A Comparative Study of Online Scheduling Algorithms for Networks of Workstations

Olaf Arndt [a], Bernd Freisleben [a], Thilo Kielmann [b,*] and Frank Thilo [a]

[a] *Dept. of Electrical Engineering and Computer Science, University of Siegen. Germany,*
E-mail: {arndt,freisleb,thilo}@informatik.uni-siegen.de

[b] *Dept. of Mathematics and Computer Science, Vrije Universiteit, Amsterdam, The Netherlands,*
E-mail: kielmann@cs.vu.nl

Networks of workstations offer large amounts of unused processing time. Resource management systems are able to exploit this computing capacity by assigning compute-intensive tasks to idle workstations. To avoid interferences between multiple, concurrently running applications, such resource management systems have to schedule application jobs carefully. Continuously arriving jobs and dynamically changing amounts of available CPU capacity make traditional scheduling algorithms difficult to apply in workstation networks. *Online scheduling algorithms* promise better results by adapting schedules to changing situations. This paper compares six online scheduling algorithms by simulating several workload scenarios. Based on the insights gained by simulation, the three online scheduling algorithms performing best were implemented in the WINNER resource management system. Experiments conducted with WINNER in a real workstation network confirm the simulation results obtained.

1. Introduction

Networks of workstations are by now ubiquitous general purpose computing platforms. Since the console users of workstations in a network typically do not fully utilize the processing capabilities of their machines (e.g. while editing text, reading mail, browsing the Web, or being physically absent), the idle time of workstations is frequently as high as 95%. On the other hand, there is a high demand for computing power driven by applications from fields like simulation and optimization [29].

Software packages that assign such compute-intensive applications to idle workstations are known as *resource management systems* (RMS) [4]. Such systems have to cope with two conflicting objectives. First, interactive tasks demand *minimal completion time* to satisfy human users sitting in front of a workstation. Second, non-interactive background tasks (sequential as well as parallel batch jobs) demand *maximal throughput* to complete as many jobs as possible within given time constraints.

Interactive tasks can be divided into two groups: (a) tasks generated by the workstation's console user, and (b) tasks scheduled by a RMS to remote machines. The tasks in group (a) have to be executed without interference by any tasks from other jobs, because otherwise users would not be willing to make their machines available to workstation pools controlled by a RMS. Under this restriction, a RMS can assign additional interactive jobs (such as compile or text formatting jobs) to fast workstations, making them accessible to owners of slower machines. Additionally, a RMS can execute batch jobs on fast or idle workstations to maximize system throughput (i.e. the efficient utilization of all resources) under the constraint of providing predictable individual completion time and fairness of execution among the tasks.

Unlike dedicated compute servers, such as workstation clusters (i.e. physically adjacent, interconnected workstations without monitors and keyboards), or symmetric multiprocessors (SMP) with shared memory, networks of workstations form heterogeneous collections of individual machines. They differ not only in CPU speeds, but also in their hardware architectures, operating systems, and application software. These restrictions split the available nodes into subsets able to service particular requests. Job placement decisions then happen by choosing nodes from members of such a subset. In general, the goals of resource management sys-

* Corresponding author

tems for such a heterogeneous workstation network environment are threefold:

1. The RMS has to keep track of the current state of the workstations in the network. For each machine, static resources like hardware architecture, computational speed, and main memory capacity have to be recorded. The current utilization of a machine and the occupancy by a console user must be monitored dynamically.
2. Interactive tasks have to be scheduled immediately on the currently best suited machine able to service a given request.
3. Batch jobs have to be scheduled in a way that follows the two objectives of maximizing throughput while minimizing individual job completion times.

Whenever the set of pending batch jobs exceeds the available computational resources, jobs have to be executed in a coordinated manner to avoid performance penalties caused by interferences between multiple concurrent jobs. Coordinated execution of batch jobs is called *scheduling*. Its goal is to make all available workstations share the total load by carefully assigning jobs to workstations.

With scheduling systems, jobs are entered into waiting queues after their creation, and the scheduler then selects jobs from these queues as resources become available. Settings in which all jobs are available before the scheduler begins planning their execution schedule are called *offline scheduling* problems. In contrast, settings where the scheduler must make a decision for presently arriving jobs while some other jobs are already running in the system are called *online scheduling* problems [34,35,40]. Since in typical workstation networks jobs are permanently created while other tasks are already running, and available CPU capacity changes dynamically due to interfering interactive jobs, traditional offline scheduling algorithms are hardly applicable.

Online scheduling has recently attracted the attention of several researchers. Theoretical results have already indicated the performance of online scheduling algorithms in terms of lower and upper bounds [34]. In contrast, the goal of this paper is to comparatively evaluate several online scheduling algorithms by studying their performance in practice. To get basic insights and to cover a large variety of scheduling scenarios in a timely manner, we developed a simulator for online scheduling algorithms. Based on our simulation results, we integrated the most suitable online scheduling algorithms into the WINNER RMS [1] and verified our findings in a real workstation network.

Since in this paper we focus on the behavior of online scheduling algorithms, we do not discuss issues like queue management, handling user priorities, and matching between job requests and available resources, although solutions to each of these problems were implemented in WINNER. Furthermore, we restrict our study to scheduling by placement of batch jobs onto available workstations in a network. Finer grained scheduling issues like task switching in operating-system kernels, or application-level load distribution (e.g. on the loop level) are beyond the scope of this particular study.

The paper is organized as follows. Section 2 discusses objective functions for online scheduling. In section 3, several online scheduling algorithms are compared by simulation. The WINNER resource management system along with its scheduling and runtime prediction algorithms is presented in Section 4. Section 5 evaluates the performance of WINNER's online scheduling algorithms in a real workstation network. Section 6 reviews related work. Section 7 concludes the paper and discusses areas for future research.

2. Online Scheduling

As already mentioned, there are two basic types of scheduling problems: *online* and *offline* scheduling [34]. In offline scheduling, the entire set of jobs to be scheduled including relevant information (in particular, the runtime of each job) is known before a scheduling decision is made. In online scheduling, a scheduling decision must be made as soon as one or more jobs are available to be started. There is no information about the arrival of additional jobs in the future, and the runtimes of the present jobs may or may not be known. In general, in both offline and online scheduling, preemption of jobs is optional. For the scope of this paper, however, preemption is not considered. Once a job has been assigned to a machine, it stays there until it is finished. This is a reasonable assumption for networks of workstations in which task migration is very costly (if at all possible) and easily outweighs the possible improvements on the schedule. Also due to the nature of workstation networks, we investigate centralized scheduling algorithms only which can be executed on a server machine with negligible overhead on the overall system performance.

Any solution to a scheduling problem aims to optimize a particular objective function. The most common objective functions are:

- *makespan* C_{max}: the difference between the finish time (i.e. the termination time) of the last job and the release time (i.e. the arrival time in the system) of the first job;
- *flowtime* $\sum F_i$: the sum (total flowtime) or the average (average flowtime) of the differences between the finish and release times of all jobs;
- *maximum waiting time (maximum lateness)* L_{max}: the maximum of the differences between the start times (i.e. the time when a job starts executing) and the release times of all jobs.

Most scheduling algorithms proposed in the literature try to minimize the makespan. The makespan measures the performance of the system in the sense that the entire set of jobs will be scheduled in such a way that it takes as little time as possible to finish all jobs, without focusing on any single job. Minimizing the flowtime means that the time each job is in the system should be as small as possible, such that jobs should be finished soon after they were released. The maximum waiting time measures the fairness of the corresponding algorithm: no job should be blocked by other jobs for a long time (especially by jobs with later release times).

Since we need a comparative value for the corresponding objective function, a scheduling algorithm is characterized by its *competitive ratio* σ [16,34]. An algorithm is called *σ-competitive* for a given objective function, if for each input instance the schedule is at most σ times larger than the optimal schedule.

There are different online scheduling paradigms discussed in the literature. We discuss the two most realistic models.

In the first online scheduling paradigm, jobs can either be started immediately upon arrival, or they may be delayed and started later (i.e. there is a queue). The running time of each job is unknown until the job is finished. If we assume that there is an offline σ-competitive algorithm (with respect to the makespan) where all jobs are given at time 0, Sgall proved that a 2σ-competitive online algorithm exists [34]. However, without knowledge of the running times of jobs, there is not much on which a reasonable scheduling decision can be based.

In the second online scheduling paradigm, jobs are arriving over time [30] and placed into a queue. Their runtimes are known when they arrive in the queue. It has been shown that there is a 2-competitive algorithm [35] (with respect to the makespan) for any online scheduling of jobs arriving over time. The lower bound of the competitive ratio of any online algorithm (with respect to the makespan) is 1.3473 [7] if no preemptions are allowed. If preemptions are allowed, there are $(1+\epsilon)$-competitive algorithms for any $\epsilon > 0$ [34]. A summary of lower and upper bound results for jobs arriving over time for several objective functions has been presented by Vestjens [40]. Throughout this paper, we will use such a setting. In an actual implementation, runtimes may either be user specified or automatically derived from historical data.

The theoretical analysis of online scheduling algorithms is concerned with proving upper and lower bounds and determining the competitive ratios for particular objective functions. In practice, however, bounds for particular objective functions are of limited value, since quite often either algorithms with a balanced behavior for different objective functions are needed, or in certain circumstances an algorithm with a high upper bound may be acceptable if this bound is reached only in a few exceptional cases. Furthermore, most theoretical analyses are based on assuming an environment with identical machines, exact running times, exclusive machine access, only sequential jobs etc. In a realistic environment, there are heterogeneous workstations running with different speeds, which usually depend on a variety of other factors, such as the current load. Quite often the set of jobs to be scheduled consists of sequential and parallel jobs (with or without exact running times).

Furthermore, in a resource management system for networks of workstations there is no single suitable objective function that should be minimized. Since in such a system jobs are arriving permanently, there is no closed set of jobs and thus no makespan according to the definition given above. Minimizing the flowtime comes closest to the goal of a resource management system, but since no job should be delayed for a very long time, and starvation of (parallel) jobs should be prevented, the maximum waiting time of each job must also be minimized in order to allow the system to be accepted by its users.

3. Comparison of Online Scheduling Algorithms

To investigate the characteristics of different online scheduling algorithms in practice, we performed a comparative study based on extensive experiments as an alternative to mathematical analysis.

The most realistic experimentation approach would be to implement different algorithms as part of the scheduling module of WINNER, create various job mixes of real applications and execute them on a workstation network. However, the completion time for running such job mixes makes it practically impossible to perform sufficient numbers of measurements. Furthermore, workload from other workstation users would prevent us from studying the behavior of our scheduling algorithms in isolation.

