Proceedings of the 2007 IEEE Symposium on Computational Intelligence in Scheduling (CI-Sched 2007)

Test Machine Scheduling and Optimization for z/OS

Matthew Kaplan Tracy Kimbrel Kevin Mckenzie Richard Prewitt
Maxim Sviridenko Clay Williams Cemal Yilmaz

Abstract

We describe a system for solving a complex scheduling problem faced by software product test organizations. Software testers need time on test machines with specific features and configurations to perform the test tasks assigned to them. There is a limited number of machines with any given configuration, and this makes the machines scarce resources. Deadlines are always short. Thus, testers must reserve time on machines. Managing a schedule for a large test organization is a difficult task to perform manually. Requirements change frequently, making the task even more onerous, yet scheduling is done by hand in most teams. Our scheduling system is able to take into account the many and varied constraints and preferences that a team of human users inevitably has.

1 Introduction

Software testing in large enterprise development centers is complex and expensive. Large test organizations, responsible for simultaneously testing numerous products, face serious difficulty scheduling testing tasks in a way that satisfies test configuration requirements, resource limitations imposed by system availability, and capacity of physical test resources. Inefficient scheduling of test runs can lead to underutilization of both human and physical resources, which can, in turn, lead to disruption of the software development cycle, premature termination of the test phase, and/or deferred delivery dates. Costs associated with poor utilization of test resources can easily run into the millions of dollars for large software systems. Scheduling test activities by hand is tedious and more importantly, it is likely to result in low-quality schedules. Yet the problem of automating scheduling to optimize resource utilization has received little attention relative to other parts of the testing process.

This paper describes a system that provides an automated solution to obtaining quality schedules. The system, which is being developed for IBM's enterprise operating system test center, collects information from test-shop personnel about test resource features and availability, testing tasks, and tester preferences and constraints. The system reformulates this testing information as a system of constraints, from which an optimizing scheduling engine computes efficient schedules during requested time periods. In order to be readily available for use by testers and test administrators, and to maintain an extendable bank of test information, the system is implemented as a J2EE-based web application, with information stored in a relational database.

Our solution was motivated by the complexity of effectively scheduling tests for z/OS development. z/OS is the operating system for IBM's eServer zSeries mainframe computer line. z/OS is a very large and complex software system, providing a comprehensive and diverse application execution environment. It must meet highly demanding performance requirements while remaining up-to-date with respect to the latest open and industry software technologies and standards. Above all, z/OS must meet stringent reliability requirements. More thorough descriptions of the z/OS can be found at [8].

In the remainder of this section we briefly describe the data-collection features of the system and enumerate requirements of the various participants in the scheduling problem. We also briefly touch on related research. Section 2 describes the mathematical model used to formalize the scheduling problem, which serves as input to the optimizing scheduling algorithm presented in Section 3. We report on the system's performance in Section 4, followed by directions for future work in Section 5.

M. Kaplan, T. Kimbrel, M. Sviridenko, C. Williams and C. Yilmaz are with the IBM T.J. Watson Research Center, Yorktown Heights, NY, 10598, USA (email: {mmk,kimbrel,sviri,clayw,cyilmaz}@us.ibm.com)

K. McKenzie and R. Prewitt are with the IBM Systems & Technology Group, Poughkeepsie, NY, 12601, USA. (email: {kmckenzi,prewitt}@us.ibm.com)

1-4244-0704-4/07/$20.00 ©2007 IEEE

1.1 Resources, Tasks, Testers, and Administrators

We define a testing problem in terms of a set of testing *tasks*, a set of computational *resources* on which tests can be run, and a set of *testers* who have specific responsibilities for running the testing tasks. In addition there are *test administrators*, who are involved in decision-making regarding the definition of the above-mentioned entities and control of the test-generation process.

These elements have the following properties:

- A resource, or test system, is characterized by its *configuration*, and an *availability schedule*. Configuration information could include number and types of processors, memory and network information, operating system version, installed software, and location. For our present purposes, though, it is sufficient to consider a configuration as defined in relation to tasks' constraints; administrators and/or testers specify, for each task, which resources are compatible with it. A unique *resource identifier* with associated descriptive information serves this purpose. Machines may be shut down for maintenance, upgrades, etc.. Thus, availability schedules can be specified to define when machines are available for use in testing.

