# An Extended Evaluation of Two-Phase Scheduling Methods for Animation Rendering

Yunhong Zhou, Terence Kelly, Janet Wiener, and Eric Anderson

Hewlett-Packard Laboratories
1501 Page Mill Rd
Palo Alto, CA 94304
{yunhong.zhou, terence.p.kelly, janet.wiener, eric.anderson4}@hp.com

**Abstract.** Recently HP Labs engaged in a joint project with DreamWorks Animation to develop a Utility Rendering Service that was used to render part of the computer-animated feature film *Shrek 2*. In a companion paper [2] we formalized the problem of scheduling animation rendering jobs and demonstrated that the general problem is computationally intractable, as are severely restricted special cases. We presented a novel and efficient two-phase scheduling method and evaluated it both theoretically and via simulation using large and detailed traces collected in DreamWorks Animation's production environment.

In this paper we describe the overall experience of the joint project and greatly expand our empirical evaluations of job scheduling strategies for improving scheduling performance. Our new results include a workload characterization of *Shrek 2* animation rendering jobs. We furthermore present parameter sensitivity analyses based on simulations using randomly generated synthetic workloads. Whereas our previous theoretical results imply that worst-case performance can be far from optimal for certain workloads, our current empirical results demonstrate that our scheduling method achieves performance quite close to optimal for both real and synthetic workloads. We furthermore offer advice for tuning a parameter associated with our method. Finally, we report a surprising performance anomaly involving a workload parameter that our previous theoretical analysis identified as crucial to performance. Our results also shed light on performance tradeoffs surrounding task parallelization.

## 1 Introduction

The problem of scheduling computational jobs onto processors arises in numerous scientifically and commercially important contexts. In this paper we focus on an interesting subclass with three special properties: jobs consist of nonpreemptible tasks; tasks must execute in stages; and jobs yield completion rewards if they finish by a deadline. In previous work we have formalized this problem as the *disconnected staged scheduling problem* (DSSP), described its computational complexity, proposed a novel scheduling method, and evaluated our solution

theoretically and empirically using traces from an animation rendering application [2]. The companion paper also mentions a range of practical domains other than animation rendering in which DSSP arises.

This paper extends our previous work three ways. First, we provide a detailed description of the joint project between our company and DreamWorks Animation that first brought DSSP to our attention. We believe that this case study is interesting in its own right because it shows how modern parallel processing technologies are applied to a commercially important problem substantially different from traditional scientific applications of parallel computing. The joint project rendered parts of the animated feature film *Shrek 2* in a 1,000-CPU cluster on HP premises and thus illustrates the intersection of parallel processing and "utility computing." Our description of the project furthermore places scheduling and other parallel computing technologies in a broader context of business considerations. Finally, the project shows how new requirements driven by business needs led to a new and interesting formal problem, which in turn created research opportunities at the intersection of theoretical Computer Science, CS systems, and Operations Research.

Second, we provide a thorough and detailed workload characterization of traces we collected in the aforementioned cluster during the rendering of *Shrek 2*. These are the traces used in our previous empirical work, and in some of the extended empirical work presented in this paper.

Our third and major technical contribution in this paper is to greatly extend our empirical results and explore via simulation several open questions raised by our previous theoretical analysis. Whereas our previous empirical results were based exclusively on traces collected in DreamWorks Animation's production environment, in this paper we supplement the production traces with randomly-generated synthetic workloads. The latter transcend the domain-specific peculiarities of the former and thus allow us to evaluate the generality and robustness of our solution. Our new empirical work addresses three main issues:

1. We have proven theoretically that our solution architecture yields near-optimal results if jobs' critical paths are short; worst-case performance can be arbitrarily poor, however, if critical paths are long. How pessimistic are these theoretical results? Do real or random workloads lead to worst-case performance?
2. In some domains, including animation rendering, it is possible to shorten the critical paths of jobs by parallelizing tasks. What are the benefits of such parallelization? Is parallelization always beneficial?
3. Our solution architecture contains an adjustable parameter. Can we offer generic, domain-independent guidance on how to tune it?

In addition to addressing these issues we also report interesting and unanticipated relationships between problem parameters and performance.

The remainder of this paper is structured as follows: Section 2 defines DSSP and summarizes our previous results. Section 3 characterizes the DreamWorks Animation rendering workload, and Section 4 presents our extended empirical results. Section 5 describes the joint HP/DreamWorks Animation project that

motivated our interest in the DSSP, Section 6 reviews related work, and Section 7 concludes.

## 2 Background & Previous Work

In our previous work on DSSP [2] we formalized the abstract scheduling problem and described its computational complexity. We furthermore introduced a novel two-phase scheduling method better suited to the special features of DSSP than existing solutions. Finally, we evaluated our solution's performance through both theoretical analysis and trace-driven simulation using workload traces collected in a commercial production environment. This section reviews our previous work and describes open questions that we address in the present paper.

### 2.1 Problem

Informally, in DSSP we are given a set of independent computational *jobs*. Each job is labeled with a *completion reward* that accrues if and only if the job finishes by a given global deadline. Jobs consist of nonpreemptible computational *tasks*, and a job completes when all of its tasks have completed. Tasks must be performed in *stages*: no task in a later stage may start until all tasks in earlier stages have finished. Tasks within a stage may execute in parallel, but need not do so. We are also given a set of *processors*; at most one task may occupy a processor at a time. Our goal is to place tasks on processors to maximize the aggregate completion reward of jobs that complete by the global deadline.

Formally, we are given $J$ jobs, indexed $j \in 1 \ldots J$. Job $j$ contains $G_j$ stages, indexed $g \in 1 \ldots G_j$. The set of tasks in stage $g$ of job $j$ is denoted $S_{gj}$. Stages encode precedence constraints among tasks within a job: no task in stage $g+1$ may begin until all tasks in stage $g$ have completed; no precedence constraints exist among tasks in different jobs. The execution time (or "length") of task $i$ is denoted $L_i$. The total processing demand of job $j$, denoted $T_1(j)$, equals $\sum_{g=1}^{G_j} \sum_{i \in S_{gj}} L_i$. The *critical path length* (CPL) of a job is the amount of time that is required to complete the job on an unlimited number of processors; it is denoted $T_\infty(j) \equiv \sum_{g=1}^{G_j} \max_{i \in S_{gj}} \{L_i\}$. Figure 1 illustrates the stage and task structure of two jobs, and their critical path lengths.

There are $P$ identical processors. At most one task may occupy a processor at a time, and tasks may not be preempted, stopped/re-started, or migrated after they are placed on processors. Let $C_j$ denote the completion time of job $j$ in a schedule and let $R_j$ denote its completion reward. $D$ is the global deadline for all the jobs. Our goal is to schedule tasks onto processors to maximize the aggregate reward $R_\Sigma \equiv \sum_{j=1}^{J} U_D(C_j)$, where $U_D(C_j) = R_j$ if $C_j \leq D$ and $U_D(C_j) = 0$ otherwise. Our objective function is sometimes called "weighted unit penalty" [4].

The computational complexity of DSSP is formidable, even for very restricted special cases. In [2] we show that general DSSP is not merely NP-hard but also NP-hard to approximate within any polynomial factor, assuming that P $\neq$ NP.

