Value-Maximizing Deadline Scheduling and its Application to Animation Rendering[*]

Eric Anderson, Dirk Beyer, Kamalika Chaudhuri, Terence Kelly, Norman Salazar,
Cipriano Santos, Ram Swaminathan, Robert Tarjan, Janet Wiener, Yunhong Zhou
Hewlett-Packard Laboratories, 1501 Page Mill Rd, Palo Alto, CA 94304
firstname.lastname@hp.com

ABSTRACT

We describe a new class of utility-maximization scheduling problem with precedence constraints, the *disconnected staged scheduling problem* (DSSP). DSSP is a nonpreemptive multiprocessor deadline scheduling problem that arises in several commercially-important applications, including animation rendering, protein analysis, and seismic signal processing. DSSP differs from most previously-studied deadline scheduling problems because the graph of precedence constraints among tasks within jobs is *disconnected*, with one component per job. Another difference is that in practice we often lack accurate estimates of task execution times, and so purely offline solutions are not possible. However we do know the set of jobs and their precedence constraints up front and therefore some offline planning is possible.

Our solution decomposes DSSP into an offline job selection phase followed by an online task dispatching phase. We model the former as a knapsack problem and explore several solutions to it, describe a new dispatching algorithm for the latter, and compare both with existing methods. Our theoretical results show that while DSSP is NP-hard and inapproximable in general, our two-phase scheduling method guarantees a good performance bound for many special cases. Our empirical results include an evaluation of scheduling algorithms on a real animation-rendering workload; we present a characterization of this workload in a companion paper. The workload records eight weeks of activity on a 1,000-CPU cluster used to render portions of the full-length animated feature film *Shrek 2* in 2004. We show that our improved scheduling algorithms can substantially increase the aggregate value of completed jobs compared to existing practices. Our new task dispatching algorithm LCPF performs well by several metrics, including job completion times as well as the aggregate value of completed jobs.

Categories and Subject Descriptors

F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems—*Sequencing and Scheduling*

General Terms

Algorithms, Theory, Performance

Keywords

Deadline Scheduling, Animation Rendering, Simulation, Multiprocessor Job Scheduling

1. INTRODUCTION

We describe a new class of scheduling problem with precedence constraints, the *disconnected staged scheduling problem* (DSSP). DSSP is a nonpreemptive multiprocessor deadline scheduling problem; we seek to maximize the aggregate value of jobs that complete by a specified deadline. It arises in many commercially-important applications, including protein sequence matching [6], certain classes of fast Fourier transform computations [45, 46], seismic signal processing workloads [22, 21], and distributed data processing in Google [14].

Our interest in DSSP began with the practical problem of scheduling computer animation rendering jobs for commercial entertainment. Each job represents a brief excerpt from a film and consists of several stages that must be processed in order (e.g., physical simulation, model baking, frame rendering, and film clip assembly). Each stage in turn consists of computational tasks that may be run in parallel; all tasks in a stage must finish before any task in the next stage can start. A job completes if and only if all of its tasks complete; otherwise, no film clip is produced. Precedence constraints exist among tasks within a job, but not among tasks in different jobs. The graph of precedence constraints is therefore *disconnected*, with one component per job. Jobs run overnight and yield value only if they complete before the artists who submitted them return the following morning. Side constraints in addition to precedence constraints may be present, e.g., fair-share constraints may limit daily or weekly CPU consumption by different teams of artists. Demand frequently exceeds available CPU capacity, making it impossible to complete all submitted jobs by the deadline. The set of jobs is known in advance but their computational demands (e.g., the run times of tasks) are not precisely known. Pure offline solutions are therefore not possible, but existing online solutions do not exploit the knowledge that is available in advance.

Existing scheduling practices rely on priority schedulers, which by themselves are not well suited to DSSP due to the semantic

[*] A 2-page poster will appear in SIGMETRICS'05 in Banff, Canada.

SPAA'05, July 18–20, 2005, Las Vegas, Nevada, USA.

weakness of ordinal priorities. Priorities can express the relative importance of jobs, e.g., "job A is more important than B, and B is more important than C." Sums and ratios of ordinal priorities are not meaningful, however, so they cannot express "B and C together are 30% more valuable than A." Poor decisions can result when demand exceeds capacity: Given two equal-priority jobs and sufficient capacity to finish only one of them, a priority scheduler may dispatch tasks from both jobs onto processors and fail to complete either job by the deadline. Further difficulties can arise if fair-share constraints are enforced during dispatching by "instantaneous fair-share" mechanisms [40]. Such mechanisms can preclude value-maximizing dispatching decisions by enforcing the constraint on fine time scales when it need only be enforced on longer time scales.

The fundamental problem in both cases (overload and side constraints) is that priority schedulers make *job selection* decisions as by-products of *task dispatching* decisions. Dispatching decisions in turn rely on ordinal priorities whose semantics cannot express essential features of DSSP. Our approach is to assign to jobs *completion rewards* whose sums and ratios are meaningful, and to perform job selection and task dispatching separately.

