Self-Tuning Job Scheduling Strategies for the Resource Management of HPC Systems and Computational Grids

Dissertation

von

Achim Streit

Schriftliche Arbeit zur Erlangung des Grades
eines Doktors der Naturwissenschaften

Fakultät für Elektrotechnik, Informatik und Mathematik
der Universität Paderborn

Paderborn, Oktober 2003

Acknowledgment

Research is not a one-man show, many individuals have contributed to the work presented in this thesis. At first, I would like to thank my supervisor Prof. Dr. Burkhard Monien for his continuous support throughout the past four years. Secondly, I would like to thank Prof. Dr. Uwe Schwiegelshohn from the Computer Engineering Institute, Dortmund, for giving me much advice, answering my questions, and also asking the right questions at the right time. Additionally, I have to thank Prof. Dr. Odej Kao for allowing me to write this thesis down in the recent months and for taking off as much workload as possible from me.

A big thank-you goes to my former colleague Dr. Jörn Gehring, whom I owe a lot. Jörn put me on the right track and showed me how research is done successfully.

I would also like to thank my colleagues Carsten Ernemann, Volker Hamscher, and Dr. Ramin Yahyapour from the Computer Engineering Institute of Prof. Schwiegelshohn. The part on grid scheduling in this thesis would not be possible without them. During long telephone discussions, fruitful work was done and nice papers were written and published.

Finally, I would like to thank all members of the PC^2 and the research group of Prof. Monien for providing a nice place to work. In particular, I have to thank Sven Grothklags, Jan Hungershöfer, Axel Keller, and Jens-Michael Wierum, who had to answer many questions, with whom I had fruitful discussions, and who had to get along with me - I know that's not easy.

Abstract

In this thesis we develop and study self-tuning job schedulers for resource management systems. Such schedulers search for the best solution among the available scheduling alternatives in order to improve the performance of static schedulers. In two domains of real world job scheduling this concept is implemented. First of all, we study the scheduling in resource management software for high performance computing (HPC) systems. Typically, a single scheduling policy like first come first serve is used, although the characteristics of the submitted jobs permanently change. Using a single scheduling policy might induce a performance loss, as other policies might be more suitable for specific job characteristics. We develop a self-tuning scheduler, which automatically checks all implemented policies and switches to the best one. This improves the performance, in terms of increased utilization and decreased waiting time.

Secondly, we develop and study an adaptive scheduler for computational grid environments. In such grids, several geographically distributed HPC machines are joined in order to increase the amount of computational power. Grid jobs might be scheduled across multiple machines, so that the communication among the job parts involves slow wide area networks. This often induces an additional communication overhead, which has to be considered by the grid scheduler. Our adaptive grid scheduler considers the slower communication over wide area networks by extending the execution time of such multi-site jobs. The developed adaptive multi-site grid scheduler automatically checks, which of the two options is more beneficial: waiting for enough resources at a single site, or using multiple sites and the slower wide area network immediately.

In both cases we use discrete event simulations for evaluating the performance of the developed schedulers. The results for the self-tuning scheduler show, that an increased utilization of the system and a decreased waiting time for the jobs are possible. We think, that such self-tuning schedulers should be used in modern resource management systems for HPC machines. The evaluation of the grid scheduler shows, that in general a combination of many small machines and multi-site scheduling can not perform as well as a single large machine with the same amount of resource. However, the adaptive multi-site scheduler decreases the performance difference significantly. We think that the participation in computational grid environments is beneficial, as larger problems requiring more computational power can be solved.

Contents

1 Introduction

Modern high performance computing (HPC) machines are becoming increasingly faster in compute and interconnect speeds, memory bandwidth, and local file I/O [62]. An efficient usage of the machines is important for users and owners, as such systems are rare and high in cost. Resource management systems for modern HPC machines consist of many components which are all vital in keeping the systems fully operational. The user interface allows the users to submit and cancel jobs. Information about the system status and submitted or executing jobs can also be obtained. The scheduler assigns resources to jobs according to the active scheduling policy. The information service stores data concerning the number of resources that are generally available and of which type they are. Furthermore, data about the current state and the current load is provided. Policies are defined in the security service, which in turn regulate who is authorized to execute jobs and whether the resource usage is restricted or not. The monitoring and watch dog component regularly checks if all components of the resource management system are fully operational.

With regards to performance aspects, all components of the resource management system should perform their assigned tasks efficient and fast, so that no additional overhead is induced. Different objectives exist and acceptability, usability, reliability, security, robustness are only some to mention. If performance metrics like throughput, utilization, wait and 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. Typically a scheduling policy that optimizes the utilization prefers those jobs, which in turn need 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 [16]. The task of the scheduler is to find a good compromise between optimizing these two contrary metrics.

Due to being rare, HPC systems usually have a large user community with different resource requirements and general job characteristics. For example, some users primarily submit parallel and long running jobs, while other users submit hundreds of short and sequential jobs [93, 41]. 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. The arrival distances might be large, as e. g. the previous run has to be evaluated before new parameters are generated. In addition, the submission behavior also varies over a longer time span, as e. g. conference or project deadlines and holidays are reflected.

All this results in a non-uniform workload and job characteristics that permanently change [24]. Hence, using only a single, pre-defined scheduling strategy can result in a loss of performance, i. e. more resources remain idle and jobs have to wait longer than necessary.

Overview

Modern resource management systems for single HPC machines usually have several scheduling policies implemented [69, 56, 5]. Most commonly used is first come first serve (FCFS) with backfilling [55, 82, 64], as on average a good utilization of the system and good response times of the jobs are 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, while shortest job first (SJF) is used with mostly short jobs [16]. Many resource management systems allow to change the active scheduling policy, but this has to be done manually by the system administrators [61, 5].

We group modern resource management systems into two classes, queuing and planning systems. The major difference between the two classes is the scheduled time frame. Queuing systems schedule only the present, while planning systems schedule the present and future. As soon as a new job is submitted, the scheduler of a planning system places the job in the schedule and assigns a potential start time. With this scheduling approach, a so called "full schedule" is generated, which contains preliminary start and end times of all currently waiting jobs.

We present the self-tuning dynP scheduler, which dynamically switches the active policy of the resource management system according to the characteristics of the currently waiting jobs. As we assume a planning system, the self-tuning dynP scheduler computes full schedules for every available policy, rates the schedules with means of a performance metrics, and switches to the best policy. We develop and compare decider mechanisms, which are fair and unfair in terms of policy usage. Different enhancements for the self-tuning dynP scheduler are also presented and studied. Similar work on schedulers with dynamic policy switching is presented in [73] and in [23] a self-tuning system is presented which uses genetic algorithms to generate new parameter settings for a scheduling system. While computing full schedules for every policy, the self-tuning dynP scheduler does a quasi off-line scheduling. Following [92], we model the scheduling problem as an integer problem. This is solved with the well-known CPLEX library [45]. With this approach we check how much performance is lost when using common scheduling strategies. Optimal schedules are not computed by CPLEX, as we have to apply time-scaling in order to reduce the number of variables of the integer problem and to make it computable.

Similar to single HPC machines, resource management functionalities are also required in grid computing environments, where several HPC machines are joined [30]. The grid middleware is responsible for providing a transparent access to these resources. All previously mentioned aspects and conclusions about resource management systems for a single HPC machine are also applicable to the grid middleware. An additional level of sophistication is added to the scheduling process, as typically each site works autonomously with its own restrictions, policies, and resource management system [65]. A grid scheduler has to find available and appropriate machines for a job and then decides on which of the machines the job is started. Jobs might also be started on multiple machines, which are typically geographically distributed. If the different parts of such a multi-site job communicate with each other, wide area network (WAN) links are involved. Compared to an interconnect inside of a single HPC machine, the bandwidth of WAN connections is typically low and the latency high. In order to reflect the communication overhead of the involved WAN connections, we increase the execution time of multi-site grid jobs. If not enough resource are available to place a job on a single machine immediately, our multi-site scheduler decides, whether the

job waits until enough resources are available at a single site, or the job is started on multiple sites immediately and the jobs execution time is increased. In both cases, the response time of the job is increased. The adaptive multi-site grid scheduler decides which of the two options is more beneficial (wait or distribute). The alternative with the shortest response time is chosen.

Like the previously mentioned self-tuning scheduler for single HPC machines, this adaptive multi-site grid scheduler is autonomous, too. Both schedulers check various scheduling alternatives on their own and automatically choose that alternative, which promises to generate the best performance.

Document Structure

This thesis is subdivided into eight chapters. Related Work is found in Chapter 2 in order to give the reader the opportunity to range this work in the broad spectrum of research on job scheduling. In Chapter 3 general aspects are described, common terminologies are introduced, and definitions are made, which are used in the remainder of this thesis.

In Chapter 4 the basic concept of dynamically switching the scheduling policy is described. We begin with a history of development and the description of the initial version that uses two bounds for deciding when the scheduling policy is switched. Following that, the self-tuning version of the dynP scheduler with all its decider mechanisms and general options is described. To conclude this chapter we present an approach for computing optimal schedules. The scheduling problem is modelled as an integer problem, which is then solved by the CPLEX library. Chapter 5 contains the evaluation with original traces and synthetic job sets with increased workloads. The evaluation of the self-tuning dynP scheduler contains studies about different decider mechanisms, their switching behavior, and several options for the decision process of the self-tuning dynP scheduler.

Chapter 6 focuses on job scheduling in grid environments. We describe how we model the grid environment and we define three scenarios for the scheduling. Of special interest is the multi-site scenario, which allows the execution of a grid job across multiple sites. Therefore, the evaluation in Chapter 7 concentrates on the multi-site scenario. To begin with, we study the effects of different machine configurations on the grid scheduling. Secondly, a comparison of job sharing and multi-site is made. The chapter ends with an evaluation of constraints for multi-site scheduling. The adaptive version of the multi-site grid scheduler is similar to the self-tuning dynP scheduler, as it also searches for the best solution to start jobs.

A brief conclusion closes this thesis. At the end an outlook and some ideas, on the effects of scheduling with advanced reservations, are given.

Publications

Many parts of this work have been published on workshops and conferences. The classification of resource management systems in Section 3.1 has been presented on the *9th Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP 2003)* [42]. The work in Chapter 4 and 5 on the self-tuning dynP scheduler has been published in the proceedings of the *8th International Conference on High Performance Computing (HiPC 2001)* [85], in the proceedings of the *11th International Heterogeneous Computing Workshop (HCW) at IPDPS 2002* [87], and in the proceedings of the *8th Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP 2002)* [86].

The work on scheduling in grid environments from Chapters 6 and 7 is a joined work with Carsten Ernemann, Volker Hamscher, and Ramin Yahyapour. It has been published in the proceedings of the *1st IEEE/ACM International Workshop on Grid Computing (Grid 2000)* [40], in the proceedings of the *International Conference on Architecture of Computing Systems (ARCS 2002)* [15], in the proceedings of the *2nd IEEE International Symposium on Cluster Computing and the Grid (CC-GRID 2002)* [13], and in the proceedings of the *3rd IEEE/ACM International Workshop on Grid Computing (Grid 2002) at Supercomputing 2002* [14].

Not mentioned in this thesis are the early experiences of the EGrid Testbed Working Group of the former European Grid Forum, which was joined with the Global Grid Forum [35] in 2000. These experiences are published in the proceedings of the *1st IEEE/ACM International Symposium on Cluster Computing and the Grid (CC-GRID 2001)* [12]. This paper is different, as all 31 people involved in the work of the EGrid Testbed Working Group are listed as authors of the paper. So writing the paper was more an organizational issue.

2 Related Work

Scheduling in a uniprocessor system is about making the decision, which application or thread runs on the single processor? In a multiprocessor system a second dimension is added, as more than one processor exists. The question is not only about when a job should run but also where, i. e. on which processors? Thus, jobs are competing for resources in time and space. Such a multi processor system scheduler assigns sets of resources to parallel applications. Scheduling is also required on a lower level by assigning threads of parallel applications to processing elements. This is either done by the resource management system, by the language run time system, or within the application itself. This work focuses on scheduling in the resource management system at the higher level, i. e. which resources a job gets assigned to and when?

In the following sections several papers and other publications which deal with various subjects of parallel job scheduling for HPC computers are presented. Section 2.1 covers general aspects about parallel job scheduling and it also contains definitions of common terminologies. Some analytical results are presented in Section 2.2. In Section 2.3 work on characterizing supercomputer workloads is presented. Related work for self-tuning and dynamic policy switching is presented in Section 2.4. Finally, Section 2.5 is about the related work for the area of scheduling in meta and grid computing environments.

2.1 Parallel Job Scheduling

In 1995 Feitelson and Rudolph presented an introductory paper [25] on the 1st Workshop on Job Scheduling Strategies for Parallel Processing [48]. The paper gives an overview on issues and approaches for parallel job scheduling. The authors state, that parallel computers are more difficult to use than single processor machines mainly because a different programming model has to be used. Additionally, more users (respectively their jobs) than resources exist on such machines, so jobs compete with each other. Nonetheless, users are willing to use parallel machines, because they come with major benefits:

- Applications that run on a parallel system potentially finish in a shorter period of time.
- The problem that needs to be solved may be more complex, but it is still computed in the same time.
- The problem may be solved more precisely.

Therefore, many people are using parallel machines and a wide range of applications are executed on these systems. Obviously the resource requirements differ from one user (or application) to the other. Some users need large batch jobs, i. e. many processors for a long run time. Others work interactively most of the time and they probably only need a single processor for a short amount of time (e. g. for developing a new application). In this case, job

scheduling and resource management becomes an even more critical issue, as nobody wants to wait half a day for a job that runs for just 10 minutes.

The authors state, that job scheduling means many different things to many different people. For example, for people from parallel programming languages, parallel computer architectures and parallel operating systems an ideal scheduling strategy does not exist. Furthermore, the problem is getting harder with the large set of diverse goals: should the scheduler try to generate short response times for individual jobs or a high overall system utilization? It is also hard to compare schedulers in commercial products that process real workloads with schedulers that are developed in academic research and are usually tested with synthetic workloads.

In [25] four classes (or levels) of scheduling are defined:

1. **Static scheduling** is often used for understanding the theoretical limitations of scheduling. It is made under the assumption that a close to complete knowledge of the computation is available. An application is modelled as a DAG (directed acyclic graph) where nodes represent tasks and direct edges describe dependencies between two tasks. The task at the origin of a direct edge has to be completed before the task at the end is executed. Furthermore, nodes and direct edges have weights which reflect the amount of computation of a task and the amount of communication or transferred data between two tasks. Many parallel programming styles like functional programming, dataflow, or other side-effect free programming models are similar to the described DAG model. Hence, in the past many heuristics have been developed for approximating good schedules. Unfortunately, many heuristics are not applicable for programming models that use other representations.

2. **Scheduling in the run time system** has the benefit that the user does not see the details of how the program is mapped on the machine. The run time system accomplishes this. One common approach is to use a thread library where the run time environment implements the functionality for creating, synchronizing, and killing threads. The application simply calls these functions. Another approach is to use parallelizing compilers, that unroll loops and parallelize them. Common to both approaches is, that the programmer does not have to deal with details of how this is done at the machine level.

3. **Scheduling in the resource management system** is induced by balancing the individual needs of every user when multiple programs co-exist in the same system. The applied mechanisms and considerations are similar to scheduling at the run time or application level, so that it is not quite clear who is in charge of the problem: the run time or resource management system. Hence, the spectrum of opinions range from: all scheduling has to be done on the application level, which leaves out the resource management scheduler, to where the resource management system should do all the work. Nevertheless, as parallel systems become more and more popular resource management at this system level is needed.

4. **Administrative scheduling** makes scheduling decisions on the administrative and political level. For example, the head of the computing center allows a certain project to use the whole machine for the next month or specific job types (short massively parallel jobs or long jobs of less parallelism) are preferred by assigning higher priorities. The authors of [25] state, that unfortunately no research has been done on this level of scheduling so far, although it effects many users significantly.

In the following scheduling on the level of the resource management system is addressed. A resource management scheduler has to decide when a parallel job is started and which resources are assigned to the job. Therefore, the main issue is how to share the available resources among the competing jobs, so that each job gets the requested level of service. Two sharing schemes exist:

In *time-sharing* (or time-slicing) several jobs share the same resource and are executed quasi-simultaneously. Resources are not exclusively assigned to a specific job. The resource usage of jobs is reduced to short time slices. Each job gets the resource assigned in a round-robin fashion. Jobs need more than a single time slice to be completed, as a time slice is typically very short, e. g. only some clock ticks of the processor. As a result new jobs are started immediately without waiting time, but their execution takes longer than with a dedicated resource. Shared memory machines are often operated in time-sharing mode.

On the other hand in *space-sharing* (or space-slicing) resources are exclusively assigned to a job until it is completed. As the amount of resources are limited, jobs may have to wait for enough free resources until they can get started. Of course, a combination of space- and time-sharing is also thinkable. Space-sharing is not useful, e. g. if the network is not partitionable or an application needs the full parallelism of the machine. Similarly, if it is not possible to switch protection domains of an application or if paging in and out large amounts of main memory takes too much time, time-sharing is not suitable.

Both sharing schemes are orthogonal to each other and a combination of these might be useful. Several examples are found in [16] and the four most popular approaches are:

1. **Global queue:** Threads of a parallel job that are ready to run are placed in a global queue. An idle processor picks the first thread of the global queue, executes it for a certain time quantum and then puts it back into the queue. This approach is commonly used in bus-based shared memory processor systems. The main advantage is automatic load sharing, but this comes with several drawbacks. Threads are typically executed on different processors, so that caches are wiped out with each re-scheduling. This is prevented when the scheduler tries to reschedule threads on the same processor as before (affinity scheduling). Furthermore, the global queue approach induces problems if the threads of an application interact and synchronize with each other, as the threads are most likely scheduled in an uncoordinated manner. An interesting fact about the global queue approach is, that the service a job receives is proportional to the number of threads it spawns. Large jobs (many threads) that require more computational time more often get the necessary resources.

2. **Variable partitioning:** All available processors are partitioned in disjointed sets. Each job is executed in a distinct partition. If the partitioning is fixed, the size of each partition is set in advance by the administrative staff. Re-partitioning requires a re-boot of the machine. It is possible to split a large partition in several smaller ones to allow small jobs to be executed in parallel. If the current system load is taken into account and the size of each partition is automatically set by the system, adaptive partitioning is done. Variable partitioning is common on distributed memory machines, like the IBM SP2 or clusters. It places the needs of individual jobs over that of the system and has many more advantages: e.g. no cache interference, no operating system overheads, and depending on the network topology there might also be no network interference. For example, in a hyper-cube each sub-cube uses a disjointed set of links. However, in a multistage network some links may be shared by different partitions. Of course, variable

partitioning has some disadvantages as fragmentation may occur, if the number of free processors is insufficient in satisfying the needs of waiting jobs. Hence, jobs may wait for a long time until the requested resources are available.

3. **Dynamic partitioning with two-level scheduling:** Changes in the partitioning during the execution is provided by dynamic partitioning. However, applications have to support this and the used programming model has to express changes in requirements (more or less resources) but should also handle system induced changes. One approach in establishing this, is by using a work-pile model which de-couples the computational task from the workers that perform the computation. It is easily possible to adjust the number of workers. Two levels of scheduling are done, as the operating system deals with the allocation of processors to a job and its application, while the application assigns chunks of work to the processors and workers respectively. Dynamic partitioning has several advantages: no loss of resources due to fragmentation, no overhead from context switching (except from the re-distribution of processors when the load changes), no waste of CPU cycles on busy waiting for synchronization, and the degree of parallelism for each application is automatically decreased under heavy load conditions.
   Interaction between the application scheduler and the resource management scheduler is needed. For example, if a vital processor is taken from the application, the application may stay in a locked state. Consequently, the operating system and programming model have to be designed together, which limits the portability of applications.

4. **Gang scheduling:** The benefits of space- and time-sharing are combined, as context switching is done coordinately across all processors of a job. All threads of a job are executed synchronously, so that fine grained interaction is possible. Additionally, threads are usually bound to a specific processor which allows the use of local memory and to benefit from a sustained cache state. Gang scheduling works with every programming model, but also comes with some drawbacks: the overall system performance is not always optimal, although it favors individual jobs. Cache interference, overhead from context switching, processor fragmentation and draining of network links might occur. The hardware requirements during context switching are high, considering the main memory.

Variable and dynamic partitioning are based on the space-sharing approach. Time-sharing concepts are used for the other two approaches.

In a paper on theory and practice in parallel job scheduling [28] Feitelson, Rudolph, Schwiegelshohn, Sevcik, and Wong describe the way in which massively parallel processors (MPP) systems with distributed memory are typically operated:

- The system is divided in partitions that are assigned for the processing of parallel jobs, doing interactive work in time-slicing mode, or handling service tasks like file I/O. The partitioning of the system can be changed manually in order to ,e.g., assign more processors to the batch partition during the night.

- Several job queues that reflect various job characteristics (e. g. a queue for heavily parallel but short running jobs and another for sequential long running jobs) and priorities (e. g. high priority, best effort) are defined. Within a queue, jobs are usually sorted in first come first serve order.

- An assignment between a partition and one or more job queues is done, so that the processors in a partition serve as a pool for the assigned queues. If processors are free, non empty queues are searched according to their priority for jobs to start. Once a job is started with the requested number of processors, it runs until completion.

The authors state, that many different models are used to describe the scheduler and the system. Usually, the constraints of these models directly affect the performance of the scheduler. Some of these constraints are inspired by real systems and by the way these are managed. The authors characterize the models according to the five criteria:

1. **Partitioning of the system:** Every job is executed in a partition of the system. Partitions are:

   - *fixed*, if the size is defined by the administrative staff and can only be modified when the machine is rebooted.
   - *variable*, if it is sized according to the user request at the submission of the job.
   - *adaptive*, if the scheduler determines the size at job start and takes the user request into account.
   - *dynamic*, if the size of the partition changes during the run time of the job, depending on the system load and different requirements.

   Table 2.1 is taken from [16] and shows a comprehensive overview on the four types of partitioning. Most common is variable partitioning, as the parallelism is hardcoded (which is simpler than expressing the parallelism e. g. a parameter and speedup function) and the requests are matched. On the other hand, adaptive and especially dynamic partitioning have to be supported by the submitted jobs. This requires more effort by the programmer and is therefore less common.

2. **Job flexibility:** According to the partitioning of the system, jobs are:

   - *rigid*, if the number of assigned resources is not specified by the scheduler. The scheduler has to assign the number of resources requested by the users at job submission and can not change it during the execution of the job. A typical rigid job will not execute with fewer resources and will not make use of more resources.
   - *moldable*, if the scheduler determines the number of resources assigned to a job when it is started. Moldable jobs usually work with a wide range of resources, but also have a minimum number of resources to use. Additional resources improve the performance, possibly up to a saturation point. The decision of the scheduler on how many resources are actually assigned to the job is influenced by the job properties, system load and other constraints. However, the number of assigned resources is not changed throughout the run time of the job.
   - *evolving*, if different phases of parallelism occur naturally in the application, so that the number of required resources changes during the run time. The job itself decides if more resources are needed or unused resources are returned to the resource management system. The system must satisfy the requests of the job, because otherwise the job will not continue with its execution.

	operating system	application run time	advantages	disadvantages
fixed	predefined partitions	parallelism is hardcoded or specified by a parameter	simple, preserves locality	internal fragmentation, limited multiprogramming, arbitrary queuing
variable	allocation according to request (possibly rounded up)	hardcoded parallelism	matches requests, preserves locality	external (and possibly internal) fragmentation, arbitrary queuing
adaptive	allocation subject to load when launched	parallelism is a parameter	preserves locality, adapts to load, improved efficiency	external fragmentation, some queuing
dynamic	allocation changes at run time to reflect changes in load and requirements	express changes in potential parallelism and adapt to changes in available parallelism	no fragmentation, queuing only under high load, adapts to load, improved efficiency	does not preserve locality, partitions are not independent, restrictions on programming model

Table 2.1: Four types of partitioning [16].

- *malleable*, if the number of resources assigned to the job can be changed by the system. The scheduler decides if the job has to release resources or if additional ones are assigned. The job has no influence on this decision.

The difference between evolving and malleable jobs is who decides on changing the number of assigned resources. However, evolving and malleable jobs should usually go together. With increased flexibility the scheduler has more opportunities to schedule the job. In general, a job is more likely to be started earlier if it is moldable or even malleable rather than being rigid. Nevertheless, the task of implementing a job of increased flexibility requires a lot of effort. Hence, only stable and frequently used applications should be modified to be malleable. Evolving jobs usually have system calls integrated, which indicate when more or less resources are needed. Table 2.2 summarizes the classification of job flexibility.

		when is it decided?	
		at submission	**during execution**
who decides?	**user**	rigid	evolving
	system	moldable	malleable

Table 2.2: The four types of jobs based on specifying the number of resources [26].

3. **Level of preemption supported:** During their run time jobs or single threads may be preempted and potentially relocated in order to decrease resource fragmentation or to increase response time of short running jobs:

- *no preemption*: once a job is started it continuously runs to completion on the same set of resources.
- *local preemption*: single threads of a parallel job may be preempted, but they are later restarted on the same processor. With that no migration and data movement is required.
- *migratable preemption*: additionally, single threads of a parallel job may also be resumed on a different processor than they were preempted. Obviously migration and data movement mechanisms are needed.
- *gang scheduling*: all active threads of a parallel job are suspended and resumed simultaneously (with or without migration).

For real systems preempting a parallel job means, that all threads are stopped in a consistent state, no messages are lost and the state of each job is preserved [28]. Whether the data of the job is removed from memory or not, depends on the memory requirements and the amount of available memory. If single threads are moved from one processor to another, communication paths have to be changed and thread data has to be moved. Especially in shared memory systems, the last point is easier to establish than in distributed memory systems.

4. **Amount of job and workload knowledge available** to the scheduler and used by it:

   - *none*: all jobs are treated the same upon submission.
   - *workload*: the overall distribution of run time is known for the whole workload. Information about individual jobs is not available. All jobs are treated the same, but scheduler parameters can be changed to match the workload.
   - *class*: each submitted job belongs to a class. Key characteristics (estimates about the processing environment, maximum parallelism, average parallelism, and possibly more detailed speedup characteristics) for each class are known.
   - *job*: the exact run time of the job is known for any given number of processors.

   In practice complete job knowledge is unrealistic [28]. Any information provided by users has to be treated carefully as the accuracy of this information is probably wrong. And users may also deceive the scheduler intentionally. User information is historically unreliable and users are not careful about making good estimates. The scheduler may either measure the efficiency during the run time and assign more processors only to those jobs which use their assigned resources efficiently. Or the scheduler may store past data about run times and speedups and use it for future executions.

5. **Memory allocation** is usually critical for high performance applications:

   - *distributed memory*: the system consists of many nodes, each having one or more processors and associated memory. Message passing is used for communication and data access in remote memory.
   - *shared memory*: the cost for accessing memory is either uniform (UMA) or nonuniform (NUMA). With UMA, the allocation of memory resources is more equitable, whereas in NUMA architectures the performance depends on the allocation of a job to processors and its data to memory.

> The authors of [28] state, that shared memory systems are usually more difficult and thereby more expensive to implement. Nevertheless, such systems are very popular as programming shared memory applications is quite easy. On the other hand, distributed memory systems are more easy to realize because standard hardware is used for nodes and they are connected with a standard network. However, the task of implementing distributed memory applications is more difficult. Message passing libraries like MPI or PVM are used to communicate with other threads of the same parallel job that run on a different node.

With the above definitions the model applied in this work uses variable partitioning, the jobs are rigid, no preemption is possible, no job or workload knowledge exist and the memory allocation is distributed.

According to [27] today's resource management systems are queuing systems as jobs arrive, potentially waiting for a specific time, then receiving the requested service and finally leaving the system again. Such systems are either on-line or off-line, with the on-line systems being further distinguished in open and closed systems (cf. Figure 2.1).

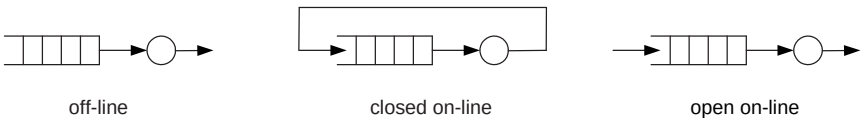


Figure 2.1: Three types of queuing systems.

In an *off-line* scenario all jobs are known at the beginning and their resource requirements, too. No further jobs arrive later while the system is running. With that, an optimal planning of all jobs is possible because the situation does not change. An off-line scenario is often used for space-sharing or batch job schedulers and their performance is often predicted by analytical methods.

In an *on-line* scenario jobs arrive all the time. The scheduler has to process new jobs without the benefit of knowing about future job arrivals. Therefore, planning is only possible for the current situation. Presumably the next submitted job makes it necessary to change the current plan and a new solution is required. Hence, computing a new solution has to be fast, but most often this is critical. In *closed* on-line systems it is assumed that the set of jobs is fixed. That is, arrivals of new jobs depend on terminations of previous jobs. The most realistic model is *open* on-line, where an endless stream of jobs arrive and departing jobs have no influence on new jobs. This model is more complex than the other two, as the arrival process also needs to be specified. Additionally, an open on-line scheduler must be able to deal with extreme situations, e. g. temporary high workloads. Hence, it is interesting to evaluate when a scheduler breaks down and the incoming load is no longer handled. The system becomes saturated, as an increased workload does not result in a higher utilization, but the quality of the provided service (e. g. average response time) drops significantly. We use the open on-line model in this work, as in real world scenarios for scheduling HPC systems and grid environments, jobs arrive over a period of time, i. e. all job properties are not known from the beginning, and newly submitted jobs do not depend on recently finished jobs.

In an open on-line scenario users typically react on the schedulers performance (e. g. the response time). If the response time is small, more jobs are submitted and the load increases

(i. e. load is a function of response time). At a certain level the load becomes too high and the scheduler can no longer provide a good service. From there on, response time is a function of the load. With this, a stable state exists [27]. It requires a suitable amount of users that are potentially willing to submit many jobs.

Each of the three models has its own performance metrics of choice. The response time is probably the most important metrics for interactive jobs, whilst the utilization is more suited for batch jobs. In an open on-line system the utilization depends mainly on the arrival process and the resource requirements of the jobs. Utilization does not depend on the schedulers performance, which leaves the response time as the main metrics. Of course, the response time should be measured at different load conditions which directly refer to the arrival process and the rate new jobs arrive. In a closed on-line system the arrival process depends on how many jobs are re-submitted after their termination. Therefore, the number of jobs processed per time unit or the throughput is a effective metrics to use. Makespan is the metrics of choice for the off-line model. It denotes the time at which the whole workload terminates. It can be seen as an off-line version of the response time. In an open on-line scenario the makespan is not very meaningful, as it typically depends on the last submitted job.

In 1999 Krallmann, Schwiegelshohn, and Yahyapour presented a general approach on designing job scheduling systems [53]. They state, that job scheduling obviously does/should not effect the outcome or accuracy of a job. However, it may have a significant influence on the efficiency of the whole system, because a good scheduler reduces the loss of resources to fragmentation and thereby increases the system utilization and job throughput. A good scheduling system is of major importance for the management of large computing resources, as supercomputers (independent of their size) often represent a significant investment for the company or research institute who owns it. Therefore, almost all developments in job scheduling are induced by the owners of these machines (e. g. EASY backfilling was developed at Argonne National Lab for their IBM SP system). The authors state, that "on the other hand some manufacturers showed only limited interest in this issue as they frequently seem to have the opinion that machines are not sold because of superior job schedulers". Hence, the authors present guidelines for the selection and evaluation of job scheduling systems for multiprocessor machines.

Scheduling a multiprocessor system is a typical on-line scenario in the real world, as a stream of job submission data arrives over a period of time. The submitted data is subdivided in three categories:

- user data (e. g. the name, project, or compute quota) is used to generate job priorities as some users jobs might be preferred.
- resource requests specify e. g. the number and type of resources needed, the length of time the resources are needed, the amount of memory, I/O, or network performance or other hardware and software requirements. Note, some of the data may only be estimated, especially the run time.
- scheduling objectives may include additional information for the scheduler, e. g. deadlines that have to be met.

Now the task of the scheduling system is to generate an allocation of system resources to individual jobs for a certain period of time. If further hardware or software restrictions

(e. g. gang-scheduling, or at most one application is active on a processor at any time) are not observed, a schedule may be invalid. Some systems with a specific interconnect topology might require the schedule also to be mappable on the resources. Neglecting the machine architecture (e. g. the topology of the interconnect) during the scheduling phase may result in schedules, where enough resources are available for a specific job, but it is impossible to generate a continuous partition with the same set of resources. Hence, validity constraints are machine dependent, but in general it is assumed that schedulers do not attempt to produce invalid schedules.

According to the authors the scheduler is divided in three parts:

1. **Scheduling policy:** This is a collection of rules that is used to determine the assignment of resources to jobs, if not enough resources are available to satisfy all jobs at this time. These rules may be formulated in a non-formal, non-mathematical style and they should allow a rough distinction between good and bad schedules. Other more general rules need not have to be specified explicitly, like e. g. finish every job as soon as possible without breaking any other rule. A good scheduling policy should also contain rules to solve conflicts between other rules. And of course the scheduling policy has to be implementable.

2. **Objective function:** The goal is to have a single scalar value (schedule cost) that represents a complete schedule. It is then possible to distinguish good from bad schedules and furthermore rank schedules. Schedules (or better the according scheduling policies) are comparable, if the same job set is used as input. An upfront calculation of the objective value may not be possible, as an execution of all jobs might be necessary e. g. to detect their actual run time. With almost all machine installations, simple objective functions are used, like the average response time, system utilization, or job throughput.

3. **Scheduling algorithm:** The algorithms task is to generate a valid schedule for the stream of submitted jobs in an on-line fashion. A good scheduling algorithm should always generate good or even optimal schedules according to the objective function. Additionally, the computation should not take 'too much' time or use 'too many' resources. Unfortunately, most scheduling problems are computationally hard, so it is not possible to have an algorithm that guarantees the best possible schedule. Especially in an on-line scenario where jobs are sometimes submitted at a high rate and a new schedule has to be computed after each job submission. The time for generating a solution is sometimes more important than the optimality of the solution.

Selecting a good scheduling algorithm depends on many constraints, but most of the time the administrative staff simply choose an algorithm from the literature and adapts it to their local environment. Despite the fact that the algorithm should compute a valid schedule, it has to be determined, whether good or bad schedules are generated. To evaluate this, either algorithmic theory or discrete event simulations are used. The job input for simulations is taken from an actual trace or from a workload model.

According to the authors the result of an evaluation without algorithmic theory is, that often competitive analysis can not be applied to methods which are based on very complex algorithms or which use specific data sets as input successfully. Furthermore, competitive factors are the worst case factors, which are not acceptable in a real world environment. For example a competitive factor of 2 for the utilization means, that half of the resources are not

used. The reliability of a simulation based evaluation depends on the availability of correct job input and the compliance of these jobs with the actual workload on the target machine.

In a recent paper [20] it is stated that the goal of performance evaluation is not to obtain absolute numbers but rather differentiate between alternatives. Both the used metrics and job input have a strong influence on the results of a performance evaluation. Hence, it is appropriate to use common metrics and workloads in order to standardize the evaluation process and make the results comparable. One approach is to use real traces from the Parallel Workloads Archive [89].

2.2 Analytical Results

In 1979 Graham, Lawler, Lenstra, and Rinnoy Khan [38] introduced a model for describing scheduling problems. It consists of three parameters $\alpha|\beta|\gamma$. n jobs are processed and the number of resources or processors is m. j describes a job and a specific processor is marked with i.

α represents the machine environment and is 1, if a machine with a single processor is used, or P, if a parallel multi-processor machine with m identical processors is used. Other possible values for α can be Q for describing a system with m processors in series where each job has to be processed on each of the m processors on the same route, or O if the route of processing is no longer fixed and some of the processing times may be zero.

β describes the job characteristics and processing restrictions. In contrast to α multiple entries are possible for β. If p_j is specified, processing times of the jobs are given. $size_j$ stands for the number of processors needed by jobs and is only relevant, if parallel jobs are scheduled and more than one processor is available. Otherwise sequential jobs are assumed. r_j is specified if jobs arrive over a period of time and information about all jobs is not available from the beginning. Jobs are not started before their release date r_j. *pmtn* indicates, that job preemption is allowed. That means jobs do not have to run to completion once they have started. The scheduler is allowed to interrupt the execution of a job at any time for starting a different job. Information and results of the preempted jobs are preserved and the original total processing time is not affected by the preemption. Several more options exist and an almost complete list is found in Section 2.1 of Pinedo's Book on Scheduling [72].

Finally γ describes the objective function. Three are most common: the weighted sum of completion times $\sum w_j C_j$, the makespan C_{max} and the sum of flow or response times $\sum F_j$ with $F_j = C_j - r_j$.

The book [1] by P. Brucker is about scheduling algorithms and is structured in three parts. In the first part of the book an introduction to and classification of scheduling problems is given. Next, classical algorithms for solving single and parallel machine problems, and shop scheduling problems are presented. Finally, multiprocessor task scheduling problems and problems connected to flexible management (e. g. scheduling with due dates) are discussed. The book also summarizes complexity results for different scheduling problems at the end of each chapter.

In [22] a new processing restriction for β, the stability constraint *stbl*, is presented. Although release dates and an on-line algorithm already describe a real system very well, an important fact has to be considered, too: on average the resource requirements of submitted

jobs must be less than the system's capacity. The authors compare it with queuing theory where this fact is expressed by the requirement of $\lambda/\mu < 1$. λ is the arrival rate in jobs per time unit and μ is the service rate in the same unit. In queuing theory the average time that jobs spend in the system is predicted, under the assumption that the system does not become saturated. If the system becomes saturated, the queuing time may rise without a bound, which is not tolerated. The authors state, that in the common theoretical model of scheduling an illusion of stability exists, as all jobs are indeed scheduled at some point. However, this is because of the finite input and not of the constraint on arriving jobs. Initially, the load increases as jobs have to wait and later on the system becomes drained. Except for limited fluctuations, this is not allowed in real systems.

Transferring the stability constraint from queuing theory to the field of job scheduling and using the model with release dates, leads to:

$$\sum_{r_i \leq r_j, i \neq j} p_i \leq r_j + c$$

If r_j is the release date for job j and p_j is its processing time, then it is required that for all j the accumulated processing time of previous jobs fit in up to a positive constant c. c allows load fluctuations which often occur at the beginning, where the offered load may be higher than the systems capacity. The connection to the $\lambda/\mu < 1$ from queuing theory is obvious: if jobs are indexed according to their arrival times ($r_i \leq r_j$ iff $i < j$) and a large number of jobs are observed, then $\lambda \approx j/r_j$. The service rate is approximated by the reciprocal of the average processing time ($\mu \approx j/\sum_{i \leq j} p_i$). Putting it all together results in $1 > \lambda/\mu \approx \frac{j/r_j}{j/\sum_{i \leq j} p_i} = \sum_{i \leq j} p_i/r_j$. And extending the expression for parallel jobs:

$$\sum_{r_i \leq r_j, i \neq j} \frac{p_i \cdot size_i}{m} \leq r_j + c \tag{2.1}$$

The constant c is also used to allow some degree of overlap between jobs that do not use all processors of the system.

The authors state about the stability constraint that is expresses the fact that the offered load to the system is bounded (on average) by its capacity. However, it does not guarantee that the system stays stable. It may saturate because of an inefficient scheduling algorithm or other constraints, e. g. if the size of all jobs is $\lceil m/2 \rceil + 1$, half of the processors will be wasted.

With this stability constraint new results for the identical parallel machine model with sequential jobs ($P|stbl|\sum w_j C_j, C_{max}, \sum F_j$) and the multi-processor machine model with parallel jobs and preemption ($P|stbl, size_j, pmtn|\sum w_j C_j, C_{max}$) are developed [22]. An overview of the results are given in Table 2.3. Although these models already come close to reality, an additional assumption is done: at most, one job can be processed at a time. Unfortunately, this is the most restricting, as it substantially limits the usage of a large space-shared supercomputer.

Much more research has been done on scheduling problems for many decades and in many research fields of computer science, operations research, and discrete mathematics. Developing efficient resource management software for computer systems has been used as a reason to drive this research. This especially holds for the field of job scheduling. Hence, it is obvious to use these theoretical approaches directly. However, according to [28] many of the theoretical

model	competitive ratio for $\sum w_j C_j$	C_{max}	$\sum F_j$
$1\|stbl\|\sum w_j C_j,\ C_{max},\ \sum F_j$	2	2	1
$P\|stbl\|\sum w_j C_j,\ C_{max},\ \sum F_j$	2	2	1
$P\|stbl, size_j, pmtn\|\sum w_j C_j,\ C_{max}$	3	3	

Table 2.3: Competitive ratios with the new stability constraint from [22].

results rely on "a creative set of assumptions in order to make their proofs tractable. This divergence from reality does not only make them hard to use in practice, but also the diversity of divergence makes them hard to compare with each other." One such example is to allow preemptions. It is difficult to realize preemptions in real systems, as the requirements from e. g. the hardware (a lot of memory and support from the interconnection network is needed) and the software side (the operating system has to suspend the whole job on all resources) are hard.

Information about approximation factors for off-line algorithms and competitive ratios for on-line algorithms are difficult to apply, as the worst case is very unlikely in the real world. An example clarifies the problem of worst case analysis: consider an algorithm with a competitive ratio of 2 and a system with an average utilization of around 90%. In a worst case scenario the utilization would drop to about 45% and more than half of the resources would be lost. As the owner of the system is interested in a good and efficient usage of the machine, such a situation is not tolerable and will surely not often occur. Studying the worst case behavior of backfilling, shows a similar situation. In the worst case it is not possible to start any job out of order, so the backfill routine has no influence on the scheduling at all. The original starting order induced by the sorting policy is retained and in this example standard FCFS list scheduling is done. Therefore, adding backfilling to a list scheduling algorithm like FCFS does not change its worst case behavior. Nevertheless, evaluating backfilling in the average case, so using real, trace based job sets as input, shows a significant performance benefit which is also measured in practice [29].

2.3 Workload Characterization

Many machine installations and their resource management systems store data about past usage of the system and moreso about executed jobs. Such data is called workload, accounting log, or trace. Traces provide a wealth of information about every job in the system. It is common to store data about scheduled and executed jobs, i. e. the time a job was submitted, started, and finished and how many resources the job used. Furthermore, less scheduler important data is stored which is later used for accounting reasons and for statistical analysis. To mention some is the name of the user, the project code, the executable, and more than likely command line parameters for the executable of the submitted job.

Although many large supercomputer installations exist today and most of them store data about executed jobs, only some traces are publicly available. Possible reasons are: a low utilization of the machine should be kept private, the machine may be classified if e. g. the machine is owned by an industry or government, but most often the traces are not published as there is no direct benefit for the administrative staff in doing this. Nevertheless, two trace archives exist today. Both are driven by research interests in the field of job scheduling. The

Parallel Workload Archive [89] and the HPC Workload Trace Repository [60] maintained by the Maui scheduler project at supercluster.org. The available traces are presented in more detail later.

While describing jobs the terms 'large' and 'small' (but also 'wide' and 'narrow') refer to the number of requested resources of the job which is also often called 'size' or 'width'. The run time, duration or '(execution) length' is characterized with the terms 'long' and 'short'.

In general a classification of supercomputer systems is possible by their size, ownership and usage:

- **Production machines** typically have many processors (≥ 128) and are equipped with powerful hardware, e. g. main memory, disk storage, interconnect, air conditioning, power supply. The hardware comes from well-known vendors like IBM, Cray, SGI, or SUN and machines of the same type are found all over the world. Users come from almost all research areas (e. g. genome sequencing, protein folding, astro physics, numerical and non numerical algorithms, ...) but also from the industry (e. g. automotive or chemical). Industrial users typically have to pay for the resource usage but sometimes parts of the machine are directly owned by companies [44]. Production machines are installed at national high performance supercomputing centers and only a minority of the users are internal from that center. Most of the users are from external institutions from all over the world.
  A large amount of users guarantee a steady arrival process with bursts being rare. The number of jobs submitted in a fixed time interval (e. g. 24 over hours) is large. Consequently jobs usually have to wait for execution, but at the same time the scheduler has more possibilities in picking an appropriate job for utilizing free resources. Hence, less resources remain idle and the utilizations is high (approx. 80 - 90%). The run time of production jobs is long and interactive application development is usually not done. Idle time or even down time due to hardware failure or maintenance is avoided as much as possible. The aim is to have the system available 24 hours a day, 7 days a week.

- **Research machines** are typically smaller than production machines (≤ 64 processors). Standardized and cheap hardware components of the shelf are used (e. g. PC based Linux clusters) in order to reduce the costs. Such machines are found in research groups or small computing centers where mostly local and internal users submit jobs. The types of applications do not differ much from production machines. However, more application development is done and job run times are on average shorter.
  Many implications arise from this usage: The arrival process is less continuous and bursts are more likely. The number of waiting jobs is small and slowdown values are close to the minimum of one (Table 3.6). The utilization is low (at about 40 to 60%) and less continuous (Figure 2.2 gives an example). Maintenance and errors in the system occur more often. However, this is not critical, as a 24/7 service is not intended. Assigning the machine to a single user or a group of users for exclusive usage is possible.

Note, the given limits on the amount of processors for each type of machine are not strict. For example, vector machines usually have much lower processor numbers, but are mainly used for production purposes. And also PC clusters with large processor and node numbers exist, which are mainly used for research.

Much work was done in the past on analyzing and characterizing workload traces. Subhlok, Gross, and Suzuoka [88] analyzed and compared traces from the 512-node IBM SP2 machine

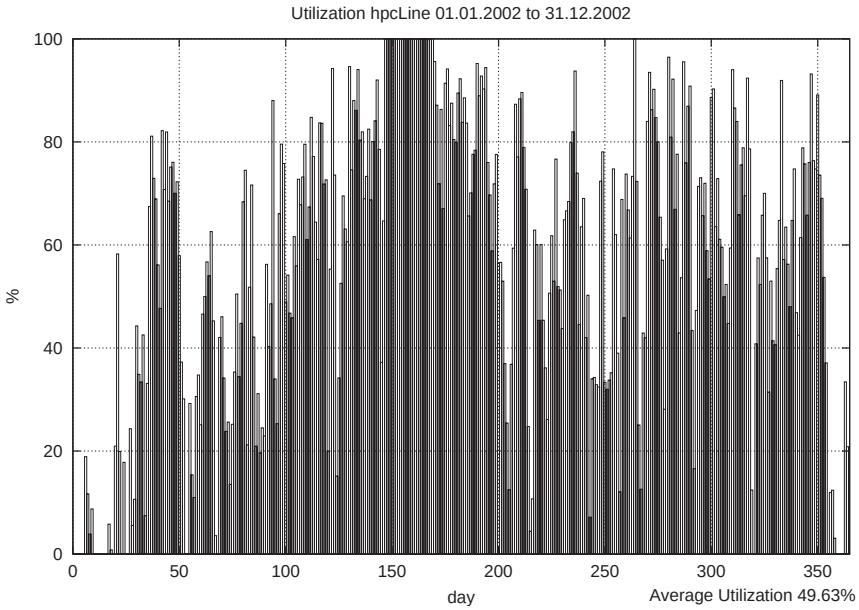


Figure 2.2: Utilization of the *hpcLine* cluster at the PC^2[70].

installed at the Cornell Theory Center, the 512-node Cray T3D at the Pittsburgh Supercomputing Center, and the 96-node Intel Paragon at the ETH Zurich. They observed a high percentage of small (sequential) jobs, but their part of the resource usage was small. On the other hand only some large jobs exist, which are in turn responsible for most of the resource usage. In general, users tend to use 'power of 2' job sizes and indeed it was observed, that most of the jobs are clustered around power of 2 sizes. The majority of jobs are of medium width (requesting 8-64 processors) and length (between 1 and 12 hours). The authors state, that one implication for scheduling is to use space-sharing. As many small and short jobs with almost no resource usage exist, a scheduler should be able to prioritize these jobs without significantly affecting large jobs. Applying backfilling by starting jobs out of order without delaying other jobs is suitable.

The tendency to use power of 2 jobs is based on the resource needs of most applications and the topology of the machine interconnect. Debugging and testing, but also jobs from the administrative staff, cause the high percentage of small and sequential jobs. Having not enough software licenses or only some nodes equipped with a large main memory further increases this percentage. Short running jobs are typically caused by errors and bugs in the according application, so such jobs fail after only some seconds of run time. On research machines small and short jobs occur more often than on production machines.

In [24] similar observations whilst analyzing the trace of the 128-node iPSC/860 system installed at NASA Ames are made. A large number of sequential jobs dominate the trace while the resource usage (measured in node seconds) is mostly generated by parallel jobs. Actually most (over 80%) of the sequential jobs were submitted by the administrative staff and consisted of a simple UNIX 'pwd' command. It is used to check, if the machine is up and running. The average run time of jobs grew with the number of requested processors, so that the area (resource usage) grew superproportionally. A distinction for the submission

rate is done by day (prime time) and night (non prime time). During the day the rate is about one job every two minutes. The average size of running jobs is small as much testing and development is done. This changes at night as more production jobs are executed. These are usually executed on a larger set of processors and their run time is longer. Even though the submission rate is low at night (the production jobs were submitted towards the end of the day shift) the achieved utilization is high. On weekends this behavior is more extreme, as the submission rate is lower than on weekdays as most of the time production runs with a high resource usage (more resources for a longer run time) are executed. The iPSC/860 system is used as a research machine during the day in the week and as a production machine for the rest of the time. The trace also shows, that applications are started repeatedly for a significant number of times which confirms the production machine's status.

In general, job run times and sizes, but also the user behavior (which is reflected in the arrival process) depend on the properties of the scheduler. For example, if the scheduler allows an estimation of continuous run times, certain run time values occur more often than others e. g. 10 minutes or 1 hour. On the other hand some schedulers (typically queuing systems like NQE/NQS) only have upper limits for the run time which result from the available queues. Users only specify the queue for their job and a precise run time estimate. The NASA Ames iPSC/860 machine is an example that works with a queuing system (cf. Table 3.1). Other systems even restrict the resource usage e. g. to power of 2 job sizes starting from 32 as it is done on the Thinking Machines CM-5 installed at Los Alamos National Lab (LANL).

Workload traces also contain a lot of system administrative induced jobs as previously described. One question is, whether these jobs should be removed from the real trace or not? As no significant work is done and these jobs do not reflect the machine users at all, one could argue that these jobs should be eliminated from the trace. However, these small and insignificant jobs influence the system behavior as they occupy resources and user jobs have to compete with them. Neglecting these jobs in the trace would falsify the users 'look and feel' of the machine and the workload would no longer represent the real world. Furthermore, the authors state, that down time has to remain in the trace. For example the iPSC/860 machine was down due to maintenance reasons (air conditioning, software and hardware failures) for about 7.5% of the traced time frame.

2.4 Self-Tuning and Dynamic Policy Switching

In 1994 Ramme and Kremer [73] described the problem of scheduling a machine room of MPP-systems. 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 [32, 73] 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.

Feitelson et al. [24, 23] described a similar approach for the NASA Ames iPSC/860 system. 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 out of a modern resource management system is described in [23]. 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. Speaking in biological terms: 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. Now 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 1996 Nguyen et al. [66] evaluated the processor allocation to iterative parallel applications on a shared memory multiprocessor machine. In their example some applications show a behavior where assigning more processors results in a degraded performance. Their definition of self-tuning is when "an application determines for itself the best number of processors to use". They present three levels of processor allocation, each being more sophisticated:

1. A good processor allocation is determined once at the beginning of the execution.
2. A new processor allocation is determined repeatedly, either in fixed intervals or after dramatic efficiency changes.
3. In each parallel phase of an iteration a new processor allocation is computed.

They show, that a dynamic selection of processor allocation matches the best static allocation. However, this is done with the potential benefit of finding even better solutions that outperform any static allocation.

2.5 Scheduling in Meta- and Grid-Computing Environments

The cooperation of geographically distributed computing resources for solving large problems became a hot topic in the research of the last ten years. In 1992 Smarr and Catlett first defined the term metacomputing [83]. They define a metacomputer as a network of heterogeneous, computational resources which are linked by software so that they can be used as easily as a desktop PC. Three stages are proposed in constructing a metacomputer:

1. A software and hardware integration effort is at the beginning in order to create and harness the necessary software components. Furthermore, this stage involves interconnecting all involved resources with high-performance networks, implementing a distributed file system, coordinating the user access inside the metacomputer, and making the environment seamless by using existing technology.
2. The second stage moves beyond the software integration of a heterogeneous network of computers. Now a single application is spread across several computers. With that many computers process a single problem cooperatively. This allows users to solve problem scenarios which are virtually impossible without a metacomputer. Problems may now be larger in size, more complex, or the precision of the solution is increased. The authors state, that in general the evolution of metacomputers is limited not only by the software layer but also by the network that connects the heterogeneous computing resources. Due to the differences in bandwidth and latency of network links the capabilities of a local area metacomputer are typically one year ahead compared to a wide area metacomputer. And this is still more or less true today, more than 10 years after the authors stated this.
3. The third stage of a metacomputer is a transparent national metacomputer. The authors state, that the amount of computational resources useable by applications will dramatically increase. However, this third stage involves more. An adequate wide area network infrastructure has to be in place and standards at the security, administrative, file system, and accounting level have to be developed. This then enables multiple local area metacomputers to cooperate.