Instead, we implemented a simulator for online scheduling algorithms for sequential and parallel jobs in networks of workstations. This approach solves both of the above problems, because competing workload does not affect the results. Furthermore, simulations can be performed orders of magnitude faster compared to the execution of real jobs. Based on our simulation results, we implemented the best-suited algorithms as part of WINNER. We verified our findings with a few, carefully selected experiments performed in a real workstation network while isolating the network from other users. We describe those measurements in Section 5. In this section, we discuss the candidate online scheduling algorithms, we describe our simulation environment, and we present the findings of our simulation-based experiments.

3.1. Scheduling Algorithms

We will now present the properties of the scheduling algorithms implemented in our simulator.

- **FIFO:**
  First **I**n **F**irst **O**ut is the most obvious and simplest algorithm for online scheduling. All jobs are started in exactly the same order in which they arrive in the queue. There is no exception to this rule. When the next job in the sequence cannot be started, FIFO will *not* try to start any of the succeeding jobs.
  The main problem with FIFO is the possible presence of large gaps of unused machines when the next job to be scheduled is one requiring a large number of nodes. E.g. consider a situation where a single sequential job is running on one node while all other nodes are idle, and the next job in the waiting queue is a parallel job that requires all of the nodes. Since one machine is still busy, the parallel job cannot be started immediately, which in turn stops all other jobs in the waiting queue from getting started, although most of them would need fewer nodes than available. This causes many nodes to idle until the single machine job finally terminates, allowing the parallel job to be executed.
  The advantage of FIFO is that it can be easily and efficiently implemented, its behavior is highly predictable and its order of job execution is fair, i.e. no job can be delayed by jobs that are submitted at a later time.
- **FirstFit:**
  The FirstFit algorithm examines the waiting jobs in the order of their arrival in the queue and executes the first of these jobs for which enough machines are available. In contrast to FIFO, a single job cannot block all other jobs from becoming active. Thus, FirstFit will generally achieve a much higher node utilization. As long as there are jobs in the waiting queue which need only a few nodes, this algorithm can keep most of the machines busy.
  The drawback, however, is that parallel jobs requiring a large fraction of the nodes can be significantly delayed by smaller jobs. In case of an infinite stream of jobs requiring a small number of nodes, it is theoretically possible that a large parallel job will never be started. Another disadvantage is FirstFit's limited predictability, because newly submitted jobs can further delay the start of a job which has already waited for some time.
- **Random:**
  Random is similar to FirstFit, with the exception that the next job is picked randomly out of the jobs in the waiting queue. Thus, each job in the queue has the same chance of getting started at a given point of time regardless of its position within the queue. Obviously, compared to FirstFit it is even harder to predict when a given job will be scheduled, and therefore the maximum waiting time is in general higher.

- **LPT:**
  Largest **P**rocessing **T**ime First simply selects the job requesting the longest processing time [23,34]. If this job cannot be started, other jobs will have to wait as well. The idea behind this scheme is to minimize the makespan, mainly by avoiding situations where a single long-running job becomes the dominating factor when started close to the end of the schedule. As a drawback, LPT increases the average flowtime, because many short jobs are delayed significantly.
  It is difficult to apply this algorithm to a scheduling problem when the available nodes have different (or – even worse – varying) processing capacity. This is due to the fact that in this case a job's runtime depends on both the amount of computations it performs and the speed of the nodes it will be scheduled on. The latter factor, however, will vary depending on the job's degree of parallelism and the current state of the nodes. The LPT algorithm studied in this paper just compares the jobs' static processing time needs.
- **SPT:**
  Similar to LPT, the **S**hortest **P**rocessing **T**ime First algorithm uses job processing times as the criterion for deciding which job to start. However, the job ordering is the exact opposite, i.e. the job having the lowest processing time is started first.
  This procedure aims to minimize the average waiting time of all jobs, since short running jobs are started near the beginning of the schedule and will not be delayed by those with higher runtimes. Accordingly, this results in a higher makespan as the most expensive jobs are started last, and there is a high probability of one of them running exclusively for a significant fraction of time, thus extending the makespan. Again, the SPT algorithm investigated in this paper uses the static processing time attribute of each job and blocks all waiting jobs if the selected job cannot be started due to a shortage of free nodes.
- **FIFO with Backfilling:**
  The Backfilling FIFO is a modified FIFO algorithm. FIFO's main problem is its low utilization of the computing nodes due to the possible occurrence of large time gaps with several nodes unused whenever a parallel job requiring a large number of nodes has to be spawned. The idea of backfilling is to place short-running jobs into this gap without causing an additional delay of the large parallel job. This is basically done by calculating the size of the remaining gap, i.e. the time difference between the predicted starting time of the parallel job, and the current time. This gap will be consecutively compared to the runtime of each waiting job. If a job's runtime is smaller than the size of the gap, the job fits into the gap and will be started right away.
  The Backfilling FIFO tries to get the best of both worlds: a good utilization of all nodes and thus a small makespan like FirstFit, and the fairness, good predictability (a job might be started *earlier* than expected, but not later) and small maximum waiting time of FIFO. However, to be able to backfill jobs successfully, the predicted runtimes must be quite accurate, since otherwise the parallel job causing the gap will be delayed by the backfilled ones.

3.2. The Online Scheduling Simulator

The simulator models a set of nodes (workstations) with possibly different processing performance, a job mix of several independent jobs and several scheduling algorithms. Each job is submitted to the waiting queue at a different point of (simulation) time. All jobs in the waiting queue are subject to the scheduling algorithm's job selection function and may hence be scheduled on a subset of the nodes. Once a node is assigned to a job, it is marked as busy and not available to process any other job until the active job has terminated.

In the scheduling model implemented in the simulator, the scheduling algorithm itself has the sole purpose of deciding which job - if any - to schedule at a given point of time. The actual placement of this job onto the corresponding nodes is the same for all algorithms: A *greedy scheme* is used that always assigns the selected job to the currently fastest idle node. This scheme is quite natural and yields proper results. There are some situations, however, where ignoring a very slow workstation in favor of a faster machine to become available soon may result in a shorter makespan, potentially at the price of increased waiting times.

The simulator's outer loop defines a number of iterations. In each iteration, a fixed number of jobs is randomly created, constituting the job mix for this run. Each of these jobs is defined by the following attributes:

- **queueing time:**

The queueing time is the point of time at which the job is submitted to the queue; prior to this point the job is not known to the scheduling algorithm.

- **degree of parallelism:**
  For sequential jobs, the degree of parallelism is set to 1. For parallel jobs, the degree of parallelism is a fixed number smaller or equal to the total number of nodes and determines the number of nodes the job will occupy. A job cannot be started if its degree of parallelism exceeds the number of available nodes.
- **processing time:**
  The processing time is the total CPU time the job will consume on a workstation with a normalized speed. For parallel jobs, the processing time is accumulated over all processes. The runtime of the job is the quotient of this processing time and the speed of the node(s) it is scheduled on. Thus, we assume that the jobs are purely CPU bound, i.e. no significant fraction of time is used for waiting on I/O events, and there is no additional load on the workstations. Furthermore, any job startup cost is neglected.

Once the job mix has been generated, the simulation itself is started using the first scheduling algorithm. Queueing and process termination events are handled by managing the state of the nodes and activating the scheduling algorithm for each event. Makespan, average flowtime and maximum waiting time are calculated after all jobs were successfully scheduled and have completed their simulated execution.

Next, the simulation is repeated with another algorithm. After all algorithms have scheduled the same given job mix, the next iteration is started by generating a new job mix. When all iterations are done, an average is calculated for each objective function and algorithm. The simulator can be controlled by the following parameters:

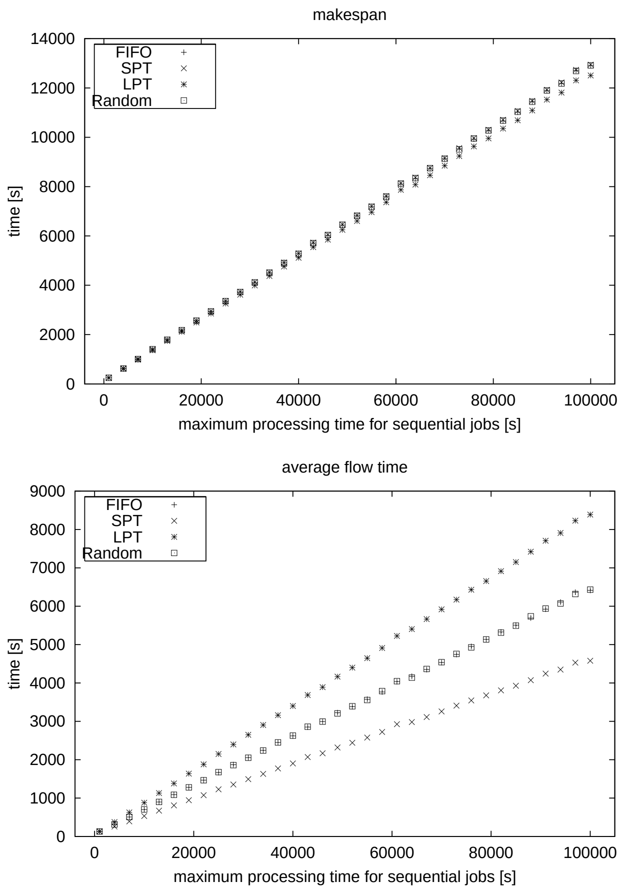
- **number of iterations:**
  The total number of iterations for the simulation; larger numbers of iterations simply yield more accurate results, since the inherent randomness causes significant fluctuations between single runs.
- **number of jobs (n):**
  The number of jobs for each job mix.
- **set of nodes:**
  The number and relative speed of the computing nodes.
- **percentage of sequential jobs (P_s):**
  The ratio between the number of sequential and parallel jobs.
- **percentage of large parallel jobs (P_{lp}):**
  The ratio between the number of small and large parallel jobs; a large parallel job is defined to require at least 50% of all nodes.
- **timespan of job creation ($T_{c,max}$):**
  The range of time during which the jobs are created, i.e. submitted to the waiting queue; for offline scheduling, $T_{c,max} = 0$, since all jobs are created at the same time.
- **sequential job processing time ($t_{s,min}$ / $t_{s,max}$):**
  The minimum and maximum processing time that sequential jobs consume; the actual processing time of each sequential job is uniformly distributed within these limits.
- **parallel job processing time ($t_{p,min}$ / $t_{p,max}$):**
  The minimum and maximum processing time for parallel jobs; the actual processing time of each parallel job is uniformly distributed within these limits.

3.3. Simulation Results

To get a better understanding of the properties of the algorithms, different scheduling scenarios are presented. Starting with the most basic case of scheduling, (sequential jobs, offline, and in a homogeneous environment), each new scenario will become more complex and realistic or will, alternatively, focus on a different parameter. Each plot shown below displays a single objective function (either makespan, average flowtime or maximum waiting time) for several algorithms as a function of some variable while all other parameters remain fixed. We will mainly concentrate on the makespan and the maximum waiting time, because the former can be interpreted to represent the overall performance of the schedule, while the latter indicates the fairness for each individual job.