- A task, also known as a test session, is characterized by: its *length*, i.e., the number of contiguous test shifts required for the task to complete; a description of which *task resources* are suitable for test operation, specified as a set of resource identifiers; and by a group of *task testers*, charged with running the task, including a specified *owner* who has primary responsibility for the task. Most tasks are considered *disruptive*, meaning that they require dedicated use of the test resource. Non-disruptive tests are able to run simultaneously with other non-disruptive tasks on a resource. Each task is designated *normal* or *priority*; priority tasks are to be scheduled in preference over non-priority tasks. Finally, some tasks may depend on the prior completion of other tasks; each task has a (possibly empty) list of *predecessor* tasks.

- Good utilization of resources often demands that testing be performed at inconvenient hours. Although as a matter of necessity testers' schedules are driven by resource availability and testing demands, there are limits on what can reasonably be expected of test employees. Limiting policies applying to all testers are imposed on consecutive shift assignments. In addition, testers can specify *scheduling preferences* that relate to their lives outside the workplace. Preferences can be positive, i.e., preferred times for being assigned testing tasks; or negative, i.e., times that the tester would like to block off from testing assignments.

  Preferences are not binding and, while the scheduler attempts to assign testers to their preferred slots, this cannot be guaranteed. To encourage responsible use of negative preferences, quotas can be established to limit the number of negative preferences that may be assigned to each tester during a given period.

1.2 User Interface

Users of this system operate in large, potentially distributed test organizations. Administrators need to monitor test assignments and testers need to provide their preferences in a uniform fashion even when distance and differing time zones can limit personal contact. Furthermore, the scheduling task inherently takes individual inputs from the various parties (administrators and testers), so it make sense that these parties be able to provide their inputs independently. These requirements of distribution, asynchronicity, and user autonomy, with all information collected into a single repository for scheduler use, are well served by an enterprise web application. Our pilot implementation is constructed using J2EE (Java 2 Platform, Enterprise Edition) with information stored in a relational database.

Users identify and authenticate themselves via an enterprise-wide login ID/password repository. Based on their login IDs, users are recognized as administrators or testers and appropriate sets of web pages and navigation paths are made available. Testers are able to work with their own preferences, while administrators can inspect information input by individual testers, selectively accept requested preferences, and initiate scheduling.

Tester input is relatively simple, consisting of a tabbed panel for specifying two classes of tester-relevant input: the tester's profile containing, e.g., e-mail address, phone number and office location; and the tester's time preferences. The administration interface is somewhat more complex, organized in a two-dimensional tabbed structure with primary tabs selecting scheduling entity (resource, task, tester, scheduling period) appearing vertically, while role-dependent horizontal tabs divide the information into coherent subsets, such as profile information and pref-

erences in the case of testers. For each role, the upper part of the page contains a selection panel. Thus, after selecting `Resources` from the vertical primary tab panel, the resources panel is displayed, containing a selection list of available tasks on top, and a horizontally-tabbed editing panel below. Administrators select a role, then a role instance and information category, to access the edit panel for the desired information. Using this interface, administrators can define new tasks, testers, resources and scheduling periods, as well as examine and/or modify their properties.

To generate a schedule, an administrator selects a scheduling period, optionally selects tester preferences to accept or reject for this scheduling run, and then initiates scheduling. The system generates the schedule by selecting information relevant to the scheduling period from the repository and passing it as input to our optimizing scheduler, which is described in later sections and is the main focus of this paper. The resulting schedule assigns tasks, with associated testers, to resources at designated dates and shifts. This schedule is displayed for inspection by the administrator, and can be distributed electronically to relevant testers.

All scheduling data is stored in the repository, making it a simple matter to regenerate a previous schedule if the need arises.

1.3 Related Literature

An introduction to the z/OS operating system and the hardware architecture can be found in [7]. Background on z/OS testing (challenges, methodology) can be found in [8].

General background on scheduling models and methodologies can be found in [4]. Perhaps the most closely related previously studied problem is known as *timetabling* [6]. In a typical application, classrooms must be allocated to classes, satisfying (non)-concurrency constraints similar to our "sliding window" constraints. However, classrooms (which correspond to our machines) are allocated exclusively; i.e., there are no classes corresponding to our "non-disruptive" jobs.

More sophisticated local search techniques than our simple method have been applied to timetabling and similar problems. These include *simulated annealing*, *tabu search*, and *genetic algorithms*; see, for example, [1].