In the remainder of [83] the two authors describe several showcases from the SIGGRAPH'92 conference. There, metacomputers were used for:

- Theoretical simulations to solve scientific equations numerically. The application requires the metacomputer to easily interconnect several computers to work on a single problem at the same time.
- Instrument and sensor control: The metacomputer translates raw data from scientific instruments and sensors into visual images. This also allows the user to interact with the instrument or sensor. Hence, a remote observation and instrument control over the network is possible.
- Data Navigation: The metacomputer is used to explore large databases and translating numerical data into human processable input. As a vision the authors state, that "over the next several years, we will see an unprecedent growth in the amount of data that is stored as a result of theoretical simulations, instruments, and sensors."

The web page of the DataGrid project [9] contains a good and summarized introduction to grid environments: The idea of computational and data grids dates back to the first half of the 90's. The vision behind them is often explained using the electric power grid metaphor. The electric power grid delivers electric power in a pervasive and standardized way. One can use any device that requires standard voltage and has a standard plug if one is able to connect it to the electric power grid through a standard socket. When electricity is used one does not worry where it is generated and how it is delivered.

As explained by Foster and Kesselman in the second chapter of their book "The Grid" [30]: "The current status of computation is analogous in some respects to that of electricity around 1910. At that time, electric power generation was possible, and new devices were being devised that depended on electric power, but the need for each user to build and operate a new generator hindered the use. The truly revolutionary development was not, in fact, electricity, but the electric power grid and the associated transmission and distribution technologies".

Currently millions of computing and storage systems exist all over the planet connected through the Internet. However, missing is an infrastructure and standard interfaces capable of providing transparent access to all this computing power and storage space in a uniform way. For the scientist, the vision that is now becoming reality with modern grid environments is as follows:

- The user submits his request through a Graphical User Interface (GUI) simply by specifying high level requirements (the kind of application he wants to use, the operating system,...) and eventually providing input data.
- The Grid finds and allocates suitable resources (computing systems, storage facilities, ...) to satisfy the user's request.
- The Grid monitors all processing requests.
- The Grid notifies the user when the results are available and then after some time presents them.

As seen, the users do not have to know which resources they are using or where they are. They just receive computing power and storage space from the Grid through a standard interface.

A quote from Foster's and Kesselman's book summarizes the definition: "A computational grid is a hardware and software infrastructure that provides dependable, consistent, pervasive and inexpensive access to high-end computational capabilities".

In other words, grid environments take up the former metacomputing idea and broaden it. Other different kinds of resources are now joined in a grid environment and they can be used collaboratively. Examples are data archives, physical instruments and sensors, and three-dimensional graphical output devices. The wide area networks themselves are also included in the grid resource management, as their bandwidth and latency is subject to frequent changes and most often overloads.

In 1999 Gehring and Preiss published work about scheduling a Metacomputer with uncooperative sub-schedulers [31]. The authors state, that in an intuitive and simple metacomputing scheduling architecture the local schedulers would be replaced by metacomputer scheduling modules. In this case local machines are no longer accessible directly, the metacomputer frontend has to be used in any case. However, drawbacks are obvious, as with this approach the site autonomy is no longer assured. Furthermore, local users can not be forced to submit all their jobs through the metacomputer.

Therefore, a two tier scheduling architecture is required, which preserves the local scheduling instances. In this approach a metacomputing super-scheduler is placed on top of the local resource management schedulers. The authors described three levels of information interchange between the two tiers:

- scheduling with no control: The metacomputer has no knowledge about locally submitted jobs. Furthermore, no information is available on the amount of resources assigned to locally submitted jobs and how many resources are free for the metacomputer.
- scheduling with limited control: The metacomputing scheduler is able to query information from the local resource management about how many resources are assigned to local jobs. The returned data can be unreliable, as the jobs' resource requirements are not typically known in advance.
- scheduling with full control: Whenever a job is submitted to the local resource management scheduler, it is forwarded to the metacomputing scheduler. Hence, site autonomy and local scheduling policies are abandoned.

Obviously the scenario with full control is the most promising scheduling approach.

Especially in this last approach the metacomputer has to find a common basis for scheduling. This means, that the least common multiple of all resource management system flavors, scheduling policies, and other local restrictions have to be found. By that a drawback is obvious: special features, which are only available at some or only one site, may not be useable through the metacomputer frontend. Most likely local users, who are now forced to use the metacomputer, will become dissatisfied.

The authors state, that scheduling with no control is the reference case, as most metacomputing environments at that time worked with this approach. Scheduling with full control represents the idealized scenario and scheduling with limited control is a compromise of both.

In 1998 Czajkowski, Foster, Karonis, Kesselman, Martin, Smith, and Tuecke presented a resource management architecture for metacomputing systems [8]. They state, that the

resource management comes with five challenging problems: site autonomy, heterogeneous substrate, policy extensibility, co-allocation, and online control.

1. Site autonomy: As resources are typically owned and operated by different organizations and under different administrative domains, a common understanding of e. g. scheduling policies and security mechanisms is difficult to obtain.

2. Heterogeneous substrate: This problem is induced by the fact that different resource management systems are used at the local sites, e. g. CONDOR, LoadLeveler, PBS, LSF, or NQE/NQS. Even if two sites use the same resource management system, the configurations might differ due to different scheduling and machine usage policies. That means, that diversity in functionality is growing even more.

3. Policy extensibility: Metacomputing applications come from a wide range of domains, each domain or application with its own requirements. A resource management system has to support the frequent development of new domain-specific management structures, without the necessity of changing already installed codes at local sites.

4. Co-allocation: Many applications come with resource requirements that can only be satisfied by using resources simultaneously at multiple sites. Because of the site autonomy and the possibility of failures during the allocation, the resource management needs special mechanisms for allocating multiple resources, initiating the computation on the resources, and monitoring and managing the running computations.

5. Online control: It might be necessary to adapt the resource requirements of an application to current resource availability, especially if the requirements and resource characteristics change during the run time. This requires the ability of the resource management system to negotiate resource requests.

The authors state, that no resource management system at that time addresses all five problems. In the remainder of [8] they present a resource management system approach that addresses all five of the named problems. Included is a description of the GRAM (Globus Resource Allocation Manager), the interface to the local resource management system.

3 Job Scheduling and Evaluation Methodologies

In the previous chapter we presented related work on parallel job scheduling, some analytical results, characterizations of commonly used workload sets, and scheduling in grid environments. By that, a global scope for this work has been defined. In this chapter we present general aspects, common terminology and several definitions for parallel job scheduling, which are used both for single machine scheduling, as well as for scheduling within grid environments. The chapter begins with a classification of resource management systems. Queuing based systems schedule only the present resource usage, while planning based systems schedule the present *and* future resource usage. The self-tuning dynP scheduler described in Chapter 4 is based on this approach of scheduling in a resource management system.

After the classification, an overview on different policies for scheduling single HPC machines is given in Section 3.2. Scheduling policies are often enhanced by *backfilling* in order to increase the utilization and response time. The concept of backfilling and its variants are presented in Section 3.3. Metrics are needed in order to rate schedulers and measure their performance. Performance metrics can be classified in owner and user centric metrics. In Section 3.4 several metrics are presented which are later used in the evaluations of the self-tuning dynP scheduler. The workloads used as job input for the evaluation are described in Section 3.5. A general overview on available traces and models is given, a subset of traces for the evaluation is chosen, and an analysis for these workloads is done. Additionally, approaches for increasing the workload are presented, so that an evaluation of the schedulers with different workloads is possible. Finally, the simulation environment used for the evaluation of the self-tuning dynP scheduler is described in Section 3.8.

3.1 Classification of Resource Management Systems

Before the classification begins we define some terms that are used in the following.

- The term *scheduling* stands for the process of computing a schedule. This may be done by a queuing or planning based scheduler.
- A *resource request* contains two information fields: the number of requested resources and a duration for how long the resources are requested for.
- A *job* consists of a resource request plus additional information about the associated application. Examples are: information about the processing environment (e. g. MPI or PVM), file I/O and redirection of stdout and stderr streams, the path and executable of the application, or startup parameters for the application. We neglect the fact that some of these extra job data may indeed be needed by the scheduler, e. g. to check the number of available licenses.
- A *reservation request* is a resource request starting at a specified time for a given duration. Once the scheduler accepted such a request, it is a reservation.

We call a reservation Fix-Time request to emphasize that it can not be shifted on the time axis during scheduling. Accordingly we call a resource request Var-Time request, as the scheduler can move the request on the time axis to an earlier or later time according to the used scheduling policy. In the following we focus on resource management systems that use space-sharing.

The criterion for the differentiation of resource management systems is the planned time frame [42]. *Queuing systems* try to utilize currently free resources with waiting resource requests. Future resource planning for all waiting requests is not done. Hence, waiting resource requests have no proposed start time. *Planning systems* in contrast, plan for the present and future. Planned start times are assigned to all requests and a complete schedule about the future resource usage is computed and made available to the users. A comprehensive overview is given in Table 3.2 at the end of this section.

3.1.1 Queuing Systems

Today almost all resource management systems fall into the category of queuing systems. Several queues with different limits on the number of requested resources and the duration exist for the submission of resource requests. Jobs within a queue are ordered according to a scheduling policy, e. g. FCFS (first come, first serve). Queues might be activated only for specific times (e. g. prime time, non prime time, or weekend). A rather old example from the 128-node NASA Ames iPSC/860 machine running NQS is taken from [93] and is shown in Table 3.1. Another more complex example for the LLNL[1] Craw T3D is found in [21].

	number of nodes			
time limit	16	32	64	128
20 minutes	q16s[1,2]	q32s[1,2]	q64s[2]	q128s[2]
1 hour	q16m[1]	q32m[1]	q64m	q128m
3 hours	q16l	q32l	q64l	q128l

[1] active during prime time

[2] active during weekend day

Table 3.1: NAS's NQS scheduling queue structure for the iPSC/860 machine. Prime time is from Monday to Friday, 6:00 to 20:00 PST.

The task of a queuing system is to assign free resources to waiting requests. The highest prioritized request is always the queue head. If it is possible to start more than one queue head, further criteria like queue priority or best fit (e. g. leaving less resources idle) are used to select a request. There might also exist a high priority queue whose jobs are preferred at any time. If not enough resources are available to start any of the queue heads, the system waits until enough resources become available. These idle resources may be utilized with less prioritized requests by backfilling mechanisms.

As the queuing systems time frame is the present, no planning of the future is done. Hence, no information about future job starts are available. Consequently guarantees can not be given and resources can not be reserved in advance. However, if participating in grid environments

[1] Lawrence Livermore National Lab

this functionality is desirable. Using reservations eases the way of starting a multi-site application which synchronously runs on different sites. By reserving the appropriate resources at all sites, it is guaranteed that all requested resources are available at the requested start time. With queuing systems this still has to be done manually by the administrative staff. Usually high priority queues combined with dummy jobs for delaying other jobs are used.

Users do not necessarily have to specify run time estimates for their jobs, as a queuing system might let jobs run to completion. Obviously users would exploit this by starting very long running jobs which then block parts of the system for a long time. Hence, run time limits were added to the queues (Table 3.1). A longer run time than the limit of the queue is not allowed and the resource management system usually kills such jobs. If the associated application still needs more CPU time, the application has to be checkpointed and later restarted by the user.

3.1.2 Planning Systems

Planning systems schedule for the present and future. With assigning start times to all requests a *full schedule* is generated. It is possible to query start and end times of requests from the system and a graphical representation of the schedule is alsopossible (Figure 3.1). Obviously duration estimates are mandatory for this planning. With this knowledge advanced reservations are easily made possible. There are no queues in planning systems. Every incoming request is planned immediately. The Computer Center Software (CCS) from the PC^2 is such a planning based resource management system [52, 5].

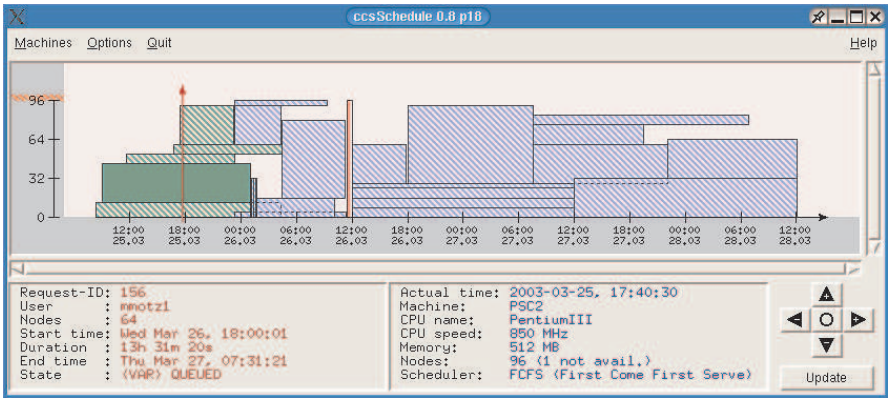


Figure 3.1: The CCS schedule browser.

The *re-planning* process is the key element of a planning system. Each time a new request is submitted or a running request ends before it was estimated to end, a new schedule has to be computed and this function is invoked. At the beginning of a re-plan all non-running requests are deleted from the schedule and sorted according to the scheduling policy. Then all requests are re-inserted at the earliest possible start time in the schedule. After this step each request is assigned a planned start and end time. The non-running requests are usually stored in a list structure and different sorting criteria are applied. They define the scheduling policy of the system.

As planning systems work with a full schedule and assign start times to all requests, resource usage is guaranteed and advanced reservations are possible. A reservation usually comes with a given start time or if the end time is given the start time is computed with the estimated

run time. When the reservation request is submitted the scheduler checks with the current schedule, whether the reservation can be made or not. That is the amount of requested resources is available from the start time and throughout the complete estimated duration. If the reservation is accepted it is stored in an extra list for accepted reservations. During the re-planning process this list is processed before the list of variable requests. It does not have to be sorted as all reservations are accepted and therefore generate no conflicts in the schedule. Furthermore, additional types of job lists are thinkable which are then integrated in the re-planning process according to their priority (reservations should have the highest and variable requests the lowest priority).

Controlling the usage of the machine as it is done with activating different queues for e. g. prime and non prime time in a queuing system has to be done differently in a planning system. One way is to use time dependent constraints for the planning process, e. g. "during prime time no requests with more than 75% of the machines resources are placed". Also project or user specific limits are possible so that the machine size is virtually decreased. Examples for such limitations are:

- Jobs requesting more than two thirds of the machine are not started during daytime, only at night and on weekends.
- Jobs that are estimated to run for more than two days are only started at weekends.
- It is not possible that three jobs are scheduled at the same time where each job uses one third of the machine.

If an already running request interferes with limits during its run time, it is not prematurely killed. It runs until the estimated end is reached. With the examples from above one could think of a job that requests all resources of a machine, is started on Sunday and runs until Tuesday. Such a job would then block the whole machine on Monday which contradicts the limits for Monday.

Planning systems also have drawbacks. The cost of scheduling is higher than in queuing systems. And as users can view the current schedule and know when their requests are planned, questions like "Why is my request not planned earlier? Look, it would fit in here." are most likely to occur [50]. Besides the pure and easily measurable performance of the schedule (e. g. utilization or slowdown), other more social and psychologic criteria might also be considered. It might be beneficial to generate a less optimized schedule in favor of having a more understandable schedule. Furthermore, the usage of reservations should be observed, especially if made reservations are really used. Again, users tend to simply reserve resources without really needing and using them [50]. This can be avoided by automatically releasing unused reservations after a specific idle time.

Table 3.2 shows a summary of the previously described differences between queuing and planning based resource management systems.

3.2 Scheduling Policies

Typical resource management systems store requests in list-like structures. Therefore, a scheduling policy consists of two parts: inserting a new request in the data structure at its

	queuing system	**planning system**
planned time frame	present	present and future
submission of resource requests	insert in queues	re-planning
assignment of proposed start time	no	all requests
run time estimates	not necessary[1]	mandatory
reservations	not possible	yes, trivial
backfilling	optional	yes, implicit
examples	PBS, NQE/NQS, LL	CCS, Maui Scheduler[2]

[1] exception: backfilling
[2] According to [46], Maui may be configured to operate like a planning system.

Table 3.2: Differences between queuing and planning systems.

submission and taking requests out during the scheduling. Different sorting criteria are used for inserting new requests and some examples are (either in increasing or decreasing order):

- by arrival time: FCFS (first come first serve) uses an increasing order. FCFS is probably the most known and used scheduling policy as it simply appends new requests at the end of the data structure. This requires very little computational effort and the scheduling results are easy to understand: jobs that arrive later are started later. With this example the term *fairness* is described [79]. In contrast, sorting by decreasing arrival time is not commonly used, as 'first come last served' makes no sense in an on-line scenario with the potential risk of waiting forever (this is also called *starvation*). However, a stack works with decreasing order of arrival time.

- by duration: Both increasing and decreasing orders are used. Sorting by increasing order leads to SJF (shortest job first) respectively FFIH (first fit increasing height[2]). Accordingly LJF (longest job first) and FFDH (first fit decreasing height) sort by decreasing run time. In an on-line scenario this requires duration estimates, as the actual duration of jobs are not known at submission time. SJF and LJF are both not fair, as very long (SJF) and short (LJF) jobs potentially wait forever. LJF is commonly known for improving the utilization of a machine.

- by area: The jobs area is the product of the width (requested resources) and length (estimated duration). FFIA (first fit increasing area) is used in the SMART algorithm (Scheduling to Minimize Average Response Time) [91, 76].

- by given job weights: Jobs may come with weights which are used for sorting. Job weights consist of user or system given weights or a combination of both. For example: all jobs receive default weights of one and only very important jobs receive higher weights, i. e. they are scheduled prior to other jobs.

- by the Smith ratio: The Smith ratio of a job is defined by $\frac{weight}{area}$ and is used in the PSRS (Preemptive Smith Ratio Scheduling) algorithm [75].

- by many others: e. g. number of requested resources, current slowdown, ...

[2]The duration or run time of a job is also known as 'height' or 'length', while the number of requested resources is usually called 'width'

In the scheduling process jobs are taken out of the ordered data structure for either a direct start in queuing systems or for placing the job in a full schedule (planning system):

- front: The first job in the data structure is always processed. Most scheduling policies use this approach as only with this a sorting policy makes sense. In queuing systems jobs might have to wait until enough resources are available. Planning systems also process the front of the data structure while placing requests as soon as possible. FCFS, SJF, and LJF use this approach.
- first fit: The data structure is traversed from the beginning and always the first job is taken, that matches the search constraints, i. e. requests equal or less resources than currently free.
- best fit: All jobs are tested to see whether they can be scheduled. According to a quality criterion the best suited job is chosen. Commonly the job which leaves the least resources idle in order to increase the utilization is chosen. Of course this approach is more compute intensive as the complete data structure is traversed and tested. If more than one job is best suited an additional rule is required, e. g. always take the first, the longest/shortest job, or the job with the most weight.
- next fit: The SMART algorithm uses this approach in a special case (NFIW) [91, 76].

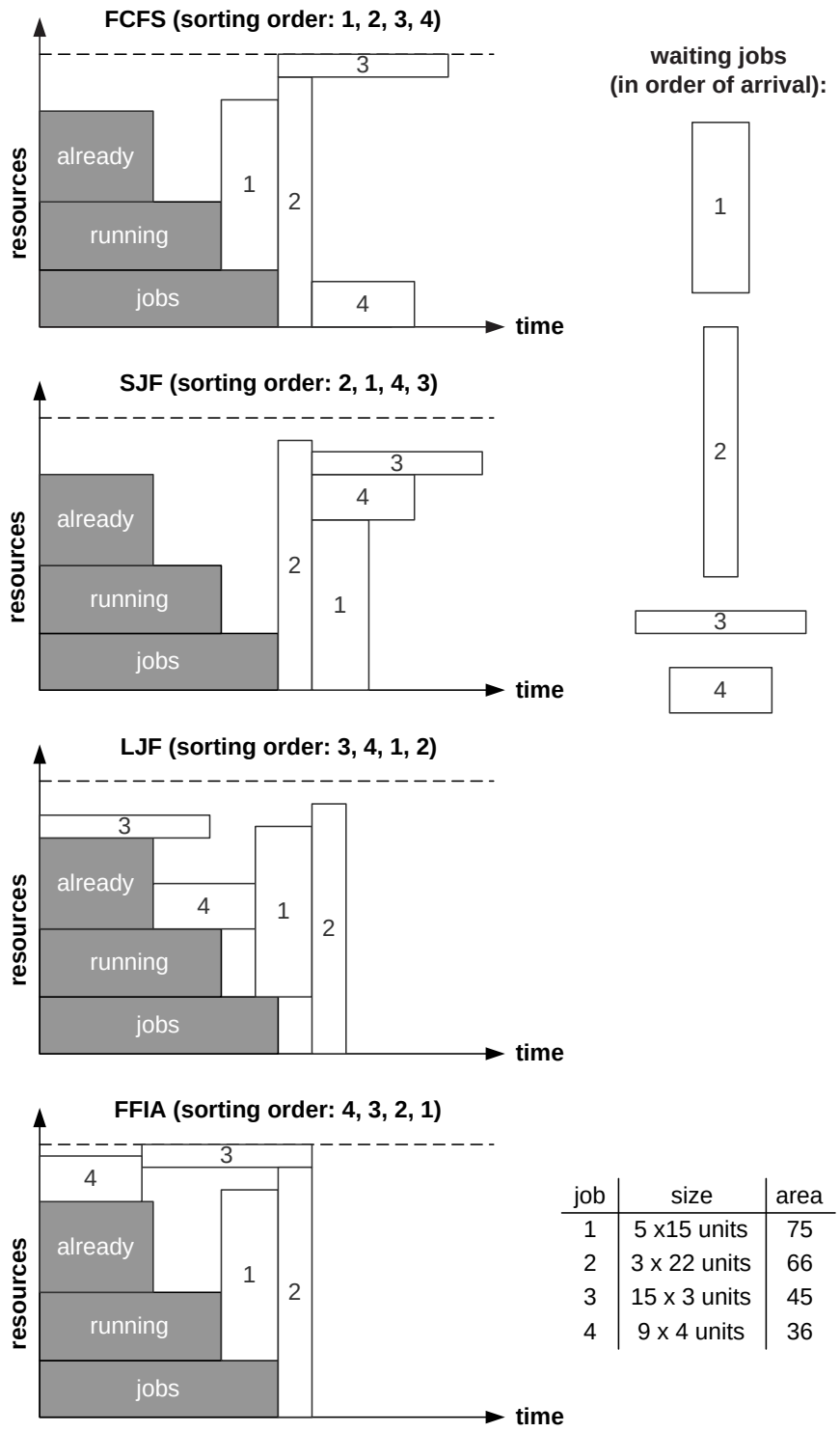
In general, all combinations are possible but only a few are applicable in practice. Figure 3.2 shows example schedules for FCFS, SJF, LJF, and FFIA. Sorting requests in any order while using first or best fit is not necessary, as the best job is always chosen regardless of its position in the sorted structure. However, a sorting policy could be used to choose one job, if many jobs are equal. Furthermore, best fit comes with the risk of making schedules unfair and opaque for users.

If fairness in common sense has to be met, i. e. the starting order equals the arrival order, only the combination of sorting by increasing arrival time and always processing the front of the job structure can be used. All other combinations do not generate fair schedules. However, such a fair scheduler is not very efficient, as jobs usually have to wait until enough free resources are available. Therefore, basic scheduling policies are extended by backfilling, a method to avoid excessive idleness of resources. Backfilling became standard in modern resource management systems today. If requests are scheduled out of their sorting order by first or best fit, some form of backfilling is carried out.

3.3 Backfilling

As previously described FCFS generates fair and understandable schedules for space-shared systems, but at the expense of fragmentation. Queuing systems plan for the present and utilize free resources with queued requests. The highest prioritized request (i. e. the queue head) is always processed. If not enough resources are available to start the queue head, the system waits until enough resources become available. One solution is to utilize these idle resources in the mean time with less prioritized jobs. If the duration of requests is not known, the highest prioritized request may have to wait longer and probably forever.

A sophisticated solution requires run time estimates. With this information a scheduler computes the time for when enough resources are available for the waiting, highest prioritized



job	size	area
1	5 x15 units	75
2	3 x 22 units	66
3	15 x 3 units	45
4	9 x 4 units	36

Figure 3.2: An example of different scheduling policies and the resulting schedules. No Backfilling has been applied.

request. The scheduler starts only those requests which do not delay the start of the highest prioritized jobs any further. This mechanism is called *backfilling* and was developed for the IBM SP at Argonne National Lab as part of the EASY (Extensible Argonne Scheduling sYstem) project [55]. Later this approach was also integrated in the IBM LoadLeveler scheduler for the IBM SP2 [82].

According to [64] two major versions of backfilling exist:

- conservative backfilling: Assume request w can not be started immediately as not enough resources are free. The earliest possible start time for job w is s. Now the list of waiting requests is scanned and as long as resources are free those requests are started which end before s. This requirement assures that all other waiting requests from the queue are not further delayed compared to pure FCFS.

- EASY: The original idea softens the last restriction. The number of free resources f during the run time of request w is computed. In addition to the conservative version such requests can be backfilled which do not require more than f resources. It is assured that only the waiting request w is not delayed. Other less prioritized requests might be delayed further compared to pure FCFS. Therefore, EASY is also called aggressive backfilling.

Note, in the original algorithm s is called *shadow time* and f *extra nodes*. An example for pure FCFS in contrast to the two backfilling variants is shown in Figure 3.3.

The following statements relate to both backfilling variants. They were extracted from [63, 29]:

- In general, backfilling improves the response time of short jobs together with no starvation for long jobs. However, one could think of examples with awkward jobs that wait forever with EASY, but do not with conservative backfilling.

- With both backfilling variants applied the scheduling process is no longer fair as jobs are executed out-of-order.

- The behavior of EASY is especially unpredictable, which might result in confused users.

- The performance improvement of backfilling does not depend on the number of backfilled jobs, but on which jobs are backfilled.

- More exact run time estimates should lead to an improved performance. However, using run time estimates at the same time means, that jobs are killed when they run longer than estimated. Therefore, users tend to over-estimate the run time. Over-estimation factors[3] are usually large. Simulation results [97] show, that this is not at all bad as unused areas in the schedule grow. Hence, more and larger jobs (requesting more resources) become backfilled. In fact, the simulation results show, that accurate estimates are not necessarily the best. Simply multiplying the user estimates by a constant factor automatically improves the performance [49].

- Backfilling is optional for queuing systems and today all common resource management systems have backfilling included.

[3] The ratio between estimated and actual run time.

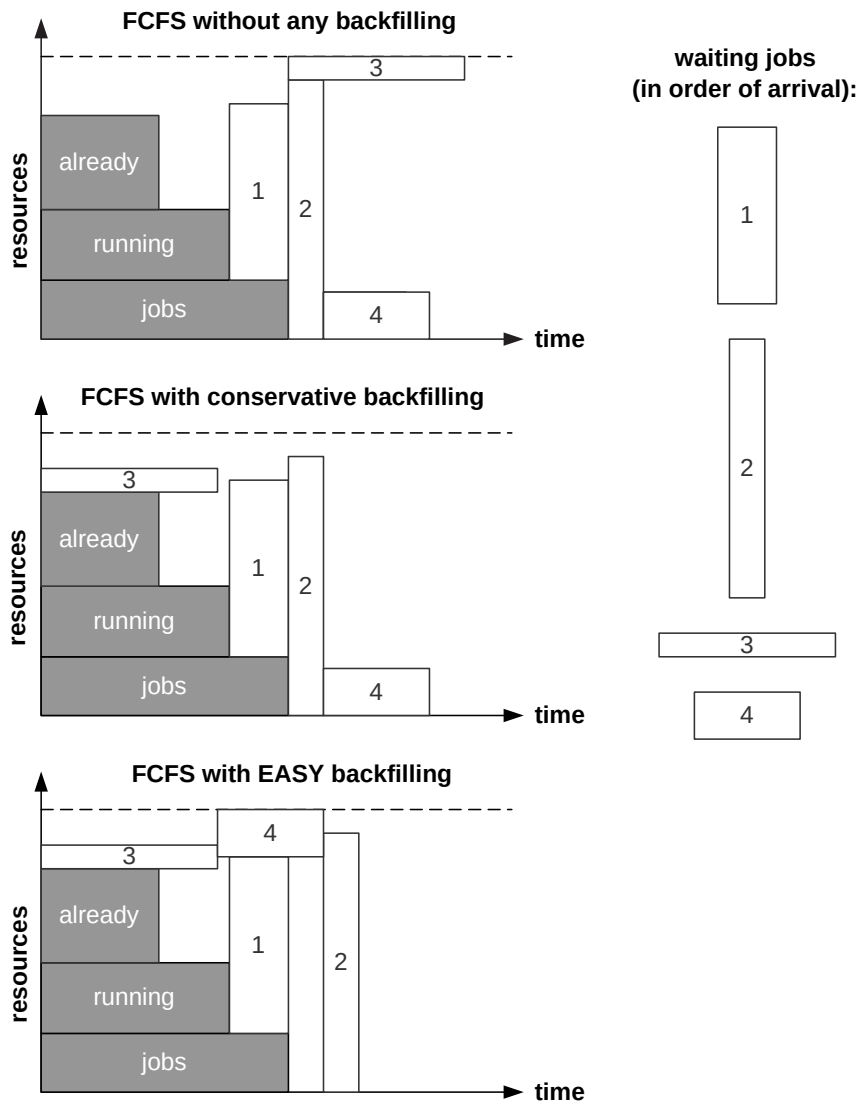


Figure 3.3: Exemplary scheduler FCFS without any backfilling, conservative, and EASY backfilling.

Further improvements of backfilling were developed in recent years. The decision of which job should be backfilled is done by first fit in the original idea. Of course, other more elaborated ways of finding backfill candidates exist. For example, that request is chosen, which best fits and leaves the least resources idle. Or even combinations of jobs are constructed which then utilize the idle resources in a more efficient way.

Another improved variant of backfilling is called slack-based backfilling [90]. The basic idea is to soften the guaranteed start time of the waiting, highest prioritized request. It might be delayed, but the additional waiting time (the *slack*) is limited (e. g. by half the jobs estimated run time). More sophisticated is a combination of the priority and several system parameters in order to reflect the importance of the request. Thus other and potentially more jobs are backfilled. The actual decision process as described in [90] consists of testing all possibilities

of backfilled jobs, computing the cost for each and finally choosing the cheapest solution. Slack-based backfilling preserves the bounded delay advantage of FCFS plus conservative backfilling over EASY.

Obviously some sort of backfilling is implicitly done during the re-planning process in planning systems. Because planning systems place requests as soon as possible in the current schedule, requests might be placed in front of already planned requests. However, these previously placed requests are not delayed (i. e. planned at a later time), as they already have a proposed start time assigned. Note, a separate backfilling routine as in queuing systems does not exist. If backfilling is 'disabled' in a planning system, Var-Time requests are not inserted as soon as possible. The scheduler begins the search for an appropriate start time at the start time of the previously planned job.

3.4 Performance Metrics

We described sorting policies and backfilling variants in the previous two sections. If different scheduling strategies are evaluated, the according schedules have to be compared with each other. This means, that a relation $schedule_a > schedule_b$ has to be defined. In other words: if it is possible to reduce a schedule of many thousands of jobs into a single number, schedules are easily comparable. For this purpose performance metrics are used. Most well-known are utilization, makespan, or average response time. Later, several more are presented. Obviously different target groups exist with opposing objectives. A general classification in *owner-* and *user-centric* performance metrics is done in the following.

The main focus of owner-centric performance metrics lies on an efficient usage of the owned resources. Commonly known for this purpose are *utilization* (how much percent of all resources were actually used on average over a specific time frame?) and *throughput* (how many jobs were processed during a specific time frame?). Others are the *makespan* (when was the last job completed?), the *loss of capacity* (how much percent of all resources were idle, although workload for processing was available?), or the *sum of completion or idle times*. This class of metrics directly refers to the resource usage or idleness respectively. Such metrics are important for the resource owner or investor and blind out job characteristics.

In contrast, user-centric performance metrics refer to the actual job performance. As the performance of the scheduler directly affects the waiting time of jobs, the time difference between submission and start of a job is the key component of all user-centric performance metrics. Most known is the average response time, which is computed from the waiting time plus the execution time (time interval from job submission to job completion). It directly refers to the time when the results of the job are available. The ratio of response time to run time is called *slowdown* and is measured dimensionless. All metrics are defined later in more detail.

However, these two classes often overlap as some performance metrics are interesting for both groups. For example, owners are also interested in user-centric metrics as they measure the quality of service delivered to the users. A high utilization often means large slowdown values as the scheduler mixes the waiting jobs so that the resources are best used. The owner is satisfied now, but surely not the users. If the slowdown values are very bad, users will submit their jobs to other resources. The utilization of these resources will drop due to a lack of jobs. Another example: users mainly focus on the average job slowdown, as it directly affects their work. However, users are also interested in a low utilization as an indication for

short waiting times.

Especially user-centric metrics are often weighted in order to emphasize certain job groups. The width (requested number of resources) of jobs is typically used as a weight. Furthermore, bounded metrics are used to ignore special job classes, e. g. very short jobs (less than 60 seconds). Such jobs often represent errors or live tests submitted by the administrative staff. With bounded performance metrics only important jobs are considered.

Nevertheless the cost of scheduling should not be forgotten. The computation time of the scheduler is important for both queuing and planning systems. A long computation time in queuing systems means, that resources remain idle while the scheduler searches for the next job. In planning systems dead time is not tolerable as this increases the time when either users get to know about their jobs start times or if reservations are accepted or not. Critical dead times are either in the range of some seconds or even up to one or two minutes. In handling reservations requests the dead time should be as short as possible.

Assume the following parameters for scheduled jobs:

- t_i^a is the arrival or submission time of job i
- t_i^s is the start time of job i
- t_i^e is the end time of job i
- w_i is the width (number of requested/used resources) of job i

From these parameters are computed:

- $l_i = t_i^e - t_i^s$ is the length (run time, duration) of job i
- $t_i^w = t_i^s - t_i^a$ is the waiting time of job i
- $t_i^r = t_i^w + l_i$ is the response time of job i
- $s_i = \frac{t_i^r}{l_i} = 1 + \frac{t_i^w}{l_i}$ is the slowdown of job i
- $a_i = w_i \cdot l_i$ is the area of job i

Note, the slowdown is always greater than one. A problem with the slowdown metrics is, that the importance of very short requests is overemphasized. For example, a request that runs for 0.5 s is delayed for 10 minutes and therefore suffers a slowdown of 1201. A request with the same waiting time but a length of 20 seconds has a slowdown of only 31. For the user both waiting times are surely unacceptable, but as the run time stands in the denominator the short job has a high weight on average. Therefore, the bounded slowdown is used, where the run time of very short jobs is limited to a threshold of e. g. 60 seconds:

- $s_i^{60} = \frac{\max(t_i^r, 60)}{\max(l_i, 60)}$ is the slowdown bounded by 60 seconds. Note, the bounded slowdown is also written as $max(\frac{t_i^r}{\max(l_i, 60)}, 1)$ in [18, 19] and Keleher and Perkovic defined it as $1 + \frac{t_i^w}{\max(l_i, 60)}$ [49, 71].

In [97] Zotkin and Keleher defined a per-processor slowdown as jobs that do the same amount of work with the same response time may suffer different slowdowns due to their different shape. A simple example clarifies this: a job with 100 seconds requests only one processor

and is therefore started immediately. It has a slowdown of 1 (zero waiting time). Another job containing the same amount of work (e. g. requesting 10 processors for 10 seconds) is delayed for 90 seconds. Its response time is also 100 seconds, but the slowdown is 10. The per-processor slowdown considers this by using $1/width$ as a weight:

- $pp\_s_i^{60} = \max(\frac{t_i^r}{w_i \cdot \max(l_i, 60)}, 1)$

The authors understand the per-processor slowdown as a further normalization of the slowdown metrics. In the above example both requests have a per-processor slowdown of 1. A user is punished who makes the effort of parallelizing an application on the other hand. Therefore, a scheduler should not treat a delayed sequential and parallel request the same way.

Further to this, user-centric performance metrics (m denotes the total number of requests which are considered for the computation of the metric):

- the average waiting time:

$$AWT = \frac{1}{m} \cdot \sum_{i=1}^{m} t_i^w \tag{3.1}$$

- the average response time:

$$ART = \frac{1}{m} \cdot \sum_{i=1}^{m} t_i^r \tag{3.2}$$

- the average response time weighted by job width:

$$ARTwW = \frac{\sum_{i=1}^{m} w_i \cdot t_i^r}{\sum_{i=1}^{m} w_i} \tag{3.3}$$

- the average slowdown weighted by job area:

$$SLDwA = \frac{\sum_{i=1}^{m} a_i \cdot s_i}{\sum_{i=1}^{m} a_i} \tag{3.4}$$

The unbounded SLDwA uses the area of requests as a weight which circumvents the problem described above: Assume both request only one processor. The 0.5 s long request has a weighted slowdown of $1201 \cdot 0.5 = 600.5$ and the 20 second job $31 \cdot 20 = 620$. They have a similar weighted slowdown which reflects that both waiting times are unacceptable for the user.

Furthermore, owner-centric performance metrics are (N denotes the total number of available resources):

- the makespan:

$$\max_{i=1,\ldots,m} t_i^e \tag{3.5}$$

- the utilization:

$$UTIL = \frac{\sum_{i=1}^{m} w_i \cdot l_i}{N \cdot (\max_{i=1,\ldots,m} t_i^e - \min_{i=1,\ldots,m} t_i^a)} \tag{3.6}$$

The denominator of the utilization is the total area of the schedule which is computed from the number of resources (e. g. processors or nodes) and the length of the schedule, i. e. from the first submission to the last end. This is the same as with the makespan, if the first request is submitted at time zero. The makespan is seldom used as a metrics, because the utilization of a machine is more convincing. The makespan contains no information about the usage of the resources. In on-line scenarios the makespan is often influenced by the last submission. Furthermore, there is no need to reduce the makespan in on-line scenarios, as there is no end of the scheduling process by definition.

Additionally, the performance of a scheduler may be measured with the loss of capacity (LOC) [95, 96] from the system's perspective. This means, that requests wait for execution and available resources, but due to fragmentation it is not possible to start any waiting request. To define the loss of capacity, *schedule events* which occur each time a new request is submitted or a running job ends are introduced. With m requests $2m$ schedule events exist at times τ_k for $k = 1, \ldots, 2m$. f_k denotes the number of free resources between the scheduling events k and $k+1$. δ_k is one, if requests wait for execution and zero otherwise. With that:

- the loss of capacity:

$$LOC = \frac{\sum_{k=1}^{2m-1} f_i \cdot (\tau_{k+1} - \tau_k) \cdot \delta_k}{N \cdot (\max_{i=1,\ldots,m} t_i^e - \min_{i=1,\ldots,m} t_i^a)} \tag{3.7}$$

A system is in a *saturated state* if the number of waiting jobs increases and the scheduler is not able to achieve a higher utilization. As a result a scheduler with backfilling can search within a larger set of jobs for backfill candidates. If the utilization is small enough, slots in the scheduler are bigger and most likely more and better suited jobs are backfilled. Even in the saturated state fragmentation exists, but the holes are that small that no requests are backfilled in these holes. Therefore, requests wait longer for their execution and thus user-centric performance metrics increase dramatically. If a system reaches the saturated state (δ_k is almost always 1) the following equation holds: $1 - LOC = UTIL$.

While evaluating job scheduling strategies with a fixed set of jobs, it is observed, that the ranking of strategies depends on the used metrics. Some strategies try to achieve good slowdown values while others optimize the utilization of the system. Moreover, scheduling strategies are designed especially for optimizing one of the metrics e. g. the SMART algorithm [91, 76]. However, when comparing the ranking of scheduling strategies for different performance metrics it is observed, that the average response time weighted by job width (ARTwW) and the average slowdown weighted by job area (SLDwA) behave similarly. In fact it is possible to convert one metrics into the other by means of a constant factor:

$$\begin{aligned}
\frac{SLDwA}{ARTwW} &= \frac{\sum_{i=1}^{m} a_i \cdot s_i}{\sum_{i=1}^{m} a_i} \cdot \frac{\sum_{i=1}^{m} w_i}{\sum_{i=1}^{m} w_i \cdot t_i^r} \\
&= \frac{\sum_{i=1}^{m} (l_i \cdot w_i) \cdot \frac{t_i^r}{l_i}}{\sum_{i=1}^{m} a_i} \cdot \frac{\sum_{i=1}^{m} w_i}{\sum_{i=1}^{m} w_i \cdot t_i^r} \\
&= \frac{\sum_{i=1}^{m} w_i}{\sum_{i=1}^{m} a_i}
\end{aligned} \tag{3.8}$$

The ratio of the average width and average area of all jobs is constant for a given set of requests. As these values are independent from the waiting time of requests, both metrics generate the same ranking for different scheduling strategies and job inputs. The ARTwW is more difficult to understand for users, as the response time contains the duration. As the SLDwA is measured without a dimension it is more user-friendly: If the system has an average slowdown of e. g. three, users have to wait three times the length of the request unless results are available. The SLDwA is used in this work.

All presented metrics are based on job properties and are therefore easy to compute. Sometimes more sophisticated metrics are helpful, e. g. the understandability or readability of a schedule. However, they are difficult or even impossible to measure from job properties as the users knowledge of scheduling is taken into account. For planning systems this is still neglected. As the current schedule is visible to the users, they want to understand the actions of the scheduler. For example: a slot in the schedule exists and a user submits a job that fits perfectly into this slot. If it is placed in this slot the user is satisfied and the scheduling action was understandable. However, if the scheduling strategy is too complex (e. g. the weight of a job is computed from job properties, user weights and current system properties) and the job is not placed in that slot, the user is not satisfied. If such cases are frequent users will probably submit their jobs to different machines. Sorting the list of variable jobs by increasing arrival time leads to more understandable schedules. If sorting is done by increasing or decreasing run time the readability of the schedule changes significantly as one very short or long job might change the complete schedule.

However, fairness helps with the problem of understanding or reading schedules. In [94, 79] a scheduling strategy is named λ-fair if all jobs submitted after a job i can not increase the response time of i by more than a factor of λ. The term is defined in the theoretical analysis of preemptive scheduling algorithms, hence the factor is related to optimal solutions. Another approach is to define the term fairness with respect to the start times in a pure FCFS schedule (without any backfilling). In [88] the term fairness is generally less defined. As schedulers may favor jobs with certain properties, the fairness is measured on a case-by-case basis. The MASC (Maximum Allowable Skipping Count) defines the maximum number of jobs that can pass a waiting job. As the MASC is given specifically for each job, jobs are not indefinitely

delayed.

3.5 Workloads

An evaluation of job scheduling strategies contains three parts: the scheduling strategy, the performance metrics and a set of jobs as input for the scheduler. The job input is also called *workload* for the job scheduler. In this context a job is defined by:

1. the time of submission
2. the number of requested resources (= width)
3. the estimated duration (= length)
4. and actual duration

These job properties are necessary for the scheduler to work with the job. As we model a planning system the run time estimate is mandatory. And as a simulation environment is used to perform the evaluation, information about the actual duration of jobs also needs to be specified in the job set. Several options exist for generating the four properties:

Random Generated A random number generator (usually in the range 0 to 1) is used to obtain the four values. However, random functions are evenly distributed in most programming languages. For the job width this means: having 32 resources in total and jobs are of arbitrary size, every width between 1 and 32 occurs with a probability of $\frac{1}{32}$. However, this is not the case in the real world. Smaller or even sequential jobs are found far more often than large jobs requesting e. g. more than 50% of all available resources [24, 41, 57].

Worst Case For each scheduling strategy a set of jobs can be created, which results in a worst case behavior of the scheduler. Obviously each scheduling strategy has its own set of worst case jobs. A simple worst case example for pure FCFS is taken from [75]: the current waiting queue of a pure FCFS scheduler (without any backfilling) consists of a sequence of jobs. Odd jobs are short (e. g. 10 seconds) and request the full machine. Even jobs are long (e. g. 10 minutes), but request only one $\frac{1}{10}$ of the machines resources. Both job types are submitted alternatingly.

The resulting pure FCFS schedule has a ladder like appearance as jobs are started in the same order as they are submitted. It is impossible to start two even jobs at the same time as an odd job is always submitted in between. However, if backfilling is applied 10 even jobs are started at the same time, because 9 even jobs are able to overtake the next odd job without delaying it. This example shows how unrealistic worst case workloads might be. If a scheduling strategy is used in a real environment, its worst case performance can be neglected as such worst case scenarios are very unlikely to occur. More important is the average performance with realistic workloads.

Job Model In order to generate more realistic workloads, job models use non-uniform distributions for the width, length and interarrival time. The parameters of the distributions are taken from a statistical analysis of different workload logs. Therefore, it is possible to

generate new job sets with similar statistical properties. At the same time these properties can be changed, e. g. for generating a higher system load.

According to [6] job models fall into two categories: those modelling rigid jobs and flexible jobs. Models for rigid jobs generate jobs with a given time of submission, number of resources, and run time. The two values describing the size of the jobs (number of resources and run time) are fixed and the scheduler can not change them. In contrast, a flexible job model describes how the application performs at different levels of parallelism. For example, this is described by data about the total computation and by specifying a speedup function. The scheduler is now able to choose a job size that fits best in the current schedule.

The following models are available in Feitelson's Parallel Workload Archive [89] for download (Table 3.3 shows a comprehensive overview):

[4] Calzarossa and Serazzi, 1985: only the arrival process of interactive jobs in a multi-user UNIX environment is modelled

[54] Leland and Ott, 1986: only the actual run time of processes in an interactive UNIX environment is modelled

[81] Sevcik, 1994: includes the speedup characteristics of parallel applications including imbalance, inherited serial work and parallel overhead

[17] Feitelson, 1996: is based on characteristics of rigid jobs from six traces; includes distribution of job width, correlation between run time and parallelism (width), and repeated runs of the same job

[10] Downey, 1997: is based on observations from the SDSC Paragon log and CTC SP2 log; includes moldable jobs where the width is chosen by the scheduler

[47] Jann et al, 1997: a detailed model of parts of the CTC SP2 log; handles rigid jobs with distribution of run times and interarrival times

[27] Feitelson and Rudolph, 1998: is a framework to create models which focus on the connections between application behavior and scheduling

[59] Lublin, 1999: a detailed model for rigid jobs, which includes arrival pattern with a daily cycle, correlation between run times and requested nodes, distinction between interactive and batch jobs

[7] Cirne and Berman, 2001: generates moldable jobs and is based on the model of Downey, 1997

Unfortunately, all job models do not contain any information about the estimated run time. However, run time estimates are mandatory for this work.

Trace Today most parallel supercomputer installations and their resource management systems generate traces. The data contained is used for accounting purposes or the evaluation of the machine usage. The data is logged in different levels of detail. Core data for every job is always stored: the submit, start and end time, the number of used resources, and probably a user ID. Some systems also store pre-scheduling data, e. g. run times estimates or memory usage. Other systems log internal scheduler information about the prioritization parameters

reference	jobs	work	parallelism	run time	speedups	arrivals
[4]	Unix	no	no	no	no	yes
[54]	Unix	yes	no	yes	no	no
[81]	moldable	no	no	yes	yes	no
[17]	rigid	no	yes	yes	no	partial
[10]	moldable	yes	yes	yes	yes	partial
[47]	rigid	no	partial	yes	no	yes
[27]	varied	yes	partial	partial	implied	no
[59]	rigid	no	yes	yes	no	yes
[7]	moldable	yes	yes	yes	yes	yes

Table 3.3: A comprehensive overview on available job models in the Parallel Workload Archive [89].

or queues. Many traces exist and are available on the internet. Two sources are the Parallel Workload Archive [89] and the Maui Scheduler Workload Trace Repository maintained by supercluster.org [60]. Furthermore, traces from the *hpcLine* [43] cluster installed at the Paderborn Center for Parallel Computing (PC^2) exist. Details on the available traces are given in the following.

Again, for this work it is important that run time estimates are logged in the trace as this information is required for the backfilling.

The advantage of traces over random generated or worst case job set is the realism of the logged job properties. For an evaluation of job scheduling strategies many traces with different characteristics have to be used in order to achieve a broad spectrum of results. A scheduling strategy might be good for one type of job, but might generate poor results for other job characteristics.

In the following some information about listed traces in the two archives and the related machines is provided.

Parallel Workload Archive

- **NASA Ames**, system: 128-node iPSC/860 hypercube, duration: fourth quarter of 1993, jobs: 42,050 total, 14,794 user:
  This is the first workload trace which was analyzed in detail [24]. For each job the number of nodes, the actual run time in seconds (not node-seconds), and the local start date and time are available. User and job names are substituted by irrelevant text strings. However, classes of users are retained, e. g. root, development, support, and user. Because of the system architecture the number of requested nodes is limited to powers of two. The NQS queue configuration is found in Table 3.1.

- **SDSC**[4], system: 416-node Intel Paragon, duration: all of 1995 and 1996, jobs: 76,872 in 1995 and 38,723 in 1996:
  In fact only 352 nodes are usable by parallel jobs via NQS, 48 nodes are for interactive access and the remaining nodes are for the service (e. g. login) and I/O. The compute partition itself is divided into 64 nodes for short jobs and 288 for long jobs. Three classes of run time limits exist: short (1 hour), medium (4 hours), and long (12 hours).

[4] San-Diego Supercomputer Center

Each job in the trace is specified by the number of nodes, the submit time, the start time (so the waiting time can be computed), the end time (so the actual run time can be computed), the used CPU time, the used NQS queue and their limits, and the users name.

- **CTC**[5], system: 512-node IBM SP2, duration: July 1996 - May 1997, jobs: 79,302:
  As with all other SP2 traces in the archive the machine is scheduled by LoadLeveler. Therefore, run time estimates are available. Additionally, on this machine EASY is attached to LoadLeveler. This machine is heterogeneous in the sense that the nodes are not identical. 430 nodes are available in the batch partition and most of them (352) are 'thin' nodes each equipped with 128 MB. For each logged job a variety of information is available. The most important for scheduling are: number of nodes, submission, start and completion time, and most important is the maximum run time. This time is entered by users in advance and is used by the EASY scheduler for backfilling.

- **KTH**[6], system: 100-node IBM SP2, duration: October 1996 - August 1997, jobs: 28,490:
  Basically, the same job properties as on the CTC SP2 are logged for this machine. As EASY is used run time estimates are available, too. This machine is also heterogeneous as different types of 'thin' and 'wide' nodes are built in. Basically all nodes are available for batch processing. However, from time to time some nodes were put aside for special/interactive usage.

- **LANL**[7], system: 1024-node Connection Machine CM-5 from Thinking Machines, duration: October 1994 - September 1996, jobs: 201,387:
  The trace of this machine is different to all others as the machine works with a fixed partitioning. Therefore, the number of nodes for a job is limited to powers of two and the smallest allocatable partition is 32 nodes. Except the SP2 traces from CTC, KTH, and SDSC this is the only trace that also comes with run time estimates.

- **LLNL**[8], system: 256-node Cray T3D, duration: June - September 1996, jobs: 21,323 (represented by 40,591 rolls):
  The scheduler of this machine is unique as it supports gang-scheduling[9] [21, 78, 96]. Therefore, jobs are occasionally preempted and swapped out in favor of other jobs. This is also called 'roll-in' and 'roll-out', and the trace contains information about every 'roll'-activity. Basic job properties are traced: number of nodes, the start time and date (when the execution actually begins, after job initiation or 'roll-in'), the run time and some others. The queues of the scheduler have different resource and duration limits. The maxima are 64 processors for 40 hours or 256 processors for only 4 hours. More information on the gang scheduler of this machine can be found in [21].

- **SDSC**, system: 128-node IBM SP2, duration: May 1998 - April 2000, jobs: 67,667:
  The trace comes from the NPACI[10] JOBLOG repository [67] and provides a rich set of

[5] Cornell Theory Center
[6] Swedish Royal Institute of Technology
[7] Los Alamos National Lab
[8] Lawrence Livermore National Lab
[9] An intuitive extension of time-sharing on multiprocessor systems. All parallel instances of an application are suspended at the same time.
[10] National Partnership for Advanced Computational Infrastructure

information. Unfortunately, no information is found as to whether EASY is used for scheduling or not. The fact that run time estimates are available like for all other SP2 machines indicates the use of EASY.

- **LANL**, system: cluster of 16 Origin 2000 machines with 128 processors each (2048 total), duration: Dec 1999 - Apr 2000, jobs: 122,233:
  This machine is the largest in the archive, but only four months of traces are available. LSF[11] [58] is used as the resource management system. Unfortunately, run time estimates are not provided.

Maui Scheduler Workload Trace Repository Much less information is available for these traces and the related machines. Common for all is the usage of the Maui Scheduler [46, 61] and as it falls in the class of planning systems run time estimates are available. Unfortunately, most of the repository itself is in a bad condition as two traces (ANL, NCSA) and information about machine and scheduler configurations is not accessible. In general, the Maui scheduler is combined with the commercial resource management system PBS pro [69]. This combination is very popular for scheduling and managing large Linux Clusters.