3.3.1. Offline Scheduling of Sequential Jobs

The first scenario is a simple offline scheduling problem: All jobs are submitted to the queue at the beginning of the simulation, the job mix consists of sequential jobs only, and there are 10 nodes of equal processing speed. Without any parallel jobs, FirstFit and Backfilling are identical to FIFO and have thus been excluded from the plots. For an offline problem like this one, the maximum waiting time differs from the makespan only by a constant factor, approximately a single job's runtime. Therefore, we focus on makespan and average flowtime.



$P_s = 100\%$	$P_{lp} = —$
$t_s/1000 = 1 - 100$	$t_p/1000 = —$
$T_{C,max} = 0$	nodes $= 10$
$n = 250$	speed $=$ equal

Figure 1. Offline scheduling of sequential jobs

In Figure 1, the makespan and the average flowtime are displayed as a function of the maximum processing time. Here and in the following figures, the selected simulator parameters (as listed above) are shown at the figure's bottom. In our first scenario, the minimum processing time $t_{s,min}$ has been set to 1000 seconds, while the maximum time $t_{s,max}$ ranges from 1000 to 100000 seconds. Clearly, both the makespan and the average flowtime get larger with increasing $t_{s,max}$. The relative deviation of the different algorithms' makespans is quite low, with LPT performing best, just as anticipated. FIFO, SPT and Random yield very similar results.

With respect to the flowtime, SPT is clearly superior to the other algorithms, in particular to LPT which maximizes the flow time. This result is in accordance to the discussion in Section 3.1.

3.3.2. Online Scheduling of Sequential Jobs

In the next experiment, the previous scenario is transformed into an online setting. The maximum processing time is set to 50000 seconds, while the timespan for creating jobs ($T_{c,max}$) varies from 0 (the offline case) to 10000 seconds. The larger the timespan, the less jobs are known to an algorithm at a given time, since the release times are distributed over a larger fraction of the makespan. As shown in Figure 2, the makespan first only slightly increases with growing $T_{c,max}$. Then, the release time of the last job becomes the dominating factor, resulting in a linear increase of the makespan for all algorithms.

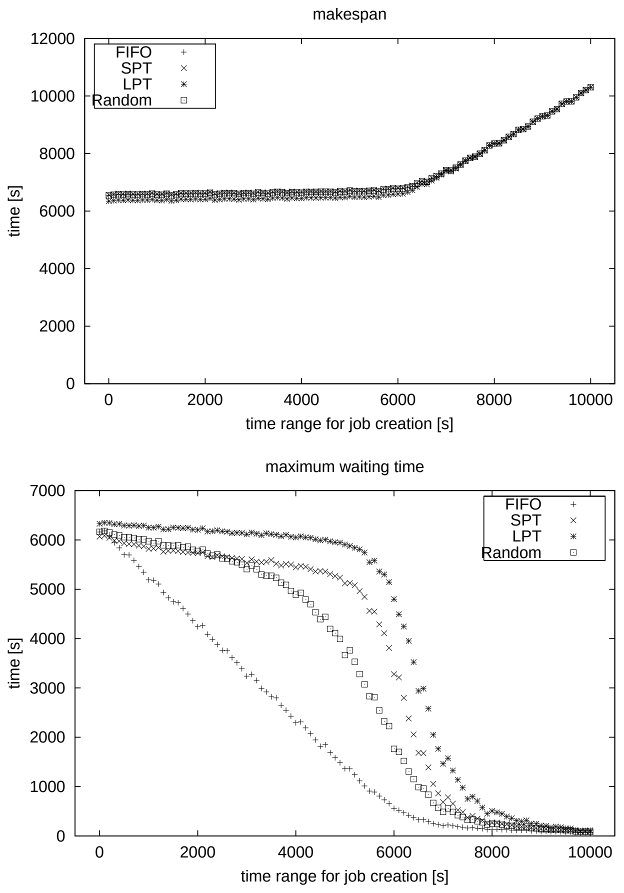
The average flowtime (not shown graphically) decreases almost linearly for increasing values of $T_{c,max}$, until it approaches an average runtime of about 255 seconds which is the average runtime of all jobs. The relative performance of the algorithms is about the same as in the previous case.

The maximum waiting time is the most interesting objective function here. It generally decreases with higher values of $T_{c,max}$, because less jobs pile up in the waiting queue. While FIFO yields an immediate and clear improvement, the other algorithms have much higher maximum waiting times for $1000 < T_{c,max} < 7000$. For very high values of $T_{c,max}$, the maximum waiting time approaches zero for all algorithms, because nearly all jobs can be started immediately without having to wait for a free node.

In this environment, FIFO is clearly the algorithm of choice. Its good results for the maximum waiting time more than compensate its slight disadvantage with respect to the makespan and flowtime functions.

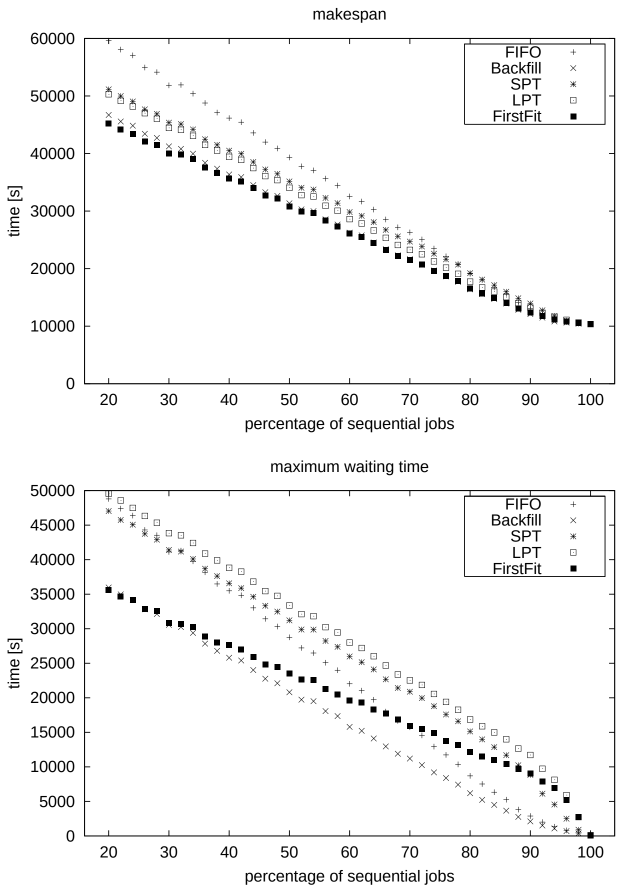
3.3.3. Online Scheduling of Mixed Sequential and Parallel Jobs

In Figure 3, parallel jobs are introduced to the simulations. $T_{c,max}$ is set to 10000 seconds. The percentage of sequential jobs P_s is varied between 20% and 100%.



$P_s = 100\%$ $t_s/1000 = 1 - 50$	$P_{lp} = —$ $t_p/1000 = —$
$T_{C,max} = 0 - 10000$s $n = 250$	nodes $= 10$ speed $=$ equal

Figure 2. online scheduling sequential jobs



$P_s = 20\% - 100\%$ $t_s/1000 = 1 - 50$	$P_{lp} = 30\%$ $t_p/1000 = 10 - 400$
$T_{C,max} = 10000$s $n = 250$	nodes $= 10$ speed $=$ equal

Figure 3. mixing sequential and parallel jobs

By introducing parallel jobs, the behaviors of FIFO, Backfilling and FirstFit become distinguishable, which is why the latter two algorithms have been added to the plots. Random, however, has been omitted, because it behaves very similar to FirstFit, with the exception of producing a higher maximum waiting time. The processing time of parallel jobs has been set to range from 10000 to 400000 seconds as opposed to 1000 - 50000 seconds for their sequential counterparts. This is motivated by the fact that parallel jobs will usually have higher needs for processing time. The actual runtime depends on the job's degree of parallelism. Since in this scenario 10 nodes of equal speed are used, the runtime of a parallel job requiring almost all of the nodes is similar to that of a sequential job.

When comparing the makespan plot to the previous ones, it becomes apparent that the relative difference between the algorithms is much higher for a sufficiently high ratio of parallel jobs. FIFO is by far the worst performer when confronted with parallel jobs. This is due to the gap problem discussed in Section 3.1. Basically, the same applies to SPT and LPT, which also block waiting jobs if the one that was selected by the scheduling algorithm cannot be started due to a shortage of free nodes. Compared to FIFO, their makespans are nevertheless significantly better. This can be explained by the fact that with SPT and LPT the selected job can vary when new jobs are submitted to the queue, while FIFO stays with its selection until enough nodes become available. FirstFit has always the lowest makespan, but Backfilling comes very close to FirstFit's performance. Both algorithms achieve this by avoiding or filling gaps that affect the other algorithms, respectively.

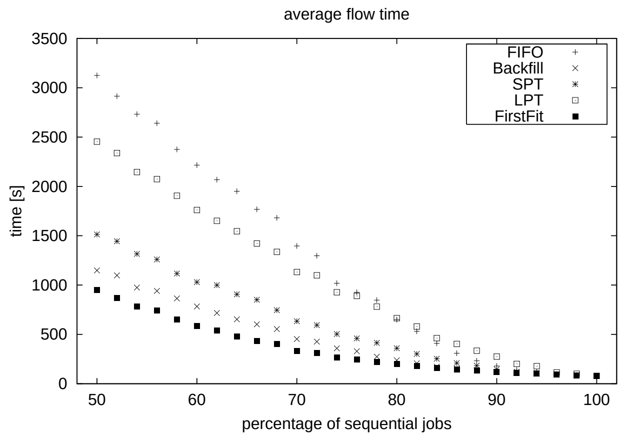
With respect to the maximum waiting time, the behavior of the algorithms changes. Backfilling is clearly the best algorithm, while FirstFit is much worse, but for high numbers of parallel jobs, both algorithms behave similarly. This is caused by the higher ratio between makespan and creation time span, such that the maximum waiting time cannot be much lower than $C_{max} - T_{C,Max}$, i.e. the time difference between the termination of the last job and the latest release time. It is also interesting that FIFO is highly competitive for high values of P_s, but performs much worse with an increasing number of parallel jobs where again the higher makespan of FIFO dominates. LPT and SPT exhibit the highest maximum waiting time, because they share the disadvantages of not starting the jobs in the order of their arrival and of a relatively large makespan.