We omit a more detailed discussion of related problems and methods in this extended abstract.

2 Mathematical Model

Summarizing the requirements for our scheduler, we have the following constraints and preferences.

- Testing must be completed by a set date in order to meet product deadlines.
- Some tasks may be more critical than others; for instance, a development organization may be waiting on the results.
- Different machines have different features (number of processors, memory capacity, network communication capabilities, etc.), so a workload may require a member of a certain subset of machines.
- Some *disruptive* test workloads require exclusive access to a machine; others may share machines with other workloads.
- There may be precedence constraints between tasks.
- Machines may be unavailable for scheduled maintenance activities.
- A task may require (a specific set of) several people to be present. Tasks requiring the same person or persons to be present must be separated in time; i.e, testers cannot work round-the-clock.
- Testers have preferred times at which they would like to be able to do their testing (typically the same for most testers: daytime shifts Monday-Friday, of course). Testers also have times at which they do not want to do testing (usually overnight and weekend shifts).

We make these requirements concrete and model our scheduling problem formally as described in the following subsections.

2.1 Enviroment

We are given a set of test tasks (jobs) $\mathcal{J} = \{J_1, \ldots, J_n\}$, a set of machines $\mathcal{M} = \{M_1, \ldots, M_m\}$ on which the jobs from the set $\mathcal{J}$ will be processed, and a set of testers $\mathcal{U} = \{U_1, \ldots, U_k\}$.

Each job $J_j \in \mathcal{J}$ has associated processing time p_j and a set of possible machines $\{M_1, \ldots, M_{m_j}\}$ on which job J_j can be processed.

Let F be a forest representing precedence constraints for the set of jobs $\mathcal{J}$, i.e., $\mathcal{J}$ is the vertex set of F and each directed edge (i, j) represents the constraint that job J_j cannot start before job J_i finishes its processing.

Each job is either *disruptive* or *non-disruptive*. Non-disruptive jobs can be processed on the same machine at the same time but each disruptive job requires the exclusive access to a machine. Let $\mathcal{D}$ be the set of disruptive jobs and $\mathcal{J} \setminus \mathcal{D}$ be the set of non-disruptive jobs.

Every job J_j has a priority $PR_j \in \{0, 1\}$, i.e. job J_j is either regular ($PR_j = 0$) or it has high priority ($PR_j = 1$) and must be scheduled even if it decreases the quality of the schedule for regular jobs.

There is a set of testers $S_j = \{U_{j_1} \ldots, U_{j_s}\}$ associated with each job. The primary constraint associated with testers is the so-called *sliding window constraint*: for every pair of jobs (tests) J_j and J_i such that $S_j \cap S_i \neq \emptyset$, J_i must be scheduled to start at least B time periods after the completion of J_j or vice versa. In other words, each tester must have B periods off between tests. In our initial application, schedule granularity is such that a unit of time corresponds to an 8-hour shift, and B is 2, or 16 hours.

It will be convenient to define the undirected graph $G = (\mathcal{J}, E)$ of incompatibility constraints. We define the edge set E as follows: if $S_j \cap S_i \neq \emptyset$ then $(J_j, J_i) \in E$. The graph G is a convenient way to encode the sliding window constraint. If there is an edge $(J_j, J_i) \in E$, then the processing of jobs J_j and J_k must be separated by at least B time periods.

Finally, each machine $M_i, i = 1, \ldots, m$ has a set of non-availability time intervals associated with it. During these intervals the machine M_i cannot be used for processing of any task.

2.2 Positive and Negative Preferences

For every job we have a set of positive preferences: periods in which the associated team of testers responsible for it would like the job to be processed. For each job we are given a set of time windows $[t_1, t_1 + p_j]$, $[t_2, t_2 + p_j], \ldots, [t_q, t_q + p_j]$. Analogously, for each job we have a set of negative preferences, times at which team would not like the job to be processed. For each job we are given a set of time windows $[t'_1, t'_1 + p_j]$, $[t'_2, t'_2 + p_j], \ldots, [t'_w, t'_w + p_j]$. All remaining time intervals are considered neutral.

Ideally we would like to assign each job to be processed in a time interval preferred by the team of testers. Unfortunately, we may have conflicts of preferences for different tester teams. Our goal is to assign jobs to machines to maximize the number of positive preferences that are satisfied and minimize the number of negative preferences that are violated.