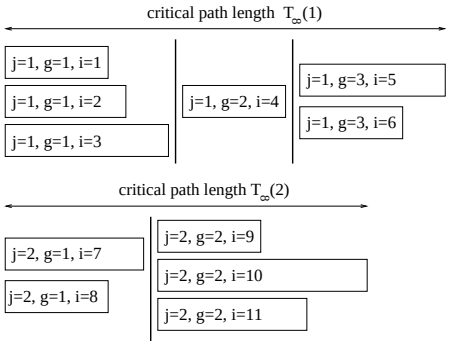


**Fig. 1.** Job ($j$), task ($i$), and stage ($g$) structure for two jobs.

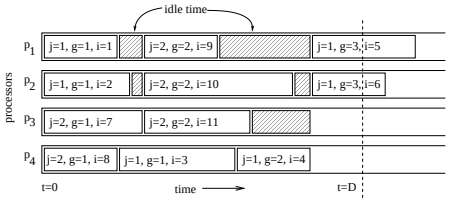


**Fig. 2.** A schedule for the jobs in Figure 1.

Even for the special case where jobs have unit rewards and tasks have unit execution times, it is *strongly NP-complete* to solve DSSP optimally.

### 2.2 Two-Phase Scheduling Method

Before reviewing our approach to DSSP, we motivate the need for a specialized solution by considering shortcomings of existing alternatives. Scheduling in modern production environments almost always relies on priority schedulers such as the commercial LSF product [18] or an open-source counterpart like Condor [5]. Priority schedulers by themselves are inadequate to properly address DSSP. The fundamental problem is that ordinal priorities are semantically too weak to express completion rewards. Ordinal priorities can express, e.g., that "job A is more important than job B." However sums and ratios of priorities are not meaningful and therefore they cannot express, e.g., "jobs B and C together are 30% more valuable than A alone." When submitted workload exceeds available computational capacity, a priority scheduler cannot make principled decisions. In our example, it cannot know whether to run A alone or B and C together if those are the only combinations that can complete by the deadline. Priority schedulers make performance-critical *job selection* decisions as accidental by-products of *task dispatching* decisions.

Our solution, by contrast, decomposes DSSP into two conceptually simple and computationally feasible phases. First, an offline job selection phase chooses a reward-maximizing subset of jobs to execute such that it expects all of these jobs to complete by the deadline. An online task dispatching phase then places tasks from the selected jobs onto processors to complete as many as possible by the deadline. Completion rewards guide the first phase but not the second. Task dispatching can achieve better performance precisely because it can safely ignore completion rewards and consider only the *computational* properties of jobs but not their business value.

Job selection chooses a subset of jobs with maximal aggregate completion reward such that their total processing demand does not exceed available capacity.

Let binary decision variable $x_j = 1$ if job $j$ is selected and $x_j = 0$ otherwise and let $P$ denote the number of processors. Our selection problem is the following integer program:

$$\text{Maximize} \quad \sum_{j=1}^{J} x_j R_j \tag{1}$$

$$\text{subject to} \quad \sum_{j=1}^{J} x_j T_1(j) \quad \leq \quad r \cdot PD \tag{2}$$

The summation in objective Equation 1 assumes that all selected jobs can be scheduled, regardless of their $T_\infty$; jobs with $T_\infty(j) > D$ have $U_D(C_j) = 0$ and may be discarded before job selection. $PD$ in the right-hand side of Equation 2 is the total amount of processor time available between $t = 0$ and $t = D$. By tuning selection parameter $r$ we may select a set of jobs whose total processor demand is less than or greater than the capacity that is actually available. The final schedule after task dispatching typically achieves less than 100% utilization because precedence constraints force idleness as shown in Figure 2. Intuitively, $r$ should therefore be set to slightly less than 1. Later we propose a way to compute good values of $r$.

The selection problem is a classic 0-1 knapsack problem, for which a wide range of solvers exist [11]. In this paper we solve the selection problem optimally using a simple knapsack algorithm, dynamic programming (DP) by profits. Our previous work explores a wider range of job selectors, including a sophisticated mixed-integer programming selector that can account for a wide range of side constraints [2].

Once a subset of jobs has been selected, a dispatcher places their tasks on processors. We employ a non-delay (or "work-conserving") dispatcher that places runnable jobs onto idle processors whenever one of each is available. The end result is a schedule that contains idle time due only to precedence constraints.

Given an idle processor and several runnable tasks, a *dispatcher policy* decides which task to run. In [2] we implemented and empirically evaluated over two dozen dispatcher policies. Our previous results show that our novel dispatcher policy LCPF outperforms a wide range of alternatives by several performance metrics. LCPF chooses a runnable task from the *job* whose critical path is longest. Intuitively, LCPF favors jobs at greatest risk of missing the deadline. To the best of our knowledge, LCPF represents the first attempt to tailor a dispatcher policy to the special features of DSSP, particularly its *disconnected* precedence constraint DAG. Our empirical results in this paper include two other dispatcher policies: STCPU chooses the runnable task whose parent job has the shortest total CPU time, and RANDOM chooses a runnable task at random. In the special case where each job contains exactly one task, LCPF coincides with the well-known *longest job first* policy, and STCPU reduces to *shortest job first*.

### 2.3 Previous Performance Evaluation

Our theoretical results on the computational complexity of DSSP show that this problem is hard to solve optimally, and it is hard even to approximate within a

polynomial factor if job completion rewards and task execution times are unrestricted. However, we have also shown in [2] that in the unweighted case (i.e., uniform job completion rewards) a two-phase solution method using a greedy (possibly sub-optimal) selector and *any* non-delay dispatcher can achieve near-optimal performance if the maximum critical path length of any input job, denoted $T_\infty^{\max}$, is small relative to the global deadline $D$. Here we re-state this result:

**Theorem 1.** *The two-phase scheduling method with the selection parameter* $r = 1 - (1 - 1/P)(T_\infty^{\max}/D)$ *and any non-delay dispatcher completes at least* $(1 - \frac{T_\infty^{\max}}{D}(1 - \frac{1}{P}))OPT - 1$ *jobs before the deadline, where OPT is the maximum number of jobs that can be completed by any algorithm.*

Theorem 1 implies that any two-phase solution (with a proper selection parameter $r$) completes at least half as many jobs as an optimal algorithm if $T_\infty^{\max} \leq D/2$. As $T_\infty^{\max}/D$ goes to 0, its performance approaches that of the optimal algorithm.

The bound of Theorem 1 can be attained by two-phase algorithms that require remarkably little information about the computational demands of tasks. Specifically, it is necessary to know only the aggregate processing demand of *subsets of tasks* during selection. Knowledge of individual task execution times is *not* required.

Our previous work includes empirical evaluations of selectors, dispatchers, and combinations of the two. Briefly, we find that for the DreamWorks Animation production scheduling traces that we used:

1. Dispatcher policies differ dramatically in terms of job completion time distributions and other performance measures; our LCPF policy outperforms alternatives by several measures.
2. LCPF is relatively insensitive to a poorly-tuned selection parameter $r$.
3. An optimal selector with a well-tuned $r$ and an LCPF dispatcher achieves 9%–32% higher aggregate value in the weighted case than a priority scheduler with no selection.

## 3 Workload Characterization

In this section, we describe a real production system where animation rendering jobs were run and then characterize the jobs and tasks in this workload.

### 3.1 Rendering Infrastructure

In early 2004, DreamWorks Animation began to supplement their in-house render farm with an extra cluster of 500 machines for production of the animated feature film *Shrek 2*. Each machine in this cluster is an HP ProLiant DL360 server with two 2.8-GHz Xeon processors, 4 GB of memory and two 36-GB 10k RPM SCSI disks. It contains a total of 1,000 CPUs and can serve up to 1,000

tasks simultaneously, because at most one task may occupy a processor at any time.

