A Self-Tuning Job Scheduler Family with Dynamic Policy Switching

Achim Streit

PC^2- Paderborn Center for Parallel Computing, Paderborn University
33102 Paderborn, Germany
streit@upb.de
http://www.upb.de/pc2

Abstract. The performance of job scheduling policies strongly depends on the properties of the incoming jobs. If the job characteristics often change, the scheduling policy should follow these changes. For this purpose the dynP job scheduler family has been developed. The idea is to dynamically switch the scheduling policy during runtime. In a basic version the policy switching is controlled by two parameters.
The basic concept of the *self-tuning* dynP scheduler is to compute virtual schedules for each policy in every scheduling step. That policy is chosen which generates the 'best' schedule. The performance of the self-tuning dynP scheduler no longer depends on a adequate setting of the input parameters.
We use a simulative approach to evaluate the performance of the self-tuning dynP scheduler and compare it with previous results. To drive the simulations we use synthetic job sets that are based on trace information from four computing centers (CTC, KTH, PC2, SDSC) with obviously different characteristics.

1 Introduction

A modern resource management system for supercomputers consists of many different components which are all vital for the everyday usage of the machine. Despite the fact that the management software should be working properly, the scheduler plays a major role in improving the acceptance, usability, and performance of the machine. The performance of the scheduler could be seen as a quality of service with regard to the performance of the users jobs (e. g. wait and response time). But also the machine owner is interested in a good scheduler performance for e. g. increasing the utilization of the machine.

Hence, much work has been done in the field of improving or developing new scheduling algorithms and policies in general. Some examples to mention are: gang-scheduling [2] (combined with migration and backfilling [20]), several backfilling variants (conservative [4], EASY [13], or slack-based [18]), or a tool for predicting job runtimes [14]. Also, research for explicit machines was done, e. g. for the IBM SP2 and the LoadLeveler System [18, 11], or the IBM ASCI Blue [5, 10].

D.G. Feitelson et al. (Eds.): JSSPP 2002, LNCS 2537, pp. 1–23, 2002.

It is common to use simulation environments for evaluating scheduling algorithms. Job sets are often based on trace information from real machines. Especially for that purpose a Parallel Workload Archive [17] was established. During the last years such simulation environments were also used to evaluate scheduling algorithms for upcoming computational grid environments [1]. Admittedly no grid job trace is available at the moment.

In this paper we follow a similar approach: we built a simulation environment tailored for our resource management system CCS (Computing Center Software) [8]. In that simulation environment the exact scheduling process of CCS is modelled. Because real trace data from our *hpcLine* cluster exists, it is possible to develop and evaluate new scheduling algorithms. Our cluster is operated in space sharing mode, as our users often need the exclusive access to the network interface and the full compute power of the nodes. Three scheduling policies are historically grown: FCFS (first come, first serve), SJF (shortest jobs first) and LJF (longest jobs first), each supplemented with conservative backfilling.

In the following we present a scheduler family developed for our system. It is based on the three single policies and dynamically switches between them automatically and in real time. Furthermore, it offers self-tuning ability, so that no interaction or startup parameter is necessary. Besides a trace based job set from our machine (PC2), we also evaluated the algorithms with three other trace based job sets from the Cornell Theory Center (CTC), the Swedish Royal Institute of Technology (KTH) and the San Diego Supercomputer Center (SDSC).

The remainder of this paper is organized as follows: In the next section some related work is presented. In section 3 the algorithms are presented, starting with the basic variant and the self-tuning dynP scheduler with two different deciders. After that the used workloads are presented and examined in section 4. The evaluation in section 5 starts with a short look on the used performance metrics and proceeds with the results. We surveyed different aspects of the algorithms and also present a comparison with previous work. Finally this paper ends with a conclusion in section 6.

2 Related Work

Ramme and Gehring [12, 6] introduced the IVS (Implicit Voting System) for scheduling the MPP-systems of a virtual machine-room in 1996. The problem was that the systems were switched between batch and interactive mode manually at fixed times of the day (i. e. interactive mode during working hours and batch mode for the rest of the day and during weekends). This static solution restricted the usage of the systems very much. The idea was, that the users themselves should vote for the used scheduling method depending on the characteristics of their favored resource requests. However, the users should not vote explicitly. Therefore the IVS was developed. Three strategies are used as a basis: FCFS, FFIH (first fit, increasing height), and FFDH (first fit, decreasing height). FFIH sorts the request list by increasing estimated job runtime, so that

short (interactive) jobs are at the front. FFDH sorts the requests in a opposite order than FFIH. Using FFIH leads to a shorter average waiting time in general, whereas FFDH commonly improves the overall system utilization. Hence the basic idea of IVS is to check, whether more batch or interactive jobs are in the system. Depending on that, IVS switches between FFDH (more batch jobs) or FFIH (more interactive jobs). If the system is not saturated, FCFS is used. IVS was never implemented and tested in a real environment, as the project finished with the Ph.D. thesis.

Feitelson and Naaman [3] published work about self-tuning systems in 1999. Modern operating systems are highly parameterized, so that the administrative staff is forced to use a trial-and-error approach for optimizing these parameters. A better way would be to automate this process. The idea is to look at past information (i. e. log files), use this information as input for simulations with different parameter values, and evaluate them. Genetic algorithms are set in to derive new parameter values. With these genetic parameter values again simulations are driven in the idle loop of the machine to conduct a systematic search for optimal parameter values. The authors call such systems which learn about their environment *self-tuning*, as the system itself automatically searches for optimized parameter values. In a case study for scheduling batch jobs on a iPSC/860 they found out that with the self-tuning search procedure the overall system utilization can be improved from 88% (with the default parameters) to 91%. That means, that the number of resources lost to fragmentation is reduced by one quarter.

3 Algorithms

In this section the different versions of the dynP (for **dyn**amic **P**olicy) scheduler family and the history of development are presented. We start with the basic dynP scheduler which needs a lower and upper bound as parameters. Then follows the self-tuning dynP scheduler with an introduction to the simple decider and its disadvantages. Finally the new, advanced decider for the self-tuning dynP scheduler is presented.

3.1 The Basic dynP Scheduler

We started our work with two job sets which were derived from traces of our 96-node hpcLine cluster. This machine is managed by CCS which is a long-term running project at the PC^2. All policies are combined with (conservative) backfilling [4]. Currently, CCS is configured to use FCFS for scheduling jobs. Now the question is, has performance suffered, because we have used FCFS instead of SJF or LJF. So we have developed a simulation framework for evaluating the three scheduling policies with two trace based job sets from our machine.

The results show, that FCFS is a good average for both job sets [15]. The other policies show opposing results: for the first job set SJF is better than FCFS and LJF is worst, and for the second job set, LJF is the best, followed by

FCFS and SJF is worst. From that we have developed the idea of dynamically switching the scheduling policy during runtime. A decision criterion was needed to decide when to switch from one policy to the other. For that we have used the average estimated runtime of all jobs currently in the waiting queue. The decider is evoked every time a new job is submitted and the algorithm works as follows:

```
basic_dynP_algorithm()
{
   IF (jobs in waiting queue >= 5) {
      AERT = average estimated runtime of all jobs currently in the waiting queue;
      IF (0 < AERT <= lower_bound) { switch to SJF; }
      ELSE IF (lower_bound < AERT <= upper_bound) { switch to FCFS; }
      ELSE IF (upper_bound < AERT) { switch to LJF; }
      reorder waiting queue according to new policy;
   }
}
```

Note, we are using a threshold of 5 jobs to prevent unnecessary policy switches, if the waiting queue is too short. An experimental search is done to find appropriate parameter values for the two bounds. With the right settings this basic dynP scheduler outperforms FCFS for both job sets. The different behavior for the two job sets (with the same bounds) is obvious when looking at the usage of the three policies. For the first job set (with more short jobs) FCFS and SJF is used more than three quarters of the whole schedule time. The second job set consisted of more long jobs, so that LJF was used most of the time.