- **CHPC**[12], system: 266 processor Linux Cluster, duration: March 2000 - March 2001, jobs: 20,000
- **MHPCC**[13], system: 224 processor IBM SP2, duration: March 1998 - April 1998, jobs: 4100:
  In contrast to the other SP2 traces from the Parallel Workloads Archive, LoadLeveler is not used for scheduling this IBM SP2. The scheduler component was replaced by the Maui scheduler.
- **ANL**[14], system: 256 node, dual processor Linux Cluster, duration: May 2001 - n/a, jobs: n/a
- **OSC**[15], system: 32 node, quad processor and 25 node, dual processor Linux Cluster, duration: January 2000 - November 2001, jobs: 80000:
  It is not clear from which of the two clusters the traces were generated.
- **NCSA**[16], system: 512 node, dual processor Linux Cluster, duration: October 2001 - n/a, jobs: n/a

The PC^2 Trace The Fujitsu-Siemens *hpcLine* system [43] is a typical HPC cluster which is a combination of a fast interconnect and powerful computing nodes. The SCI (Scalable Coherent Interface [80]) interconnect comes with a high bandwidth (80 - 280 MByte/s) and low latency (4 - 10 μs for ping-pong/2). The nodes hardware is taken from reliable server products and each node is equipped with two Intel Pentium III 850 MHz processors and 512 MB main memory. Altogether 96 nodes are installed. The PC^2 has developed its own

[11] Load Sharing Facility
[12] Center for High Performance Computing, University of Utah
[13] Maui High Performance Computing Center
[14] Argonne National Lab
[15] Ohio Supercomputing Center
[16] National Center for Supercomputing Applications, University of Illinois at Urbana- Champaign

resource management system which is called CCS (Computing Center Software) [3, 51]. It is a planning system and not a queuing system. On the *hpcLine* node scheduling is applied, which means, that the smallest entity that can be requested by users is a node, i. e. two processors.

The available data covers approximately two years of machine usage. From January to December in 2001 and 2002. The data is not joined to one trace as the machine is shut down for maintenance between Christmas and the first days in January (cf. Table 3.5). Hence, the PC2-2001 trace contains 35,094 jobs and the PC2-2002 32,212 respectively.

3.6 Analysis of Traces

As already stated above, run time estimates are needed by queuing systems with backfilling applied and by all planning systems run time estimates are needed. Therefore, the IBM SP2 traces from CTC, KTH, SDSC, and LANL are chosen from the Parallel Workloads Archive. Unfortunately, only the CHPC and MHPCC are downloadable from the Maui Scheduler Workload Trace Repository. Finally this makes eight traces that are used in this work: CTC, KTH, SDSC, LANL, PC2-2001, PC2-2002, CHPC, and MHPCC.

The following tables and figures contain detailed information on the eight traces and their differences. Table 3.4 shows basic job properties which include data about requested resources, estimated and actual run times. The *interarrival time* is the time distance between two consecutive job submissions and gives information on the arrival process. For each of the four job properties the minimum, average, and maximum values are printed. As stated earlier the LANL users can only request power of two resources starting at 32 (i. e. requested resources have values of 32, 64, 128, 256, 512, and 1024). On all other machines users are able to request an arbitrary number of resources.

On some systems (CTC, CHPC, MHPCC) users either did not or were not allowed to request all available resources of the machine. Run time data and the *over-estimation factor* (ratio of estimated to actual run time) in the bottom table of Table 3.4 show, that users estimate their jobs about two to three times longer than the jobs actually run. Comparing the values for available resources and maximum estimated run time shows, that for the four SP2 machines these values are coupled similarly to a speedup function: the more total resources the machine has, the smaller the maximum requested run time is.

The high maximum interarrival time of 1.8 million seconds results from a 20 day period between the 27th of May and 17th of June 2002. During this time the machine was dedicated to a single user and resource management was not done. Therefore, no trace data is available, although a utilization of 100% is seen in Figure 2.2 on page 19.

Table 3.5 shows the exact dates of the first and last job submission. For the traces taken from the Parallel Workloads Archive job submissions are logged in seconds from the beginning of the trace. With the exact information about the date and time of the first job submission it is possible to convert all subsequent events to a real date and time. Therefore, it is possible to observe the scheduler performance on a monthly or even daily basis.

In order to get a feeling for the trace and how much the machine was utilized at the time the trace was generated, simulations with a FCFS + EASY backfill scheduler were done. Table 3.6 shows the results. For some traces less jobs are simulated and scheduled as available, because the simulation environment had to reject jobs which came with a zero actual run time. After the simulation ended and while computing the slowdown, jobs with zero actual run time crashed the analyzer as the run time was positioned as the denominator. Therefore, such jobs

trace	requested resources min	requested resources avg.	requested resources max	available resources on machine
CTC	1	10.72	336	430
KTH	1	7.66	100	100
LANL	32	104.95	1,024	1,024
SDSC	1	10.54	128	128
PC2-2001	1	6.34	96	96
PC2-2002	1	8.14	96	96
CHPC	1	5.80	100	266
MHPCC	1	8.08	180	224

trace	estimated run time [sec.] min	estimated run time [sec.] avg.	estimated run time [sec.] max	actual run time [sec.] min	actual run time [sec.] avg.	actual run time [sec.] max	average overest. factor	interarrival time [sec.] min	interarrival time [sec.] avg.	interarrival time [sec.] max
CTC	0	24,324	64,800	0	10,958	64,800	2.220	0	369	164,472
KTH	60	13,678	216,000	0	8,858	216,000	1.544	0	1,031	327,952
LANL	1	3,683	30,000	1	1,659	25,200	2.220	0	509	201,006
SDSC	2	14,344	172,800	0	6,077	172,800	2.360	0	934	79,503
PC2-2001	1	11,717	1,209,600	1	4,346	604,800	2.696	0	870	313,861
PC2-2002	1	33,942	604,800	1	6,310	604,800	5.379	0	944	1,835,392
CHPC	5	168,024	10,800,000	0	47,838	2,275,790	3.512	0	1,750	170,365
MHPCC	30	20,079	129,600	8	6,246	116,128	3.215	0	740	94,979

Table 3.4: Basic properties of the used traces (86,400 seconds = 1 day).

trace	last job submission [sec.]	first job submit	last job submit
CTC	29,299,062	Wed, 26 Jun 96, 16:06:00	Sat, 31 May 97, 22:11:26
KTH	29,363,618	Mon, 23 Sep 96, 12:00:31	Fri, 29 Aug 97, 08:55:01
LANL	62,288,828	Tue, 04 Oct 94, 07:01:12	Tue, 24 Sep 96, 06:44:55
SDSC	63,183,029	Wed, 29 Apr 98, 16:05:28	Sun, 30 Apr 00, 04:08:32
PC2-2001	30,538,430	Tue, 02 Jan 01, 08:15:43	Fri, 21 Dec 01, 22:26:20
PC2-2002	30,439,963	Mon, 07 Jan 02, 11:35:11	Wed, 25 Dec 02, 19:07:57
CHPC	34,274,155	Sat, 08 Apr 00, 04:41:54	Wed, 09 May 01, 21:17:49
MHPCC	2,417,836	Sat, 28 Feb 98, 20:58:41	Sat, 28 Mar 98, 19:34:37

Table 3.5: Length, start and end dates of used traces.

are left out and not scheduled. This corresponds with the column 'min. actual run time' in Table 3.4 and zero entries.

The workload for the scheduler is also often called *backlog*. There is no common definition for the backlog, however the average queue length can be used. At the end of a simulated schedule a potential backlog is obvious. If the backlog is low or does not exist, the last submitted job usually doesn't have to wait until it is executed. By comparing the *virtual end* and the actual makespan of a schedule the size of the backlog at the end of the schedule can be estimated. The virtual end denotes the time at which the last submitted job would end if it is directly started after submission. It is computed from the arrival time plus actual run time of the last submitted job. However, this only represents the backlog towards the end of the schedule. The average queue length (6th column) new jobs see at their submission represents the backlog during the total length of the trace. Additionally, how many newly submitted jobs had to wait for execution (last column in Table 3.6) is counted. The four

SP2 traces (CTC, KTH, LANL, SDSC) come with considerable large numbers for both, as these machines are real production machines at large computing centers. All other remaining traces do not show such large numbers. All remaining traces are derived from PC-based cluster systems.

In the bottom part of Table 3.6 several job performance metrics are presented. The average response time, measured in seconds, is difficult to handle, unless the average length of all scheduled jobs is not printed (the response time is computed from its waiting time and run time). This confirms the choice for the slowdown metrics, as it is measured without any dimension. The very small waiting times (even without comparing them to the job length) for PC2-2001, PC2-2002, CHPC, and MHPCC are also reflected by SLDwA values very close to one. Comparing the slowdown values near one with the achieved utilization shows their dependence: if the utilization is low (e. g. less than 50%) the schedule is empty, the backlog is small, jobs wait only for a short amount of time and so in the end they achieve good slowdowns. Large slowdowns represent a packed schedule where jobs have to wait for execution.

trace	total jobs	scheduled jobs	makespan [s]	virtual end [s]	avg. queue length	waiting jobs > 1
CTC	79,302	79,279	29,306,682	29,301,671	49.56	44,126
KTH	28,487	28,479	29,363,626	29,363,626	16.37	20,801
LANL	122,305	122,305	62,292,428	62,292,428	24.24	55,588
SDSC	67,631	67,620	63,213,412	63,189,634	40.79	54,496
PC2-2001	35,094	35,094	30,552,830	30,552,830	8.41	4,154
PC2-2002	32,212	32,212	30,439,966	30,439,966	91.37	3,471
CHPC	19,583	19,270	34,274,160	34,274,160	4.34	646
MHPCC	3,273	3,267	3,102,871	3,102,871	9.29	146

trace	AWT [s]	ARTwW [s]	avg. job length [s]	SLDwA	UTIL [%]
CTC	4,829	19,917	10,961	2.0455	65.70
KTH	7,989	28,680	8,860	3.1015	68.72
LANL	1,340	4,642	1,658	1.6801	55.61
SDSC	21,064	63,918	6,077	6.8261	82.48
PC2-2001	216	6,586	4,345	1.0846	46.07
PC2-2002	279	5,744	6,310	1.0798	47.64
CHPC	3	30,929	48,615	1.0004	38.39
MHPCC	298	7,245	6,246	1.2362	28.55

Table 3.6: Performance scheduled by FCFS + EASY backfill. The virtual end is the arrival time of the last submitted job plus its actual run time. 'jobs waiting > 1' denotes the number of job submissions at which previously submitted jobs are still waiting for execution.

Nevertheless the downloaded traces could not be used straightaway. While converting the traces to the MuPSiE job format, jobs are not converted:

- if the submission time is shorter than the submission time of the last job. Thereby, negative time steps are prevented.
- if the number of requested resources is zero, negative or larger than the number of available resources on the machine.
- if the actual or estimated run time is negative.

Furthermore, the actual run time is cut off at the estimated run time as jobs are not allowed to run longer than estimated. It does not matter, whether this is done whilst converting jobs or whilst reading jobs in the simulation. Additionally, about 40 jobs were left out at the beginning of the original MHPCC trace due to very large interarrival times. These 40 jobs were submitted during the first 20% of the original trace and they generated no noticeable utilization. If these jobs had been considered in the evaluation process, the average utilizations would have been incorrect. User centric performance metrics like the SLDwA would not change dramatically as the slowdown is already close to one for the rest of the trace.

The provided additional information for the LANL trace states, that the last job was submitted in September 1996 (Table 3.5). However, two jobs are logged after this last job submission. We deleted these two jobs as their submission dates are in the December time frame with almost two months of interarrival time in between. If these two jobs were to be used for the evaluation, the utilization results would be false with two months of no job submission.

The minimum, maximum and average values as stated above are typically insufficient for describing the job traces in detail. Hence, distribution plots for the arrival process, the estimated, and actual run time are presented in the following. Common to all figures is, that the accumulated number of jobs relative to the total number of jobs is printed on the y-axis. In Figure 3.4 the arrival time relative to the previous arrival time is plotted on the x-axis. Obviously all curves have to start at (0,0) and have to end at (1,1) as 100% of all jobs are submitted with the last job. A direct line from these two points indicates a constant submission pattern. A larger gradient (e. g. SDSC at 0.2) indicates a burst of job submissions. It means, that many jobs are submitted over a relatively short amount of time. The 20 days down time in the PC2-2002 trace are reflected by a horizontal line from 0.4 to 0.5. All in all an irregular arrival process is observable for the four cluster system traces PC2-2001, PC2-2002, CHPC, and MHPCC. The traces from the SP2 machines show a smoother arrival pattern which is only interrupted by a burst in the SDSC trace at the beginning and towards the end in the LANL trace.

Run time distribution plots are used to describe the job run time in more detail. Compared to the arrival process plot the x-axis shows the run time in a logarithmic scale. All traced jobs are categorized according to their estimated or actual run time. The accumulated sizes for each category are printed on the y-axis. Therefore the curves end at the maximum estimated/actual run time from Table 3.4. Figure 3.5 shows, that job length categories with many jobs exist. Examples are: 600 seconds (10 minutes) in the two PC^{2}trace as 10 minutes are the default value for the run time estimate, if nothing is specified by the user. 3,600 seconds (1 hour) for almost all traces (KTH, PC2-2001, PC2-2002, SDSC, and CHPC) and 86,400 seconds (1 day) for CTC. In the PC2-2002 trace almost half of the jobs are estimated to have a run time of 10 minutes. This was induced by a small test script that was continuously (every hour) started by one of the projects through the Globus interface [37]. Most of these jobs were submitted in the last two months of the trace.

The large increase at the beginning of the CTC trace is misleading. Only 6 out of 79,302 jobs come with an estimated run time of zero seconds. Due to the fact that the next run time category is 300 seconds (with 6,802 jobs) and an logarithmic scale is used, a direct line is drawn between one and 300 seconds. The exact start and end points (i. e. their according run times) for each curve correspond with the entries for min and max estimated run time in Table 3.4. All curves have a staircase like shape as users usually choose round and common values (e. g. 10 minutes, 1 hour, 1 day) as run time estimates.

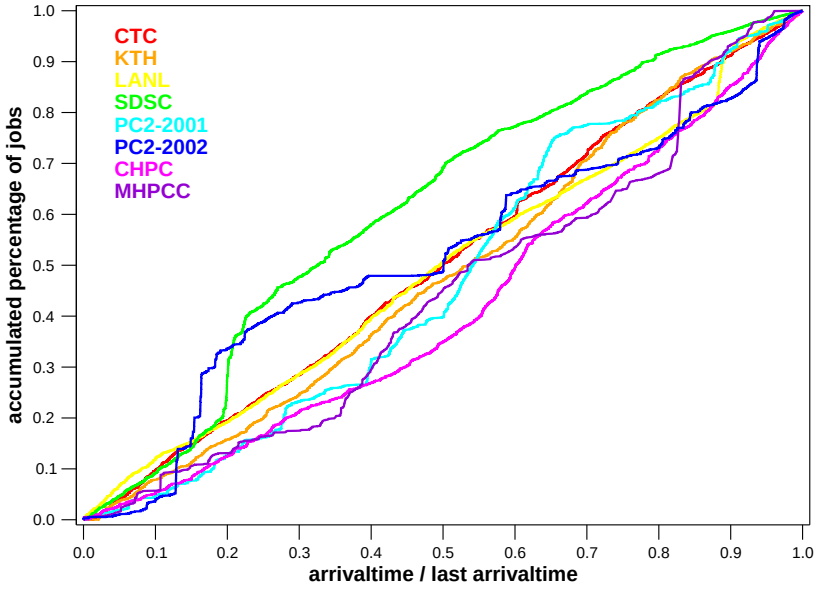


Figure 3.4: The arrival process: relative arrival (each arrival time is divided by the maximum arrival time) on the x-axis, accumulated number of jobs on the y-axis.

In contrast, the curves for the actual run times are smoother as applications typically do not end after common durations, but jumps are still observable in Figure 3.6. This is the result of killed jobs that tried to run longer than estimated. For example in the PC2-2001 trace at 7,200 seconds: 6,849 jobs are estimated to have had a run time of two hours and 5,553 jobs also ended after a two hour duration. These are not necessarily the same jobs, but it seems that they are killed by the resource management system. The curve for PC2-2002 rapidly increases at the start, indicating that many jobs (more than 60%) have a run time of less than one minute. Continuously started test jobs finished their work after some seconds, especially the already named.

Like with the estimated run time, jobs with a zero or one second actual run time exist. Such jobs might represent errors, like a missing or mistyped path, filename or input parameter. It is also observed, that for some traces (CTC, KTH, LANL, SDSC and MHPCC) almost no jobs with actual run times below a certain threshold exist (e.g. 13 seconds for CTC). A conjecture might be that this indicates the fixed time every resource management system needs for preparing and shutting down a partition[17]. In such systems a job start is probably logged when the related resources are taken from the system (and are therefore no longer available for other jobs) and not when the application is actually started on the set of resources.

For some of the evaluations the original traces can not be used, as they either contain too many (CTC, LANL) or not enough (MHPCC) jobs. Too many jobs elongate the run time of the simulation and usually without generating different results. Less jobs are enough, hence we arbitrarily choose an amount of 10,000 jobs. However, these 10,000 jobs are not a subset of

[17] A set of resources assigned to the same job is often called partition.

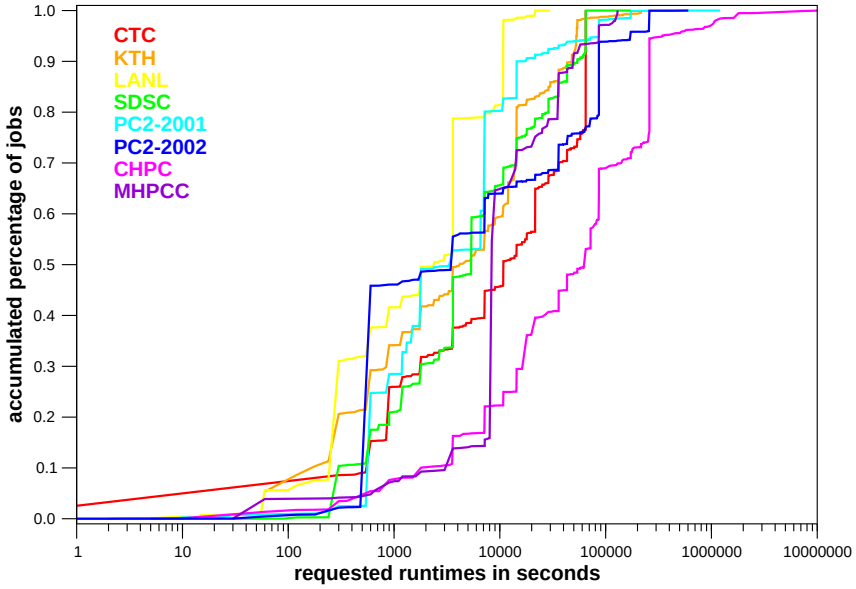


Figure 3.5: Distribution of estimated run times.

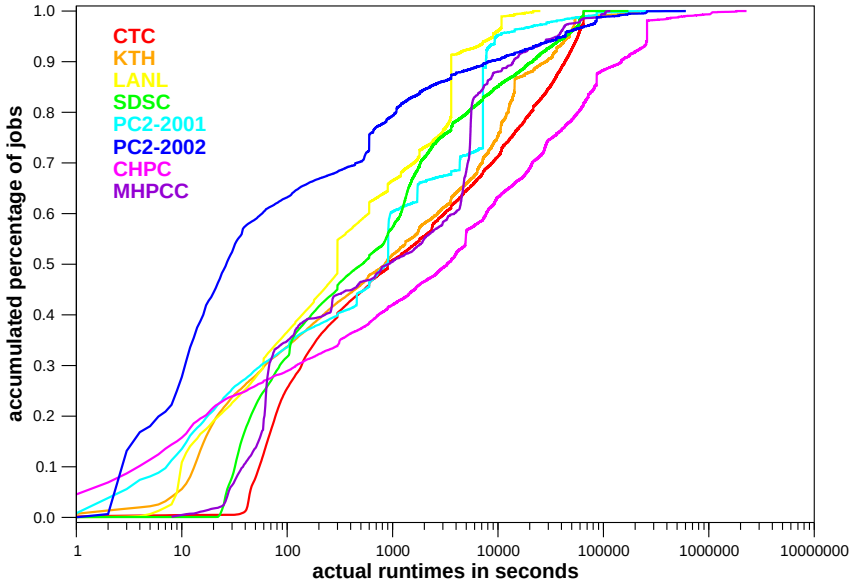


Figure 3.6: Distribution of actual run times.

the original trace or randomly chosen. Here we use synthetically generated jobs which retain the characteristics of the original traces. Several advantages arise, as it is possible to:

- generate any number of jobs,
- modify the analyzed information for generating job sets with different characteristics,
- apply other modifications, 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 does not change.

The four elementary job properties to model are: arrival time, number of requested resources, estimated and actual run time. The arrival time of each job is modelled by the interarrival time and the arrival time of the previous job. The interarrival time is best expressed by a Weibull distribution $f(x) = 1 - e^{-(\frac{x}{\beta})^{\alpha}}$. Values for α and β and the likelihood of the interarrival time for each trace are given in Table 3.7.

trace	likelihood of interarrival time	weibull distribution α	β
CTC	0.0063932	0.35	60
KTH	0.0116895	0.35	200
SDSC	0.0104242	0.40	290
LANL	0.0001635	0.45	180
PC2-2001	0.0659942	0.25	40
PC2-2002	0.0523407	0.25	40
CHPC	0.0328346	0.35	300
MHPCC	0.0012243	0.35	120

Table 3.7: Weibull parameters for the distribution of interarrival time.

Although the other three job properties all depend on each other, no distribution was found to model them [86, 53]. Hence, a 3-dimensional matrix holds probability values for all combinations of job width, estimated and actual length. The trace analyzer generates the parameters for the Weibull distribution and the probability matrix.

With this statistical information as input the job generator is started. Each time a new job is generated, first its submission time is computed from the previous arrival time, a random number and the Weibull distribution. Three more random numbers and the 3-dimensional probability matrix are used to generate the remaining values for the number of requested resources, the estimated, and actual run time.

The complete process of analyzing a trace, retrieving the statistical parameters and the probability matrix, and finally generating synthetic job sets is described in [53]. Synthetic job sets generated by this approach are used in many other *NWIRE*[18] related publications [84, 40, 13, 15, 14, 94].

3.7 Increasing the Workload

Traces consist of a set of jobs and as described each trace comes with different job properties (i. e. long/short jobs, small/large jobs, continuous/bursty arrival rates, etc.). However, a trace also reflects the performance of the scheduler and the user behavior induced by the scheduler. If such traces are used again as input for a scheduler and furthermore to evaluate

[18] Net-Wide-Resources, `http://www-ds.e-technik.uni-dortmund.de/~rmg/de/`

different scheduling strategies, the results are not meaningful. Hence, scheduling jobs which have already been scheduled by a scheduler before is not a hard task.

However, an interesting question to evaluate is "How do different scheduling strategies react on less or more workload?" Obviously, less workload is not interesting to analyze as the scheduler has less opportunities to sort a list of jobs differently or to find jobs for backfilling. Hence, an increased workload resulting from more users is emulated. Two general approaches are imaginable: reducing the average interarrival time and increasing the run time (both estimated and actual):

1. **shrinking factor:** The arrival time of each job in the job set is multiplied by this factor. This can easily be done as the arrival times are given in seconds from the first job submission. Smaller arrival times and thereby smaller interarrival times mean, that the same number of jobs are submitted in a shorter amount of time. The workload for the scheduler is increased with a shrinking factor of less than one. A benefit is, that the job area (either number of requested resources or run time) is not changed at all. However, a drawback is, that the simulated day becomes shorter and characteristic submission patterns for prime time and non prime time are also shortened.

2. **run time extension:** The run time of a job is extended by multiplying it with an extension factor greater than one. It is necessary to change both run time values (the estimated and actual run time). If only the estimated run time is increased, the job still ends at the original time. If only the actual run time is increased, the job runs longer than estimated and schedulers typically kill such jobs. However, the submission process is not changed. Characteristic arrival patterns are retained and the length of the day is not shortened. A drawback is the change of the characteristic usage of common estimated run times (Figure 3.5), e. g. common values like 10 minutes, 1 hour, or 1 day.

The following example shows, that both approaches are similar: Assume a run time extension factor of 1.1. Each job gets 10% longer and the total area of all jobs is also increased by 10%. Furthermore, assume that the makespan of the schedule only depends on the last job, which is submitted after a long submission pause. Hence, the makespan of the complete schedule is extended only by 10% of the length of the last job. And the last job may be very short. As the total area of processed jobs is 10% larger, the utilization of the machine also increases by almost 10% (definition 3.6). If the shrinking factor had been used instead for this example, the total area of all jobs (the numerator) would not have changed. Assume that the submission pause is so long that after reducing the arrival time of the last job, the submission pause still exists. The makespan is then also reduced and therefore the utilization is increased. By choosing a proper shrinking factor the same utilization as for the extension factor of 1.1 is achievable.

In the end, both approaches increase the workload and both modify job properties. However, the shrinking factor does not change the shape of the jobs. The emulation of more users is represented in a better way by the shrinking factor, as the average interarrival time is reduced. Additionally, extending the run times of jobs induces problems with algorithms that use run time bounds for changing their behavior. Such bounds would have to be adapted according to the run time extension with increased workload.

Therefore, we use the shrinking factor, in the following, to increase the workload.

3.8 Simulation Environment

As already mentioned above it is common practice to evaluate job scheduling strategies in simulation environments. For this purpose we developed MuPSiE[19]. It consists of a large set of tools which we describe in the following.

With the job-converter different trace formats like the MAUI, CCS or the SWF format (used in the Parallel Workload Archive [89]) are converted to the MuPSiE specific job format. The MuPSiE job format is simple, because jobs are described by their job number (this is also a unique ID for identifying jobs), arrival time (in seconds from the start of the job set), requested number of resources, actual run time, and estimated run time. At the start the number of resources to use is specified by the string "Machine Size:". Comments are marked with '#' and the rest consists of one job per line. As an example, the beginning of the PC2-2001 trace looks as follows:

```
# Machine Size: 96
# Start Time: Tuesday, 2 Jan 01, 08:15:43
## now the jobs !
## format: job number, submit time, req. resources, actual run time, est. run time
      0          1   64      79   21600
      1        221   96     163    1200
      2        994    8      10      60
      3       1017   64   86400   86400
      4       4966    4  172800  172800
      5       5554    8   88787  172800
      6       5688    8      25    1200
      7       6152    8      61    1200
      8       6238    4      22    1200
      9       6474   16      58    1200
     10      17661    4    3970  172800
```

We developed our own job format as it is lean and contains only the relevant information needed by the simulation environment.

Changing the average interarrival time with the shrinking factor from the previous section is done with the job-shrinker. Like all other job set related tools the job-shrinker requires the input in the MuPSiE job format. Analyzing a set of jobs is done with the job-analyzer. It provides information like presented in Table 3.4. The job-generator is used to generate synthetic job sets. For input this tool needs a trace statistic as described in Table 3.7. For obtaining the statistical data an analyzer tool from the NWIRE environment [94] is used.

Both concepts of resource management systems (queuing and planning system) and a variety of scheduling policies are implemented in MuPSiE. Especially the re-planning process of CCS [52] is implemented. It consists of: clearing the whole current schedule, reinserting all running jobs, reinserting all made reservations, and finally reinserting the sorted list of variable jobs by placing each job asap.

Besides these three basic sorting schemes FCFS, SJF, and LJF several other sorting policies are also implemented. And of course all algorithms presented in this work are implemented. Furthermore, it is also possible to compute optimal schedules by means of the CPLEX library.

Each schedule event (submit, start, actual end) is printed by the scheduler in a special format, which is later used to analyze the performance of the scheduler. Examples are:

[19] Multi Purpose Simulation Environment (for job scheduling)

- `SUBMIT-0 8008950, e_n 15, bl 11, ID 4538`:
  The job with ID 4538 was submitted at 8,008,950 seconds since the start of the simulation. After the job is submitted the backlog consists of 11 jobs (including this one) and 15 nodes are empty. The information about empty resources (e_n) and the backlog (bl) is needed by the schedule analyzer to compute the loss of capacity (definition 3.7). The number after the keyword SUBMIT is used as an identifier to differentiate between several machines, if a grid-environment with more than one machine is simulated.

- `START-0 8011163, ar 8006040, wi 16, e_n 0`:
  8,011,163 seconds after the start of the simulation the job submitted at time 8,006,040 is started by the scheduler. 16 resources are allocated at this time and 0 empty resources are leftover afterwards. Note, this event is not needed for the schedule-analyzer. It is printed for an improved human readability of the schedule.

- `END-0 8011178, ar 8006040, st 8011163, wi 16, e_n 16, bl 11, ID 4527`:
  Job 4527 that arrived at 8,006,040 and was started at 8,011,163 seconds finally finished its execution at 8,011,178 seconds and had used 16 resources. Again, information for measuring the loss of capacity is printed as it is done in the SUBMIT event. The END event is the most important event in the schedule printout. From the given information the waiting time, the response time and the run time of a job are computed. From these values the more sophisticated performance metrics like slowdown are derived.

At the end of a simulation run average data about the scheduling process itself is printed. The following example is retrieved from a simulation of a planning system with FCFS and uses the original CTC trace of 79,302 jobs as input:

```
#queue with more than 1 job: 44126
#average queue length: 49.5574
#average schedule time: 1775 clock-ticks = 0.001775 seconds
#number of reschedules done: 156791
#simulation ended normally
```

The first two lines correspond with the last two columns in the upper part of Table 3.6. 156,791 re-schedules were done and each computation lasted for an average of 1.7 milliseconds. Additionally, the computational time for each re-scheduling step is printed during the simulation run, so that a detailed observation is possible. The number of reschedules is slightly less than twice the number of simulated jobs. This is because a re-schedule is done for each job submit and job end, but sometimes more than one job is submitted or ends at the same time. In this case only one re-schedule after the last job submit or end is necessary.

Schedules stored in the above given format are read-in by the schedule-analyzer to compute average performance metrics. Using the information from Table 3.5 it is also possible to print out monthly averaged performance numbers (Chapter 5). Thereby it is possible to evaluate the performance more precisely. A schedule-viewer is used to present a schedule graphically. Such a representation is often helpful during the development and debugging of new scheduling strategies.

The simulation environment is developed with Borland C++ Builder 5 under MS Windows. It is implemented platform independent in C++ using STL (Standard Template Library) classes. The simulation environment has been successfully tested on various architectures.

Primarily, simulation runs were done on the *hpcLine* [43] cluster running Linux. The CPLEX-scenarios required more memory, hence an 8-way SMP, 64-bit SUN and dual IA-64 Itaniums were used.

Additional functionality is available in the MuPSiE environment which ease the everyday usage. A restart mechanism stores the simulation state on a regular basis (e. g. every hour) and therefore makes it possible to simulate large and long-running scheduling problems. In combination with automatic startup scripts many simulation runs are automatically distributed on the cluster nodes, so that autonomous computations are possible.

3.9 Summary

In the previous sections we presented general aspects, terminologies, and definitions for job scheduling in resource management systems. A classification of resource management systems based on the scheduled time frame was done as a basis for our work. In contrast to queuing systems, planning systems schedule the present and future resource usage. Every waiting job in the system is placed in the schedule and gets a proposed start time assigned. Of course run time estimates are mandatory in order to plan the schedule. Planning based resource management systems and their approach to scheduling jobs are the basis for the self-tuning dynP scheduler.

The scheduling process is influenced by the scheduling policy, which often defines a sorting order for the waiting jobs. Many policies exist and the focus is on FCFS, SJF, and LJF. These three policies are also implemented in the resource management software CCS, which is another basis for this work. As we used the planning based scheduling approach, some kind of backfilling is done implicitly. Backfilling is a common extension to queuing systems, which improves the response time and utilization by neglecting the sorting order of the jobs. If the highest prioritized job has to wait for enough free resources, backfilling starts other jobs with the guarantee that the highest prioritized job will not wait any longer.

In order to measure the schedulers performance and for comparing different schedulers, performance metrics are needed. User and owner centric metrics exist. We chose the slowdown (weighted by the area of jobs) as the user centric metrics and the utilization of the system as the owner centric metrics.

We used a discrete event simulation environment called MuPSiE for the evaluation of different schedulers. For job input into the simulations we used traces from real HPC systems. The Parallel Workload Archive and the Maui Scheduler Workload Repository contains many traces for this purpose. As a planning based scheduling approach requires information about run time estimates, only a subset of the available traces are usable. These are CTC, KTH, LANL, and SDSC, respectively CHPC and MHPCC (the names represent the institutions where the corresponding machines are installed). The first four traces were derived from IBM SP2 installations, while the others came from cluster systems. Additionally, trace data from CCS and the *hpcLine* cluster installed at the PC^2 were available. We extracted two traces of the year 2001 and 2002 from the archive. We analyzed these eight traces in detail and extracted basic characteristics. Later the original traces were used to evaluate the performance of the different schedulers.

In order to evaluate the schedulers with an increased workload, two approaches were introduced: 1) increasing the jobs run time and 2) decreasing the interarrival time. In the first approach the outlook (i. e. area) of the job is changed. Hence, performance metrics that

are based on the run time (e. g. slowdown or run time) will automatically change, although the schedulers performance might not be worse. The second approach leaves the area of jobs untouched and changes the submission behavior (i. e. interarrival time). As the day becomes shorter, common submission patterns for prime and non prime time are also changed. Although both approaches have drawbacks, we used the second approach in this work. Especially with the dynP scheduler with bounds the first approach is not suitable, as the bounds are set according to the run time of the jobs. If the run time is changed, so as to increase the workload, it would be necessary to adapt the bounds, too.

Finally we described the discrete event simulation environment, which we use for the evaluation of the dynP scheduler.

4 Dynamic Policy Switching

In the previous chapter we presented general aspects, definitions, and common terminology for job scheduling and the evaluation of job scheduling policies for resource management systems. Herewith, the basis for the following work has been set. 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 watch and change the scheduling policy permanently.

We developed the family of dynP schedulers, which automatically switch the active scheduling policy during run time. In general, the set of scheduling policies to choose from can consist of many or even all policies one can think of. However, we restrict the set of used policies to first come first serve (FCFS), shortest job first (SJF), and longest job first (LJF). This is done, as these three scheduling policies are commonly known and used in many resource management systems. SJF is known for reducing the average waiting time of jobs, while LJF is known for increasing the utilization of the system. FCFS is a good compromise between these two contradicting objectives and generates understandable schedules. Therefore it is used as the active policy at many sites. We restrict the set of used policies to the mentioned three, as we want to evaluate the general behavior and performance of the dynP scheduler family and if it is beneficial to switch the policy during run time. We do not want to evaluate, which combination of policies is best suited for specific job characteristics. Presumably, combinations with other and more scheduling policies exist, which generate better results, than presented in the following.

Initially, we present 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. This version and some performance numbers are presented in Section 4.2. 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.

The aim is to have no more input parameters which depend on the job characteristics. In Section 4.3 we present the basic concept of such a self-tuning scheduler for the resource management of HPC systems, which automatically switches the active scheduling policy. Decider mechanisms with different levels of sophistication can be applied to the self-tuning dynP scheduler. In Section 4.4 a simple and an advanced decider are presented. The advanced decider is fair in its decisions, i. e. it does not prefer any policy. Furthermore, the currently active policy has to be considered in certain decision scenarios in order to find the right policy to use. The preferred decider explicitly prefers a single policy. Some options for the self-tuning dynP scheduler are presented in Section 4.5. With these options it is possible to influence the self-tuning dynP scheduler in a general way, e. g. to add some slackness for reducing the amount of policy switches.

As the self-tuning dynP scheduler generates full schedules for each policy in order to measures them, a quasi off-line scheduling is done. In Section 4.6 we give answer to how much performance is lost when common scheduling policies are used. The scheduling problem is modelled as an integer problem. We use the ILOG CPLEX library to solve the integer problem and to compute the optimal schedule.

4.1 History of Development

At the Paderborn Center for Parallel Computing (PC^2) the self-developed resource management system CCS (Computer Center Software, [52]) is used for managing the *hpcLine* cluster [43]. Three scheduling policies are currently implemented: FCFS, SJF, and LJF. According to the classification of resource management systems, CCS is a planning based resource management system. Jobs are placed in the schedule as soon as possible, hence backfilling is done implicitly. Although three policies are implemented, FCFS is used most of the time. It is chosen because it provides a certain level of fairness among the jobs and the understandability of the schedule is granted. For debugging and accounting CCS stores information about scheduled jobs in trace files.

From these facts a simple question concludes: Is or was FCFS the best choice, in terms of response time, for scheduling the submitted workload or do SJF or LJF perform better? To answer this question we analyzed the scheduling process of CCS and implemented it in the MuPSiE simulation environment. Two representative job sets were extracted from the available job traces. Each job set consists of roughly three months of the year 2000. Table 4.1 shows the basic job characteristics. In `set1` the duration of jobs is short and the jobs are submitted at a high rate. In `set2` jobs are more than two times longer and they are submitted with a greater average distance (interarrival time).

job set	number of jobs	average requested nodes	average actual run time [sec.]	average estimated run time [sec.]	average interarrival time [sec.]
`set1`	8,469	13.49	2,836	6,554	628
`set2`	8,166	10.11	6,698	21,634	1,277

Table 4.1: Characteristics of the two jobs sets used in [85].

Simulations with the three scheduling policies were performed for these two job sets. Various shrinking factors were used to generate different workloads. The average response time (ART) was used for measuring the performance. The results show, that when focusing on medium utilizations, SJF was the best choice for one job set, followed by FCFS and LJF. For the other job set LJF was best and SJF was worst. FCFS is a good average for both job sets. However, a clear winner could not be found.

`set1`	SJF	FCFS	LJF
~63%	8,335 s (+49%)	16,587 s (0%)	26,459 s (-60%)
~78%	34,840 s (+23%)	45,292 s (0%)	93,641 s (-106%)

`set2`	SJF	FCFS	LJF
~58%	18,388 s (-7%)	17,182 s (0%)	13,883 s (+19%)
~73%	45,080 s (-19%)	37,998 s (0%)	26,673 s (+30%)

Table 4.2: Average response times for SJF, FCFS, and LJF at medium utilizations. FCFS is used as a reference for computing the percentages. Smaller values are better.

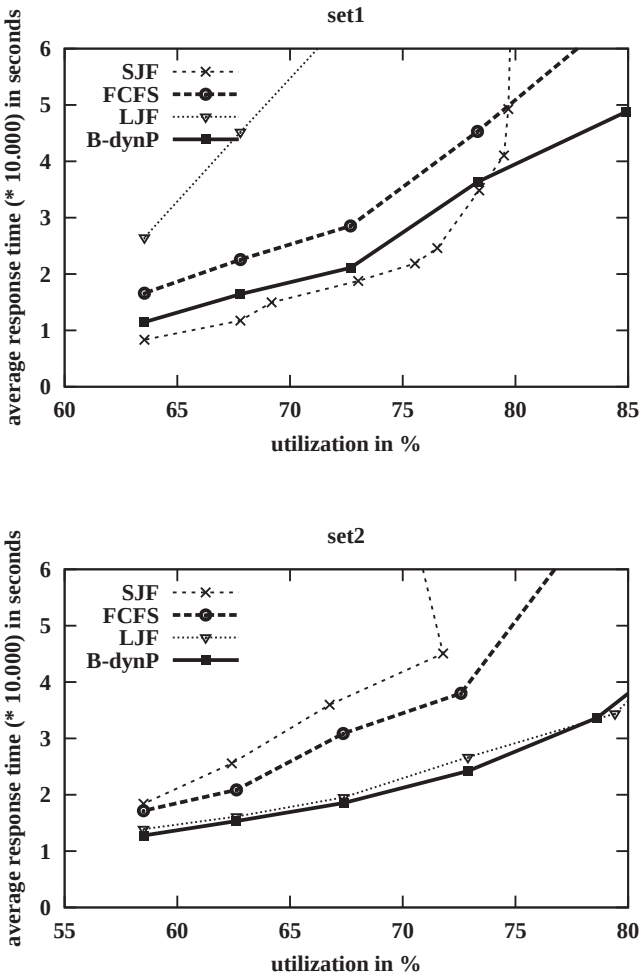


Figure 4.1: Performance (average response time) focusing on medium utilizations. `set1` left, `set2` right. The dynP scheduler uses 2 hours as the lower bound and 2 hours and 30 minutes as the upper bound.

Obviously the performance (and ranking) of the three scheduling policies is strongly related to the characteristics of incoming jobs. Therefore, the basic idea of dynamic policy switching (dynP) is to dynamically switch the scheduling policy while the system is running. The aim is to always work with the best scheduling policy for the current waiting jobs.

Two aspects have to be considered in a dynP scheduler:

- When should the scheduler decide to switch the policy?
- Which new policy should the scheduler choose?

Possible answers to the first question might be: every time the schedule changes (i. e. at each job submit and job end), only when new jobs are submitted, every 10 new jobs, or in fixed intervals (e. g. every hour).

Basically, the first alternative is chosen in order to avoid any loss of performance. As only job submits and job ends induce a schedule change, checking for a new policy at these

events is sufficient. Note, a job start does not change a planned schedule. Checking for a new policy only at job submits might be sufficient. Also job ends might be neglected, as the characteristics of waiting jobs do not change, when an already running job terminates.

Answers to the second question are given in the following.

4.2 Use of Bounds

As mentioned at the beginning of this chapter, we use three policies for evaluating the benefits of policy switching schedulers. FCFS, SJF, and LJF are commonly known and are typically used in modern resource management system for scheduling HPC systems. As SJF and LJF use the estimated duration of waiting jobs for sorting them, an intuitive approach is to also use this criterion for policy switching. Therefore, the average estimated run time (AERT) of all currently waiting jobs is computed. The new scheduling policy is found by testing the computed AERT value against a lower and upper bound:

$$new\ scheduling\ policy = \begin{cases} SJF, & if\ lower\ bound \geq AERT \\ FCFS, & if\ lower\ bound < AERT \leq upper\ bound \\ LJF, & if\ AERT > upper\ bound \end{cases} \tag{4.1}$$

The shown switching mechanism is short and simple to understand. However, a proper setting of the two bounds is essential for a good performance. Finding good bounds to improve the schedulers performance is difficult. Doing an experimental search process is one solution, but it usually takes much time and effort. If the characteristics of the waiting jobs change significantly, the bounds probably have to be re-adapted so that the scheduler still achieves the best possible performance.

The set of used policies may also be restricted. For example:

- if the lower bound is set to zero, only FCFS and LJF are chosen
- if the upper bound is set to infinity, only SJF and FCFS are chosen
- if the lower and upper bound are the same, the decider switches only between SJF and LJF.

By choosing bounds like e. g. 24 hours for the upper bound and one hour for the lower bound, a generalized behavior of the machine is defined. Or a dynP scheduler only considers the time of day and the day of week. For example, LJF during the weekend, FCFS during prime time (during the day) in the week, and SJF during non prime time in the week.

We did an experimental search process in [85] for the two jobs sets from Table 4.1. At that time these two jobs sets were representative for the usage of the new machine. Tested bounds varied between ten minutes and one hour for the lower bound and one hour to half a day for the upper bound. A setting of two hours (7200 s) for the lower bound and two hours, 30 minutes (9000 s) as an upper bound generates the best results for both job sets. The performance for this dynP setting is also shown in Figure 4.1. For the `set2` the dynP scheduler even outperforms the best basic policy LJF at medium utilizations. Figure 4.2 shows the overall usage of the three policies under different workloads. With different job characteristics the dynP adapts its behavior. As the average estimated run time for `set1` is short compared to `set2`, SJF and FCFS are commonly chosen by the dynP scheduler. The

usage of LJF drops with increasing load. In contrast, the dynP scheduler behaves completely different for `set2` although the bounds are the same. As the average estimated run time is almost three times larger than in `set1`, LJF is used most of the time. With increasing load the usage of LJF increases from 60% to around 90%. At high loads SJF is occasionally used.

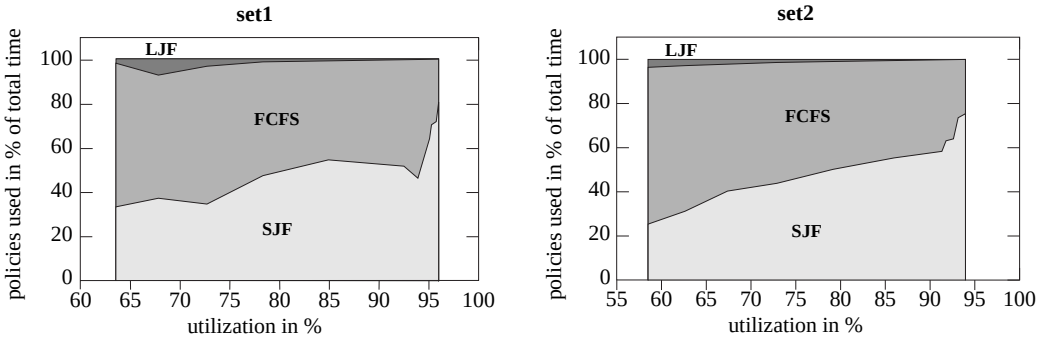


Figure 4.2: Accumulated percentages of policies used by the dynP scheduler with lower bound 7200 s and upper bound 9000 s. `set1` left, `set2` right.

4.3 Concept of Self-Tuning

The dynP scheduler with bounds is able to achieve good results. However, this depends on the proper setting of the two bounds. Finding them with an experimental search takes a lot of effort and time. Hence, in a real world scenario the scheduler should be easier to use for the administrative staff. Usually, they have no time to perform a lengthy experimental search for good parameter settings. Especially if the bounds might have to be re-adapted often, in order to reflect changing job characteristics. Therefore, a scheduler with dynamic policy switching has to work autonomously. That means, that no more input parameters are required, which might depend on the job characteristics.

To achieve this, the dynP scheduler has to find good bounds by itself. One way is to integrate the experimental search process in the scheduler. Simulations with the previously executed jobs are done automatically with various bound settings. However, the experimental search still has to be done. Furthermore, the decision for the future is based on past information.

With the ability to plan the future resource usage like a planning based resource management system does, a more sophisticated approach is possible to find a new policy. For all waiting jobs the scheduler computes a full schedule and planned start times are assigned to every waiting job in the system. With this information it is possible to measure the schedule by means of a performance metrics (e. g. response time, slowdown, or utilization). Now the basic idea 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 metrics. Thereby, the performance of each policy is expressed by a single value. These values are compared and a decider mechanism chooses the best

policy, i. e. the lowest (average response time, slowdown) or highest (utilization) value.

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

4.4 Decider Mechanisms

For the required decision several levels of sophistication are thinkable. We presented the *simple decider* that basically consists of three if-then-else constructs in [87]. The decider searches for that policy, which generates the minimum value. At first the performances of SJF and LJF are compared, finally the best one is compared to FCFS:

```
1   IF (SJF <= LJF) {
2      IF (FCFS <= SJF) {
3         new_policy = "FCFS";
4      } ELSE {
5         new_policy = "SJF";
6      }
7   } ELSE {
8      IF (FCFS <= LJF) {
9         new_policy = "FCFS";
10     } ELSE {
11        new_policy = "LJF";
12     }
13  }
```

Example 4.1: The simple decider for the self-tuning dynP scheduler. FCFS, SJF, and LJF are used as abbreviations for the performance values of the corresponding policies.

In the given example the best performance is achieved by the smallest value. The algorithm is directly usable with common user centric performance metrics (e. g. average response time or slowdown). If performance is measured as utilization, either the logical operators or the performance values have to be inverted.

Whilst comparing the performance, it might occur that two policies are equal. In these cases arbitrarily FCFS (lines 2 and 8) and SJF (line 1) are preferred.

However, the simple decider also has drawbacks. It does not consider the old policy. Especially if two policies are equal and a decision between them is needed, information about the old policy is helpful. We did a detailed analysis of the simple decider (Table 4.3) in [86], which shows, that in four cases even a wrong decision is made by the simple decider (cases: 1, 6b, 8c, and 10c). FCFS is favored in three and SJF in one case, although staying with the old policy is the correct decision with these cases.

The *advanced decider* generates decisions as shown in the last column of Table 4.3. However, complex decision scenarios might occur. For example, the old policy is worse than the others, but all others achieve an equal performance. So what is the new policy? In three cases no exact decision is possible with the three performance numbers and the old policy:

- case 6c: the old policy (LJF) needs to be switched as it is obviously the worst. Either FCFS or SJF could be chosen, as both are equal. FCFS is chosen arbitrarily, as it might be beneficial to the average response time of the generated schedule.

case		combinations	simple decider	correct decision
1		FCFS = SJF = LJF	**FCFS**	**old 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	old policy = FCFS	FCFS (= old policy)	old policy (= FCFS)
	b	old policy = SJF	**FCFS**	**old policy (= SJF)**
	c	old policy = LJF	FCFS	FCFS
7		FCFS = LJF, SJF < FCFS (⇔ SJF < LJF)	SJF	SJF
8		FCFS = LJF, FCFS < SJF (⇔ LJF < SJF)		
	a	old policy = FCFS	FCFS (= old policy)	old policy (= FCFS)
	b	old policy = SJF	FCFS	FCFS
	c	old policy = LJF	**FCFS**	**old policy (= LJF)**
9		SJF = LJF, FCFS < SJF (⇔ FCFS < LJF)	FCFS	FCFS
10		SJF = LJF, SJF < FCFS (⇔ LJF < FCFS)		
	a	old policy = FCFS	SJF	SJF
	b	old policy = SJF	SJF (= old policy)	old policy (= SJF)
	c	old policy = LJF	**SJF**	**old policy (= LJF)**

Table 4.3: Detailed analysis of the simple decider. Decisions printed in bold highlight the differences.

- case 8b: similar to case 6c, but FCFS or LJF could be chosen.
- case 10a: similar to case 6c, but SJF or LJF could be chosen. SJF is chosen in order to prefer short jobs.

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 jobs run time and usually the jobs 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 the *preferred decider* was developed. The preferred policy is not switched unless any other policy is clearly 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, in this work we evaluate the general behavior and performance of self-tuning schedulers. In this chapter, for the domain of resource management

of HPC systems. We do not want to 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.

4.5 Options for the Self-Tuning dynP Scheduler

The aim of the self-tuning dynP scheduler is to eliminate job input parameters, especially those which depend on the characteristics of the processed jobs (i. e. the two bounds of the simple decider) and need to be re-adapted continuously. Nevertheless, options 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-adaption is not needed. Some options to mention are:

- At what time or event is self-tuning invoked? Only when a new job is submitted. Or also each time the schedule changes, i. e. when a running job ends earlier than estimated and when a new job is then placed in the schedule.
  If jobs end earlier than estimated other jobs might be scheduled at an earlier start time and therefore the schedule changes. Note, starting a job does not change the schedule. It is an implementation of the planned schedule.
  In the following *full self-tuning* means, that at each job submit and at each time a running job ends self-tuning is invoked. With *half self-tuning* this is done only when new jobs are submitted. Hence, roughly half as much self-tuning is performed.

- Adding slackness while switching policies. The motivation comes from the field of electrical engineering and the hysteresis of electric ferromagnets whilst changing their polarity.
  A self-tuning dynP scheduler with slackness requires the new policy to be better than the old policy by at least the given slackness threshold (e. g. 2%). This prevents rapid and consecutive policy switching.
  Additionally, this also prevents user induced policy switches, subsequently switching between SJF and LJF might change the schedule significantly. In worst case scenarios a user may submit dummy jobs which induce the self-tuning dynP scheduler to switch the policy. Consequently the important job from the user is started earlier than jobs from other users which are then further more delayed.

- As the self-tuning dynP scheduler requires a planning system, planning is done for all waiting jobs. Hence, all waiting jobs influence the decision. However, in some scenarios one could think of restricting the number of jobs that influence the decision made by the self-tuning dynP scheduler. Many approaches are possible: if the complete schedule is planned only the first n jobs are used to evaluate the performance, or only those jobs which are started within the next t hours are used. More sophisticated criteria like e. g. the duration, parallelism, or priority of jobs might also be used for filtering jobs.

One could think of many more options which influence the decision process of the self-tuning dynP scheduler, but are independent of incoming jobs. In the following evaluation only the first and second option are used.

4.6 Optimal Schedules with CPLEX

As described before, the self-tuning dynP scheduler generates full schedules for each available policy (here: FCFS, SJF, and LJF). Full schedules contain a start and end time for every submitted job. Therefore, it is possible to compute waiting and response times for each job. With a performance metrics the schedules are analyzed and the self-tuning dynP scheduler chooses the policy that generates the best schedule and switches to it. This process is called a self-tuning step and each time a quasi off-line scheduling is done as the number of jobs are fixed. However, it is not a classic off-line scheduling by optimizing the makespan. And the schedule does not start with an empty machine, i. e. some resources are not available. The history of resource usage has to be considered as a result of jobs started in the past.

From this two questions arise:

1. What is the optimal schedule in each self-tuning step?
2. What is the performance difference between the optimal schedule and the best schedule generated with one of the scheduling policies?

The second question is interesting, as it gives answers to how much performance is lost when a common scheduling policy like FCFS (+ backfilling) is used.

An approach to compute optimal schedules is to model the scheduling problem as an integer problem, which is then solved with the well-known ILOG CPLEX library [45].