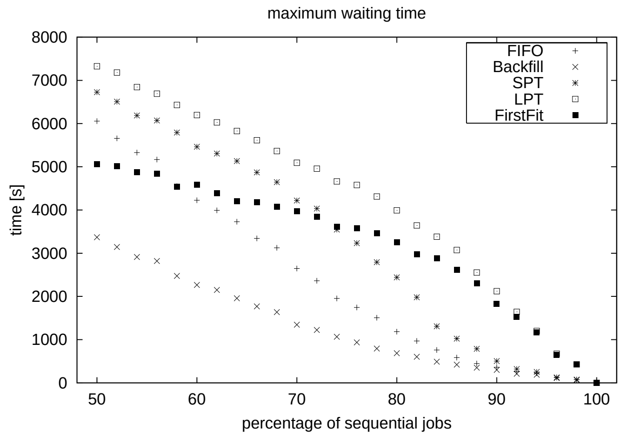
3.3.4. Online Scheduling with Different CPU Speeds

The next scenario (Figure 4) deals with nodes having different computing speeds. Their relative speeds as measured by WINNER were taken from 22 ALPHA workstations in our local network. In this simulation, the job processing times and the creation time span were adjusted to better match the higher available CPU power. Otherwise, most jobs could have been started right away without any need of queueing.

Because the graph for the makespan is very similar to the previous one, it has been omitted for this experiment and replaced by a plot of the average flowtime. Similar to the makespan, FirstFit performs best for the average flowtime, followed by Backfilling. SPT yields rather good results which is not surprising as it has been designed to minimize flowtime. LPT and FIFO perform worst; LPT because of its inherent property to minimize the makespan at the expense of a high flowtime, while FIFO suffers from its bad makespan results.

Backfilling and FIFO perform best with respect to the maximum waiting time, because the creation time range has been enlarged in relation to the makespan. This leads to a situation which exhibits stronger characteristics of an online problem, i.e. most jobs are arriving over time without too many of them being present in the waiting queue at the same time. Algorithms which tend to start jobs in the order of their release times can achieve much lower maximum waiting times than other algorithms, which is true as long as the makespan is sufficiently small. This is the reason why FirstFit outperforms FIFO for higher ratios of parallel jobs.





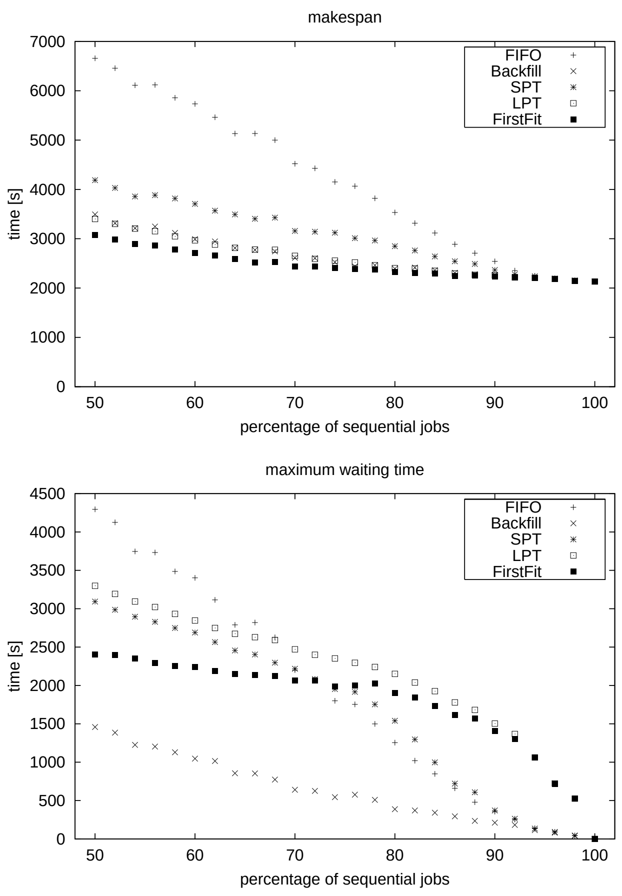
$P_s = 50\% - 100\%$ $t_s/1000 = 2 - 100$	$P_{lp} = 30\%$ $t_p/1000 = 20 - 800$
$T_{C,max} = 4000\text{s}$ $n = 250$	nodes = 22 speed = varying

Figure 4. Online Scheduling with different CPU speeds

3.3.5. Online Scheduling in a Large Network

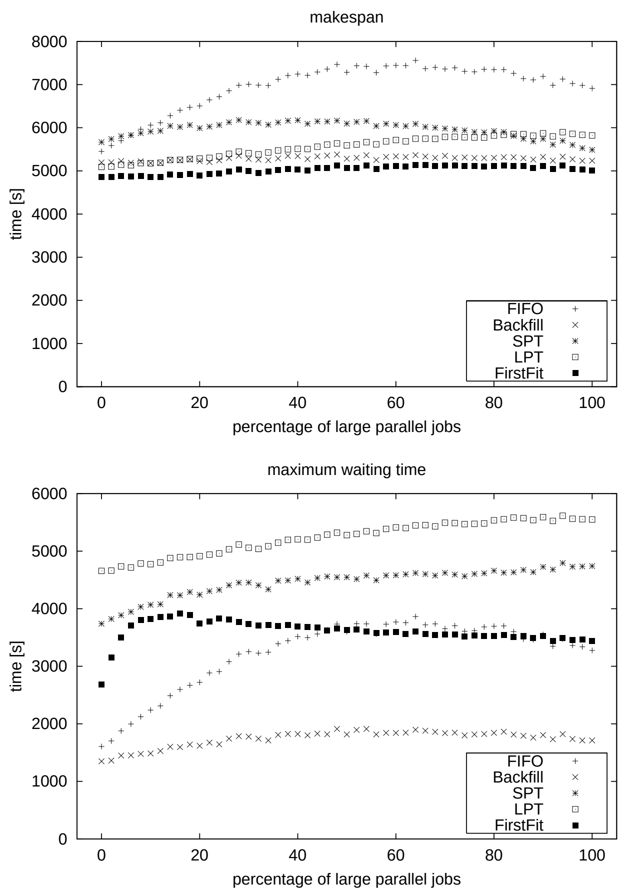
To see whether the simulation results scale to a larger network without qualitative difference, the number of nodes has been changed to 100. The distribution of the nodes' relative speeds was adopted from the previous scenario.

The only algorithm whose results have undergone a significant change is FIFO. Its problem of building large gaps of unused nodes has become worse, since there are simply more workstations that can be blocked from being used at the same time. This manifests itself in both a higher makespan and maximum waiting time when a sufficiently high number of parallel jobs is present. Apart from this effect, increasing the number of nodes has not affected the simulation very much.



$P_s = 50\% - 100\%$	$P_{lp} = 30\%$
$t_s/1000 = 2 - 100$	$t_p/1000 = 20 - 800$
$T_{C,max} = 2000\text{s}$	nodes = 100
$n = 500$	speed = varying

Figure 5. Online scheduling in a large network



$P_s = 70\%$	$P_{lp} = 0 - 100\%$
$t_s/1000 = 2 - 100$	$t_p/1000 = 20 - 800$
$T_{C,max} = 3500\text{s}$	nodes = 22
$n = 250$	speed = varying

Figure 6. Online scheduling with a varying degree of parallelism

3.3.6. Online Scheduling with a Varying Degree of Parallelism

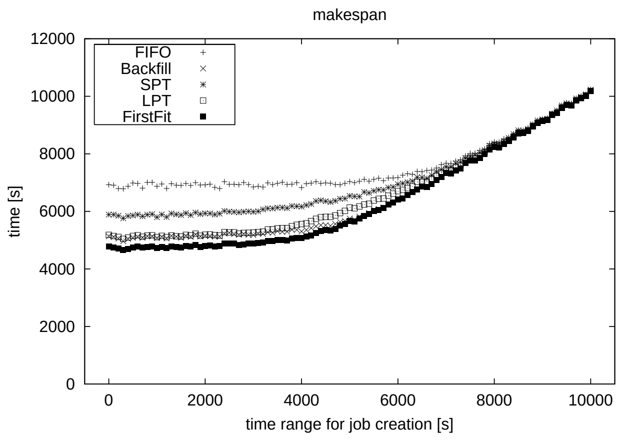
Next, we focus on the importance of the degree of parallelism. We do this by altering the percentage of large parallel jobs P_{lp}. These are defined as jobs which require at least 50% of all workstations, so no two of them can be running in parallel.

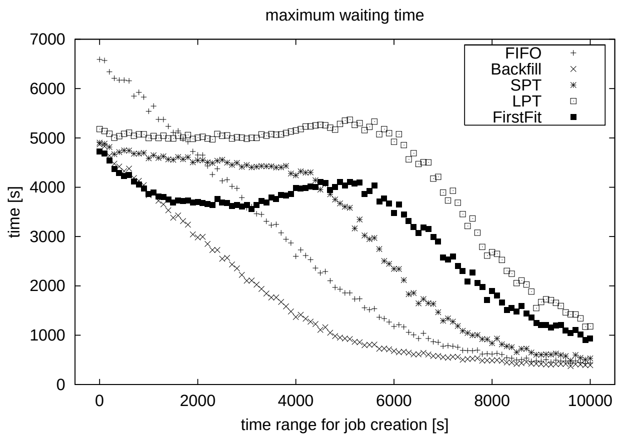
Whereas the makespan of the schedules generated by FirstFit and Backfilling is nearly independent of P_{lp}, FIFO gets steadily worse for $0\% < P_{lp} < 50\%$ as can be seen in Figure 6. For small values of P_{lp}, LPT can compete with FirstFit and Backfilling (with respect to the makespan) but falls back when more and more jobs require more than half of the nodes. SPT actually performs best with very low or very high percentages of large jobs while behaving worse when confronted with an even distribution.

For the maximum waiting time, the situation changes slightly: While the effect on FIFO is greatly amplified, FirstFit's maximum waiting time curve rises steeply until P_{lp} reaches about 20% before it gradually declines. Again, LPT and SPT yield the worst results with a higher percentage of large jobs causing higher waiting times. Backfilling is by far the best performer for all values of P_{lp}.

3.3.7. Online Scheduling with Varying Creation Range

As already mentioned for some of the previous scenarios, the ratio between the makespan and the time range for job creation has a strong effect on the relative performance of the different algorithms with respect to the maximum waiting time. If the makespan is much bigger than the job creation time range, a high percentage of all jobs are present in the waiting queue concurrently. Thus, after an initial phase with jobs arriving at the queue over time, the problem is effectively converted into an offline setting. The longer the phase of job creation is in relation to the makespan, the more the characteristics of an online problem become effective.





