Enhancements to the Decision Process of the Self-Tuning dynP Scheduler

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

Abstract

The self-tuning dynP *scheduler for modern cluster resource management systems switches between different basic scheduling policies dynamically during run time. This allows to react on changing characteristics of the waiting jobs. In this paper we present enhancements to the decision process of the self-tuning* dynP *scheduler and evaluate their impact on the performance: (i) While doing a self-tuning step a performance metric is needed for ranking the schedules generated by the different basic scheduling policies. This allows different objectives for the self-tuning process, e. g. more user centric by improving the response time, or more owner centric by improving the makespan. (ii) Furthermore, a self-tuning process can be called at different times of the scheduling process: only at times when the characteristics of waiting jobs change (half self-tuning), i. e. new jobs are submitted; or always when the schedule changes (full self-tuning), i. e. when jobs are submitted or running jobs terminate.*

We use discrete event simulations to evaluate the achieved performance. As job input for driving the simulations we use original traces from real supercomputer installations. The evaluation of the two enhancements to the decision process of the self-tuning dynP *scheduler shows that a good performance is achieved, if the self-tuning metric is the same as the metric used measuring the overall performance at the end of the simulation. Additionally, calling the self-tuning process only when new jobs are submitted, is sufficient in most scenarios and the performance difference to full self-tuning is small.*

1 Introduction

Resource management systems (RMS) for modern high performance computing (HPC) clusters consist of many components which are all vital in keeping the machine fully operational. An efficient usage of the machines is important for users and owners, as such systems are rare and high in cost. With regards to performance aspects all components of a modern RMS should perform their assigned tasks efficiently and fast, so that no additional overhead is induced. However, if resource utilization and job response times are addressed, the scheduler plays a major role. A clever scheduling strategy is essential for a high utilization of the machine and short response times for the jobs. However, these two objectives are contradicting. Jobs tend to have to wait for execution on a highly utilized system with space sharing. Short or even no waiting times are only achievable with low utilizations or time-sharing. Typically a scheduling policy that optimizes the utilization prefers jobs needing many resources for a long time. Jobs requesting few resources for a short amount of time may have to wait longer until adequate resources are available. If such small and short jobs are preferred by the scheduler, the average waiting time would be reduced. As jobs typically have different sizes and lengths, fragmentation of the schedule occurs and the utilization drops [1]. The task of the scheduler is to find a good compromise between optimizing these two contrary metrics.

Cluster systems usually have a large user community with different resource requirements and general job characteristics [3]. For example, some users primarily submit parallel and long running jobs, whilst others submit hundreds of short and sequential jobs. Furthermore, the arrival patterns vary between specific user groups. Hundreds of jobs for a parameter study might be submitted in one go via a script. Other users might only submit their massively parallel jobs one after the other. This results in a non-uniform workload and job characteristics that permanently change. The job scheduling policy used in a RMS is chosen in order to achieve a good overall performance for the expected workload. Most commonly used is first come first serve (FCFS) combined with backfilling [7, 14, 9], as on average a good performance for the utilization and response time is achieved. However, with certain job characteristics other scheduling policies might be superior to FCFS. For example, for mostly long running jobs, longest job first (LJF) is beneficial, whilst shortest job first (SJF) is used with mostly short jobs [1]. Hence, a single policy is not enough for an efficient resource management of clusters. Many modern RMSs have several scheduling policies implemented, or it is even possible to replace the scheduling component.

The remainder of this paper is structured as follows. In Section 2 some related work on self-tuning and dynamic policy switching is given. Section 3 starts with a short history of development, contains the concept of the self-tuning dynP scheduler, and presents the different decider mechanisms and enhancements to them. The used performance metric and the applied workload for the evaluation are presented in Section 4. The evaluation in Section 5 starts with a look on the performance of the three basic policies. Then the evaluation results of the different self-tuning metrics and of the comparison of half vs. full self-tuning are presented. The paper ends with conclusions an a brief outlook on future work.

2 Related Work

In [13] the problem of scheduling a machine room of MPP-systems is described. Users either submit long running batch jobs or they work interactively (typically only for a short time). To accomplish this on a single MPP-system the resource management system has to switch from batch mode (preferring batch jobs) to interactive mode (preferring interactive jobs) and back. Usually this is done manually by the administrative staff, e. g. at fixed times of the day: interactive mode during working hours, batch mode for the rest of the day and over weekends. In general, the overall job throughput is the main objective of batch processing. As batch jobs typically have a long run time, waiting is not very critical. On the other hand, a user that works interactively counts the five minutes until he/she can start working with the requested resources. Other issues like the overall job throughput or the utilization are less important while operating in interactive mode. Which in comparison to batch mode jobs are rather short. The idea [4] is to allow the users to decide in which mode the system should be operating. Hence, the Implicit Voting System (IVS) is introduced, as users should not vote explicitly:

- If most of the waiting jobs are submitted for batch processing, IVS switches to LJF (longest job first). As batch jobs are typically long, they receive a higher priority in the scheduling process. Hence, resources are longer bound to jobs, less resource fragmentation is caused and the utilization and throughput of the system is increased.

