled to start earlier than the previous window end time without anything else changing), and $\sigma_{incl}(t_n(j))$ is the standard deviation in the time to complete $j$ including $t_n(j)$. Then, $P_f(j)$ is given by Equation 1 if $t_n(j)$ does not exist, is given by Equation 2 if $t_s(j)$ does not exist, and is the average of these two terms otherwise.

One detail we passed over is how we estimate quantities such as $T_{foll}(t)$. We posit

$$T_{foll}(t) = \sum_{t_f \in T_f} (T_{delay}(t_f) + T_{exec}(t_f) + T_{idle}(t_f))$$

where $T_f$ is the set of all tasks following $t$ in sequence, $T_{delay}(t_f)$ is task precedence delay time, $T_{exec}(t_f)$ is estimated task execution time and $T_{idle}(t_f)$ is the time beyond the minimum that $t_f$ has to wait to obtain a resource. While we know $T_{delay}$ and $T_{exec}$ a priori, we just make a guess (with constant set via experimentation) that $T_{idle}$ is approximately equal to $T_{exec}$. So,

$$T_{foll}(t) = \sum_{t_f \in T_f} (T_{delay}(t_f) + 2.0 \cdot T_{exec}(t_f))$$

Similarly, we estimate

$$\sigma_{foll}(t) = \sum_{t_f \in T_f} (1.4 \cdot T_{exec}(t_f))$$

$$T_{incl}(t) = T_{foll}(t) + 1.7 \cdot T_{exec}(t)$$

$$\sigma_{incl}(t) = 0.7 \cdot T_{incl}(t)$$

## 4.2 Dispatch-Rule Scheduler

The dispatch-rule (reactive) scheduler does no look-ahead in terms of planning schedules, instead waiting for a resource to become free before choosing a single task to assign this resource. The general approach for a dispatch rule is to define a score associated with assigning a given task to a given resource and select the eligible task that minimizes (or maximizes) that score for

the chosen resource. For our dispatch rule, we have defined a scoring function that uses the same basic idea as the optimization criterion for our optimizing scheduler. Not only does this work effectively, but it also reduces the differences between the two schedulers beyond the fundamental difference, the amount of look ahead.

For resource $r$ and task $t$ in job $j$, we define the score for $t$ as

$$\rho(j) \cdot (P_{assign}(t) - P_{not}(t))$$

where $P_{assign}(t)$ is the probability that $j$ fails to complete on time if $t$ is assigned to $r$ and $P_{not}(t)$ is the probability that $j$ fails to complete on time if $t$ is not assigned to $r$. This will usually be a negative number, with a more negative value indicating a bigger difference made by scheduling $t$ now rather than waiting. Essentially, this is doing triage, throwing away those jobs that are unlikely to finish on time and postponing those jobs that can safely be postponed. We estimate $P_{assign}(t)$ just as in in Equation 1 with $t_s(j) = t$

$$\mathrm{Prob}\{E_{req}(j) < \mathrm{rand}(E_{sch}(t) + T_{foll}(t),$$
$$\sigma_{foll}(t) + \sigma_{exec}(t))\}$$

We estimate $P_{not}(t)$ as

$$\mathrm{Prob}\{E_{req}(j) < \mathrm{rand}(E_{sch}(t) + T_{foll}(t) + T_{exec}(t) -$$
$$T_{idle}(t), \sigma_{foll}(t) + \sigma_{exec}(t))\} \tag{3}$$

where in this case $T_{idle}$ is the time between the current time and the start of any transition for $t$. Underlying Equation 3 is the assumption that $t$ would be scheduled immediately after the conclusion of a similar type of task that started immediately.

## 5 Experiments

### 5.1 Datasets

As discussed in Section 3, there are a variety of parameters that specify the nature of the data, much of it in a statistical fashion. A listing of these different parameters is given in Table 1. We have developed a data generator that can create datasets with the properties specified by the parameters. Generally, because of the probabilistic nature of the data, each time the data generator is run with a given set of parameters it will result in datasets that are markedly different, albeit with the same statistical properties. To gain repeatability in running the simulation, we have developed the capability for the data generator to save and then replay dataset instances. For experimentation, this repeatability is important because it greatly reduces the "noise" in the experiments. With the different scheduling algorithms all working on the exact

same data, differences in performance reflect more accurately differences in the quality of the schedules generated.