This paper presents both theoretical and empirical results. We first formalize DSSP as an optimization problem in which we seek to maximize aggregate reward. We show that this problem is NP-hard even in the offline case where all task run times are known, and even for restricted variants. We therefore propose a tractable two-phase solution. The first phase, job selection, chooses a set of jobs that maximizes aggregate reward, satisfies side constraints, and is likely to finish on time. The second phase, task dispatching, can therefore concentrate exclusively on finishing jobs by the deadline. For job selection, we explore methods ranging from simple greedy heuristics to a sophisticated mixed integer programming (MIP) formulation. For task dispatching, we consider a wide range of dispatcher policies including a novel policy based on the critical path lengths of jobs.

Our theoretical results show that our two-phase solution framework is an approximation algorithm whose performance depends on the critical path lengths of jobs. Our empirical results are based on an eight-week trace of 2,388 jobs and 280,011 tasks collected in a 1,000-CPU production system that rendered part of the DreamWorks feature film *Shrek 2* in 2004. Trace-driven simulations show that task dispatching algorithms differ markedly in terms of aggregate reward and several secondary desiderata; our new dispatching algorithm excels by all measures. Both our theoretical and empirical results show that, remarkably, fine-grained knowledge of individual task execution times is unnecessary if we have coarse-grained estimates of the aggregate computational demand of jobs.

The rest of this paper is organized as follows. We describe DSSP formally and analyze its computational complexity in Section 2. We describe our two-phase scheduling method in Section 3 and analyze its worst-case performance in Section 4. We present our simulation results in Section 5, review related work in Section 6, and conclude in Section 7.

2. PROBLEM STATEMENT

Formally, DSSP consists of J jobs, indexed $j \in 1 \dots J$. Job j contains G_j stages, indexed $g \in 1 \dots G_j$. 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; stages represent a special case of "series-parallel" precedence constraints [8, 38]. No precedence constraints exist among tasks in different jobs, i.e., the directed acyclic graph (DAG) of task precedence constraints is *disconnected*, with one component per job.

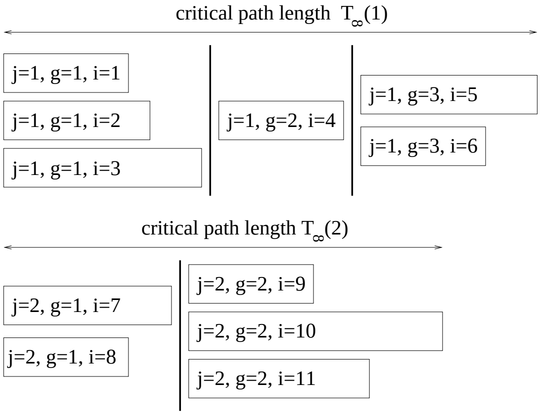

Figure 1: Job, task, and stage structure for two jobs. *j* = job, *g* = stage, *i* = task. *j*,*i* are unique.

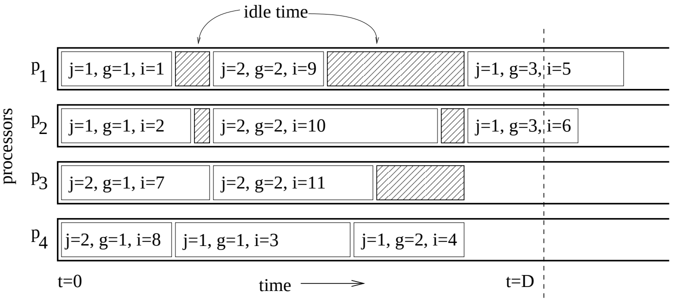

Figure 2: A schedule for the jobs in Figure 1.

The execution time (or "length") of task i is denoted L_i. The total processing demand of job j, denoted $T_1(j)$, is the sum of task lengths over all tasks in the job, $T_1(j) \equiv \sum_{g=1}^{G_j} \sum_{i \in S_{gj}} L_i$. $T_1(j)$ is the minimum time required to complete the job j if it runs on a single processor. By contrast, the *critical path length* 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/restarted, or migrated after they are placed on processors. A job completes when all of its tasks have completed, and we receive a reward if the job completes by global deadline D. Let C_j denote the completion time of job j in a schedule. Let R_j denote its completion reward. Our goal is to dispatch tasks onto processors in such a way that the final schedule maximizes 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. This objective function is sometimes called "weighted unit penalty" [8]. Note that jobs with $T_\infty(j) > D$ have $U_D(C_j) = 0$. The definition of $U_D(C_j)$ can be extended to allow rewards between zero and R_j for jobs that complete after the deadline; we have considered this generalization but do not present results in this paper due to space limitations.