- If most of the waiting jobs are submitted for interactive access, IVS switches to SJF (shortest job first). As interactive jobs are usually short in their run time and short jobs are preferred, the average waiting time is reduced.

- If the system is not saturated, the default scheduling strategy FCFS (first come first serve) is used. Note, a threshold for defining when a system is saturated and when not is defined by the administrative staff. For the authors a MPP system becomes saturated, if more than five jobs can not be scheduled immediately.

Unfortunately, the idea of IVS was never realized nor implemented and tested in a real environment.

In [3] a similar approach for the NASA Ames iPSC/860 system is presented. In the prime time during the day only a fraction of the resources is allocated to the batch partition, while most of the resources are available for interactive access. During non prime time all resources are assigned to the batch partition. The re-partitioning is done manually and at fixed times of the day.

The problem of getting the best performance from a modern resource management system is described in [2]. Commonly such software systems are highly parameterized and the administrative staff performs a lot of trial and error testing in order to find a good parameter setting for the current workload. If the workload changes, new parameter settings have to be found. However, they are notoriously overworked and have little or no time for this fine tuning, so the idea is to automate this process. Much information about the current and past workload is available, which is used to run simulations in the idle loop of the system (or on a dedicated machine). Various parameter settings are simulated and the best setting is chosen. The authors call such a system *self-tuning*, as the system itself searches for optimized parameter settings. To create new parameter settings for the simulations, genetic algorithms are used. New parameter settings are generated by randomly combining several potential combinations from the previous step. Chromosomes are the binary representation of a parameter. A parameter setting is called individual and the according parameter values are concatenated in their binary representation. In this example the fitness function is the average utilization of the system achieved by the according parameter setting. All simulated parameter settings (individuals) in one step represent a generation. The chromosomes of the fittest individuals of a generation are used to produce new individuals for the next generation. New generations are continuously created with the latest system workload as input. The process is started with default values. In a case study for scheduling batch jobs of the NASA Ames iPSC/860 system the authors observed that with the self-tuning search for parameter settings the overall system utilization is improved from 88% (with the default parameters) to 91%.

In [8] heuristics for the dynamic mapping of a class of independent tasks onto heterogeneous computing systems are introduced. The mapping problem consist of two parts: matching and scheduling. In the matching phase the assignment of tasks to machines is computed, whilst during scheduling the execution order of the tasks on each machine is computed. In their work a dynamic mapping is needed, as the arrival times of the tasks may be random and the set of available machines is changing. Machines may go off-line and new machines may come on-line. Mapping heuristics can be grouped in two classes: on-line mode and batch-mode heuristics. In the on-line mode tasks are mapped onto a machine immediately after their submission. In the batch-mode, tasks are not immediately mapped when they arrive, instead they are stored and the mapping is invoked at pre-scheduled mapping events. The heuristic MCT (minimum completion time) assigns each task to that machine which results in the task's earliest completion time. Tasks may be assigned to machines, which do not have the minimum execution time. In contrast, MET (minimum execution time) assigns each task to that machine that performs the task in the least amount of execution time. As the machines ready times are not considered by MET, load imbalance across the machines may occur. The new batch-mode SA (switching algorithm) heuristic uses these two heuristics (MCT and MET) in a cyclic fashion depending on the load distribution across the machines. By switching between MCT and MET a new heuristic with the desirable properties of the two single heuristics is generated.

3 Self-Tuning dynP

A single scheduling policy is usually used in a resource management system and it typically generates good schedules only for jobs with specific characteristics (e. g. short jobs). If the job characteristics change, other scheduling policies might perform better and it might be beneficial that the system administrator changes the scheduling policy. However, system administrators are not able to monitor the situation and continuously alter the scheduling policy in response to workload changes.

We developed the dynP scheduler, which automatically switches the active scheduling policy during run time. In general, the set of scheduling policies to choose from can consist of many policies one can think of. We started with a variant of the dynP scheduler, which uses bounds for the average estimated run time of waiting jobs to check, which policy is best suited for the current job characteristics. A major drawback of this version is obvious, as the performance depends on a proper setting of the bounds. And in order to reflect different job characteristics, these bounds need to be changed. We developed the self-tuning dynP scheduler which automatically searches for the best suited policy.

3.1 Concept

At the PC^2 (Paderborn Center for Parallel Computing) the self-developed resource management system CCS (Computing Center Software, [6]) is used for managing the *PSC* Pentium3 cluster [12] and the *pling* cluster [11]. Three scheduling policies are currently implemented: FCFS, SJF, and LJF. According to the classification in [5], CCS is a planning based resource management system. Planning based resource management systems schedule the present and future resource usage, so that newly submitted jobs are placed in the active schedule as soon as possible and they get a start time assigned. With this approach backfilling is done implicitly. By planning the future resource usage, a sophisticated approach is possible for finding a new policy. For all waiting jobs the scheduler computes a full schedule, which contains planned start times for every waiting job in the system. With this information it is possible to measure the schedule by means of a performance metric (e. g. response time, slowdown, or makespan). The concept of self-tuning dynP is:

> The self-tuning dynP scheduler computes full schedules for each available policy (here: FCFS, SJF, and LJF). These schedules are evaluated by means of a performance metric.