Our workload characterization is based on LSF [18] scheduler logs collected on this cluster during the period 15 February–10 April 2004. The logs associate tasks with their parent render jobs and we reconstructed their stage structure. We removed from consideration all jobs that did not complete successfully, e.g., because a user canceled them. Our final trace contains 56 nights, 2,388 jobs, and 280,011 tasks.

### 3.2 Jobs, Tasks, and Stages

| | Minimum | Maximum | Average | Std Dev | Median |
|---|---|---|---|---|---|
| Number of tasks/job | 3 | 1328 | 117 | 114 | 84 |
| Number of jobs/night | 5 | 79 | 43 | 18 | 41 |
| Number of tasks/night | 880 | 8908 | 5000 | 1970 | 4976 |
| Task length $L_i$ | 0 | 85686 | 8500 | 9899 | 5356 |
| Maximum tasks in stage | 1 | 700 | 93 | 93 | 91 |
| Percent of jobs rerun/night | 0% | 26.4% | 12.4% | | |
| $T_\infty$ in hours (all nights) | 0.06 | 24.07 | 4.35 | 3.46 | 3.42 |
| $T_\infty$ in hours (average/night) | 0.59 | 14.04 | 4.61 | 0.50 | |
| $T_1$ in hours (all nights) | 0.23 | 5026.36 | 276.77 | 420.08 | 134.72 |
| $T_1$ in hours (average/night) | 6.12 | 1849.48 | 301.22 | 396.49 | |

**Table 1.** Task and job statistics.

| Number of stages per job with: | 1 task | 2 tasks | 3 tasks | more than 3 tasks |
|---|---|---|---|---|
| Average (all jobs) | 2.24 | 1.07 | 0.21 | 1.62 |

**Table 2.** Most stages of jobs have 1-3 tasks. Jobs have one or two stages with many tasks.

Table 1 shows statistics about jobs, tasks, and stages in the eight week workload. While the number of tasks and jobs varied widely across all of the nights — there were a few weekend nights where very little rendering was done, as shown in Figure 3 — the medians shown in the fifth column represent the majority of nights. Most nights have a few tens of jobs and a few thousands of tasks. Task length varies widely; some tasks complete in less than a second while many tasks take hours. The median task length is 1.5 hours.

The middle section of Table 1 shows the percentage of jobs that were run twice during a single night. These jobs completed on the first run and produced

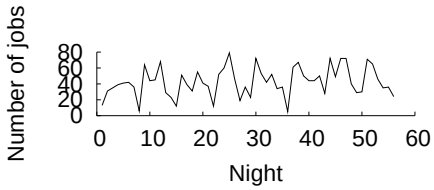


**Fig. 3.** Time series of the number of jobs each night. Low workloads correspond to weekends.

the correct number of frames, but the frames were not satisfactory for some reason and the job was resubmitted. Note that in order for a job to run twice in one night, $2 \cdot T_\infty(j)$ must fit in the 13 hour time window.

The number of stages per job is shown in Figure 4. Most jobs have 3–7 stages, depending on whether all of physical simulation, model baking, and frame rendering need to be done (with some gluing stages in between). However, only 1 or 2 of the stages have more than 3 tasks, as shown in Table 2. Those stages usually compute something per frame (e.g., render the frame), which is why Figure 5 shows that the maximum number of tasks in a single stage is strongly correlated with the number of frames being rendered. For 82% of the jobs, the maximum number of tasks in a stage *equals* the number of frames. Figure 6 shows that the maximum length of that single stage is nearly equal to $T_\infty$, i.e., in most cases a single stage accounts for most of a job's critical path length.

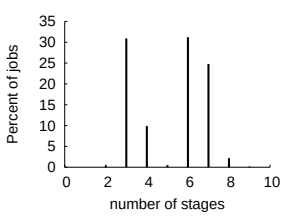


**Fig. 4.** Histogram of number of stages per job.

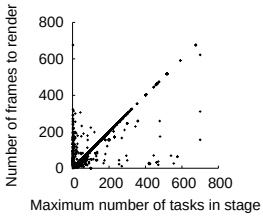


**Fig. 5.** The maximum number of tasks in a single stage is highly correlated with number of frames that the job renders.

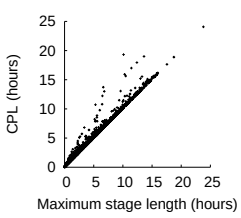


**Fig. 6.** The length of the longest single stage accounts for most of $T_\infty(j)$.

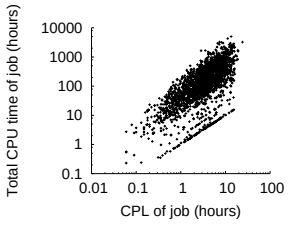


**Fig. 7.** Critical path lengths vs. total CPU times of jobs, logarithmic scales.

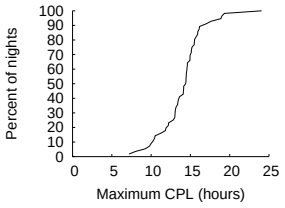


**Fig. 8.** CDF of $T_\infty^{\max}$ per night.

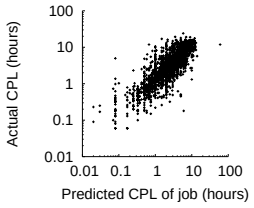


**Fig. 9.** Predicted and actual $T_\infty$ appear correlated on log scales but often differ by 30–100%.

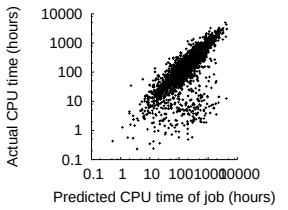


**Fig. 10.** Predicted and actual $T_1$ also appear correlated but often differ by a factor of 2–10.

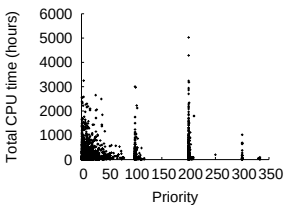


**Fig. 11.** Jobs' CPU demand are not correlated with priorities.

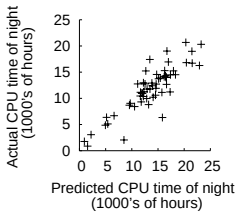


**Fig. 12.** Sums of predicted vs. actual CPU demand for all jobs in each night.

### 3.3 $T_\infty$ and $T_1$ Distributions for Jobs

Figure 7 shows the relationship between $T_\infty$ and $T_1$ for each job. $T_1$ is generally around 100 times as much as $T_\infty$, because the longest stage has a median of 91 tasks. The median $T_\infty$ of 3.4 hours shows that most jobs can complete in a short time, given enough resources. However, Figure 8 demonstrates that on most nights (75%), there is at least one job that cannot be completed within a 13 hour time window.

### 3.4 Predictions of $T_\infty$ and $T_1$

The previous section analyzed the actual $T_\infty$ and $T_1$ of jobs. In practice, the actual times are not known before the jobs execute. Scheduling decisions must be made based on predicted times. In this section, we compare the predictions that DreamWorks Animation artists supplied before a job was run with the actual run times.

Figure 9 shows a scatterplot of predicted versus actual $T_\infty$ for jobs in our trace. Predictions are based on an estimate of the time required to render a single frame from the job. While Figure 9 suggests a strong correlation between the predicted and actual times, only 14% of the predictions are within 10% of the actual $T_\infty$. Only 46% of the predictions are within 30% of the actual $T_\infty$, and 21% of the predictions are wrong by a factor of 2 or more.