4.6.1 Modelling the Scheduling Problem

Following [92], we model the scheduling problem as an integer problem [74]:

Three values are used to describe the properties of a job i. The number of requested resources is denoted with w_i (width). The estimated duration is described by d_i and the job is submitted at time s_i.

The history of resource usage is a list of tuples. A tuple consists of a time stamp and the number of resources that are free from that time on. Figure 4.3 shows an example. The number of free resources are increasing monotonously as only already running jobs are considered. And if more than one job ends at the same time, a single time stamp is sufficient. Note, the estimated duration of already running jobs has to be used for generating the time stamps.

time [sec.]	free resources
100	16
120	48
140	64
160	80
200	96

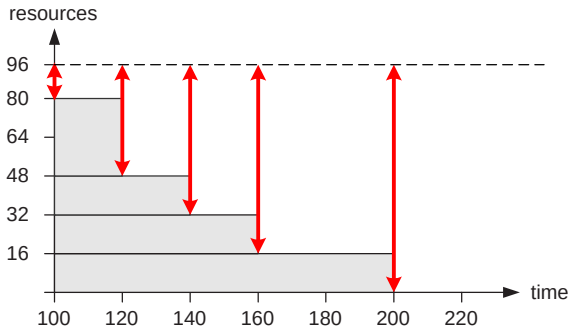


Figure 4.3: Example for a machine history.

The variables are defined as:

$$x_{it} = \begin{cases} 1\,, & \text{if job i is started at time t} \\ 0\,, & \text{else} \end{cases} \tag{4.2}$$

The average response time weighted by width is used as the objective function. As previously mentioned, ARTwW and SLDwA behave similarly in comparing schedules if the same job input is used. The objective function is defined as:

$$\text{Minimize} \qquad \sum_{i,t} x_{it}\ (t - s_i + d_i)\ w_i \tag{4.3}$$

The constraints are:

$$\sum_{t} x_{it}\ = 1 \quad \forall i \tag{4.4}$$

$$\sum_{i,\max(0,t-d_i)\leq j\leq t} x_{ij}\ w_i\ \leq M \quad \forall t \in [0,T] \tag{4.5}$$

Constraint 4.4 describes that every job is started only once. In constraint 4.5 T is the maximum possible length of the schedule. Usually this is infinity, but the resulting integer problem would contain too many variables. Assuming that the schedules for FCFS, SJF, and LJF are already computed, the best solution is to use the maximum makespan of the three schedules. This is most likely the makespan of the LJF-generated schedule. The sum in constraint 4.5 describes the fact that the machine consists of M resources in total. In order to reflect the machine history the number of available resources has to be reduced accordingly (see Figure 4.3).

The smallest time step in resource management systems is usually one second. This requires t in Equation 4.2 to be at a second scale. However, this induces too many variables (number of jobs times T in seconds) and therefore too much memory (roughly the number of variables times T). For example: the maximum makespan is two days (172,800 seconds) and an optimal schedule for eight jobs has to be computed. The number of variables is already more than a million, although the scheduling problem is rather small for a real world scenario. Such problems would need a considerable amount of memory. If the problems grow in size, the 8 GB of the available simulation hardware are not enough. A commonly used solution to solve this, is to use time-scaling [39]. This means, that the schedule is computed at a greater time scale (e. g. one minute). By that a certain amount of time in the real schedule is reduced to a single point of time. For most real world scenarios this is enough, as jobs are usually estimated to run for several minutes, hours, or even days, so significantly longer than one second.

We use the following approximation for computing a suitable time-scale: the size of the integer problem (i. e. the number of matrix entries and constraints) roughly depends on the number of jobs multiplied with the square of T. The variable matrix is sparse and the degree of sparseness depends on the accumulated run time of all jobs [74]. Hence, the size of the integer problem in memory (i. e. the total amount of memory that should be used) is computed by

$$\text{number of jobs} \cdot \left(\frac{\text{max. makespan}}{\text{time-scale}}\right)^2 \cdot \frac{\text{acc. run time}}{\text{max. makespan} \cdot \text{number of jobs}} \cdot x \tag{4.6}$$

x denotes the memory size of each matrix entry in bytes. In initial testings we discovered that good values for x are 0.1 kB or 0.0001 MB.

Then:

$$\text{time-scale} = \sqrt{\frac{\text{max. makespan} \cdot \text{acc. run time} \cdot x}{\text{available memory}}} \tag{4.7}$$

The time-scale is rounded up to the next 60 seconds, so that the schedules are solved in a full minute scale. Additionally, the amount of memory used for the integer problem should be about four times smaller than the total memory available, as the additional memory is needed by CPLEX during the solving phase. For the computations a machine with 8 GB of total main memory is used.

However, time-scaling has drawbacks. Jobs are scheduled at the beginning of a time-scaled interval, e. g. at the beginning of a one minute interval in the schedule. The duration of jobs is not time-scaled and they still end at any time in the time-scaled interval. Hence, from the time a job ends to the next interval start resources remain unused, although they could be utilized without time-scaling. To solve this problem each job is moved forward as much as possible after the optimal starting order of the jobs is found. With that, unused slots in the schedule are avoided. The maximum time a job is moved forward is (time-scale $-$ 1) seconds and on average $\frac{\text{time-scale}-1}{2}$ seconds. To implement this in practice (and also in the simulation environment MuPSiE) each job is inserted in the schedule according to the starting order of the optimal schedule computed by CPLEX. Each job is placed as soon as possible and unused time slots, due to time-scaling, do no longer occur. By applying time-scaling the final schedule might not be optimal, as heuristics are used. However, if time-scaling is not applied, the problem might not fit in the available memory and no solution is computed.

This CPLEX-computed schedule is analyzed and the results are compared to the performance of the three basic policies. We define the quality of a policy p and according to a performance metrics m(e. g. SLDwA) as:

$$\mathit{quality}(p, m) = \frac{\text{performance measured with } m \text{ of the CPLEX-computed schedule}}{\text{performance measured with } m \text{ of the schedule generated by } p} \tag{4.8}$$

If $\mathit{quality}(p, m) < 1$ the schedule computed by CPLEX is better. The percentage $(1 - \mathit{quality}(p, m)) \cdot 100$ depicts how much performance is lost by using the policy p. Due to time-scaling the $\mathit{quality}(p, m)$ might be > 1. In this case, the policy p is better than the schedule computed by CPLEX with time-scaling applied.

Despite solving the integer problem and computing the optimal schedule, the time CPLEX needs for the computation is of an additional importance. Computing an optimal schedule and using it in a resource management system surely improves the performance of the system. However, if the computation takes too much time (e. g. one hour), it is not practical. In general, the scheduling component of a resource management system should generate new schedules as fast as possible, which typically means close to one second or less in a real world scenario.

The time for scheduling a new job is not critical in queuing based resource management systems as the job is appended to a queue. In contrast, a planning based resource management system re-plans the resource usage and considers the new job. This has to be done fast, as the updated schedule is required for the following requests. For example, a request for a reservation is submitted right after. An answer is expected immediately as other reservation requests might depend on the acceptance of this request. Hence, the updated resource plan

has to be computed fast. Therefore, finding and using optimal schedules in the real world is not practical. With the basic policies of the self-tuning dynP scheduler, the time of scheduling is less than 10 milliseconds for an average number of 25 waiting jobs.

Therefore, a mixture of quality and computational time has to be used in a comparison. In other words the physical definition of power, i. e. work per time unit, is well suited for measuring the performance of a scheduler.

4.6.2 Results

In the following we would like to answer the questions from the beginning of this section, i. e. How much performance is lost, compared with the optimal schedule by using common scheduling policies like FCFS or SJF? For this the following configuration for the self-tuning dynP scheduler is used: SJF-preferred decider, self-tuning is invoked only at job submits, ARTwW as self-tuning metrics, and a slackness of 4%. Initially, we planned that simulations are performed with several job sets. However, as solving the integer problem with the CPLEX library need a lot of computational time, we used only one job set in the end (the CTC trace). The shrinking factor is not applied, i. e. it is one and the workload is not further increased.

Optimal schedules are computed in each self-tuning step, hence at every job submission. Although optimal schedules are available, they are not used for the actual scheduling process. They are only used for the comparison with the schedule of the best basic policy in each step of the self-tuning process. Hence, it is possible to directly state how much performance is lost in each step of the self-tuning dynP scheduler. If optimal schedules were to be used directly for scheduling the jobs, future scheduling decisions and optimal schedules would be influenced by a different past resource usage and a fair comparison would not be possible.

As previously stated the CPLEX approach with solving an integer problem needs a lot of memory. This is due to the definition of the variable $x_{i,t}$ and the constraints. Hence, time-scaling is used to reduce the number of variables, which implies that schedules are only solved on a one minute or greater scale.

Table 4.4 shows exemplary CPLEX runs for various problem sizes, i. e. schedules. For each line in the table the time at which the integer problem was solved is given, i. e. a new job was submitted and self-tuning was invoked. The three columns show the values used to compute the time scaling according to Equation 4.7. The resulting time scale used for solving the integer problem is given in the following column. Finally, the last columns depict the results and performance of computing an optimal scheduler with CPLEX. As stated above, if the performance loss is positive, the CPLEX computed schedule is better than the schedule generated by the scheduling policy. Of course this should be the normal case. However, sometimes the performance loss is negative. This indicates, that the scheduling generated by the best scheduling policy is better than the solution found by CPLEX. Obviously this is caused by the time scaling in the integer problem. If no time-scaling is applied and enough memory and compute time is available, the CPLEX should always at least find the same schedule as any scheduling policy and most likely a better one.

It is noted that the problem sizes in the first block are considerably large. This is indicated by the amount of jobs, the accumulated run time, and finally the time scale. The performance loss of the scheduling policy (in all four cases SJF + backfilling) is very small and in the 1% range. CPLEX needs much compute time for achieving this result.

The second block of examples show, that it is impossible to predict the compute time of CPLEX from previous runs. In these two successive job submissions both scheduling problems

	CPLEX problem size				CPLEX result				
submission time	jobs	makespan [sec.]	acc. run time [sec.]	time scale [min.]	quality	perf. loss	comp. time hr.	min.	sec.
38,589	40	189,559	1,798,837	7	0.9920	0.80%	8	14	17
38,590	40	189,596	1,862,437	8	0.9869	1.31%	1	47	35
40,284	31	190,899	1,395,637	7	0.9934	0.66%	12	58	51
40,493	32	191,509	1,395,937	8	0.9968	0.32%	23	22	48
50,356	18	194,121	1,030,782	6	0.9865	1.35%	2	26	32
50,360	19	194,161	1,095,582	6	0.9974	0.26%	40	52	51
70,628	17	259,141	715,905	6	0.9821	1.79%	76	15	8
71,271	17	256,741	723,405	6	0.9804	1.96%	129	16	25
71,285	17	256,741	728,505	6	0.9804	1.96%	237	7	59
36,037	39	178,232	1,617,937	7	0.8913	10.87%	3	26	4
52,698	19	201,983	906,331	6	1.0014	-0.14%	14	43	40
69,073	9	239,417	519,300	5	1.0022	-0.22%	0	23	40
averages	21.7	166,766	931,168	5	0.9930	0.70%	5	34	29

Table 4.4: Examples of CPLEX problem sizes, the quality, and the compute time.

are of an equal size. In the first case, CPLEX needs 2.5 hours to find an optimal solution which is 1.3% better than the best scheduling policy. With the submission of the next job, the scheduling problem increases only slightly in size. The performance loss, hence the difference between the scheduling policy and the CPLEX-computed solution is much smaller than before. However, in this case almost 20 times more compute time is needed. This might be the results of the newly submitted job, which might have increased the degree of difficulty substantially without changing the size of the problem.

In the third block, examples for extremely long compute times of CPLEX are given. Note, 237 hours equals approximately 10 days. However, the CPLEX-computed solutions are clearly better than the schedules of the best basic policies. A time scaling of 6 minutes is used, so that an even larger improvement might be possible, if a second precise scaling is applied. Of course this would require a considerable amount of main memory.

In all previous examples the performance loss of the scheduling policy is within the 1% range. However, the case might be that the scheduling policy is significantly worse. As shown, the largest measured performance loss is close to 11%. About 3.5 hours are needed by CPLEX to compute this solution. The other two examples show, that it is also possible that the CPLEX-computed schedule is worse than the SJF-generated schedule. Obviously this is due to the time scaling in the integer problem. However, the long compute time (almost 15 hours) of CPLEX has to be considered.

Finally, the last row in Table 4.4 shows the averages of all CPLEX computations. It is seen that the performance loss of the scheduling policies is only 0.7% compared to the CPLEX-computed solution with a 5 minute average time scaling applied. On average more than 5 hours are needed to solve the integer problems. The average size of the according scheduling problem is 22 jobs with a maximal makespan of close to 2 days as an upper bound. In real world scenarios such schedule would be considered to be small. Many more jobs are usually processed in a resource management system, with a much larger makespan. This indicates, that CPLEX-computed schedules are unpractical for a real implementation in a resource management system. The response times of real schedulers need to be considerably small, so that user decisions (e. g. accepting or declining a reservation) can be made quickly. Approaches

are thinkable, where the scheduling policy is used to generate an initial schedule and CPLEX is used to find better schedules while the initial schedule is active and implemented. However, in online scheduling systems jobs are submitted continuously and with short average interarrival times (on average 369 seconds for the CTC trace). Hence, a new schedule is required, while CPLEX is still solving the previous scheduling problem.

5 Evaluation of the dynP Scheduler

It is common practice to use simulation environments for the evaluation process. Simulation environments need job sets and information about the simulated machine as input, i. e. the total number of resources, if a homogenous machine and no topology of the network is assumed.

At first, we use the original eight traces and the corresponding machines. As exact start dates and times are known for all traces, it is possible to evaluate the results of the scheduling policies on a monthly basis. With this, differences in the trace become visible. Additionally, the total number, the average width (number of requested resources), and the average actual and estimated duration of submitted jobs are also given for each month. Furthermore, the resulting over-estimation factor and the performance of the three basic policies FCFS, SJF, and LJF measured in the average slowdown weighted by job area (SLDwA) are given. As already mentioned, the SLDwA metrics is used for measuring the overall performance of the simulated schedule.

Later we evaluate the performance of the self-tuning dynP scheduler at increased workloads. For that we generated synthetic job sets, which are based on the statistical properties of the original traces and they contain only 10,000 jobs each. With the shrinking factor the interarrival times between jobs are reduced and the workload is increased.

The scheduling process of a planning based resource management system is modelled in the simulation environment. As jobs are placed in the schedule as soon as possible, backfilling is implicitly done with all policies. Note, if we use the term slowdown for simplicity, we mean the average slowdown weighted by the jobs area (SLDwA).

At the beginning in Section 5.1 the original traces from Section 3.5 are used for the evaluation. As precise information concerning the start date and time of each trace is available, a monthly evaluation is done. In Section 5.2 the self-tuning dynP scheduler is evaluated with increased workloads. For that the shrinking factor and synthetically generated jobs, based on the traces, are used. In this chapter only major results are presented and the whole set of numbers is found in Appendix A.1.

5.1 Results Based on Original Traces

A first look at the overall results for the original traces in Table 5.1 shows, that the last four traces (PC2-2001, PC2-2002, CHPC, and MHPCC) are not well suited for an evaluation. The differences between the basic policies are only marginal and the slowdowns are close to the minimum of one. This means, that the corresponding schedules are empty and jobs do not have to wait a long time for their start.

However, scheduling strategies only influence waiting jobs by sorting them according to the policy. This in turn means, that if no or only some jobs have to be sorted, no difference in performances occurs. Evaluating the self-tuning dynP scheduler with such job sets does not make sense, as only a different performance of the basic policies induce the scheduler to

switch the policy. In contrast, the slowdowns for the first four traces (CTC, KTH, LANL, and SDSC) show differences between the three basic policies.

5.1.1 Basic Policies

We begin with presenting the results for the three basic policies. This shows which of the policies is the best choice for each job set. It is also observed, that the same policy is not the best choice for all applied workloads. The presented results are used as the reference case for the following evaluations.

In Table 5.1 the best basic policy is highlighted in blue. Particularly for the SDSC trace, the differences in slowdown are large as SJF is worse than FCFS by a factor of almost two and the same factor applies to the ratio of LJF to SJF.

	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
PC2-2001	1.0846	1.0758	1.0838
PC2-2002	1.0798	1.0765	1.1033
CHPC	1.0038	1.0037	1.0038
MHPCC	1.2362	1.2319	1.2353

Table 5.1: Overall average slowdown weighted by area (SLDwA) for the three basic policies FCFS, SJF, and LJF. Blue indicates the best policy for each trace.

CTC The monthly evaluation of the CTC trace in Table A.1 shows, that the first month should not be taken into account. The trace starts five days before the end of the month and only a few jobs are submitted in these days. Nevertheless, it is interesting that these few jobs request almost three times as many resources and are estimated to be twice as long as all remaining jobs in the other months.

In general, the CTC trace comes with a steady arrival process, which means, that the average job data does not change much over the months. Except for the last three months, where about 1,000 jobs less are submitted as in the months before. However, the jobs request more resources and are estimated to run longer. In all months the differences in the slowdown between FCFS and SJF are minimal and LJF is worse, by far. The performance in February 1997 is completely different, as LJF is better than SJF and almost competes with FCFS.

KTH SJF is the best basic policy throughout all months of the KTH trace (Table A.2). Again, the first month should not be taken into account as it only consists of seven days and some jobs. Although the submitted jobs of the KTH trace do not change much, three months are noticeable: in July 1997 the jobs are longer (almost by a factor of two) than before, both in the estimated and actual duration. This has no influence on the performance of LJF and the performance gap of the other two policies.

In November 1996 and February/March 1997 the performance of LJF is worse compared to the previous and following months by almost a factor of two. However, the performance difference between FCFS and SJF and the submitted job data does not change.

LANL The LANL trace (Table A.3) is different than the previous two. No clear winner between FCFS and SJF is found when the months are viewed independently. At the beginning of the traced time frame, SJF is clearly better than FCFS, by about 5%. Then from June 1995 to June 1996 the best policy continuously switches between SJF and FCFS, then finally FCFS becomes superior to SJF by about 5% in the last four months. The overall average shows, that FCFS is slightly better than SJF. This behavior makes the LANL trace very interesting for the evaluation process.

Note, the large amount of submitted jobs in July 1996. Almost three times as many jobs were submitted. And these jobs are estimated to be three times longer than in all other months. However, the over-estimation factor for this month is almost 12, which indicates, that the jobs actual duration was very short. Also note, the performance of LJF and SJF in the last four months of the trace increases significantly. It even outperforms SJF and becomes the second best policy.

SDSC Observing the best basic policy for each month in the SDSC trace (Table A.4), a similar behavior is found as with the LANL trace. At the beginning SJF is clearly the best policy. However, from July 1999 on FCFS is the best policy. The only major difference is in the width of the submitted jobs. It significantly increases, which probably induces this change.

Similar to the LANL trace FCFS is the best policy on average of all months. Noticeable is the submission behavior in September 1998. Almost three times more jobs are submitted in this month. At the same time the average width and duration is shorter. Hence, the workload (i. e. area of jobs) stays almost the same and the slowdowns do not increase much. In the second half of the trace the slowdown values increase dramatically and the performance of LJF is up to ten times worse than FCFS or SJF. Similar to the LANL trace LJF is better than SJF in the last three months. Overall, the behavior of the SDSC trace is quite similar to the LANL trace. Except that in the LANL trace only power-of-2 job widths larger than 32 exist whereas in the SDSC trace such a restriction does not exist.

PC^2 Both PC^2 traces have a much lower utilization than the previous four traces from the Parallel Workload Archive. Possible reasons could be the different types of machine (IBM SP2s vs. Linux-Clusters) and the machine usages. Furthermore, the *hpcLine* cluster of the PC^2 is not used as a production machine like the others. Hence, a steady and daily workload is not submitted and bursts of utilization occur over a period of time. Therefore, differences between the policies do not occur often. In only three months of the first year (Table A.5) one of the policies is superior to the others. In the second year (Table A.6) this happens only in a single month (March 2002). For all other months no best policy is found and the slowdown values for these month are close to the minimum of one.

However, in some months the data of submitted jobs is interesting. In September 2001 only a few jobs are submitted, the jobs are rather long and with this request many resources. In the second year large over-estimation factors occur from June to August 2002. This means, that jobs were estimated to run for a long time, but they end much earlier. In the following two months September and October 2002 the jobs were estimated to run longer and the estimation is more precise. Their average actual run time is almost 6 hours. Regardless of the differences in the submitted jobs the schedules are empty. Therefore, the policies generate an equal performance with slowdowns close to one. This means, that jobs do not have to

wait for their start. Only in March 2002 a significant workload is submitted and differences between the policies become visible. Nonexistent or very small differences in the policies already indicate, that a self-tuning dynP scheduler does not work with such workloads. As mentioned earlier the self-tuning dynP scheduler requires waiting jobs, which are then started in different orders. In both PC^2 traces the overall value from Table 5.1 shows, that almost no difference between the three basic policies exist, although SJF was slightly better for both traces.

CHPC As with the last two traces from the PC^2 the CHPC trace also contains no significant workload except for June 2000. Only in this month the slowdowns are different from their minimum of one and SJF is marginally better than FCFS and LJF. Note the large differences in the numbers and properties of the submitted jobs for different months. Almost all combinations of width and duration are found: small and short (November and December 2000), small and long (March and April 2001), large and short (May 2000), and large and long (June and July 2000). Despite this totally contrary submission behavior the policies can not generate different results, as the schedules are not loaded enough.

MHPCC A monthly evaluation for the MHPCC trace also makes no sense as the trace covers only one month (Table 3.5). Nevertheless, in this month enough jobs are submitted so that the achieved slowdowns are not as close to one as before. Therefore, the differences between the three policies are larger than before. Note, the 18 jobs submitted in February request a large number of resources (about 20% of the total machine).

Concluding Remarks

In the previous section we evaluated the performance of the three basic scheduling policies FCFS, SJF, and LJF with the original traces. Besides observing the overall results, we did a monthly evaluation which reveals the changing characteristics of the traces throughout their length. The evaluation of the traces CTC, KTH, LANL, and SDSC shows, that no clear winning policy is found. For the CTC and KTH trace SJF is the best choice with respect to the average slowdown weighted by the area (SLDwA), whereby for the LANL and SDSC trace FCFS is the best. For all traces LJF generates the worst slowdown performance. The results of the last four traces (PC2-2001, PC2-2002, CHPC, and MHPCC) show, that differences between the scheduling policies are only marginal. Due to low utilization and empty schedules, the scheduler starts new jobs directly without having to wait. However, SJF is the best basic policy for these four traces.

We evaluated the basic policies in order to get a reference for the following evaluation of the self-tuning dynP scheduler. Because of the low utilizations in the last four traces, different scheduling policies are not able to generate different schedules, hence evaluating the self-tuning dynP scheduler would be useless. Therefore, the four traces PC2-2001, PC2-2002, CHPC, and MHPCC are no longer observed in the following.

5.1.2 Advanced vs. Simple Decider and Half vs. Full Self-Tuning

At first, the slowdown performance of the simple and advanced decider is compared. For this initial evaluation the average response time weighted by width (ARTwW) is used as

self-tuning metrics. This is similar to using the average slowdown weighted by area (SLDwA) metrics. The comparison is either done for full and half self-tuning. After that, full and half self-tuning themselves are compared in their performance. The four traces CTC, KTH, LANL, and SDSC are used as job input. They proved to generate enough workload so that different results for the three basic policies occurred.

In Table 5.2 the slowdown results for the simple and advanced decider are presented. The percentages indicate the advantage of the advanced decider compared with the simple decider. One can 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. However, for the CTC, KTH, and LANL trace almost no differences occur whether half or full self-tuning is applied. For the SDSC trace and full self-tuning the difference between the two deciders is almost 70%. Unfortunately, in this case the advanced decider is not that good (in fact it is still worse than the best basic policy, cf. Table A.12), but rather the simple decider performs that worse. Although the performance with the SDSC trace stays behind the best basic policy. However, the switching behavior is much more interesting than with the CTC or LANL trace, where the self-tuning dynP scheduler outperforms the best basic policy.

	best policy		simple decider		advanced decider			
			half	full	half		full	
CTC	1.9277	SJF	2.3036	2.2834	1.9085	(+ 17.15%)	1.8809	(+ 17.63%)
KTH	2.5488	SJF	4.7256	5.6562	2.5812	(+ 45.22%)	2.5885	(+ 54.36%)
LANL	1.6801	FCFS	1.7538	1.7489	1.6101	(+ 8.19%)	1.6075	(+ 8.09%)
SDSC	6.8260	FCFS	13.3353	26.3414	10.0953	(+ 24.30%)	9.8325	(+ 62.67%)

Table 5.2: Comparison of the simple and advanced decider using ARTwW as self-tuning metrics and half or full self-tuning. The given values are the overall SLDwA performance.

According to Table 4.3 the differences between the two deciders appear in four cases. In case 1, 6b, and 8c the simple decider chooses FCFS as the new policy, whereas the advanced decider correctly stays with the old and current policy. The same applies to case 10c where the simple decider chooses SJF. Hence, the simple decider favors FCFS. The four cases are responsible for the large differences in the performance of both deciders.

As the difference between the simple and advanced decider is most prominent for the SDSC trace and with full self-tuning, a detailed case analysis was done. Table 5.3 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 112,606 (77.69%) of 144,942 total self-tuning decisions this situation occurs and the advanced decider stays with SJF. In contrast, the simple decider only runs 44.89% 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 policy like the advanced decider does. The simple decider arbitrarily favors FCFS. The other two cases 8c and 10c are not reached by the simple or advanced decider, hence they are not responsible for the different 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 choose the same policy (LJF) as their new policy.

case		combinations	simple decider	counted	advanced decider	counted
1		FCFS = SJF = LJF	**FCFS**	**10,437**	**old policy**	**17,657**
2		SJF < FCFS, SJF < LJF	SJF	70,543	SJF	996
3		FCFS < SJF, FCFS < LJF	FCFS	82	FCFS	55
4		LJF < FCFS, LJF < SJF				
	a	FCFS < SJF	LJF	20	LJF	13
	b	FCFS = SJF	LJF	2,121	LJF	10,719
	c	FCFS > SJF	LJF	19	LJF	5
5		FCFS = SJF, LJF < FCFS (⇔ LJF < SJF)	LJF	0	LJF	0
6		FCFS = SJF, FCFS < LJF (⇔ SJF < LJF)				
	a	old policy = FCFS	FCFS	383	FCFS	777
	b	old policy = SJF	**FCFS**	**69,868**	**SJF**	**112,606**
	c	old policy = LJF	FCFS	249	FCFS	1136
7		FCFS = LJF, SJF < FCFS (⇔ SJF < LJF)	SJF	0	SJF	0
8		FCFS = LJF, FCFS < SJF (⇔ LJF < SJF)				
	a	old policy = FCFS	FCFS	1,919	FCFS	978
	b	old policy = SJF	FCFS	0	FCFS	0
	c	old policy = LJF	**FCFS**	**0**	**LJF**	**0**
9		SJF = LJF, FCFS < SJF (⇔ FCFS < LJF)	FCFS	1	FCFS	0
10		SJF = LJF, SJF < FCFS (⇔ LJF < FCFS)				
	a	old policy = FCFS	SJF	2	SJF	2
	b	old policy = SJF	SJF	0	SJF	0
	c	old policy = LJF	**SJF**	**0**	**LJF**	**0**
		totally counted		155,642		144,942

Table 5.3: Case analysis for the SDSC trace and the simple vs. advanced decider. Full self-tuning is applied and ARTwW is used as self-tuning metrics.

The differences between the two deciders are also seen when focusing on the policy usage, i. e. the times the deciders switched to the policies and how many jobs were started with each policy as in Table 5.4.

		simple decider		advanced decider	
switches to each policy	FCFS	70,723	(45.44%)	1,137	(0.78%)
	SJF	70,545	(45.33%)	998	(0.69%)
	LJF	360	(0.23%)	1,136	(0.78%)
	no policy switch	14,014	(9.00%)	141,671	(97.75%)
job started with each policy	FCFS	39,554	(58.49%)	4,268	(6.31%)
	SJF	26,784	(39.61%)	52,590	(77.77%)
	LJF	1,282	(1.90%)	10,762	(15.92%)

Table 5.4: Comparison of the decision behavior and the usage of policies for the SDSC trace. Full self-tuning is applied and ARTwW is used as a self-tuning metrics.

If the advanced decider is applied almost 80% of all jobs are started by SJF and only a minority of 6% by FCFS. About 16% of the jobs were started with LJF, which is surprising as LJF is always the worst policy when the overall or monthly averages are evaluated (cf. Table A.12 and Table A.4). 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 (almost 98%). 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 (51% and 47% resp.) and the difference in the number of jobs started with FCFS and SJF is also considerably smaller than with the advanced decider. In only 10% 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 (roughly 2%) of all jobs are started with LJF.

The number of self-tuning calls and invocations of one of the deciders is larger for the simple decider (155,642) than for the advanced decider (144,942). 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).

Concluding Remarks

In the previous section we evaluated the slowdown performance of the self-tuning dynP scheduler. We compared the simple and advanced decider with half and full self-tuning applied. We only used half of the original traces, as the other half generates not enough workload, so that differences between the schedules do not occur. The results show, that the advanced decider is clearly better than the simple decider. A detailed case analysis shows the differences in the switching behavior of the two deciders. In general full self-tuning is superior to half self-tuning. Although only about half as many self-tuning calls are done with half self-tuning, the performance is only behind slightly. Therefore, if less self-tuning calls are intended, for example to reduce the switching behavior of the self-tuning dynP scheduler, and the remaining percentages of performance optimization are unimportant, half self-tuning is a good compromise.

5.1.3 Comparing Self-Tuning Metrics

Due to the fact that several metrics can be used in the self-tuning process for measuring the performance of the generated schedules, an evaluation focusing on this topic is to follow. The following metrics are compared: average response time (ART), average response time weighted by area (ARTwA), average response time weighted by width (ARTwW), makespan, average slowdown (SLD), average slowdown weighted by width (SLDwW), average slowdown per processor bound by 60 seconds (ppSLD_60).

Table 5.5 shows the overall slowdown performance (SLDwA) of the advanced decider with different self-tuning metrics and either half or full self-tuning. Clearly the ARTwW metrics

generates the best results among all metrics for all job traces. Equivalent to the above presented results full self-tuning is superior to half self-tuning. However, if the other metrics are observed, no clear tendency towards half or self-tuning is found. Except for ARTwW no further tendency is reported to say that one of the performance metrics always performs better with one of the self-tuning variants.

Nevertheless, the influence of the switching behavior on the performance of the self-tuning dynP scheduler is obvious. As the average slowdown weighted by area is used for comparing the overall performance, the use of makespan as self-tuning metrics always results in poor results. This is seen in Table 5.5 and especially for the LANL or SDSC trace. With both traces the slowdown values with makespan as self-tuning metrics are considerably higher than with any other metrics.

Comparing the best case (ARTwW with full self-tuning) with the best basic policy from Table 5.1 shows, that only with the CTC and LANL trace the self-tuning dynP scheduler is better. Note that the second best self-tuning metrics SLDwW is also better than the best basic policy for these two traces.

	quality metrics used for self-tuning						
self-tuning	ART	ARTwA	ARTwW	Makespan	SLD	SLDwW	ppSLD_60
CTC							
full	2.0417	2.2888	1.8809	2.4663	1.9752	1.8955	2.0787
half	2.0318	2.2997	1.9085	2.4675	1.9547	1.8919	2.0657
KTH							
half	3.1749	5.4043	2.5665	5.3979	2.6526	2.5686	3.7325
full	3.0742	4.4683	2.5812	5.5312	2.6579	2.5826	3.4218
LANL							
half	1.6746	1.7752	1.6075	2.0378	1.6602	1.6333	1.6927
full	1.6882	1.7568	1.6101	2.0357	1.6614	1.6332	1.7126
SDSC							
half	14.3471	18.9294	9.8325	25.1183	11.2207	10.8829	18.1592
full	13.6907	13.1993	10.0953	25.0682	11.8253	10.9371	19.4111

Table 5.5: SLDwA values for different self-tuning metrics used in the self-tuning dynP scheduler with the advanced decider. Green colors indicate whether full or half self-tuning is better, while the blue color indicates the best quality metrics for self-tuning.

A subsequent question is how the results change when an owner-centric metrics like e. g. the utilization is used to evaluate the overall performance? And which self-tuning metrics is the best choice now, possibly the makespan?

Unfortunately, it is not possible to answer the questions for all four traces with ease. For the CTC, KTH, and LANL the utilization does not change, regardless of whether half or full self-tuning is applied or if any different self-tuning metrics is used (cf. Table 5.6). Only the SDSC trace contains enough workload so that nonequal utilization values are generated. As expected the makespan metrics is the best choice for measuring the overall performance with the utilization. Comparing these results with Table 3.6 shows, that the utilization of the best basic policy FCFS is slightly less than with the advanced decider and makespan as the self-tuning metrics.

Finally the results from Table 5.5 show, that for all four relevant traces a single combination is always the best choice: ARTwW as self-tuning metrics and full self-tuning applied.

It seems to be obvious that ARTwW as self-tuning metrics performs best due to its depen-

self-tuning	quality metrics used for self-tuning: ART	ARTwA	ARTwW	Makespan	SLD	SLDwW	ppSLD_60
CTC							
half	for all combinations: 65.701%						
full							
KTH							
half	for all combinations: 68.716%						
full							
LANL							
half	for all combinations: 55.607%						
full							
SDSC							
half	81.739%	81.806%	81.763%	82.499%	81.719%	81.494%	81.532%
full	81.826%	82.081%	81.623%	82.467%	81.735%	81.469%	81.544%

Table 5.6: Utilization for different self-tuning metrics used in the self-tuning dynP scheduler with the advanced decider. Green colors indicate whether full or half self-tuning is better, while the blue color indicates the best quality metrics for self-tuning.

dence on the SLDwA. Especially as two similar metrics are used for measuring the overall performance and also for the measurement in the self-tuning process. However, the best possible performance is desired at each time of the scheduling process.

Evaluating the option on the invocation of self-tuning shows, that applying full self-tuning is best. In case the costs of scheduling and self-tuning are too high, half self-tuning is an option as only half of the self-tuning calls are omitted. However, this comes with a small decrease in overall performance. Hence, if the costs of scheduling are high, abandoning the last percent of performance optimization might be beneficial and half self-tuning should be applied.

However, 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 metrics, CTC trace, no slackness). Therefore, it is not necessary to apply half self-tuning.

Concluding Remarks

In the previous section we studied the influence of different performance metrics on the self-tuning process and the performance of the self-tuning dynP scheduler. It is seen that for all four observed traces the average response time weighted by width (ARTwW) generates the best results with regards to the slowdown performance. At all times, full self-tuning is the best choice. With other self-tuning metrics and other job traces, half self-tuning is better than full self-tuning in some cases, although a clear trend is not observed. If the performance is measured with the utilization, the makespan metrics is the best choice for the SDSC trace. For all other traces the same utilization is achieved, regardless of self-tuning metrics or whether half or full self-tuning is applied.

5.1.4 Preferred Decider

In the following the preferred decider is evaluated and its performance is compared to the advanced decider. With defining a preferred policy the decider only changes to a different policy, if the preferred policy is clearly worse. On the other hand the decider switches back

to the preferred policy, if its performance is at least equal to the current used policy. As for two of the traces FCFS is the best basic policy (LANL and SDSC) and SJF for the other two (CTC and KTH), the evaluation is done with a *FCFS-preferred* and *SJF-preferred* decider.

In Table 5.7 the slowdowns (SLDwA) are printed. It is seen that in general the FCFS-preferred decider is not able to outperform the best basic policies, and especially not FCFS for the LANL and SDSC trace. Although one expects, that the FCFS-preferred decider could improve the performance of FCFS, one has to consider the difference between the estimated and actual run time of jobs. The estimated run time is used for scheduling and therefore influences the decisions of the self-tuning dynP scheduler. As jobs are usually finished before their estimated run time is reached, schedules change unexpectedly. Hence, the previous made decision by the self-tuning dynP scheduler to switch away from the preferred policy may have been wrong. This is checked for the LANL and SDSC trace by making the job run times perfect, i. e. the estimated run time of every job is set to the value of the actual run time. As seen in the last rows of Table 5.7, unfortunately this is not the case for any of the two traces.

trace	FCFS	SJF	advanced	SJF-preferred	FCFS-preferred
CTC	2.0455	1.9277	1.8809	1.8762	2.2808
KTH	3.1015	2.5488	2.5665	2.5818	5.6985
LANL	1.6801	1.7031	1.6074	1.6275	1.7523
SDSC	6.8261	12.5662	9.8325	11.1882	26.3240
perfect estimates					
CTC	1.9006	1.6323	1.6317	1.6232	2.4083
KTH	2.9228	2.3823	2.3685	2.3524	5.8390
LANL	1.6139	1.5110	1.4711	1.4712	1.8471
SDSC	5.3995	6.5780	5.1924	5.4704	21.1062

Table 5.7: Comparison of the FCFS-/SJF-preferred decider with the advanced decider. The performance is measured in slowdown weighted by area (SLDwA). Full self-tuning is applied and ARTwW is used as the self-tuning metrics. Green colors indicate the best basic policy. If a self-tuning decider outperforms the best basic policy, blue colors are used. The best self-tuning decider is marked red.

Choosing SJF as the preferred policy results in a better performance. For the LANL trace the SJF performance is outstanding. Here FCFS is the best basic policy and although choosing SJF as the preferred policy the self-tuning dynP scheduler with the preferred decider is better. Even more interesting is the fact that 88.6% of all jobs are started with SJF, 11.4% with LJF and surprisingly no jobs at all with FCFS.

Concluding Remarks

The preferred decider intentionally prefers a single policy and is therefore unfair with its decision process. We evaluated the performance in the case that SJF or FCFS is used as the preferred policy. The comparison with the advanced decider shows, that choosing FCFS as the preferred policy is not beneficial. However, choosing SJF as the preferred policy improves the performance for the CTC trace. With all other traces the advanced decider is always superior. Unfortunately, the preferred decider does not deliver the expected performance, although it was especially designed to improve the best basic policy for a given job trace.

5.1.5 Slackness

As previously seen the self-tuning dynP scheduler can not outperform the performance of the best basic policy for all traces (especially not for KTH). This can be induced by the following scenario: Assume that the currently used policy is SJF and a set of new very long jobs are submitted at the same time. With these new jobs the schedule changes, which results in the self-tuning dynP scheduler's decision to switch the policy to LJF. By switching the policy these jobs are started immediately. Now assume that the actual run time of these new jobs is rather short. Hence, it might have been better to stay with the old policy SJF.

The slackness option for the self-tuning dynP scheduler is used to avoid such scenarios and to reduce the number of unnecessary policy switches (Table 5.4). The slackness is specified in percent, i. e. the percentage specifies how much better than the currently used policy the new policy has to be. In other words: the performance of the current policy is virtually improved by the given percentage and the decider mechanism uses this new value for comparing the policies.

In Table 5.8 the slowdown performance is given for slackness values of 2, 4, 6, 8, and 10%. The grey column with 0% indicates, that no slackness is used. These results for 0% slackness are taken from Table 5.2 and Table 5.7. If the results for the simple and advanced decider and slackness values $> 0\%$ are compared, it is seen that the results are exactly the same. And this holds for all traces. The question is why?

			slackness in percent					
	FCFS	SJF	0	2	4	6	8	10
			simple decider					
CTC	2.0455	1.9277	2.2834	1.8480	1.8787	1.9100	1.8903	1.9018
KTH	3.1015	2.5488	5.6562	2.5819	2.5672	2.5719	2.5470	2.5493
LANL	1.6801	1.7031	1.7489	1.6391	1.6304	1.6596	1.6760	1.6878
SDSC	6.8261	12.5662	26.3414	11.1859	10.7924	12.5575	12.0984	12.1065
			advanced decider					
CTC	2.0455	1.9277	1.8809	1.8480	1.8787	1.9100	1.8903	1.9018
KTH	3.1015	2.5488	2.5665	2.5819	2.5672	2.5719	2.5470	2.5493
LANL	1.6801	1.7031	1.6075	1.6391	1.6304	1.6595	1.6760	1.6878
SDSC	6.8261	12.5662	9.8325	11.1859	10.7924	12.5575	12.0984	12.1065
			SJF-preferred decider					
CTC	2.0455	1.9277	1.8762	1.8641	1.9000	1.8946	1.9012	1.9198
KTH	3.1015	2.5488	2.5818	2.5450	2.5402	2.5385	2.5479	2.5543
LANL	1.6801	1.7031	1.6275	1.6706	1.6869	1.6983	1.7005	1.7065
SDSC	6.8261	12.5662	11.1882	11.6090	10.6430	12.5653	12.1149	12.1111

Table 5.8: Comparison of different slackness values for different decider mechanisms. The performance is measured in slowdown weighted by area (SLDwA). Full self-tuning is applied and ARTwW is used as the self-tuning metrics. Green indicates the best basic policy. If a self-tuning decider outperforms the best basic policy, blue colors are used. The best self-tuning decider is marked red.

According to Tables 4.3 and 5.3 the differences between the simple and advanced decider are the cases 1, 6b, 8c, and 10c:

- case 1 (all policies are equal): If the performance of the old policy is virtually improved, the old policy is clearly better than the other two policies. Hence, the old policy is retained and this case no longer appears in the decision process.

- case 6b (FCFS and SJF are equal, LJF is clearly worse, SJF is the old policy): With slackness applied, the performance of SJF is virtually improved and is therefore better than the performance of FCFS. Hence, SJF is retained and this case no longer appears.
- case 8c: similar to case 6b, but with LJF instead of SJF. LJF is retained.
- case 10c: similar to case 6b, but with SJF and LJF. LJF is retained.

Therefore, using any amount of slackness reduces the number of cases observed in the advanced decider. Its choice of decisions is reduced to that of the simple decider.

Observing each of the traces in detail shows, that for the CTC trace slackness is useful. With any of the evaluated slackness values a better slowdown is achieved than with the best basic policy SJF. For the KTH trace a considerable amount of slackness (8%) has to be chosen in order to outperform the best basic policy. If the LANL trace is used, 0% already achieves the best performance and slackness is not needed. As previously observed the self-tuning dynP scheduler does not outperform the best policy FCFS, if the complete SDSC trace is used. However, by evaluating only the first 15 months of the traces (until June 1999), SJF is the best basic policy and minimal slackness values of 2-4% are beneficial.

Table 5.9 shows the policy usage for the CTC trace. As previously mentioned the simple and advanced decider are equal when slackness is used. If slackness is used with the advanced decider it can be seen that less switches to each of the policies are made, but the numbers do not change much. Without slackness, the advanced decider decided in 97.5% of all cases, that a policy switching is not necessary. Applying 2% of slackness to the advanced decider, increases this number to 99.3%. This shows the aim of using slackness, as the decider is more stable in its decisions. The decider does not switch the active policy that often and continuous with the currently used policy. The numbers of jobs started with each policy also do not change much, although with slackness even more jobs are started with FCFS and less with SJF and LJF.

slackness 0%		simple decider		advanced decider	
switches to each policy	FCFS	39,613	(35.19%)	795	(0.74%)
	SJF	40,006	(35.53%)	964	(0.89%)
	LJF	625	(0.56%)	963	(0.89%)
no policy switch		32,331	(28.72%)	105,386	(97.48%)
job started with each policy	FCFS	16,678	(21.04%)	49,422	(62.34%)
	SJF	58,538	(73.84%)	10,305	(13.00%)
	LJF	4,063	(5.13%)	19,552	(24.66%)

slackness 2%		simple decider		advanced decider	
switches to each policy	FCFS	253	(0.23%)		
	SJF	275	(0.25%)		
	LJF	274	(0.25%)		
no policy switch		108,187	(99.27%)		
job started with each policy	FCFS	60,521	(76.34%)		
	SJF	6,831	(8.62%)		
	LJF	11,927	(15.04%)		

Table 5.9: Comparing the switching behavior of the simple and advanced decider with and without slackness for the CTC trace. Full self-tuning is applied and ARTwW is used as self-tuning metrics.

Without slackness the simple decider switches back and forth between FCFS and SJF most of the time and in less than one third of all decisions no policy switching is made. Almost three-quarters of all jobs are started with SJF and 20% by FCFS. If slackness is applied, the simple mutates into the advanced decider and the policy usage changes completely. Almost no jobs are started with SJF now, and in most of the decisions no policy switching is made.

Concluding Remarks

Applying slackness to the decision process improves the performance of the currently in use policy. Therefore, such scenarios no longer occur where policies achieve an equal performance compared with the current policy and the decider switches the policy. A detailed analysis of the switching behavior shows, that with slackness applied the advanced decider is reduced in its decisions with the simple decider. This is like cases where two policies are equal.

The evaluation results for all four traces show, that everything depends on the trace, if, and how much, slackness is beneficial. The CTC and KTH trace benefit from small slackness values, regardless of which decider mechanism is used. Especially for the LANL trace and the advanced decider it is seen that slackness degrades the performance, although the results are still better than the best basic policy of up to 8% of slackness.

5.2 Results Based on Increased Workload

In the previous section the self-tuning dynP scheduler was evaluated with original traces. Due to the characteristics of the PC2-2001, PC2-2002, CHPC, and MHPCC traces, the submitted workload is rather low. Therefore, the schedules are empty and most of the jobs are started without a wait. As a result, the final slowdown (SLDwA) performance is close to the minimum value of 1 (Table 5.1). If the schedule is empty, different scheduling strategies are equal in their performance as different sorting criteria for the waiting jobs make no sense. In consequence these four traces were not used previously.

In this section, these four traces are used again, as the performance at increased workloads is now evaluated. Different approaches for increasing the workload exist: submitting more jobs (e. g. each job two or three times), increasing the jobs resource usage (e. g. by extending the estimated and actual run time), or submitting the same number of jobs at a faster rate. Here we use the last mentioned approach.

The submit time of jobs is given in seconds relative to the first job submit. By multiplying every submit time with a shrinking factor smaller than one, jobs are submitted faster. Table 5.10 shows the shrinking factors we use for the eight job sets. The last four job sets need smaller shrinking factors as their original utilization is much lower. The smaller shrinking factors are chosen in order to achieve utilizations close to 100%, i. e. the saturated state is reached. If the shrinking factor is set to zero, all jobs are submitted at once and off-line scheduling is done.

We use synthetic job sets instead of the original traces as the size of the traces (i. e. number of jobs) is either not sufficient (e. g. MHPCC, only one month) or too large (e. g. LANL or SDSC with two years). Of course the synthetically generated jobs have the same basic characteristics as the original traces. Hence, it is possible to simulate fewer jobs. In the following, we use synthetic job sets with 10,000 jobs. To exclude singular effects resulting from the generation process, ten synthetic job sets with 10,000 jobs are generated for each trace and are used as input for the simulations. After the simulation and analysis of each

	shrinking factors
CTC	1.0, 0.9, 0.8, 0.7, 0.6
KTH	1.0, 0.9, 0.8, 0.7, 0.6
LANL	1.0, 0.9, 0.8, 0.7, 0.6
SDSC	1.0, 0.9, 0.8, 0.7, 0.6
PC2-2001	1.0, 0.9, 0.8, 0.7, 0.6, 0.5, 0.4
PC2-2002	1.0, 0.9, 0.8, 0.7, 0.6, 0.5, 0.4
CHPC	1.0, 0.9, 0.8, 0.7, 0.6, 0.5, 0.4, 0.3
MHPCC	1.0, 0.9, 0.8, 0.7, 0.6, 0.5, 0.4, 0.3

Table 5.10: Used shrinking factors.

job set, the results are combined. This is done by neglecting the largest and smallest result and the average is computed from the remaining eight values. Thereby, singular effects (both good and bad) are not considered in the final values that are presented in the following.

As performance numbers for different workloads are presented in the following, simple tables are no longer sufficient to show all results. Therefore, we mostly use diagrams for presenting the results, as the schedulers performance can be observed over a wide range of workloads. The appendix contains the corresponding numbers.

Again, the evaluation starts with the basic policies. The evaluation of the self-tuning dynP scheduler and all its options are shown in a condensed format. Those configurations, which already proved to be worse in the evaluation with the original traces, are left out.

5.2.1 Basic Policies

As with the original traces, the evaluation starts again with a comparison of the three basic policies FCFS, SJF, and LJF. In the following and Appendix A.2 diagrams and tables with detailed results can be found. Three types of diagrams exist: slowdown on the y-axis and shrinking factor on the x-axis (Figure 5.1), utilization on the y-axis and shrinking factor on the x-axis (Figure 5.2), and a combination of the first two diagrams slowdown on the y-axis and utilization on the x-axis (Figure A.1). In Table A.13 and Table A.14 the corresponding numbers are shown.

User and owner centric performance metrics are contrary just like the slowdown and utilization are contrary. That is, only one of the two is typically optimized with a scheduling strategy. As commonly known LJF increases the utilization of a machine and SJF generates short response times and slowdowns. This is seen in the two diagrams for slowdown (Figure 5.1) and utilization (Figure 5.2). The green curve for LJF is always the highest of all. Note, high utilization values are good, but high slowdown values are bad. Hence, LJF generates the best utilization and the worst slowdown performance. Observing the blue curve for SJF in the same two diagrams shows the opposite behavior as both curves are the lowest. A good slowdown performance is achieved with SJF, but at the price of poor utilizations. In general, the difference between SJF and LJF in utilization for high workloads is 10 percentage-points and more.

FCFS seems to be a good compromise. The utilization is considerably higher than with SJF and almost as high as with LJF. At the same time the slowdown is close to SJF and in some cases (especially for CTC and SDSC) FCFS generates even smaller slowdowns than SJF. Hence, FCFS is a good reference for comparing the slowdown and utilization with the self-tuning dynP scheduler.

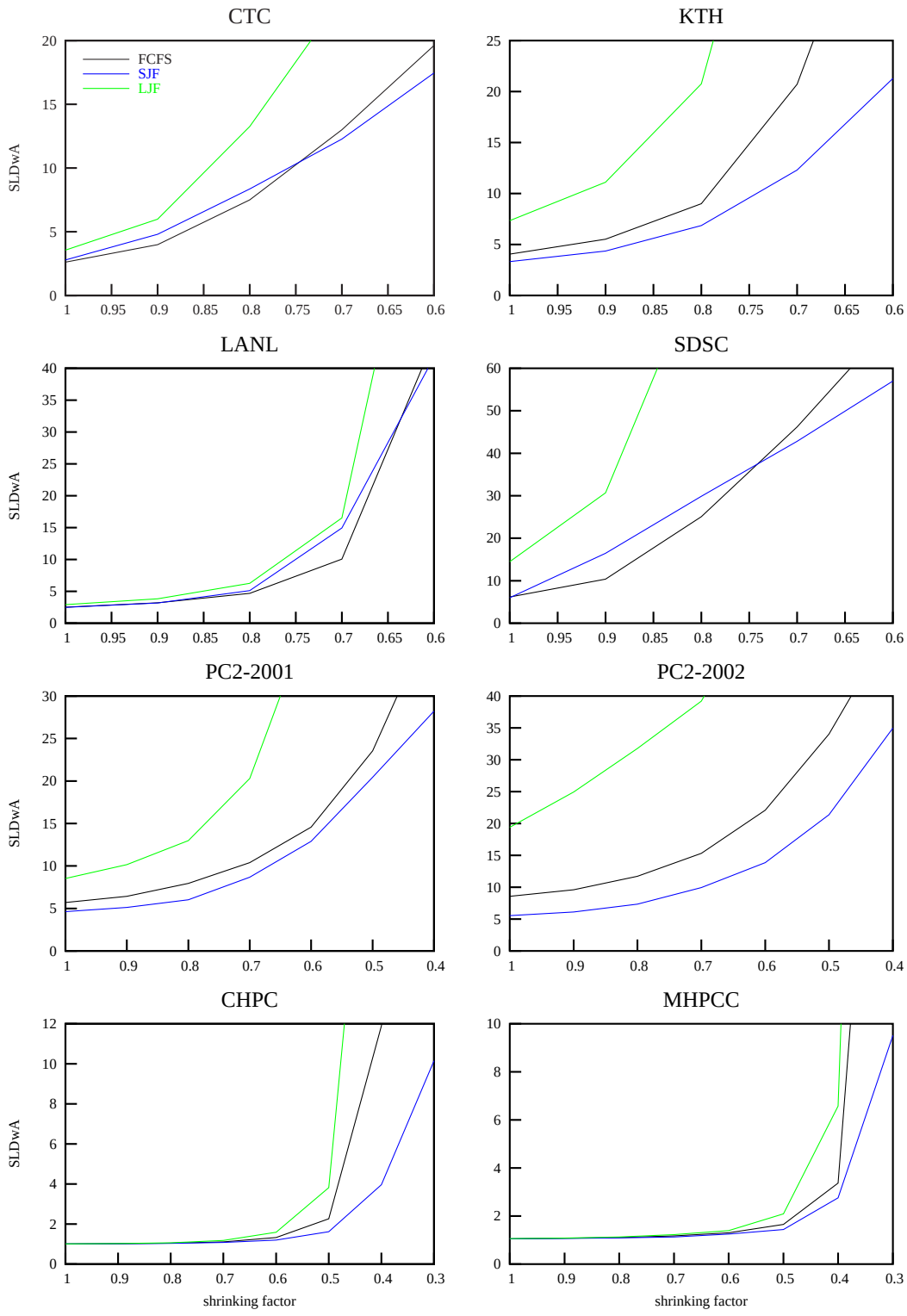


Figure 5.1: Average slowdown weighted by area (SLDwA) of FCFS, SJF, and LJF with different workloads/shrinking factors.

