Adaptive Job Scheduling via Predictive Job Resource Allocation

Lawrence Barsanti and Angela C. Sodan

University of Windsor, Canada, {barsant,acsodan}@uwindsor.ca
http://davinci.newcs.uwindsor.ca/~acsodan

Abstract. Standard job scheduling uses static job sizes which lacks flexibility regarding changing load in the system and fragmentation handling. Adaptive resource allocation is known to provide the flexibility needed to obtain better response times under such conditions. We present a scheduling approach (SCOJO-P) which decides resource allocation, i.e. the number of processors, at job start time and then keeps the allocation fixed throughout the execution (i.e. molds the jobs). SCOJO-P uses a heuristic to predict the average load on the system over the runtime of a job and then uses that information to determine the number of processors to allocate to the job. When determining how many processors to allocate to a job, our algorithm attempts to balance the interests of the job with the interests of jobs that are currently waiting in the system and jobs that are expected to arrive in the near future. We compare our approach with traditional fixed-size scheduling and with the Cirne-Berman approach which decides job sizes at job submission time by simulating the scheduling of the jobs currently running or waiting. Our results show that SCOJO-P improves mean response times by approximately 70% vs. traditional fixed-size scheduling while the Cirne-Berman approach only improves it 30% (which means SCOJO-P improves mean response time by 59% vs. Cirne-Berman).

1 Introduction

Most job-scheduling approaches for parallel machines apply space sharing which means allocating CPUs/nodes to jobs in a dedicated manner and sharing the machine among multiple jobs by allocation on different subsets of nodes. Some approaches apply time sharing (or better to say a combination of time and space sharing), i.e. use multiple time slices per CPU/node [23]. This is typically done via so-called gang scheduling which explicitly synchronizes the time slices over all nodes. Such time sharing creates multiple virtual machines which offers more flexibility for scheduling. Consequently, gang scheduling is shown in several studies to provide better response times and higher machine utilization (see, e.g., [9][10]). On the downside, gang scheduling involves process-switching overhead and increases the memory pressure.

A different option of flexible scheduling that avoids additional memory pressure is adaptive CPU/node-resource allocation. The standard resource-allocation approach in job schedulers uses static job sizes: jobs request a certain number of CPUs/nodes to run (therefore, called rigid). Adaptive resource allocation means that the number of resources can be decided dynamically by the system. The precondition is that the jobs

can deal with this dynamic resource allocation: either being moldable, i.e. able to adjust to the resource allocation at job start time, or being malleable, i.e. able to adjust to changes in the resource allocation during the job's execution. Then, adaptation may be used 1) to reduce fragmentation by adjusting the jobs' sizes to better fit into the available space, and 2) to adapt to varying system loads by reducing sizes if the system load is high and increasing sizes if the system load is low.

Malleability requires a special formulation of the program because the work to be performed per node changes dynamically—thus, we cannot expect every job to be malleable (though, in separate work, we address making applications malleable [22]). Moldability is easier to accomplish because often programs anyway initialize themselves according to the size with which they are invoked: a survey conducted among supercomputing-center users [5] found that most jobs (98%) were moldable, i.e. able to configure themselves as needed at start time. Based on the exploitation of moldability, Cirne-Berman [5] present a scheduler that employs an egoistic model and lets each job, after schedule simulation with different sizes, select the size which provides the best response time for the job. Indeed, results in [3][16] suggest that molding provides sufficiently good results though our results with SCOJO [21] suggest that adaptation with runtime changes of job sizes performs clearly better.

Our SCOJO scheduler presented in [21] supports both start time adaptation for moldable and runtime adaptation for malleable jobs, while avoiding molding and only applying runtime adaptation if the jobs are long. In this paper, we present SCOJO-P, an extension of SCOJO that supports simpler workloads with only rigid and moldable jobs and also molds long jobs. To solve the problem of determining proper sizes, which is especially critical for long jobs, we employ a heuristic system-load prediction model.

In summary, SCOJO-P provides the following innovative contributions:

- employment of adaptation for both reduction of fragmentation and adjustment to differently high system load
- provision of heuristics for choosing job sizes under molding that are based on knowledge about the overall system load
- a solution with low time complexity
- consideration of the system load (including estimated future arrivals of jobs) over the whole runtime of the job

We compare SCOJO-P to a traditional non-adaptive scheduler and to the Cirne-Berman approach by evaluating all approaches in a simulation study. For both, the workload modeling and the prediction, we employ the Lublin-Feitelson model [13]. Our results show that SCOJO-P outperforms the other approaches.

2 Related Work

Almost all existing work on adaptive scheduling is done in the context of space sharing. A number of such approaches aim at minimizing the makespan, i.e. the overall runtime, for a static set of jobs, while focusing on the provision of tight worst-case bounds [8][26]. These approaches apply a two-phase scheduling: they first determine the size for the jobs and then schedule the jobs. Realistic approaches need

to consider dynamic job submission and they aim at a reduction of average response times and average slowdowns (response times in relation to runtimes). Furthermore, most adaptive approaches apply molding only. Mere molding of jobs bears the problem that a job might run earlier with fewer CPUs but get a better response time if started later with more CPUs/nodes. Thus, the prediction quality regarding what the best solution for the job is becomes critical. The approach of Cirne and Berman [5] molds jobs at the time of job submission without using any central control: predictions are based on simulating the schedule for different job sizes and then selecting the size for which the best response time is obtained. We discuss this approach in more detail below. A few approaches are based on runtime adaptation for malleable jobs [6][15][17]. Most of these approaches exploit adaptation with the goal to adapt to varying system load. The approach by Naik et al. [15] adapts resource allocation only for medium- and long-running jobs. Short jobs are molded. The approach attempts to schedule all jobs from the queue but sets a limit for medium and long jobs to prevent starvation of short jobs. Dynamic adaptation for malleable jobs may keep jobs scheduled while adjusting the resource allocation [6][15] or checkpoint/preempt jobs and re-decide the job allocation [17].

The two basic approaches to decide about the job sizes are resource-based partitioning and efficiency-based partitioning [9]. Resource-based partitioning typically comes in the form of EQUI partitioning which means assigning the same number of resources to each job. This approach yields suboptimal performance in the general case as it does not consider how well the jobs use the resources [3][14]. However, resource-based allocation can be improved by defining different job-size classes like small, medium, large [15][2] and applying EQUI per job-size class—which comes close to efficiency-based partitioning. Efficiency-based partitioning exploits the efficiency characteristics of the applications and allocates more resources to jobs that make better use of them, which typically leads to the overall best results [3][14]. Similar to resource-based partitioning, efficiency-based partitioning may be applied in the form of providing equal efficiency to all jobs in the system (EQUI-EFF). In [12], the ratio of runtime to efficiency is used for efficiency-based partitioning. Job sizes may also be chosen to keep some CPUs/Nodes idle in anticipation of future job arrivals. The work of Rosti et al. [18] combined this idea with EQUI partitioning and limiting the job sizes to a certain percentage of the machine size, either statically or in dependence of the waiting-queue length. In the approach of Parsons and Sevcik [17], first the minimum size is allocated and, then, any leftover resources are assigned to reduce fragmentation.

If exploiting the jobs' efficiency characteristics, speedup/efficiency functions are needed. Secvik's model presented in [19] addresses dynamically changing parallelism but the ideas are related to changing job sizes to obtain better efficiency: the model uses phase-wise linearly approximation for CPU/node allocations between minimum, average, and maximum parallelism. Downey [7] presents a more sophisticated model which also originally was meant to describe variations in parallelism and is adopted by the Cirne-Berman scheduler for speedup-curve modeling. It is briefly discussed in Section 4.6.

Furthermore, all partitioning approaches should consider minimum allocations (potentially defined by memory constraints), maximum allocations (beyond which speedup drops), and potential other job-size constraints like power-of-two [5][13][15].