We have created two different datasets, which we refer to as *predictable* and *unpredictable*. As shown in Table 1, they are the same in many ways, but there are some critical differences (highlighted in bold). One similarity is that they both have two job classes, JC1 and JC2, with jobs from JC1 consisting of a sequence of three tasks and those from JC2 consisting of a single task. Both datasets also have two resource classes, RC1 and RC2, with resources from RC1 handling the first task of jobs from JC1 as well as the tasks from JC2 and resources from RC2 handling the last two tasks of jobs from JC1. In both datasets, the resources from RC2 require two seconds to transition between the two different types of tasks they handle.

The key differences that make the unpredictable dataset less predictable are

- higher resource failure rate: Resource failures represent a large unanticipated glitch in the schedule. The tasks assigned to the failed resource can no longer run as scheduled, affecting not only these tasks but other tasks that depend on these via a "ripple effect".
- greater variance of task execution times: The completion times of tasks will often be different from the expected end time. Again, this has ripple effects for other tasks on the same resource and other tasks in the same job.
- shorter holding and execution durations: With less advance warning about upcoming tasks and a greater urgency to start tasks as soon as possible, there is less time to plan in advance. This makes it more likely that new tasks will need to fit into the middle, rather than at the end of, the existing schedule and disrupt it.

The other differences between the two datasets are an attempt to keep the inherent level of difficulty of the two datasets approximately the same. Most notably, the increase from 2 instances of resource class RC2 (which is the bottleneck) to 3 instances going from the predictable dataset to the unpredictable dataset is a way to compensate for the fact that the resources are available less and are harder to allocate efficiently.

Both datasets are ten simulated hours long, providing roughly 5000-6000 instances of each job class.

### 5.2 Results

To evaluate the performance of a given scheduling algorithm on a given dataset, the key statistic we measure is the fraction of jobs finished on time. We divide this into separate measures for the two different classes of jobs, JC1 and JC2, since each of these job classes provides different challenges. JC1, with its three-task sequence

Table 1: Parameter Values for the Predictable and Unpredictable Datasets

| Parameter | Value 1 (Predictable) | Value 2 (Unpredictable) |
|---|---|---|
| Resource Classes | RC1, RC2 | RC1, RC2 |
| Number of Resources | **RC1: 3; RC2: 2** | **RC1: 3; RC2: 3** |
| Task Types Handled | RC1: TT1; RC2: TT2, TT3 | RC1: TT1; RC2: TT2, TT3 |
| Mean Task Execution Time | RC1: 2; RC2: 4, 6 | RC1: 2; RC2: 4, 6 |
| Std. Dev. Task Execution | **RC1: 0; RC2: 0, 0** | **RC1: 1; RC2: 2, 3** |
| Resource Transition Times | TT2→TT3, TT3→TT2: 2 | TT2→TT3, TT3→TT2: 2 |
| Mean Time Between Fails | **RC1, RC2: $5 \cdot 10^8$** | **RC1, RC2: 150** |
| Mean Time For Repair | RC1, RC2: 5 | RC1, RC2: 5 |
| Job Classes | JC1, JC2 | JC1, JC2 |
| Mean Time Bet. Arrivals | **JC1: 5.7; JC2: 8** | **JC1: 6.5; JC2: 4** |
| Min Holding Time | **JC1: 15; JC2: 4** | **JC1: 0; JC2: 0** |
| Mean Holding Time | **JC1: 3; JC2: 4** | **JC1: 0.5; JC2: 0.5** |
| Min Time to Complete | **JC1: 35; JC2: 2** | **JC1: 25; JC2: 5** |
| Mean Time to Complete | **JC1: 45; JC2: 4** | **JC1: 30; JC2: 6** |
| Dev. Time to Complete | **JC1: 5; JC2: 1** | **JC1: 2; JC2: 1** |
| Priorities | JC1: 1; JC2: 2 | JC1: 1; JC2: 2 |
| Task Type Sequence | JC1: TT1→TT2→TT3 | JC1: TT1→TT2→TT3 |
|  | JC2: TT1 | JC2: TT1 |
| Precedence Delay Times | JC1: 0, 2, 1; JC2: 0 | JC1: 0, 2, 1; JC2: 0 |

and longer holding and execution times, places more emphasis on planning. JC2, with just one task and quick required turnaround, puts a bigger emphasis on reaction time.