$P_s = 70\%$ $t_s/1000 = 2 - 100$	$P_{lp} = 30\%$ $t_p/1000 = 20 - 800$
$T_{C,max} = 0 - 10000$s $n = 250$	nodes $= 22$ speed $=$ varying

Figure 7. Online scheduling with varying creation range

To explore the consequences of the $C_{max}/T_{C,max}$ ratio more deeply we created a scenario where $T_{C,max}$ is varied (Figure 7). For the offline problem ($T_{C,max} = 0$) the different algorithms' makespans are in the range of about 4500s - 7000s; their relative performance is similar to the previous simulations. When $T_{C,max}$ approaches the offline C_{max}, the makespan begins to rise concurrently with $T_{C,max}$, since the latest job release date determines the minimum makespan that can be achieved. Thus, the algorithms yield almost identical makespans for high values of $T_{C,max}$.

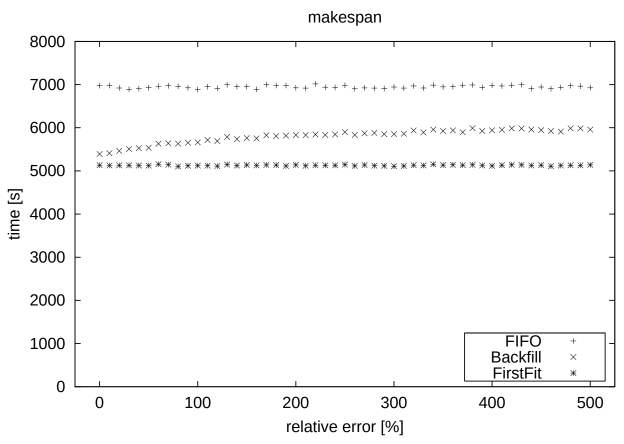
The maximum waiting time objective function is much more interesting. For the offline case, it is close to the makespan C_{max} for all algorithms. When $T_{C,max}$ is raised, the maximum waiting time instantly decreases for FIFO and Backfilling. SPT and LPT remain on their offline levels until the creation time span is close to the makespan. After that, the maximum waiting time decreases rapidly. The turning point for LPT is located at higher values of $T_{C,max}$ than that of SPT and, interestingly, after an initial decline, the maximum waiting time actually rises before the turning point. The same is observable with FirstFit, but to an even greater extent: At first the waiting time behaves similarly to that of the Backfilling algorithm. After slowly leveling out, it rises significantly, with its peak located near $T_{C,max} = 5000s$ before finally sloping down.

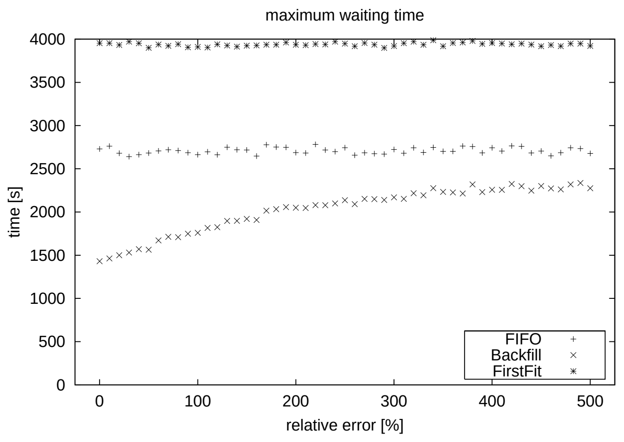
The Backfilling algorithm is clearly superior to all other algorithms for the middle range of $T_{C,max}$. SPT and FirstFit are competitive if all release dates are close to the beginning of the simulation. FirstFit outperforms Backfilling with respect to the makespan. It needs no processing time information and is easy to implement. Hence, it is the algorithm of choice when parallel jobs have to be scheduled is a setting which is mostly offline.

3.3.8. Online Scheduling with Fuzzy Knowledge of Processing Times

So far, perfect a-priori knowledge of the jobs' processing times was assumed in all simulations. Clearly, this assumption is not realistic. To estimate the impact of erroneous runtime predictions, the simulator was modified to deliver deliberately wrong processing time information to the algorithms. The erroneous processing time t_{err} is derived from the real one t_{proc} by the simple scheme $t_{err} = t_{proc} \cdot f$ respectively $t_{err} = \frac{t_{proc}}{f}$ (one of these two equations is chosen randomly for each job), where $f = 1 + (rand() \cdot p_{err})$. $rand()$ denotes a random number generator which generates evenly distributed values in the interval $[0, 1)$ and p_{err} denotes the relative error, i.e. $p_{err} = 1$ allows $0.5\, t_{proc} < t_{err} < 2\, t_{proc}$.

In Figure 8 the results of erroneous processing time information are shown for relative errors of $0\% - 500\%$ ($p_{err} = 0 \ldots 5$). Above, we identified FirstFit and Backfilling as being superior, so SPT and LPT have been dropped. FIFO has been included as a reference. Because both FIFO and FirstFit do not use processing time





$P_s = 70\%$	$P_{lp} = 30\%$
$t_s/1000 = 2 - 100$	$t_p/1000 = 20 - 800$
$T_{C,max} = 4000\text{s}$	nodes $= 22$
$n = 250$	speed $=$ varying

Figure 8. Online Scheduling with fuzzy knowledge of processing times

information, the results for these algorithms should be independent of p_{err}. However, minor deviations in the plots can be observed, because for each value of p_{err} different job mixes were randomly created.

Backfilling, however, can be expected to suffer from fuzzy processing time data, since it uses this information to decide whether or not to place a specific job into a gap of unused nodes. As shown in Figure 8, increasing p_{err} does actually enlarge the makespan of the Backfilling schedules, but only by about 15% for $p_{err} = 500\%$. The effect on the maximum waiting time is much bigger (about 50% for $p_{err} = 500\%$), but Backfilling can still perform better than FIFO and much better than FirstFit with respect to maximum waiting time.

The impact of fuzzy processing time information on Backfilling is clearly noticeable but less than one might have expected. Obviously, with a relative error of 500%, there will be a significant number of false placement decisions, i.e. Backfilling will try to start jobs within a gap which, in fact, is too small for the job's runtime. However, these faults do not affect the objective functions as strongly as expected. If the simulation results carry over to scheduling in a real workstation cluster where knowledge of a job's processing time is often very limited, using a Backfilling algorithm in these environments seems to be reasonable. From the simulations it is evident that Backfilling is especially strong when the jobs do not arrive too rapidly, i.e. jobs are created over a significant fraction of time with respect to the corresponding makespan. Otherwise, the FirstFit algorithm may be preferred.

3.4. Summary

In this section, we studied the behavior of six algorithms for online scheduling of batch jobs. The performance of these algorithms was compared with respect to three different objective functions. The *makespan* indicates the overall system utilization whereas *average flowtime* and *maximum waiting time* represent how well individual jobs are treated by the scheduler.

If only sequential jobs have to be scheduled, the simple FIFO algorithm performs best by minimizing the maximum waiting time while having good to average results with respect to makespan and flow time. LPT and SPT suffer from starvation problems (high maximum waiting time) for short or long running jobs, respectively. The random algorithm performs surprisingly well for minimizing makespan and flowtime, but also suffers from high maximum waiting times.

When parallel jobs have to be scheduled too, the FIFO algorithm looses its efficiency because its strict execution order may cause workstations to be idle until enough machines become available for the next job to be scheduled. The backfilling version of FIFO and the FirstFit algorithm perform better, because they try to fill the gaps caused by standard FIFO. In offline settings, FirstFit performs best, because in this case the maximum waiting time directly depends on the makespan which is slightly better than the makespan of backfilling FIFO. In online scheduling situations, the latter is superior, because FirstFit tends to delay parallel jobs that require many workstations until the final execution phase in which the machine utilization then drops significantly. In this case, backfilling FIFO performs better because it fills gaps based on estimated runtimes.

To verify our findings, we varied the basic parameters used in our simulations. We scaled up our simulation to a large number of machines, we varied the percentage of parallel jobs in the job mix, and we varied the time range in which jobs are created. All three kinds of variations had only minor influence on our simulation results, confirming their validity. Finally, we investigated the influence of erroneous runtime estimation on the efficiency of the backfilling FIFO. Surprisingly, even extremely wrong estimates had only moderate impact on the computed schedules, as observed by the objective functions. This result provides a strong argument for the efficacy of backfilling FIFO.

4. Online Scheduling in WINNER

The WINNER resource management system supports the execution of interactive and batch jobs in networks of Unix workstations [1,2]. Interactive jobs may either use text terminals or graphical user interfaces based on the X-window system for input/output. Batch jobs may either be sequential or parallel; as parallel jobs, the GNU make tool and custom PVM applications are supported. WINNER supports uniprocessor and SMP machines, and runs on Digital Unix, FreeBSD, HP-UX, Linux, and Solaris.

In this section, we present WINNER's system design along with its support for online scheduling, consisting of data acquisition, runtime normalization, and runtime prediction.

4.1. WINNER *system structure*

WINNER has been designed for typical Unix networks of workstations, consisting of a central server and several workstations. The various tasks of the WINNER system are performed by three kinds of *manager* processes as shown in Figure 9: system managers, node managers, and job managers. Additionally, there are several user interface tools e.g. for status reports, for influencing the availability of individual workstations, and for managing batch queues.

The system manager is the central server process of a WINNER network. Its duties include (a) collecting the load information of all respective workstations, (b) controlling the currently active jobs, (c) assigning hosts to

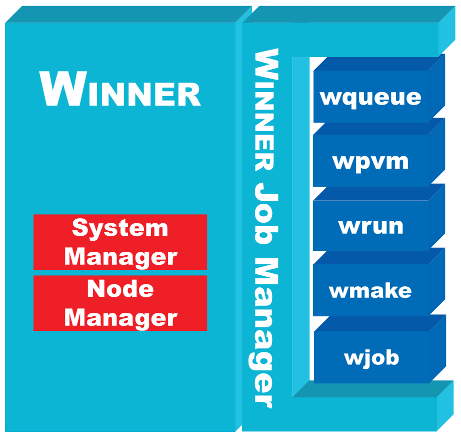


Figure 9. Manager processes in a WINNER cluster.

interactive job requests, and (d) managing batch queues and executing scheduling algorithms.

On every host in a WINNER network, a node manager performs the tasks related to the machine it runs on. First of all, it periodically measures the host's utilization. It reports to the system manager the host's architecture, memory size, as well as the machine's base speed S_b (measured in the absence of workload at boot time with a standardized benchmark), and the currently available speed S_c, taking the current workload into account. Furthermore, node managers are responsible for starting and controlling WINNER processes on their node. Whenever a job terminates, the node manager reports the consumed CPU time to the system manager.