3 The Cirne-Berman Scheduler

The scheduler presented by Cirne and Berman in [5] decides the best job size at job-submission time. The scheduler takes a list of different possible job sizes and corresponding runtimes. The number of different sizes is determined randomly as well as the probability that the sizes are power-of-two. The scheduler then simulates the scheduling of the job for each possible size separately, taking into account current system load, i.e. the jobs currently in the waiting queue or running. After performing all simulations for all possible sizes, the size is chosen which provides the best response time for the job, and the job is submitted to the waiting queue with this size. This means that the approach can be set on top of an existing scheduler, provided that a simulator is available with the same scheduling algorithm as employed in the actual job scheduler. For the simulation, it is assumed that the job's actual runtime is equal to the estimated runtime. The scheduler uses conservative backfilling with best-fit selection. The scheme used for priority assignment and aging is not specified. The approach was evaluated with traces from supercomputer centers (considering all jobs to be moldable), combined with Downey's speedup model which we briefly discuss in Section 4.6.

4 The SCOJO-P Space Sharing Scheduler

4.1 The Original SCOJO Scheduler

SCOJO [20][21] incorporates standard job-scheduling approaches like priority handling (classifying jobs into short, medium, and long and assigning higher priorities to shorter jobs), aging (to prevent starvation), and EASY backfilling. EASY backfilling means to permit jobs to be scheduled ahead of their normal priority order if not delaying the start time of the first job in the waiting queue.

The original SCOJO scheduler applies either standard space sharing or gang scheduling and can combine both with adaptive resource allocation. SCOJO can handle mixtures of rigid, moldable, and malleable jobs. SCOJO supports

- Adaptation to varying system load (jobs running and jobs in the waiting queue)
- Fragmentation reduction

The former exploits the fact that speedup curves are typically approximately concave (due to increasing relative overhead), i.e. if job sizes are reduced, the jobs run at higher levels of efficiency which improves the effective utilization of the system towards the progress of the jobs' execution. Then, running more jobs while reducing their sizes utilizes the resources better if the system load is high. Though the jobs run longer, in the end, all jobs (on average) benefit by shorter wait and shorter response times. If the system load is low, the jobs can use more resources to reduce their runtime up to their maximum size (N_{max}) beyond which the runtime would decline. Furthermore, SCOJO adjusts job sizes in certain situations to fit jobs into the machine that otherwise could not run, while leaving resources unused.

To implement system-load adaptation and fragmentation reduction, SCOJO divides into the following major steps (details can be found in [21]):

- Determine the job target sizes in dependence on the system load
- Shrinkage or expansion of running malleable jobs to their target sizes; allocation of all new malleable/moldable jobs with their target sizes
- During backfilling, potentially further shrinkage of new short or medium adaptable (moldable or malleable) jobs to fit them into the machine
- Potentially expansion of new moldable or malleable jobs to exploit any unused resources

The system load is estimated by calculating the needed number of nodes $N_{needed} = \sum_i N_{opt,i}$ which represents the sum of the optimum size requirements of all currently running and waiting jobs. We then classify the current resource needs into a) low, b) normal, and c) high according to whether all jobs in running and waiting queue with their optimum sizes N_{opt}: a) fit into the machine with a multiprogramming level of 1 while still leaving some space, b) fit with a potentially higher multiprogramming level, or c) do not fit with even the maximum multiprogramming level. This means we have either unused space, utilize the machine well, or have more jobs than fit without adaptation. If the system load is normal, optimum sizes are used. A high system load suggests to shrink sizes; and a low system load suggests to expand sizes. The exact factors for expanding and shrinking are calculated by trying to fit all jobs into the machine (high load) or utilize all resources (low load). This is done by decreasing or increasing all adaptable jobs' sizes relative to their optimum size, i.e. by the same percentage vs. their optimum size. This makes sure that long jobs are not given any advantage if having high efficiency. To avoid configuration thrashing and adaptation with minor benefits, we consider reconfiguration only in certain time intervals and only if the change in the resource needs is relevant. Note that the system load is likely to change with day-night cycle as otherwise the machine would be overcommitted/saturated.

SCOJO does not apply any special measures to address power-of-two jobs as studies found that the power-of-sizes appear in most cases to be superficial, i.e. to stem more from standard practice rather than inherent properties of the applications [4].

Jobs are classified according to runtime. The original SCOJO takes long jobs as either rigid or malleable but does not mold them because the system load is likely to change over the runtime of long jobs. Then with a lack of prediction and consideration of details in the schedule, the initial size may prove to be disadvantageous to the job (if chosen smaller than desirable during a high-load phase) or disadvantageous to other jobs (if chosen too large during a low-load phase). Similarly, size reduction or size expansion to reduce fragmentation may especially be harmful regarding long jobs. Short jobs are not worth runtime adaptation and are treated as either rigid or moldable. Medium jobs can be rigid, moldable, or malleable.

The adaptive resource allocation of SCOJO was shown to improve response times and bounded slowdowns by up to 50% and to also tolerate reservations for local or grid jobs well [20][24]. These results were obtained with artificial workloads and the Lublin-Feitelson workload model, and combination with either space sharing or gang scheduling. Thus, for space sharing with the Lublin-Feitelson workload model and

60% moldable / 40% malleable jobs, we obtain 43% improvement in average response times and even 60.5% improvement in slowdowns [24].

4.2 The New SCOJO-P Scheduler

SCOJO-P [1] extends SCOJO in various ways, while restricting it regarding application characteristics. SCOJO-P is strictly space sharing and only handles rigid and moldable jobs. This makes SCOJO-P suitable for jobs which are not especially designed for adaptation and matches standard job mixes in supercomputer centers as found by Cirne and Berman [5]. It also makes the results comparable to the Cirne/Berman approach.

The most important extensions of SCOJO-P are to consider the average load on the system over the runtime of a job when choosing a size for the job and to include the prediction of future job submissions.

The overall algorithm includes the following steps:

- Adaptive target-size determination: selects a size (N_{target}) for the candidate job under concern for being scheduled (J_S) that will help the system maintain a consistent workload.
- Try to start J_S: if the target size of J_S is greater than the number of currently available processors (i.e. $N_{avail} < N_{target}$), then J_S can start with less than N_{target} processors if doing so provides a benefit (shorter response time) to J_S vs. being scheduled at a later time (when $N_{target} \leq N_{avail}$).
- Adaptive backfilling: adaptation is considered during backfilling in a simplified form.

Note that whereas SCOJO applies adaptation both at start time and, for malleable jobs, during their runtime, SCOJO-P only applies adaptation at start time as it exclusively supports molding. Fragmentation reduction is, however, considered when trying to fit a job into the system by shrinking its size below N_{target}.

Below we describe the different steps in detail.

4.2 Adaptive target-size determination

When determining the target size (N_{target}) of a job (J_s), all jobs that are currently running, that are in the waiting queue, or that are expected to arrive during the execution of J_s, are considered (the latter considers the corresponding statistical distribution of runtimes/sizes and the jobs' interarrival times). The target size of J_s is calculated using the following heuristic. The *Work* (average load per processor) is estimated over the runtime of J_s, assuming that J_s, the waiting jobs, and future jobs will all run with their optimal size, whereas, for running jobs, their allocated size is taken, i.e. initially

$$Load(J_s) = \sum_i work(job_i) / (MN * J_s) \text{ with}$$

$$work(job_i) = \sum_{i \text{ in Js, waiting, future}} N_{opt,i} * min(runtime_i(N_{opt,i}), J_s) + \sum_{i \text{ in running}} N_{allocated,i} * min(runtime_{remaining,i}, J_s)$$

with *MN* being the number of nodes in the machine. Since the load is calculated over the runtime of J_s, for all jobs, only the overlapped runtime is considered. For future jobs, average optimum sizes and corresponding optimum runtimes are used.