Figure 10 shows a similar scatterplot for $T_1$. These predictions are formed by multiplying the estimated time to render a single frame by the number of frames in the job, which leaves even more room for error, since not all frames take the same amount of time to render. Only 37% of the predictions are within 20%, 25% are wrong by a factor of 2 or more, and 7% are wrong by a factor of 10 or more.

However, while $T_1$ predictions are not very accurate for individual jobs (and we have no predictions of individual task execution times), our predictions are quite good when aggregated over all jobs for a given night. 32% of the predictions are within 10% and 80% are within 30% of the actual sums. Only 4% of the predictions are wrong by a factor of 2 or more. Figure 12 shows the sums of the predicted and actual $T_1$ for all jobs in each night, relative to each other. These sums are the *only* quantity for which we have good predictions. Fortunately, in job selection, we only need the sum of $T_1$ for a set of jobs rather than the individual jobs' requirements; that sum indicates whether the set can fit in the knapsack capacity of $PD$.

### 3.5 Job Priorities

| Priority | 0-99 | 100-199 | 200-299 | 300-399 |
|---|---|---|---|---|
| Meaning | Must do first | Must do tonight | Good to do | If there is time |
| Percent of jobs (all nights) | 55.2% | 8.1% | 35.7% | 1.0% |
| Number of jobs (all nights) | 1318 | 193 | 853 | 24 |
| Number of jobs (average/night) | 24 | 4 | 15 | 0.4 |

**Table 3.** Number and percent of jobs in each priority band

Currently, DreamWorks Animation uses job priorities to decide which jobs to run first. Table 3 shows the priority categories that they use and the percentage of jobs assigned to each category. In Figure 11, we compare the CPU demand of jobs in different priority categories. Unsurprisingly, we find little correlation because job priorities are intended to reflect the relative importance of jobs, not their computational demands.

## 4 Sensitivity Analysis

In this section, we evaluate how sensitive the performance of our scheduling method is to two variables: a parameter of our method and a property of our workload. We describe these variables first and then present the questions that the rest of this section tries to answer.

The scheduling method presented in Section 2.2 is a two-phase method. The first phase, job selection, uses a selection parameter $r$ to decide the subset of

jobs to run. As $r$ approaches 1, the number of chosen jobs increases to fill all of the CPU time. The lower the value of $r$, the more idle time is allowed in the job schedule but the lower the possible reward if all jobs complete.

The workload itself contains many jobs, each of which has a critical path length $T_\infty$, and the workload has a maximum CPL $T_\infty^{\max}$. Theorem 1 shows that our two-phase scheduling method achieves close-to-optimal performance if $T_\infty^{\max}/D$ is small and jobs have unit rewards. It leaves open the case where $T_\infty^{\max}/D$ is large, as it is in DreamWorks Animation's workload, and jobs have non-unit rewards. In addition, while Theorem 1 gives tight worst-case bounds for pathological examples, we wanted to explore average case performance. Our evaluation therefore aims to answer the following questions:

1. How should the selection parameter $r$ be set? Does it depend on the dispatching policy used?
2. How sensitive is dispatcher performance to different maximum critical path lengths?
3. What happens to performance as $T_\infty^{\max} \approx D$? Is it as bad as the worst case given by Theorem 1?
4. Can we improve performance by breaking long tasks into small pieces (thereby shortening $T_\infty^{\max}$)?

In order to answer the latter three questions, we needed to generate workloads with varying values of $T_\infty^{\max}$. We therefore decided to generate synthetic workloads. We first describe our synthetic workload generation and then present our results.

### 4.1 Synthetic Workload Generation

In our standard synthetic workloads, $T_\infty^{\max}/D \approx 1$. We then transform these workloads to create new workloads with lower values of $T_\infty^{\max}$. We first describe how we generate a standard synthetic workload.

For all experiments in the next few subsections, the number of CPUs $P = 100$ and the global deadline $D = 13$ hours $= 4680 \times 10$ seconds. For job completion rewards, we use the size-dependent reward function $R_j = T_1(j)$. For each standard workload, we add jobs to the workload until the total CPU demand of the workload exceeds $2 \times P \times D$.

While the workload is not full, we create new jobs as follows: For each new job, we draw a random number of stages from $U[5, 10]$ where $U[a, b]$ denotes an integer drawn with uniform probability from the set $\{a, a + 1, \ldots, b\}$. For each stage, we draw a number of tasks from $U[1, 10]$. For each task, we draw a task length from $U[1, 600]$. After choosing task lengths for every task in a job, we then compute the job's critical path length $T_\infty$. If $T_\infty \leq D$, we add the job to the workload; otherwise, we discard it.

To transform an already generated workload into a new one with a desired $T_\infty^{\max}$, we alter the number of stages that it has. A job with fewer stages will probably have a shorter $T_\infty$, since it will have fewer dependencies between tasks.

We therefore also generated workloads with a fixed number of stages. For each such workload, we first create a standard workload where the number of stages in each job is always 10, and the task lengths are drawn from $U[1, 600]$. We then alter the workload so that each job has a smaller (fixed) number of stages, say 6. For each job, we reassign all tasks that were previously in a stage $> 6$ to a stage drawn from $U[1, 6]$. The new workload therefore has the same number of jobs and the same total processing time as the original workload and, unlike the previous transformation, the same number of tasks. We call this transformation T1.

A different way to alter the $T_\infty$ of a job is as follows: While the job's $T_\infty$ exceeds the new $T_\infty^{\max}$, we find the task with the longest length and replace it with two tasks that are half as long (in the same stage as the original task). The CPU time of the job (and of each stage of each job) therefore remains constant, but the job now has more parallizable tasks. We call this transformation T2.

### 4.2 Scheduling Performance when $T_\infty^{\max} \approx D$

While Theorem 1 shows that the two-phase scheduling method achieves near-optimal performance if $(T_\infty^{\max}/D)$ is small and jobs have unit rewards, it does not apply to the more general case where $T_\infty^{\max}/D$ is large and jobs have non-unit completion rewards. Furthermore we do not know how pessimistic the bound of Theorem 1 is when critical paths are long: We know that the bound is tight because we can construct pathological inputs that result in worst-case performance given by the theorem. However our theoretical results alone shed little light on whether such inputs are likely to arise in practice. How does the worst-case bound of Theorem 1 compare with average performance for workloads with $T_\infty^{\max} \approx D$?

In this section we answer this question by generating synthetic workloads with $T_\infty^{\max} \approx D$ and applying our two-phase scheduling method to them in simulation. We use DP by profits to solve the job selection problem optimally. Ideally, we would like to compare the performance achieved by our scheduler with the optimal solution value. However, as discussed in Section 2 and proven in [2], it is computationally infeasible to solve DSSP optimally. We therefore instead compute an *upper bound* equal to the aggregate value of jobs selected when selection parameter $r = 1$. This is the reward that would accrue if we select jobs to utilize available processor capacity as fully as possible, *and* all selected jobs complete by the deadline. It is clearly an upper bound for the optimal value of the overall scheduling problem (which may be lower if some jobs fail to complete on time). When evaluating our two-phase scheduler we use the term *performance ratio* to denote the ratio of its actual performance to the upper bound discussed above.

Section 4.1 describes how to generate synthetic workloads with $T_\infty^{\max}$ close to $D$. For each workload generated in this way, we run the two-phase scheduling method with varying values of selection parameter $r$ during job selection. In this experiment we use three dispatcher policies during the task dispatching phase: LCPF, STCPU, and RANDOM, and we vary the selection parameter $r$ from 0.7

to 1.0 in increments of 0.01. For each ($r$, dispatcher) pair we generate 20 different random workloads and report mean performance ratio. Figure 13 presents our results.