Additionally we would like our schedule to be envy-free or fair in some sense. We do not want a schedule in which some testers get all their positive preferences satisfied and some get all their negative preferences violated. There are many natural ways to incorporate such fairness constraints into the objective function. We handle this as described in the next section.

2.3 Objective Function

Since we use the local search technique for finding the schedule, we do not need to define the value of a schedule explicitly. Given two schedules we need only to define which schedule is better. This is a very general mechanism that can be tailored as desired by different organizations with different needs. In this section we describe the notion of "better than" used in our present application.

Some constraints are absolute; thus we never consider schedules that violate them. They are

- job/machine compatibility;
- precedence;
- "sliding window" constraints as described previously;
- exclusivity for disruptive jobs.

Other constraints are softer: a low-priority job may be cancelled if necessary, and user preferences can be relaxed.

Our first goal is to schedule as many priority jobs as possible, and then to schedule as many regular jobs as possible. Next we try to minimize non-preferred time slot assignments in a fair way. We first try to bring all testers down to a bound on maximum number of bad assignments. Next we try to minimize the number of testers with this number of bad assignments, and finally to minimize the total number of bad assignments. Similarly, we try to raise the minimum number of good assignments over all testers, then minimize the number of testers at this minimum level, and finally to maximize the total number of good assignments. However, if any tester receives all good time slots, we do not consider that tester when determining the minimum number of good slots for any tester. A single tester with no jobs, for instance, would cause this value to be 0 and render it meaningless.

For each job, one tester is designated as the "main tester" for the job. For purposes of determining schedule fairness as described later, we count only jobs for which a tester is the main tester when we try to even out the numbers of good and bad assignments across the testers. Let $bad(u)$ and $good(u)$ denote the numbers of time slots assigned to jobs for which u is the main tester and u has negative and

positive preferences, respectively. However, if all of a tester's assignments are preferred, then $good(u) = \infty$ for that tester.

Thus we compare two schedules based on the following measures, in this order. The first one for which one schedule is better than another determines the winner.

1. Number of priority jobs scheduled. (Larger is better.)
2. Number of non-priority jobs scheduled. (Larger is better.)
3. Maximum number *maxbad* of negative-preference slots assigned to any tester. (Smaller is better.)
4. Number of testers u such that $bad(u) = maxbad$. (Smaller is better.)
5. Total number of negative-preference assignments. (Smaller is better.)
6. Minimum number *mingood* of preferred slots assigned to any tester that does not get all jobs assigned to preferred slots. (Larger is better.)
7. Number of testers u such that $good(u) = mingood$. (Smaller is better.)
8. Total number of preferred assignments. (Larger is better.)

2.4 Practical Simplifications

The problem defined above is very difficult in its full generality. It includes many classical machine scheduling problems that are intractable both from practical and theoretical viewpoints. In particular our problem includes job shop scheduling, parallel machine scheduling with precedence constraints, graph coloring, and other problems. Fortunately, practical instances of our problem are much easier than worst case.

The first simplification is that most jobs have unit processing time; even when testers have long jobs that might require several time periods, they prefer to split the jobs into manageable pieces of eight hours each (one time period). On the other hand, sometimes we have jobs that run for days, but in real life such jobs can be scheduled during weekends, and thus they tend to have preferred time periods that are undesirable for most other teams of testers.

The second simplification is that real life precedence constraints are very easy. In the test instances we obtained from the customer there were no precedence relations between jobs. The customer expects testers to use this feature in the future, but the expectation is that simple chain precedence constraints are enough to model all real life instances.

Another simplification is that the incompatibility graph G has a simple structure. In most cases there is only one tester associated with a job. This means that graph G consists mainly of disjoint cliques. Some jobs have more than one tester assigned, but these jobs tend to be jobs with long processing times and appear rarely in the real life instances.

Finally, the system in general is underutilized. The total job processing time is less than half of the number of machines multiplied by the planning horizon length.

Our scheduler is capable of producing a schedule on arbitrary problem instances, but we would not expect it to perform well on the hardest ones. These simplifications make the problem tractable in practice. We describe below in section 4 our scheduler's performance on both a real problem instance obtained from our customer and on a more challenging instance generated randomly.

3 Algorithm

As was mentioned before, we use the Local Search Framework to design an algorithm for finding an approximate solution to our problem.