3.2 The Self-Tuning dynP Scheduler

One problem still is, that a long lasting trial-and-error process is needed, to find proper values for the two bounds. The fact that our simulation environment is now working with full schedules[1] and inspired by Feitelsons and Naamans work about self-tuning systems [3], brought us to the idea of the self-tuning dynP scheduler: Let the scheduler generate full schedules for each of the three strategies in every scheduling step. And switch to that policy which generates the best schedule for the current situation.

```
core_self_tuning_dynP_algorithm()
{
   mySchedule = JobQueue.generateSchedule("FCFS");
   FCFS = mySchedule.getQuality(QualityParameter);
   mySchedule = JobQueue.generateSchedule("SJF");
   SJF = mySchedule.getQuality(QualityParameter);
   mySchedule = JobQueue.generateSchedule("LJF");
   LJF = mySchedule.getQuality(QualityParameter);
   // call the decider
   switch_to_best_policy(FCFS,SJF,LJF);
}
```

The *quality parameter* in getQuality() specifies the metrics for evaluating the virtual schedule and is one of the following:

[1] A newly submitted job is directly placed in the schedule and a proposed starttime is assigned to it. This feature allows to work with advanced reservations.

– Makespan (MS)

$$MS = \max_{j \in Jobs} j.EndTime$$

– Average Response Time (ART)

$$ART = \frac{\sum_{j \in Jobs} (j.EndTime - j.SubmitTime)}{|Jobs|}$$

– Average Response Time weighted by Width (ARTwW)

$$ARTwW = \frac{\sum_{j \in Jobs} (j.requestedResources * (j.EndTime - j.SubmitTime))}{\sum_{j \in Jobs} j.requestedResources}$$

Note, that the best schedule has the lowest quality number. Preliminary evaluations have shown that using the average response time weighted by job width leads to good results for different workloads. By weighting the response time with the job width, jobs requesting more resources have a greater influence on the quality of the schedule. Otherwise small, often insignificant jobs would have the same influence as such large jobs. The initial policy is FCFS.

A Simple Decider At first we use a quite simple decider mechanism for the switch_to_best_policy() method. In [16] we show that the self-tuning dynP scheduler combined with this simple decider achieves only average results for the two trace based job sets. The simple decider works as follows:

```
switch_to_best_policy(FCFS,SJF,LJF) {
   // the simple decider
   IF (SJF <= LJF) {
      IF (FCFS <= SJF) { newPolicy = "FCFS"; }
      ELSE { newPolicy = "SJF"; }
   } ELSE {
      IF (FCFS <= LJF) { newPolicy = "FCFS"; }
      ELSE { newPolicy = "LJF"; }
   }
}
```

When looking at how often each policy was used during the whole schedule to start jobs, we found out that the simple decider preferred to use FCFS and SJF, but seldom uses LJF. Therefore, we analyzed all possible combinations of schedule quality numbers (cf. Tab. 1). Note, that we are using the following abbreviations in the table:

– with FCFS (SJF and LJF respectively) we mean the quality (ARTwW) of the schedule generated with FCFS
– the three symbols $<$, $=$, $>$ are used for comparing the quality numbers

For example: FCFS $<$ SJF means that the FCFS generated schedule has a lower ARTwW than the SJF generated schedule and is therefore better. Note, case 4 is split up for counting the cases (cf. Tab. 6) and better understanding.

Table 1. Behavior of the single and advanced decider for all combinations of schedule quality numbers

case		combinations	simple decider	advanced decider
1		FCFS = SJF = LJF	**FCFS**	**current policy**
2		SJF < FCFS, SJF < LJF	SJF	SJF
3		FCFS < SJF, FCFS < LJF	FCFS	FCFS
4		LJF < FCFS, LJF < SJF		
	a	FCFS < SJF	LJF	LJF
	b	FCFS = SJF	LJF	LJF
	c	FCFS > SJF	LJF	LJF
5		FCFS = SJF, LJF < FCFS (→ LJF < SJF)	LJF	LJF
6		FCFS = SJF, FCFS < LJF (→ SJF < LJF)		
	a	current policy = FCFS	current policy	current policy
	b	current policy = SJF	**FCFS**	**current policy**
	c	current policy = LJF	FCFS	FCFS
7		FCFS = LJF, SJF < FCFS (→ SJF < LJF)	SJF	SJF
8		FCFS = LJF, FCFS < SJF (→ LJF < SJF)		
	a	current policy = FCFS	current policy	current policy
	b	current policy = SJF	FCFS	FCFS
	c	current policy = LJF	**FCFS**	**current policy**
9		SJF = LJF, FCFS < SJF (→ FCFS < LJF)	FCFS	FCFS
10		SJF = LJF, SJF < FCFS (→ LJF < FCFS)		
	a	current policy = FCFS	SJF	SJF
	b	current policy = SJF	current policy	current policy
	c	current policy = LJF	**SJF**	**current policy**

The Advanced Decider In Tab. 1 four cases are marked (cases: 1, 6b, 8c, and 10c) with bold fonts. In these cases the simple decider favors FCFS (three times) and SJF. We developed a new, more advanced decider which generates decisions as found in the last column. Note, that in three cases no exact decision is possible based on the quality of the three schedules and the current policy:

- case 6c: the current policy (LJF) needs to be changed, as it is obviously the worst. FCFS or SJF can be taken, as both are equal. We choose FCFS, as it might be beneficial for the average response time of the generated schedule.
- case 8b: similar to case 6c, but FCFS or LJF can be taken.
- case 10a: similar to case 6c, but SJF or LJF can be taken. We choose SJF for preferring short jobs.

4 Workloads

In previous work we used two job sets which were 3-month traces from our machine of the year 2000. They consist of roughly 8 000 jobs and are described more detailed in [15]. Now for this work we generated a job trace of the complete year 2001. Additionally, we downloaded three job traces from Feitelsons Parallel Workload Archive [17] which were already used in many other publications before. We used traces from the Cornell Theory Center (CTC), the Swedish

Royal Institute of Technology (KTH), and the San Diego Supercomputer Center (SDSC). All logs were derived from IBM SP2 machines. Besides the standard job information (e. g. job width, submit time, etc.) these three traces also hold information about the estimated runtime of the jobs. This information is essential when working with a backfilling scheduler.

The four job traces were then analyzed to build synthetic job sets with 10 000 jobs each. These synthetic job sets retained the characteristics of the original traces (e. g. unused nodes during the night or weekends). This mechanism of analyzing the trace and generating new jobs from the obtained information has some advantages:

1. Job sets of various sizes (number of jobs) can be generated. This method is used here.
2. The information from the trace analysis can be modified to generate job sets with different characteristics.
3. Other modifications can easily be applied, e. g. large jobs are split up in width, so that the maximum job width is only 64, but the total area of all jobs is not changed.

The complete process of analyzing the trace and generating synthetic job sets is described in [9]. The four most important job properties to be analyzed are: submission time (more precisely: interarrival time), width (number of requested resources), estimated and actual runtime. The submission time of all jobs is best expressed by a Weibull distribution ($f(x) = 1 - e^{-(\frac{x}{\beta})^{\alpha}}$). The three other properties could not be expressed by any distribution although they are dependent on each other. Hence, a 3-dimensional matrix is generated which holds probability values for all possible combinations of width, estimated and actual runtime. Tab. 2 shows the output of the trace analyzer. Note that except for PC2_trace jobs with a longer actual than estimated runtime occurred in the traces. CCS kills jobs that try to run longer than estimated.