The figure shows that LCPF clearly dominates both STCPU and RANDOM for any fixed $r$ value. This is not surprising because LCPF takes into account the critical paths of jobs while the other policies do not. For workloads with long critical paths, performance will suffer if the jobs with the longest critical paths are started too late to complete on time. Both STCPU and RANDOM ignore jobs' critical path lengths and therefore start many long jobs too late to complete by the deadline, while LCPF starts long jobs as early as possible.

Comparing STCPU with RANDOM, we see that STCPU is slightly better when $r \geq 0.92$ and RANDOM is slightly better when $r \leq 0.87$; their performance is similar when $r$ is in the range [0.87, 0.92]. Note also that RANDOM becomes less effective when more jobs are selected whereas STCPU is relatively insensitive to the tuning of $r$. This is intuitive because RANDOM treats all *tasks* equally, and if too many *jobs* are selected it will spread available processor capacity among them roughly evenly, with the result that many fail to complete by the deadline. By contrast, STCPU imposes a near-total priority order on jobs, because it is rare for two jobs to have the same total processing demand. If $r$ increases and more jobs are selected, some of the additional jobs have small processing demand and STCPU finishes them early during task dispatching. This has relatively little impact on the remaining processing capacity available to larger jobs, and the overall result is that many jobs still finish by the deadline.

Figure 13 shows that the performance of LCPF reaches a maximum of roughly 0.84 when $r = 0.87$. In other words, LCPF achieves at least 84% of the optimal performance even though the maximum critical path lengths $T_\infty^{\max}$ in the workloads used are at least 97% of the deadline $D$. This result stands in stark contrast to the very weak performance bound that Theorem 1 would guarantee with similarly long critical paths: If $T_\infty^{\max}/D = 0.97$, our previous theoretical results state that a two-phase scheduler can achieve as little as 3% of optimal performance in the unit-reward case.

Finally, we observe in Figure 13 that LCPF's performance is relatively flat when $r$ is in the range [0.85, 1]. When $r < 0.85$, all selected jobs finish by the deadline and selecting more jobs simply increases the number that finish. As the selection parameter increases beyond $r > 0.85$, performance does not improve because the additional jobs selected do not complete by the deadline; more processing capacity is used, but it is wasted on jobs that do not complete quickly enough to yield a reward. Because a job that fails to complete by the deadline simply wastes any capacity devoted to it, it might be best in practice to set $r$ to a relatively low value rather than a higher value that achieves comparable reward (e.g., 0.85 as opposed to 1 in the figure). Our results suggest that for LCPF and the workload studied ($T_\infty^{\max} \approx D$) a value of $r$ in the range [0.85, 9] is a good choice.

### 4.3 Performance vs. MaxCPL

In this section we empirically evaluate the performance of the two-phase scheduling method as the maximum critical path length $T_\infty^{\max}$ of the workload varies. Intuitively, jobs with relatively long critical paths face higher risk of finishing after deadline $D$ and thus yielding no reward. Theorem 1 shows that the worst-case performance of two-phase scheduling for unit-reward DSSP is a strictly decreasing function of $(T_\infty^{\max}/D)$. However our theoretical worst-case bounds for unit-reward DSSP do not necessarily predict *typical* performance in the weighted case. We therefore explore via simulation the relationship between maximum critical path length $T_\infty^{\max}$ and the performance ratio that our scheduling method achieves.

We describe two classes of simulation experiments to address this issue and compare their results qualitatively. Our first approach uses the workloads with a fixed number of stages and transform them into new workloads with fewer number of stages using transformation T1. While it directly alters only the number of stages per job, Figure 14 shows that the number of stages is closely related to $T_\infty^{\max}$ for the workload. $T_\infty^{\max}$ is almost a linear function of the number of stages for jobs in the synthetic workload.

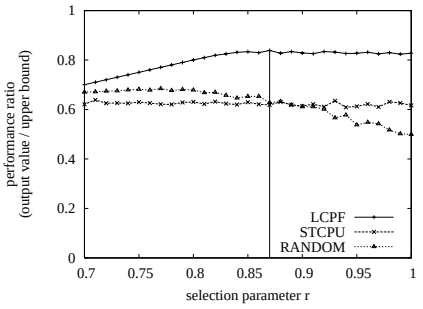


**Fig. 13.** Performance ratio vs. selection parameter $r$, for synthetic workloads with $T_\infty^{\max}/D \approx 1$.

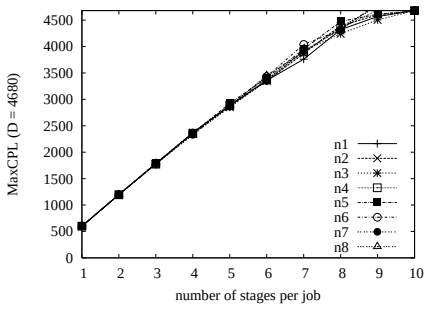


**Fig. 14.** MaxCPL vs. number of stages for each of eight synthetic workloads generated by transformation T1.

Figures 15, 16, and 17 show the performance ratio of three dispatching policies as the number of stages per job varies. One feature of these figures is that RANDOM differs qualitatively from LCPF and STCPU: The latter two policies dominate RANDOM for most values of $r$ and for most numbers of stages. Furthermore, for all $r$ values except $r = 1.0$, the performance of RANDOM decreases slowly as the number of stages (and $T_\infty^{\max}$) increases, but suffers only slightly. Finally, the performance of RANDOM relies heavily on a well-tuned $r$. If $r$ is close to 1 its performance degrades substantially. For the workload studied, $r$ of 0.8 to 0.9 seems appropriate for RANDOM and yields far better performance than $r \approx 1$.

Figures 16 and 17 show that LCPF outperforms STCPU by a wide margin when the number of stages per job is large. Furthermore, contrary to our intuition that performance should decrease monotonically as the number of stages increases, we notice a local minimum of the performance ratios of both LCPF and STCPU at roughly 5 stages/job. Upon further investigation of Figure 14 we found that 5 stages corresponds to $T_\infty^{\max} \approx D/2$. We conjecture that LCPF and STCPU exhibit non-monotonicity while RANDOM does not for the following reason: Whereas RANDOM disperses processor capacity across tasks from *all* jobs, LCPF and STCPU take a different approach. They impose an order on jobs and try to finish some jobs (the ones with greatest CPL and smallest total CPU demand, respectively) before devoting any processing capacity to others. If $T_\infty^{\max} \approx 0.5 \times D$, STCPU and LCPF will process a set of jobs and finish them slightly after $t = D/2$. By the time they finish these jobs, there may not be time to start the remaining jobs and finish them by the deadline. The remaining jobs are started late and narrowly miss the deadline, thus contributing no value and wasting processing resources.

Our second set of experiments uses the workloads that altered $T_\infty^{\max}$ directly using transformation T2. We again present results for three dispatching policies: Figure 19 for LCPF, Figure 20 for STCPU, and Figure 18 for RANDOM.

The results for this set of experiments are similar to those for the first set: STCPU and LCPF resemble each other but not RANDOM; LCPF outperforms STCPU when MaxCPL is long; and RANDOM is more sensitive to selection parameter $r$ than the other two. In both experiments, the results show minima for MaxCPL slightly larger than $D/2$. Furthermore these results shed additional light on the relationship between performance ratio and MaxCPL: Figures 19 and 20 show *second* local minima at around $T_\infty^{\max} = 0.35 \times D$, that is, at $T_\infty^{\max} \approx D/3$. It is possible for the performance of our scheduling method to have multiple local minima with respect to $T_\infty^{\max}$, and these local minima occur at approximately $D/n$ from the right side, where $n = 2, 3, \ldots$ is an integer. We conjecture that LCPF and STCPU finish jobs in "rounds." If $T_\infty^{\max} \approx D/n$ these policies will finish the first $n-1$ rounds; jobs in the last round will narrowly miss the deadline, thereby wasting processor resources without contributing value.