3.1 Finding a Feasible Solution

On the first step we are trying to find a feasible schedule of jobs within the planning horizon H. The primary goal of the first step is to find a schedule that assigns as many jobs as possible without considering more sophisticated goals such as fairness.

More precisely, we fix two nonnegative integers $NegBound$ and $PosBound$, where $NegBound$ is an upper bound on the total number of negative preferences for each tester that may be violated and $PosBound$ is a lower bound on the number of positive preferences for each tester that must be satisfied. If at some point a tester gets more than $PosBound$ of its preferences satisfied the algorithm treats all remaining positive preferences as neutral. By using the greedy algorithm for fixed $NegBound$ and $PosBound$ and enumerating over possible values for these numbers we find the minimal $NegBound$ for which the algorithm finds the feasible solution. For this value of $NegBound$ we find the maximal value of $PosBound$ such that each tester gets at least $PosBound$ of positive preferences satisfied.

We now describe the greedy algorithm for fixed $NegBound$ and $PosBound$. The greedy algorithm builds the schedule time period by time period. Let $\tau \in 1, \ldots, H$ be the current time period. First the algorithm finds the set of unscheduled jobs with either no predecessors or all predecessors already completed by the time τ. Let $\mathcal{J}_1$ be this set of jobs.

Next we further decrease the set of jobs $\mathcal{J}_1$ to incorporate the sliding window constraint. For each job $J_j \in \mathcal{J}_1$ if there is another job J_i that completes at time $\tau - 1$ or $\tau - 2$ and there is an edge $(J_i, J_j) \in E$ then we delete the job J_j from $\mathcal{J}_1$. Let $\mathcal{J}_2 \subseteq \mathcal{J}_1$ be the set of remaining jobs. Now we can schedule an arbitrary subset of jobs from $\mathcal{J}_2$ during the time period τ without violating the precedence and sliding window constraints.

On the next step we associate a reward R_{js} for starting job $J_j \in \mathcal{J}_2$ on machine M_s at time τ.

1. If machine M_s is incompatible with job J_s, i.e. J_s cannot be processed on M_s, then $R_{js} = 0$.
2. If there is a disruptive job running on machine M_s at time τ (which must have starting time $< \tau$) then $R_{js} = 0$ for all jobs $J_j \in \mathcal{J}_2$.
3. If there is a non-disruptive job running on machine M_s at time τ then $R_{js} = 0$ for all disruptive jobs $J_j \in \mathcal{J}_2$.
4. If the main tester associated with job J_j has $\geq NegBound$ negative preferences violated by the current schedule and the time period τ belongs to the set of negative preferences for J_j, then $R_{js} = 0$ for all machines M_s.
5. If the interval $[\tau, \tau + p_j]$ overlaps with machine non-availability intervals then then $R_{js} = 0$ for such machines M_s.
6. Otherwise, the reward R_{js} defined as follows. Let $h(j)$ be the height of the job in the precedence constraint forest F, i.e. the length of the longest path F from J_j to a leaf (a job that does not have any successors in the precedence relation).
   (a) $R_{js} = 100(1 + 10PR_j)(1 + h(j))$ if time period τ belongs to the set of positive preferences for J_j and the main tester has at most $PosBound$ of positive preferences satisfied by the current schedule.
   (b) $R_{js} = 10(1 + 10PR_j)(1 + h(j))$ if time period τ is neutral for for J_j or the main tester has at least $PosBound$ of its positive preferences satisfied by the current schedule.
   (c) $R_{js} = (1+10PR_j)(1+h(j))$ if time period τ is a negative preference for J_j and the total number of negative preferences violated by the current schedule for the main tester is at most $NegBound$.

Note that rewards are defined somewhat arbitrarily and we could choose many other functions. The important property of the reward function is that it is monotone with respect to user preferences.

After computing the reward for each job and machine combination, we solve the weighted maximum matching problem (also known as the *assignment problem*; see, for example, Ajuha et al. [2]) of pairing jobs with machines by a straightforward greedy algorithm. A job may be started on a machine only if it has a positive reward. After matching as many jobs as we can, we move on to the next time step.

3.2 Local Improvements

Our Local Search algorithm utilizes two types of local improvements.