Thereby, the performance of each policy is expressed by a single value. These values are compared and a decider mechanism chooses the best value, i. e. the smallest value.

In the following, the performance metric used in the self-tuning process is called self-tuning metric for simplicity.

3.2 Decider Mechanisms

For the required decision, several levels of sophistication are thinkable. In [16] we presented the *simple decider* that basically consists of three if-then-else constructs. It chooses that policy which generates the minimum value of the applied self-tuning metric. The simple decider also has drawbacks, as it does not consider the old policy. In particular if two policies are equal and a decision between them is needed, information about the old policy is helpful. Table 5 shows a detailed analysis of the simple decider. FCFS is favored in three and SJF in one case, although staying with the old policy is the correct decision with these cases. This behavior is implemented in the *advanced decider*. At a first glance it does not make any difference which policy among equals is chosen. At this stage the scheduler only knows estimates of the job's run time and usually the job's actual run time is shorter than estimated. When a job ends earlier than estimated, the schedule changes and new planning is necessary. Depending on the chosen policy different jobs might have been started in the meantime. Therefore, even a decision between two equal policies is required. Previously, the fairness among the policies was of major interest. However, it might be interesting to explicitly prefer one of the policies and neglect the others. For that purpose we developed the *preferred decider* [17]. The preferred policy is not switched unless any other policy is better. Whenever any of the other policies are currently used, the preferred policy only has to achieve an equal performance and the decider switches back.

The deciders of the self-tuning dynP scheduler consider only the three policies FCFS, SJF, and LJF for three reasons. First of all, we evaluate the general behavior and performance of self-tuning dynP schedulers for the resource management of HPC systems. We do not evaluate, which combination of policies is best suited for specific job characteristics. Presumably, combinations with other and more scheduling policies exist, which generate even better results. Secondly, FCFS, SJF, and LJF are the most known scheduling policies and many resource management systems have at least these three implemented. And thirdly, these three policies are implemented in the resource management software CCS, which depicts the basis and starting position for our work.

In previous work we already presented the basics of the self-tuning dynP scheduler. It started with the simple decider in [16]. Next, we developed the advanced decider [15] and recently the preferred decider [17]. In this paper we present further enhancements to the decision process, which can be applied to all three mentioned decider mechanisms.

3.3 Enhancements to the Decision Process

As previously mentioned the aim of the self-tuning dynP scheduler is to eliminate input parameters for the scheduler, especially those which depend on the characteristics of the processed jobs and need to be re-adapted continuously. Nevertheless, enhancements that influence the scheduler in a more general way are thinkable. Of course they should be independent of any job characteristics and easy to handle, so that a continuous manual re-adaptation is not needed.

Different Self-Tuning Metrics

The concept of the self-tuning dynP scheduler is based on the planning-based scheduling approach, where all waiting jobs are placed in a schedule. With assigning a proposed start time to each job, it is possible to compute the waiting time of the jobs, so that schedules can be compared. For this, different self-tuning metrics can be applied, e. g. user centric metrics like the average response time or the slowdown (both unweighted or weighted), and owner centric metrics like the makespan. By choosing a specific metric, the self-tuning dynP scheduler optimizes its behavior according to this metric. We use the following metrics, which are all defined in the next section: average response time (ART), average response time weighted by area (ARTwA), average response time weighted by width (ARTwW), average slowdown (SLD), average slowdown weighted by area (SLDwA), average slowdown weighted by width (SLDwW) and makespan.

Calling of Self-Tuning

Doing self-tuning and potentially switching the active scheduling policy, should be done whenever the schedule or the set of waiting jobs changes, i. e. whenever a new job is submitted or an already running job terminates earlier than estimated. In the following, we call this *full self-tuning*. However, it might also be sufficient to do the self-tuning step only when new jobs are submitted, i. e. the characteristics of waiting jobs in the system change. We call this option *half self-tuning*, as roughly only half as much self-tuning calls are performed. This option is interesting to evaluate in combination with the compute time to do the self-tuning process. It might be beneficial to save up some compute time and make the scheduling behavior more comprehensible for the users.

4 Evaluation

It is common practice to use discrete event simulations for the evaluation of job-scheduling strategies. For this purpose we developed MuPSiE (Multi Purpose Scheduling Simulation Environment). Several policies and the planning-based scheduling approach from [5] are implemented. All presented results are generated with MuPSiE.

4.1 Performance Metrics

We use the user centric slowdown metric for measuring the simulated schedules with all processed jobs. The slowdown of a job is also often called stretch [10] or relative response time, as the jobs response time is divided by the jobs run time. The slowdown comes without a dimension in contrast to e. g. response time. Additionally, we weight each job's slowdown with its area. Thereby, it is circumvented that jobs with the same run time, but with different resource requirements, have the same impact on the overall performance.