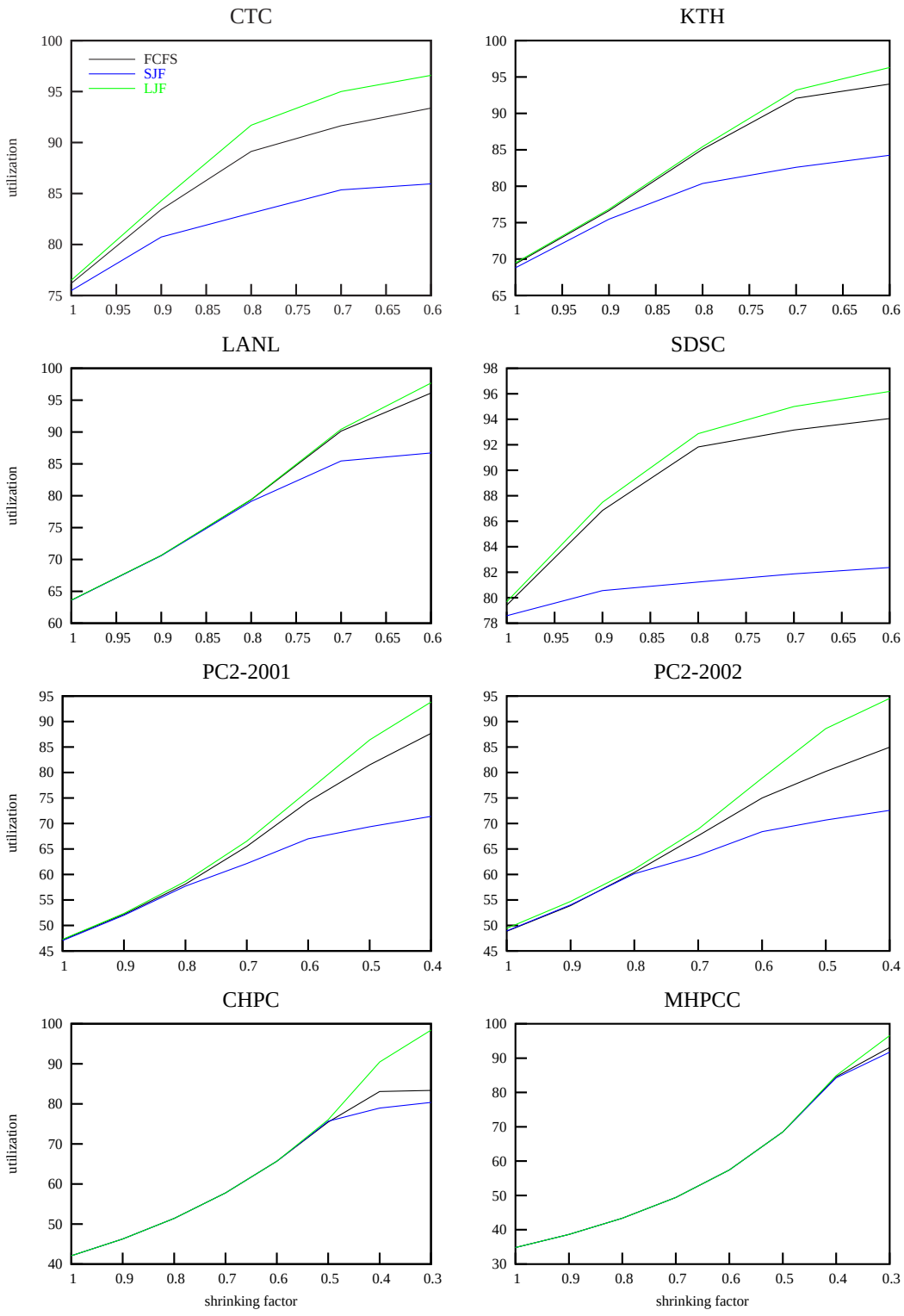


Figure 5.2: Utilization of FCFS, SJF, and LJF with different workloads/shrinking factors.

Therefore, Table A.13 and Table A.14 have additional columns where the difference of SJF and LJF to FCFS is given. For the slowdown the relative difference is printed as a percentage, whereas for the utilization the absolute difference is printed in percentage points of utilization. A negative difference indicates a poorer performance (i. e. higher slowdown or lower utilization). The table also contains average values for the differences of all shrinking factors. Although averaging over different workloads, this value summarizes the performance difference reasonably well.

For the first four job sets (CTC, KTH, LANL, and SDSC) in Table A.13 it can be seen that, except for the KTH based jobs, the overall average slowdown of SJF is worse than FCFS. Additionally, the utilization is worse, even for KTH. This is different for the other four job sets (PC2-2001, PC2-2002, CHPC, and MHPCC) in Table A.14 which originally comes with a low workload. Here SJF is at least 15% better, averaged of all shrinking factors. For all eight job sets the utilization benefit of LJF compared with FCFS is small (at most 3.6 percentage-points).

Table 5.11 shows a comparison of FCFS and SJF using both the original traces and the synthetic job sets with a shrinking factor of 1. In some cases (e. g. SDSC, PC2-2001, and PC2-2002) the slowdown values with the synthetic jobs are smaller (e. g. the SJF value for SDSC is halved and is smaller than the FCFS value). In other cases (e. g. CTC, KTH, and LANL) the slowdowns are greater for the synthetic jobs. This is probably induced by computing the average and leaving out the minimum and maximum value, or this is a result of the analysis of the original trace and the generation of new job sets. Due to the randomized generation less awkward jobs occur, which smoothes out the schedule and generates different results.

	original traces			synthetic jobs		
	FCFS	SJF	LJF	FCFS	SJF	LJF
CTC	2.0455	1.9277	2.5212	2.6086	2.7838	3.5543
KTH	3.1015	2.5488	5.8118	4.0572	3.3198	7.3312
LANL	1.6801	1.7031	2.0507	2.5263	2.4729	2.9228
SDSC	6.8260	12.5662	26.8207	6.1586	6.0003	14.4946
PC2-2001	1.0846	1.0758	1.0838	5.6913	4.6235	8.5279
PC2-2002	1.0798	1.0765	1.1033	8.5661	5.5526	19.4095
CHPC	1.0038	1.0037	1.0038	1.0091	1.0088	1.0115
MHPCC	1.2362	1.2319	1.2353	1.0541	1.0507	1.0640

Table 5.11: Comparison of the slowdown (SLDwA) for both the original traces and the synthetic job sets (with shrinking factor 1.0). Blue color indicates which of the basic policies is the best. Those cases where a different policy is best for the original traces and synthetic jobs are marked red.

Concluding Remarks

In this section we extended the evaluation to increased workload scenarios. We used synthetically generated job sets which are based on the original traces. As a result all eight traces are re-usable. By decreasing the average interarrival time the jobs are submitted over a shorter time, hence the scheduler has to process more workload. Additionally to the slowdown, we also studied the utilization. At very high workloads the system runs in a saturated state, in which a further increase of workload does not yield to a higher utilization. The evaluation results show, that LJF generates the highest possible utilization, although this comes with

immense slowdown values. In contrast, SJF achieves good slowdown values, but at the cost of poor utilizations. A good compromise is FCFS, as the achieved slowdown values are close to SJF while the utilization is closer to LJF.

5.2.2 Self-Tuning dynP Scheduler

Secondly, the performance of the self-tuning dynP scheduler is evaluated. In the previous section SJF proved to be the best basic policy for the user-centric slowdown metrics. Therefore, the performance of the self-tuning dynP scheduler is compared with SJF during the following evaluation. The aim is to have a better overall performance, i. e. a smaller slowdown at an increased utilization.

In this evaluation we focus on the advanced and SJF-preferred decider. According to previous evaluation results, the simple decider is left out as it already proved to be worse. Again, ARTwW is used as the self-tuning metrics and in the first step no slackness is applied.

The detailed evaluation results and corresponding diagrams can be found in the appendix (Table A.15, Table A.16, Figure 5.3, Figure 5.4, and Figure A.2). Additional to the slowdown and utilization results for each shrinking factor, the difference to SJF is also shown. For the slowdown metrics (SLDwA) the relative difference in % of the SJF performance, and for the utilization the absolute difference in percentage-points is presented. Finally averages of all used shrinking factors are computed in order to join all results in a single value. The value is then used as a comparison. This is done because of the scaling of the y-axis in the diagrams. Therefore, small differences are difficult to spot. The average values are also shown in Table 5.12 in a condensed form.

	SLDwA: relative difference to SJF in %		**utilization:** absolute difference to SJF in percentage-points	
	advanced	SJF-preferred	advanced	SJF-preferred
CTC	9.04	9.92	1.22	1.21
KTH	0.15	-0.72	0.13	0.12
LANL	1.51	1.29	0.07	0.09
SDSC	6.36	6.22	0.93	0.91
PC2-2001	0.27	0.69	0.09	0.11
PC2-2002	-0.60	-0.32	0.39	0.41
CHPC	-1.46	-0.61	0.16	0.07
MHPCC	2.22	1.24	0.04	0.03

Table 5.12: Difference in slowdown and utilization of the self-tuning dynP scheduler compared with the best basic policy SJF. ARTwW is used as self-tuning metrics and no slackness is applied. Shown are the average values of all workloads/shrinking factors.

It can be noticed that for all job sets the self-tuning dynP scheduler achieves a better utilization, although in some cases only slightly. Nevertheless, the average increase of 1.22 percentage-points with the CTC is good, especially at higher workloads. For example, at a shrinking factor of 0.6 (cf. Table A.15), SJF's utilization of 85.94% is increased by 1.45 percentage-points to 87.39% with the advanced decider. The difference appears to be small, but at high utilizations even small improvements are difficult to achieve. Observing the amount of resources that remain unused, so $1 - \mathit{utilization}$, clarifies this. A utilization of 85% means, that 15% of the resources remain unused. Increasing the utilization by 1.5

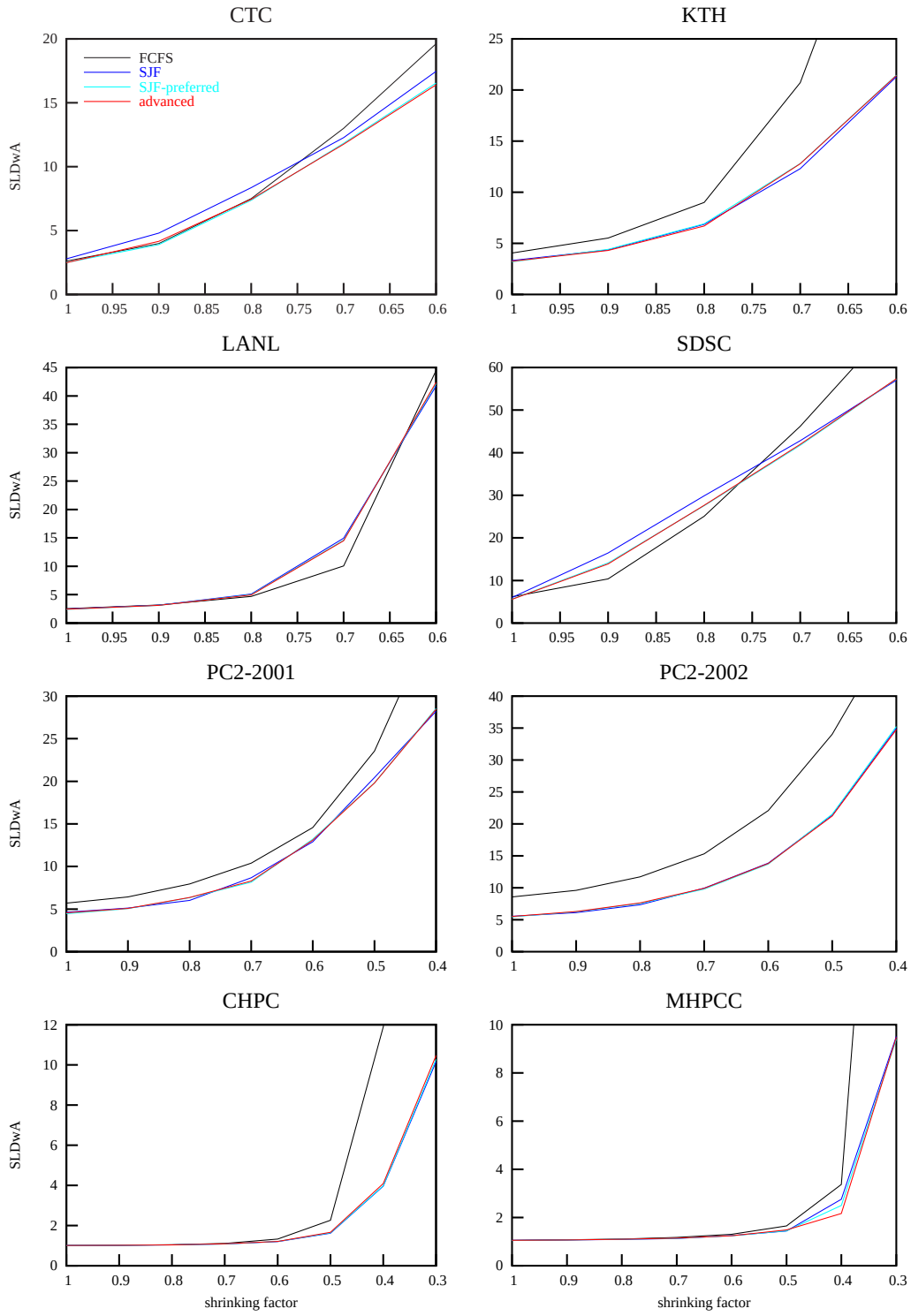


Figure 5.3: Average slowdown weighted by area (SLDwA) of the self-tuning dynP scheduler with different workloads/shrinking factors. ARTwW is used as the self-tuning metrics and no slackness is used.

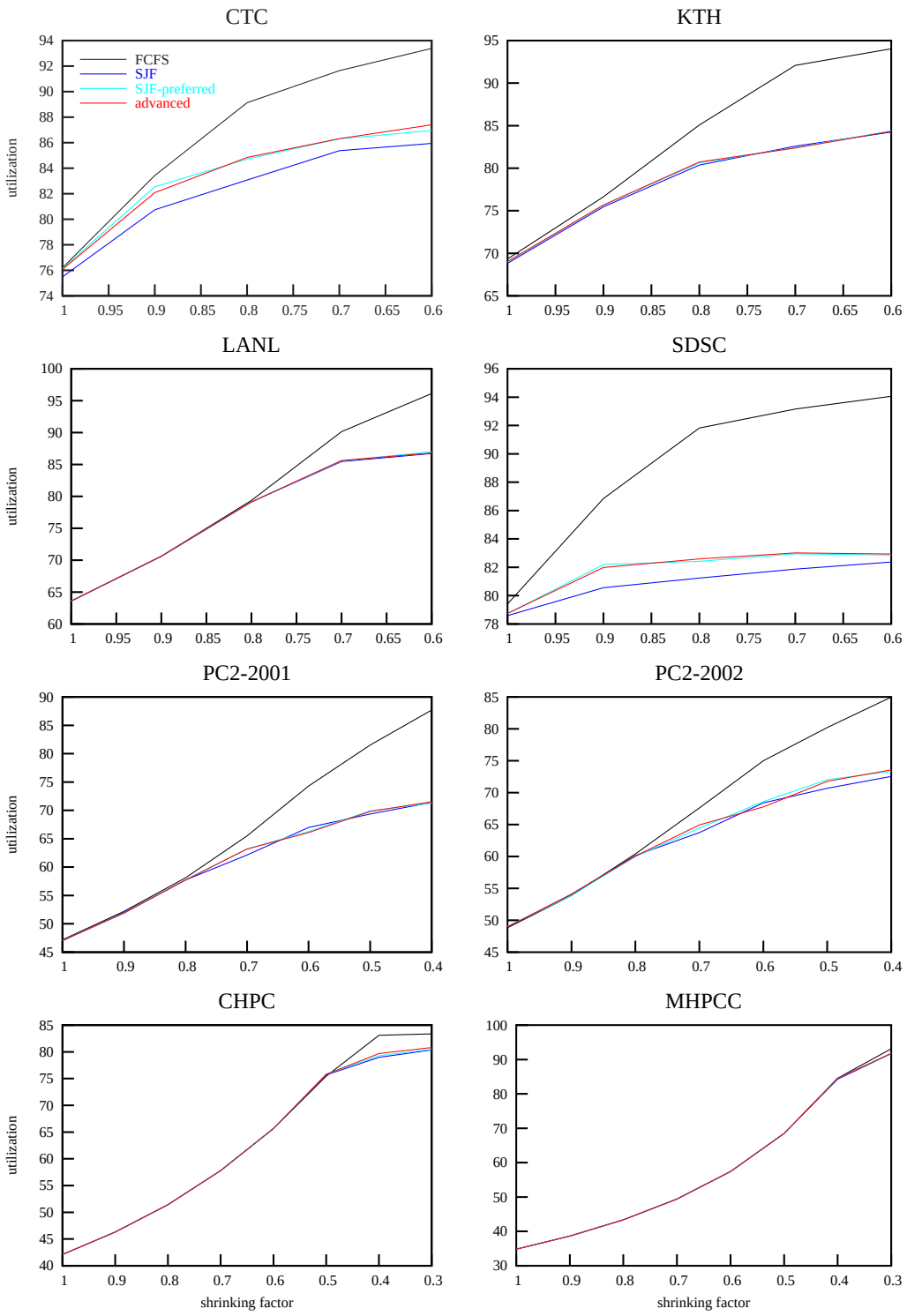


Figure 5.4: Utilization of the self-tuning dynP scheduler with different workloads/shrinking factors. ARTwW is used as the self-tuning metrics and no slackness is used.

percentage-points means at the same time, that the amount of unused resources is reduced by one tenth, which is good.

In general, improving the utilization at small workloads is not a hard task because the schedule is empty. If the utilization is already considerably high (e. g. when the system is close to the saturated state), even small improvements are difficult to achieve, as the schedule is quite packed. The results show, that differences between the advanced and SJF-preferred decider are large enough that one of the deciders is always better than the other. However, if all job inputs are observed, none of the deciders is superior. Hence, a general recommendation can not be made.

If the slowdown metrics is observed, the large improvements for the CTC (almost 10%) and SDSC based synthetic job sets are obvious. These improvements are also seen in Figure 5.3, especially the CTC performance shows the desired behavior of the self-tuning dynP scheduler. At the beginning with low workloads FCFS is the best basic policy and the advanced decider (red curve) follows it. Later (from 0.8 to smaller shrinking factors) the advanced decider follows the blue curve of SJF, as now SJF is better than FCFS. Furthermore, at high workloads the advanced decider achieves a better performance than SJF. Unfortunately, with PC2-2002 and CHPC based job sets the self-tuning dynP scheduler is worse than SJF, although the performance loss is not large: less than 0.7% (SJF-preferred) and 1.5% (advanced) for the slowdown metrics.

The same applies to the utilization, as it is difficult to favor one of the two deciders. Both deciders are worse than SJF for PC2-2002 and CHPC. Interesting is the slowdown performance with the KTH based synthetic jobs, as the advanced decider is better than SJF and the SJF-preferred decider is worse. However, the presented behavior is similar to Table 5.8, where the advanced decider does not outperform SJF, except if a slackness value of 8% is applied.

Again, slackness is applied in order to reduce the number of policy switches and thereby possibly improve the performance. In those cases, where two policies are equal and one of them is the current policy, slackness may be beneficial. Confer to Table 4.3 for these cases. The performance of the current policy is virtually improved, so that policy switching is avoided. Again, simulations with the advanced and SJF-preferred decider and slackness values of 2, 4, 6, 8, and 10% are driven. All results can be found in Tables A.17 - A.20. Additional to the slowdown values the relative difference to SJF is given. From these relative differences an average value using all shrinking factors can be computed, which allows us to get a comprehensive overview on the performance. Table 5.13 gives a comprehensive overview with the average values. The grey columns in the tables represents no use of slackness and these numbers are also found in Table A.15 and Table A.16.

The results show, that for most job sets slackness is only useful, if small values in the range of 1 to 2% are used. For CTC based synthetic jobs the best performance difference of almost 10% is achieved without slackness. With a slackness of 2 or 4%, the performance is already significantly lower for both deciders. A similar behavior can be observed for SDSC based jobs. The advanced decider is less vulnerable to slackness for these two jobs sets. The detailed numbers in Tables A.17 - A.20 show, that for specific combinations of slackness and shrinking factors extremely bad results are generated, while changing one of both parameters in any direction changes the performance difference significantly. It would have been expected that the performance changes more continuously, especially with increasing slackness. With the KTH based job input a slackness of 2% already allows both deciders to be superior to SJF,

synthetic jobs based on	slackness in % 0%	2%	4%	6%	8%	10%
advanced decider						
CTC	9.04%	7.24%	2.78%	-0.14%	-0.72%	-0.38%
KTH	9.92%	2.06%	0.48%	0.71%	0.36%	0.51%
LANL	1.51%	1.40%	0.83%	0.33%	0.03%	0.02%
SDSC	6.36%	2.69%	0.47%	-0.20%	0.30%	0.13%
PC2-2001	0.27%	-0.14%	-0.85%	0.07%	-0.08%	0.47%
PC2-2002	-0.60%	0.84%	0.96%	0.75%	0.59%	0.36%
CHPC	-1.46%	0.09%	0.14%	0.01%	-0.01%	-0.10%
MHPCC	2.22%	1.56%	0.43%	0.45%	0.31%	0.22%
SJF-preferred decider						
CTC	9.92%	2.06%	0.48%	0.71%	0.36%	0.51%
KTH	-0.72%	0.91%	0.75%	0.34%	0.24%	0.36%
LANL	1.29%	1.48%	0.34%	0.01%	-0.13%	-0.17%
SDSC	6.22%	0.81%	1.10%	0.23%	0.12%	0.03%
PC2-2001	0.69%	-0.15%	-1.19%	-0.17%	0.06%	-0.03%
PC2-2002	-0.32%	-0.07%	-0.05%	0.19%	0.19%	0.39%
CHPC	-0.61%	-0.07%	-0.03%	-0.03%	-0.05%	-0.05%
MHPCC	1.24%	-0.68%	0.66%	0.11%	0.10%	0.07%

Table 5.13: Slowdown as a relative difference compared to the basic policy SJF. Full self-tuning is done and ARTwW is used as self-tuning metrics. Shown are average values of all workloads/shrinking factors. Red colors indicate the best slackness value for each job set.

whereas with no slackness the SJF-preferred decider is worse than SJF. Table A.17 shows, that the advanced decider gets worse than SJF from slackness values of 6% increasing. At the same time the SJF-preferred decider still remains superior to SJF and the advanced decider, although it is significantly worse than without any slackness at all. All other job sets do not show such behavior.

For the remaining four job sets no general behavior is observable and possible improvements are insignificant. For the advanced decider with PC2-2001 and the SJF-preferred decider with PC2-2002 higher slackness values are beneficial and induce a performance gain over SJF. Here 10% slackness allows a 0.5% performance gain. While for all other results no performance gain can be found. In some cases (CHPC with the SJF-preferred decider) the self-tuning dynP scheduler always stays behind SJF, and in other cases slackness only decreases the performance.

In combination with Table 5.8 it can be noted that slackness should not generally be used with higher workloads. The evaluation of slackness and half vs. full self-tuning shows, that the abilities of the self-tuning dynP scheduler in switching the scheduling policy should not be limited in any way. Neither reducing the number of chances for a policy switching, nor virtually increasing the performance of the current policy increases the performance. Hence, the original idea of switching the policy whenever possible, achieves the best results.

Concluding Remarks

We evaluated the self-tuning dynP scheduler at increased workloads in the previous section. We focused on the advanced and SJF-preferred decider, full self-tuning, and ARTwW as the self-tuning metrics. We compared the performance of the self-tuning dynP scheduler with

SJF, computed relative differences for the simulated workloads, and averaged the relative differences. The results show, that for all job sets the self-tuning dynP scheduler achieves a better utilization, although in some cases only slightly. Improving the utilization at small workloads is not challenging. However, at high workloads and utilizations this becomes a hard task for the scheduler and even small improvements are harder to achieve. Especially for the CTC based job sets the performance of the self-tuning dynP scheduler is remarkable. The utilization is increased by 1.22 percentage-points and additionally the slowdown values are reduced by 9% of all applied workloads. The self-tuning dynP scheduler improves two contradicting performance metrics. Applying slackness to the advanced and SJF-preferred decider shows, that at increased workloads only very small slackness values up to 2% are useful to further improve the performance.

5.3 Summary

In the previous two chapters we presented the self-tuning dynP scheduler, its basic idea, different deciders, options, and evaluation results. 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 current waiting jobs. At the beginning we used a lower and upper bound for this decision. The average estimated run time of all waiting jobs was compared with two bounds and either FCFS, SJF, or LJF was chosen. We restricted the set of possible scheduling policies to the mentioned three, as they are implemented in the resource management system CCS. The evaluation of the dynP scheduler with bounds shows, that reasonable good results are achieved over a wide range of applied workloads. However, a major drawback is, that the bound setting needs to be adapted with different job characteristics. Using such a scheduler in a real world scenario is almost useless, as there is no time for a permanent re-adaptation of the bounds.

Hence, a major aim whilst developing the self-tuning dynP scheduler was to establish autonomy from job induced parameter changes. The scheduler switches the scheduling policies without the need of a permanent intervention of the system administrator. With the ability of future resource scheduling like planning based resource management systems do, the idea of the self-tuning dynP scheduler is as follows: The self-tuning dynP scheduler generates full schedules for each available policy, measures the generated schedules with a performance metrics, and then chooses the best policy.

A decider mechanism is in charge of choosing the best policy according to a performance metrics. We presented different levels of sophistication for the decider mechanisms. As its name already implies, the simple decider is simply coded and specific policies are intentionally preferred. The advanced decider is truly fair in its decisions and does not favor any policy, unless the decider has to arbitrarily choose between policies. In contrast, the preferred decider stays with a preferred policy and only switches to a different policy, if it is clearly better than the preferred policy. Additional options for the self-tuning dynP scheduler exist: Is self-tuning invoked only when jobs end (half self-tuning), or at job submits and ends (full self-tuning)? With slackness applied, the performance of the current policy is virtually improved for the comparison.

The evaluation of the self-tuning dynP scheduler started with a comparison of the three basic policies FCFS, SJF, and LJF as the reference. Eight traces from HPC machines are used

as job input and the performance is measured by the slowdown weighted by area (SLDwA). Furthermore, the results show, that half of the traces do not generate meaningful results, as the corresponding schedules are that empty, that the slowdown is close to its minimum of one. Scheduling policies can not generate different results, if the workload is low and there are many holes in the schedule. Therefore, we focused on the traces CTC, KTH, LANL, and SDSC. It is observed, that SJF achieves the lowest slowdown values for the CTC and KTH trace, while FCFS is best for LANL and SDSC. Again, the results show, that a single policy is not enough for changing workloads.

In the next step we compared the simple and advanced decider either with half or full self-tuning. The performance of the self-tuning dynP scheduler is compared to the best policy for each trace. The results show, that the advanced decider is clearly better than the simple decider. Also, if full self-tuning is applied with the advanced decider, the best basic policy is already outperformed for two traces (CTC and LANL). A detailed analysis of the decision process and the switches to each basic policy for the simple and advanced decider show their different behaviors.

Next, we compared different metrics for measuring the schedules in the self-tuning process. The results show, that using the average response time weighted by the job width (ARTwW) achieves the best results. Therefore, it was used for the remainder of the evaluation as the self-tuning metrics. The preferred decider is compared with the advanced decider, either with FCFS and SJF as the preferred policy. Choosing SJF as the preferred policy leads to the best results, but by choosing FCFS the performance of the advanced decider can not be achieved for any trace. Finally, we applied slackness to the decision process in the self-tuning dynP scheduler. We compared slackness values of up to 10% with no slackness at all. The results show, that for some traces (CTC, KTH, and LANL), applying slackness pays off. However, a general slackness value that always improves the performance was not to be found. A case analysis of the switching behavior shows, that with slackness applied the advanced decider falls back to the simple decider, as cases with two equal policies no longer exist.

In a second step, we evaluated the self-tuning dynP scheduler with increasing workloads. Different shrinking factors were used to decrease the interarrival time of jobs, so that the same amount of jobs is submitted in a shorter time. We increased the workload until the scheduler reaches a saturated state, i. e. the utilization of the system could not increase anymore. For this evaluation we used synthetically generated job sets. These are based on the previously used eight traces and have the same statistical job parameters. By using synthetic job sets, the number of simulated jobs could be decreased without changing the job set characteristic.

Again, first the three basic policies were evaluated. The results prove, that with increasing workloads LJF increases the utilization and SJF generates short response times and slowdowns. FCFS is a good compromise, as the utilization is considerably higher than with SJF and at the same time the slowdown is close to SJF. In the evaluation of the self-tuning dynP scheduler, we focused on the advanced and SJF-preferred decider and used SJF as a reference for comparing the performance of the different schedulers. For the CTC and SDSC based job sets both deciders improve the slowdown by about 9 and 6% respectively, if measured as the average of all applied workloads/shrinking factors. At the same time the utilization is improved by about 1.2 and 0.9 percentage-points. Finally we applied different slackness values in order to reduce the number of policy switches and thereby possibly increase the performance of the self-tuning dynP scheduler. However, the evaluation shows, that slackness is not as beneficial as expected, especially for higher workloads.

6 Job Scheduling in Grid Environments

In the previous two chapters we focused on the implementation and evaluation of self-tuning schedulers in the domain of resource management for single HPC systems. In the following chapter we present an adaptive multi-site grid scheduler for executing jobs across multiple sites in so called meta or grid computing environments.

In metacomputing environments [83] many computing resources are used in a cooperative way. The classical metacomputing use case is the multi-site execution of an application. This means, that the application is synchronously started on a set of geographically distributed resources at the same time. The running parts of the application collaborate closely and therefore have to communicate with each other. One major problem in the past was, that the parts of the application are able to communicate. For this purpose many solutions are available nowadays. One to mention is the metacomputing MPI library PACX [68]. After solving this, the next problem is to have all needed resources available at the same time. From this, new requirements for the job scheduling arise.

Computational grids, grid environments, or in short grids [30] take up the idea of metacomputing and broaden it, as not only computing resources are considered. Other types of resources like network connections, data archives, special I/O hardware (telescopes, physical detectors, or 3D visualization devices), software licenses, and even human resources are joined in grid environments. The aim of grids is to make these different types of resources transparently accessible and usable for the users.

The most well-known analogy is the electric power grid: consumers simple use the electricity without knowing how and where it is generated. At some time in the future, grids should be able to deliver a similar service: users connect their laptop to the internet and they can use different services for solving their problems, without knowing which resources are involved or where they are located. Computing portals for e. g. CFD, FEM, or weather forecast applications are a first step towards this vision.

Many more and more different use cases exist today, which have to be solved with grid environments. For example, applications with pre- and postprocessing: Most often these stages have to be computed on different types of computer architectures, e. g. preprocessing is done on an distributed memory MPP, the actual computation is done on a vector machine, whilst the postprocessing and visualization is done on an SGI Origin connected to a 3D cave. Such jobs with pre- and postprocessing are also called job chains. However, such job types might generate non-trivial problems, e. g. transferring the data from one stage/site to the next and often over long distances (i. e. low bandwidth and high latency). In contrast, multi-site applications need network connections with low latency, as the synchronously started parts of the application communicate whilst they are running. For this, an additional network reservation may be needed.

Other applications generate Terabytes of data during their execution. This data has to be stored. Hence, space on a local storage server or in a remote data archive is needed. In case the data is transferred to a remote archive, an additional network reservation is needed in order to complete the data transfer in time. It is also thinkable that other types

of resources (e. g. radio telescopes or 3D caves) are involved in grid jobs. Obviously, such resources are not typically managed by common resource management systems that are used for supercomputers or network connections. Hence, interfaces to different kinds of resource management systems are needed in modern grid environments.

In the following, we concentrate on job scheduling aspects in such grid environments. However, job scheduling in grid environments is different compared to the previously described job scheduling for single HPC machines. The local machine schedulers focus on packing the submitted jobs in a way that the utilization or response time of the system is improved. The task of a grid scheduler is more complex as more than one machine can execute a job. Hence, a higher level of scheduling has to be implemented on top of the local machine schedulers.

As previously described, grid environments combine different types and sizes of computing resources. Diverse benefits arise for the users and owners of grid resources. The users benefit from the larger set of machines and their different sizes. If they leave their job requirements unchanged, they may notice a shorter waiting time, as load sharing is done by the grid scheduler on the available machines. Users may also request more resources than locally available, so that either larger problems are solved or the same problem size is solved with a higher precision. Owners also benefit from participating in grids. A higher utilization is more than likely as more users have access to the machine. However, owners also have to pay attention to their local users as they might be unhappy due to an over-loaded machine.

The standard resource requests known from single machines are denoted as *single-site* jobs in grid environments. The grid scheduler has to assign resources that belong to the same machine, hence the name single-site. As previously mentioned single-site jobs benefit from load sharing effects in the grid.

In contrast, *multi-site* jobs can span across several machines. The grid scheduler does not necessarily have to assign resources that belong to a single machine. The usage of multi-site applications has been theoretically discussed for quite some time [2]. With multi-site computing the execution of a job is done at different sites. This results in a larger number of totally available resources for a single job. The effect on the average response time is yet to be determined as there are only few real world multi-site applications. This lack of real multi-site applications may be the result of an absence of a common grid computing environment which is able to support the allocation of resources in parallel on remote sites. In addition, many users fear a significant adverse effect on the computation due to the limitations in the network bandwidth and latency over wide area networks. This overhead depends on the communication requirements of the application parts. As WAN networks become ever faster, this overhead may decrease over a period of time. Hence, we evaluate which amount of overhead will still result in an overall user benefit.

To evaluate the effect of multi-site applications in a grid environment, we examine the usage of multi-site jobs in addition to job sharing. To this end, the NWIRE[1] [94] discrete event simulation environment was used. Again, we used workload traces as the basis for the evaluation of sample grid configurations. The potential benefit of a computing site participation in a computational grid is evaluated. This evaluation focuses on the question whether sharing jobs between sites and/or multi-site applications provides advantages in processing the existing workload.

[1] Net WIde REsources

6.1 Site Model

We assume a computing grid consisting of independent computing sites. Each site retains its local workload. That means, that each site has its own computing resources as well as local users that submit jobs to the local job scheduling system. In a typical single site scenario all jobs are only executed on local resources.

The sites may combine their resources and share incoming job submissions in a grid computing environment. Here, jobs can be executed on local *and* remote machines. The computing resources are expected to be completely committed to grid usage. That means job submissions of all sites are redirected to and distributed by a grid scheduler. This scheduler exclusively controls all grid resources. For a real world application this requirement may be difficult to fulfill. There are other possible implementations where site-autonomy is still maintained. Obviously, a centralized grid scheduler depicts a single point of failure, especially if the number of participating sites is great. However, we neglect the distribution of this centralized instance as our focus is on the scheduling algorithms. Using the MOL-Kernel [34, 33] is one way to implement such a distributed grid scheduler.

6.2 Machine Model

Massive parallel processor systems (MPP) are assumed as the computing resources. Each site has a single parallel machine that consists of several nodes. Each node has its own processor, memory, disk etc. The nodes of the machine are connected with a fast interconnection network that does not favor any communication pattern inside the machine [25]. This means, that an arbitrary subset of nodes can be allocated to the parallel job. This model comes reasonably close to real systems like an IBM RS/6000 Scalable Parallel Computer, a Sun Enterprise 10000, or HPC clusters.

For simplicity, all nodes in this study are identical. The machines at the different sites only differ in the number of nodes they have. The existence of different resource types would limit the number of suitable machines for a job. In a real implementation such a pre-selection phase is part of the grid scheduling process and is normally executed before the actual scheduling process takes place. After the pre-selection phase the scheduler can ideally choose from several resources that are suitable for the job request. In this study, we neglect this pre-selection phase and focus on the scheduling result. Therefore, it is assumed that all resources are of the same type and all jobs can be executed on all nodes.

The machines support space-sharing and run the jobs in an exclusive fashion. Jobs are not preempted or time-shared. That means, that once started, a job runs until completion. Furthermore, we do not consider that jobs exceed their allocated time. After submission, a job requests a fixed number of resources that are necessary for starting the job. This number is not changed during the execution of the job. That means jobs are rigid and not moldable, malleable, or evolving [28, 16].

6.3 Job Model

As previously mentioned, the local workload is retained. Hence, jobs are submitted by independent users at the local sites. This generates an incoming stream of jobs over a period

of time. Therefore, the scheduling problem is an on-line scenario without any knowledge of future job submissions.

We restrict our simulations to batch jobs, as this job type is dominant on most MPP systems. For interactive jobs there are usually dedicated machine partitions where the effect of the scheduling algorithm is limited. In addition, interactive jobs are usually executed on local resources.

The local scheduling system allocates resources to the incoming jobs and determines a starting time. The jobs are executed without any further user interaction. Data management of any files is surely of major importance in grid environments. However, we neglect the data management in this study. This means, that for our grid computing scenario a job (and its data) is transmitted to a remote site without any overhead. In real implementations the transport of data requires additional time. Usually this effect is hidden by prefetching before the execution starts. Postfetching is used to transfer the generated output of the application. In these cases the resulting overhead is not necessarily part of the scheduling process.

It is the task of the data manager in a grid environment to handle these data transfers. However, some scenarios exist in which the amount of data is, that large that it can not be hidden. Therefore, the data manager has to closely interact with the job scheduler. It has to be ensured that the input data is available at the start of the execution. If large output data is generated by the running application the data manager has to assure that this data is transferred to a storage site before the assigned resources are released and all associated data is probably deleted.

In a grid environment we assume the jobs are able to run in multi-site mode. That means a job can run in parallel (started synchronously) on a set of nodes distributed over different sites. This allows the execution of large jobs that require more nodes than available at a single site in the grid environment. The distributed parts of the application often communicate with each other. Hence, the impact of bandwidth and latency has to be considered as wide area networks can be involved. In the simulations we address this subject by increasing the job length, if a multi-site execution is applied to a job.

6.4 Scheduling System

Simple first-come-first-serve (FCFS) scheduling policies have often been applied for the parallel job scheduling in queuing based resource management systems for single parallel machines. As an advantage, this algorithm provides some kind of fairness [79] and is deterministic for the user. However, pure FCFS can result in poor quality, if jobs with large node requirements are submitted. To circumvent this problem, a strategy called backfilling became standard on almost all queuing based resource management systems today. It requires knowledge of the estimated duration of all jobs and can be applied to any greedy list scheduling. If the next job in the list can not be started due to a lack of available resources, backfilling tries to find another job in the list which can use the idle resources straight away. It will not postpone the execution of the next job in the list. The original backfilling algorithm was introduced by Lifka in 1995 [55].

In our scenario for grid computing, the task of job scheduling is delegated to a grid scheduler. The local scheduler is only responsible for starting the jobs after allocation by the grid scheduler. Note that we use a central grid scheduler. In a real implementation the architecture of the grid scheduler can differ as single central instances usually lead to drawbacks in

performance, fail-safety or acceptance of resource users and owners. Nevertheless, distributed architectures can be designed to act similar to a central grid scheduler.

6.5 Scenarios

From the different use cases mentioned above we extract three scenarios, which are:

- **local job processing** as the reference case. All jobs are submitted and executed locally.
- **job sharing** between all computing sites in the grid environment. The grid scheduler can only assign nodes from one site to a job.
- **multi-site** computing, in which the grid scheduler can assign nodes from different sites.

These scenarios are most common in today's grid environments and most interesting from a scheduling point of view. In the following each scenario is described in detail. Later the evaluation results are presented.

6.5.1 Local Job Processing

This scenario represents the common situation where no job exchange or load sharing is done between sites (Figure 6.1). Local computing resources are only available to local users. In difference to Chapter 4 queuing based resource management systems are now assumed to be used for the local job scheduling. As the name implies the locally generated workload is not shared with other sites. Hence, load balancing effects do not occur. In the common sense this scenario is no real grid environment, but it depicts a good basis for our evaluations. We concentrate on EASY backfilling [55], as its performance proves to be superior to conservative backfilling.

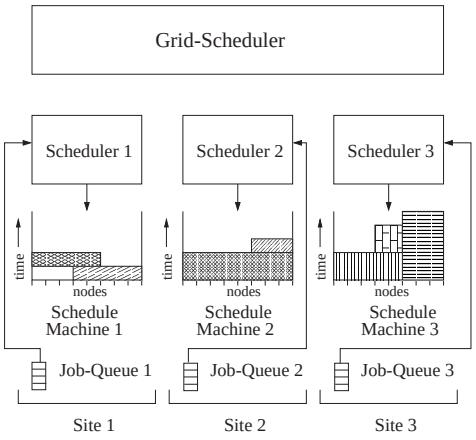


Figure 6.1: Sites executing all jobs locally.

6.5.2 Job Sharing

In this scenario all jobs submitted at any site are delegated to the grid scheduler as seen in Figure 6.2. In this study the applied scheduling algorithms consist of two phases. First an appropriate machine is selected. Then the allocation in time for this machine takes place:

1. **Machine Selection:** Several methods exist for selecting machines (BiggestFree, Random, BestFit, EqualUtil). Simulation studies [40, 84] showed good results for a selection strategy called *BestFit*. With this strategy that machine on which the job leaves the least number of free resources if started is selected.

2. **Scheduling Algorithm:** The backfilling strategy is applied for the single machines as well. This algorithm has shown best results in previous studies.

According to the definition in [31] the job sharing scenario refers to *full control* as the central instance has full control over the local machines and local schedulers do not process any locally submitted workload.

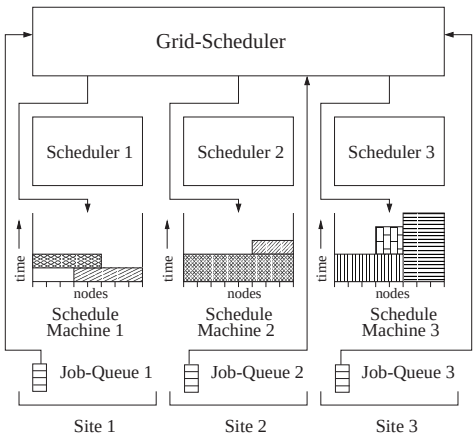


Figure 6.2: Sites sharing jobs and resources.

6.5.3 Multi-Site Job Execution

This scenario is similar to job sharing: a grid scheduler receives all submitted jobs. However, the grid scheduler is no longer forced to choose nodes from a single site. Jobs can now be executed across site boundaries (Figure 6.3).

Again, several strategies exist for multi-site scheduling [40, 84]. Here we use a scheduler which first tries to find a site that has enough free resources for starting the job. If such a machine is not available, the scheduler tries to allocate the jobs to resources from different sites. To this end, the sites are sorted in descending order of free resources. The allocation of free resources for a multi-site job is done in this order. This assures that the number of combined sites is minimized. If there are not enough resources free for a job, it is queued and normal backfilling is applied.

Spawning job parts over different sites usually generates an additional overhead. This overhead is caused by the communication via slow networks (e. g. wide area networks). In consequence the overall execution time of the job will increase depending on the communication pattern. For jobs with a limited communication demand there is only a small impact. Note that, neglecting the additional communication overhead for multi-site execution, the grid would behave like a single large supercomputer. Hence, in the ideal case multi-site scheduling would outperform all other scheduling strategies. In this study, we examine the effects of multi-site processing on the schedule quality under the influence of different communication

overheads. To this end, the communication overhead is modelled as an extension of the jobs duration d_i to d_i^* for a job i that runs on multiple sites at a constant factor of: $d_i^* = (1+p) \cdot r_i$ with p = 0 .. 40 % in steps of 5%.

Of course, in a real world implementation of multi-site scheduling this simple model of considering the communication overhead would not be sufficient. Due to more complex communication patterns of multi-site applications and different network speeds more sophisticated models have to be used. Furthermore, we do not consider how many parts the grid scheduler partitions the multi-site application, or how many nodes belong to each part. Both scheduler induced values surely influence the communication overhead and should therefore be considered during job scheduling. We assume that all the mentioned overheads are summarized in the above given simple extension model.

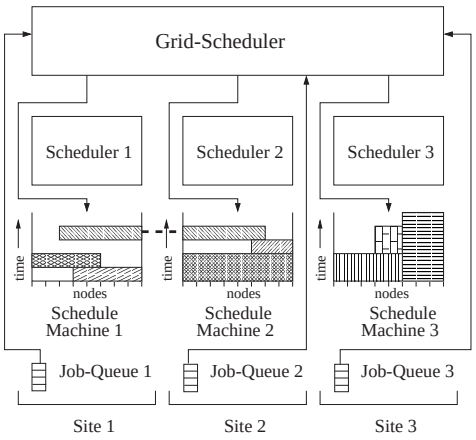


Figure 6.3: Support for multi-site execution of jobs.

7 Evaluation of Multi-Site Grid Scheduling

For the comprehensive evaluation of the described algorithms and scheduling schemes discrete event simulations have been performed. Various machine and workload configurations have been examined during these simulations [13, 15, 14] and are described in the following.

7.1 Machine Configurations

All configurations use a total of 512 resources. These resources are partitioned in the various machines as shown in Table 7.1. A larger grid environment with more total resources would require additional scaling of the workload without improving the evaluation validity.

identifier	configuration	max. size	sum
m64	$4 \cdot 64 + 6 \cdot 32 + 8 \cdot 8$	64	512
m64-8	$8 \cdot 64$	64	512
m128	$4 \cdot 128$	128	512
m256	$2 \cdot 256$	256	512
m256-5	$1 \cdot 256 + 4 \cdot 64$	256	512
m384	$1 \cdot 384 + 1 \cdot 64 + 4 \cdot 16$	384	512
m512	$1 \cdot 512$	512	512

Table 7.1: Used Machine Configurations.

The configurations *m64-8*, *m128*, and *m256* are representations of sites with equal machines. They are balanced as there is an equal number of resources at each machine. The configuration *m384* and *m256-5* are examples for a large computing center with several client sites. In the *m256-5* configuration the largest machine is smaller than the largest machine in the *m384* configuration. The configuration *m64* represents a cluster of several sites with smaller machines. The reference configuration *m512* consists of a single site with one large machine. In this case no grid computing is used and a single scheduler can control the whole machine without any need to split jobs.

The *m384* machine configuration was chosen, as the largest job, in the later described workloads, requests 336 resources. Hence, it is ensured that these large jobs can be started on a single site. In order to stick with the power of 2 size of the remaining machines, 384 was chosen.

7.2 Workloads

Unfortunately, no real workload is currently available for grid computing. For our evaluation we derived a suitable workload from real machine traces. These traces have been obtained from the Cornell Theory Center (CTC) and are based on an IBM RS6000/SP parallel computer with 430 nodes. This trace was already used in the previous evaluations of

the self-tuning dynP scheduler. A description and more details on the trace are found in Section 3.5.

In order to use the CTC trace for this study it was necessary to modify the traces in order to simulate submissions at independent sites with local users. To this end, the jobs from the real trace have been assigned to the different sites in a round robin fashion. It is typical for many known workloads that jobs favor requesting power of 2 number of nodes. The CTC workload shows the same characteristic. The modelling of configurations with smaller machines would put these machines at a disadvantage if the number of nodes is not a power of 2. Hence, the machine configurations consists of 512 nodes. Nevertheless, the trace consist of enough workload to keep a sufficient backlog on all systems [40]. The backlog is the workload that is queued in every time step if there are not enough free resources to start the jobs. A sufficient backlog is important as a small or even no backlog indicates, that the system is not fully utilized. In this case, there is not enough workload available to keep the machines working. Many schedulers, e. g. the mentioned backfilling strategy, require enough jobs are available for backfilling in order to utilize idle resources. Hence, insufficient backlog leads to a bad scheduling quality and unrealistic results.

Over all, the quality of a scheduler is highly dependent on the workload. To minimize the risk in achieving singular effects the simulations have been done for all workload sets:

- a synthetic probabilistic generated workload on the basis of the CTC traces.
- three extracts of the original CTC traces.

The synthetic workload is very similar to the CTC trace. It has been generated to prevent singular effects in real traces (e. g. short and rare bursts of submission or down time of the machine) affecting the accuracy of the results. Additionally, the usage of three extracts of the real traces are used to get information on the consistency of the results for the CTC workload. Each workload set consists of 10,000 jobs which corresponds to a period of more than three months in real time.

A problem with these simulations is the handling of wide jobs contained in the original workload traces. The widest job in the CTC traces e. g. requests 336 processing nodes. On the one hand, these jobs can be used in simulations with multi-site. Here, jobs can be split across different sites to use more resources than available at a single site. On the other hand, these large jobs can not be started in simulations of scenarios with smaller machines and only with local job processing or job sharing.

To permit a valid comparison of schedules and simulation results, no jobs must be neglected. Therefore, we assume that the workload of large jobs is still generated at a single site. Large jobs are split up into several parts, so that they can be processed on the given machine size. This is 64 for *m64* and *m64-8*. Appropriately, the job size is limited by the size of the largest machines in the job sharing scenario. Here, users can submit jobs that request more resources than locally available. To allow the comparison of different scenarios the following modifications have been applied to each of the four mentioned workloads.
The applied workload modifications are:

1. large jobs are split in parts to match the local machine size,
2. large jobs are split in parts to match the largest machine size in the machine configuration,

3. large jobs are split in parts with a maximum of 64 nodes,

4. large jobs remain unchanged.

The workloads with modification *1* were executed in all three scenarios. The workloads with modification *2* were used for the job sharing and multi-site scenarios, whereas modification *3* was only used for the multi-site scenario. All of these modifications do not change the overall amount of submitted workload. We assume that a certain amount of workload exists at a local site. Depending on the scenario, a user may submit jobs larger than the local machine. The simulations allow the examination of the impact caused by larger multi-site jobs on the schedule.

Table 7.2 shows a summary of all used workloads in this study and an identifier is introduced for each workload.

identifier	description
10_20k_org	An extract of the original CTC traces from job 10,000 to 20,000.
10_20k_max64	The workload 10_20k_org, split up in jobs with a maximum of 64 processors.
30_40k_org	An extract of the original CTC traces from job 30,000 to 40,000.
30_40k_max64	The workload 30_40k_org, split up in jobs with a maximum of 64 processors.
60_70k_org	An extract of the original CTC traces from job 60,000 to 70,000.
60_70k_max64	The workload 60_70k_org, split up in jobs with a maximum of 64 processors.
syn_org	The synthetically generated workload derived from the CTC workload traces.
syn_max64	The workload syn_org, split up in jobs with a maximum of 64 processors.

Table 7.2: Workload Configurations.

7.3 Results

In the following we present the results for different scenarios, resource configurations and workloads. All presented work on scheduling in grid environments is done in cooperation with colleagues from the Computer Engineering Institute at the University of Dortmund: Carsten Ernemann, Volker Hamscher, and Ramin Yahyapour.

The results on the effects of machine configurations in Section 7.3.1 are published in [15]. The results on job sharing and multi-site job execution as presented in Section 7.3.2 are published in [13]. Further enhancement to the multi-site scheduling approach and their results on the schedule quality are presented in Section 7.3.3 and published in [14].

7.3.1 Machine Configurations

The simulation results show, that configurations with equal sized machines provide significantly better scheduling results than machine configurations that are not balanced. Machine configurations with a small number of machines produce better scheduling results under the precondition that each configuration has the same amount of resources in its sum.

As a performance metrics, the average weighted response time (AWRT) and the average weighted waiting time (AWWT) are used in this study. The response time of each job is the difference between the completion time and the submission time. The response time of each job is weighted by its resource consumption. The average weighted response time is the sum of all weighted response times divided by the number of all jobs. The waiting time of each

job is the difference between the start time and the submission time. The weights are defined the same way as for the average weighted response time. Additional performance metrics are:

Average Response Time weighted by Width:

$$AWRT = \frac{\sum_{j \in Jobs} (j.requestedResources \cdot (j.endTime - j.submitTime))}{\sum_{j \in Jobs} j.requestedResources} \tag{7.1}$$

Average Waiting Time weighted by Width:

$$AWWT = \frac{\sum_{j \in Jobs} (j.requestedResources \cdot (j.startTime - j.submitTime))}{\sum_{j \in Jobs} j.requestedResources} \tag{7.2}$$

Note that, the mentioned weights prevent any prioritization of small over large jobs in regard to the average weighted response and average weighted waiting time, if no resources are left idle [77]. The average weighted response time is a mean for the schedule quality from a user perspective. A shorter AWRT indicates, that the users have to wait less for the completion of their jobs. A shorter AWWT indicates, that the jobs have to wait less, before they are started. Note, the average weighted waiting time should not be too small, as this indicates, that the backlog is very small and that jobs are started shortly after their submission.