### 4.4 Selection Parameter Tuning

DSSP is a deadline scheduling problem where a job's completion reward is zero if it completes after the global deadline $D$. If too many jobs are selected during the job selection phase, then during task dispatching these selected jobs will compete for limited processor capacity and each job has a higher risk of finishing too late. It is thus important to set the selection parameter $r$ properly for each dispatching policy; in most cases it should be strictly less than 1. Let $r_0 \equiv 1-(1-1/P)(T_\infty^{\max}/D)$. Theorem 1 has proved that a general two-phase solution with $r = r_0$ completes at least $r_0 \cdot \mathrm{OPT} - 1$ jobs before the deadline, where OPT is the maximum number of jobs that can be completed by any scheduler. This seems to suggest a default selection parameter value. This section tries to find a reasonably good value for $r$ for various dispatching policies.

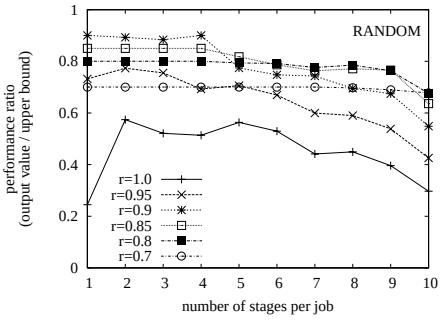


**Fig. 15.** Performance ratio vs. number of stages with dispatching policy RANDOM.

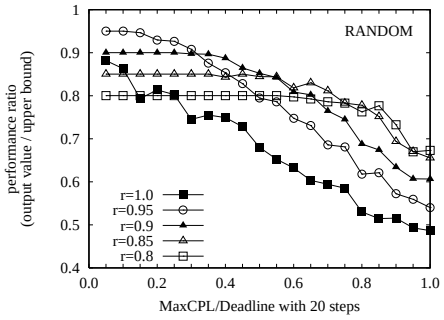


**Fig. 18.** Performance ratio vs MaxCPL with dispatching policy RANDOM.

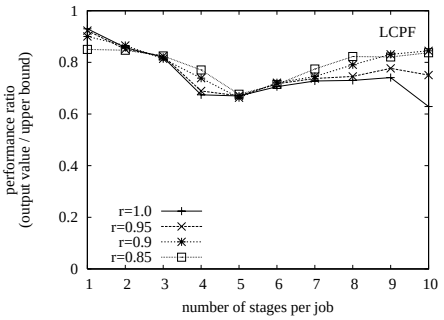


**Fig. 16.** Performance ratio vs. number of stages with dispatching policy LCPF.

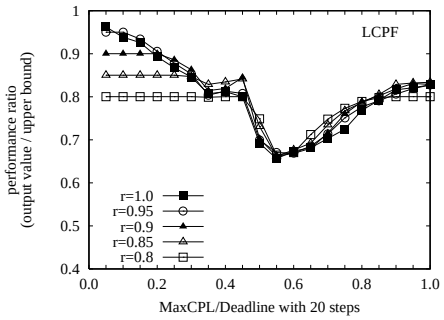


**Fig. 19.** Performance ratio vs. MaxCPL with dispatching policy LCPF.

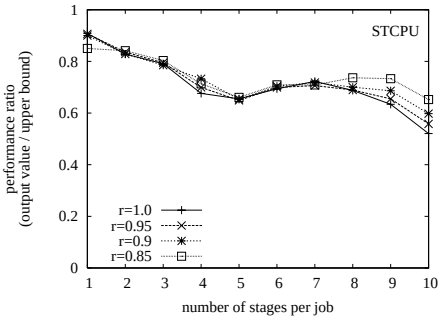


**Fig. 17.** Performance ratio vs number of stages with dispatching policy STCPU.

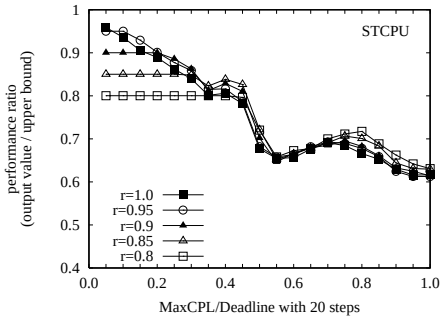


**Fig. 20.** Performance ratio vs. MaxCPL with dispatching policy STCPU.

We begin by briefly reviewing our simulation results from Section 4.3. Figure 15 shows that for dispatcher policy RANDOM, the *worst* strategy is to choose $r \geq 1$. Figure 15 suggests that the best choice of $r$ is in the range $[0.85, 0.9]$. For any $r \in [0.85, 0.9]$, performance is strictly better than for $r = 1$.

For dispatching polices such as LCPF or STCPU, it is also true that $r$ slightly less than 1 is better than $r = 1$. However the best selection parameter is no longer a fixed constant. Given two fixed selection parameters $r_1 < r_2$, we consider their corresponding performance ratio curves which we denote $f_1, f_2$ respectively. It is highly likely that when $T_\infty^{\max}$ is small, then $f_1 > f_2$. At some value ${T_\infty^{\max}}_0$ these two curves intersect. After that $T_\infty^{\max} > {T_\infty^{\max}}_0$, then the order is reversed $f_1 < f_2$.

Figures 19 and 20 suggest that the best value of $r$ is correlated to the maximum critical path length of the workload. Inspired by the value of $r_0$ above, we suspect that it is possible to find a constant $\lambda \in [0, 1]$, such that $r = 1 - \lambda(1 - 1/P)(T_\infty^{\max}/D)$ is a good choice for the selection parameter. Figure 21 shows the performance ratio of our algorithm using LCPF dispatching policy with three $\lambda$ values: $\lambda = 0.0$, 0.2, and 1.0. $\lambda = 0.0$ corresponds to the selection parameter $r$= 1.0; $\lambda = 1.0$ corresponds to the selection parameter $r = r_0$; and $\lambda = 0.2$ corresponds to $r = 1 - 0.2(1 - 1/P)(T_\infty^{\max}/D)$.

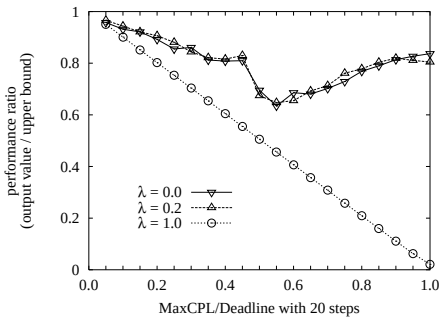


**Fig. 21.** Performance ratio vs. MaxCPL with LCPF and $r = 1 - \lambda(1 - 1/P)(T_\infty^{\max}/D)$, for three $\lambda$ values.

Figure 21 shows that for dispatching policy LCPF, the curve corresponding to $\lambda = 0.2$ consistently performs better than the curve corresponding to $\lambda = 0.0$, i.e., $r = 1.0$. Furthermore, both $\lambda = 0.2$ and $\lambda = 0.0$ perform much better than $\lambda = 1.0$. This is because $r_0$ is too conservative; it is appropriate only for worst-case inputs. For the class of workloads studied and for our LCPF dispatcher policy, a good selection parameter is around $r = 1 - 0.2(1 - 1/P)(T_\infty^{\max}/D)$.