If *Load* is lower or higher than the ideal *Load* per processor, a modifying factor (determined by the fail ratio of the ideal load vs. the resulting load) is calculated and used to resize all jobs proportionally, and the load is recalculated. This recalculation of modifying factor and load is done iteratively until a load close to the ideal load (or as close as possible) is obtained. Note that the load calculation has to be redone as the runtime of J_s and the overlaps change. The ideal load cannot always be obtained because moldable jobs cannot expand/shrink beyond a maximum/minimum value and rigid jobs cannot be resized at all. If the ideal *Load* is set < 1, it means that all waiting and future jobs should ideally be scheduled immediately (rather than being queued) by reducing their size. The load then corresponds to utilization. Since the algorithm does not consider packing but only the load, it may be the case that neither the currently considered job nor any of the waiting or future jobs can actually fit into the machine at the current point in time; even with ideal *Load*. If set near the expected utilization the ideal *Load* can take average fragmentation loss from packing problems into considerations. The algorithm For the details of the algorithm, see **Fig 1**.

```
curr_target_runtime = Js.runtime (Js.optSize);  isOk_load = false;  sMod = 1.0;
curr_target_size = Js.optSize;  best_avg_load = Max_Integer;

do {
relevant_work = curr_target_runtime_Js * sMod * Js.optSize;

// sum up work of running jobs as far as execution would overlap with Js
for (all j in running_jobs) {
    relevant_runtime = min (j.remaining_runtime, curr_target_runtime_Js);
    relevant_work += relevant_runtime * j.size;
}

// sum up work of waiting jobs as far as execution would overlap with Js
for (all j in waiting_jobs) {
    relevant_runtime = min (j.runtime (sMod * j.optSize), curr_target_runtime_Js);
    relevant_work += relevant_runtime * sMod * j.optSize;
}

// sum up work for future jobs as far as execution would overlap with Js
// consider different job arrivals in different time intervals during the day-night cycle
future_short = future_med = future_long = 0;
for (all time_intervals that current_target_runtime_Js spans) {
   future_short += expected_short_jobs (time_interval);
   future_med += expected_medium_jobs (time_interval);
   future_long += expected_long_jobs (time_interval);
}
relevant_work +=
  future_short * (avg_short_size*sMod) *
```

```
      min(runtime(sMod*avg_short_runtime), curr_target_runtime_Js) +
      future_med * (avg_med_size * sMod) *
           min (runtime(sMod*avg_med_runtime), curr_target_runtime_Js) +
     future_long * (avg_long_size * sMod) *
          min (runtime(sMod*avg_long_runtime),, curr_target_runtime_Js);

// calculate the average system load
available_workProcessing = n_machine * curr_target_runtime_Js;
avg_load = relevant_work / available_workProcessing;

if ((avg_load ≥ ideal_avg_load – deltaS) && (avg_load ≤ ideal_avg_load + deltaS))
   isOk_load = true;
else  {  // determine size modifier
  prev_sMod = sMod;
  sMod = sMod * ( ideal_avg_load / avg_load);
  if (prev_sMod*Js.optSize == sMod*Js.optSize) break;   // no change in size
          curr_target_size_Js = sMod * Js.optSize
 curr_target_runtime_Js = Js.runtime(curr_target_size_Js);
}

if ( | avg_load - ideal_avg_load | < |  best_avg_load - ideal_avg_load| ) {
          best_avg_load = avg_load;
   best_sMod = prev_sMod;   counter=0;
} else {
     counter++;     if (counter == maxBadModifiers) break;
  }
} while (! isOk_load);  // loop terminates if load o.k. or if almost no change anymore
```

Fig. 1. Algorithm applied when calculating target size N_{target} for job J_s.

Note that, though the calculation changes all sizes of the job considered for scheduling, waiting jobs, and future jobs proportionally, the target size is only determined for J_s. The other sizes are not recorded but are determined when the jobs are up for scheduling. Nevertheless the algorithm considers the global picture of the overall load.

Furthermore, by calculating the average load over the entire runtime of the job, the job gets a size which is appropriate for both potential high load and low load phases. This is important when scheduling long running moldable jobs because it prevents the jobs from starving the system in order to help themselves and from starving themselves to help the system.

The complexity of this algorithm depends on how quickly it converges to the ideal load. In a worst case, every size of the job being scheduled will be tested. Because the runtime changes with every iteration step, the load incurred by running, waiting, and future jobs also changes. Thus, using the modifier does not always provide better results and could even cause the algorithm to thrash. We prevent this from happening by comparing the load produced by each modifier to the best load obtained so far (i.e. the load that came closest to the ideal load). If after a couple iterations no new

modifier has produced a load that is better than the current best load the algorithm terminates and uses the modifier that provided the current best load. In practice, we found only very few iterations to be needed.

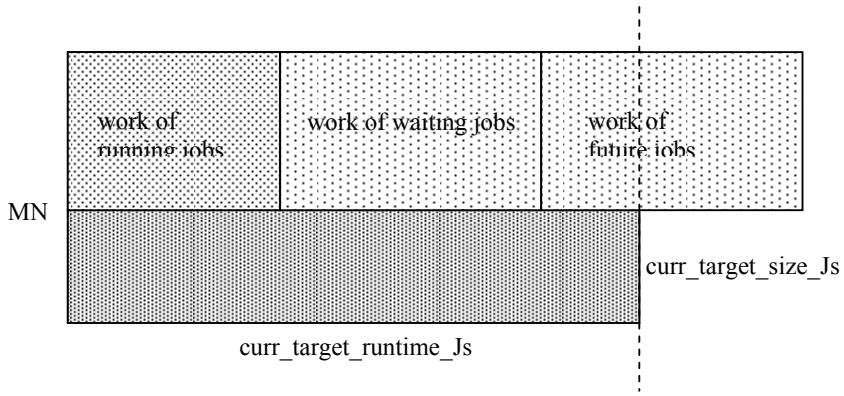

Fig. 2. Visualization of the load-estimation heuristic. The graphic shows a situation where not all jobs would fit into the machine with current size and corresponding runtime during the runtime of J_s. If relating the workload to the runtime of J_s, $Load > 1$. Whether the resulting load is considered ideal or not, depends on the setting of the parameters. However, with our settings, we would normally modify the job size to obtain a $Load < 1$.

4.3 Trying to schedule the job with adaptive target size

After determining the target size of the job, the scheduler tries to allocate the job to the machine. It is possible that, however, not enough nodes are currently available to schedule the job. Rather than considering the attempt of scheduling the job as failed, the scheduler decides whether to start the job right away with smaller than the target size (i.e. allocate fewer resources) or whether to start the job at a later point of time with more processors (up to the calculated target size).

To make this decision, the scheduling of all currently running jobs is simulated to determine the different times at which the job can be started with larger sizes. The latest possible start time would be when it can run with the calculated target size. Then, it is decided whether the current or any later start time with increased size ($N_{avail} < \text{size} \leq N_{target}$) provides a better response time for the job. If the current time provides the best response time, the job is started with that size. Otherwise, the size with the calculated best response time is memorized and guaranteed as the jobs' later minimum size (worst-case scenario) with which it will be run. If the job is started with $a\ size < N_{target}$, this can be considered fragmentation reduction.

```
bestStartTime = currentTime;
bestResponseTime = Js.runtime (freeProcs);
bestSize=Js.target_size;

while (freeProcs < Js.target_size) {
    startTime = sim.time (sim.nextJob_finished);
    size = min(target_size_Js,sim.freeProcs);
    responseTime = startTime–currentTime+Js.runtime (size);
    if (responseTime < bestResponseTime)
       {bestResponseTime=responseTime; bestStartTime=startTime;    bestSize=size;}
}
if (bestStartTime == currentTime) schedule (Js, freeProcs);
else fixJobSize (bestSize);
```

Fig. 3. Finding the start time that delivers the best response time.