With the parameters for a finished job i of a total of m processed jobs:

- t_i^a is the arrival or submission time
- t_i^s is the start time
- t_i^e is the end time
- w_i is the width (number of requested/used resources)
- $l_i = t_i^e - t_i^s$ is the length (run time, duration)
- $t_i^w = t_i^s - t_i^a$ is the waiting time
- $t_i^r = t_i^w + l_i$ is the response time
- $s_i = \frac{t_i^r}{l_i} = 1 + \frac{t_i^w}{l_i}$ is the slowdown
- $a_i = w_i \cdot l_i$ is the area

The average slowdown weighted by job area (SLDwA) for all jobs is defined as:

$$SLDwA = \frac{\sum_{i=1}^{m} a_i \cdot s_i}{\sum_{i=1}^{m} a_i} \tag{1}$$

If the processed jobs are not changed in their width or run time, the average slowdown weighted by job area is equal to the average response time weighted by job width and the equation holds:

$$SLDwA = ARTwW \cdot \frac{\sum_{i=1}^{m} w_i}{\sum_{i=1}^{m} a_i} \tag{2}$$

For completeness, the other metrics used during the self-tuning process are defined as follows:

- the average response time:

$$ART = \frac{1}{m} \cdot \sum_{i=1}^{m} t_i^r \tag{3}$$

- the average response time weighted by job area:

$$ARTwA = \frac{\sum_{i=1}^{m} a_i \cdot t_i^r}{\sum_{i=1}^{m} a_i} \tag{4}$$

- the average response time weighted by job width:

$$ARTwW = \frac{\sum_{i=1}^{m} w_i \cdot t_i^r}{\sum_{i=1}^{m} w_i} \tag{5}$$

- the average slowdown:

$$SLD = \frac{1}{m} \cdot \sum_{i=1}^{m} s_i \tag{6}$$

- the average slowdown weighted by job width:

$$SLDwW = \frac{\sum_{i=1}^{m} w_i \cdot s_i}{\sum_{i=1}^{m} w_i} \tag{7}$$

- the makespan:

$$\max_{i=1,\ldots,m} t_i^e \tag{8}$$

4.2 Workload

An evaluation of job scheduling policies requires to have job input. In this work a job is defined by the submission time, the number of requested resources (= width), and the estimated run time (= length). As we model a planning based resource management system [5], run time estimates are mandatory. Additionally, for the simulation the actual run time is needed.

In this paper, we use four traces from the Parallel Workloads Archive [18], as all other traces do not come with information about run time estimates. The characteristics of the four traces are shown in Table 1 (taken from [17]).

- **CTC** (Cornell Theory Center), system: 512-node IBM SP2 (only 430 nodes are available for batch processing), duration: July 1996 - May 1997, jobs: 79,302
- **KTH** (Swedish Royal Institute of Technology), system: 100-node IBM SP2, duration: October 1996 - August 1997, jobs: 28,490
- **LANL** (Los Alamos National Lab), system: 1024-node Connection Machine CM-5 from Thinking Machines, duration: October 1994 - September 1996, jobs: 201,387
- **SDSC** (San Diego Supercomputing Center), system: 128-node IBM SP2, duration: May 1998 - April 2000, jobs: 67,667

5 Results

We start with presenting the results for the three basic policies FCFS, SJF, and LJF in short. This gives a good reference for the subsequent evaluations. At first, the results of the comparison of different self-tuning metrics are presented. A comparison of half and full self-tuning for the mentioned decider mechanisms follows. Finally, an analysis of the switching behavior for the simple and advanced decider is done.

5.1 Basic Policies

As previously stated, the evaluation shows that none of the policies is the best for every job set characteristic. In Table 2 the best basic policy with respect to the average slowdown weighted by area is highlighted with bold font. Particularly for the SDSC trace, the differences in slowdown are large as SJF is worse than FCFS by a factor of almost two and LJF is even worse (twice as bad as FCFS).

Backfilling is done implicitly with the planning-based scheduling approach for all three basic scheduling policies.

	FCFS	SJF	LJF
CTC	2.0455	**1.9277**	2.5212
KTH	3.1015	**2.5488**	5.8118
LANL	**1.6801**	1.7031	2.0507
SDSC	**6.8260**	12.5662	26.8207

Table 2. Overall average slowdown weighted by area (SLDwA) for the three basic policies FCFS, SJF, and LJF.

5.2 Different Self-Tuning Metrics

In the following, the results with different self-tuning metrics are presented. We used the following user centric metrics: average response time (ART), average response time weighted by area (ARTwA), average response time weighted by width (ARTwW), average slowdown (SLD), average slowdown weighted by area (SLDwA), and average slowdown weighted by width (SLDwW). Additionally, the owner centric metric makespan is used.

One can assume that the best performance is achieved by using the same metric during the self-tuning process and after the simulation is finished. As stated earlier, we use the average slowdown weighted by area (SLDwA) metric for measuring all simulated jobs. Hence, using SLDwA during self-tuning should lead to the best performance (i. e. the smallest values). This is seen in the upper part of Table 3. For the CTC, LANL, and SDSC trace the best self-tuning metric is SLDwA and the mentioned expectation is fulfilled. However, it is interesting that for the KTH trace using the job's width as a weight is slightly (0.8%) better than using the job's area as a weight. A possible reason for this is the overestimation factor, which is smaller for the KTH trace (1.5) than for the other three (2.2, cf. Table 1).