The job generator is started with the information of the trace analyzer. When a new job should be generated, first the submission time is computed from a random number and the Weibull distribution of job interarrival times. Another three random numbers and the probability matrix for the width, estimated and actual runtime are used to generate the three missing parameters of a new job. This way of generating job sets was also used in [7, 1] before. The four job sets we constructed are described in Tab. 3. Note that again the maximum values for the actual runtime are often greater than estimated, so this property was taken over from the traces. Of course our simulation environment kills all jobs, that try to run longer than estimated.

Fig. 1 shows distributions for the estimated and actual runtimes. The staircase shape of the estimated runtime curves indicates that users tend to use common or round values, (e. g. PC2_syn: 10 minutes (600 seconds), 30 minutes (1800 seconds), or 2 hours (7200 seconds). These estimates are used for about 52% of all jobs). The curves for the actual runtime are smoother, as jobs usually have different runtimes and do not end at specific times. However steps are also

Table 2. Output of the trace analyzer

		CTC_trace	KTH_trace	PC2_trace	SDSC_trace
number of jobs		79 302	28 490	35 094	67 667
maximum job width		336	100	96	128
width of machine (batch)		430	100	96	352
average requested resources		10.72	7.68	6.34	10.53
estimated runtime	avg	24 324 s	13 677 s	11 716 s	14 337 s
	min	0 s	60 s	1 s	0 s
	max	64 800 s	216 000 s	1 209 600 s	172 800 s
actual runtime	avg	10 983 s	8 876 s	4 346 s	6 119 s
	min	0 s	0 s	1 s	0 s
	max	71 998 s	226 709 s	604 800 s	510 209 s
jobs with actual > estimated runtime		7 180 (= 9.05 %)	478 (= 1.68 %)	0 (= 0 %)	4 327 (= 6.39 %)
interarrival time	avg	369 s	1 031 s	870 s	934 s
	min	0 s	0 s	0 s	0 s
	max	164 472 s	327 952 s	313 861 s	79 503 s
likelihood of interarrival time		0.0063932	0.0116883	0.0659942	0.0104926
Weibull: α		0.35	0.35	0.25	0.4
Weibull: β		60	200	40	290
original scheduler		EASY	EASY	FCFS + cons. backfilling	EASY

Table 3. Properties of the four synthetic job sets

		CTC_syn	KTH_syn	PC2_syn	SDSC_syn
number of jobs		10 000	10 000	10 000	10 000
maximum job width		330	100	96	128
width of machine (batch)		430	100	96	352
average requested resources		10.67	7.73	6.48	10.71
estimated runtime	avg	24 260 s	13 643 s	11 516 s	14 305 s
	min	0 s	60 s	1 s	0 s
	max	64 800 s	216 000 s	604 800 s	64 800 s
actual runtime	avg	10 924 s	9 060 s	4 442 s	6 085 s
	min	0 s	0 s	1 s	0 s
	max	71 998 s	215 965 s	604 800 s	64 884 s
jobs with actual > estimated runtime		929 (= 9.29 %)	178 (= 1.78 %)	0 (= 0 %)	640 (= 6,40 %)
interarrival time	avg	287 s	1 047 s	1 013 s	945 s
	min	0 s	0 s	0 s	0 s
	max	24 815 s	191 258 s	236 546 s	76 019 s

visible, e.g. for PC2_syn and the 2 hour mark. This indicates that many jobs that were estimated to run for 2 hours also ended at that time. Likely these jobs were underestimated, as CCS kills jobs that try to run longer than estimated.

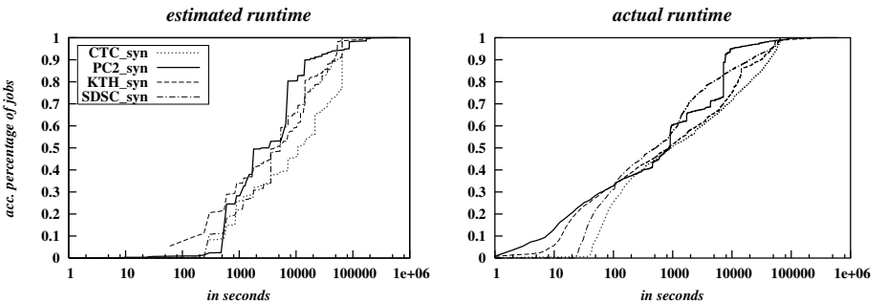

Fig. 1. Distributions of job runtime for the four synthetic job sets

5 Evaluation

It is common practice to evaluate new or modified scheduling algorithms with simulations before they are actually deployed in a real environment. For that we developed a simulation framework, called MuPSSiE (= Multi Purpose Scheduling Simulation Environment).

Today, many scheduling systems only look at the current time and try to start as many waiting jobs as possible.[2] With that a scheduler cannot specify a starting time for a submitted job. A unique feature of our simulation environment is that it always generates the full schedule for all jobs in the system. Two advantages of such an approach are:

1. Proposed start times are assigned to jobs right after their submission as they are directly placed in the schedule.
2. Advanced reservations with guaranteed start times are possible.

Like backfilling schedulers such an approach needs information about the job's estimated runtime. Reservations also need information about their start time, which could either be a keyword (now, asap) or a time/date string. All jobs without reservations are called *variable jobs* as they can be started at any time. A rescheduling process is done every time a job ends. At that time, every running and pending job is placed into the schedule. The order in which variable jobs are newly placed is specified by the scheduling policy. Note that the scheduling policy has no influence on reservations.

Generating the full schedule is also the basis for the self-tuning scheduler. In principle it would be possible to evaluate this algorithm with job sets which contain variable jobs and reservations. But as no real job/reservation traces exist, we concentrate only on variable jobs in this context.

The simulation framework basically is a set of stand-alone tools, like job set converters/modifiers, job set and schedule analyzers, a schedule viewer, a single

[2] Of course, information about the near future (i. e. runtime of already running jobs) is also needed when backfilling is applied.

machine scheduler (with several policies and the dynP algorithm implemented), and a multi-site grid scheduler that can work with advanced reservations. It is developed and implemented in ANSI C++ on a Windows system (Borland C++ Builder), but the simulation runs are done on Linux systems. The execution time of a single simulation run is between some seconds and up to 24 hours (on AMD Athlon XP 1800+ systems). This strongly depends on the length of the average backlog, as the longer the backlog gets, the longer the rescheduling process takes. Note, that the self-tuning dynP scheduler has to compute two additional schedules in each step for comparing the three policies. A detailed survey on the execution time of a self-tuning step showed, that it took about 0.05 seconds for an average of 185 jobs (CTC_syn, shrinking factor of 0.80) to find the best scheduling policy. For resource management systems like CCS, where 2 minutes are estimated for constructing a partition of nodes, checking and cleaning them, the execution time of the self-tuning step can be neglected, even if it grows.

For evaluating the self-tuning scheduler and the advanced decider we did simulation runs with all combinations of all three quality metrics and the four synthetic job sets. Additional simulation runs for the three single policies (FCFS, SJF, LJF) as an evaluation basis were done.

We also used a *shrinking factor* to decrease the average interarrival time. With that the 10 000 jobs are submitted in a shorter time. The consequences are: the average backlog grows, the utilization increases, and jobs wait longer for their start.

5.1 Metrics

Measuring the performance of a scheduler respectively the quality of the generated schedule can be done with different metrics. In general they can be classified in two groups: owner centric and user centric. Owner centric criteria mainly focus on the schedule as a whole, e. g. the makespan, utilization, loss of capacity [19], or number of jobs processed over a given time period. Machine owners make use of such numbers to document how good/effective their machine was used or to emphasize the need for a faster machine. The machine users are generally not interested in such numbers, as they only want the output of their jobs as soon as possible. For them metrics like the average response or wait time or the slowdown of their jobs is interesting. But the requirements of both groups sometimes overlap. Owners may be interested in short response times for their users, as this somehow represents a quality of service they provide.