4.4 Adaptive backfilling

SCOJO-P also considers size adaptation during backfilling, using a simplified calculation for the target size. The algorithm considers all potential backfill jobs together. All jobs which would fit up to the top job become candidate jobs (note that this is not the full backfill condition) and their summed-up work (with optimum size) related to the available work up to the start time of the top job. Then, the jobs are uniformly resized by the same factor, aiming at fitting them all into the backfill hole. For each job, then an attempt is made to schedule it (with full backfill condition).

```
// determine the max.  possible runtime for a job not to delay the top waiting job J_top
max_runtime = Jtop.startTime – currentTime;

// determine size modifier (uniform for all backfilled jobs at current time)
available_workProcessing = max_runtime * freeProcs;
backfillWork = 0;
for (all jobs j in waitingQueue)
  if ((j.optSize < freeProcs && j.runtime(j.optSize) < max_time)
        backfillWork += j.runtime (j.optSize) * j.optSize;

sMod = backfillWork / available_workProcessing;

// resize all backfillable jobs by same factor
for (all jobs j in waitingQueue) {
    target_size = round (j.optSize * sMod);
    if ((target_size ≤ freeProcs && j.runtime (target_size) ≤ max_runtime) ||
       (target_size < freeProces – Jtop.optSize))
        { schedule (j, target_size);  freeProcs -= target_size; }
```

Fig. 4. Adaptation during backfilling.

4.5 Discussion of Expected Behavior and Benefits

The main benefits of the SCOJO-P algorithm as presented above are that the workload is estimated over the whole runtime of the job that is the candidate for scheduling. This estimation provides a good global picture, though it is heuristic. Sizes for long jobs are determined to provide an average reasonable size if both low and high load phases occur during the jobs' runtime. This reduces the risk that sizes are chosen too high which would benefit the candidate job or too small which would benefit the other jobs. Rather optimal efficiencies are targeted.

If comparing SCOJO-P to the Cirne-Berman approach, Cirne-Berman makes decisions per job at job submission time based on simulation of the schedule. However, new jobs with higher priorities can change the picture though the Cirne-Berman scheduler may still work well as long as only short jobs can get ahead. If priorities would be assigned with a different scheme such as giving long jobs higher priority, the Cirne-Berman approach is likely not to work well anymore whereas SCOJO-P considers them as part of the statistically based estimate. Furthermore, in SCOJO-P, prediction and runtime overestimates are easier to integrate. As shown above, predication only adds a term in the estimation of the load. Regarding overestimates, for future jobs, anyway statistics based on actual runtimes are used. For running and waiting jobs, the workload estimation from above can be refined by taking the runtimes as user-estimated runtimes and adding a statistical over-estimate model such as [25]. This may not correctly estimate the runtime per job but, at least with a large number of jobs in the system, provide a reasonable statistical approximation of the overall load in which we are interested only. Alternatively performance databases may be employed to obtain estimates of the actual runtimes [11] which would work well for Cirne-Berman, too.

4.6 The Speedup Model Used

The implementation of the function *rutime(size)* requires a speedup model. The Cirne-Berman [4] statistical model could have been used to generate random min/max sizes and a random speedup curve for each job. The Cirne-Berman model is based on the Downey speedup model [7], originally meant to model parallelism behavior like [19]. With adoption to speedup-up curves, this model defines the curve by the average parallelism (the maximum speedup a job can achieve) and a coefficient of the job's variance in parallelism (which determines how fast the job reaches its maximum speedup). Cirne-Berman obtained distribution functions for these two parameters and coefficients' values fitting the observed data from their study and, based on the resulting statistical model, randomly generate speedup curves for the jobs. The moldability model is combined with the general workload by randomly generating the maximum speedup (independently from the runtime generated by the workload model) and mapping the generated runtime onto this curve. We implemented this model and found that the created speedup curves are not correlated with the runtimes/sizes produced by the Lublin-Feitelson model. Thus, the combined workload model often produces jobs with a maximum size far beyond the machine size. Furthermore, it can produce, for example, a job that runs in 20 seconds on 4

processors, while the Cirne-Berman speedup model could produce a speedup curve where the optimum job size is 32 processors yielding a runtime of 2 seconds. This would be similar to generating job runtimes and job sizes independently (though indeed they are correlated). This lack of correlation does not affect the Cirne-Berman scheduler as it simply chooses the size/runtime combination that produces the best simulation results. However, this approach does not work well with SCOJO-P because it tries to run all jobs using their optimum size and only shrinks and expands when appropriate.

Thus, for our main tests, we have reverted back to a simpler model as used in [21], assuming that the sizes produced by the workload model (or given by the user) represent a size for which a good cost/efficiency ratio is obtained. Though not required by the scheduler, this size is ideally the *processor working set* (*PWS)*, i.e., the number of processors for which the ratio of runtime to efficiency is optimal [12]:

$N_{PWS} = \{N \mid with\ T_N / E_N = T_1 / N * 1 / E_N^2) \text{ is minimal}\}$

with T_N being the runtime and E_N the efficiency for a corresponding job size N. No larger size should be chosen unless otherwise resources are idle.

Then, we calculate the speedup curve in the following way:

- We take the size created for the job by the statistical workload model as its optimum size N_{opt}. The assumption is that the user approximately knows which is the most meaningful size for the job. If the job is rigid, this will remain its size, if the job is moldable, this is the base size of the job. Though it is not necessarily N_{PWS}, we can perceive it as the size which makes sense under normal load conditions. Then, consequently, $Runtime(N_{opt})$ is the time generated by the workload model. In the specific test setting which we use, $N_{opt}=N_{PWS}$.
- We define N_{max} and N_{min} relative to N_{opt} with always the same proportional factor, and interpolate the speedup curve between these points linearly (which is similar to [19]. N_{max} represents the size beyond which the speedup curve declines and N_{min} the minimum size needed by the job, e.g. because of memory constraints, or the size below which no further significant efficiency benefits can be obtained. Note, that typically $N_{min} > 1$.

The SCOJO-P algorithm always considers N_{max} and N_{min} as bounds when determining sizes (which is omitted above in the pseudo code to keep it readable).

We also show results for using the Cirne-Berman adoption of Downey's model. To have a proper comparison to their implementation, we follow their approach in not correlating the generated speedup curve to the generated sizes/runtime though we agree with Downey's comment that user submissions are likely to be proportional to the maximum speedup [7]. (The latter means that a user is likely to choose a larger size—even if the machine is very busy—if the maximum speedup is very high. Then, we calculate N_{opt} by finding N_{PWS} from the speedup formula. For predictions of speedup for future jobs, we use median maximum speedups and median variances.

5 Experimental Evaluation

5.1 Test Environment and Measured Metrics

We evaluate utilization, wait times, response times (elapsed runtimes plus waiting times), and bounded slowdowns (response times in relation to runtimes with adjustment to a minimum runtime bound). The bounded slowdown (*BSl*), however, needs to be redefined for moldable jobs. We relate the slowdown to $runtime(N_{opt})$ which represents the standard size as it would be used without molding:

$runtime(N_{opt}) < bound \rightarrow BSl = max\ (T_{response} / max\ (runtime(N_{opt}), bound), 1)$
$runtime(N_{opt}) \geq bound \rightarrow BSl = T_{response} / runtime(N_{opt})$

We have set the bound to 30 seconds. Rather than using the geometric mean like Cirne-Berman [5] to avoid too much influence from long jobs, we not only calculate the overall arithmetic mean, but also perform separate evaluations for short jobs, medium jobs, and long jobs.