1. If it is possible to add a currently unscheduled job to the schedule without moving other jobs and the new schedule is better than the old one, then do so. (Note that with our particular better-than relation, the new schedule will always be better, but for other relations it might not be).
2. Take any two jobs j and k such that j is already scheduled; k may or may not be scheduled. Start k at the starting time of j in the old schedule and on the same machine, and start j in an arbitrary feasible location. If any such move is feasible and improves the schedule, accept it.

We search for the second type of improvement only if the first type cannot improve the current schedule.

4 Experimental performance

We made no attempt to minimize the running time of our scheduler. Running times on all instances described below were modest and we will not discuss them further. Instead, we focus on schedule quality.

In our z/OS application, test schedules are determined on a weekly basis, and time slots are 8-hour shifts. Thus our time horizon is 21 time slots. In simple test cases obtained from our users, there were 4 machines and 17 testers typically with a total of 21 jobs. Our scheduler found ideal schedules in these cases: all testers received preferred slots for all their

jobs. These instances corresponded to slow weeks in our users' environment. In the future, we expect much more demanding problems.

We generated more challenging test cases randomly, and we report here some of these. We assumed the same 21-shift work week as before, but increased the number of machines to 10. We randomly chose down times for machines during overnight and weekend shifts. For each of these 11 time units, each machine was designated unavailable with probability 0.25. Thus the expected number of available $(machine, time)$ slots is $10 \cdot 21 - 10 \cdot 11 \cdot 0.25 = 182.5$.

We assigned 1 to 4 randomly chosen testers to each job, with probability 0.8, 0.1, 0.05, and 0.05, respectively; i.e., with probability 0.8 a job has 1 tester, etc. Each job is disruptive with probability 0.9. Each job has length equal to 1 shift with probability 0.85, 2 with probability 0.1, and 3 with probability 0.05. Each job is assigned between 0 and 3 predecessor jobs among those with lower indices (thus ensuring acyclicity). The probabilities, from 0 predecessors to 3, are 0.8, 0.1, 0.05, and 0.05. Each job/machine pair is compatible with probability 0.8. Finally, for each job, each of the 5 weekday shifts is designated "good" (i.e., a positive preference) with probability 0.9, and each of the 11 overnight and weekend shifts is designated "bad" with probability 0.95.

We describe the scheduler's performance at two points as the number of jobs increases. Around 75 jobs, all jobs are scheduled but it becomes difficult to avoid some bad assignments. Thus the interesting measures of performance, according to our objective criteria, are the numbers of good and bad assignments and the degree to which these are spread among users for fairness. Table 1 gives these values on three randomly generated instances. The last column, labeled "U.B. good," represents an upper bound on the number of preferred assignments based on the number of weekday slots (50) and the number of non-disruptive jobs; i.e., if there are n non-disruptive jobs, at most $50+n-1$ jobs can be placed in preferred slots. (There may be tighter bounds in a given instance due to other constraints.)

From the table we see that the scheduler performs well regarding fairness, i.e., even distribution of preferred and non-preferred assignments, with one exception. In the case of an unlucky user receiving 2 non-preferred assignments, it turns out that this user's 5 jobs with a total of 8 time units cannot be scheduled with fewer bad assignments, given the sliding window constraints. We also see that the scheduler is always within 2 of the maximum possible number of preferred assignments.

We found that with 140 jobs or fewer, the algorithm scheduled nearly all jobs. Most of the exceptions could not be scheduled even in an optimal schedule due to sliding window constraints; i.e., some unlucky tester was assigned too many jobs. With 160 or more jobs, nearly all available machine/time slots were filled, so there is no point in adding more jobs. Thus we focus on instances with 150 jobs (and 75 testers).

max bad	#max bad	total bad	
1	3	3	
2	1	3	
0	75	0	
min good	#min good	total good	U.B. good
0	3	53	54
0	1	50	52
1	17	54	56

Table 1: Scheduling performance on random instances of 75 jobs each

Since the first scheduling criterion is the number of priority jobs scheduled, we examined the behavior of the algorithm as the expected number of priority jobs is increased from 0 to 80%. (Note 100% is in effect no different from 0.) We present the results of three test cases in Table 2 below.

We make the following observations and conclusions from the experimental data about the primary and secondary scheduling criteria. Recall that these are the numbers of priority jobs scheduled and all jobs scheduled, respectively. For lack of space we omit discussion of the other criteria.