In addition to using different datasets and different scheduling algorithms, a third aspect we can vary is the schedule creation time. For the experiments, we only used the dispatch-rule scheduler with schedule creation time equal to zero. The reasoning here is that the benefit of the dispatch-rule scheduler is its ability to make fast decisions, so we assume that the time required for the decision process is insignificant. In contrast, we used a variety of different schedule creation times for the optimizing scheduler. This allowed us to explore a range of possibilities for schedule creation time to understand how it affects scheduling performance.

For all the runs with the optimizing scheduler, the scheduling window was set to 30 seconds.

The results are shown in Table 2. The delay is the schedule creation time.

A few observations about this data follow:

- When the scheduler delay (i.e., schedule creation time) is zero, the optimizing scheduler outperforms the dispatch-rule scheduler. This confirms our intuition that, all other things being equal, the ability of the optimizing scheduler to plan ahead gives it an advantage over a reactive scheduler.
- For zero delay, the difference between the optimizing and dispatch-rule schedulers is much greater for the predictable data than the unpredictable data. This

confirms our intuition that planning ahead loses much of its benefit when the future deviates significantly from what has been planned.
- As the simulated delay for the optimizing scheduler increases, the performance of this scheduler montonically decreases. The dropoff in performance is much faster for the unpredictable data ( 15% dropoff for a 3 second delay) than for the predictable data ( 15% dropoff for a 10 second delay). This confirms our intuition that scheduling delay is more harmful with less predictable data.
- The decline in performance is also much steeper for jobs of class JC2 than for jobs of class JC1. This confirms our intuition that scheduling delay is more harmful for jobs that require quick turnaround.

# 6 Conclusion

There are two basic causes of schedule disruption in online scheduling. One is the schedule not executing according to plan, e.g. due to unexpected task execution times or resource failures. The other is the arrival of new tasks with constraints dictating that they must be placed in the middle, rather than at the end of, the schedule. In response to schedule disruption, it is best to react quickly because a delay in reaction results in a decrease in scheduling performance. This leads to an inherent tradeoff between planning carefully and reacting quickly, since careful planning takes time that delays the response.

Table 2: Percentage of jobs completed on time for the different scenarios

| | Predictable | | Unpredictable | |
|---|---|---|---|---|
| Scheduler | JC1 | JC2 | JC1 | JC2 |
| Dispatch-Rule (no delay) | 85.8 | 99.3 | 89.7 | 99.6 |
| Optimizing (no delay) | 94.2 | 100.0 | 92.9 | 99.8 |
| Optimizing (1 sec. delay) | 93.6 | 99.9 | 90.3 | 99.0 |
| Optimizing (2 sec. delay) | 93.4 | 96.2 | 84.9 | 75.6 |
| Optimizing (3 sec. delay) | 93.1 | 85.2 | 78.0 | 45.5 |
| Optimizing (5 sec. delay) | 89.4 | 61.4 | 59.0 | 5.4 |
| Optimizing (7 sec. delay) | 85.2 | 33.6 | | |
| Optimizing (10 sec. delay) | 79.4 | 12.3 | | |
| Optimizing (15 sec. delay) | 73.5 | 2.3 | | |

We have investigated empirically this tradeoff between schedule quality and reaction time using a simulation of an online scheduling problem. To do this, we have implemented two different schedulers for this problem, one using combinatorial optimization and the other employing a dispatch rule. We have varied the simulated reaction time (scheduling delay) for the optimizing algorithm (while assuming that the dispatch rule reacts virtually instantaneously). To test the algorithms under different conditions, we have created datasets with different statistical properties. We have evaluated the scheduling performance under different combinations of scheduling algorithm, dataset, and reaction time. By doing so, we have verified experimentally that indeed there does exist a tradeoff between reaction time and schedule quality. Furthermore, less predictable, more disruptive data tilts the tradeoff further towards reaction time and enables the dispatch rule to outperform the optimizing scheduler with only a short scheduling delay.