### 4.5 Tradeoff Between Parallelism and Performance

Parallelism is an important technique in scientific computing and parallel computing to improve system utilization and job throughput by breaking long se-

quential tasks into multiple parallel tasks. For a multi-processor system, if there is only one long-running task, then only one processor is fully occupied and all other processors are left idle. Breaking long tasks into multiple short tasks will definitely increase system utilization. Remarkably, however, for DSSP parallelization is not necessarily helpful even if it entails no overhead. In other words, if we have an opportunity to parallelize tasks in a particular instance of DSSP and thereby reduce critical path lengths, it is *not* always advantageous to do so.

Consider, for instance, the workloads with $T_\infty^{\max}/D \approx 0.8$ in Figures 19 and 20. If by parallelizing tasks we reduce $T_\infty^{\max}/D$ to roughly 0.55, performance will be *worse* for both LCPF and STCPU even though parallelization does not increase total processing demand. If LCPF is the task dispatching policy, then parallelism is beneficial only if the given workload has $T_\infty^{\max} < D/2$. Finally, if $T_\infty^{\max}/D$ is small, then the two-phase scheduling method produces a solution with near-optimal performance regardless of the dispatcher policy used; even if parallelism incurs no extra cost, it cannot yield large performance improvements.

## 5 DreamWorks Animation Engagement Experience

In August 2003, HP Labs embarked on a challenge to provide remote rendering services to DreamWorks Animation. In early February 2004, we went into full production for the movie *Shrek 2*. We learned that the utilification [22] process of bringing the customer's workload up on a remote facility was not straightforward, and had many unexpected challenges.

We went through four stages in utilification. First a feasibility stage where we determined whether or not a remote service was feasible. Second, an instantiation stage where we brought up the service. Third, a confidence-building stage where we demonstrated to the customer's satisfaction that the remote service could correctly support their workload. Fourth, an ongoing maintenance stage where we optimize our delivery of the service, and keep the service up to date for the customer needs. In the following we describe these stages in detail.

### 5.1 Feasibility Stage

Our evaluation started in August 2003. We needed to determine whether we would be able to place a remote facility about 20 miles away from the primary site. We found that there were four feasibility questions:

1. Schedule - the proposed schedule said that we needed to be in production in only five months, by January 2004. Could we acquire all of the equipment and implement the system in that time?
2. Bandwidth and latency - will the 1 Gbit/s network connection between the sites be sufficient to support a cluster of 1,000 2.8 GHz CPUs?
3. Software and configuration - will we be able to install, configure, and adapt the existing software and configuration to work with a remote cluster?
4. Business Model - can the lawyers agree on an appropriate contract for the project?

While we expected (3) to provide the most difficulty, we in fact discovered that (4) presented the most substantial difficulties, in particular because the *Shrek 2* franchise had a large estimated monetary value, and DreamWorks Animation therefore had an understandable concern about exposing content outside of their company. Hence, we negotiated protections such as sanitization of gathered data for analysis, an isolated network to protect their content, a double-stage firewall to protect access from the HP network, and a camera to monitor the physical installation.

Performing the bandwidth and latency analysis was the hardest technical challenge we faced during the feasibility stage. The sustained and burst packet rates we needed to handle normally would require specialized network analyzer hardware, but we needed general purpose, full-packet capture for weeks of data. We developed a solution based on commodity hardware that used improved software for packet capture, buffering to local memory to handle bursts of data, parallel use of multiple disks spread across multiple trays, and opportunistic compression of data to increase the effective disk space and increase the time-periods for contiguous captures. The solution could handle traffic for hours at 30-50MB/s and bursts above 100MB/s with negligible drops on the tracing machine.

Our second challenge was to analyze the data. We have collected billions of NFS requests and replies, so putting this in a database or directly processing the raw traces would be either too slow or too expensive. Luckily we had previously developed a new, highly efficient trace storage format called "DataSeries" for handling block I/O traces and process traces. The format was general, and provided streaming access to database-like tables. We therefore developed a converter from the raw tcpdump traces into DataSeries, and built our analysis on the converted files. This has provided us with a flexible, extensible data format and structure for our analysis. Multiple people have been able to add new analysis in to our existing structure within a few days.

We did not hit the original schedule because of the length of time it took to negotiate the legal agreements and the unexpected length of the confidence-building stage. Both of these parts were originally scheduled as taking at most a few weeks, and in fact both took months. Luckily, the need for the service was not excessively strong until early February 2004, so we were able to provide the service on an appropriate schedule, just one different than we expected.

### 5.2 Instantiation Stage

The primary difficulty that occurred during the instantiation stage was installing and configuring all 500 machines. When our racks of machines arrived, we discovered that some of them were configured both physically and logically wrong. The physical mistakes involved cabling errors, which were straightforward if tedious to repair. The logical mistakes were more difficult because they involved incorrect firmware versions. After a little study, we developed a tool for automatically updating the firmware and firmware configuration on all of our machines automatically. We wrote an extensive document about the problems with the order fulfillment process for rack systems which was presented back to the HP

order fulfillment team. The key lesson was that many traditional tools that are used involve per-machine human effort. While those tools are acceptable if you have 1-10 machines, they become unusable at 500. We needed to automate many of these actions, and found moreover that it was important to write idempotent tools: any time we ran a task across 500 machines (even one as simple as removing a file), a few machines would fail to execute the task correctly. Our solution was to design our automation to take a machine to a particular state and report on changes it made, which meant that we could simply execute global tasks multiple times until we received a report of no changes.

### 5.3 Testing Stage

Once we had instantiated the rendering service, we then had to verify that it worked to the satisfaction of various people responsible for making the movie. Moreover, we wanted to perform these read-write tests with no risk to the production data. Our problem therefore was to clone an appropriate subset of the total 15-20 TB of data such that we could show that our cluster rendered frames correctly, and so that there would be minimal changes from testing to production.

While we solved this problem by using a DreamWorks Animation specific feature in their rendering system of having a few variables to change expected file locations, and setting the source file systems to read only, we found a better solution by having a write-redirector that snapshot the backend file systems to isolate us from underlying changes and store our writes in a second location. This solution enables us to test and verify our solution much faster. Then moving into production would merely have required removing the write redirector to send all of the accesses directly at the file systems.

### 5.4 Maintenance and Optimization

We moved into production in two stages, first on a movie that was not due to release until 2005, and then on *Shrek 2*. When we moved into production we identified some reporting and job submission issues that we had not addressed during the instantiation phase that we needed to solve. Once we entered into full production, our goal was really to optimize and maintain the cluster. We made many small, but cumulative improvements to our service: simple service-specific host monitoring to detect failed hosts, automatic job retry for failed jobs, farm usage analysis and reporting, tracing through render job executions, and NFS performance analysis. Specifically, we found an interesting aspect to explore scheduling improvements over this multi-processor rendering environment, and that motivated our current work.

## 6 Related Work

Scheduling is a basic research problem in both computer science and operations research. The space of problems is vast; [4, 17] provide good reviews. In this

section we focus on non-preemptive multiprocessor scheduling without processor-sharing.

### 6.1 Minimizing Makespan or Mean Completion Time

Much scheduling research focuses on minimizing makespan for tasks with arbitrary precedence constraints. Variants of list scheduling heuristics and their associated dispatching policies are the main focus of both theoretical and empirical studies [8, 3, 9, 1]. See Kwok and Ahmad [14] for a recent survey of static scheduling algorithms and Sgall [19] for online scheduling algorithms.