- In all cases with 60% or fewer priority jobs, all priority jobs are scheduled and thus the schedule is optimal with respect to our first criterion.
- Machine utilization is in general very high, and never less than 90%. Simultaneously, the number of priority jobs scheduled is always more than 97%, and the number of all jobs scheduled is always more than 95%. We believe that these values are quite good with respect to our users' needs.
- *When all priority jobs are scheduled*, as we go from left to right across the table, we would expect the total number of jobs scheduled to be non-increasing. This is because a solution to an instance on the right is valid for one to its left and is optimal with respect to priority jobs in this less-constrained instance. In some cases the more-constrained instance has more jobs scheduled than the less-constrained instance. This

# priority jobs	0%	20%	40%	60%	80%
jobs scheduled	146	147	147	145	146
pr. jobs scheduled	n/a	45/45	73/73	94/95	114/115
utilization	92.7%	91.0%	91.5%	91.5%	92.7%
all jobs scheduled	143	145	144	143	144
pr. jobs scheduled	n/a	35/35	53/53	82/82	117/120
utilization	90.7%	95.1%	96.7%	96.7%	92.9%
all jobs scheduled	149	148	147	147	145
pr. jobs scheduled	n/a	34/34	59/59	92/92	117/119
utilization	94.3%	93.1%	93.1%	92.5%	91.4%

Table 2: Scheduling performance on random instances of 150 jobs each

means that the algorithm sometimes gets stuck in a local maximum. The worst difference is 2.

- These random instances should be much harder than our users' real instances. The algorithm's good performance on these instances leads us to project excellent performance on real instances of this practical scheduling problem.

5 Conclusion and further work

We described the model and the algorithm we used to solve a real-life test scheduling problem for the z/OS test organization. The next phase of this project is to generalize model and the algorithm to handle even more complicated scheduling problems. In particular:

1. Some jobs could require more than one machine at the same time. Such problems are usually referred as scheduling with multiprocessor tasks in the optimization literature.

2. Currently we schedule test teams project-by-project and assume that resources do not overlap across projects. This assumption is not necessarily true in real-life. Some machines are used by teams from different projects. The main problem in handling this generalization is that the problem size increases dramatically and the running time of our algorithms could become an issue.

3. Continual Optimization [3] is an approach to enable a system to react to events in real life in an online fashion. To handle such events we need to design an algorithm that can reschedule some jobs or insert new ones without modifying the rest of the schedule.

4. Currently we schedule on a weekly basis. Increasing the planning horizon could increase bottleneck resource utilization and improve planning, but on the other hand it would increase the running time of our algorithm.

5. The added complexity of these extensions may require different and larger local search neighborhoods. We may need to employ more sophisticated search methods, such as those mentioned in Section 1.3.

References

[1] *Local Search in Combinatorial Optimization.* Edited by E. Aarts and J. Lenstra. Reprint of the 1997 original [Wiley, Chichester; MR1458630]. Princeton University Press, Princeton, NJ, 2003.

[2] R. Ahuja, T. Magnanti, and J. Orlin. *Network Flows, Prentice Hall*, Englewood Cliffs, NJ.

[3] O. Gunluk O., T. Kimbrel, T., L. Ladanyi, B. Schieber, and G. Sorkin. Vehicle Routing for Sedan Service. *Transportation Science*, Vol. 40, No. 3, August 2006, pp. 331-326.

[4] *Handbook of scheduling: Algorithms, Models, and Performance Analysis.* Edited by Joseph Y.-T. Leung. Chapman & Hall/CRC Computer and Information Science Series. Chapman & Hall/CRC, Boca Raton, FL, 2004.

[5] S. Loveland, G. Miller, R. Prewitt and M. Shannon, Testing z/OS: The premier operating system for IBM's zSeries server, *IBM Systems Journal*, Volume 41, Number 1, 2002, pp. 55-73.

[6] S. Petrovic and E. Burke. University Timetabling. In *Handbook of scheduling. Algorithms, models, and Performance Analysis*, edited by Joseph Y.-T. Leung. Chapman & Hall/CRC Computer and Information Science Series. Chapman & Hall/CRC, Boca Raton, FL, 2004, chap. 45.

[7] ABCs of z/OS System Programming, *IBM Redbooks*, 2006. Available at http://www.redbooks.ibm.com

[8] www.ibm.com/servers/s390/os390/