System and node managers run as daemon processes. In contrast, job managers are invoked by a WINNER user in order to execute interactive jobs, or to enqueue a batch job. Thus, job managers are part of WINNER's user interface. For interactive jobs, their duties are (a) acquiring resources from the system manager, (b) starting processes on the acquired nodes via the respective node managers, and (c) controlling and possibly redirecting input and output of the started processes. For batch jobs, WINNER provides a special "meta" job manager wqueue that enqueues a job instead of executing it immediately. The respective job manager is then activated under the control of the system manager when the job is actually scheduled. A detailed description of WINNER's batch queueing facility can be found in [2].

To implement the backfilling FIFO algorithm, the system manager needs proper runtime information about each job it has to schedule. In the following, we describe how runtime information is collected and how future runtimes are predicted using this information.

4.2. Collecting Information

There are two possibilities to obtain the runtime of a job. One is to urge the user to specify the estimated runtime of each submitted job. The second possibility is to measure the processing time a job has consumed during execution and to use this information for predicting the runtimes of similar jobs in the future.

Forcing a user to specify the runtime of a job is problematic. Often users cannot properly estimate the runtime, and even if they could, wrong timings may be specified on purpose to gain scheduling advantages over other users. Such a behavior could only be defeated by aborting jobs which exceed their estimated runtimes, but this is certainly unsatisfactory for users trying to estimate their runtimes properly.

Automatic runtime collection and prediction by the system is not trivial, either. The runtime of each job or process depends on a variety of factors. Even if a program consumes a constant amount of processing units, the speed of the workstation in charge determines the actual execution time. The speed of the workstation, however, depends on both its maximum performance and on the actual concurrent work load. Furthermore, hardly any program consumes a constant amount of processing units, due to possibly different command line arguments or input files being processed.

Nevertheless, in WINNER we decided to gather runtime information automatically. Each job that is submitted to the system is specified by an identifier. The identifier consists of the name of the program without a leading path or command line arguments. Obviously, the major drawback of this approach is that different programs may have the same identifier and thus their runtimes may differ significantly. On the other hand, the advantage is that the runtime associated with each identifier consists of many values. Thus, if most identifiers correctly indicate the same program, the average may be a good estimation.

To take advantage of the runtimes of previous runs of a program, WINNER has to calculate a time which neither depends on a particular workstation, nor on any other simultaneously executed task. Thus, it is essential to convert the measured times to a value which is independent of a workstation. In the following, we call this value *normtime* (T_{norm}).

WINNER computes the average T_{new} of several normtimes similar to the load average value in Unix kernels as a weighted moving average using the former average T_{old} and the recently calculated normtime T_{norm}, where $0 < \alpha < 1$:

$$T_{new} = \alpha \cdot T_{old} + (1 - \alpha) \cdot T_{norm} \qquad (1)$$

4.3. Calculating Normtime

Whereas the runtime of each process of a job is measured by the corresponding node manager, the system manager computes the normtime based on the runtimes of all processes belonging to a job. To do so, the system manager has to calculate the percentage $P_{CPU,i}$ of the CPU obtained by each process i. This calculation is based on the times $t_{user,i}$ and $t_{system,i}$ the process executed user and system routines, respectively, and $t_{real,i}$, the total execution time of the process. With these times, we get the percentage of CPU utilization for a single process i:

$$P_{CPU,i} = 100 \cdot \frac{t_{user,i} + t_{system,i}}{t_{real,i}} \qquad (2)$$

Using (2) and the (known) maximum speed $S_{b,j}$ of workstation j where process i was executed, the system manager can determine the normtime T_{norm} as follows:

$$T_{norm} = \sum_{processes} t_{real} \cdot \frac{S_{b,j} \cdot P_{CPU,i}}{100} \qquad (3)$$

Combining (2) and (3), we get the normtime as:

$$T_{norm} = \sum_{processes} S_{b,j} \cdot (t_{user,i} + t_{system,i}) \qquad (4)$$

To verify whether the normtime is independent of the speed and current load of a workstation, several measurements were performed. A program for computing the value of π was used to measure the normtime on four workstations of different speeds. Additionally, disturbing load was produced by executing several computation-intensive programs. The result is shown in Figure 10, where the x-axis shows the percentage of CPU time obtained by the program computing π, and the y-axis shows the calculated normtime.

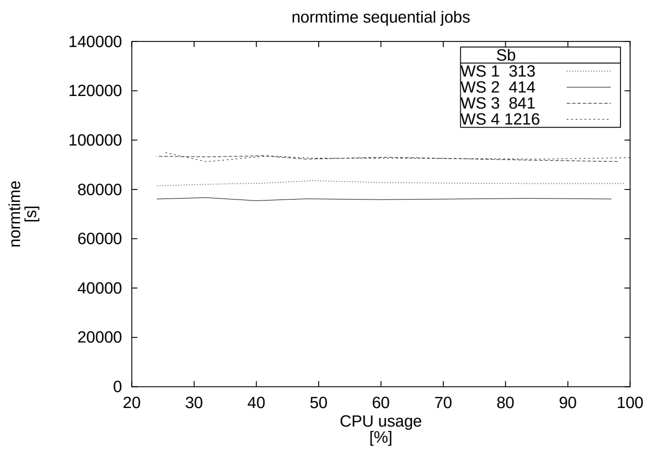


Figure 10. normtime of sequential pi-calculation

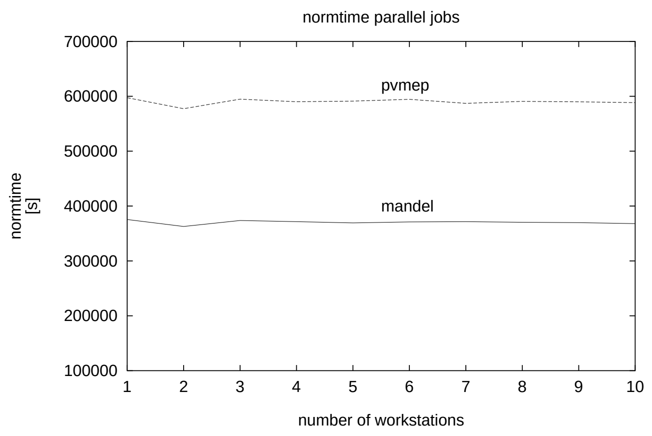


Figure 11. normtime of parallel mandel and pvmep calculation

Ideally, all curves should form a single horizontal line if the normtime is completely independent of the speed and current load of a workstation. Since each single curve is very close to a horizontal line, the normtime actually is nearly independent of the background load. With respect to the speed of the workstations there is a deviation of about 20% between the highest and lowest normtime calculated. This deviation is an inherent property of the way of representing the performance of a workstation by a single number. To overcome this problem, a suitable benchmark would have to take several parameters into account and deliver a full spectrum of performance results [19,37]. Since the kind of performance requirements of a specific application is not known to Winner, there is no way of deciding which of these results would best correlate with the behavior of the application. Thus, simply using a scalar value is the best we can do. Despite of the deviations observed, the normtime is independent of both the speed and background load of a workstation to a large extent.

Another important property of the normtime should be its independence of the number of workstations used by parallel jobs. To verify this, we used two different parallel applications based on PVM [14]. The results are shown in Figure 11, where the x-axis contains the number of nodes that participated in the calculation, and the y-axis shows the calculated normtime. Obviously, the normtime is almost completely independent of the number of workstations.

4.4. Predicting Runtimes

To obtain proper runtime estimations for future runs of the same job, we have to distinguish between two types of jobs. The first type contains sequential and load-balanced parallel jobs, where the work performed by each workstation is proportional to its actual speed value. In this case, the formula for the predicted runtime $T_{predict}$ is obtained by dividing the normtime by the sum of the current speed $S_{C,i}$ of each workstation i participating in the computation:

$$T_{predict} = \frac{T_{norm}}{\sum_i S_{C,i}} \tag{5}$$

The second type of job contains parallel jobs where the workstation running at minimum current speed limits the execution time of the whole job. These are parallel programs where a given problem is divided by the number of available nodes, and each node performs the same amount of computation. In this case, $T_{predict}$ is given by

$$T_{predict} = \frac{T_{norm}}{n \times S_{min}} \tag{6}$$

where n is the number of available nodes.

To decide which formula has to be used, we have to know the type of job. Unfortunately, a decision between both types of job is possible only if the current speeds of at least two workstations of the parallel job are significantly different. This is essential because on workstations having nearly the same current speed both types of jobs perform equal work. Fortunately, experimental tests indicated that a proper identification of the job type only requires a current speed difference of at least 10% between the slowest and fastest machine.

Once this condition is satisfied, Winner detects the type of job by computing the runtime difference between the two processes of the job in discussion running on the slowest and fastest machine, respectively. If this difference is smaller than 10%, then the job is classified as belonging to type 1, otherwise to type 2.

5. Experimental Results in a Workstation Network

In order to investigate whether the results of the simulations described in Section 3 carry over to a real job mix on a network of workstations, we performed a series of measurements. For these experiments, several different versions of the WINNER system manager were implemented, each providing a different scheduling algorithm. The measurements were restricted to FirstFit, Backfilling, and FIFO, because the first two are those we are primarily interested in as a result of the simulations. FIFO was taken as a reference algorithm.

The job mix was randomly generated out of five different types of jobs: three sequential applications and two parallel PVM applications. The total number of jobs was set to 100. The jobs were executed on a network of 22 Digital ALPHA workstations, mostly the same which acted as a template for the distribution of processing performance used in Section 3.

The job types we used are listed in Table 1. pi is a simple program which calculates the first 20000 decimal digits of π. It mainly uses lots of simple integer operations. The well-known raytracer POV–Ray [10] has been used for rendering a simple scene as a representative for applications with a high portion of floating point operations. Its normalized runtime is about the double than that of pi. The third sequential job, sim, is the scheduling simulator used in Section 3. It uses a mix of integer and floating point operations and has a medium sized memory footprint. Its runtime is about twice that of povray.

As parallel jobs two simple PVM applications have been selected: mandel is a manager/worker style application computing images of the Mandelbrot set. Its way of performing the parallel computation is inherently load balancing, so WINNER will assign job type 1 to it as explained in Section 4.4. Its normalized runtime is between those of pi and povray, resulting in quite short run times when lots of workstations are used in parallel. The second PVM application was taken from the PVM implementation of the NAS Parallel Benchmarks [41]. pvmep creates pairs of random numbers and applies some stochastic tests to them. It uses a static partitioning of the problem which can cause a load imbalance when workstations with different processing speeds are used, thus it is classified as job type 2.

Table 1
Job types.

Name	normalized runtime	type
pi	190s	sequential
povray	420s	sequential
sim	850s	sequential
mandel	300s	parallel
pvmep	4800s	parallel

The job mix was randomly assembled from these jobs. The ratio of sequential jobs has been selected to be 70%. The parallel jobs are subdivided into 70% of small and 30% of large parallel jobs requiring more than 11 CPUs. The time span for job creation was 90 minutes. It has been chosen according to the makespan that was estimated to be somewhat less than two hours.