As an example for the scheduling quality, the average weighted response time for the workload *60_70k_org* and all resource configurations is shown in Figure 7.1. The other workloads show a similar behavior. Several parameter settings for the increase of the corresponding run time for multi-site jobs were examined. The AWRT for the machine configuration *m512* is constant for all parameters, due to the fact that no multi-site scheduling is applied in this configuration. As een in Figure 7.1 the AWRT decreases from *m64-8* to *m128* to *m256*. All of these machine configurations have equal sized machines, but the size of the machines increases between the configurations. The AWRT decreases from machine configuration *m64* to *m64-8*, as the configuration *m64-8* consists of more larger machines than *m64*. As more jobs are executed in multi-site mode in configurations with a higher number of smaller machines, an increase of the communication overhead p has a higher impact on the AWRT. The same effect is observed for *m256* and *m256-5*. Here the configuration *m256-5* is not balanced and contains smaller machines. This turns out to be a disadvantage. In conclusion, there is no hard rule, neither for the advantage of equal sized nor of the biggest machines. This will depend highly on the characteristic of the workload as seen in the comparison between *m384* and *m256*. In this example, *m384* outperforms *m256* with a communication overhead of up to 45%.

In Figure 7.2 the average weighted response time is shown relative to the average weighted response time of configuration *m512*. This underlines the above statements in more detail.

The average weighted response times for the configurations *m64* and *m64-8* are almost similar to the values of the multi-site parameter p between 0% and 35%. If the overhead exceeds 35%, the AWRT of configuration *m64* increases dramatically and is at least 25% bigger than the AWRT for the configuration *m64-8*. The decreasing AWRT from *m64-8* to *m128* and *m256* is evaluated with at least 20% for each step for overhead parameters over

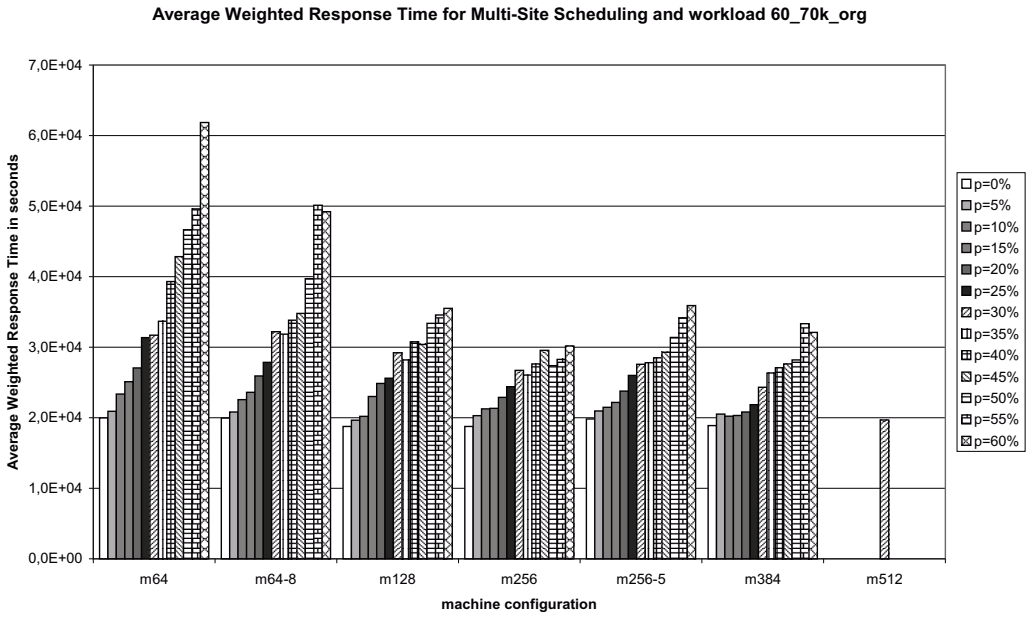


Figure 7.1: The average weighted response time in seconds for workload *60_70k_org* and all machine configurations and multi-site scheduling.

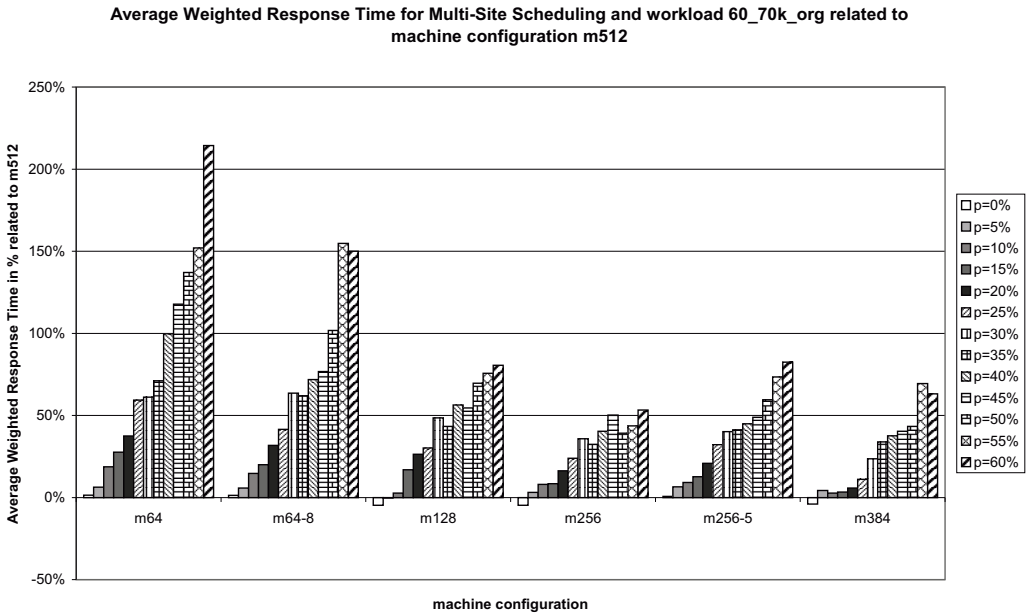


Figure 7.2: The average weighted response time for workload *60_70k_org* and all machine configurations relative to machine configuration *m512* for multi-site scheduling.

35%. The difference of the AWRT for the parameters under 35% is not significant. The comparison between configuration *m256* and *m256-5* results in a similar conclusion. The

AWRT increases for parameters over 45% by at least 25%.

file	10_20k_org [$jobs$]≙[10^{-2}%]	30_40k_org [$jobs$]≙[10^{-2}%]	60_70k_org [$jobs$]≙[10^{-2}%]	syn_org [$jobs$]≙[10^{-2}%]
p=0%	539	431	840	774
p=5%	543	436	850	769
p=10%	582	448	928	827
p=15%	572	490	917	906
p=20%	583	546	946	946
p=25%	601	567	951	1042
p=30%	622	521	979	1026
p=35%	637	523	1036	1063
p=40%	647	534	1106	1012
p=45%	673	597	1008	1181
p=50%	755	579	1029	1188
p=55%	748	578	1086	1233
p=60%	746	638	1114	1177

Table 7.3: Number of multi-site jobs for different workloads and different parameters using machine configuration *m128*.

The increase of AWRT results from two effects. First of all, the whole workload increases, as all multi-site jobs have a longer execution time. Second, the number of multi-site jobs increase. Table 7.3 shows the number of multi-site jobs for machine configuration *m128*, for different workloads and different multi-site parameters. For all workloads the number of multi-site jobs increases. This process is not continuous. However, the difference between the number of multi-site jobs for p = 0% and p = 60% is always at least 30%.

This behavior results from the policy to schedule all jobs as soon as possible after submission. Due to an increased execution time of all multi-site jobs the number of free time slots within the schedule decreases and the probability of free time slots within one machine decreases, too. Therefore, jobs can only be started as soon as possible, if free time slots from different machines are combined.

Table 7.4 indicates, that the increasing number of multi-site jobs corresponds with an increasing part of the squashed area[1] of all multi-site jobs related to the squashed area of the whole workload. Between parameters p = 0% and p = 60% this part increases at least by 50%.

Table 7.3 and Table 7.4 also indicate, that jobs running in multi-site mode are mostly larger jobs. This is concluded, as about 5% to 10% of all jobs are responsible for about 20% to 40% of the whole workload depending on the used job trace. This effect even increases for a higher multi-site parameter p.

As seen in Figure 7.2, the AWRT for the machine configurations *m128*, *m256*, and *m384* is smaller in comparison to *m512* for the multi-site parameter p = 0%. This effect results from the scheduling algorithm for multi-site. Some jobs can surpass queued jobs during the multi-site scheduling, because the scheduler tries to start the jobs as soon as possible or to split them. In contrast, normal backfilling queues the job in first-come-first-serve order.

Figure 7.3 shows the already mentioned behavior for different workloads whilst using multi-site scheduling with p = 50%. Therefore, it is reasonable to assume that the above explained effects apply in general.

[1]The squashed area is defined as: $\sum_{j \in Jobs} (j.requestedResources \cdot j.runtime)$.

file	10_20k_org	30_40k_org	60_70k_org	syn_org
p=0%	22,19%	23,97%	34,01%	40,75%
p=5%	22,87%	25,09%	36,36%	41,21%
p=10%	24,71%	27,21%	37,28%	46,85%
p=15%	25,30%	28,24%	38,97%	47,95%
p=20%	26,05%	31,36%	40,43%	46,91%
p=25%	29,40%	31,93%	41,02%	52,81%
p=30%	29,11%	30,84%	44,53%	53,62%
p=35%	30,24%	31,01%	44,38%	54,01%
p=40%	31,21%	32,82%	48,10%	56,01%
p=45%	33,22%	37,20%	45,87%	59,77%
p=50%	37,15%	34,18%	46,25%	59,99%
p=55%	37,87%	36,05%	49,81%	62,72%
p=60%	41,46%	39,17%	52,38%	60,46%

Table 7.4: The squashed area of the multi-site jobs related to the squashed area of the whole workload for different workloads and multi-site communication overheads p using machine configuration *m128*.

The AWWT for simulation with the workload *60_70_org* for all used machine configurations is shown in detail in Figure 7.4. The value of the AWWT is at least two hours. This indicates the existence of an appropriate backlog, which is necessary for the backfilling algorithm.

The results presented in Figure 7.4 show a similar behavior like the results in Figure 7.1, as the values only differ in the execution time of the jobs.

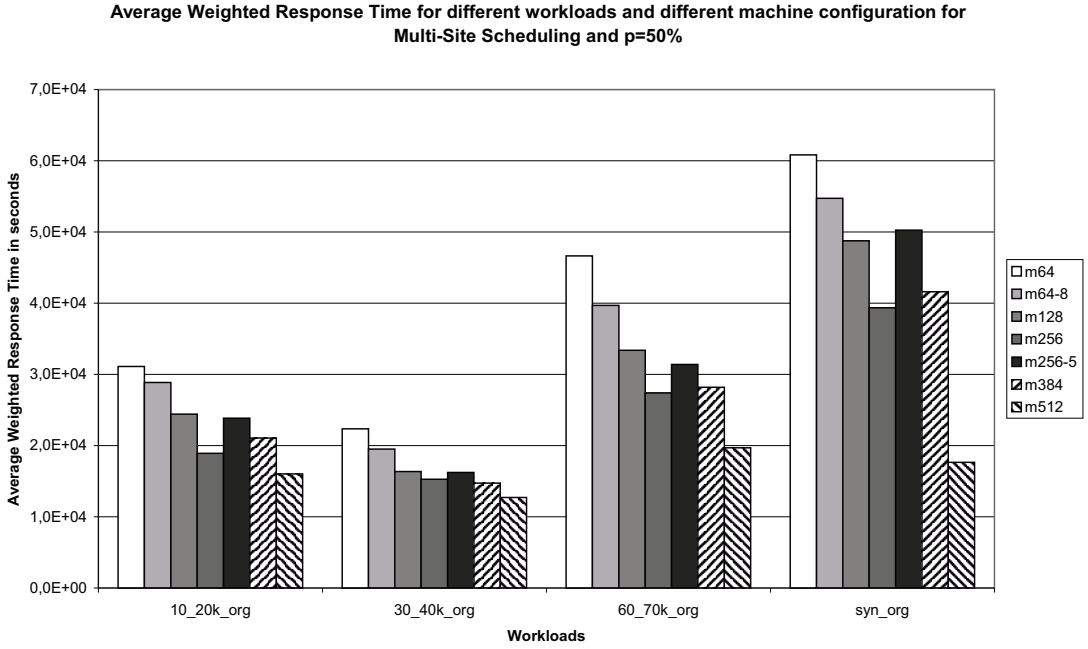


Figure 7.3: The average weighted response time in seconds for all workloads and all machine configurations for multi-site scheduling and p = 50%.

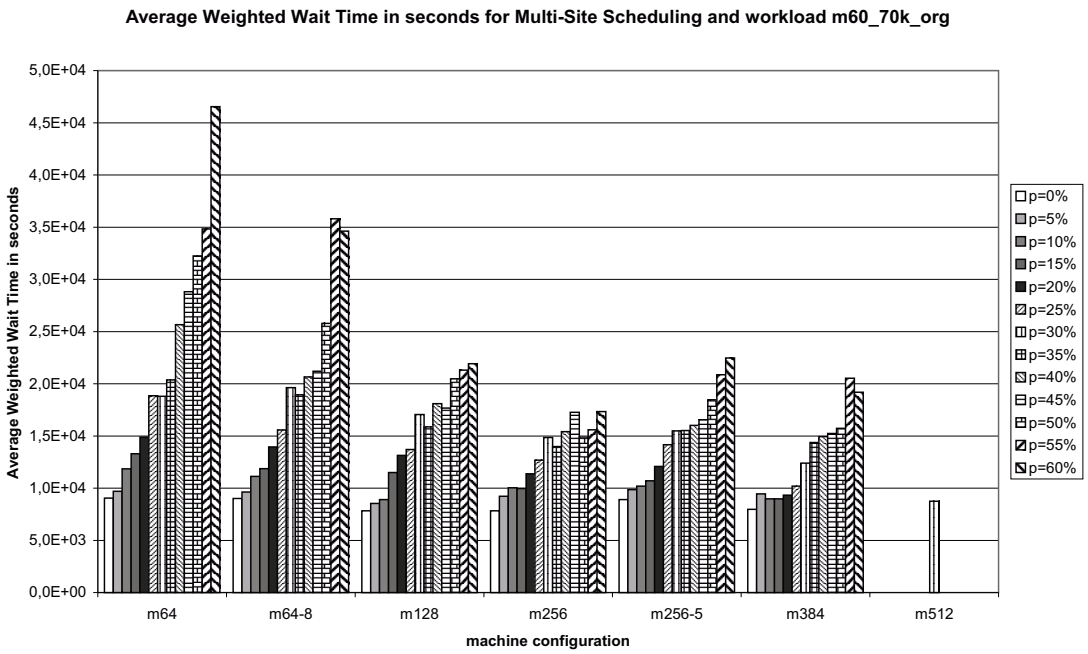


Figure 7.4: The average weighted waiting time in seconds for workload 60_70k_org and all machine configurations for Multi-Site Scheduling.

The next aspect of this study is to identify whether all workloads show the described behavior for each machine configuration. The configuration *m128* is chosen to demonstrate the results. All other machine configuration produce similar results.

Figure 7.5 demonstrates the examined behavior for the AWRT for multi-site scheduling. Depending on the multi-site parameter p the AWRT increases for all workloads. The actual deterioration of performance differs between the workloads.

After the evaluation of multi-site scheduling the results for job sharing will be presented. Jobs with limited resource demands (e. g. 50% of the largest machines) suffer only a minor drawback for configurations with smaller machines. In general, the increase of the average weighted response time is lower than 20% for all partitioned configurations. An exception is *m64* compared to a single large machine with 512 nodes. The workload *syn_max64*, an exception, shows, that the results are highly dependent on the workload characteristics.

In Figure 7.6 the AWRT for all workloads and all machine configurations is presented. The results correspond to the statements made for multi-site scheduling. The AWRT for all workloads and machine configuration *m64* is higher than the AWRT for machine configuration *m64-8*. This indicates, that the balanced configuration with larger machines is advantageous to the unbalanced configuration with other smaller machines. The comparison between the configurations *m64-8*, *m128*, and *m256* shows, that the use of larger machines produces favorably better scheduling results. The analysis of the scheduling behavior of *m256* and *m256-5* also comes to the same results as with multi-site scheduling. The use of a configuration with two big machines is superior to the usage of a system with one big and several smaller machines. Again, the comparison between *m256* and *m384* provides no clear result.

The already mentioned conclusions for the job sharing scenario are seen in Figure 7.7 as

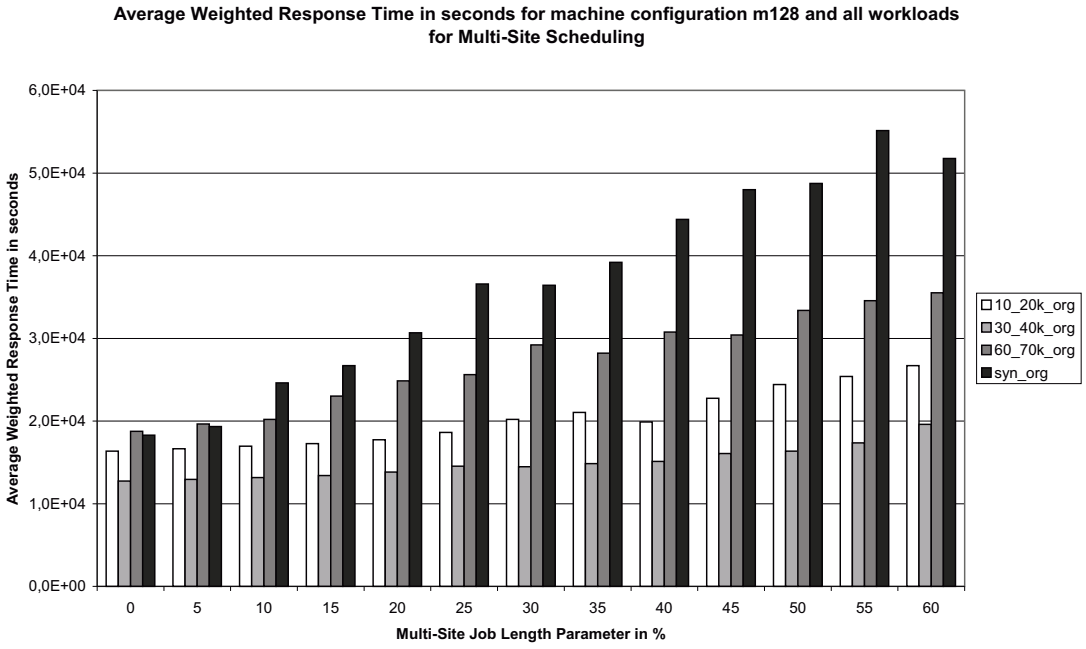


Figure 7.5: The average weighted response time in seconds for machine configuration *m128* and all workloads for multi-site scheduling.

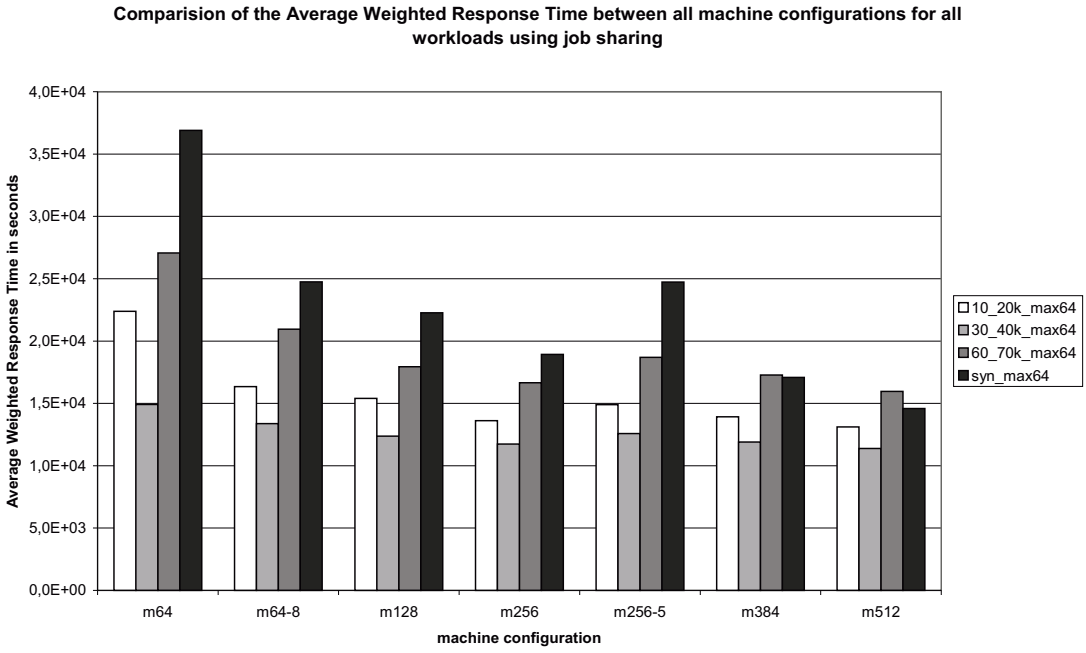


Figure 7.6: The average weighted response time in seconds for all workloads and all machine configurations for job sharing.

well. Here, the AWWT is presented for all workloads and all machine configurations. The minimal AWWT is approximately 45 minutes and so there is evidence of a sufficient backlog for the backfilling strategy.

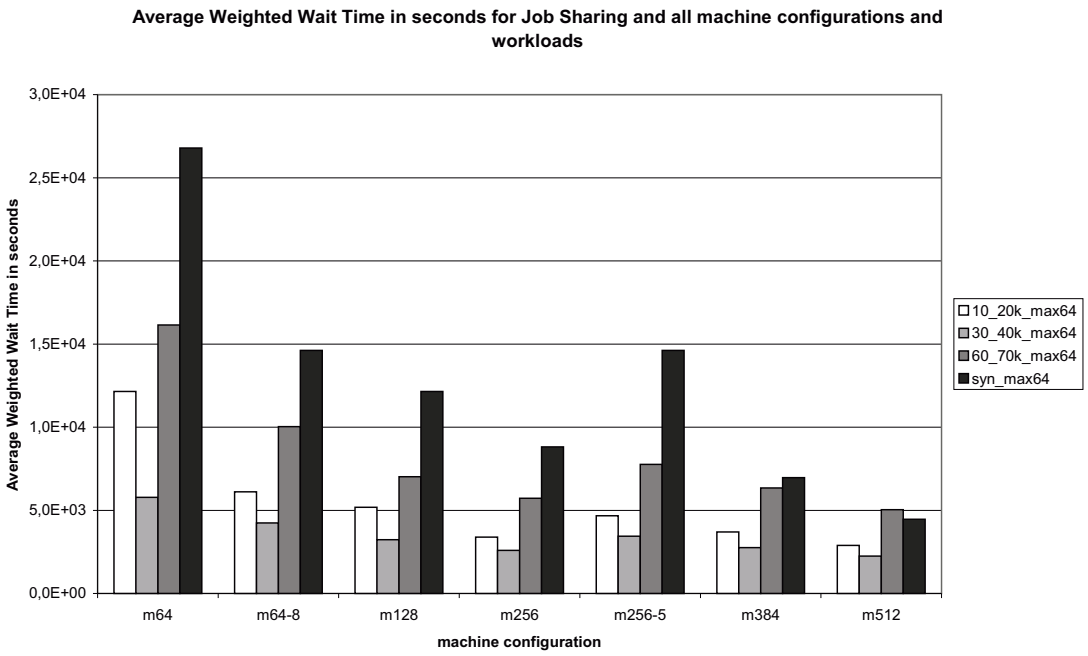


Figure 7.7: The average weighted waiting time in seconds for all workloads and all machine configurations for job sharing.

Concluding Remarks

As expected, the results show, that configurations with large machines are superior to configurations with smaller machines. However, as long as jobs are limited in resource demand (e. g. 50% of the largest machines) configurations with smaller machines result only in a minor drawback. For example, the increase for the average weighted response time remains below 15% for a configuration with four machines of 128 nodes compared to a single large machine with 512 nodes. We expect, that an adequate ratio between the workload of large jobs and the available computing power of the large machines is necessary to guarantee an acceptable response time. As long as this requirement is met, the remaining resources may consist of smaller machines without implying a significant drawback.

In contrast to job sharing, the usage of multi-site scheduling allows the execution of jobs that are not limited by the maximum machine size. As shown here, the resource configuration and the overhead, due to multi-site scheduling, have a strong impact on the AWRT of the schedule. Configurations with larger machines are superior to those with smaller machines. However, scenarios with small overheads ($p < 20\%$) only show a slight advantage of balanced large machine configurations compared to unbalanced systems with small machines.

Especially on smaller resource configurations a large overhead results in a steep increase of the AWRT, as more jobs are executed in multi-site mode in configurations with a higher

number of smaller machines. Nevertheless, there are some factors for the overhead that lead to a decrease, or at least no increase of the AWRT compared to smaller factors in the same scenario. This may be caused for two reasons. First, the number of jobs used for multi-site scheduling increases corresponding to the size of the overhead, whereas the squashed area of these jobs shows a much lower increase than the overhead. Therefore, each job must be smaller on average. This leads to the assumption, that jobs used for multi-site scheduling differ in each scenario for each size of the overhead. Second, the increase of the communication overhead incidentally leads to more suitable job sizes as certain patterns of job execution times are more common than others.

7.3.2 Job Sharing and Multi-Site Scheduling

The simulation results show, that job sharing provides significant improvement for the user, compared to local job processing. As a measure, the average weighted response time is used in this study.

The mentioned improvement of job sharing is seen in the results for all configurations and workload sets. Figure 7.8 shows the average weighted response time for the *m128* configuration with an improvement of over 50%. Similar results can be found in the other simulations. The average weighted response time of locally executed jobs certainly depends on the local workload modelling. The results shown in this thesis for single-site execution and job sharing are generated using an EASY backfill scheduler. In contrast to job sharing, single-site execution is restricted to keep the workload local. That means that no job is transferred to a remote site. As mentioned before large jobs that are wider than the local machine have to be split up into smaller jobs which are sequentially executed on the local system. This leads to a significant increase in AWRT, due to the applied weight on each job part. Job sharing on the other hand allows the transfer of jobs to remote machines.

Further improvements of the AWRT are achieved by using multi-site execution. As a reference the result for a single machine with 512 nodes is also given in Figure 7.8. The result of this *m512* configuration depicts the lower bound for the backfilling algorithm. In this configuration no machine partitioning has to be taken into account, in contrast to any other configuration. Figure 7.8 shows simulation results for multi-site execution with different run time overheads for split jobs (0% ... 40%). As expected, the average weighted response time without overhead for a multi-site is similar to the *m512* result. In this case splitting a job for multi-site execution causes no penalty.

Moreover, multi-site execution is beneficial compared to job sharing even for an overhead on execution time of about 25%.

Figure 7.9 shows the improvement for other configurations with jobs limited to the maximum machine size. The configurations with equal sized machines show better results than for the *m384* or *m64* configurations. For equal-sized machines and multi-site scheduling the overhead can be even larger and the AWRT results are still better than with job sharing.

Figure 7.10 shows the average weighted response time for some sample workloads. Note that the workload as shown in Figure 7.8 produces the least effective improvements. The average weighted response time in other configurations delivers better results. Here, the overhead on multi-site executed jobs could be even larger.

The example given in Figure 7.8 represents the workload where all original wide jobs with node requirements larger than 128 were split up into jobs requesting 128 or less nodes. As mentioned before, simulations were computed for multi-site execution with the original job

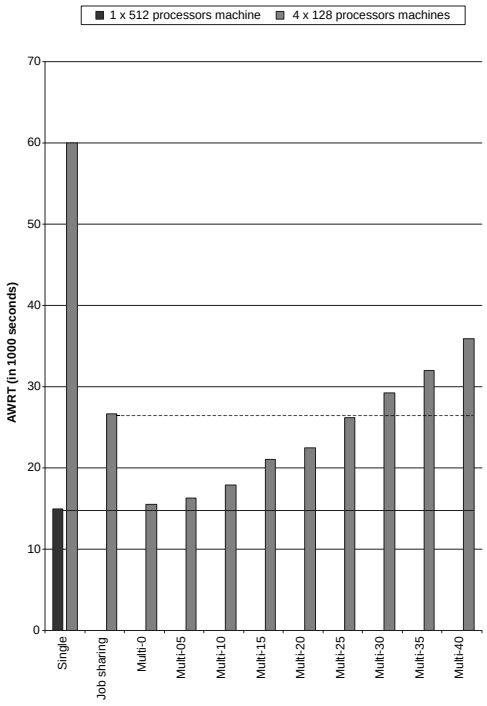


Figure 7.8: Average weighted response time for the *m128* configuration and workload *syn_org* with modification *2* applied.

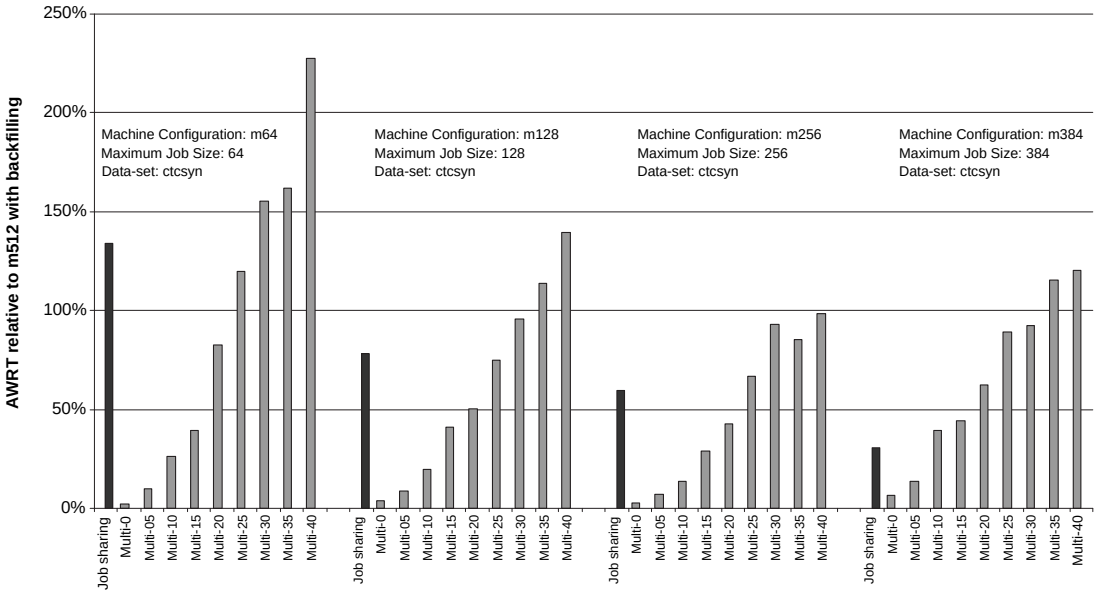


Figure 7.9: Results for different resource configurations compared to configuration *m512* with backfilling (equals 0%).

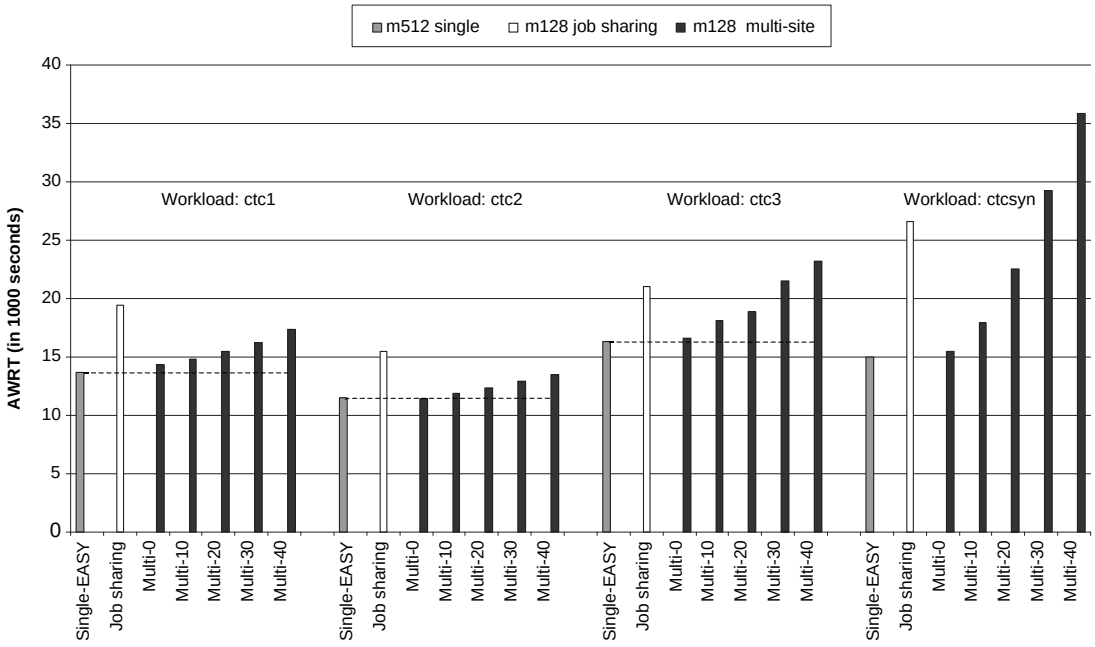


Figure 7.10: Results of different workloads.

size. Figure 7.11 shows, that submitting these wide jobs does not significantly increase the average weighted response time. The improvement over job sharing is valid even though these wider jobs are actually more complex to schedule as the job start time has to be synchronized between all job parts. This simulation can not be computed for the job sharing scenario as these wide jobs can only be executed in a multi-site scenario.

Concluding Remarks

The results show, that the collaboration between sites, by exchanging jobs even without multi-site execution, significantly improves the average weighted response time. This is already achieved with a simple algorithm by a central scheduler. Furthermore, the usage of multi-site applications leads to even better results under the assumption of a limited increase on job execution time due to communication overheads. Even with an increase of 25%, the execution time of multi-site proved to be more beneficial compared with job sharing. In terms of latency WAN networks are in the order of 2-3 magnitudes slower than common fast interconnection networks between nodes inside a parallel computer, e. g. an IBM SP Switch. Therefore, it can not be concluded that multi-site is suitable for all applications. However, multi-site is beneficial for applications with a limited demand in communication.

As grid environments and networks are becoming more common, it seems reasonable for resource owners to participate in such initiatives. Simple strategies like job sharing significantly improve the average weighted response time and therefore the quality of service to the users. The research and effort in developing multi-site programs for suitable applications with limited demands in network communication can also provide even better results. Furthermore, multi-site applications can effectively use more resources for a single job than available on any single machine. The drawback, due to submitting a larger job instead of several smaller jobs, was limited in our simulations. Of course this may vary with the amount of large jobs

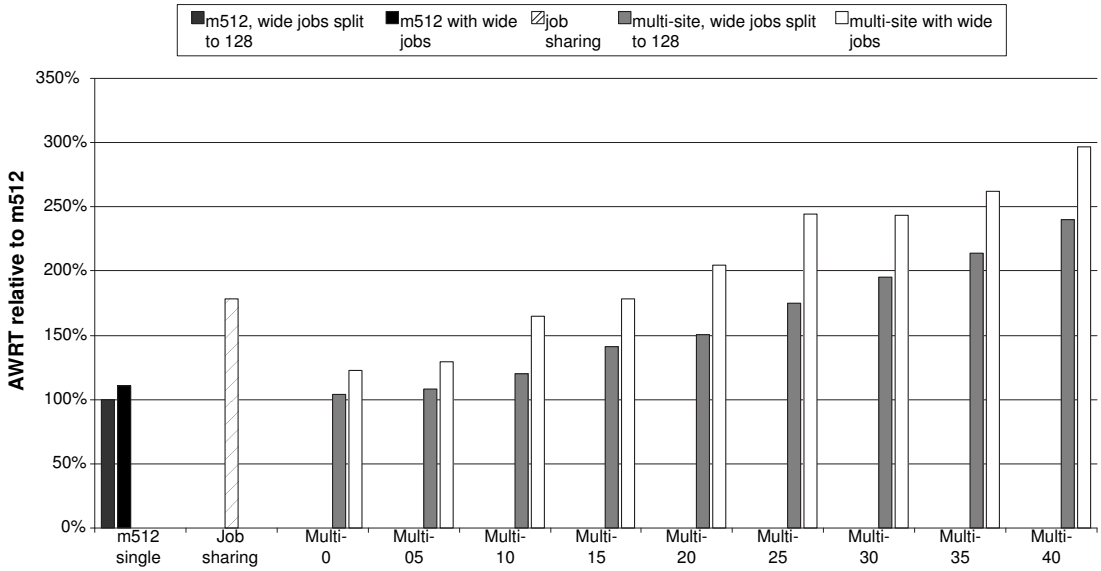


Figure 7.11: Comparing workloads with original jobs split into 128 parts with keeping wide jobs.

in a workload.

7.3.3 Constraints for Multi-Site Scheduling

In the following, we examine the behavior of multi-site scheduling, if additional constraints for the job fragmentation are applied. To this end, two parameters are introduced for the scheduling process. The first parameter *lower bound* restricts the jobs that are potentially fragmented during the multi-site scheduling by a minimal number of necessarily requested processors. The second parameter is implemented as a vector that describes the maximal number of job fragments for certain intervals of processor numbers. Presumably, this leads to a further improvement of the scheduling process. Based on those results the algorithm is extended with the ability to react on the current status of the system.

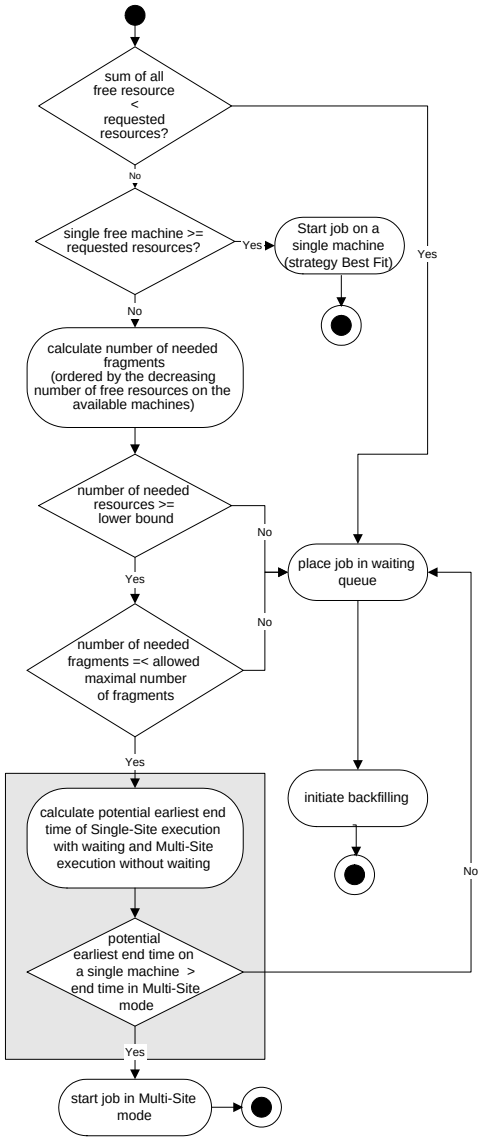


Figure 7.12: Algorithm for multi-site scheduling with constraints.

The basic scheduling process in the grid scheduler works as described in Figure 7.12. In its first step the algorithm determines whether or not there are enough free resources at this point in time. If the job can not be immediately started in single- or multi-site mode, the job is stored in a waiting queue.

Jobs are scheduled from this queue by a First-Come-First-Serve (FCFS) algorithm in combination with backfilling [55, 82, 36] where each step consists of the method described above. Otherwise for a single machine with enough free resource is searched. Upon a successful search, the job is started on this machine, which is chosen by immediately using the BestFit-strategy [17, 40]. A search failure leads to the check of the parameter *lower bound* e. g. a value of eight implies that only jobs requesting more than eight resources may be split up. Next, the required number of fragments is calculated by using a machine list ordered by the decreasing number of free resources on each machine. This allows minimizing the number of fragments (job parts running multi-site) for a job. If the number of fragments is larger than a given maximal allowed number of fragments the job is placed in the waiting queue and again backfilling is initiated.

An Adaptive Version of Multi-Site Scheduling One can think of scenarios with large communication overheads, where it would be beneficial to wait for a single-site execution of the job. Hence, a different scheduling behavior is induced, if the grid scheduler

has not found enough resources for immediately starting the job on a single site, but on multiple sites. Hence, in the following we have to distinguish between the *adaptive* and the *non-adaptive* case. The non-adaptive algorithm directly executes the job in multi-site mode.

In the adaptive case the *lower bound* is set to zero and the number of fragments is not limited. Additionally, the boxed off greyed area in Figure 7.12 is added to the algorithm. In this part of the algorithm an earliest potential end time of the job running on a single machine ($t^e_{i,single}$) is compared to the end time of a multi-site execution with its additional overhead ($t^e_{i,multi}$).

Now the idea of the adaptive version of the multi-site grid scheduler is to compare both end times. If $t^e_{i,multi} < t^e_{i,single}$ the grid scheduler immediately starts the jobs on multiple sites and extends its duration. If $t^e_{i,multi} > t^e_{i,single}$ the job remains in the waiting queue of the grid scheduler and waits for a future start. Potentially, it is started at the initial computed start time. Due to the fact that reservations are not supported by the local schedulers, the single-site start is not guaranteed. The adaptive multi-site grid scheduler holds the job back on a best effort basis. It might occur that the resources are taken by other backfilled jobs. One can expect, that, especially for large communication overheads, the adaptive multi-site grid scheduler favors waiting for a single-site execution of jobs. Jobs requesting more resources than available at the largest site in the grid environment are not influenced by this adaptive version. They are directly started in multi-site mode.

The described behavior of the adaptive multi-site scheduler is similar to the self-tuning dynP scheduler. Both schedulers check several options for scheduling jobs and choose the best solution in order to reduce the response time of jobs.

Although several performance metrics were already described before, we use a different weight for the average weighted response time metrics. The resource consumption (or squashed area) of a job is described as the product of the job's run time and the number of requested resources. Therefore, large jobs have a larger resource consumption than smaller jobs. The resource consumption of a single job j is defined as follows:

$$\text{Resource\_Consumption}_j = (\text{reqResources}_j \cdot (\text{endTime}_j - \text{startTime}_j))$$

One measurement for the schedule quality is the average response time weighted by the job's resource consumption [77]. We define it as follows:

$$\text{AWRT} = \frac{\sum_{j \in \text{Jobs}} (\text{Resource\_Consumption}_j \cdot (\text{endTime}_j - \text{submitTime}_j))}{\sum_{j \in \text{Jobs}} \text{Resource\_Consumption}_j}$$

By weighting the response time of each job with its resource consumption, all jobs with the same resource consumption have the same influence on the schedule quality. Otherwise a small, often insignificant job would have the same impact on this metrics as a big job with the same response time.

Fragmentation Parameters The analysis of the used workloads shows a tendency of jobs to size with the power of 2 [11]. This is seen in Figure 7.13. Herein the resource consumption is summed up for all jobs of a specific width (i. e. the number of requested resources) for the syn_org workload. The other workloads show a similar behavior. Because of this "power of

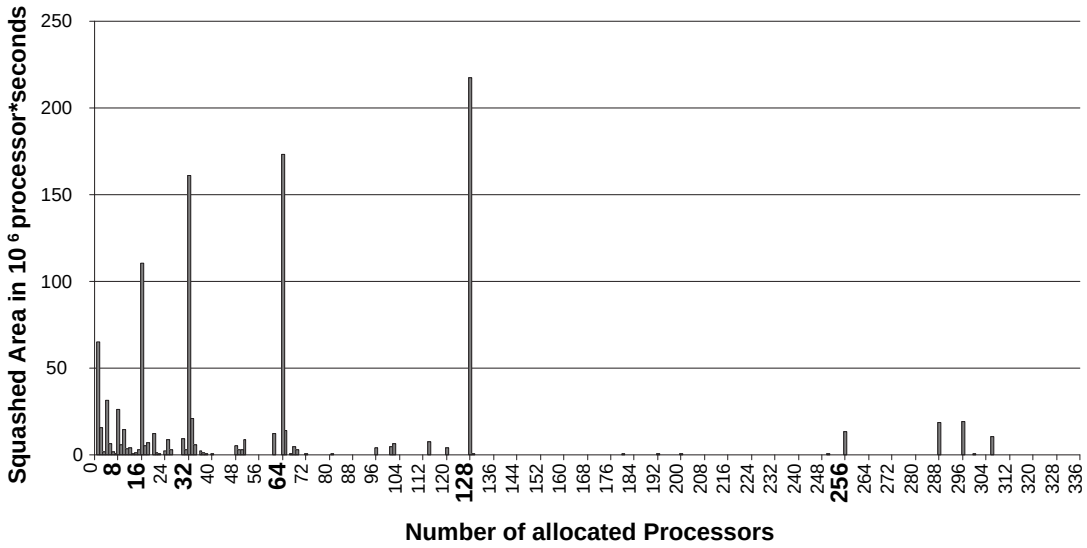


Figure 7.13: Distribution of the resource consumption (or squashed area) in the synthetic CTC workload.

2" focus we also used power of 2 values for the *lower bound* parameter. Jobs that fall short of this *lower bound* are only scheduled on a single machine and are not distributed across several sites. For the maximum number of job fragments we use a function defined over several intervals. We define two configurations with different values for each interval:

1. a *limited* configuration, wherein only necessary fragmentation of a job is allowed and
2. an *unlimited* configuration, which does not restrict the fragmentation process at all.

For the *unlimited* case the number of fragments is only limited by the maximum number of machines in the used resource configurations and the maximum number of requested resources for a specific job, as shown in Table 7.5.

requested resources		maximal number of fragments	
lower limit	upper limit	**limited configuration**	**unlimited configuration**
1	4	1	4
5	8	1	8
9	16	1	16
17	32	1	18
33	64	1	18
65	128	2	18
129	512	8	18

Table 7.5: Maximal fragmentation for the *limited* and *unlimited* configuration.

Impact of lower bounds All displayed results were achieved using the CTC_syn workload. As mentioned earlier this workload represents the average behavior of all used data sets. The results vary depending on the used workload, but only within close ranges.

The influence of the *lower bound* is highly dependent on the additional run time caused by the overhead. Obviously, without an overhead a multi-site execution would be beneficial in any case. Jobs which run unnecessarily in multi-site mode can be forced to run on a single machine. This reduces the impact of the overhead on the overall performance. Up to a certain level of overhead the use of multi-site (besides mandatory fragmentation to execute the job at all) leads to a better average weighted response time in our simulations.

This correlation is observed for the *m384* configuration and a *lower bound* of 128 as presented in Figure 7.14. A decrease of the average weighted response time by 16,3% for p = 80%, and by 29,9% for p = 120%. Here the *limited* and the *unlimited* configurations show no difference with regard to the average weighted response time.

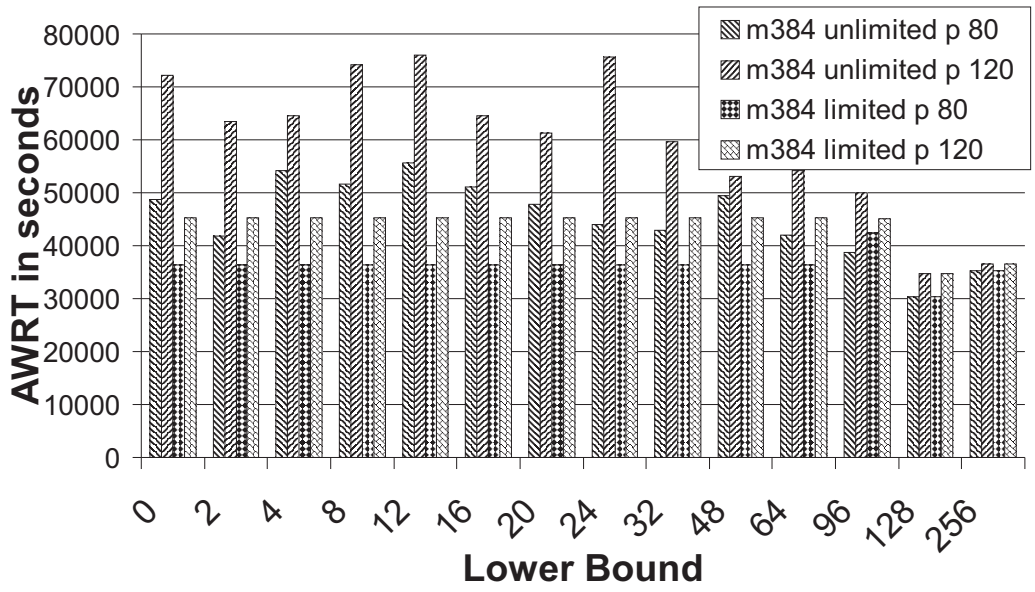


Figure 7.14: Influence of the *lower bound* for *m384* configuration.

Similar results are achieved for the *m256* configuration. Here a *lower bound* of 128 proves to be beneficial compared to any other values. In this configuration 128 is half the size of the biggest machine, which verifies the theorem, that it is beneficial for the scheduling that the job size stays below half the machine size, mentioned in [79]. For an overhead of p = 80% a reduction of about 12% is reached and for p = 120% 32% is reached.

For the *m384* configuration, it is seen that choosing an unappropriate *lower bound* may lead to a performance decrease, e. g. a *lower bound* of 256 results in an increased average weighted response time of 16% and a *lower bound* of 96 increases the average weighted response time by at least 29 % compared to a *lower bound* of 128 in the *m384* configuration, as shown in Figure 7.14.

An optimal lower bound is difficult to specify as it varies between the different machine configurations. In our simulations a lower bound of half the machine size proved to be beneficial in most cases. Exceptions are the configurations *m384* and the small configuration *m64*.

Comparison of limited and unlimited fragmentation In configurations with smaller machines e. g. *m64* and an overhead up to 60% the *unlimited* multi-site strategy is superior to the *limited* fragmentation strategy, as shown by the first four bars of each group in Figure 7.15.

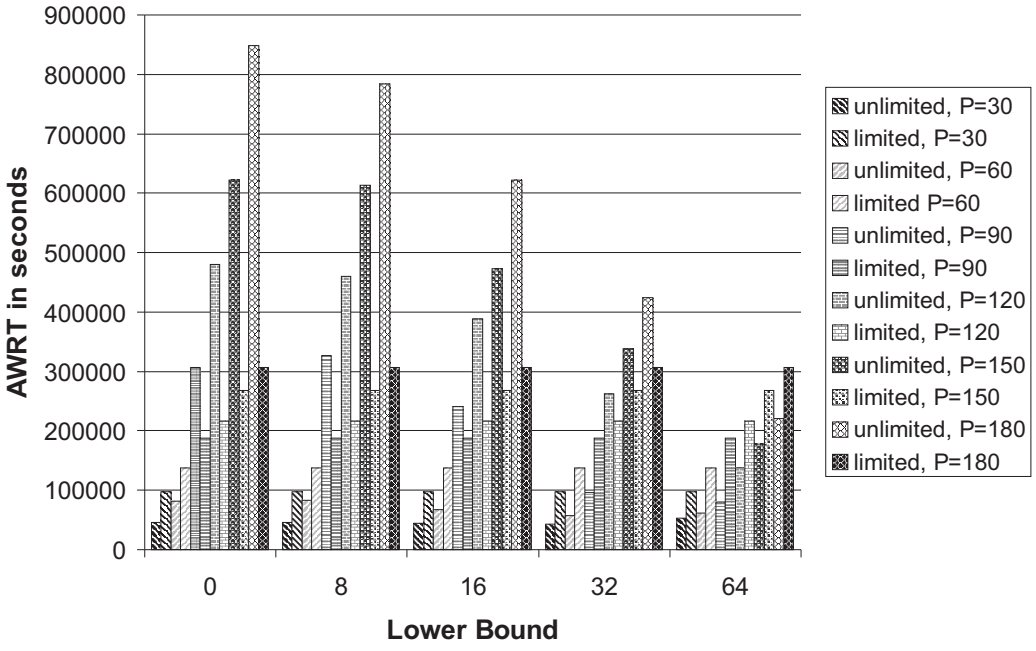


Figure 7.15: Comparison for *m64*.

In [15] and Section 7.3.2 we have presented a study of the impact of machine configurations on strategies, where multi-site execution is not limited. In comparison we evaluated enhancements of those strategies by using constraints and the new adaptive scheduler in the following.

Applying a higher number of fragments is beneficial. This can be seen in Figure 7.15, where the average weighted response time is decreased by about 60% for the *unlimited* scenario compared to the *limited* scenario for an overhead up to 60%. Whereas the *m64_8* configuration with equally sized machines shows a decrease of the average weighted response time of only around 10% for a higher fragmentation. In this scenario the *lower bound* has a higher impact. Major parts of the workload can only be scheduled in multi-site mode, as shown earlier in Figure 7.13. Hence, especially in resource configurations with smaller machines the average weighted response time scales directly with the overhead parameter p. Figure 7.15 underlines the advantages of unlimited fragmentation under the condition of a *lower bound* of 64 and an overhead of up to 180% for small machine configurations. In this case the average weighted response time may reach values of up to twelve times as high as the average weighted response time on a single large machine with 512 nodes.