5.2 Workload Model

We evaluate SCOJO-P via simulation. As already mentioned above, we apply the Lublin/Feitelson statistical model for the workload generation [13], including runtimes, sizes, and interarrival times. This model is derived from existing workload traces and incorporates correlations between job runtimes and job sizes and daytime cycles in job-interarrival times. We cut off the head and the tail of the created schedule (the first and last 10% of the jobs in the schedule) to avoid that the fill and drain phase influence the results. We test 2 different variations of the Lublin-Feitelson workload: the basic one and a higher workload (one with shortened interarrival times).

Since there is no information yet about speedup curves from real application traces, we apply the model as described in Section 4.6. Regarding moldability, the study in [5] suggests that 98% of the jobs are moldable. The figure, however, sounds a bit too optimistic—if users say that they can submit jobs as moldable, it does not necessarily mean that, in practice, they would do so and that applications are moldable in such a high percentage of cases. Furthermore, these are so far results from a single study only. Thus, we test different percentages of moldable jobs, including 100%. If less than 100% jobs are moldable, moldability is distributed over the different job classes short, medium, long with equal probability.

We assume all generated runtimes to represent correct runtimes (i.e. we do not consider over-estimates as would be possible if adding the model presented in [25]) which is sufficient for our evaluation. For SCOJO-P, wrong estimates would actually be relatively easy to incorporate: only the average percentage of the overestimate would be needed to model the predictions for running, waiting, and future jobs as we consider averages of runtimes only. The Cirne-Berman approach is more heavily depending on estimates as the approach determines sizes by simulating the actual schedule. Since we apply the same workload model to all approaches, comparing to

the Cirne-Berman approach on the bases of correct runtimes is a conservative comparison regarding SCOJO-P. In other words, if including wrong estimates into the model, we expect SCOJO-P to perform relatively even better.

For details of the workload parameters, see **Table 1.** Note that in addition, we model the Cirne-Berman-Downey speedup model as described above.

Table 1. Workload parameters used for basic evaluation.

Machine size *MN*	128
Number of jobs in workload	10,000
Cut off for each fill and drain phase	5% of overall jobs each
ά parameter of Lublin/Feitelson model with impact on system load	ά =10.23 (basic workload W1) and ά = 9.83 (heavier workload W2)
Classification short jobs	*runtime* (N_{opt}) < 60 sec
Classification medium jobs	60 sec ≤ *runtime* (N_{opt}) < 1 hour
Classification long jobs	1 hour ≤ *runtime* (N_{opt})
% moldable jobs	80%, 90%, 100%
N_{opt}	as created by Lublin/Feitelson model
N_{min}	*max* {½ * N_{opt}, 1}
N_{max}	*min* {2 * N_{opt}, *MN*}
$E(N_{opt})$	0.65
$E(N_{min})$	0.8
$E(N_{max})$	0.4
runtime(N_{opt})	as created by Lublin/Feitelson model
runtime(N_{min})	*runtime*(N_{opt}) * $E(N_{opt})$ * ½ / $E(N_{min})$
runtime(N_{max})	*runtime*(N_{opt}) * $E(N_{opt})$ * 2 / $E(N_{max})$

We have set the efficiency values $E = speedup/MN$ such that, in our test cases, $N_{opt} = N_{PWS}$.

Future job submissions in different time intervals are determined by using 30-minute intervals as in the Lublin-Feitelson model and evaluating actual workload simulations to extract the numbers of short, medium, and long jobs submitted on average in each of 48 time intervals per day.

5.2 Approaches Tested

As mentioned above, SCOJO-P employs EASY backfilling and priority assignment according to runtime, giving highest priority to short jobs. Long and medium jobs are aged to prevent starvation; that is, their priority is increased after they have waited 5 times as long as their optimum runtime. We use the same basic approaches, including the priority handling and EASY backfilling, for all approaches used in our comparison to have a fair comparison. (Note that the original Cirne-Berman approach applied conservative backfilling.) We also do not impose any size constraints in neither of the approaches though the original Cirne-Berman approach generates only a

certain number of sizes and imposes a certain probability that the jobs's sizes have power-of-two constraints. We compare the following approaches:

- Basic scheduler without any adaptation (traditional)
- SCOJO-P with adaptation with prediction (predictive) or without prediction (non-predictive)
- Cirne-Berman approach for adaptation

The non-predictive of SCOJO is introduced to investigate how much the prediction contributes to the final results. For SCOJO-P, we additionally tested different load values for the target utilization. The one that performed best is 90% utilization. This is not surprising as this value corresponds to the maximum utilization which typically can be achieved on a machine, considering that there is always some fragmentation.

5.3 Experimental Results

We ran all tests four times with different random seeds and use the average for our results. We first test the scheduling approaches using our simple speedup model. The results for Workload W1 and 100% moldable jobs are shown in Figure 5 to Figure 8.

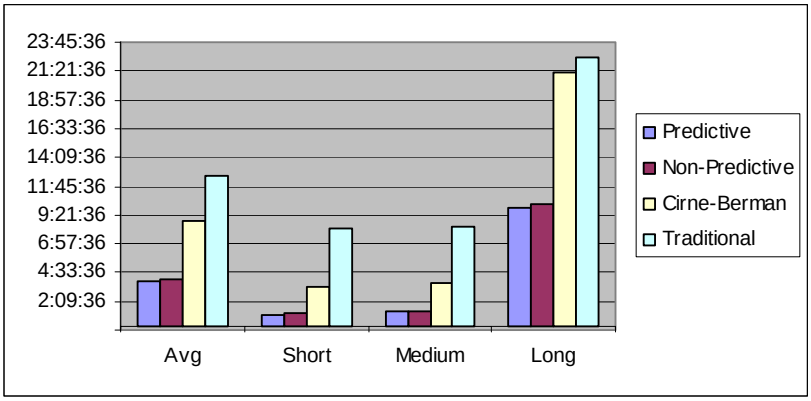

Fig. 5. Mean response times with basic Workload W1 (in hours), 100% moldable

From **Fig. 5.** , it can be seen that mean response times for jobs scheduled with SCOJO-P vs. Cirne-Berman are better for all job classes. Short and medium jobs are reduced to about 1/3 of their response times and long jobs to about 1/2. Regarding wait times, short and medium job again are cut to 1/3 but long jobs to 1/4, see **Fig. 6.** This suggests that SCOJO-P typically starts long jobs earlier, but with fewer processors than the Cirne-Berman approach does. Thus, runtime is increased but response time is actually decreased because of the earlier start time. Furthermore, using fewer processors for long jobs also leaves more room for short and medium jobs to squeeze through which explains their marginal improvement. To get a better insight into the behavior than the averages can provide for the highly varying result values and skewed distributions, we have calculated histograms. The response-time

graph is shown in **Fig. 9** (the other graphs are similar in their trend). We can see that SCOJO-P schedules more jobs with shorter response times (except for the initial classes of long jobs) and fewer jobs with excessively long response times. This applies to all job classes short, medium, and long, and supports that SCOJO-P produces better overall results.

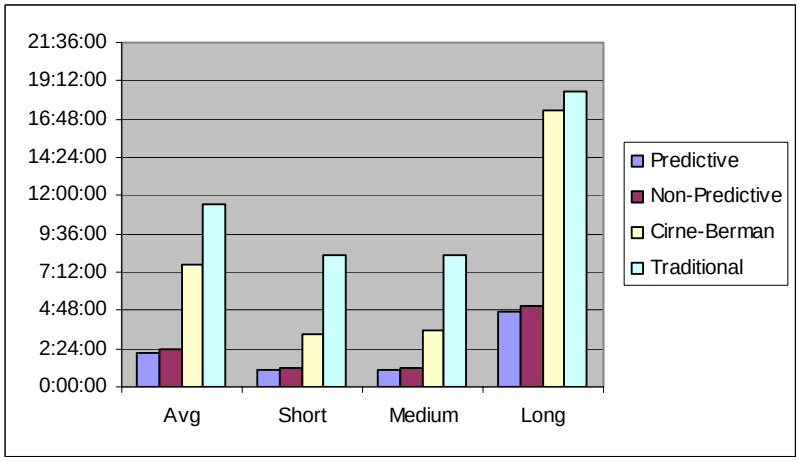

Fig. 6. Mean wait times for basic Workload W1 (in hours), 100% moldable.