In this work we use three main metrics in the evaluation: the utilization of a schedule, the average response time (ART) and the average job slowdown (SLD). We further weight the jobs response time and slowdown with its width for giving larger jobs a larger influence. Additionally we bound the slowdown by 60 seconds, so that very short jobs are not considered [4].

Utilization (N = number of total resources):

$$UTIL = \frac{\sum_{j \in Jobs} (j.RequestedResources * j.RunTime)}{N * (lastEndTime - firstSubmitTime)}$$

Average response time weighted by job width:

$$ARTwW = \frac{\sum_{j \in Jobs} (j.RequestedResources * j.ResponseTime)}{\sum_{j \in Jobs} j.RequestedResources}$$

Average Slowdown weighted by width and bound by 60 seconds:

$$SLDwW\_60 = \frac{\sum_{j \in Jobs} (j.RequestedResources * j.Slowdown)}{\sum_{j \in Jobs} j.RequestedResources}$$

with:

- $j.RunTime = j.EndTime - j.StartTime$
- $j.ResponseTime = j.EndTime - j.SubmitTime$
- $j.Slowdown = \frac{max(j.ResponseTime, 60)}{max(j.RunTime, 60)}$

Later we will show diagrams where these three metrics are plotted on the y-axis for different shrinking factors on the x-axis. Note, that each data point in a diagram refers to one simulation run. Points connected by a line are simulation runs with equal configurations (quality parameter, decider) and job input. Only the shrinking factor is decreased to simulate a higher workload for the scheduler.

5.2 Results

In the following we present the performance of the self-tuning dynP scheduler with the advanced decider. We compare these results against the simple decider and the three basic strategies. When talking about metrics (e. g. ARTwW) we will use the following definitions: With 'quality metrics' we mean the metrics used in the self-tuning scheduler for measuring the three schedules in each step (cf. Sec. 3.2). But with 'performance metrics' we are talking about measuring the complete schedule after the simulation ended. Additionally, when 'response time' is written, the average response time weighted by job width (ARTwW) of the whole schedule is meant.

Comparison of Simple and Advanced Decider At first we compare the results with three quality metrics: ART, ARTwW, and MS (Makespan). In Tab. 4 the most important criteria are presented. In the first column different performance metrics for measuring the complete simulated schedule are given. With

LOC we mean the loss of capacity and when the system is in a saturated state the equation *loss of capacity = 1 - utilization* holds. A high loss of capacity indicates that many resources are lost, as jobs are waiting and they could not be started (they need more resources than available). A small loss of capacity indicates that either no jobs were waiting or the machine was fully utilized. The numbers in the table were obtained using a shrinking factor of 1.00. The columns show the performance numbers with the simple or advanced decider and a quality metrics for the self-tuning function. Finally, the last three rows of each block show the number of jobs started with each of the three policies during the whole scheduling process.

Table 4. Comparing the results (rows) of the simple and advanced decider with shrinking factor = 1.00 and three different quality metrics. Note, the numbers in 'jobs started with ...' are also 1/100th % of all jobs

	ARTwW		ART		MS	
CTC_syn	simple	advanced	simple	advanced	simple	advanced
ARTwW	25 980 s	21 912 s	26 082 s	21 648 s	27 268 s	25 829 s
UTIL	75.36 %	75.70 %	74.02 %	74.96 %	75.64 %	75.08 %
LOC	0.12650	0.10911	0.13731	0.12099	0.09054	0.10395
jobs started with SJF	2 703	6 115	2 472	5 860	130	551
jobs started with FCFS	6 817	1 296	6 418	1 144	5 295	953
jobs started with LJF	477	2 586	1 107	2 993	4 572	8 493
KTH_syn						
ARTwW	48 077 s	33 299 s	49 301 s	40 663 s	50 497 s	50 395 s
UTIL	66.00 %	65.64 %	66.01 %	65.64 %	66.14 %	66.11 %
LOC	0.16398	0.16456	0.17121	0.18386	0.16588	0.16629
jobs started with SJF	2 865	7 885	2 768	5 730	142	322
jobs started with FCFS	6 854	1 345	6 313	1 313	5 951	686
jobs started with LJF	278	767	916	2 954	3 904	8 989
PC2_syn						
ARTwW	43 124 s	31 493 s	42 795 s	31 994 s	47 747 s	47 260 s
UTIL	42.60 %	42.49 %	42.60 %	42.58 %	42.60 %	42.60 %
LOC	0.22423	0.23792	0.22147	0.23937	0.22724	0.22722
jobs started with SJF	2 553	7 550	2 615	6 651	65	391
jobs started with FCFS	7 344	1 062	7 106	766	6 789	229
jobs started with LJF	103	1 388	279	2 673	3 146	9 380
SDSC_syn						
ARTwW	9 969 s	9 936 s	9 963 s	9 921 s	9 982 s	9 991 s
UTIL	31.32 %	31.32 %	31.32 %	31.32 %	31.32 %	31.32 %
LOC	0.00133	0.00119	0.00135	0.00124	0.00135	0.00135
jobs started with SJF	317	3 924	309	1 031	1	0
jobs started with FCFS	9 561	4 206	9 517	4 797	9 570	3 087
jobs started with LJF	121	1 869	173	4 171	428	6 912

Obviously the SDSC_syn job set is not really useful, as the schedule seems to be quite empty (low utilization), and the schedules do not change very much with different deciders or quality metrics (e. g. the response times are almost the same). Combined with the low utilization the loss of capacity shows, that almost no jobs were waiting when resources could be utilized.

When looking at the user-centric performance metric ARTwW the advanced decider is almost always better than the simple decider, with only one exception: MS as quality metrics and the SDSC_syn job set (by 0.09%). The maximum benefit is reached with the KTH_syn job set and quality metrics ARTwW, where the response time gets better by 30.74%. Also a user-centric quality metrics for the self-tuning scheduler improves the response time for the whole simulated schedule more than the utilization. On the other hand the MS quality metrics does not improve the total utilization that much. Because the job area does not change, a shorter makespan directly results in a higher utilization. Possibly the utilization can easier be improved with the MS quality metrics, if a higher workload is used (shrinking factor < 1.00). Overall, the ARTwW quality metrics generates the best results for KTH_syn and PC2_syn, only for CTC_syn ART is slightly better.

The usage of the policies over the time shows that the simple decider favors FCFS regardless which quality metrics or job set was used. Especially when the utilization should be increased, it is better to use the LJF policy. This behavior is seen for the advanced decider with the MS quality metrics as over about 7 000 jobs are started with LJF. In case of the two user-centric quality metrics ARTwW and ART, only between 750 and 1 350 jobs are started with FCFS (for CTC_syn, KTH_syn and PC2_syn). Here SJF is the best choice for optimizing the schedulers performance (both for response time and utilization), as about two-thirds of all jobs are started with that policy. In general the advanced decider achieves better results than the simple decider, especially when focusing on user-centric performance metrics. And this does not depend on the used quality metrics (even when using MS as quality metrics). Except for the KTH_syn job set this also holds for the owner-centric performance metrics. The advanced decider does not favor any specific policy, like it was with the simple decider and FCFS. Hence, completely different numbers for the jobs started with each policy are achieved with the advanced decider. This shows, that the self-tuning scheduler with the advanced decider really reacts on different job set characteristics.