Our measurements have been performed on weekends when the workstations were almost idle. The job mix was scheduled once by each algorithm, except for Backfilling where two runs were performed: Prior to the first run, WINNER had no knowledge at all about the jobs' runtimes. Due to this fact, no backfilling could be used during the first minutes of the schedule. After each type of job has been successfully run at least once, there is sufficient data available for WINNER to predict the runtime of further runs of the same job, thus enabling backfilling. Thus, for the second Backfilling run, WINNER was allowed to use the information it gathered during the execution of the first run, and backfilling could be used for all jobs right from the beginning.

Figure 12 presents the results of our measurements. First of all, FIFO generally performs worst with the exception of the maximum waiting time objective where it is at least able to compete with FirstFit. All three algorithms yield similar makespans. Backfilling, on its second run, actually achieves a lower makespan than FirstFit. Still, FirstFit performs better than FIFO.

With respect to the average flowtime, FirstFit outperforms the other algorithms, especially FIFO. When looking at the maximum waiting time results, both FIFO and Backfilling yield a much lower maximum waiting time than FirstFit. This was predicted by the simulation results for an online scenario.

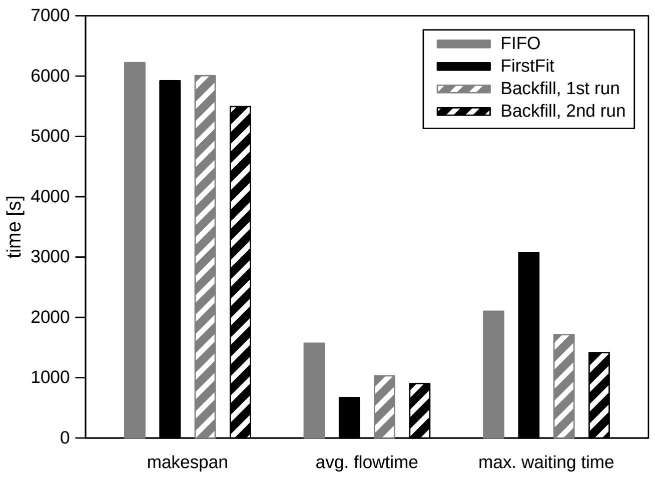


Figure 12. Results of scheduling a random job mix on the local network of ALPHA workstations

Finally, the second run of the Backfilling algorithm improves the results for all three objective functions from its corresponding first run. This difference is obviously caused by the initial phase in the first run where Backfilling behaves like FIFO due to lack of information about the jobs' runtimes.

To summarize, the results match those of the scheduling simulator quite well. Except for the average flowtime, Backfilling performs better than or equal to FirstFit. When compared to FIFO, it yields better values for all objective functions. These results confirm our simulation results in which Backfilling was the best of the algorithms we considered for an online scheduling problem.

6. Related Work

Schedulers are, in a very general sense, concerned with selecting resources on which to run application processes in a coordinated way such that certain objectives (such as high resource utilization or fast application execution) are met. For sequential applications, scheduling is only concerned with selecting computing resources (CPUs) for application execution. The most prominent system of this kind is the Condor resource manager [28]. Condor is oriented towards *high throughput computing* [29]. In this sense, it assigns applications to suitable, idle CPUs to run on. As with WINNER, Condor's scheduling is non-preemptive in the sense that process migration is considered to be very time consuming and hence not used for improving schedules but only as a "last resort action" whenever a CPU currently used by Condor becomes unavailable. Because Condor is focused on sequential applications, it schedules processes in a FIFO order, augmented by a user priority scheme. Condor has limited support for parallel applications based on PVM [14] for which it bundles several application processes to a PVM virtual machine while burdening the administration of added or removed processes to the application code itself. The CARMI library [33] provides such application code for running manager/worker style parallel applications on top of PVM and Condor.

For parallel applications, the processes not only have to be assigned to available CPUs, they also have to cooperate as efficiently as possible. This can be achieved in multiple software layers. So-called "application-level schedulers" [5] come into play once a parallel application has been started. They use application-specific performance models (e.g. for completion times of given subtasks), parameterized by dynamic resource performance capacities. The objective of application-level schedulers is to minimize the completion time of an application based on a given (possibly dynamically changing) set of resources. Techniques like *guided self-scheduling* [31] and *trapezoid self-scheduling* [39] are simple examples of application-level schedulers for loop-based, data-parallel applications.

In *space-sharing systems*, application processes are granted exclusive access to their resources. In this case, application-level scheduling often is sufficient to efficiently use the given processors. Alternatively, in *time-sharing systems* application processes have to compete for their CPUs with other, unrelated processes. In this case, parallel execution of communicating processes can be very inefficient when the time slices assigned to the processes belonging to the same application are not synchronized. This problem may cause additional delays when sending or receiving messages to unsynchronized peer processes. A technique called *gang scheduling* or *coscheduling* [13,15] aims to overcome this problem, either by changes to the operating system-level scheduler or by a user-level approximation technique that simultaneously sends signals to process groups. *Proportional-share scheduling* [3] extends this idea by additionally assigning credits to applications such that all processes (sequential, interactive, and parallel jobs) get fair shares of CPU time.

The study presented in this paper discusses online *batch scheduling* algorithms, aiming to optimize system utilization and application turnaround time by elaborate placement of processes to CPUs. The scheduling techniques discussed so far in this section hence rather complement our work. Those approaches should preferably be used in addition to WINNER's batch scheduler. We will now discuss other resource management systems providing batch schedulers.

The BATCH system [32] was designed to manage multiple batch job queues. It provides remote queues, which may be assigned to a specific machine, and also load balanced queues. Load balanced queues are specified by a set of hosts which decide among themselves on which host the current job will be executed. Such a scheduling decision is mainly based on the current load averages of the hosts. Users can start and suspend jobs at specific times and/or days. BATCH does not support parallel jobs.

The Generic Network Queueing System (GNQS) [21] was developed by the NASA as a successor of the UNIX based Network Queueing System (NQS). The goals of its design were to provide full support of both batch and device requests (e.g. line printer requests) and the remote queueing and routing of such requests through the network of machines. The standard algorithm is FIFO, but the scheduler is supposed to be easily modifiable on an installation by installation basis. Parallel jobs are not supported.

QBATCH [4] is a batch processing system which handles multiple queues. Each queue is totally independent of the others, and the administrator can reorder the jobs in a queue. The scheduling algorithm is FIFO. Parallel jobs are not supported.

The experimental UNIX based Distributed Queueing System DQS [17] was developed at the Supercomputer Computations Research Institute (SCRI) at Florida State University. DQS (formerly DNQS) was in its first version conceptually similar to NQS. In the current version, the scheduler is based on queue complexes which can be defined by the administrator. There is no information about the scheduling algorithms in use. DQS supports parallel jobs using the PVM, P4, and P5 communication libraries.

EASY, the Extensible Argonne Scheduler System [27] designed by Argonne National Laboratory was developed to provide fair and simple job scheduling while efficiently using the available resources. The scheduler uses a backfilling FIFO algorithm which depends on user-specified runtimes. To enable the users to predict runtimes and to have optimum performance, each job has exclusive access to the requested nodes, i.e. background load on a workstation is not permitted. EASY supports parallel jobs using PVM, MPI, P4, and MPL.

CODINE (Computing in Distributed Networked Environments) [8] was developed by GENIAS Software in order to provide satisfactory resource utilization in heterogeneous workstation networks. It provides several queues which are assigned to workstations. Additionally, there exists a central pending queue, where incoming jobs are stored until they can be scheduled to a machine. The codine scheduler uses a FIFO algorithm, but additional attributes can be specified by the administrator. CODINE supports parallel jobs using the PVM and EXPRESS communication libraries.

CONNECT:QUEUE is a commercial version of NQS [4] that is very similar to GNQS. It provides three types of queues, namely batch queues which support a percentage mechanism for scheduling and running jobs, device queues and pipe queues. The batch queue scheduling algorithm is based on the percentage of physical memory used, CPU utilization and a queue job limit. Parallel jobs using PVM and Linda are supported.

The Load Sharing Facility (LSF) [4,42] is a load sharing and batch queueing system for heterogeneous UNIX environments. It allows several parameters to be specified by the administrator. Such parameters are resource units for particular users, preemptive scheduling (i.e. jobs of lower priority can be preempted by jobs of higher priority) or run- and dispatch-windows, which can be used to limit queues to certain time spans. Users can customize their own scheduling algorithms. LSF supports parallel jobs based on PVM, Linda, DSM and TCGMSG.

In comparison, WINNER's batch scheduling system supports sequential as well as parallel PVM and *make* jobs. Its modular structure allows to easily add further parallel job types like e.g. MPI [18] applications. As a result of this study, WINNER uses a backfilling FIFO algorithm while the scheduler learns application runtimes from previous runs, removing the burden of runtime estimation from the user. The feasibility of runtime prediction from historical data has also been shown by others [22,36]. WINNER's batch scheduler assigns at most one process at a time to a workstation in order to avoid performance degradation by memory contention. This is a reasonable design decision for networked workstations which are usually rather limited in main memory. Even in dedicated parallel machines like the ASCI Blue-Pacific, at most 2 or 3 processes are scheduled to each CPU to avoid memory congestion [13].

WINNER uses a conservative backfilling FIFO algorithm that tries not to delay any waiting job by filling gaps in the schedule. The work in [11] backs our decision in showing that conservative backfilling is equivalent to aggressive backfilling which only tries not to delay the first waiting job in the queue. A further improvement could be obtained by slack-based backfilling [38] which uses so-called slack values reflecting the relative importance of the jobs. This techniques needs both user-selected and administrative priorities which are currently not available to WINNER's scheduler. Another possible improvement could so far not be implemented in WINNER's batch scheduler due to missing information. So-called *malleable jobs* [9] are parallel jobs that can run with different numbers of processors from a given interval, and possibly use additional CPUs in the middle of the run when they become available. The presence of such jobs would give a large degree of freedom to WINNER's backfilling scheduler but would also complicate scheduling strategies.

7. Conclusions

In this paper, we studied the behavior of six algorithms for online scheduling of batch jobs, namely first in first out (FIFO), FIFO with backfilling, FirstFit, random, largest processing time first (LPT), and shortest processing time first (LPT), by simulation. We compared the performance of these algorithms with respect to three different objective functions: the *makespan, average flowtime*, and *maximum wait time*. The *makespan* indicates the overall system utilization whereas *average flowtime* and *maximum wait time* represent how well individual jobs are treated by the scheduler.

If only sequential jobs have to be scheduled, the simple FIFO algorithm performs best by minimizing the maximum waiting time while having good to average results with respect to makespan and flow time. LPT and SPT suffer from starvation problems (high maximum waiting time) for short or long running jobs, respectively. The random algorithm performs surprisingly well for minimizing makespan and flowtime, but also suffers from high maximum waiting times.