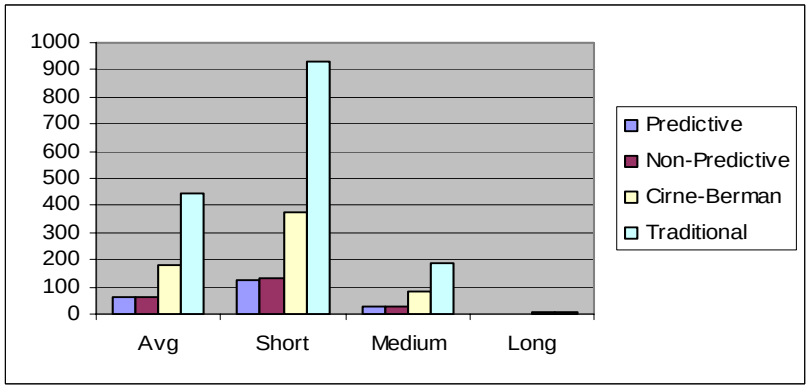

Fig. 7. Mean bounded slowdowns for basic Workload W1, 100% moldable.

Fig. 8. shows the number of adaptations that took place with each approach. Because it is considering the system as a whole, the SCOJO-P scheduler tends to shrink jobs rather than expand them; conversely, because the Cirne-Berman approach is trying to optimize each job individually it tends to expand jobs. The Cirne-Berman

approach actually produced higher system utilization than SCOJO-P (89.69% vs 78.6%). The reason is most likely that SCOJO-P shrinks more jobs during phases with high load and may leave processors empty so they can service jobs in the near future. However, SCOJO-P still obtains better mean response times which makes sense if shrinking jobs to run with higher efficiency.

Looking at the results for the non-predictive SCOJO, we find them to be only a little worse. This means that the prediction—at least, in its current version—does not provide as much benefit as we had originally expected.

Similar results were achieved with a workload where only 80% of the jobs were moldable. However, SCOJO-P actually performed slightly better (4%) with 80% moldable jobs, while Cirne-Berman performed a bit worse (-5%). This indicates that job shrinking in SCOJO-P might be a little too aggressive.

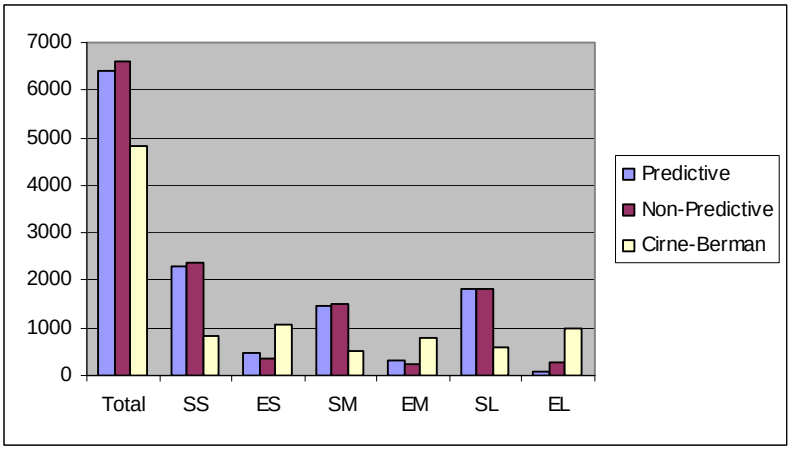

Fig. 8. Number of adaptations (W1, 100% moldable) that shrink (S*) or expand (E*) the job size vs. N_{opt}, calculated for short jobs (*S), medium jobs (*M), and long jobs (*L).

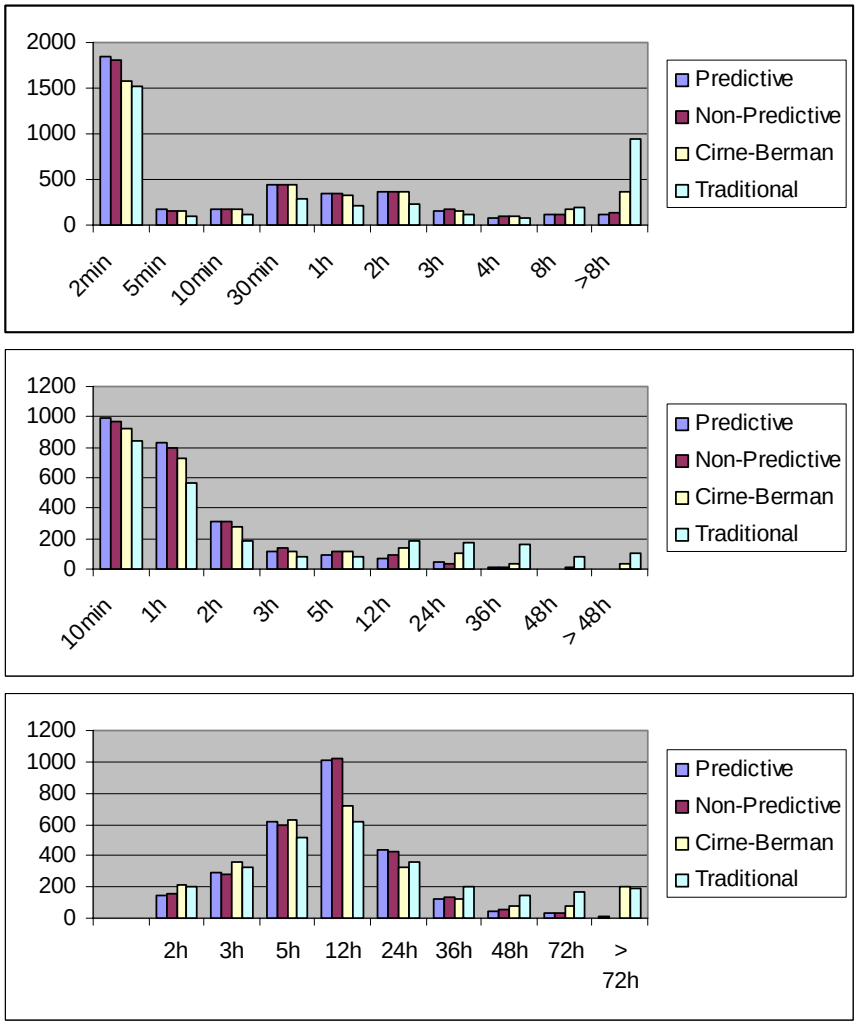

Fig. 9. Histograms for response times and short (top), medium (middle), and long (bottom) jobs. Note that the histogram categories are not equidistant to accommodate the skewed distributions. The labels mean: label value of the preceding category < result values ≤ label value of the current category. The histogram shows the number of job results falling into each category.

Fig. 10. to **Fig. 13.** show results for the higher Workload W2. As with the lower workload, SCOJO-P produces much better (67%) mean wait times for long jobs than the Cirne-Berman approach. This translates into a 48% improvement in the mean response time of long jobs which now benefit most. Looking at the adaptation statistics in **Fig. 13.** , we see that even when there is a heavy workload on the system,

the Cirne-Berman approach still tends to expand jobs. On the other hand, SCOJO-P is shrinking a greater number of jobs, thus allowing a greater number of jobs to run simultaneously. SCOJO-P is also benefiting from the increased processor effectiveness obtained from smaller job sizes.