Detailed Look at the Advanced Decider After this brief performance overview we now take a more detailed look at the advanced decider. In Tab. 5 we used a shrinking factor of 1.00 and ARTwW as quality metrics for the self-tuning function. Again some average values, the utilization, loss of capacity, and the number of jobs started with each policy are shown. The number of switches to each policy, the number of cases with the same policy as before, and the average size of the backlog (length of waiting queue) at the start of the self-tuning function is printed.

Table 5. Detailed results of the self-tuning dynP scheduler and the advanced decider using ARTwW as quality metrics. The numbers are equal to the third column of Tab. 4

	CTC_syn	KTH_syn	PC2_syn	SDSC_syn
average actual runtime	10 927 s	9 063 s	4 442 s	6 085 s
average wait time	3178 s	6 925 s	4 577 s	44.02 s
ART	14 105 s	15 989 s	9 018 s	6 129 s
ARTwW	21 912 s	33 299 s	31 493 s	9 936 s
utilization	75.70%	65.64%	42.49%	31.32%
LOC	0.10911	0.16456	0.23792	0.00119
jobs started with SJF	6 115	7 885	7 550	3 924
jobs started with FCFS	1 296	1 345	1 062	4 206
jobs started with LJF	2 586	767	1 388	1 869
total tries of switches	16 946	18 840	15 552	509
switches to SJF	4.49 %	5.80 %	3.61 %	3.73 %
switches to FCFS	4.59 %	5.85 %	3.64 %	3.93 %
switches to LJF	0.59 %	0.45 %	0.23 %	0.98 %
same policy	90.32 %	87.92 %	92.52 %	91.36 %
average backlog at start of self-tuning	22.05	14.48	15.04	6.36

Like before the SDSC_syn job set is not very helpful for our evaluations, as jobs only wait some 44 seconds in average, which is roughly 0.7% of the average actual runtime. For the other job sets the percentage are 35% for CTC_syn, 84% for KTH_syn and even 91% for PC2_syn. Also the numbers for the 'total tries of switches' and 'average backlog at start of self-tuning' show, that submitted jobs in the SDSC_syn job set are started right away, without getting delayed. Additionally the low number of total tries of policy switches shows that the self-tuning functionality only makes sense, when more than one job is in the schedule and not yet running. Later we will use the shrinking factor for increasing the workload with the SDSC_syn job set to achieve useful results.

We now take a look at the number of policy switches. Obviously almost all calls ($> 90\%$) to the advanced decider (total tries of switches) result in using the same policy as before, although the average size of the backlog at the start of the self-tuning function is considerably large (again except for SDSC_syn). When a decision is made and the new policy is different from the current policy, most of the switches are done between SJF and FCFS for all four job sets. Only a minority of policy switches is done to LJF. With the fact that most of the jobs are started with SJF, the advanced decider seems to use FCFS only for one or two job starts and then switches back to SJF. But if it was decided to switch to LJF, many jobs are started with that policy. So the switches to LJF are quite effective and useful, and the decisions to switch to FCFS are often undone after a short time.

We also did a case analysis, where the occurrence of each case in the decision algorithm (cf. Tab. 1) is counted. Tab. 6 shows the results. Over two-thirds of all cases are 6b (FCFS is equal to SJF, LJF is worse, the current policy is SJF

Table 6. Case analysis of the advanced decider using ARTwW as quality metrics. Refer to Tab. 1 for details

	CTC_syn	KTH_syn	PC2_syn	SDSC_syn
case: 1	2 549	1 693	2 209	271
cases: 2 & 7	1 324	1 756	920	27
cases: 3 & 9	692	1 029	535	15
cases: 4b & 5	1 298	403	304	31
case: 4a	5	5	2	1
case: 4c	4	3	0	0
case: 6a	81	116	58	8
case: 6b	10 669	13 664	11 449	150
case: 6c	100	81	36	5
case: 8a	224	89	39	1
case: 8b	0	1	0	0
case: 8c	0	0	0	0
case: 10a	0	0	0	0
case: 10b	37	26	17	1
case: 10c	0	0	0	0
total	16 946	18 840	15 552	509
differences to the simple decider (1+6b+8c+10c)	13 218 (=78.00%)	15 357 (=81.51%)	13 658 (=87.82%)	421 (=82.71%)
taking the current policy (1+5+6a+6b+8a+8c+10b+10c)	14 858 (=87.68%)	15 932 (=84.56%)	14 076 (=90.51%)	462 (=90.76%)

and SJF is chosen). Later, Fig. 4 shows, that SJF is the best single policy for a shrinking factor of 1.00. Hence, the advanced decider chooses SJF most of the time. And as FCFS is not much worse than (or equal to) SJF, it can often happen that FCFS and SJF generate equal schedules in a scheduling step, so the current policy SJF is used further on. Case 6b also effects the percentages of different decisions to the simple decider and of cases, where the current policy is taken, so that both are above 78%.

Looking at the three critical cases (6c, 8b, and 10a) from Sec. 1 (a new policy has to be taken but the decider can choose from two) shows that case 6c is seldom used ($< 1\%$) and the other two never. With that the advanced decider only favors FCFS in an insignificant number of cases, so that the advanced decider can be called fair with regard to the used policies.

Performance at Different Workloads The following shrinking factors for the interarrival time are used to generate varying workloads:

- CTC_syn: from 1.00 down to 0.65
- KTH_syn: from 1.00 down to 0.60
- PC2_syn: from 0.80 down to 0.40
- SDSC_syn: from 0.60 down to 0.30

Note, these values for the shrinking factor are chosen for generating useful performance numbers for medium utilizations. If the values are further decreased

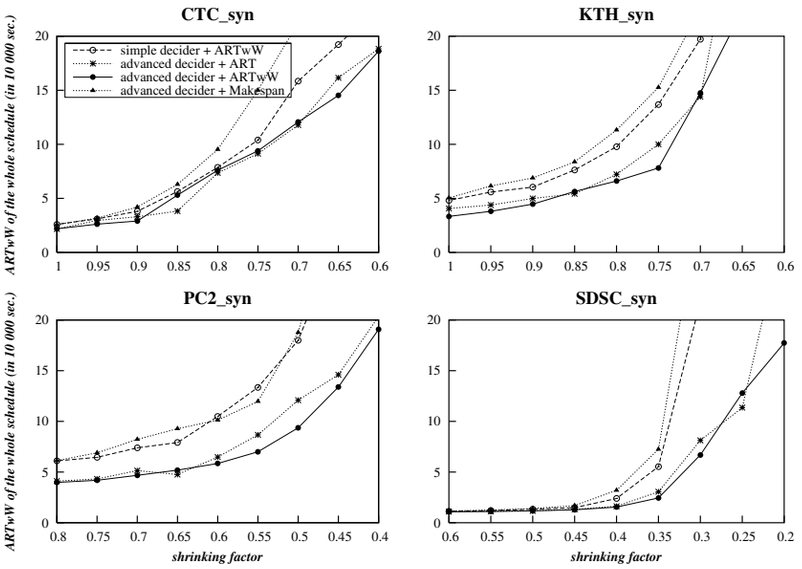

Fig. 2. Average Response Time weighted by width (ARTwW) of complete schedules: comparing the three quality parameters for the advanced and single decider. Each data point refers to a single simulation run using different shrinking factors. The legend applies to all four diagrams. Smaller values are better

the saturated state is reached. In the diagrams we plot the shrinking factor on the x-axis and one of the three metrics introduced in Sec. 5.1 on the y-axis. Each data point refers to a single simulation run. Reducing the shrinking factor (and thereby the average interarrival time) increases the workload for the scheduler.