Results for Adaptive Multi-Site Scheduling In the following we show only the lower bound for each configuration presenting the best achieved results. In Figure 7.16 the average weighted response time for the resource configurations *m512* and *m384* for the best non-adaptive and

the adaptive scheduling results are given. The improvements, by using adaptive scheduling in comparison to the non-adaptive case, are clearly seen. The average weighted response time, using the adaptive scheduling process, increases only by about 35% in comparison to the single site machine *m512* using an overhead of 300%. On the contrary, the average weighted response time of the non-adaptive scheduling system increases by about 244% with the same overhead of 300%.

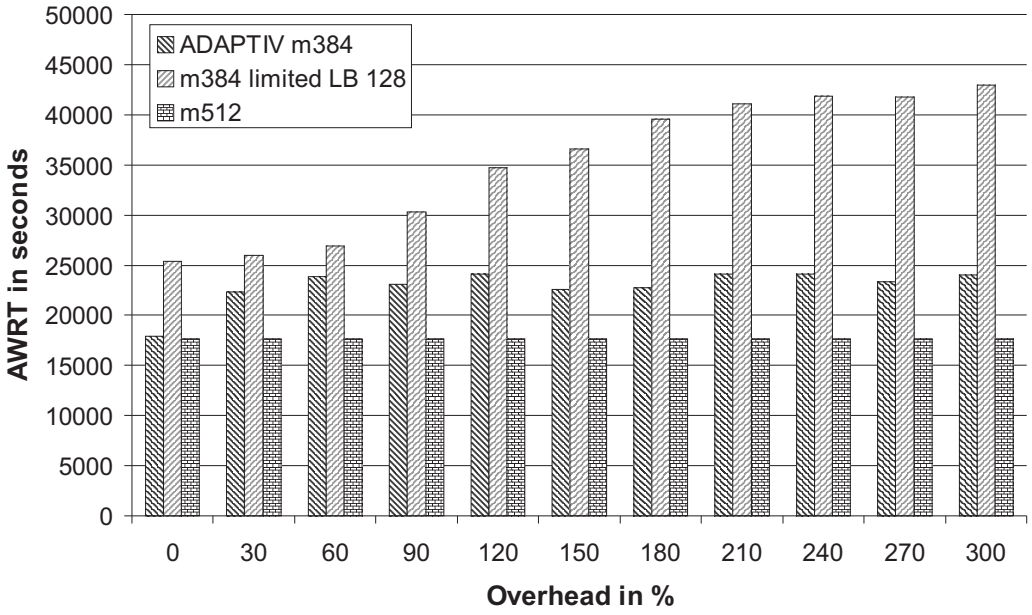


Figure 7.16: Comparison of adaptive and best non-adaptive schedules for m384.

Using the resource configuration *m384* even for the largest jobs, multi-site scheduling is not necessary, as the largest job within the workload requests only 336 nodes and can therefore be executed on the largest machine which consists of 384 nodes. Even with an overhead of 300%, using resource configuration *m384*, the adaptive scheduling system executes about 4% of all jobs in multi-site mode. This is only a 42% reduction compared to an overhead of 30% in the same configuration. Whereas the resource consumption of the multi-site jobs decreases to 30% in the same case. This indicates a shift towards the use of jobs with a smaller resource consumption for multi-site scheduling. All in all, increasing the overhead ten times from 30% to 300% only results in an increase of the average weighted response time of about 7% in this configuration.

In the best non-adaptive case increasing the overhead from 30% to 300% results in a 66% higher average weighted response time. Note, in the *m384* configuration with an, or even without, overhead the best non-adaptive multi-site scheduling system performs worse than the adaptive system with an overhead of 300% as seen in Figure 7.16. A similar behavior can be observed in all other resource configurations as shown in Table 7.6.

In Figure 7.17 all resource configurations with the adaptive scheduling system are compared. The average weighted response time for the resource configuration *m512*, given as a reference, is always constant as no multi-site scheduling is invoked. The results of configurations *m64*

resource configuration	number of multi-site jobs		Δ Jobs	Δ resource consumption
	overhead 30 %	overhead 300 %	in %	in %
m64	1069	430	-60	+176
m64-8	815	387	-53	+132
m128	478	337	-29	+88
m256	336	189	-44	≈ 0
m256-5	589	411	-30	+90
m384	637	368	-42	-71

Table 7.6: Comparison of adaptive configurations with overheads of 30% to 300% with regard to the alternation of the number and the resource consumption of multi-site jobs.

and *m64_8* show an increasing disadvantage which almost scales with the overhead from 30% to 300% up to a factor of about 15. However, the influence of the overhead on the overall performance for all other resource configurations is not as significant. The configurations *m128*, *m256*, and *m256_5* show a similar behavior in terms of performance. For overheads between 30% and 300%, none of these three configurations clearly outperforms any of the other two in the adaptive case as shown in Figure 7.17.

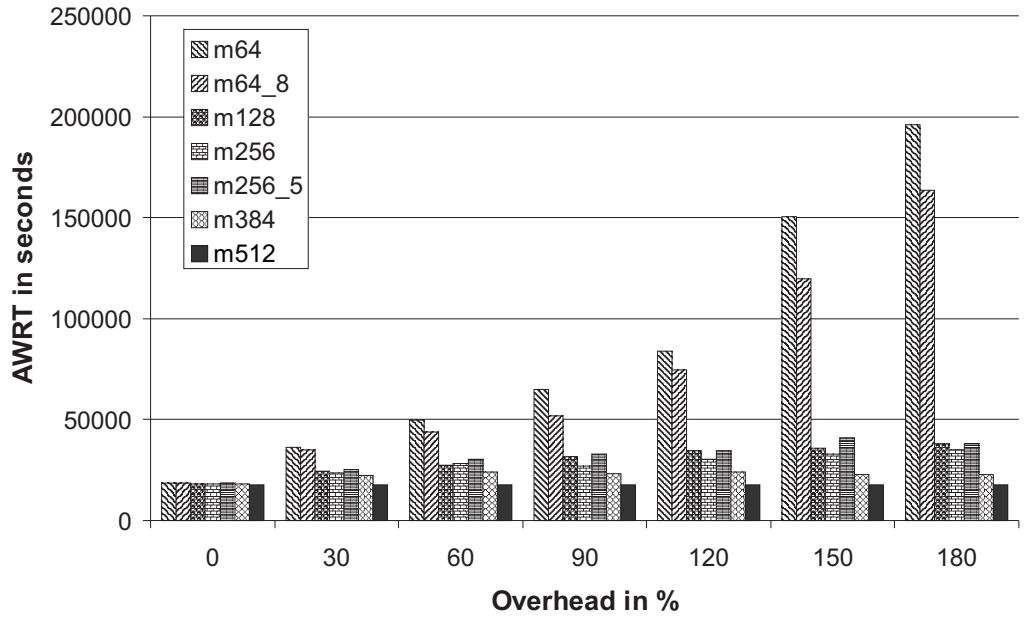


Figure 7.17: Comparison of adaptive schedules.

This figure also displays the influence of different machine configurations. The equal partitioned configuration *m64_8* outperforms its non-balanced counterpart *m64*. This is due to the fact that larger machines are available for the execution of large jobs, leading to more flexibility for the overall scheduling process. The most significant impact of the adaptive scheduling system is observed in the results of the resource configuration *m384*. Most resources are combined in a single machine and therefore no multi-site scheduling is necessary. Here the multi-site scheduling is only used to enhance the quality of the overall schedule.

Concluding Remarks

We introduced the *lower bound* parameter in order to restrict the multi-site execution to jobs which require, at the least, a certain number of resources. Setting the *lower bound* resulted in a significant improvement of up to 30% in the average weighted response time for certain scenarios. Additionally, we restricted the number of fragments to be generated by the multi-site scheduler. For communication overheads with over 60% of the original execution time it proved to be beneficial to limit the fragmentation process.

We examined a different approach by using an adaptive version of the multi-site scheduler. Here the scheduler compares the completion times of single- and multi-site execution. For single-site execution the waiting time, until the job can be started, has to be considered. Whereas for multi-site the overhead has to be added to the run time. Depending on the result the job either is added to the queue or is directly started in the multi-site mode. The evaluation showed that the adaptive version substantially improved the performance of the multi-site scheduler, regardless of the overhead percentage for the multi-site execution that was chosen. Therefore, adaptive multi-site scheduling seems to be the best algorithm for a multi-site environment. As the presented adaptive algorithm is a basic implementation, more sophisticated adaptive algorithms may further increase the performance.

7.4 Summary

We used a simulation environment to evaluate and compare the performance of different grid scheduling scenarios: local job processing, job sharing, and multi-site execution. As job input we extracted job sets from real trace logs. Additionally, different machine configurations were evaluated in order to measure the impact of different machine sizes and the way multi-site jobs are distributed across the grid.

The results show, that the participation of sites in a grid environment pays off. Even without the option to start jobs in multi-site mode the average weighted response time is significantly improved in the job sharing scenario with a central grid scheduler. Observing the structure of the grid, i. e. the size of participating machines, shows, that configurations with large machines are obviously superior to configurations with smaller machines. However, configurations with smaller machines have only minor drawbacks, if the resource demands of the jobs are limited to 50% of the largest machine in the grid. For example, comparing four 128 node machines with the reference case of a single 512 machine shows, that the average weighted response time is increased by at most 15%. We expect, that an acceptable response time is guaranteed, if the ratio between the workload of large jobs and the available computing power of the large machine is adequate. As long as this requirement is met, the remaining resources may consist of smaller machines without implying a significant drawback.

In contrast to job sharing, the usage of multi-site scheduling allows the execution of jobs that are not limited by the maximum machine size. The use of multi-site applications leads to even better results under the assumption that a limited increase on job execution time is due to communication overheads. Even with an increase of the execution time by 25%, multi-site is beneficial compared to job sharing. In terms of latency, WAN networks are in the order of 2-3 magnitudes slower than common fast interconnection networks between nodes inside a parallel computer, e. g. an IBM SP Switch. Therefore, it can not be concluded that multi-site is suitable for all applications. However, multi-site is beneficial for applications with a limited demand in communication. As showed in Section 7.3.1 the resource configurations and the

overhead due to multi-site scheduling have a strong impact on the average weighted response time of the schedule. Similar to job sharing configurations, larger machines are superior to those with smaller machines. Though scenarios with small overheads ($p < 20\%$) only show a small advantage of balanced large machine configurations compared to unbalanced systems with small machines.

Finally, in Section 7.3.3 we examined three enhancements of the multi-site scheduler for grid computing environments. If a job can not be placed directly on a single machine, the multi-site scheduler splits the original job in several fragments which are then started synchronously on multiple sites/machines. The communication between job fragments and the additional effort of data migration is considered in extending the jobs run time by a given percentage. The *lower bound* parameter is used to restrict the multi-site execution to jobs requiring at least a certain number of resources. Setting the *lower bound* appropriately, results in a significant improvement of up to 30% in the average weighted response time for certain scenarios. Additionally, the number of fragments to be generated by the multi-site scheduler are restricted. For communication overheads with over 60% of the original execution time it proves to be beneficial to limit the fragmentation process.

A different approach was done by using an adaptive version of the multi-site scheduler. Here the scheduler compares the completion times of single- and multi-site execution. For single-site execution the waiting time, until the job can be started, has to be considered, whereas for multi-site the overhead has to be added to the run time. Depending on the result of the comparison, the job is either stored in the waiting queue or it is directly started in multi-site mode. The evaluation shows, that the adaptive version substantially improves the performance of the multi-site scheduler, regardless of what overhead percentage for the multi-site execution is chosen. Therefore, adaptive multi-site scheduling seems to be the best algorithm for a multi-site environment. As the presented adaptive algorithm is a basic implementation, more sophisticated adaptive algorithms may further increase the performance.

The applied algorithms for grid scheduling are simple extensions of backfilling and node selection strategies. Additional research on more sophisticated scheduling algorithms is necessary, which in turn may produce even better results. It has to be kept in mind, that the quality of a schedule depends on the actual configuration and workload. The improvements presented in this thesis were achieved using example configurations and workloads derived from real traces. The outcome may vary in other settings. Nevertheless, the results show, that job sharing and multi-site execution in a grid environment are capable of significantly improving the scheduling quality for the users. In any case, the participation in grids is beneficial, as the use of multi-site computing enables the execution of jobs that need more resources than available on the largest single machine in such a grid environment.

8 Conclusion

In this thesis we presented self-tuning schedulers, which dynamically adapt their scheduling behavior to the current situation. We developed such schedulers for two areas of resource management: single high performance computing (HPC) machines and computational grid environments. The self-tuning dynP[1] scheduler for single HPC machines switches its active policy according to the characteristics of currently waiting jobs. As we assumed a planning based resource management system, the scheduler is able to compute full schedules for each implemented policy. Full schedules contain exact start and end times for all waiting jobs. Each schedule can be measured with a performance metrics, so that the scheduler can rank the available scheduling policies according to their performance. Then the self-tuning dynP scheduler switches to the best policy. For scheduling in a computational grid we defined three scenarios: local job processing, job sharing, and multi-site job execution. If not enough resources are available to place a job on a single machine immediately, the multi-site grid scheduler can use multiple machines for executing the job. As machines in grid environments are typically geographically distributed, slow wide are networks (WAN) are involved and this has to be considered during scheduling. This is done by increasing the execution time of the job appropriately. A decision has to be made when scheduling newly submitted jobs. The adaptive multi-site grid scheduler either waits until enough resources are available at a single site, or starts the job immediately on multiple sites and consider the WAN communication overhead.

We evaluated the self-tuning dynP scheduler by means of discrete event simulations. As job input we used different traces from real machine installations at supercomputing centers. We measured the performance with the slowdown weighted by area as a user centric metrics and the system utilization as an owner centric metrics. Unfortunately, half of the available traces generated low utilizations and corresponding empty schedules. Jobs do not have to wait for execution and no significant results are generated. The measured slowdowns are close to their minimum of one. The results of the remaining traces CTC, KTH, LANL, and SDSC show, that the self-tuning dynP scheduler with the advanced decider improves the slowdown and utilization compared to the best basic policy (SJF for the CTC and KTH trace, and FCFS for the LANL and SDSC trace). The evaluation of the decision process in the self-tuning dynP scheduler shows, that restricting the dynamic policy switching in any way (e. g. by adding slackness to the decision process or by reducing the tests for policy switching) does not improve the performance.

We observed similar results, if the workload is increased by reducing the average interarrival time with a shrinking factor. If the performance improvement is averaged over the applied range of workloads, the self-tuning dynP scheduler improves the slowdown by about nine percent for the CTC jobs and six percent for the SDSC jobs. At the same time, the utilization is increased by about one percentage-point for both job sets. The improvements seem to be small, but with utilizations in the saturated state of around 90% an improvement to 91%

[1] **dyn**amic **P**olicy switching

means, that the amount of unused resources is reduced by one tenth. At high workloads it is much more difficult for the scheduler to improve the utilization than at low workloads.

As a quasi off-line scheduling is done in each self-tuning step, we also compared the policies' generated schedules with optimal schedules. For this purpose, we modelled the scheduling problem as an integer problem and solved it with the CPLEX library. We applied time-scaling in order to reduce the problem size and to make the problem computable. By that, CPLEX does no longer compute optimal schedules with a second-precise scaling. The comparison shows, that on average the best basic policy in each self-tuning step is less than one percent worse than the solution computed by CPLEX. Because of time-scaling cases occur,in which the solution computed by CPLEX is slightly worse than the best basic scheduling policy. This is a result of time-scaling. Besides requiring a lot of main memory, CPLEX needs a lot of computational time[2]. Therefore, using CPLEX to compute schedules it not feasible for the scheduling in real world resource management systems.

We evaluated the three mentioned grid scheduling scenarios with different machine configurations and the results show, that overall a participation in computational grid environments pays off. Even without the option of starting jobs multi-site, the performance measured as average weighted response time is improved significantly in the job sharing scenario. Considering the structure of the grid, i. e. the size of participating machines, shows, that configurations with large machines are superior to equivalent configurations with small machines. Performance drops for configurations with small machines, if the resource requirements of the jobs exceed the size of the largest machine in the grid. We expect, that an adequate ratio between the workload of large jobs and the available computing power of the large machines, guarantees an acceptable response time. As long as this requirement is met, the remaining resources may consist of smaller machines without implying a significant drawback. Multi-site scheduling allows the execution of jobs that are not limited by the maximum machine size. The usage of multi-site applications leads to even better results assuming that the communication overhead is limited to 25% for most of the evaluated scenarios.

We applied further enhancements to the multi-site scheduling scenario and evaluated its influence on the performance. The lower bound parameter is used to restrict the multi-site execution to jobs requiring at least a certain number of resources. Setting the lower bound appropriately, results in a significant improvement of up to 30% for the average weighted response time for certain machine configurations. In addition, we restricted the generation of job parts. The results show, that for communication overheads with over 60% of the original execution time, it proves to be beneficial to limit the fragmentation process. The evaluation of the adaptive multi-site scheduler shows, that the adaptive version substantially improves the performance of the multi-site scheduler for all applied communication overheads.

In general, the evaluation results indicate, that the use of small systems in combination with multi-site scheduling can not perform as well as a single large machine with the same amount of resources. However, the presented algorithms decrease this difference. In any case, the participation in computational grid environments seems to be beneficial. The use of multi-site computing enables the execution of jobs that consume more resources than available on the largest single machine in the grid.

The presented areas of job scheduling are different at a first glance. However, a closer view on the scheduling approaches reveals, that both schedulers search for the best solution in a dynamic way and only different terms are used to describe this functionality: "self-tuning"

[2] on average more than five hours for a small schedule of 22 jobs and a makespan of two days

and "adaptive". In this work we showed that dynamic scheduling approaches are beneficial. Therefore, we think that dynamic scheduling approaches, as presented in this thesis, should be implemented in modern resource management systems and grid middleware.

Future Work

Future work in both areas may include support for advanced reservations. If users are allowed to reserve resource in a single machine scenario, the resource management scheduler is significantly limited in its abilities to plan a schedule. Once accepted, reservations can not be moved on the time axis and standard batch jobs have to be planned around these fixed jobs. Among other aspects it is of special interest to evaluate in which way the scheduler is limited. For example, up to what percentage of reservations in the workload and utilization of the machine is the scheduler able to handle reservations without neglecting normal batch jobs?

Planning based resource management systems include advanced reservations in their design. Additional functionality is thinkable for such modern resource management systems, which guarantee the resource usage. We gave some initial thoughts for such extensions in [42]. In the context of computational grids the support for Service Level Agreements (SLA) is important, as not only a single resource usage can be guaranteed, but a general relationship between the resource owner and user is specified.

Particularly, for the multi-site job execution scenario in grids, the ability to use reservations would be a major improvement for the quality of the provided resource usage. If the adaptive multi-site scheduler decides to wait for a single-site job start, the grid scheduler simply reserves the appropriate number of resources and therefore the response time of the waiting job is guaranteed. An evaluation has to show, that this actually increases the performance, because at the same time reservations might constrain the local scheduler in their abilities.

A Detailed Results

A.1 Original Traces

The following two sections contain the detailed simulation results from Section 5.1.1 to Section 5.1.4. The performance of a scheduling strategy is measured by the average slowdown weighted by area (SLDwA). The eight original traces are used as job input for simulation runs. With the exact start dates of Table 3.5 an analysis for each month of the trace is done. Blue and green colors are used in the following to emphasize interesting results, which are addressed in the text.

A.1.1 Basic Policies

month	year	jobs	average width	average actual run time	average estimated run time	over-estimation factor	SLDwA FCFS	SLDwA SJF	SLDwA LJF	best policy
6	1996	37	32.1	29,038	50,181	1.73	1.0000	1.0000	1.0000	-
7	1996	7,954	9.8	9,786	22,732	2.32	1.9326	1.7131	2.2870	SJF
8	1996	7,302	11.6	11,148	24,859	2.23	1.8303	1.8554	2.3735	FCFS
9	1996	6,188	11.3	13,237	26,637	2.01	2.3081	2.1746	2.7575	SJF
10	1996	7,288	10.1	9,106	20,857	2.29	1.8861	1.6894	2.0935	SJF
11	1996	7,874	10.8	10,027	22,451	2.24	1.6421	1.4862	2.0023	SJF
12	1996	7,881	9.6	8,810	22,016	2.50	1.4895	1.3953	1.9351	SJF
1	1997	7,535	8.9	9,989	25,944	2.60	2.0089	1.9857	2.2119	SJF
2	1997	8,190	9.3	11,766	25,344	2.15	1.7852	1.9870	1.8523	FCFS
3	1997	6,946	12.2	13,913	28,112	2.02	2.4485	2.0449	3.3134	SJF
4	1997	6,129	12.3	11,130	22,059	1.98	2.5380	2.5880	3.5569	FCFS
5	1997	5,978	13.2	12,632	27,449	2.17	2.5650	2.2275	3.2364	SJF
		average	10.7	10,958	24,324	2.22	2.0455	1.9277	2.5212	SJF

Table A.1: Monthly analysis of the CTC trace: submitted job data and performance (average slowdown weighted by job area) of the three basic policies.

month	year	jobs	average width	average actual run time	average estimated run time	over-estimation factor	SLDwA FCFS	SLDwA SJF	SLDwA LJF	best policy
9	1996	108	11.9	5,549	10,501	1.89	1.1035	1.1035	1.1035	-
10	1996	2,405	12.4	6,010	9,075	1.51	4.1895	3.4751	6.9391	SJF
11	1996	1,991	10.2	8,324	13,777	1.65	4.4634	3.6699	12.6893	SJF
12	1996	2,302	9.1	9,760	14,917	1.53	3.7131	3.2411	5.4525	SJF
1	1997	2,938	7.9	9,272	14,319	1.54	3.0440	2.2562	4.6119	SJF
2	1997	2,916	7.3	6,548	10,158	1.55	4.9779	3.5847	10.4477	SJF
3	1997	2,081	7.8	8,661	13,711	1.58	3.3362	3.0692	9.0681	SJF
4	1997	2,861	6.5	8,532	12,735	1.49	2.9512	2.1044	4.8270	SJF
5	1997	4,081	6.7	6,396	10,241	1.60	2.4835	2.1072	3.5145	SJF
6	1997	2,697	5.3	9,752	14,954	1.53	1.6373	1.5597	1.8178	SJF
7	1997	2,184	5.8	16,297	24,725	1.52	1.3082	1.2669	1.4926	SJF
8	1997	1,923	6.3	11,165	16,711	1.50	1.7919	1.5389	2.3467	SJF
		average	7.7	8,858	13,678	1.54	3.1015	2.5488	5.8118	SJF

Table A.2: Monthly analysis of the KTH trace: submitted job data and performance (average slowdown weighted by job area) of the three basic policies.

month	year	jobs	average width	average actual run time	average estimated run time	over-estimation factor	SLDwA FCFS	SLDwA SJF	SLDwA LJF	best policy
10	1994	6,134	81.3	1,274	2,081	1.63	1.5804	1.5466	2.2557	SJF
11	1994	6,014	90.3	1,429	2,114	1.48	1.7291	1.6193	2.3306	SJF
12	1994	4,595	103.3	1,576	2,357	1.50	1.7419	1.5588	2.2388	SJF
1	1995	3,472	118.6	1,784	2,802	1.57	1.6565	1.4963	2.0059	SJF
2	1995	3,877	99.5	1,618	2,603	1.61	1.6990	1.6551	1.9483	SJF
3	1995	4,885	88.9	1,495	2,394	1.60	1.5247	1.4035	1.7781	SJF
4	1995	4,651	102.1	1,845	2,832	1.53	1.6025	1.4648	1.9032	SJF
5	1995	4,778	108.6	1,861	3,089	1.66	1.9210	1.8521	2.5048	SJF
6	1995	5,637	103.6	1,777	3,089	1.74	2.6770	2.9987	4.2842	FCFS
7	1995	6,301	90.6	1,579	2,887	1.83	1.9940	2.0094	2.5278	FCFS
8	1995	6,261	83.1	1,250	2,124	1.70	1.5758	1.4076	1.8026	SJF
9	1995	5,714	110.3	1,734	3,017	1.74	2.1396	2.8997	2.7245	FCFS
10	1995	5,048	115.3	1,692	3,078	1.82	1.6669	1.5212	2.3309	SJF
11	1995	3,649	137.6	2,328	3,422	1.47	1.5907	1.8039	1.8494	FCFS
12	1995	4,151	114.9	2,071	3,826	1.85	1.4968	1.4963	1.6288	SJF
1	1996	4,163	109.4	1,923	3,443	1.79	1.2398	1.2222	1.3118	SJF
2	1996	3,697	105.7	1,739	2,821	1.62	1.1414	1.1501	1.1585	FCFS
3	1996	4,129	106.8	2,215	3,534	1.60	1.4335	1.3567	1.6069	SJF
4	1996	4,249	101.6	2,187	3,632	1.66	1.5114	1.5449	1.6258	FCFS
5	1996	4,954	97.2	2,150	3,342	1.55	1.5336	1.5167	1.7672	SJF
6	1996	5,240	102.8	1,751	5,399	3.08	1.3084	1.3525	1.3238	FCFS
7	1996	14,627	123.3	708	8,341	11.78	2.1841	2.2640	2.2157	FCFS
8	1996	3,668	110.9	2,234	3,695	1.65	1.3515	1.4127	1.4061	FCFS
9	1996	2,411	115.0	3,057	3,903	1.28	1.1977	1.2616	1.2217	FCFS
		average	104.95	1,659	3,683	2.22	1.6801	1.7031	2.0507	FCFS

Table A.3: Monthly analysis of the LANL trace: submitted job data and performance (average slowdown weighted by job area) of the three basic policies.

month	year	jobs	average width	average actual run time	average estimated run time	over-estimation factor	SLDwA FCFS	SLDwA SJF	SLDwA LJF	best policy
4	1998	5	6.2	145,434	145,440	1.00	1.0000	1.0000	1.0000	-
5	1998	2,767	14.7	7,706	18,028	2.34	3.3645	2.5552	4.8491	SJF
6	1998	2,535	9.2	8,601	25,724	2.99	2.0738	1.9523	2.6489	SJF
7	1998	2,832	7.4	6,999	17,509	2.50	2.3979	2.1876	2.8259	SJF
8	1998	3,618	6.0	6,666	15,230	2.28	3.1907	2.3388	4.1563	SJF
9	1998	12,536	3.1	2,621	7,438	2.84	3.2224	2.6290	5.5883	SJF
10	1998	4,491	5.9	6,063	11,122	1.83	2.3432	1.9787	2.9487	SJF
11	1998	3,105	12.4	6,915	20,576	2.98	2.8461	2.6827	3.4484	SJF
12	1998	2,894	13.1	7,456	17,524	2.35	6.1737	4.1766	9.7122	SJF
1	1999	2,802	9.3	7,836	20,174	2.57	4.1467	3.3003	12.6212	SJF
2	1999	2,720	9.7	6,275	15,643	2.49	6.8112	7.0362	14.0235	FCFS
3	1999	2,914	12.4	6,536	13,033	1.99	7.2553	7.0065	15.2942	SJF
4	1999	3,713	13.4	3,610	10,204	2.83	9.3420	5.9265	21.9462	SJF
5	1999	2,515	19.6	5,855	12,732	2.17	12.3298	7.3162	92.1742	SJF
6	1999	2,447	12.5	7,407	15,045	2.03	13.5694	11.9581	69.2463	SJF
7	1999	1,252	14.1	10,236	19,264	1.88	7.3964	11.2254	45.3261	FCFS
8	1999	1,934	12.9	7,554	15,257	2.02	8.0159	16.0247	24.4480	FCFS
9	1999	2,122	12.9	6,328	13,963	2.21	7.6722	22.3486	20.4491	FCFS
10	1999	1,957	16.6	5,636	13,126	2.33	9.6773	25.3462	102.4740	FCFS
11	1999	2,039	13.5	4,682	11,161	2.38	10.4231	27.0653	34.1716	FCFS
12	1999	1,695	22.0	5,675	11,065	1.95	11.2749	20.8298	48.5418	FCFS
1	2000	1,496	15.9	8,184	20,991	2.56	6.3182	21.1915	35.1660	FCFS
2	2000	1,140	14.6	10,313	20,143	1.95	8.1268	28.7438	15.4552	FCFS
3	2000	1,170	17.6	8,314	17,198	2.07	4.4993	37.3391	7.5649	FCFS
4	2000	932	18.7	11,742	24,730	2.11	5.9322	14.0702	9.6173	FCFS
		average	10.54	6,077	14,344	2.36	6.8260	12.5662	26.8207	FCFS
only from 4-1998 until 6-1999										
		average	10.33	15,732	24,362	2.35	5.3378	4.2696	17.4989	SJF

Table A.4: Monthly analysis of the SDSC trace: submitted job data and performance (average slowdown weighted by job area) of the three basic policies.

month	year	jobs	average width	average actual run time	average estimated run time	over-estimation factor	SLDwA FCFS	SLDwA SJF	SLDwA LJF	best policy
1	2001	1,508	8.1	6,884	41,548	6.04	1.0000	1.0000	1.0000	-
2	2001	1,549	15.5	2,140	6,966	3.25	1.0000	1.0000	1.0000	-
3	2001	2,610	6.1	2,821	6,195	2.20	1.0000	1.0000	1.0000	-
4	2001	3,184	7.7	2,684	6,455	2.40	1.1549	1.1704	1.2058	FCFS
5	2001	2,462	9.9	2,799	9,030	3.23	1.0000	1.0000	1.0000	-
6	2001	3,564	4.1	3,624	14,099	3.89	1.0010	1.0010	1.0010	-
7	2001	6,396	2.6	7,315	11,554	1.58	1.0011	1.0011	1.0011	-
8	2001	5,789	4.1	4,565	10,141	2.22	1.0000	1.0000	1.0000	-
9	2001	827	10.3	8,942	29,676	3.32	1.0000	1.0000	1.0000	-
10	2001	2,038	8.7	3,573	14,353	4.02	1.0000	1.0000	1.0000	-
11	2001	3,732	7.5	2,369	6,545	2.76	1.6354	1.5399	1.5817	SJF
12	2001	1,435	8.4	4,428	12,342	2.79	1.0040	1.0038	1.0041	SJF
		average	6.34	4,346	11,717	2.70	1.0846	1.0758	1.0838	SJF

Table A.5: Monthly analysis of the PC2-2001 trace: submitted job data and performance (average slowdown weighted by job area) of the three basic policies.

month	year	jobs	average width	average actual run time	average estimated run time	over-estimation factor	SLDwA FCFS	SLDwA SJF	SLDwA LJF	best policy
1	2002	664	16.7	2,929	10,610	3.62	1.0000	1.0000	1.0000	-
2	2002	8,068	11.5	1,727	4,141	2.40	1.1005	1.0837	1.1265	-
3	2002	3,847	10.5	4,033	10,719	2.66	2.2477	2.2103	2.6207	SJF
4	2002	1,660	15.9	4,424	10,079	2.28	1.0002	1.0002	1.0002	-
5	2002	1,197	12.2	5,865	12,275	2.09	1.0000	1.0000	1.0000	-
6	2002	1,762	2.6	6,607	76,858	11.63	1.0000	1.0000	1.0000	-
7	2002	3,478	3.5	5,831	98,398	16.88	1.0000	1.0000	1.0000	-
8	2002	1,398	8.7	5,736	56,154	9.79	1.0010	1.0010	1.0010	-
9	2002	1,064	9.9	26,008	62,398	2.40	1.0000	1.0000	1.0000	-
10	2002	2,862	3.9	19,658	32,371	1.65	1.0000	1.0000	1.0000	-
11	2002	4,339	4.4	5,530	54,812	9.91	1.0000	1.0000	1.0000	-
12	2002	1,873	3.8	5,136	14,522	2.83	1.0000	1.0000	1.0000	-
		average	8.14	6,310	33,942	5.38	1.0798	1.0765	1.1033	SJF

Table A.6: Monthly analysis of the PC2-2002 trace: submitted job data and performance (average slowdown weighted by job area) of the three basic policies.

month	year	jobs	average width	average actual run time	average estimated run time	over-estimation factor	SLDwA FCFS	SLDwA SJF	SLDwA LJF	best policy
4	2000	618	8.3	20,280	61,719	3.04	1.0000	1.0000	1.0000	-
5	2000	823	10.8	17,611	52,997	3.01	1.0008	1.0008	1.0008	-
6	2000	1,302	15.5	37,113	103,226	2.78	1.0036	1.0035	1.0036	SJF
7	2000	1,253	6.8	48,410	141,998	2.93	1.0000	1.0000	1.0000	-
8	2000	945	4.7	49,405	200,005	4.05	1.0000	1.0000	1.0000	-
9	2000	897	5.8	71,779	170,934	2.38	1.0000	1.0000	1.0000	-
10	2000	1,423	5.8	76,528	178,308	2.33	1.0000	1.0000	1.0000	-
11	2000	2,323	4.9	25,585	66,726	2.61	1.0000	1.0000	1.0000	-
12	2000	2,155	4.8	25,221	79,424	3.15	1.0000	1.0000	1.0000	-
1	2001	1,606	6.0	42,368	120,474	2.84	1.0000	1.0000	1.0000	-
2	2001	1,523	3.2	56,187	223,146	3.97	1.0000	1.0000	1.0000	-
3	2001	1,860	4.1	65,138	333,267	5.12	1.0000	1.0000	1.0000	-
4	2001	2,277	3.0	75,309	317,014	4.21	1.0000	1.0000	1.0000	-
5	2001	578	4.1	35,980	171,379	4.76	1.0000	1.0000	1.0000	-
		average	5.80	47,838	168,024	3.51	1.0038	1.0037	1.0038	SJF

Table A.7: Monthly analysis of the CHPC trace: submitted job data and performance (average slowdown weighted by job area) of the three basic policies.

month	year	jobs	average width	average actual run time	average estimated run time	over-estimation factor	SLDwA FCFS	SLDwA SJF	SLDwA LJF	best policy
2	1998	18	43.9	15,426	29,133	1.89	2.2233	2.2394	2.2233	FCFS, LJF
3	1998	3,249	7.9	6,196	20,029	3.23	1.1581	1.1328	1.1572	SJF
		average	8.08	6,246	20,079	3.21	1.2362	1.2319	1.2353	SJF

Table A.8: Monthly analysis of the MHPCC trace: submitted job data and performance (average slowdown weighted by job area) of the three basic policies.

A.1.2 Self-Tuning dynP Scheduler

month	year	best policy		simple half	simple full	advanced half	advanced full	advanced better than simple half	advanced better than simple full	difference of applying half self-tuning simple	difference of applying half self-tuning advanced
6	1996	1.0000	-	1.0000	1.0000	1.0000	1.0000	0.00%	0.00%	0.00%	0.00%
7	1996	1.7131	SJF	2.0379	2.0420	1.7544	1.7155	13.91%	15.99%	0.20%	-2.27%
8	1996	1.8303	FCFS	2.2397	2.3152	1.9661	1.9473	12.22%	15.89%	3.26%	-0.96%
9	1996	2.1746	SJF	2.5391	2.6146	2.1414	2.1422	15.67%	18.07%	2.89%	0.04%
10	1996	1.6894	SJF	1.9280	1.9876	1.7261	1.7267	10.47%	13.13%	3.00%	0.04%
11	1996	1.4862	SJF	1.9898	2.0481	1.5469	1.5508	22.26%	24.28%	2.85%	0.25%
12	1996	1.3953	SJF	1.8049	1.7208	1.4118	1.4133	21.78%	17.87%	-4.89%	0.10%
1	1997	1.9857	SJF	2.0460	1.9729	1.9383	1.9008	5.27%	3.66%	-3.71%	-1.97%
2	1997	1.7852	FCFS	1.8126	1.7827	1.7922	1.7751	1.12%	0.43%	-1.67%	-0.96%
3	1997	2.0449	SJF	2.8573	2.6035	2.1001	2.0949	26.50%	19.54%	-9.75%	-0.25%
4	1997	2.5380	FCFS	3.2636	2.9913	2.4753	2.2878	24.15%	23.52%	-9.10%	-8.19%
5	1997	2.2275	SJF	2.7334	2.9875	2.0737	2.0784	24.13%	30.43%	8.50%	0.23%
average		1.9277	SJF	2.3036	2.2834	1.9085	1.8809	17.15%	17.63%	-0.88%	-1.47%
relative to best policy				-19.50%	-18.45%	0.99%	2.43%				

Table A.9: Monthly comparison of the simple and advanced decider for the CTC trace: ARTwW is used as the quality metrics in the self-tuning process. The overall given performance is SLDwA. Blue font indicates a better performance than the best basic policy SJF.

month	year	best policy		simple half	simple full	advanced half	advanced full	advanced better than simple half	advanced better than simple full	difference of applying half self-tuning simple	difference of applying half self-tuning advanced
9	1996	1.1035	-	1.1035	1.1035	1.1035	1.1035	0.00%	0.00%	0.00%	0.00%
10	1996	3.4751	SJF	6.0425	8.6447	3.5121	3.3737	41.88%	60.97%	30.10%	-4.10%
11	1996	3.6699	SJF	9.2196	12.8200	3.7814	4.0015	58.99%	68.79%	28.08%	5.50%
12	1996	3.2411	SJF	4.8263	4.7933	3.2393	2.9323	32.88%	38.83%	-0.69%	-10.47%
1	1997	2.2562	SJF	4.1738	4.1939	2.3723	2.4332	43.16%	41.98%	0.48%	2.50%
2	1997	3.5847	SJF	7.1177	9.1039	3.6633	3.7126	48.53%	59.22%	21.82%	1.33%
3	1997	3.0692	SJF	8.1751	9.5222	3.1509	3.2291	61.46%	66.09%	14.15%	2.42%
4	1997	2.1044	SJF	3.9127	4.1480	2.0341	2.1745	48.01%	47.58%	5.67%	6.46%
5	1997	2.1072	SJF	2.7665	2.9853	2.0420	2.0007	26.19%	32.98%	7.33%	-2.07%
6	1997	1.5597	SJF	1.6634	1.7336	1.5481	1.5531	6.93%	10.41%	4.05%	0.33%
7	1997	1.2669	SJF	1.4253	1.4406	1.2983	1.3000	8.91%	9.76%	1.06%	0.13%
8	1997	1.5389	SJF	1.9747	2.0339	1.5774	1.5643	20.12%	23.09%	2.91%	-0.84%
average		2.5488	SJF	4.72559	5.65619	2.58847	2.58121	45.22%	54.36%	16.45%	-0.28%
relative to best policy				-85.41%	-121.92%	-1.56%	-1.27%				

Table A.10: Monthly comparison of the simple and advanced decider for the KTH trace: ARTwW is used as the quality metrics in the self-tuning process. The overall given performance is SLDwA. Blue font indicates a better performance than the best basic policy SJF.

month	year	best policy		simple half	simple full	advanced half	advanced full	advanced better than simple half	advanced better than simple full	difference of applying half self-tuning simple	difference of applying half self-tuning advanced
10	1994	1.5466	SJF	1.9162	1.8143	1.5080	1.5128	21.30%	16.62%	-5.62%	0.32%
11	1994	1.6193	SJF	1.9093	1.9441	1.5879	1.5842	16.83%	18.51%	1.79%	-0.24%
12	1994	1.5588	SJF	1.9844	2.0052	1.6182	1.6163	18.45%	19.39%	1.04%	-0.12%
1	1995	1.4963	SJF	1.7093	1.7536	1.5368	1.5145	10.09%	13.64%	2.53%	-1.47%
2	1995	1.6551	SJF	1.7331	1.7259	1.5821	1.5796	8.71%	8.48%	-0.41%	-0.16%
3	1995	1.4035	SJF	1.5097	1.4858	1.3983	1.3933	7.38%	6.23%	-1.61%	-0.36%
4	1995	1.4648	SJF	1.7143	1.6623	1.4983	1.4974	12.60%	9.92%	-3.13%	-0.06%
5	1995	1.8521	SJF	1.9302	1.9573	1.8732	1.8558	2.95%	5.18%	1.39%	-0.93%
6	1995	2.6770	FCFS	3.2082	3.2048	2.7059	2.6629	15.66%	16.91%	-0.11%	-1.62%
7	1995	1.9940	FCFS	1.9498	1.9161	1.7841	1.8333	8.50%	4.32%	-1.76%	2.68%
8	1995	1.4076	SJF	1.5479	1.5212	1.4423	1.4457	6.82%	4.97%	-1.75%	0.23%
9	1995	2.1396	FCFS	2.3717	2.3934	2.3441	2.3147	1.16%	3.29%	0.91%	-1.27%
10	1995	1.5212	SJF	1.6828	1.6246	1.5078	1.4986	10.40%	7.75%	-3.59%	-0.61%
11	1995	1.5907	FCFS	1.5261	1.6355	1.4475	1.4408	5.15%	11.91%	6.69%	-0.47%
12	1995	1.4963	SJF	1.5183	1.5044	1.4609	1.4643	3.78%	2.67%	-0.93%	0.23%
1	1996	1.2222	SJF	1.2469	1.2577	1.2246	1.2160	1.79%	3.32%	0.86%	-0.70%
2	1996	1.1414	FCFS	1.1491	1.1383	1.1319	1.1382	1.50%	0.01%	-0.95%	0.56%
3	1996	1.3567	SJF	1.4892	1.4810	1.3531	1.3663	9.14%	7.74%	-0.55%	0.97%
4	1996	1.5114	FCFS	1.5376	1.5471	1.4790	1.4962	3.81%	3.29%	0.61%	1.15%
5	1996	1.5167	SJF	1.5694	1.5128	1.4381	1.4321	8.37%	5.33%	-3.74%	-0.41%
6	1996	1.3084	FCFS	1.3065	1.3113	1.3000	1.3013	0.50%	0.76%	0.37%	0.10%
7	1996	2.1841	FCFS	2.1397	2.1342	2.1242	2.1184	0.72%	0.74%	-0.25%	-0.27%
8	1996	1.3515	FCFS	1.3386	1.3374	1.3322	1.3361	0.48%	0.09%	-0.10%	0.29%
9	1996	1.1977	FCFS	1.1656	1.1601	1.1597	1.1563	0.50%	0.33%	-0.47%	-0.30%
average		1.6457	FCFS	1.7538	1.7489	1.6101	1.6075	8.19%	8.09%	-0.28%	-0.16%
relative to best policy				-4.38%	-4.09%	4.17%	4.32%				

Table A.11: Monthly comparison of the simple and advanced decider for the LANL trace: ARTwW is used as thequality metrics in the self-tuning process. The overall given performance is SLDwA. Blue font indicates a better performance than the best basic policy FCFS.

month	year	best policy		simple half	simple full	advanced half	advanced full	advanced better than simple half	advanced better than simple full	difference of applying half self-tuning simple	difference of applying half self-tuning advanced
4	1998	1.0000	-	1.0000	1.0000	1.0000	1.0000	0.00%	0.00%	0.00%	0.00%
5	1998	2.5552	SJF	3.9804	4.1777	2.5709	2.5973	35.41%	37.83%	4.72%	1.02%
6	1998	1.9523	SJF	2.2676	2.3921	1.8456	1.8492	18.61%	22.69%	5.21%	0.19%
7	1998	2.1876	SJF	2.2523	2.3247	2.2678	2.2091	-0.69%	4.97%	3.11%	-2.66%
8	1998	2.3388	SJF	3.8311	2.5395	2.4038	2.2651	37.26%	10.81%	-50.86%	-6.12%
9	1998	2.6290	SJF	4.8500	5.2526	2.4917	2.4654	48.62%	53.06%	7.66%	-1.07%
10	1998	1.9787	SJF	2.5058	2.7115	1.9464	1.9727	22.32%	27.25%	7.59%	1.33%
11	1998	2.6827	SJF	3.2585	3.2806	2.4571	2.4617	24.59%	24.96%	0.67%	0.19%
12	1998	4.1766	SJF	6.5063	8.6952	4.0959	4.2356	37.05%	51.29%	25.17%	3.30%
1	1999	3.3003	SJF	5.9550	13.7324	3.2695	3.2121	45.10%	76.61%	56.64%	-1.79%
2	1999	6.8112	FCFS	11.9343	11.7164	7.4516	5.9569	37.56%	49.16%	-1.86%	-25.09%
3	1999	7.0065	SJF	7.9153	10.9714	6.4692	5.5629	18.27%	49.30%	27.86%	-16.29%
4	1999	5.9265	SJF	11.5757	24.2140	5.7675	6.0719	50.18%	74.92%	52.19%	5.01%
5	1999	7.3162	SJF	29.1250	72.0855	6.8181	7.0593	76.59%	90.21%	59.60%	3.42%
6	1999	11.9581	SJF	15.6778	38.3388	11.0933	12.2020	29.24%	68.17%	59.11%	9.09%
7	1999	7.3964	FCFS	16.9402	33.9734	9.5879	9.7302	43.40%	71.36%	50.14%	1.46%
8	1999	8.0159	FCFS	20.5530	27.3264	12.2373	13.6685	40.46%	49.98%	24.79%	10.47%
9	1999	7.6722	FCFS	23.1646	98.2952	16.5200	17.2090	28.68%	82.49%	76.43%	4.00%
10	1999	9.6773	FCFS	19.8268	38.3693	20.5178	19.9565	-3.49%	47.99%	48.33%	-2.81%
11	1999	10.4231	FCFS	17.6656	42.6313	19.3444	18.0743	-9.50%	57.60%	58.56%	-7.03%
12	1999	11.2749	FCFS	16.3706	79.9789	17.0810	16.2964	-4.34%	79.62%	79.53%	-4.81%
1	2000	6.3182	FCFS	25.2349	27.9630	15.5033	15.4515	38.56%	44.74%	9.76%	-0.34%
2	2000	8.1268	FCFS	18.1489	16.6975	19.8885	18.8583	-9.59%	-12.94%	-8.69%	-5.46%
3	2000	4.4993	FCFS	16.1985	15.8643	26.6457	23.1310	-64.49%	-45.81%	-2.11%	-15.19%
4	2000	5.9322	FCFS	18.8293	19.2047	13.5590	13.0022	27.99%	32.30%	1.95%	-4.28%
average		6.8261	FCFS	13.3353	26.3414	10.0953	9.8325	24.30%	62.97%	49.38%	-2.67%
relative to best policy				-95.36%	-285.89%	-47.89%	-44.04%				
only from 4-1998 until 6-1999											
average		4.2696	SJF	7.5090	13.5622	4.1299	4.0748	45.00%	69.95%	44.63%	-1.35%
relative to best policy				-75.87%	-217.58%	3.27%	4.56%				

Table A.12: Monthly comparison of the simple and advanced decider for the SDSC trace: ARTwW is used as the quality metrics in the self-tuning process. The given overall performance is SLDwA. Blue font indicates a better performance than the best basic policy FCFS.

A.2 Increased Workload

In the following tables and diagrams the performance of FCFS, SJF, LJF, and the self-tuning dynP scheduler at increased workloads are presented. Synthetic job sets with the statistical properties of the original traces are used. In order to exclude singular effects, ten job sets were generated and simulated. The presented numbers are averages of these job sets with the best and worst results in each case being neglected. The difference of SJF and LJF to FCFS for the slowdown is given as a percentage. The difference in the achieved utilization is given in percentage points. Note, smaller slowdowns and higher utilization are better.

A.2.1 Basic Policies

shrinking factor	SLDwA FCFS	SLDwA SJF	SLDwA LJF	rel. difference to FCFS in % SJF	rel. difference to FCFS in % LJF	utilization in % FCFS	utilization in % SJF	utilization in % LJF	abs. difference to to FCFS in %-points SJF	abs. difference to to FCFS in %-points LJF
CTC										
1.0	2.61	2.78	3.55	-6.72	-36.25	76.20	75.48	76.50	-0.71	0.30
0.9	3.99	4.80	5.99	-20.14	-50.08	83.43	80.74	84.29	-2.69	0.86
0.8	7.51	8.36	13.25	-11.34	-76.56	89.13	83.07	91.70	-6.06	2.57
0.7	13.01	12.27	23.42	5.69	-80.08	91.65	85.36	95.01	-6.28	3.36
0.6	19.61	17.46	36.22	10.98	-84.74	93.38	85.94	96.60	-7.44	3.22
average				-4.31	-65.54				-4.64	2.06
KTH										
1.0	4.06	3.32	7.33	18.18	-80.70	69.33	68.81	69.48	-0.52	0.15
0.9	5.51	4.35	11.11	21.05	-101.59	76.64	75.46	76.84	-1.18	0.20
0.8	9.00	6.85	20.75	23.87	-130.55	85.08	80.37	85.41	-4.71	0.33
0.7	20.72	12.29	54.58	40.65	-163.49	92.08	82.59	93.20	-9.49	1.13
0.6	45.73	21.29	120.84	53.46	-164.22	94.03	84.25	96.30	-9.78	2.28
average				31.44	-128.11				-5.13	0.82
LANL										
1.0	2.53	2.47	2.92	2.12	-15.69	63.61	63.61	63.63	0.00	0.02
0.9	3.20	3.16	3.83	1.07	-19.89	70.64	70.59	70.66	-0.05	0.02
0.8	4.69	5.11	6.26	-8.92	-33.25	79.37	79.11	79.42	-0.26	0.05
0.7	10.05	14.93	16.52	-48.57	-64.35	90.13	85.46	90.43	-4.67	0.30
0.6	44.46	41.73	82.88	6.16	-86.40	96.10	86.71	97.67	-9.40	1.57
average				-9.63	-43.92				-2.88	0.39
SDSC										
1.0	6.16	6.00	14.49	2.57	-135.36	79.41	78.59	79.69	-0.82	0.28
0.9	10.36	16.48	30.70	-58.98	-196.17	86.85	80.55	87.49	-6.30	0.64
0.8	25.06	29.86	84.77	-19.17	-238.35	91.83	81.23	92.87	-10.59	1.04
0.7	46.20	42.83	121.05	7.30	-161.99	93.15	81.87	95.00	-11.28	1.85
0.6	71.08	57.01	162.54	19.80	-128.65	94.05	82.38	96.19	-11.68	2.14
average				-9.70	-172.10				-8.13	1.19

Table A.13: Average slowdown weighted by area (SLDwA) of FCFS, SJF, and LJF with different workloads/shrinking factors. Part I.

shrinking factor	SLDwA FCFS	SLDwA SJF	SLDwA LJF	rel. difference to FCFS in % SJF	rel. difference to FCFS in % LJF	utilization in % FCFS	utilization in % SJF	utilization in % LJF	abs. difference to to FCFS in %-points SJF	abs. difference to to FCFS in %-points LJF
PC2-2001										
1.0	5.69	4.62	8.53	18.76	-49.84	47.18	47.03	47.27	-0.15	0.09
0.9	6.44	5.12	10.15	20.41	-57.72	52.20	52.00	52.37	-0.20	0.18
0.8	7.95	6.02	12.98	24.30	-63.20	58.10	57.72	58.64	-0.38	0.54
0.7	10.39	8.68	20.28	16.48	-95.20	65.50	62.13	66.54	-3.38	1.03
0.6	14.57	12.91	39.74	24.30	-63.20	74.27	67.00	76.39	-7.27	2.12
0.5	23.55	20.45	79.79	16.48	-95.20	81.52	69.36	86.38	-12.16	4.86
0.4	39.68	28.23	139.63	11.41	-172.82	87.68	71.40	93.83	-16.29	6.15
average				18.88	-85.31				-5.69	2.14
PC2-2002										
1.0	8.57	5.55	19.41	35.18	-126.59	48.84	48.94	49.44	0.09	0.59
0.9	9.60	6.11	24.94	36.38	-159.82	53.88	54.03	54.70	0.15	0.82
0.8	11.71	7.35	31.76	37.24	-171.15	60.39	60.15	61.04	-0.24	0.65
0.7	15.31	9.95	39.19	35.03	-155.90	67.59	63.74	68.88	-3.85	1.29
0.6	22.09	13.85	56.97	37.30	-157.88	75.00	68.40	78.90	-6.61	3.90
0.5	34.02	21.36	92.02	37.21	-170.51	80.20	70.68	88.62	-9.52	8.41
0.4	51.30	34.96	155.17	31.85	-202.48	84.98	72.55	94.52	-12.43	9.54
average				35.74	-163.47				-4.63	3.60
CHPC										
1.0	1.01	1.01	1.01	0.04	-0.23	42.11	42.11	42.11	0.00	0.00
0.9	1.02	1.02	1.03	0.25	-0.91	46.29	46.29	46.29	0.00	0.00
0.8	1.04	1.04	1.07	0.80	-2.17	51.40	51.40	51.40	0.00	0.00
0.7	1.11	1.08	1.18	2.85	-6.30	57.78	57.78	57.78	0.00	0.00
0.6	1.33	1.20	1.59	9.85	-19.91	65.67	65.66	65.68	0.00	0.01
0.5	2.26	1.62	3.82	28.58	-68.61	75.46	75.69	76.04	0.24	0.59
0.4	11.89	3.96	31.39	66.66	-164.05	83.11	78.97	90.43	-4.13	7.33
0.3	30.71	10.15	67.41	66.94	-119.55	83.38	80.36	98.42	-3.02	15.04
average				22.00	-47.72				-0.87	2.87
MHPCC										
1.0	1.05	1.05	1.06	0.33	-0.94	34.83	34.83	34.83	0.00	0.00
0.9	1.08	1.07	1.09	0.88	-0.86	38.63	38.63	38.63	0.00	0.00
0.8	1.11	1.09	1.13	1.20	-2.11	43.37	43.37	43.37	0.00	0.00
0.7	1.18	1.13	1.23	3.80	-4.32	49.42	49.42	49.42	0.00	0.00
0.6	1.31	1.25	1.40	4.08	-6.89	57.42	57.42	57.42	0.00	0.00
0.5	1.65	1.44	2.09	12.99	-26.72	68.52	68.52	68.52	0.00	0.00
0.4	3.37	2.76	6.57	18.24	-94.90	84.56	84.26	84.90	-0.30	0.34
0.3	32.69	9.51	72.31	70.90	-121.20	93.10	91.73	96.53	-1.36	3.43
average				14.05	-32.24				-0.21	0.47

Table A.14: Average slowdown weighted by area (SLDwA) of FCFS, SJF, and LJF with different workloads/shrinking factors. Part II.