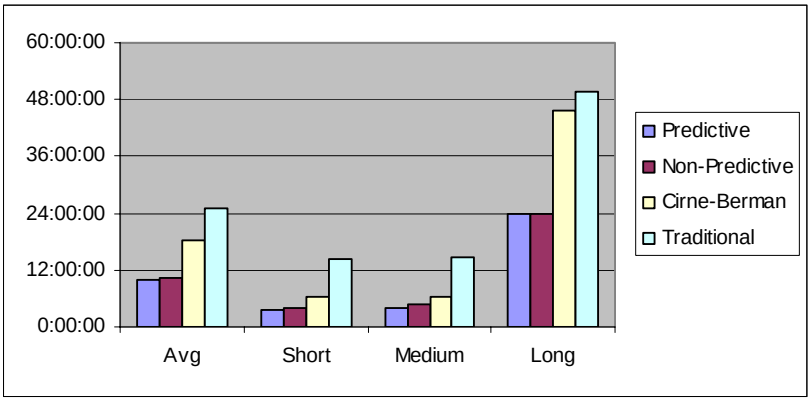

Fig. 10. Mean response times for Workload W2 (in hours), 100% moldable jobs.

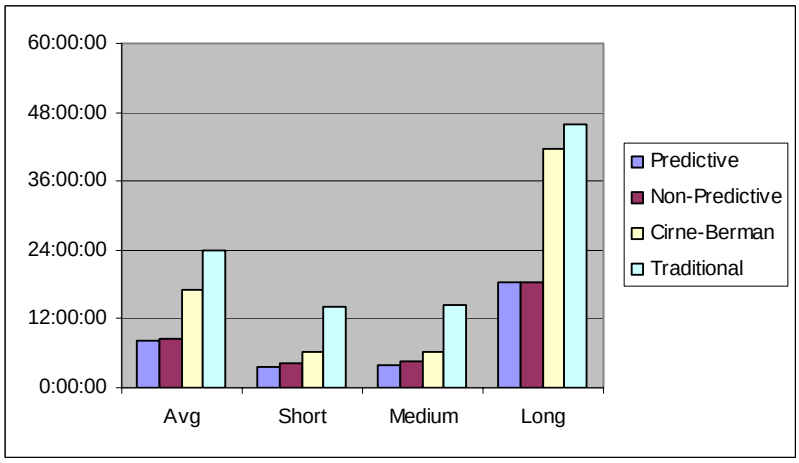

Fig. 11. Mean wait times for Workload W2 (in hours), 100% moldable jobs.

We also checked the results from the original SCOJO. Since our test environment and the generated random workloads are not exactly the same, a direct comparison is not possible. However, SCOJO reduces average response times by 50% if 80% of the long jobs are malleable (while 80% of the short and medium jobs are moldable). Adaptation with all classes being 80% moldable improves response times by approx.

35% vs. scheduling without adaptation. This means that the approx. 50% improvement which we get with SCOJO-P can in SCOJO only be accomplished with dynamic adaptation for malleable jobs.

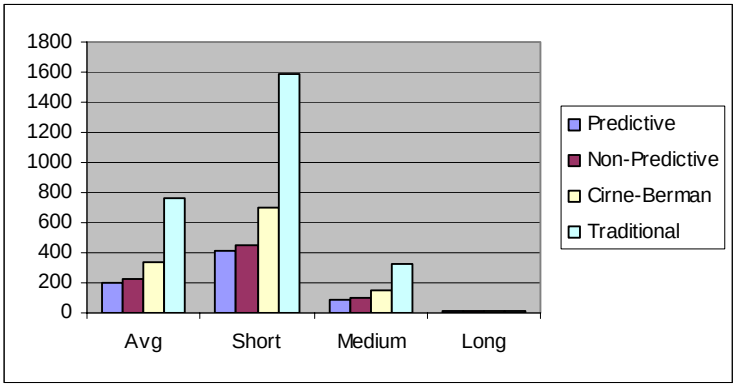

Fig. 12. Mean bounded slowdowns for Workload W2, 100% moldable jobs.

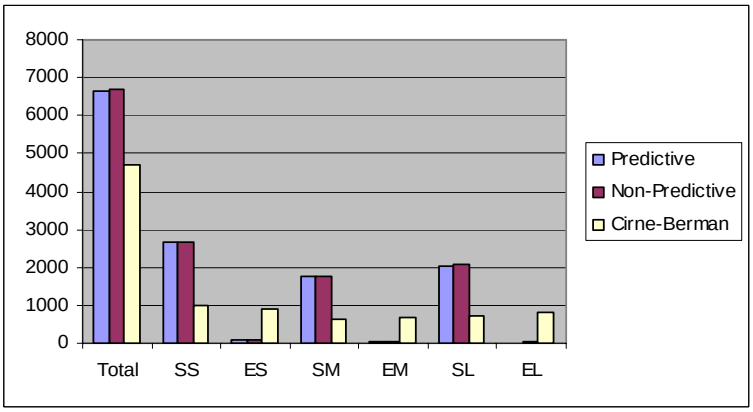

Fig. 13. Number of adaptations (W2, 100% moldable jobs) that shrink (S*) or expand (E*) the job size vs. N_{opt}, calculated for short jobs (*S), medium jobs (*M), and long jobs (*L).

Finally, we ran the tests (using two test runs) for W1 and 100% moldable again with the Cirne-Berman-Downey speedup model. The results for response times and bounded slowdowns are shown in **Fig. 14**. SCOJO-P still performs better, though only slightly. We found that N_{max} and therefore N_{opt} are created very high. Thus, with our speedup model, the average N_{opt} is 12 (8 for short, 9 for medium, and 20 for long jobs) and with the Cirne-Berman-Downey model it is 69. There is not much

difference for the different job classes with the latter (61 for short, 89 for medium, and 65 for long jobs). Note that the classification into short, medium, and long is based on the N_{opt} runtimes which changes the overall distribution of the jobs. The high values of N_{opt} greatly reduce the benefit of shrinking job sizes. However, as discussed above, we consider the created sizes as too large and as not properly correlated to the submitted sizes. Using this model, the non-predictive variant of SCOJO now performs better than the predictive variant. The reason is that the overly high N_{opt} values (which are far beyond the sizes with which the jobs are finally scheduled) negatively affect the predictions.

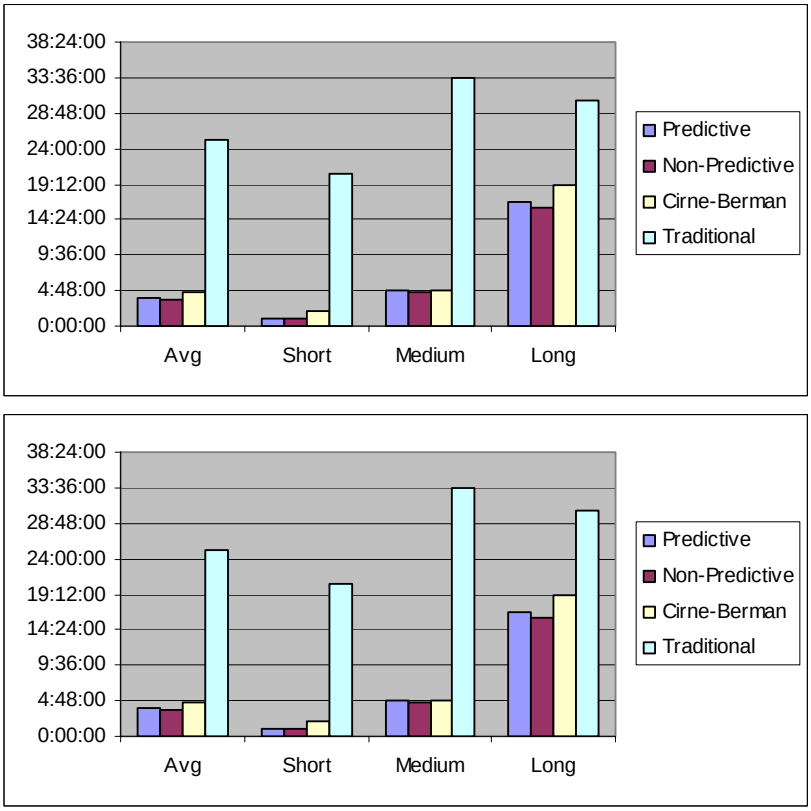

Fig. 14. Response times (top) and slowdowns (bottom) for W1 and 100% moldable jobs, using the Cirne-Berman-Downey speedup model.