Using all other self-tuning metrics results in a significantly worse performance. In particular this is seen for the ARTwA metric and the SDSC trace, as the achieved performance is twice as bad. Observing the numbers for the ARTwW and the SLDwA self-tuning metric shows that the numbers are equal. This is a result of Equation 2 and the fact that the processed jobs are not changed in their width or run time.

The bottom part of Table 3 shows the overall utilization with all simulated jobs. Independent of the applied self-tuning metric the exact same utilizations

trace	requested resources min	avg.	max	estimated run time [sec.] min	avg.	max	actual run time [sec.] min	avg.	max	average overest. factor	interarrival time [sec.] min	avg.	max
CTC	1	10.72	336	0	24,324	64,800	0	10,958	64,800	2.220	0	369	164,472
KTH	1	7.66	100	60	13,678	216,000	0	8,858	216,000	1.544	0	1,031	327,952
LANL	32	104.95	1,024	1	3,683	30,000	1	1,659	25,200	2.220	0	509	201,006
SDSC	1	10.54	128	2	14,344	172,800	0	6,077	172,800	2.360	0	934	79,503

Table 1. Basic properties of the used traces (86,400 seconds = 1 day).

are generated for the traces CTC, KTH, and LANL. This indicates that those jobs submitted towards the end of the schedule are always scheduled at the same start time and are responsible for the makespan and therefore for the utilization. Only for the SDSC trace different utilizations are achieved, as the differences of the basic policies are large (cf. Table 2). Using the owner centric metric makespan leads to the best utilization. This matches to the results with SLDwA, as makespan and utilization are connected via the total sum of job areas and the totally available resources on the simulated machine.

These observations reflect the different switching behavior of the self-tuning dynP scheduler, if different performance metrics are applied. It is possible to tune the system performance in either way: user or owner centric. Using either owner or user centric metrics in the self-tuning process to generate good overall results for opposing metrics, i. e. user and owner, generally leads to a poor performance and should be avoided.

Comparing the best self-tuning metric with the best basic policy from Table 2 shows that only for the CTC and LANL trace the self-tuning dynP scheduler is better. The performance loss of the self-tuning dynP scheduler is marginal (0.4%) for the KTH trace, but almost 200% for the SDSC trace. This is a result of the overestimation of the job's run time by the users and the large differences of the basic policies. This misleads the self-tuning dynP scheduler in the decision process and results in wrong decisions. Although this also happens with the other traces, the impact is most seen for the SDSC trace.

5.3 Half vs. Full Self-Tuning

For the comparison of half and full self-tuning one can assume that applying full self-tuning is the best option. With this the self-tuning dynP scheduler chooses the best scheduling policy every time the schedule changes, i. e. when a new job is placed in the schedule and a running job terminates earlier than estimated and a re-scheduling is required. The self-tuning dynP scheduler plans schedules for each available scheduling policy and for all waiting jobs. This results in an increased computational time of the scheduler. As this can be unappropriate in certain scenarios, half self-tuning is an option, as roughly only half as many self-tuning calls are done. Some performance loss might occur with half self-tuning, as with new job submissions the scheduling policy might be changed. In the MuPSiE simulation environment a single self-tuning call for finding a new policy is completed within 6 ms for an average of 22.5 waiting jobs (simulated configuration: advanced decider, full self-tuning, ARTwW as self-tuning metric, CTC trace) and applying half self-tuning might not be necessary.

In Table 4 the slowdown results for the simple, advanced, SJF- and FCFS-preferred decider are presented. One can see that the assumption from above is not always true. In particular for the SDSC trace, half self-tuning is better than full self-tuning. Also with the simple decider full self-tuning is not beneficial. Looking at the performance of the advanced and SJF-preferred decider shows that for the CTC and LANL trace the self-tuning dynP scheduler is always better than the best basic policy. Furthermore, it is interesting to observe that half and full self-tuning have no major impact on the performance of these two deciders. The generated SLDwA values are closer together. Similar to the comparison of the different self-tuning metrics, the SDSC trace is more vulnerable for the switching behavior of the self-tuning dynP scheduler. In particular the simple and FCFS-preferred deciders generate very bad results with full self-tuning applied. In this case doing more self-tuning calls increases the SLDwA by a factor of two. Again this can be a result of overestimating the job run times. By doing self-tuning when jobs terminate and by potentially switching to a disadvantageous scheduling policy, different jobs are immediately started.

One can also see that the advanced decider obviously outperforms the simple decider due to its design. This is independent of whether full or half self-tuning is performed. The performance benefit of the advanced decider is different for the four traces; quite large for the KTH and SDSC trace and smaller for the LANL trace. For the SDSC trace and full self-tuning the difference between the two deciders is almost 70%.