There is much work that remains to be done in this area. For one, the experimental results are just preliminary. There are multiple dimensions of unpredictability and disruptiveness, and exploring these dimensions would require more variety than provided by the two datasets we used. Secondly, it would be highly desirable to have a better theoretical understanding of how the scheduling data affects the tradeoff between schedule quality and reaction time. Ideally, there exists some well-defined measure of unpredictability and how that affects the choice of scheduling algorithm. As a purely speculative example, consider the possibility of some defined fundamental time constant that says how long on average it takes before the scheduling problem changes by a certain fraction, roughly equivalent to the concept of half-life.

While there remains more to be done, we have taken some significant first steps by empirically demonstrating the dependence of the tradeoff between schedule quality and reaction time on the scheduling data, and by providing an approach for an experimental investigation of this tradeoff.

# Acknowledgements

# References

[1] BBN Technologies: 2004, 'Vishnu Reconfigurable Scheduler Home Page'. http://vishnu.bbn.com.

[2] Bent, R. and P. V. Hentenryck: 2004, 'Regrets Only. Online Stochastic Optimization under Time Constraints'. *Proc. 19th National Conference on Artificial Intelligence (AAAI'04)*.

[3] Bertsekas, D. and D. C. non: 1999, 'Rollout Algorithms for Stochastic Scheduling Problems'. *Journal of Heuristics* **5**(1), 89–108.

[4] Burke, P. and P. Prosser: 1991, 'A Distributed Asynchronous System for Predictive and Reactive Scheduling'. *International Journal for Artificial Intelligence in Engineering* **6**(3), 106–124.

[5] Chang, H., R. Givan, and E. Chong: 2000, 'Online Scheduling Via Sampling'. *Proc. Fifth International Conference on Artificial Intelligence Planning and Scheduling*. pp. 62–71.

[6] Dertouzos, M. and A. Mok: 1989, 'Multiprocessor Online Scheduling of Hard-Real-Time Tasks'. *IEEE Transactions on Software Engineering* **15**(12), 1497–1506.

[7] Kunnathur, A., P. Sundararaghavan, and S. Sampath: 2004, 'Dynamic rescheduling using a

simulation-based expert system'. *Journal of Manufacturing Technology Management* **15**(2), 199–212.

[8] Montana, D.: 2001, 'A Reconfigurable Optimizing Scheduler'. *Proc. Genetic and Evolutionary Computation Conference*. pp. 1159–1166.

[9] Montana, D., M. Brinn, S. Moore, and G. Bidwell: 1998, 'Genetic Algorithms for Complex, Real-Time Scheduling'. *Proc. IEEE International Conference on Systems, Man, and Cybernetics*. pp. 2213–2218.

[10] Montazeri, M. and L. V. Wassenhove: 1990, 'Analysis of Scheduling Rules for an FMS'. *International Journal of Production Research* **28**(4), 785–802.

[11] Panwalker, S. and W. Iskander: 1977, 'A Survey of Scheduling Rules'. *Operations Research* **25**(1), 45–61.

[12] Rana-Stevens, S., B. Lubin, and D. Montana: 2000, 'The Air Crew Scheduling System: The Design of a Real-world, Dynamic Genetic Scheduler'. In: *Genetic and Evolutionary Computation Conference Late Breaking Papers*. Morgan Kaufmann.

[13] Rangsaritratsamee, R., W. Ferrell, and M. Kurz: 2004, 'Dynamic rescheduling that simultaneously considers efficiency and stability'. *Computers and Industrial Engineering* **46**(1), 1–15.

[14] Smith, S.: 1994, 'OPIS: A Methodology and Architecture for Reactive Scheduling'. In: Zweben and Fox (eds.): *Intelligent Scheduling*. Morgan Kaufmann, pp. 29–66.

[15] Vieira, G., J. Hermann, and E. Lin: 2003, 'Rescheduling Manufacturing Systems: A Framework of Strategies, Policies, and Methods'. *Journal of Scheduling* **6**(1), 39–62.