6 Summary and Conclusion

We have presented the SCOJO-P scheduler for adaptive resource allocation at job start time. SCOJO-P considers the estimated load of the machine over the whole

runtime of the job to determine its ideal size. The load estimation includes an estimate about future job submissions. The Cirne-Berman approach for molding jobs, tries to maximize the benefits per jobs, which still converges to a situation where each job (on average) benefits. SCOJO-P directly considers the whole picture to balance the interests of the scheduled jobs with the interests of the other jobs. In the experimental study, SCOJO-P improves response times by 70% vs. traditional scheduling and by about 59% vs. the Cirne-Berman approach (which improves traditional scheduling by about 30%) if using a simple speedup model which takes the submission size as the optimal one. Investigating the effect of prediction, we found it contribute less to the good results than originally expected (though improvements are possible) and the main benefit stemming from considering the whole set of jobs on the system together. With the Cirne-Berman-Downey speedup model, optimal sizes for the generated curves are much higher, leading to less efficiency gain if shrinking jobs and therefore to SCOJO-P only being slightly better than the Cirne-Berman scheduler.

Acknowledgements

This research was in part supported by NSERC and by CFI (Grant No. 6191) with contributions from OIT and IBM.

References

[1] L. Barsanti, "An Alternative Approach to Adaptive Space Sharing", Honors Thesis, University of Windsor, Computer Science, September 2005.

[2] S.-H. Chiang, R.K. Mansharamani, and M.K. Vernon, "Use of Application Characteristics and Limited Preemption for Run-to-Completion Parallel Processor Scheduling Policies", *Proc. ACM SIGMETRICS Conf. on Measurement and Modeling of Computer Systems*, 1994.

[3] S.-H. Chiang and M.K. Vernon, "Dynamic vs. Static Quantum-Based Parallel Processor Allocation", *Proc. Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP)*, May 1996, D. G. Feitelson and L. Rudolph (eds.), Springer Verlag, Lecture Notes in Computer Science Vol. 1162, pp. 200-223.

[4] W. Cirne and F. Berman, "A Model for Moldable Supercomputer Jobs", Proc. Int'l Parallel and Distributed Computing Symposium (IPDPS), April 2001.

[5] W. Cirne and F. Berman, "When the Herd is Smart: Aggregate Behavior in the Selection of Job Request", *IEEE Trans. on Par. and Distr. Systems*, 14(2), Feb. 2003.

[6] J. Corbalan, X. Mortarell, and J. Labarta, "Improving Gang Scheduling through Job Performance Analysis and Malleability", *Proc. ICS*, June 2001.

[7] A. Downey, "A Model for Speedup of Parallel Programs", Technical Report CSD-97-933, Univ. of California Berkeley, Jan. 1997.

[8] P.-F. Dutot and D. Trystram, "Scheduling on Hierarchical Clusters Using Malleable Tasks", *Proc. Symp. on Parallel Algorithms and Architectures (SPAA)*, July 2001.

[9] D.G. Feitelson, L. Rudolph, U. Schwiegelsohn, K.C.Sevcik, and W. Parsons, "Theory and Practice in Parallel Job Scheduling", *Proc. Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP)*, 1997, Springer Verlag, Lecture Notes in Computer Science, Vol. 1291.

[10] H. Franke, J. Jann, J.E. Moreira, P. Pattnik, and M.A. Jette, “An Evaluation of Parallel Job Scheduling for ASCI Blue-Pacific”, Proc. IEEE/ACM Supercomputing Conf. (SC), 1999.

[11] R.A. Gibbons Historical Application Profiler for Use by Parallel Schedulers. Proc. IPPS Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP), April 1997, Lecture Notes in Computer Science 1291, Springer Verlag.

[12] D. Ghosal, G. Serazzi, and S. K. Tripathi. The Processor Working Set and Its Use in Scheduling Multiprocessor Systems. *IEEE Trans. Software Engineering*, Vol. 17, No. 5, May 1991, pp. 443-453.

[13] U. Lublin and D.G. Feitelson, “The Workload on Parallel Supercomputers: Modeling the Characteristics of Rigid Jobs”, *J. of Parallel and Distributed Computing*, Nov. 2003, 63(11), pp. 1105-1122.

[14] C. McCann and J. Zahorjan, “Processor Allocation Policies for Message Passing Parallel Computers”, *Proc. SIGMETRICS Conf. Measurement & Modeling of Comput. Syst.*, May 1994, pp. 208-219.

[15] V. K. Naik, S. K. Setia, and M. S. Squillante, “Processor Allocation in Multiprogrammed Distributed-Memory Parallel Computer Systems”, *J. of Parallel and Distributed Computing*, 46(1), 1997, pp. 28-47.

[16] J.D. Padhye and L. Dowdy, “Dynamic Versus Adaptive Processor Allocation Policies for Message Passing Parallel Computers: An Empirical Comparison”, *Proc. Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP)*, 1996, Springer Verlag, Lecture Notes in Computer Science Vol. 1162, pp. 224-243.

[17] E. W. Parsons and K. C. Sevcik, “Implementing Multiprocessor Scheduling Disciplines”, *Proc. Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP)*, 1997, Springer Verlag, Lecture Notes in Computer Science.

[18] E. Rosti, E.Smirni, G. Serazzi, and L.W. Dowdy. “Analysis of Non-Work-Conserving Processor Partitioning Policies”,. *Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP)*, 1995.

[19] K.C. Sevcik. “Characterization of Parallelism in Applications and Their Use in Scheduling”, *Performance Evaluation Review* 17, 1989, pp. 171-180.

[20] A.C. Sodan and X. Huang. SCOJO—Share-Based Job Coscheduling with Integrated Dynamic Resource Directory in Support of Grid Scheduling. *Proc. Ann. Int. Symposium on High Performance Computing Systems (HPCS*), Sherbrooke, Canada, May 2003, pp. 213-221.

[21] A.C. Sodan and X. Huang, “Adaptive Time/Space Scheduling with SCOJO”, *Proc. HPCS,*Winnipeg,May 2004.

[22] A.C. Sodan and L. Han, “ATOP—Space and Time Adaptation for Parallel and Grid Applications via Flexible Data Partitioning”, *Proc. 3rd ACM/IFIP/USENIX Workshop on Reflective and Adaptive Middleware*, Toronto, Oct. 2004.

[23] A.C. Sodan, “Loosely Coordinated Coscheduling in the Context of Other Dynamic Approaches for Job Scheduling—A Survey”, *Concurrency & Computation: Practice & Experience*, 17(15), Dec. 2005, pp. 1725-1781.

[24] A.C. Sodan, C. Doshi, L. Barsanti, and D. Taylor, “Gang Scheduling and Adaptive Resource Allocation to Mitigate Advance Reservation Impact”, IEEE CCGrid, Singapore, May 2006.

[25] D. Tsafrir, Y. Etsion, and D.G. Feitelson, ``Modeling User Runtime Estimates". *11th Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP),* June 2005. Lecture Notes in Computer Science Vol. 3834, pp. 1-35.

[26] J. Turek, J.L Wolf, K.L. Pattipati, and P.S. Yu, “Scheduling Parallelizable Tasks: Putting it All on the Shelf”, *SIGMETRICS Performance Evaluation Review*: Proc. SIGMETRICS Conf. on Measurement and Modeling of Computer Systems, 20(1), June 1982.