Another important optimization metric for job scheduling research is to minimize mean task completion time. The classic *shortest job first* (SJF) heuristic is optimal in the offline case with no precedence constraints and each job consists of a single task; SJF works well in many online scheduling systems.

The large *queueing-theoretic* literature on processor scheduling typically assumes continuous online job arrivals and emphasizes mean response times and fairness, e.g., Wierman and Harchol-Balter [21]. Kumar and Shorey analyze mean response time for stochastic "fork-join" jobs, where fork-join jobs closely resemble the stage-structured jobs of DSSP [13]. Our work on DSSP differs because we are confronted with a fixed set of jobs rather than a continuous arrival process. *Deadline scheduling* is therefore a more appropriate goal for DSSP.

### 6.2 Grid and Resource Management

For heterogeneous distributed systems such as the Grid, job scheduling is a major component of resource management. See Feitelson *et al.* [7] for an overview of theory and practice in this space, and Krallmann *et al.* [12] for a general framework for the design and evaluation of scheduling algorithms. Most work in this space empirically evaluates scheduling heuristics, such as backfilling [15], adaptive scheduling [10], and task grouping [20], to improve system utilization and throughput. Markov [16] described a two-stage scheduling strategy for Sun's Grid Engine that superficially resembles our two-phase decomposition approach. In fact there is no similarity: The first stage of Markov's approach assigns static priorities to jobs and the second stage assigns dynamic priorities to server resources. Most of the work in the Grid space does not emphasize precedence constraints among jobs/tasks.

### 6.3 Commercial Products

Open-source schedulers such as Condor manage resources, monitor jobs, and enforce precedence constraints [6]. Commercial products such as LSF additionally enforce fair-share constraints [18]. These priority schedulers have no explicit selection phase, so they must handle overload and enforce fair-share constraints through dispatching decisions. Our two-phase deadline scheduler for DSSP can employ a priority scheduler for task dispatching after an optimal solver has selected jobs. Selection can enforce a wide range of constraints, thereby allowing

greater latitude for dispatching decisions. Furthermore, the completion rewards of our framework are more expressive than ordinal priorities and thus better suited to DSSP.

## 7 Concluding Remarks

In this paper we evaluated the two-phase scheduling method for DSSP through parameter tuning and sensitivity analysis. Contrary to our intuition that the performance of our scheduling method should decrease as the maximum critical path length of the input workload increases, our empirical results show that even though there is a close correlation between performance ratio and MaxCPL value, it is *not monotonic* for dispatching polices such as LCPF. More exploration is needed to determine why the performance ratio decreases significantly when $T_\infty^{\max} \approx D/2$. We tentatively conjecture that when deadline is an integral multiple of MaxCPL, dispatcher policies such as LCPF that associate task priorities with job properties suffer because many of the jobs narrowly fail to complete by the deadline, thus achieving no reward and wasting processor resources.

Furthermore, contrary to the worst-case performance bound in our previous work which is pessimistically bad if $T_\infty^{\max} \approx D$, our new empirical evaluation shows that our algorithm performs very well for this special case, with a performance ratio of more than 80%.

Based on these empirical evaluation results, we believe that MaxCPL alone is insufficient to describe the workload and predict the performance of our scheduling methods. It will be interesting to explore the *distribution* of critical path lengths for all the jobs in the workload and determine its impact on the performance of the two-phase scheduling method.

## References


1. Thomas L. Adam, K. M. Chandy, and J. R. Dickson. A comparison of list schedules for parallel processing systems. *Communications of the ACM*, 17(12):685–690, December 1974.
2. Eric Anderson, Dirk Beyer, Kamalika Chaudhuri, Terence Kelly, Norman Salazar, Cipriano Santos, Ram Swaminathan, Robert Tarjan, Janet Wiener, and Yunhong Zhou. Value-maximizing deadline scheduling and its application to animation rendering. In *Proceedings of 17th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, Las Vegas, NV, July 2005. ACM press. A 2-page poster also appears in SIGMETRICS 2005.
3. Richard P. Brent. The parallel evaluation of general arithmetic expressions. *JACM*, 21(2):201–206, 1974.
4. Peter Brucker. *Scheduling Algorithms*. Springer, 3rd edition, 2001.
5. The Condor Project. `http://www.cs.wisc.edu/condor/`.
6. Directed acyclic graph manager (DAGMan) for Condor scheduler. `http://www.cs.wisc.edu/condor/dagman/`.
7. Dror G. Feitelson, Larry Rudolph, Uwe Schwiegelshohn, Kenneth C. Sevcik, and Parkson Wong. Theory and practice in parallel job scheduling. In *Proceedings of JSSPP, LNCS 1291*, pages 1–34, 1997.

8. Ron Graham. Bounds for certain multiprocessor anomalies. *Bell Sys Tech J*, 45:1563–1581, 1966.
9. Ronald Graham. Bounds on multiprocessing time anomalies. *SIAM J Appl Math*, 17:263–269, 1969.
10. Elisa Heymann, Miquel A. Senar, Emilio Luque, and Miron Livny. Adaptive scheduling for master-worker applications on the computational grid. In Mark Baker Rajkumar Buyya, editor, *Proceedings of the First IEEE/ACM International Workshop on Grid Computing (GRID 2000), LNCS 1971*, pages 214–227. Springer, 2000.
11. Hans Kellerer, Ulrich Pferschy, and David Pisinger. *Knapsack Problems*. Springer, 2004.
12. Jochen Krallmann, Uwe Schwiegelshohn, and Ramin Yahyapour. On the design and evaluation of job scheduling algorithms. In *Proceedings of JSSPP, LNCS 1659*, pages 17–42, 1999.
13. Anurag Kumar and Rajeev Shorey. Performance analysis and scheduling of stochastic fork-join jobs in a multicomputer system. *IEEE Trans Par Dist Sys*, 4(10), October 1993.
14. Yu-Kwong Kwok and Ishfaq Ahmad. Static scheduling algorithms for allocating directed task graphs to multiprocessors. *ACM Computing Surveys*, 31(4):406–471, December 1999.
15. David A. Lifka. The ANL/IBM SP scheduling system. In *Proceedings of JSSPP, LNCS 949*, pages 295–303, 1995.
16. Lev Markov. Two stage optimization of job scheduling and assignment in heterogeneous compute farms. In *Proc. IEEE Workshop on Future Trends in Distributed Computing Systems*, pages 119–124, Suzhou, China, May 2004.
17. Michael Pinedo. *Scheduling: Theory, Algorithms, and Systems*. Prentice-Hall, 2nd edition, 2002.
18. Platform Computing. LSF Scheduler. `http://www.platform.com/products/LSFfamily/`.
19. Jiri Sgall. On-line scheduling—a survey. In A. Fiat and G.J. Woeginger, editors, *Online Algorithms: The State of the Art*, number 1442 in LNCS, pages 196–231. Springer-Verlag, 1998.
20. Ling Tan and Zahir Tari. Dynamic task assignment in server farms: Better performance by task grouping. In *Proc. of the Int. Symposium on Computers and Communications (ISCC)*, pages 175–180, July 2002.
21. Adam Wierman and Mor Harchol-Balter. Classifying scheduling policies with respect to unfairness in an M/GI/1. In *SIGMETRICS*, pages 238–249, June 2003.
22. John Wilkes, Jeffrey Mogul, and Jaap Suermondt. Utilification. In *Proceedings of the 11th ACM SIGOPS European Workshop*, 2004.