As for the CTC and KTH trace SJF is the best ba-

	self-tuning metrics						
	ART	ARTwA	ARTwW	SLD	SLDwA	SLDwW	Makespan
SLDwA							
CTC	2.0073	2.2866	**1.8781**	1.9585	**1.8781**	1.9021	2.4582
KTH	3.1459	5.6542	**2.5754**	2.6939	**2.5754**	2.5594	5.3823
LANL	1.7008	1.8328	**1.6177**	1.6626	**1.6177**	1.6179	2.0357
SDSC	14.6495	20.5247	**10.1598**	12.6742	**10.1598**	11.8321	24.8958
Utilization							
CTC	for all self-tuning metrics: 65.701%						
KTH	for all self-tuning metrics: 68.716%						
LANL	for all self-tuning metrics: 55.607%						
SDSC	81.787%	81.812%	81.633%	81.762%	81.633%	81.309%	**82.473%**

Table 3. Overall SLDwA and utilization values for different self-tuning metrics. Advanced decider and full self-tuning applied. Values for ARTwW and SLDwA are equal because of Equation 2.

	best basic policy	self-tuning	decider mechanisms			
			simple	advanced	SJF-preferred	FCFS-preferred
CTC	1.9277	half	2.3036	**1.9085**	**1.8567**	2.3270
	(SJF)	full	2.2812	**1.8781**	**1.8873**	2.2804
KTH	2.5488	half	4.7256	2.5812	2.5734	4.9281
	(SJF)	full	5.7433	2.5754	2.5578	5.7492
LANL	1.6801	half	1.7534	**1.6027**	**1.6330**	1.7605
	(FCFS)	full	1.7610	**1.6177**	**1.6143**	1.7680
SDSC	6.8260	half	13.3353	10.0953	10.2896	16.1766
	(FCFS)	full	32.6934	10.1598	10.5198	32.6965

Table 4. Overall SLDwA comparison of full and half self-tuning with different decider mechanisms.

sic policy and FCFS for the LANL and SDSC trace, a SJF- and FCFS-preferred policy makes sense. The results indeed show that the SJF-preferred decider can improve the performance of the advanced decider for the CTC and KTH trace, but the same does not apply for the FCFS-preferred decider and the LANL and SDSC traces. In fact the FCFS-preferred decider is worse (almost three times for the SDSC trace) than the advanced and SJF-preferred decider. This is surprising, as FCFS proves to be a good basic policy. The poor performance can be based on the fact that some jobs, which are not started by FCFS, alter the schedule in such a way that many subsequent jobs have to wait long and therefore the SLDwA drops. A possible example for this scenario could look like the following: some jobs with a large area (requesting many resources and/or with a long estimated run time) may induce a policy change in order to favor these jobs. However, the estimate of the run time may have been wrong, so that the end after some time. In this case delaying these jobs would be beneficial, as due to their short actual run time their influence on the overall SLDwA performance may only be small.

5.3.1 Detailed Analysis of the Switching Behavior

With full self-tuning the difference between the simple and advanced decider is best seen for the SDSC trace, hence a detailed case analysis is done in the following. Table 5 shows the amount each case is reached during the decision process. The numbers show a significant difference in case 6b: the performance of FCFS is equal to SJF, LJF is worse than both, and the old policy is SJF. In 80,419 (75.11%) of 107,066 total self-tuning decisions this situation occurs and the advanced decider stays with SJF. In contrast, the simple decider reaches this case in only 42.56% of all self-tuning decisions and switches to FCFS in this situation.

In case 1 all three policies have the same performance. The correct decision is to stay with the old policy like the advanced decider does, but the simple decider arbitrarily favors FCFS. The other two cases 8c and 10c are not reached by the simple or advanced decider, hence they can not induce the difference in performance. With the large differences in case 6b the number of appearances of the other cases is also influenced. This is best seen for case 4b. However, the other cases have no influence on the different performance of the simple and advanced decider, as both deciders

case		combinations	simple decider	counted	advanced decider	counted
1		FCFS = SJF = LJF	**FCFS**	**11,135**	**old policy**	**15,861**
2		SJF < FCFS, SJF < LJF	SJF	47,285	SJF	962
3		FCFS < SJF, FCFS < LJF	FCFS	60	FCFS	119
4		LJF < FCFS, LJF < SJF				
	a	FCFS < SJF	LJF	19	LJF	8
	b	FCFS = SJF	LJF	2,007	LJF	7,178
	c	FCFS > SJF	LJF	26	LJF	10
5		FCFS = SJF, LJF < FCFS ($\Leftrightarrow$ LJF < SJF)	LJF	0	LJF	0
6		FCFS = SJF, FCFS < LJF ($\Leftrightarrow$ SJF < LJF)				
	a	old policy = FCFS	FCFS	362	FCFS	603
	b	old policy = SJF	**FCFS**	**46,617**	**SJF**	**80,419**
	c	old policy = LJF	FCFS	254	FCFS	1085
7		FCFS = LJF, SJF < FCFS ($\Leftrightarrow$ SJF < LJF)	SJF	0	SJF	0
8		FCFS = LJF, FCFS < SJF ($\Leftrightarrow$ LJF < SJF)				
	a	old policy = FCFS	FCFS	1,751	FCFS	820
	b	old policy = SJF	FCFS	0	FCFS	0
	c	old policy = LJF	**FCFS**	**0**	**LJF**	**0**
9		SJF = LJF, FCFS < SJF ($\Leftrightarrow$ FCFS < LJF)	FCFS	3	FCFS	0
10		SJF = LJF, SJF < FCFS ($\Leftrightarrow$ LJF < FCFS)				
	a	old policy = FCFS	SJF	2	SJF	1
	b	old policy = SJF	SJF	0	SJF	0
	c	old policy = LJF	**SJF**	**0**	**LJF**	**0**
		totally counted		109,521		107,066

Table 5. Case analysis for the SDSC trace and the simple vs. advanced decider with full self-tuning applied and SLDwA as self-tuning metric.

choose the same policy (LJF) as their new policy.