When parallel jobs have to be scheduled too, the FIFO algorithm looses its efficacy because its strict execution order may cause workstations to be idle until sufficiently many machines become available for the next job to be scheduled. The backfilling version of FIFO and the FirstFit algorithm perform better, because they try to fill the gaps caused by standard FIFO. In offline settings, FirstFit performs best, because in this case the maximum waiting time directly depends on the makespan which is slightly better than the makespan of backfilling FIFO. In online scheduling situations, the latter is superior, because FirstFit tends to delay parallel jobs that require many workstations until the final execution phase in which the machine utilization then drops significantly. In this case, backfilling FIFO performs better because it fills gaps based on estimated runtimes.

Based on our simulation results, we then implemented the algorithms FIFO, FIFO with backfilling, and FirstFit in the framework of the WINNER resource management system, and performed experiments with the three algorithms in a real workstation network. The results of the experiments showed that both FirstFit and the backfilling FIFO perform significantly better than a strict FIFO. Both algorithms produce similar values for the makespan, while backfilling FIFO yields smaller maximum waiting times. Whenever our backfilling FIFO lacks runtime information for a given job, it acts like a strict FIFO. This drawback only affects the initial "learning phase" of the algorithm. As soon as the algorithm has information about (almost) all job identifications, its benefits become apparent.

The behavior of WINNER's scheduler may further be improved by adaptively switching between FirstFit and the backfilling FIFO, depending on the length of the job queue which indirectly indicates whether a given situation is "more online" or "more offline". Furthermore, alternative job placement strategies may be considered, e.g. by trying to assign presumably long-running jobs to faster workstations. Finally, the effects of preemptive scheduling (trading the overheads of checkpointing and migration against a higher flexibility for applying backfilling) seems to be worth investigating.

References

[1] O. Arndt, B. Freisleben, T. Kielmann and F. Thilo. Scheduling Parallel Applications in Networks of Mixed Uniprocessor/Multiprocessor Workstations. *Proc. ISCA 11th International Conference on Parallel and Distributed Computing Systems (PDCS-98)*, Chicago, IL, Sep. 1998, pp. 190-197.

[2] O. Arndt, B. Freisleben, T. Kielmann and F. Thilo. Batch Queueing in the Winner Resource Management System. *Proc. 1999 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'99)*, Vol. V, pp. 2523–2529, Las Vegas, NV, Jun./Jul. 1999.

[3] A.C. Arpaci-Dusseau and D.E. Culler. Extending Proportional-Share Scheduling to a Network of Workstations. *Proc. 1997 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'97)*, Las Vegas, NV, Jul. 1997.

[4] M. A. Baker, G. C. Fox and H. W. Yau. A Review of Commercial and Research Cluster Management Software. Technical Report NPAC TR SCCS-748, NPAC, Syracuse University, Now York, Nov. 1995.

[5] F. Berman, R. Wolski, S. Figueira, J. Schopf, and G. Shao. Application-level Scheduling on Distributed Heterogeneous Networks. In *Proc. Supercomputing 96*, Nov. 1996.

[6] P. Brucker. *Scheduling Algorithms*, Springer-Verlag, 1998.

[7] B. Chen and A. P. A. Vestjens Scheduling on Identical Parallel Machines: How Good is LPT in an On-line Setting?. Technical Report Memorandum COSOR 96-11, Eindhoven University of Technology, 1996.

[8] CODINE 4.1.1, Computing in Networked Environments. Technical Overview, GENIAS Software GmbH, 1996.

[9] A.B. Downey. Using Queue Time Predictions for Processor Allocation. In *Proc. International Parallel Processing Symposium (IPPS'97) Workshop on Job Scheduling Strategies for Parallel Processing*, 1997.

[10] A. Enzmann, L. Kretzschmar, and C. Young. *Ray Tracing Worlds with POV–Ray.* The Waite Group, 1994.

[11] D.G. Feitelson and A. Mu'alem Weil. Utilization and Predictability in Scheduling the IBM SP2 with Backfilling. In *Proc. 12th International Parallel Processing Symposium (IPPS'98)*, pp. 542–546, Apr. 1998.

[12] D. G. Feitelson, L. Rudolph (Eds.). Job Scheduling Strategies for Parallel Processing, *Lecture Notes in Computer Science*, Vol. 1459, Springer-Verlag 1998.

[13] H. Franke, J. Jann, J.E. Moreira, P.Pattnaik, and M.A. Jette. An Evaluation of Parallel Job Scheduling for ASCI Blue-Pacific. In *Proc. Supercomputing 99*, Nov. 1999.

[14] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM: Parallel Virtual Machine – A Users Guide and Tutorial for Network Parallel Computing.* MIT Press, 1994.

[15] D.P. Ghormley, D. Petrou, S.H. Rodigues, A.M. Vahdat, and T.E. Anderson. GLUnix: a Global Layer Unix for a Network of Workstations. Software: Practice & Experience, 28(9):929–961, 1998.

[16] R. L. Graham. Bounds for Certain Multiprocessor Anomalies. *Bell System Technical Journal*, 45:1563-1581, Nov. 1966.

[17] T. P. Green. DQS 3.0.2 Readme/Installation Manual. *Supercomputer Computations Research Institute*, Florida State University, Tallahassee, Jun. 1995.

[18] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message–Passing Interface.* MIT Press, 1994.

[19] J. L. Gustafson and R. Todi. Conventional Benchmarks as a Sample of the Performance Spectrum. *Proc. 31st Hawaii International Conference on System Sciences*, 1998.

[20] L. A. Hall, A. S. Schulz, D. B. Shmoys and J. Wein. Scheduling to Minimize Average Completion Time: Off-line and On-Line Approximation Algorithms. *Mathematics of Operations Research*, 22:513-544, Aug. 1997.

[21] S. Herbert. Generic NQS - Free Batch Processing for UNIX. Academic Computing Services, the University of Sheffield, UK. http://www.shef.ac.uk/uni/projects/nqs/Generic-NQS/.

[22] M.A. Iverson, F. Özgüner, and L.C. Potter. Statistical Prediction of Task Execution Times Through Analytic Benchmarking for Scheduling in a Heterogeneous Environment. In *Proc. 8th Heterogeneous Computing workshop (HCW)*, Apr. 1999, San Juan, Puerto Rico.

[23] D. Karger, C. Stein and J. Wein. Scheduling Algorithms. In M. J. Atallah, editor, *Handbook of Algorithms and Theory of Computation*, *CRC Press*, 1997.

[24] T. Kielmann. Programming Heterogeneous Workstation Clusters based on Coordination. *Proc. 8th International Conference of Computing and Information*, Waterloo, Ontario, Canada, Jun. 1996. Published as special issue of the CD-ROM Journal of Computing and Information (JCI).

[25] P. Krueger and R. Chawla. The Stealth Distributed Scheduler. *Proc. IEEE 11th International Conference on Distributed Computing Systems (ICDCS)*, pp. 336–343, 1991.

[26] S. Leonardi and D. Raz. Approximating Total Flow Time with Preemption. In *Proc. of the 29th Annual ACM Symposium on Theory of Computing*, pp. 110–119, ACM, 1997

[27] D. A. Lifka, M. W. Henderson and K. Rayl. Users Guide to the Argonne SP Scheduling System. *Mathematics and Computer Science Division Technical Memorandum No. ANL/MCS-TM-201*, Argonne National Laboratory, USA, May 1995.

[28] M. J. Litzkow, M. Livny, and M. W. Mutka. Condor – A Hunter of Idle Workstations. *Proc. of the 8th International Conference on Distributed Computer Systems (ICDCS)*, pp. 104–111. IEEE, 1988.

[29] M. Livny and R. Raman. High-Throughput Resource Management. In *The GRID: Blueprint for a New Computing Infrastructure*, pp. 311–337, Morgan Kaufmann, 1998.

[30] C. A. Phillips, C. Stein and J. Wein. Scheduling Jobs that Arrive over Time. In S. G. Akl, editor, *Algorithms and Data Structures*, LNCS 955, pp. 86-97, Springer-Verlag, 1995.

[31] C. D. Polychronopoulos and D.J. Kuck. Guided Self-

Scheduling: A Practical Scheduling Scheme for Parallel Supercomputers. *IEEE Transactions on Computers*, Vol. C-36 (12):1425–1439, 1987.

[32] S. R. Presnell. The Batch Reference Guide. 3rd Edition, Batch Version 4.0, *Dept. of Pharmaceutical Chemistry*, University of California, Mar. 1994.

[33] J. Pruyne and M. Livny. Parallel Processing on Dynamic Resources with CARMI. In *Proc. International Parallel Processing Symposium (IPPS'95) Workshop on Job Scheduling Strategies for Parallel Processing*, LNCS 949, pp. 259–278, Springer-Verlag, 1995.

[34] J. Sgall. On-Line Scheduling - A Survey. In A. Fiat and G. J. Woeginger, editors, *LNCS 1442*, pp. 196-231, 1996, Springer.

[35] D. B. Shmoys, J. Wein and D. P. Williamson. Scheduling Parallel Machines On-line. *SIAM Journal on Computing*, 24:1313-1331, 1995.

[36] W. Smith, I. Foster, and V. Taylor. Predicting Application Run Times Using Historical Information. In *Proc. joint International Parallel Processing Symposium, Symposium on Parallel & Distributed Processing (IPPS/SPDP '98), Workshop on Job Scheduling Strategies for Parallel Processing*, 1998.

[37] Q. Snell, G. Judd, and M. Clement. Load Balancing in a Heterogeneous Supercomputing Environment *Proc. 1998 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'98)*, vol. II, pp.951-957, 1998.

[38] D. Talby and D.G. Feitelson. Supporting Priorities and Improving Utilization of the IBM SP Scheduler Using Slack-Based Backfilling. In *Proc. joint International Parallel Processing Symposium, Symposium on Parallel & Distributed Processing (IPPS/SPDP '99)*, Apr. 1999, San Juan, Puerto Rico.

[39] T.H. Tzen and L.M. Ni. Trapezoid Self-Scheduling: A Practical Scheduling Scheme for Parallel Compilers. *IEEE Transactions on Parallel and Distributed Systems*, 4(1):87–98, 1995.

[40] A. P. A. Vestjens. On-line Machine Scheduling. Ph.D. thesis, Eindhoven University of Technology, 1997.

[41] S. White, A. Alund and V. S. Sunderam. Performance of the NAS Parallel Benchmarks on PVM-Based Networks. *Journal of Parallel and Distributed Computing*, vol 26, pp. 61-71, 1995.

[42] S. Zhou. LSF: Load Sharing in Large-Scale Heterogeneous Distributed Systems. University of Toronto, Dec. 1992.