In Fig. 2 the performance of the simple and advanced decider with the three quality metrics from Sec. 3.2 is compared. The quality metrics ART and ARTwW generate better results than the makespan criterion, which better improves the utilization (not shown in a diagram). As can be seen in Tab. 4 the MS quality parameter for the advanced decider prefers to take LJF. LJF (combined with backfilling) is known for improving the utilization, as it allows the backfilling routine to start jobs in the holes of the LJF schedule. Comparing the simple and advanced decider for the ARTwW quality parameter shows, that the advanced decider is mostly superior when looking at the response times. Noticeable improvements are achieved for the PC2_syn job set, where the advanced decider is approximately 30% better than the simple decider with the same quality metrics. For KTH_syn the improvements are still at 20% whereas for CTC_syn both deciders are not that much different.

If a different metrics (slowdown) is used for the comparison (Fig. 3 only shows job sets CTC_syn and PC2_syn) especially for the CTC_syn job set the order changes. Only in that case the simple decider achieves smaller (i.e. better)

slowdown values than the advanced decider. Whereas for PC2_syn the difference between the simple and advanced decider even grows for small shrinking factors, and the slowdown value remains almost constant over a wide range of shrinking factors. Note, the PC2_syn job set generates 10 times higher slowdowns than CTC_syn. In the following we are concentrating on the quality parameter ARTwW for the self-tuning scheduler with the advanced decider.

In three diagrams (performance metric ARTwW in Fig. 4, SLDwW_60 in Fig. 5, and utilization in Fig. 8) we compare the advanced decider with quality metrics ARTwW, the three single strategies FCFS, SJF, and LJF (each combined with conservative backfilling), and the basic dynP (with 7200s for the lower and 9000s for the upper bound). The bounds for basic dynP were the best in [15] for two job sets extracted form the PC2 workload. Here the old bound setting for the basic dynP scheduler performs bad for the four synthetic job sets. This shows that in a real world environment the system administrator would have to re-adapt these bounds once in a while. Especially when the overall job set characteristic often change, a self-tuning scheduler is easier to handle. The single policy SJF combined with conservative backfilling achieves the lowest response times and slowdowns for all four job sets and also over a wide range of shrinking factors. For maximum utilization LJF is the best choice. In some way this is interesting, FCFS is commonly used in many resource management systems, as it might

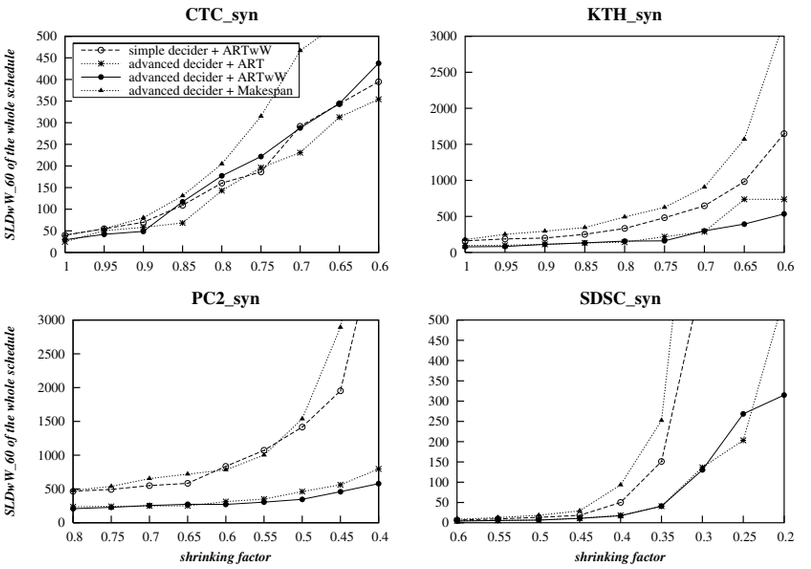

Fig. 3. Slowdown weighted by width and bound by 60 seconds (SLDwW_60) of complete schedules. Similar diagrams as in Fig. 2 but with a different metrics on the y-axis. Again, smaller values are better

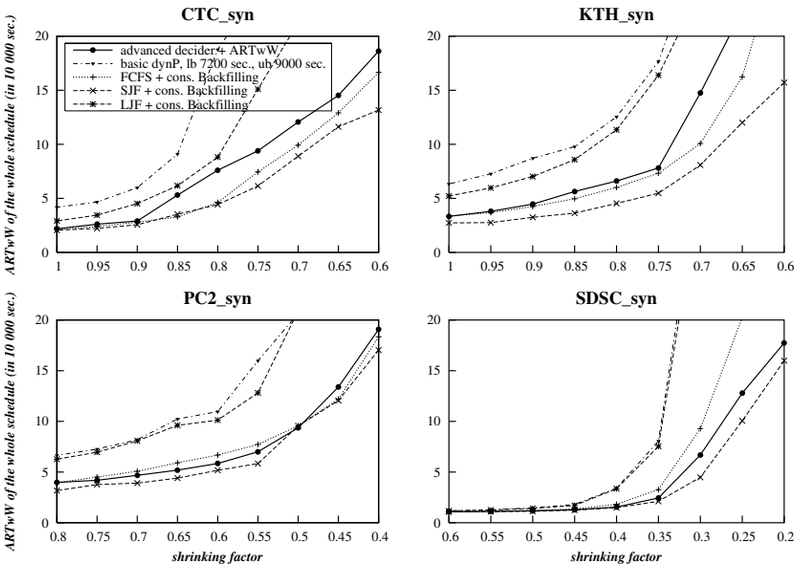

Fig. 4. ARTwW of complete schedules: comparing the advanced decider with quality parameter ARTwW with the three single policies FCFS, SJF, and LJF (each combined with conservative backfilling) and basic dynP with lower bound 7200s and upper bound 9000s (this setting achieved the best results in [15])

be a good tradeoff between low response times and high utilizations on many machines.

For the PC2_syn job set the advanced decider (with the ARTwW quality parameter) achieves a similar performance than SJF, but cannot outperform it. For the CTC_syn job set and smaller shrinking factors than 0.9 the performance drops significantly. The advanced decider cannot compete with FCFS and even reaches the performance of the taillight LJF. For the other two job sets the advanced decider generally follows FCFS. Note, these statements are true for response time (ARTwW in Fig. 4) as well as for slowdown (SLDwW_60 in Fig. 5).

The diagrams for the utilization (Fig. 8) shows that for small shrinking factors all schedulers achieve similar or equal utilizations. As soon as the saturated state is reached, the values differ. Obviously the LJF scheduler achieves the highest utilizations (95% and more), as the sorting of the waiting queue might leave much room for the backfilling method.

It is very interesting and at the same time unexplainable that the self-tuning scheduler with the advanced decider and quality parameter ARTwW can follow SJF and/or FCFS for 3 of the 4 job sets, but not for CTC_syn.

Pure curiosity lead us to the idea of computing the slowdown for different actual runtimes separately. Arbitrarily we used the following categorization: $\leq$5m, $\leq$10m, $\leq$1h, $\leq$12h, and $\geq$12h. Almost 50% of all jobs fall in the first class with a

maximum actual runtime of 5 minutes. Therefore we analyzed the results again and now computed the average slowdown weighted by width with a 300 seconds (= 5 minutes) bound. The diagrams in Fig. 7 (SLDwW_300) are similar to the 60 second bound version, except that the scaling on the y-axis has changed to smaller values. This shows, that very short running jobs (roughly 50% of all jobs) have a great influence on the overall quality of the schedule, but they have no influence on the ranking of the schedulers itself. Still, these small jobs are considered in the self-tuning step (Sec. 3.2). So another solution would be to neglect these small jobs there, too, but the problem is, that the self-tuning process only knows about runtime estimates (and not actual values).