Focusing on the policy usage, Table 6 shows the differences, i. e. how many times the decider switched to each policy and how many jobs were started with each policy. If the advanced decider is applied almost 80% of all jobs are started by SJF and only a minority of 6% by FCFS. About 15% of the jobs are started with LJF. Focusing on the number of switches to each of the policies shows that the advanced decider stays with the current policy and does not switch it in most cases (97%). Only in about 1,000 cases the advanced decider switches to one of the policies. This means that once the decider switched to a policy, many jobs are started with this policy. This applies in particular to SJF.

If on the other hand the simple decider is applied, its switching behavior is much more spontaneous. In only 10% of all cases the simple decider does not switch its policy. Most of the time it switches back and forth between FCFS and SJF. This results in an almost equal usage of the two policies over a period of time (43%) and the difference in the number of jobs started with FCFS and SJF is also considerably smaller than with the advanced decider. In only 13% of all self-tuning decisions the simple decider stays with its current policy. Discarding its previous decision leads to a scenario where preceding jobs are started by alternating policies. Compared to the advanced decider about ten times more jobs (60%) are started with FCFS by the simple decider, whereas about only half as many jobs (40%) are started by SJF. Only a minority of all jobs are started with LJF.

The number of self-tuning calls with the simple decider (109,521) is larger than with the advanced decider (107,066). This results from the fact that more than one job ends at the same time. Why? As full self-tuning is applied and the same job trace is used, the amount of self-tuning calls at job submission does not change for one of the deciders. However, if more than one job ends at the same time, a reschedule takes place only once and therefore self-tuning is also called only once. Hence, the advanced decider performs better than the simple decider and at the same time induces less self-tuning calls.

From this fact another question arises: If only half self-tuning is applied, i. e. self-tuning is not done when jobs end, the number of self-tuning calls should almost be the same for both deciders? And yes, if half self-tuning is applied the simple decider is called 56,738 times whereas the advanced decider is called 56,208 times. Both amounts are a slightly less than the number of totally scheduled jobs (67,620). This is based on the fact that if enough resources are free to start all jobs immediately, these jobs do not have to wait. Self-tuning and scheduling policies in general make no sense in this case, as the starting order of jobs does not matter, as long as they are started immediately.

		simple decider		advanced decider	
switches to each policy	FCFS	47,499	(43.37%)	1,086	(1.01%)
	SJF	47,287	(43.17%)	963	(0.90%)
	LJF	395	(0.36%)	1,085	(1.01%)
	no policy switch	14,340	(13.09%)	103,932	(97.07%)
job started with each policy	FCFS	39,936	(59.06%)	4,120	(6.09%)
	SJF	26,737	(39.54%)	53,634	(79.32%)
	LJF	947	(1.40%)	9,866	(14.59%)

Table 6. Comparison of the decision behavior and the usage of policies for the SDSC trace with full self-tuning applied.

6 Conclusions and Future Work

In this paper we presented two options for the decision process of the self-tuning dynP scheduler. The idea of dynamically switching the scheduling policy (dynP) is based on the fact that usually no single policy generates good schedules for every possible job characteristic. In order to achieve the best possible performance, it becomes necessary to switch the active scheduling policies according to the currently waiting jobs. The scheduler switches the scheduling policies without the need of a permanent intervention of the system administrator. With the planning-based scheduling approach, the self-tuning dynP scheduler generates full schedules for each available basic policy, measures the generated schedules with a performance metric, and finally switches to the best policy. A decider mechanism is in charge of choosing the best policy according to the applied performance metric. In previous papers we presented different decider mechanisms.

In this paper we evaluated two general enhancements, which can be applied to all decider mechanisms. We compared different owner and user centric performance metrics for the self-tuning process and studied their influence. By using different self-tuning metrics the objective of the self-tuning dynP scheduler can be altered. Additionally, we studied the behavior of calling the self-tuning process at different times (half and full self-tuning) of the scheduling process. The evaluation is done by using discrete event simulation with job traces from real supercomputer installations as input.

Studying the different self-tuning metrics one assumes that most likely the best performance is achieved by using the same metric during the self-tuning process and after the simulation is finished to measure all jobs. This is true, the user centric average slowdown weighted by area (SLDwA) is the best self-tuning metric for three traces (CTC, SDSC, and LANL). For the KTH trace the average slowdown weighted by width (SLDwW) slightly improves the performance slightly by 0.8%. If the objective of the self-tuning dynP scheduler is to optimize the owner centric overall utilization of the system, the makespan generates the best results, although only for the SDSC trace. The characteristics of the remaining three traces generate equal overall utilizations for all applied self-tuning metrics.