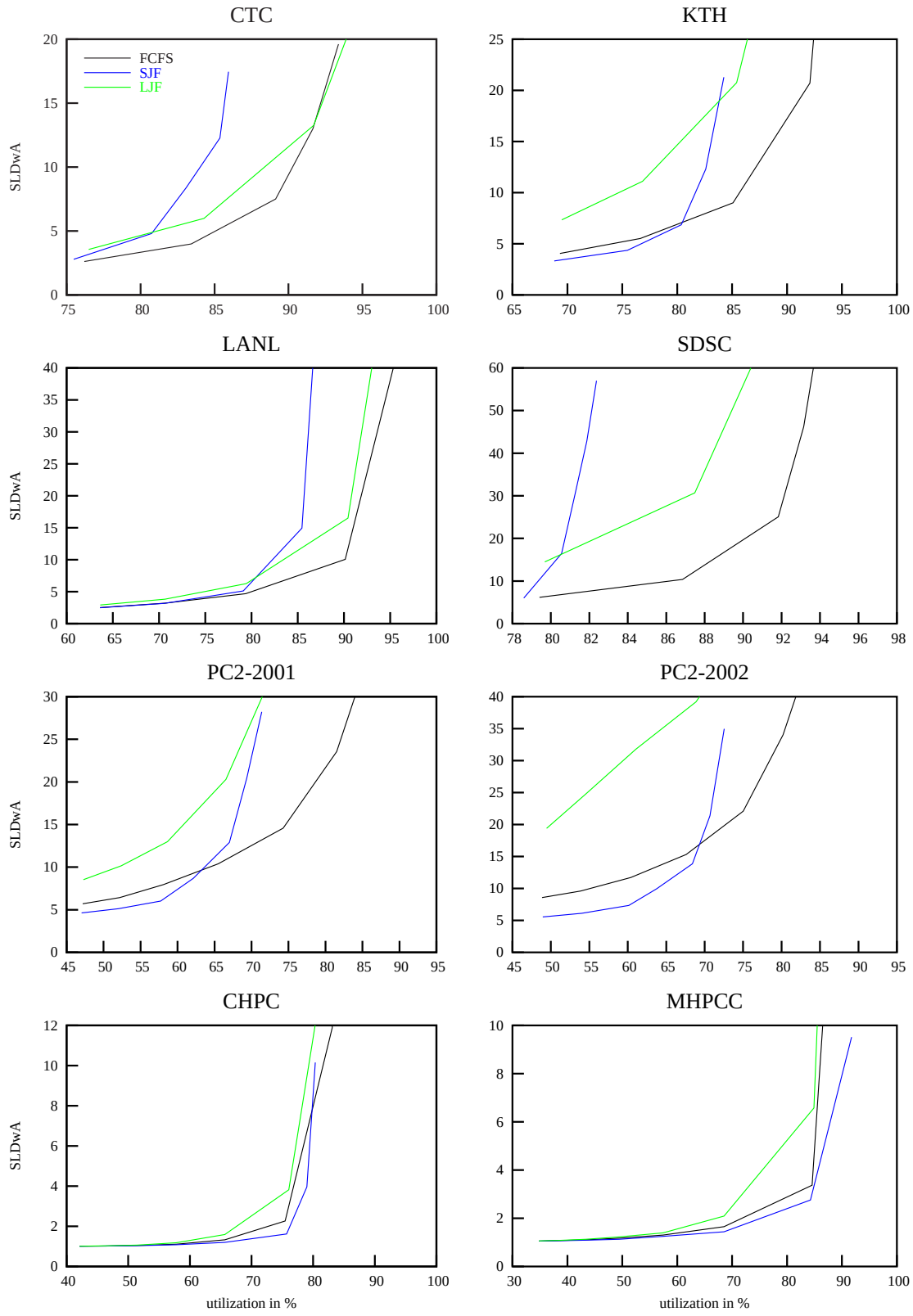


Figure A.1: Combination of Figure 5.1 and Figure 5.2 with utilization on the x-axis and slowdown (SLDwA) on the y-axis.

A.2.2 Self-Tuning dynP Scheduler

shrinking factor	SLDwA SJF	SLDwA adv.	SLDwA SJF-pref.	rel. difference to SJF in % adv.	rel. difference to SJF in % SJF-pref.	utilization in % SJF	utilization in % adv.	utilization in % SJF-pref.	abs. difference to to FCFS in %-points adv.	abs. difference to to FCFS in %-points SJF-pref.
CTC										
1.0	2.78	2.48	2.49	10.92	10.39	75.48	76.07	76.13	0.59	0.64
0.9	4.80	4.16	3.90	13.19	18.65	80.74	82.09	82.54	1.35	1.80
0.8	8.36	7.44	7.37	10.92	11.79	83.07	84.84	84.72	1.77	1.65
0.7	12.27	11.76	11.83	4.12	3.56	85.36	86.32	86.30	0.96	0.94
0.6	17.46	16.40	16.54	6.03	5.23	85.94	87.39	86.95	1.45	1.01
average				9.04	9.92				1.22	1.21
KTH										
1.0	3.32	3.25	3.20	2.15	3.56	68.81	69.04	68.98	0.23	0.17
0.9	4.35	4.31	4.42	1.04	-1.46	75.46	75.68	75.68	0.22	0.22
0.8	6.85	6.70	6.91	2.17	-0.87	80.37	80.72	80.63	0.35	0.26
0.7	12.29	12.79	12.80	-4.04	-4.07	82.59	82.37	82.42	-0.22	-0.17
0.6	21.29	21.41	21.45	-0.57	-0.75	84.25	84.33	84.40	0.08	0.15
average				0.15	-0.72				0.13	0.12
LANL										
1.0	2.47	2.43	2.42	1.81	1.96	63.61	63.61	63.61	0.00	0.00
0.9	3.16	3.13	3.13	1.11	0.95	70.59	70.63	70.63	0.04	0.04
0.8	5.11	4.95	5.00	3.28	2.19	79.11	79.14	79.12	0.03	0.01
0.7	14.93	14.50	14.58	2.92	2.34	85.46	85.64	85.57	0.18	0.11
0.6	41.73	42.37	42.13	-1.55	-0.97	86.71	86.81	87.00	0.10	0.29
average				1.51	1.29				0.07	0.09
SDSC										
1.0	6.00	5.56	5.59	7.29	6.78	78.59	78.75	78.73	0.16	0.14
0.9	16.48	13.90	14.09	15.66	14.48	80.55	81.99	82.20	1.44	1.64
0.8	29.86	27.64	27.54	7.43	7.76	81.23	82.59	82.42	1.36	1.19
0.7	42.83	41.95	41.74	2.05	2.56	81.87	83.01	82.96	1.14	1.08
0.6	57.01	57.35	57.29	-0.61	-0.50	82.38	82.94	82.86	0.56	0.49
average				6.36	6.22				0.93	0.91

Table A.15: Average slowdown weighted by area (SLDwA) of the self-tuning dynP scheduler with different workloads/shrinking factors. ARTwW is used as the self-tuning metrics and no slackness is used. The abbreviations 'adv.' stands for the advanced decider and 'SJF-pref.' for SJF-preferred decider respectively. Part I.

shrinking factor	SLDwA SJF	SLDwA adv.	SLDwA SJF-pref.	rel. difference to SJF in % adv.	rel. difference to SJF in % SJF-pref.	utilization in % SJF	utilization in % adv.	utilization in % SJF-pref.	abs. difference to to FCFS in %-points adv.	abs. difference to to FCFS in %-points SJF-pref.
PC2-2001										
1.0	4.62	4.56	4.47	1.43	3.37	47.03	47.05	47.05	0.03	0.03
0.9	5.12	5.09	5.06	0.70	1.17	52.00	51.87	51.91	-0.13	-0.09
0.8	6.02	6.36	6.34	-5.56	-5.28	57.72	57.73	57.77	0.01	0.05
0.7	8.68	8.29	8.16	4.47	5.93	62.13	63.20	63.19	1.07	1.07
0.6	12.91	13.09	13.20	-1.45	-2.26	67.00	66.09	66.29	-0.92	-0.71
0.5	20.45	19.80	19.83	3.16	3.05	69.36	69.81	69.93	0.45	0.57
0.4	28.23	28.47	28.55	-0.85	-1.14	71.40	71.51	71.29	0.11	-0.10
average				0.27	0.69				0.09	0.11
PC2-2002										
1.0	5.55	5.53	5.47	0.45	1.54	48.94	48.97	49.07	0.04	0.13
0.9	6.11	6.29	6.24	-3.00	-2.27	54.03	54.12	53.88	0.09	-0.15
0.8	7.35	7.62	7.54	-3.64	-2.53	60.15	60.07	59.98	-0.08	-0.17
0.7	9.95	9.91	9.81	0.43	1.45	63.74	64.94	64.45	1.19	0.71
0.6	13.85	13.81	13.72	0.32	0.96	68.40	67.77	68.60	-0.63	0.20
0.5	21.36	21.22	21.51	0.66	-0.68	70.68	71.76	72.02	1.08	1.34
0.4	34.96	34.76	35.22	0.59	-0.73	72.55	73.58	73.35	1.03	0.80
average				-0.60	-0.32				0.39	0.41
CHPC										
1.0	1.01	1.01	1.01	-0.09	-0.01	42.11	42.11	42.11	0.00	0.00
0.9	1.02	1.02	1.02	0.02	-0.03	46.29	46.29	46.29	0.00	0.00
0.8	1.04	1.04	1.04	-0.31	-0.25	51.40	51.40	51.40	0.00	0.00
0.7	1.08	1.09	1.09	-0.86	-0.76	57.78	57.78	57.78	0.00	0.00
0.6	1.20	1.21	1.21	-1.32	-1.23	65.66	65.67	65.67	0.01	0.01
0.5	1.62	1.66	1.63	-2.83	-1.07	75.69	75.79	75.83	0.10	0.14
0.4	3.96	4.09	3.98	-3.21	-0.37	78.97	79.69	79.26	0.72	0.29
0.3	10.15	10.46	10.27	-3.07	-1.15	80.36	80.80	80.48	0.45	0.13
average				-1.46	-0.61				0.16	0.07
MHPCC										
1.0	1.05	1.05	1.05	-0.08	0.28	34.83	34.83	34.83	0.00	0.00
0.9	1.07	1.07	1.07	-0.06	0.02	38.63	38.63	38.63	0.00	0.00
0.8	1.09	1.10	1.10	-0.72	-0.36	43.37	43.37	43.37	0.00	0.00
0.7	1.13	1.14	1.14	-0.82	-0.62	49.42	49.42	49.42	0.00	0.00
0.6	1.25	1.24	1.24	0.73	0.71	57.42	57.42	57.42	0.00	0.00
0.5	1.44	1.48	1.45	-3.06	-0.53	68.52	68.52	68.52	0.00	0.00
0.4	2.76	2.17	2.50	21.41	9.30	84.26	84.52	84.49	0.26	0.23
0.3	9.51	9.48	9.40	0.34	1.15	91.73	91.78	91.77	0.05	0.04
average				2.22	1.24				0.04	0.03

Table A.16: Average slowdown weighted by area (SLDwA) of the self-tuning dynP scheduler with different workloads/shrinking factors. ARTwW is used as the self-tuning metrics and no slackness is used. The abbreviations 'adv.' stands for the advanced decider and 'SJF-pref.' for SJF-preferred decider respectively. Part II.

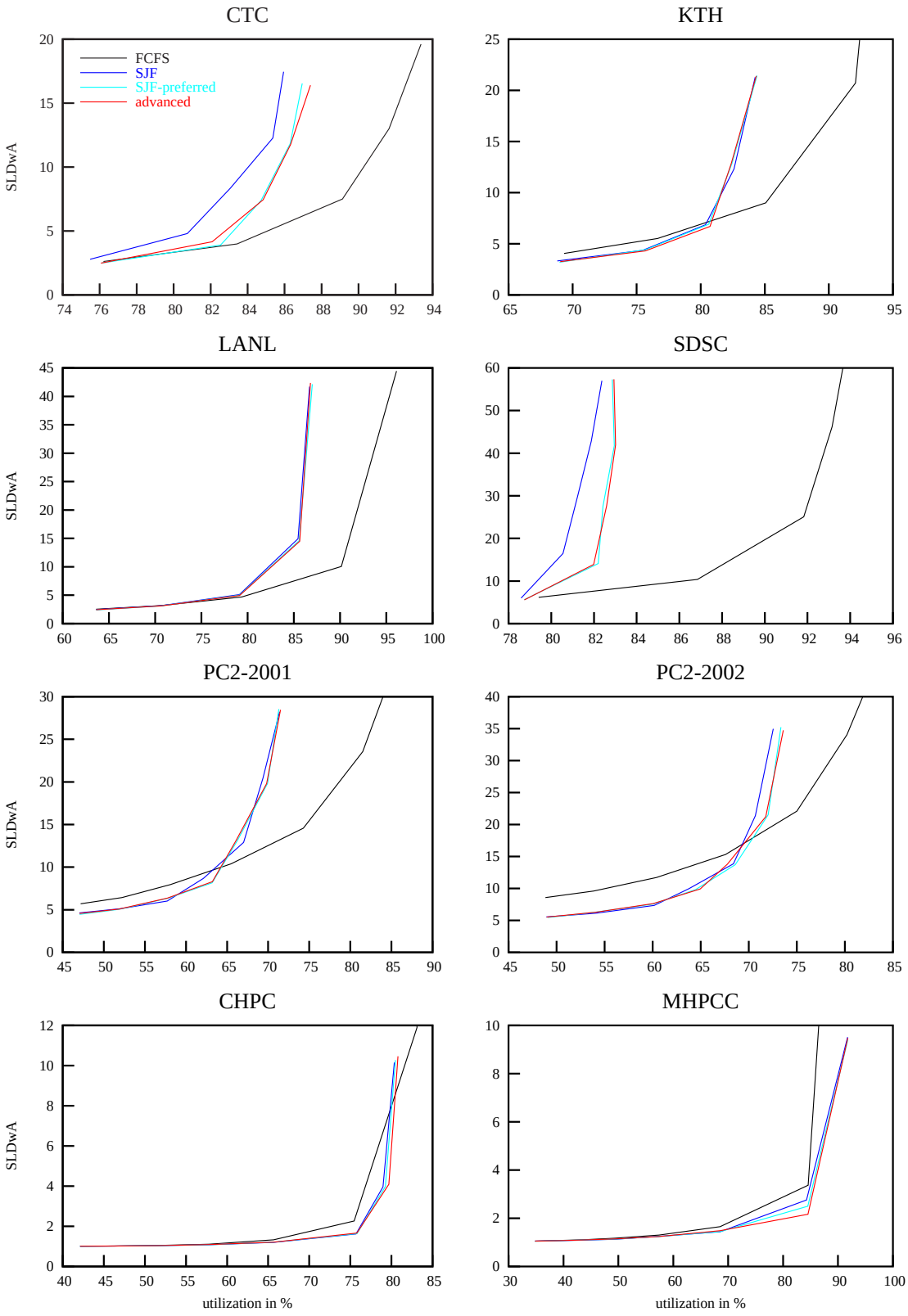


Figure A.2: Combination of Figure 5.3 and Figure 5.4 with the utilization on the x-axis and the slowdown (SLDwA) on the y-axis.

shrink. factor	SJF	SLDwA 0%	2%	4%	6%	8%	10%	relative to SJF 0%	2%	4%	6%	8%	10%
CTC		advanced decider											
1.0	2.78	2.48	2.50	2.69	2.71	2.75	2.75	10.92%	10.31%	3.36%	2.57%	1.13%	1.07%
0.9	4.80	4.16	4.13	4.48	4.86	4.94	4.92	13.19%	13.78%	6.52%	-1.46%	-3.11%	-2.56%
0.8	8.36	7.44	7.63	7.70	8.20	8.29	8.34	10.92%	8.76%	7.92%	1.88%	0.76%	0.22%
0.7	12.27	11.76	12.17	12.82	12.64	12.61	12.41	4.12%	0.80%	-4.52%	-3.06%	-2.83%	-1.17%
0.6	17.46	16.40	17.01	17.35	17.56	17.38	17.36	6.03%	2.58%	0.62%	-0.63%	0.45%	0.57%
average								9.04%	7.24%	2.78%	-0.14%	-0.72%	-0.38%
		SJF-preferred decider											
1.0	2.78	2.49	2.60	2.61	2.65	2.67	2.70	10.39%	6.53%	6.25%	4.94%	3.99%	2.88%
0.9	4.80	3.90	4.66	4.96	4.84	4.94	4.84	18.65%	2.83%	-3.40%	-1.01%	-3.00%	-0.85%
0.8	8.36	7.37	8.13	8.30	8.39	8.27	8.28	11.79%	2.68%	0.70%	-0.41%	0.99%	0.87%
0.7	12.27	11.83	12.50	12.47	12.42	12.35	12.37	3.56%	-1.93%	-1.63%	-1.23%	-0.65%	-0.83%
0.6	17.46	16.54	17.43	17.38	17.24	17.38	17.38	5.23%	0.17%	0.46%	1.26%	0.46%	0.46%
average								9.92%	2.06%	0.48%	0.71%	0.36%	0.51%
KTH		advanced decider											
1.0	3.32	3.25	3.24	3.23	3.24	3.29	3.33	2.15%	2.25%	2.77%	2.27%	0.80%	-0.39%
0.9	4.35	4.31	4.26	4.29	4.32	4.35	4.36	1.04%	2.05%	1.38%	0.62%	0.15%	-0.21%
0.8	6.85	6.70	6.78	7.01	7.05	7.15	7.08	2.17%	1.07%	-2.36%	-2.90%	-4.30%	-3.30%
0.7	12.29	12.79	12.21	12.29	12.73	12.34	12.40	-4.04%	0.67%	0.06%	-3.56%	-0.34%	-0.83%
0.6	21.29	21.41	21.29	21.34	21.16	21.04	21.10	-0.57%	-0.02%	-0.23%	0.57%	1.17%	0.90%
average								0.15%	1.21%	0.32%	-0.60%	-0.50%	-0.77%
		SJF-preferred decider											
1.0	3.32	3.20	3.26	3.27	3.27	3.28	3.32	3.56%	1.89%	1.50%	1.60%	1.19%	-0.07%
0.9	4.35	4.42	4.23	4.29	4.29	4.29	4.31	-1.46%	2.85%	1.37%	1.45%	1.46%	1.04%
0.8	6.85	6.91	6.86	6.90	6.98	7.03	6.88	-0.87%	-0.14%	-0.79%	-1.89%	-2.61%	-0.39%
0.7	12.29	12.80	12.35	12.20	12.27	12.19	12.19	-4.07%	-0.45%	0.81%	0.21%	0.85%	0.85%
0.6	21.29	21.45	21.20	21.11	21.21	21.22	21.20	-0.75%	0.41%	0.85%	0.35%	0.31%	0.39%
average								-0.72%	0.91%	0.75%	0.34%	0.24%	0.36%

Table A.17: Slowdown (SLDwA) values for different slackness values (in %) with different workloads/shrinking factors. The advanced and SJF-preferred deciders is applied with ARTwW as the self-tuning metrics. Red color highlights the best slackness value. Part I, CTC and KTH.

shrink. factor	SJF	SLDwA 0%	2%	4%	6%	8%	10%	relative to SJF 0%	2%	4%	6%	8%	10%
LANL		advanced decider											
1.0	2.47	2.43	2.42	2.43	2.43	2.45	2.47	1.81%	2.30%	1.76%	1.82%	0.80%	0.28%
0.9	3.16	3.13	3.10	3.12	3.17	3.16	3.16	1.11%	1.98%	1.49%	-0.10%	0.24%	0.20%
0.8	5.11	4.95	5.05	5.04	5.10	5.12	5.10	3.28%	1.28%	1.41%	0.17%	-0.19%	0.34%
0.7	14.93	14.50	14.72	14.84	14.97	15.04	15.04	2.92%	1.45%	0.65%	-0.27%	-0.71%	-0.72%
0.6	41.73	42.37	41.73	42.22	41.73	41.73	41.73	-1.55%	-0.02%	-1.18%	0.00%	0.00%	0.00%
average								1.51%	1.40%	0.83%	0.33%	0.03%	0.02%
		SJF-preferred decider											
1.0	2.47	2.42	2.43	2.43	2.44	2.44	2.47	1.96%	1.78%	1.73%	1.49%	1.16%	0.14%
0.9	3.16	3.13	3.11	3.18	3.16	3.17	3.16	0.95%	1.61%	-0.37%	0.00%	-0.31%	0.22%
0.8	5.11	5.00	4.94	5.01	5.14	5.16	5.17	2.19%	3.43%	2.09%	-0.49%	-0.98%	-1.12%
0.7	14.93	14.58	14.75	15.01	14.97	15.01	14.94	2.34%	1.20%	-0.49%	-0.25%	-0.54%	-0.07%
0.6	41.73	42.13	41.98	42.25	42.02	41.73	41.73	-0.97%	-0.62%	-1.27%	-0.70%	0.00%	0.00%
average								1.29%	1.48%	0.34%	0.01%	-0.13%	-0.17%
SDSC		advanced decider											
1.0	6.00	5.56	5.84	6.08	6.31	6.14	5.98	7.29%	2.69%	-1.32%	-5.09%	-2.39%	0.26%
0.9	16.48	13.90	15.65	16.30	16.31	16.30	16.63	15.66%	5.02%	1.09%	1.04%	1.07%	-0.93%
0.8	29.86	27.64	28.67	29.41	29.47	29.39	29.85	7.43%	3.97%	1.50%	1.28%	1.57%	0.02%
0.7	42.83	41.95	42.49	42.65	42.40	42.57	42.56	2.05%	0.79%	0.42%	1.00%	0.62%	0.63%
0.6	57.01	57.35	56.46	56.63	56.56	56.64	56.64	-0.61%	0.97%	0.67%	0.78%	0.65%	0.65%
average								6.36%	2.69%	0.47%	-0.20%	0.30%	0.13%
		SJF-preferred decider											
1.0	6.00	5.59	6.03	5.90	6.02	6.00	6.03	6.78%	-0.51%	1.63%	-0.40%	0.01%	-0.53%
0.9	16.48	14.09	16.23	16.02	16.48	16.47	16.47	14.48%	1.52%	2.75%	-0.01%	0.02%	0.02%
0.8	29.86	27.54	29.46	29.64	29.60	29.86	29.86	7.76%	1.34%	0.73%	0.86%	-0.01%	0.00%
0.7	42.83	41.74	42.42	42.98	42.63	42.63	42.53	2.56%	0.97%	-0.34%	0.47%	0.47%	0.71%
0.6	57.01	57.29	56.60	56.60	56.87	56.94	57.04	-0.50%	0.72%	0.72%	0.24%	0.11%	-0.05%
average								6.22%	0.81%	1.10%	0.23%	0.12%	0.03%

Table A.18: Slowdown (SLDwA) values for different slackness values (in %) with different workloads/shrinking factors. The advanced and SJF-preferred decider is applied with ARTwW as the self-tuning metrics. Red color highlights the best slackness value. Part II, LANL and SDSC.

shrink. factor	SJF	SLDwA 0%	2%	4%	6%	8%	10%	relative to SJF 0%	2%	4%	6%	8%	10%
PC2-2001		advanced decider											
1.0	4.62	4.56	4.52	4.67	4.66	4.67	4.65	1.43%	2.21%	-0.91%	-0.79%	-1.06%	-0.57%
0.9	5.12	5.09	5.11	5.25	5.11	5.15	5.12	0.70%	0.27%	-2.37%	0.30%	-0.57%	0.12%
0.8	6.02	6.36	6.29	6.25	6.11	6.04	6.00	-5.56%	-4.45%	-3.80%	-1.45%	-0.30%	0.29%
0.7	8.68	8.29	8.85	8.91	8.83	8.66	8.66	4.47%	-2.01%	-2.64%	-1.77%	0.20%	0.25%
0.6	12.91	13.09	12.69	12.61	12.71	13.05	12.74	-1.45%	1.67%	2.31%	1.52%	-1.09%	1.28%
0.5	20.45	19.80	20.07	20.07	19.83	19.91	19.97	3.16%	1.85%	1.86%	3.03%	2.67%	2.35%
0.4	28.23	28.47	28.38	28.34	28.33	28.35	28.35	-0.85%	-0.54%	-0.38%	-0.38%	-0.43%	-0.43%
average								0.27%	-0.14%	-0.85%	0.07%	-0.08%	0.47%
		SJF-preferred decider											
1.0	4.62	4.47	4.45	4.62	4.62	4.64	4.64	3.37%	3.83%	0.04%	0.15%	-0.45%	-0.43%
0.9	5.12	5.06	5.02	5.17	5.04	5.09	5.09	1.17%	2.10%	-0.89%	1.62%	0.59%	0.73%
0.8	6.02	6.34	6.33	6.27	6.06	6.02	6.03	-5.28%	-5.07%	-4.19%	-0.61%	0.00%	-0.16%
0.7	8.68	8.16	8.79	8.87	8.92	8.64	8.74	5.93%	-1.26%	-2.20%	-2.78%	0.41%	-0.75%
0.6	12.91	13.20	12.83	12.86	12.87	13.01	12.94	-2.26%	0.61%	0.33%	0.29%	-0.78%	-0.25%
0.5	20.45	19.83	20.56	20.56	20.30	20.30	20.30	3.05%	-0.52%	-0.53%	0.74%	0.74%	0.74%
0.4	28.23	28.55	28.45	28.49	28.39	28.25	28.25	-1.14%	-0.77%	-0.92%	-0.58%	-0.08%	-0.08%
average								0.69%	-0.15%	-1.19%	-0.17%	0.06%	-0.03%
PC2-2002		advanced decider											
1.0	5.55	5.53	5.48	5.55	5.47	5.46	5.44	0.45%	1.31%	-0.03%	1.43%	1.60%	2.06%
0.9	6.11	6.29	6.26	6.06	6.14	6.10	6.13	-3.00%	-2.51%	0.72%	-0.56%	0.17%	-0.32%
0.8	7.35	7.62	7.35	7.34	7.42	7.31	7.37	-3.64%	0.04%	0.12%	-0.98%	0.59%	-0.29%
0.7	9.95	9.91	9.71	9.67	9.60	9.93	9.96	0.43%	2.39%	2.84%	3.47%	0.24%	-0.13%
0.6	13.85	13.81	13.65	13.61	13.81	13.97	13.97	0.32%	1.47%	1.78%	0.29%	-0.82%	-0.84%
0.5	21.36	21.22	21.11	21.22	21.14	20.98	21.04	0.66%	1.15%	0.67%	1.03%	1.76%	1.48%
0.4	34.96	34.76	34.26	34.76	34.76	34.76	34.76	0.59%	1.99%	0.59%	0.59%	0.59%	0.59%
average								-0.60%	0.84%	0.96%	0.75%	0.59%	0.36%
		SJF-preferred decider											
1.0	5.55	5.47	5.55	5.49	5.46	5.46	5.45	1.54%	0.13%	1.14%	1.70%	1.72%	1.76%
0.9	6.11	6.24	6.18	6.14	6.16	6.16	6.12	-2.27%	-1.13%	-0.54%	-0.84%	-0.85%	-0.20%
0.8	7.35	7.54	7.29	7.32	7.39	7.32	7.35	-2.53%	0.88%	0.45%	-0.51%	0.47%	-0.02%
0.7	9.95	9.81	9.81	9.64	9.60	9.95	9.96	1.45%	1.37%	3.07%	3.47%	0.00%	-0.09%
0.6	13.85	13.72	13.89	14.15	14.16	14.17	13.99	0.96%	-0.25%	-2.16%	-2.23%	-2.28%	-1.02%
0.5	21.36	21.51	21.96	21.75	21.40	21.08	21.08	-0.68%	-2.80%	-1.81%	-0.20%	1.31%	1.31%
0.4	34.96	35.22	34.50	35.14	34.99	34.62	34.62	-0.73%	1.33%	-0.52%	-0.07%	0.97%	0.97%
average								-0.32%	-0.07%	-0.05%	0.19%	0.19%	0.39%

Table A.19: Slowdown (SLDwA) values for different slackness values (in %) with different workloads/shrinking factors. The advanced and SJF-preferred decider is applied with ARTwW as the self-tuning metrics. Red color highlights the best slackness value. Part III, PC2-2001 and PC2-2002.

shrink. factor	SJF	SLDwA 0%	2%	4%	6%	8%	10%	relative to SJF 0%	2%	4%	6%	8%	10%
CHPC		advanced decider											
1.0	1.01	1.01	1.01	1.01	1.01	1.01	1.01	-0.09%	-0.01%	0.00%	-0.03%	-0.04%	-0.03%
0.9	1.02	1.02	1.02	1.02	1.02	1.02	1.02	0.02%	0.00%	-0.02%	-0.01%	0.00%	0.01%
0.8	1.04	1.04	1.04	1.04	1.04	1.04	1.04	-0.31%	-0.14%	-0.05%	-0.04%	-0.09%	-0.16%
0.7	1.08	1.09	1.08	1.08	1.08	1.08	1.08	-0.86%	-0.57%	-0.19%	-0.16%	-0.29%	-0.14%
0.6	1.20	1.21	1.19	1.20	1.20	1.20	1.20	-1.32%	0.18%	-0.14%	0.02%	-0.17%	-0.16%
0.5	1.62	1.66	1.62	1.62	1.62	1.61	1.61	-2.83%	-0.37%	-0.53%	-0.20%	0.11%	0.18%
0.4	3.96	4.09	3.92	3.96	3.95	3.95	3.98	-3.21%	1.15%	0.08%	0.34%	0.24%	-0.30%
0.3	10.15	10.46	10.10	9.96	10.13	10.13	10.17	-3.07%	0.46%	1.93%	0.17%	0.17%	-0.16%
average								-1.46%	0.09%	0.14%	0.01%	-0.01%	-0.10%
		SJF-preferred decider											
1.0	1.01	1.01	1.01	1.01	1.01	1.01	1.01	-0.01%	-0.01%	0.00%	0.00%	0.00%	0.00%
0.9	1.02	1.02	1.02	1.02	1.02	1.02	1.02	-0.03%	-0.02%	-0.01%	-0.01%	0.00%	0.00%
0.8	1.04	1.04	1.04	1.04	1.04	1.04	1.04	-0.25%	-0.03%	-0.01%	0.01%	0.00%	0.00%
0.7	1.08	1.09	1.08	1.08	1.08	1.08	1.08	-0.76%	-0.07%	0.02%	0.01%	0.01%	0.01%
0.6	1.20	1.21	1.20	1.20	1.20	1.20	1.20	-1.23%	-0.13%	-0.02%	0.00%	0.00%	0.00%
0.5	1.62	1.63	1.62	1.62	1.62	1.62	1.62	-1.07%	-0.02%	-0.19%	-0.23%	-0.41%	-0.43%
0.4	3.96	3.98	3.98	3.96	3.96	3.96	3.96	-0.37%	-0.32%	0.00%	0.00%	0.00%	0.00%
0.3	10.15	10.27	10.14	10.15	10.15	10.15	10.15	-1.15%	0.07%	0.00%	0.00%	0.00%	0.00%
average								-0.61%	-0.07%	-0.03%	-0.03%	-0.05%	-0.05%
MHPCC		advanced decider											
1.0	1.05	1.05	1.05	1.05	1.05	1.05	1.05	-0.08%	0.01%	0.03%	-0.10%	-0.11%	-0.13%
0.9	1.07	1.07	1.07	1.06	1.06	1.07	1.07	-0.06%	-0.56%	0.34%	0.26%	-0.04%	-0.39%
0.8	1.09	1.10	1.10	1.09	1.10	1.10	1.10	-0.72%	-0.42%	0.09%	-0.24%	-0.59%	-0.55%
0.7	1.13	1.14	1.13	1.13	1.14	1.14	1.14	-0.82%	0.00%	-0.03%	-0.32%	-0.44%	-0.55%
0.6	1.25	1.24	1.24	1.24	1.24	1.24	1.24	0.73%	0.96%	1.27%	1.40%	1.23%	1.31%
0.5	1.44	1.48	1.45	1.44	1.43	1.43	1.43	-3.06%	-1.22%	-0.12%	0.47%	0.44%	0.46%
0.4	2.76	2.17	2.39	2.76	2.75	2.75	2.76	21.41%	13.33%	0.06%	0.27%	0.18%	-0.19%
0.3	9.51	9.48	9.47	9.34	9.34	9.34	9.34	0.34%	0.42%	1.82%	1.81%	1.82%	1.82%
average								2.22%	1.56%	0.43%	0.45%	0.31%	0.22%
		SJF-preferred decider											
1.0	1.05	1.05	1.05	1.05	1.05	1.05	1.05	0.28%	0.06%	0.00%	0.00%	0.00%	0.00%
0.9	1.07	1.07	1.07	1.07	1.07	1.07	1.07	0.02%	0.00%	-0.06%	-0.06%	-0.01%	0.00%
0.8	1.09	1.10	1.10	1.09	1.09	1.09	1.09	-0.36%	-0.21%	0.00%	-0.02%	0.00%	0.02%
0.7	1.13	1.14	1.13	1.13	1.13	1.13	1.13	-0.62%	-0.11%	-0.05%	0.04%	0.08%	0.04%
0.6	1.25	1.24	1.26	1.25	1.26	1.25	1.25	0.71%	-0.51%	-0.13%	-0.20%	-0.12%	0.06%
0.5	1.44	1.45	1.42	1.43	1.43	1.43	1.43	-0.53%	1.16%	0.64%	0.54%	0.36%	0.41%
0.4	2.76	2.50	2.95	2.62	2.74	2.74	2.76	9.30%	-7.02%	4.89%	0.56%	0.53%	0.00%
0.3	9.51	9.40	9.40	9.51	9.51	9.51	9.51	1.15%	1.22%	0.00%	0.00%	0.00%	0.00%
average								1.24%	-0.68%	0.66%	0.11%	0.10%	0.07%

Table A.20: Slowdown (SLDwA) values for different slackness values (in %) with different workloads/shrinking factors. The advanced and SJF-preferred decider is applied with ARTwW as the self-tuning metrics. Red color highlights the best slackness value. Part IV, CHPC and MHPCC.

Bibliography

[1] P. Brucker. *Scheduling Algorithms*. Springer, 3^{rd} edition, 2001.

[2] M. Brune, J. Gehring, A. Keller, and A. Reinefeld. Managing Clusters of Geographically Distributed High-Performance Computers. *Concurrency - Practice and Experience*, 11(15):887–911, 1999.

[3] M. Brune, A. Keller, and A. Reinefeld. Resource Management for High-Performance PC Clusters. In *Proc. of 7th International Conference High-Performance Computing and Networking Europe*, volume 1593 of *Lecture Notes in Computer Science*, pages 270–281. Springer, 1999.

[4] M. Calzarossa and G. Serazzi. A Characterization of the Variation in Time of Workload Arrival Patterns. *IEEE Trans. Comput. C-34(2)*, pages 156–162, February 1985.

[5] The CCS Software. `http://www.upb.de/pc2/projects/ccs/`, October 2003.

[6] S.J. Chapin, W. Cirne, D. G. Feitelson, J. P. Jones, S. T. Leutenegger, W. Schwiegelshohn, W. Smith, and D. Talby. Benchmarks and Standards for the Evaluation of Parallel Job Schedulers. In D. G. Feitelson and L. Rudolph, editor, *Proc. of 5th Workshop on Job Scheduling Strategies for Parallel Processing*, volume 1659 of *Lecture Notes in Computer Science*, pages 1–16. Springer, 1999.

[7] W. Cirne and F. Berman. A Model for Moldable Supercomputer Jobs. In *Proc. of the 15th International Conference on Parallel and Distributed Processing Symposium*, 2001.

[8] K. Czajkowski, I. Foster, N. Karonis, C. Kesselman, S. Martin, W. Smith, and S. Tuecke. A Resource Management Architecture for Metacomputing Systems. In D. G. Feitelson and L. Rudolph, editor, *Proc. of 4th Workshop on Job Scheduling Strategies for Parallel Processing*, volume 1459 of *Lecture Notes in Computer Science*, pages 62–82. Springer, 1998.

[9] The DataGrid Project. `http://www.eu-datagrid.org/`, October 2003.

[10] A. B. Downey. A Parallel Workload Model and Its Implications for Processor Allocation. In *Proc. of the 6th International Symposium on High Performance Distributed Computing (HPDC-6)*, August 1997.

[11] A. B. Downey and D. G. Feitelson. The Elusive Goal of Workload Characterization. Technical report, Hebrew University, Jerusalem, March 1999.

[12] EGrid Testbed Working Group: G. Allen, G. Aloisio, Z. Balaton, M. Cafaro, T. Dramlitsch, H.-H. Frese, J. Gehring, T. Goodale, P. Kacsuk, A. Keller, H.-P. Kersken, T. Kielmann, H. Knipp, G. Lanfermann, B. Ludwiczak, L. Matyska, A. Merzky, J. Nabrzyski J. Pukacki, T. Radke, A. Reinefeld, M. Ruda, F. Schintke, E. Seidel, A. Streit, F. Szalai,

K. Verstoep, and W. Ziegler. Early Experiences with the EGrid Testbed. In *Proccedings of 1st IEEE/ACM International Symposium on Cluster Computing and the Grid (CC-GRID 2001)*, pages 130–137. IEEE Computer Society Press, 2001.

[13] C. Ernemann, V. Hamscher, U. Schwiegelshohn, A. Streit, and R.Yahyapour. On Advantages of Grid Computing for Parallel Job Scheduling. In *Proc. of the 2nd IEEE International Symposium on Cluster Computing and the Grid (CC-GRID 2002)*, pages 39–46. IEEE Computer Society Press, 2002.

[14] C. Ernemann, V. Hamscher, A. Streit, and R. Yahyapour. Enhanced Algorithms for Multi-Site Scheduling. In *Proc. of 3rd IEEE/ACM International Workshop on Grid Computing (Grid 2002) at Supercomputing 2002*, volume 2536 of *Lecture Notes in Computer Science*, pages 219–231. Springer, 2002.

[15] C. Ernemann, V. Hamscher, A. Streit, and R. Yahyapour. On Effects of Machine Configurations on Parallel Job Scheduling in Computational Grids. In *Proc. of the International Conference on Architecture of Computing Systems (ARCS 2002)*, pages 169–179. VDE-Verlag, 2002.

[16] 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.

[17] D. G. Feitelson. Packing Schemes for Gang Scheduling. 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 89–101. Springer, 1996.

[18] D. G. Feitelson. Metrics for Parallel Job Scheduling and their Convergence. In D. G. Feitelson and L. Rudolph, editor, *Proc. of 7th Workshop on Job Scheduling Strategies for Parallel Processing*, volume 2221 of *Lecture Notes in Computer Science*, pages 190–208. Springer, 2001.

[19] D. G. Feitelson. Analyzing the Root Causes of Performance Evaluation Results. TR 2002-4, Hebrew University, Jerusalem, March 2002.

[20] D. G. Feitelson. The Forgotten Factor: Facts on Performance Evaluation and its Dependence on Workloads. In *Proceedings of EUROPAR 2002*, volume 2400 of *Lecture Notes of Computer Science*, pages 49–60. Springer, 2002.

[21] 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 of *Lecture Notes in Computer Science*, pages 238–262. Springer, 1997.

[22] D. G. Feitelson and A. W. Mu'alem. On the Definition of "On-Line" in Job Scheduling Problems. TR 2000-32, Hebrew University, Jerusalem, March 2000.

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

[24] 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.

[25] D. G. Feitelson and L. Rudolph. Parallel Job Scheduling: Issues and Approaches. In *Proc. of 1st Workshop on Job Scheduling Strategies for Parallel Processing*, volume 949 of *Lecture Notes in Computer Science*, pages 1–18. Springer, 1995.

[26] D. G. Feitelson and L. Rudolph. Toward Convergence in Job Schedulers for Parallel Supercomputers. In D. G. Feitelson and L. Rudolph, editor, *Proc. of 2nd Workshop on Job Scheduling Strategies for Parallel Processing*, volume 1162, pages 1–26. Springer, 1996.

[27] D. G. Feitelson and L. Rudolph. Metrics and Benchmarking for Parallel Job Scheduling. In D. G. Feitelson and L. Rudolph, editor, *Proc. of 4th Workshop on Job Scheduling Strategies for Parallel Processing*, volume 1459 of *Lecture Notes in Computer Science*, pages 1–24. Springer, 1998.

[28] D. G. Feitelson, L. Rudolph, U. Schwiegelshohn, and K. C. Sevcik. Theory and Practice in Parallel Job Scheduling. In D. G. Feitelson and L. Rudolph, editor, *Proc. of 3rd Workshop on Job Scheduling Strategies for Parallel Processing*, volume 1291 of *Lecture Notes in Computer Science*, pages 1–34. Springer, 1997.

[29] D. G. Feitelson and A. Weil. Utilization and Predictability in Scheduling the IBM SP2 with Backfilling. In *Proc. of the 1st Merged International Parallel Processing Symposium and Symposium on Parallel and Distributed Processing*, pages 542–547. IEEE Computer Society Press, 1998.

[30] I. Foster and C. Kesselman (Eds.). *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers Inc. San Fransisco, 1999.

[31] J. Gehring and T. Preiss. Scheduling a Metacomputer With Uncooperative Subschedulers. In D. G. Feitelson and L. Rudolph, editor, *Proc. of 5th Workshop on Job Scheduling Strategies for Parallel Processing*, volume 1659 of *Lecture Notes in Computer Science*, pages 179–201. Springer, 1999.

[32] 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.

[33] J. Gehring and A. Streit. Robust Resource Management for Metacomputers. In *Proceedings of the 9th IEEE International Symposium on High PerformanceDistributed Computing (HPDC-2000)*, pages 105–111. IEEE Computer Society Press, 2000.

[34] J. Gehring and A. Streit. The MOL-Kernel - A Platform for Multiform Metacomputing Services. In *1st EGRID Workshop at ISThmus 2000*, pages 269–277, April 2000.

[35] Global Grid Forum. `www.globalgridforum.org`, October 2003.

[36] R. Gibbons. A Historical Application Profiler for Use by Parallel Schedulers. In D. G. Feitelson and L. Rudolph, editor, *Proc. of 3rd Workshop on Job Scheduling Strategies for Parallel Processing*, volume 1291 of *Lecture Notes in Computer Science*, pages 58–77. Springer, 1997.

[37] The Globus Project. `http://www.globus.org/`, October 2003.

[38] R. L. Graham, E. L. Lawler, J. K. Lenstra, and A. H. G. Rinnoy Kan. Optimization and Approximation in Deterministic Sequencing and Scheduling: A Survey. *Annals of Discrete Mathematics*, pages 287–326, 1979.

[39] L. A. Hall, D. B. Shmoys, and J. Wein. Scheduling To Minimize Average Completion Time: Off-line and On-line Algorithms. In *Proc. of the of the Seventh ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 142–151, 1996.

[40] 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 of *Lecture Notes in Computer Science*, pages 191–202. Springer, 2000.

[41] S. Hotovy. Workload Evolution on the Cornell Theory Center IBM SP2. In D. G. Feitelson and L. Rudolph, editor, *Proc. of 2nd Workshop on Job Scheduling Strategies for Parallel Processing*, Lecture Notes in Computer Science, pages 27–40. Springer, 1996.

[42] 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*. Springer, 2003, to appear.

[43] The *hpcLine* at the Paderborn Center for Parallel Computing (PC^2). `http://www.upb.de/pc2/services/systems/psc/index.html`, October 2003.

[44] Access to the HWW Compute Platforms. `http://www.hlrs.de/hw-access/access/`, October 2003.

[45] ILOG CPLEX. `http://www.ilog.com/products/cplex/`, October 2003.

[46] D. Jackson, Q. Snell, and M. Clement. Core Algorithms of the Maui Scheduler. In D. G. Feitelson and L. Rudolph, editor, *Proc. of 7th Workshop on Job Scheduling Strategies for Parallel Processing*, volume 2221 of *Lecture Notes in Computer Science*, pages 87–103. Springer, 2001.

[47] J. Jann, P. Pattnaik, H. Franke, F. Wang, J. Skovira, and J. Riodan. Modeling of Workload in MPPs. In D. G. Feitelson and L. Rudolph, editor, *Proc. of 3rd Workshop on Job Scheduling Strategies for Parallel Processing*, volume 1291 of *Lecture Notes in Computer Science*, pages 95–116. Springer, 1997.

[48] Workshops on Job Scheduling Strategies for Parallel Processing. `http://www.cs.huji.ac.il/~feit/parsched/`, October 2003.

[49] P. Keleher, D. Zotkin, and D. Perkovic. Attacking the Bottlenecks in Backfilling Schedulers. *Cluster Computing: The Journal of Networks*, 3, 2000.

[50] A. Keller. Personal communication about users of CCS and how they behave, May–October 2002.

[51] A. Keller and A. Reinefeld. CCS Resource Management in Networked HPC Systems. In *Proc. of Heterogenous Computing Workshop HCW'98 at IPPS*, pages 44–56. IEEE Computer Society Press, 1998.

[52] 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.

[53] J. Krallmann, U. Schwiegelshohn, and R. Yahyapour. On the Design and Evaluation of Job Scheduling Algorithms. In D. G. Feitelson and L. Rudolph, editor, *Proc. of 5th Workshop on Job Scheduling Strategies for Parallel Processing*, volume 1659 of *Lectures Notes in Computer Science*, pages 17–42. Springer, 1999.

[54] W. E. Leland and T. J. Ott. Load-Balancing Heuristics and Process Behavior. In *Proc. of SIGMETRICS Conference Measurement & Modeling of Computer Systems*, pages 54–69, 1986.

[55] 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.

[56] IBM Scalable Parallel (SC) software - LoadLeveler. `http://www.ibm.com/servers/eserver/pseries/library/sp\protect\unhbox\voidb@x\kern.06em\vbox{\hrulewidth.3em}books/loadleveler.html`, October 2003.

[57] V. Lo, J. Mache, and L.Windisch. A Comparative Study of Real Workload Traces and Synthetic Workloads for Parallel Job Scheduling. In D. G. Feitelson and L. Rudolph, editor, *Proceedings of the 4th Workshop on Job Scheduling Strategies for Parallel Processing*, volume 1459 of *Lecture Notes in Computer Science*, pages 25–46. Springer, 2002.

[58] Platform Computing, LSF - Load Sharing Facility. `http://www.platform.com/products/wm/LSF/index.asp`, October 2003.

[59] U. Lublin and D. G. Feitelson. The Workload on Parallel Supercomputers: Modeling the Characteristics of Rigid Jobs. TR 2001-12, Hebrew University, Jerusalem, October 2001.

[60] HPC Workload/Resource Trace Respository. `http://supercluster.org/research/traces/index.html`, October 2003.

[61] Official MAUI Homepage. `http://supercluster.org/maui`, October 2003.

[62] G.E. Moore. Cramming More Components Onto Integrated Circuits. *Electronics*, 38(8), 1965.

[63] A. Mu'alem and D. G. Feitelson. Utilization, Predictability, Workloads, and User Runtime Estimates in Scheduling the IBM SP2 with Backfilling. Technical Report 2000-33, Institute of Computer Science, The Hebrew University, July 2000.

[64] 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.

[65] J. Nabryzski, J.M. Schopf, and J. Weglarz (Eds.). *Grid Resource Management: State of the Art and Future Trends*. Kluwer Academic Publishers, 2003.

[66] T. D. Nguyen, R. Vaswani, and J. Zahorjan. Maximizing Speedup through Self-Tuning of Processor Allocation. In *Proc. of the 10th International Parallel Processing Symposium*, pages 463–468. IEEE Computer Society Press, 1996.

[67] NPACI JOBLOG Job Trace Repository. `http://joblog.npaci.edu/`, October 2003.

[68] PACX-MPI Project Homepage. `http://www.hlrs.de/organization/pds/projects/pacx-mpi/`, October 2003.

[69] PBS Pro Home. `http://www.pbspro.com`, October 2003.

[70] PC^2 Annual Report - 2002, February 2003.

[71] D. Perkovic and P. Keleher. Randomization, Speculation, and Adaptation in Batch Schedulers. In *Supercomputing (SC2000)*, 2000.

[72] M. Pinedo. *Scheduling: Theory, Algorithms and Systems*. Prentice Hall, New Jersey, 2^{nd} edition, 2002.

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

[74] G. Kliewer S. Grothklags. Personal communication about modelling the scheduling process as an integer problem and solving it with the ILOG CPLEX library, November-December 2002.

[75] U. Schwiegelshohn. Preemptive Weighted Completion Time Scheduling of Parallel Jobs. In *Proceedings of the European Symposium of Algorithms (ESA 96)*, volume 1136 of *Lecture Notes in Computer Science*, pages 39–51. Springer, 1996.

[76] U. Schwiegelshohn, W. Ludwig, and P.S. Yu J.L. Wolf, J. Turek. Smart SMART Bounds for Multiprocessor Response Time Scheduling. *SIAM Journal of Computing*, 28:237–253, 1998.

[77] U. Schwiegelshohn and R. Yahyapour. Analysis of First-Come-First-Serve Parallel Job Scheduling. In *Proceedings of the 9^{th} SIAM Symposium on Discrete Algorithms*, pages 629–638, 1998.

[78] U. Schwiegelshohn and R. Yahyapour. Improving First-Come-First-Serve Job Scheduling by Gang Scheduling. In D. G. Feitelson and L. Rudolph, editor, *Proc. of 4th Workshop on Job Scheduling Strategies for Parallel Processing*, volume 1459 of *Lecture Notes in Computer Science*, pages 180–198, 1998.

[79] U. Schwiegelshohn and R. Yahyapour. Fairness in Parallel Job Scheduling. *Journal of Scheduling*, 3(5):297–320, 2000.

[80] Dolphin Interconnect Solutions Inc. `http://www.dolphinics.com/products/`, October 2003.

[81] K. C. Sevcik. Application Scheduling and Processor Allocation in Multiprogrammed Parallel Processing Systems. *Performance Evaluation 19(2-3)*, pages 107–140, March 1994.

[82] 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.

[83] L. Smarr and C. E. Catlett. Metacomputing. *Communications of the ACM*, 35(6):44–52, June 1992.

[84] A. Streit. Evaluation of Scheduling Strategies in Metacomputing (in German). Master's thesis, Dortmund University, 1999.

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

[86] 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.

[87] 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.

[88] J. Subhlok, T. Gross, and T. Suzuoka. Impact of Job Mix on Optimizations for Space Sharing Schedulers. In *Proc. of Supercomputing'96 Conference*. ACM Press and IEEE Computer Society Press, 1996.

[89] Parallel Workloads Archive. `http://www.cs.huji.ac.il/labs/parallel/workload/`, October 2003.

[90] D. Talby and D. G. Feitelson. Supporting Priorities and Improving Utilization of the IBM SP2 Scheduler Using Slack-Based Backfilling. In *Proc. of 13th International Parallel Processing Symposium*, pages 513–517. IEEE Computer Society Press, 1999.

[91] J. Turek, W. Ludwig, J.L. Wolf, L. Fleischer, P. Tiwari, J. Glasgow, U. Schwiegelshohn, and P. S. Yu. Scheduling Parallelizable Tasks to Minimize Average Response Time. In *Proc. of the 6th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA-94)*, pages 200–209. ACM Press, 1994.

[92] J.M. van den Akker, C.A.J. Hurkens, and M.W.P. Savelsbergh. Time-Indexed Formulations for Single-Machine Scheduling Problems: Column Generation. *Journal on Computing*, 12(2):111–124, 2000.

[93] K. Windisch, V. Lo, R. Moore, D. Feitelson, and B. Nitzberg. A Comparison of Workload Traces from Two Production Parallel Machines. In *6th Symposium Frontiers Massively Parallel Computing*, pages 319–326, 1996.

[94] R. Yahyapour. *Design and Evaluation of Job Scheduling Strategies for Grid Computing.* PhD thesis, Computer Engineering Institute, University Dortmund, 2002.

[95] Y. Zhang, H. Franke, J. E. Moreira, and A. Sivasubramaniam. Improving Parallel Job Scheduling by Combining Gang Scheduling and Backfilling Techniques. In *Proc. of the 14th International Conference on Parallel and Distributed Processing Symposium*, pages 133–144. IEEE Computer Society Press, 2000.

[96] 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 of *Lecture Notes in Computer Science*, pages 133–158. Springer, 2001.

[97] D. Zotkin and P. Keleher. Job-Length Estimation and Performance in Backfilling Strategies. In *Proc. of 8th High Performance Distributed Computing Conference (HPDC)*, pages 236–243. IEEE Computer Society Press, 1999.