One possible reason for the poor performance of the self-tuning scheduler could be false or inaccurate runtime estimates. Therefore we ran the simulations again, but now with each jobs estimated runtime set to the actual runtime.

Comparing the diagrams in Fig. 6 with Fig. 4 shows only a small performance gain of the advanced decider. SJF is still the best choice and the self-tuning scheduler comes close to its performance. But the perfect estimates diagram show a different interesting aspect: good runtime estimates become more important with an increased workload (i. e. smaller shrinking factor; saturated state). Exemplary we take a look at the SJF curve and the CTC_syn job set. Anyhow the response time starts at about 20.000 seconds (for 1.00 as shrinking factor), with either real or perfect estimates used. As soon as the workload increased (e. g. to a shrinking factor of 0.7) the difference is remarkable (102.725 s for per-

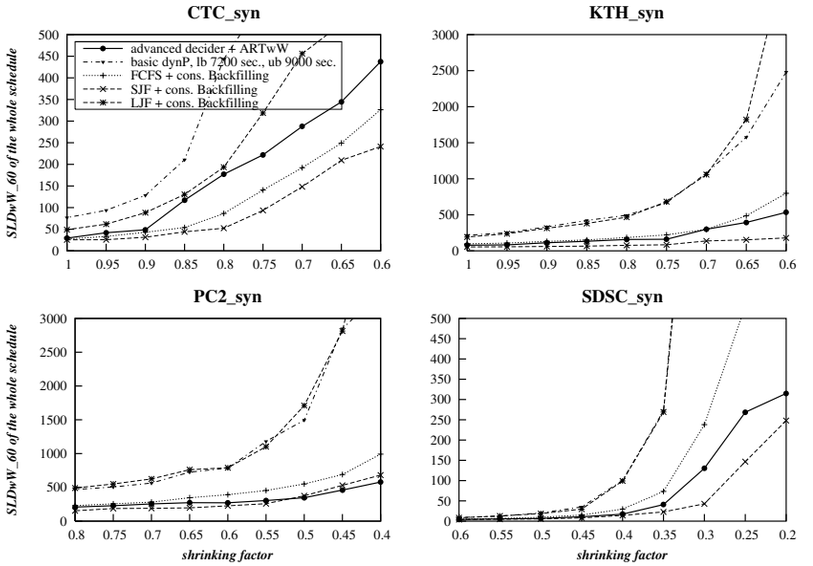

Fig. 5. SLDwW_60 of complete schedules. Similar diagrams as in Fig. 4, but with a different metrics on the y-axis

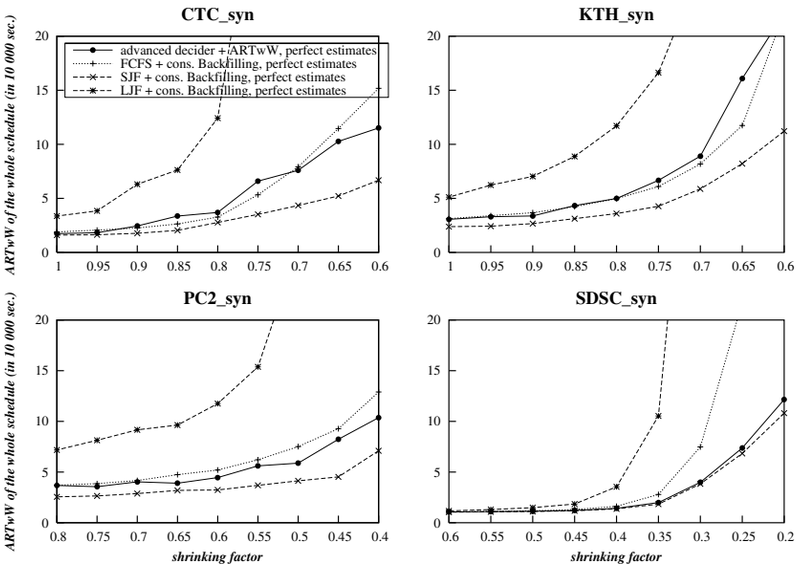

Fig. 6. Similar diagrams as in Fig. 4 (response time), but now with perfect estimates, i.e. estimated runtime = actual runtime. Note, the curve for basic dynP is left out

fect and 145.212 s for real estimates). Even bigger differences can be observed for the PC2_syn job set.

6 Conclusions and Future Work

In this paper we presented the process of developing a job scheduler family with dynamic policy switching and self-tuning ability. The basic idea of self-tuning is to generate complete schedules for each of the three policies (FCFS, SJF, LJF) in each scheduling step. Then these schedules are measured by means of a quality metrics and that policy is chosen, which generates the best schedule. We use the average response time (unweighted and weighted by the job width) and the makespan as quality metrics.

At first we developed a simple decider mechanism which already achieved reasonable good results for two trace job sets. When investigating the behavior of that decider in detail, we found out that this decider sometimes prefers FCFS when other policies obviously should have been taken. Therefore we developed the advanced decider.

We evaluated the self-tuning dynP scheduler with this advanced decider in a simulation environment. To have more diversity in the results we used four synthetic job sets, which are based on real trace logs from four different computing centers. Additionally we decreased the average interarrival time between two jobs to generate a higher workload for the scheduler.

The results show that the advanced decider achieves a better performance (up to 30%) than the simple decider using the same quality metrics. The best quality metrics to use is ARTwW. A future deployment of the algorithm in the resource management software CCS will show, whether the simulated results can be achieved, or the influence of overestimation is too large. A real world deployment also has to show, whether the non-predictability of the scheduler discourages users from working with the system or not. Additionally users might submit fake jobs (long estimated runtime and short or none actual runtime) to trick the system, so that their real computing job is favored.

Future versions of the self-tuning dynP scheduler might come with several enhancements. With a 'slackness' option the schedule quality of the current schedule is virtually increased by e. g. 5%. Hence a new policy has to be better than the current policy by 5%. With a 'reduced future' option enabled, the qualities of the three schedules in each self-tuning step are only computed from e. g. the first 20 started jobs or all jobs that are started within the next 6 hours. Thereby jobs which will be started far in the future are not considered. Additionally more scheduling policies (e. g. FFIA) and quality parameters (e. g. slowdown and wait time) might be added. Also combining several quality parameters for the self-tuning process might improve the performance. A desirable performance curve for the self-tuning dynP scheduler would always stay below or is equal to the best single policy at any workload.

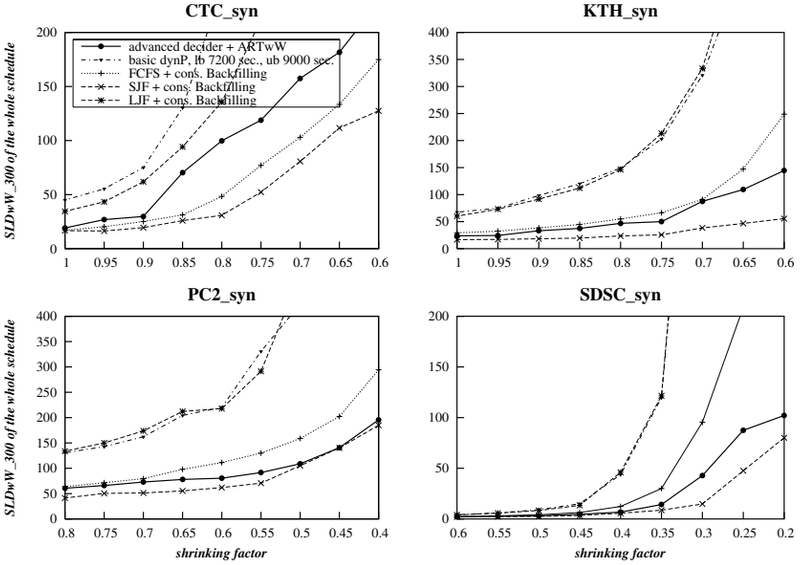

Fig. 7. Similar diagrams as in Fig. 5 (slowdown), but now with a bound of 300 seconds