We also compared half and full self-tuning, i. e. calling the self-tuning process only when new jobs are submitted or additionally when running jobs terminate. Although much less self-tuning calls are done with half self-tuning, the performance different to full self-tuning is small with the advanced and SJF-preferred decider. Similar to the comparison of different self-tuning metrics, the SDSC trace is more vulnerable for the switching behavior of the self-tuning dynP scheduler. In particular the simple and FCFS-preferred decider generate very bad results, which are almost three times as bad as for the other deciders. Therefore, if less self-tuning calls are intended by the system administrators, e. g. to reduce the switching behavior of the self-tuning dynP scheduler, the generated performance is only sightly behind and half self-tuning is a good compromise.

We showed that in general the presented self-tuning scheduler with dynamic policy switching can be beneficial to commonly used static scheduling approaches. Therefore, we think that self-tuning schedulers, which are able to adapt their scheduling behavior according to the job characteristics of the currently waiting jobs, should be implemented in modern cluster resource management systems. From a practical perspective the self-tuning dynP scheduler might cause problems for the users, as the scheduling behavior of the system might become unpredictable. For practical matters a policy switching might only be done at well established times, e. g. only once an hour.

In the future it will be interesting to study, whether a combination of different self-tuning performance metrics is beneficial. For example, the owner centric makespan metric is considered with 20%, and the user centric average response time weighted by job width is considered with 80%.

References

[1] D. G. Feitelson. A Survey of Scheduling in Multiprogrammed Parallel Systems. Research report rc 19790 (87657), IBM T.J. Watson Research Center, Yorktown Heights, NY, 1995.

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

[3] D. G. Feitelson and B. Nitzberg. Job Characteristics of a Production Parallel Scientific Workload on the NASA Ames iPSC/860. In D. G. Feitelson and L. Rudolph, editor, *Proc. of 1st Workshop on Job Scheduling Strategies for Parallel Processing*, volume 949 of *Lecture Notes in Computer Science*, pages 337–360. Springer, 1995.

[4] 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 of *Lecture Notes in Computer Science*, pages 65–80. Springer, 1996.

[5] M. Hovestadt, O. Kao, A. Keller, and A. Streit. Scheduling in HPC Resource Management Systems: Queuing vs. Planning. In D. G. Feitelson and L. Rudolph, editor, *Proc. of the 9th Workshop on Job Scheduling Strategies for Parallel Processing*, volume 2862 of *Lecture Notes in Computer Science*, pages 1–20. Springer, 2003.

[6] 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.

[7] D. A. Lifka. The ANL/IBM SP Scheduling System. In D. G. Feitelson and L. Rudolph, editor, *Proc. of 1st Workshop on Job Scheduling Strategies for Parallel Processing*, volume 949 of *Lecture Notes in Computer Science*, pages 295–303. Springer, 1995.

[8] M. Maheswaran, S. Ali, H. J. Siegel, D. Hensgen, and R. F. Freund. Dynamic Mapping of a Class of Independent Tasks onto Heterogeneous Computing Systems. *Journal of Parallel and Distributed Computing*, 59(2):107–131, 1999.

[9] A. 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. IEEE Computer Society Press, June 2001.

[10] S. Muthukrishnan, R. Rajaraman, A. Shaheen, and J. E. Gehrke. Online Scheduling to Minimize Average Stretch. In *Proceedings of the 40th Annual IEEE Symposium on Foundations of Computer Science*, pages 433–442, 1999.

[11] The *pling* Itanium2 Cluster at the Paderborn Center for Parallel Computing (PC^2). `http://www.upb.de/pc2/services/systems/pling/index.html`, April 2004.

[12] The *PSC* Pentium3 Cluster at the Paderborn Center for Parallel Computing (PC^2). `http://www.upb.de/pc2/services/systems/psc/index.html`, April 2004.

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

[14] 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 of *Lecture Notes in Computer Science*, pages 41–47. Springer, 1996.

[15] A. Streit. A Self-Tuning Job Scheduler Family with Dynamic Policy Switching. In D. G. Feitelson and L. Rudolph, editor, *Proc. of the 8th Workshop on Job Scheduling Strategies for Parallel Processing*, volume 2537 of *Lecture Notes in Computer Science*, pages 1–23. Springer, 2002.

[16] A. Streit. The Self-Tuning `dynP` Job-Scheduler. In *Proc. of the 11th International Heterogeneous Computing Workshop (HCW) at IPDPS 2002*, pages 87 (book of abstracts, paper only on CD). IEEE Computer Society Press, 2002.

[17] A. Streit. Evaluation of an Unfair Decider Mechanism for the Self-Tuning dynP Job Scheduler. In *Proc. of the 13th International Heterogeneous Computing Workshop (HCW) at IPDPS*, pages 108 (book of abstracts, paper only on CD). IEEE Computer Society Press, 2004.

[18] Parallel Workloads Archive. `http://www.cs.huji.ac.il/labs/parallel/workload/`, April 2004.