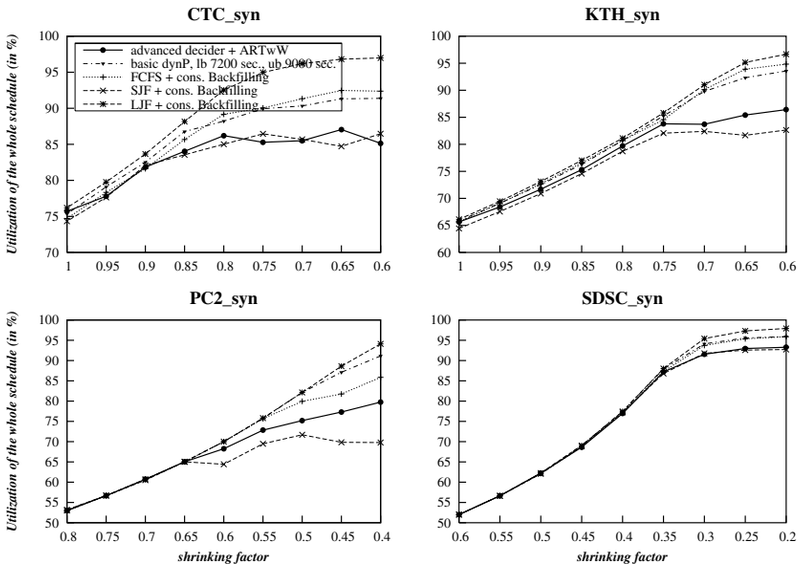

Fig. 8. Utilization of complete schedules. Similar diagrams as in Fig. 4, but with a different metric on the y-axis. Note, only here higher values on the y-axis are better.

References

[1] C. Ernemann, V. Hamscher, U. Schwiegelshohn, A. Streit, and R.Yahyapour. On Advantages of Grid Computing for Parallel Job Scheduling. In *Proceedings of the 2nd IEEE International Symposium on Cluster Computing and the Grid (CCGRID 2002)*, 2002. 2, 7

[2] D. G. Feitelson and M. A. Jette. Improved Utilization and Responsiveness with Gang Scheduling. In D. G. Feitelson and L. Rudolph, editor, *Proc. of 3rd Workshop on Job Scheduling Strategies for Parallel Processing*, volume 1291, pages 238–262. Springer Verlag, 1997. 1

[3] D. G. Feitelson and M. Naaman. Self-Tuning Systems. In *IEEE Software 16(2)*, pages 52–60, April/May 1999. 3, 4

[4] D. G. Feitelson and A. Mu'alem Weil. Utilization and Predictability in Scheduling the IBM SP2 with Backfilling. In *Proceedings of the 1st Merged International Parallel Processing Symposium and Symposium on Parallel and Distributed Processing (IPPS/SPDP-98)*, pages 542–547, Los Alamitos, March 1998. IEEE Computer Society. 1, 3, 10

[5] H. Franke, J. Jann, J. Moreira, P. Pattnaik, and M. Jette. An Evaluation of Parallel Job Scheduling for ASCI Blue-Pacific. In *Proceedings of SC'99, Portland, Oregon*, pages 11–18. ACM Press and IEEE Computer Society Press, 1999. 1

[6] J. Gehring and F. Ramme. Architecture-Independent Request-Scheduling with Tight Waiting-Time Estimations. In D. G. Feitelson and L. Rudolph, editor, *Proc. of 2nd Workshop on Job Scheduling Strategies for Parallel Processing*, volume 1162, pages 65–80. Springer Verlag, 1996. 2

[7] V. Hamscher, U. Schwiegelshohn, A. Streit, and R. Yahyapour. Evaluation of job-scheduling strategies for grid computing. In *Proceedings of 1st IEEE/ACM International Workshop on Grid Computing (Grid 2000)*, volume 1971, pages 191–202, 2000. 7

[8] A. Keller and A. Reinefeld. Anatomy of a Resource Management System for HPC Clusters. In *Annual Review of Scalable Computing, vol. 3, Singapore University Press*, pages 1–31, 2001. 2

[9] J. Krallmann. A Quantative Analysis of Scheduling-Algorithms for Parallel Machines (in German). Master's thesis, Dortmund University, 1998. 7

[10] J. E. Moreira, H. Franke, W. Chan, and L. L. Fong. A Gang-Scheduling System for ASCI Blue-Pacific. In *Proceedings of the 7th International Conference in High-Performance Computing and Networking (HPCN'99)*, volume 1593 of *Lecture Notes in Computer Science*, pages 831–840. Springer, 1999. 1

[11] A. W. Mu'alem and D. G. Feitelson. Utilization, Predictability, Workloads, and User Runtime Estimates in Scheduling the IBM SP2 with Backfilling. In *IEEE Trans. Parallel & Distributed Systems 12(6)*, pages 529–543, June 2001. 1

[12] F. Ramme and K. Kremer. Scheduling a Metacomputer by an Implicit Voting System. In *3rd Int. IEEE Symposium on High-Performance Distributed Computing*, pages 106–113, 1994. 2

[13] J. Skovira, W. Chan, H. Zhou, and D. Lifka. The EASY — LoadLeveler API Project. In D. G. Feitelson and L. Rudolph, editor, *Proc. of 2nd Workshop on Job Scheduling Strategies for Parallel Processing*, volume 1162, pages 41–47. Springer Verlag, 1996. 1

[14] W. Smith, I. Foster, and V. Taylor. Predicting Application Run Times Using Historical Information. In D. G. Feitelson and L. Rudolph, editor, *Proc. of 4th Workshop on Job Scheduling Strategies for Parallel Processing*, volume 1459, pages 122–142. Springer Verlag, 1998. 1

[15] A. Streit. On Job Scheduling for HPC-Clusters and the dynP Scheduler. In *Proceedings of the 8th International Conference on High Performance Computing (HiPC 2001)*, volume 2228 of *Lecture Notes in Computer Science*, pages 58–67. Springer, December 2001. 3, 6, 17, 18

[16] A. Streit. The Self-Tuning dynP Job-Scheduler. In *Proceedings of the 11th International Heterogenous Computing Workshop (HCW) at IPDPS 2002*, Lecture Notes in Computer Science, 2002. 5

[17] Parallel Workloads Archive. http://www.cs.huji.ac.il/labs/parallel/workload/, February 2002. 2, 6

[18] D. Talby and D. G. Feitelson. Supporting Priorities and Improving Utilization of the IBM SP2 Scheduler Using Slack-Based Backfilling. TR 98-13, Hebrew University, Jerusalem, April 1999. 1

[19] Y. Zhang, H. Franke, J. E. Moreira, and A. Sivasubramaniam. Improving Parallel Job Scheduling by Combining Gang Scheduling and Backfilling Techniques. In D. G. Feitelson and L. Rudolph, editor, *Proceedings of the 14th International Conference on Parallel and Distributed Processing Symposium (IPDPS-00)*, pages 133–144. Springer Verlag, 2000. 10

[20] Y. Zhang, H. Franke, J. E. Moreira, and A. Sivasubramaniam. An Integrated Approach to Parallel Scheduling Using Gang-Scheduling, Backfilling, and Migration. In D. G. Feitelson and L. Rudolph, editor, *Proc. of 7th Workshop on Job Scheduling Strategies for Parallel Processing*, volume 2221, pages 133–158. Springer Verlag, 2001. 1