Finally, we are *not* required to complete all of the jobs; indeed, an optimal schedule may discard one or more jobs. Figure 2 illustrates a final schedule containing the two jobs from Figure 1; job 2 completes before the deadline but job 1 does not.

2.1 Computational Complexity

In this section we present computational complexity results for two DSSP variants: 1) the *general* case, in which completion rewards R_j are arbitrary and tasks have arbitrary execution times; and 2) the *unweighted unit execution time* case in which $R_j = 1$ for all jobs j and $L_i = 1$ for all tasks.

Our first result shows that general DSSP is not merely NP-hard but also NP-hard to approximate within any polynomial factor, assuming that $\mathrm{P} \neq \mathrm{NP}$.

THEOREM 1. *General* DSSP *is NP-hard to approximate within any polynomial factor.*

PROOF. Let R_Σ^{OPT} denote the aggregate reward obtained by an optimal algorithm. If Theorem 1 is false, then there exists a polynomial time algorithm $\mathcal{A}$ with output A, such that $R_\Sigma^{OPT} \leq a_d A^d + \cdots + a_1 A + a_0$, where d is the degree of the polynomial and $a_d, \ldots, a_0$ are constant coefficients. Consider the special case where there are only two processors and we have a single job ($J = 1$) with a single stage and completion reward $R > a_0$. Both the optimal algorithm and $\mathcal{A}$ will output either R or 0. If $R_\Sigma^{OPT} = R$ and $A = 0$, then $R \leq a_0$, a contradiction. Thus if the optimal algorithm can schedule the job, $\mathcal{A}$ can also schedule it. However, for the special case where the total processing time for the job equals $2D$, this problem is equivalent to the classic *partition problem*, which is known to be NP-complete [27]. Thus, it is NP-hard to approximate general DSSP within a polynomial factor. □

We further prove that even the special case of DSSP with unit rewards and unit execution times is *strongly NP-complete*. A problem is strongly NP-complete if it remains NP-complete when all input coefficients are polynomially bounded. If a problem is strongly NP-complete, no pseudo-polynomial time algorithm can solve it unless $\mathrm{P} = \mathrm{NP}$ [17].

THEOREM 2. *Unweighted* DSSP *with unit task execution time is strongly NP-complete.*

PROOF. The proof is through a simple reduction from *3D-Matching*, one of the original problems proven by Karp to be NP-complete in the strong sense [27]. Suppose we are given a 3D-Matching instance where $M \subseteq X \times Y \times Z$, with $|X| = |Y| = |Z| = J$. For each triple $(x_a, y_b, z_c) \in M$, we construct a job with $3J$ stages. Each stage has only one task, except for stages $a, J+b$, and $2J+c$, each of which has two tasks. All the tasks have unit execution time. Let $D = 3J$. Let the number of machines be $J+1$. Each job takes $3(J+1)$ time in total, thus the scheduler can satisfy at most J jobs. If J jobs are satisfied, then it corresponds to a solution for 3D-Matching. □

Theorems 1 and 2 show that DSSP is hard to solve, in particular it is strongly NP-complete even with unit job rewards and unit task lengths. Moreover, with arbitrary rewards and task lengths, even polynomial-factor approximation of the optimal solution is NP-hard. Section 3 introduces a two-phase scheduling method of DSSP, and Section 4 analyzes its worst-case performance. Our analysis assumes that jobs have unit completion rewards but permits arbitrary task execution times. We show that performance depends on the ratio of jobs' critical path lengths to the global deadline; relatively short critical paths yield near-optimal performance. We also establish an alternative performance bound that depends on the number of processors rather than on critical path lengths.

3. TWO-PHASE SCHEDULING

Our approach decomposes DSSP into two tractable phases, an offline job selection phase followed by an online task dispatching phase. We present algorithms for both phases. The job selection phase chooses a subset of jobs to execute. The task dispatching phase places tasks from the selected jobs onto processors.

Our two-phase approach has several advantages. First, each phase requires only estimates of $T_1(j)$ and $T_\infty(j)$ for each job j. We do *not* require task execution times L_i. In practical applications, including animation rendering at DreamWorks, estimates of T_1 and T_∞ for jobs are more readily available than individual task execution times. Second, job selection can be expressed as a straightforward problem for which efficient optimal solvers exist. Side constraints (e.g., fair share) can be addressed during the job selection phase, and therefore they do not interfere with task dispatching decisions in the second phase.

3.1 Phase 1: Job Selection

The goal of job selection is to select 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. P is the number of processors. Our selection problem is the following integer program:

$$\text{Maximize} \quad \textstyle\sum_{j=1}^{J} x_j R_j \qquad (1)$$

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

The summation in objective Equation 1 assumes that all selected jobs can be scheduled, regardless of their T_∞; 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 $t = D$. The selection parameter r allows us to select a set of jobs whose total processor demand is less than or greater than the total 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 be set to slightly less than 1. r is a tunable parameter; Theorem 4 suggests an exact formula for r.

This selection problem is a classic 0-1 knapsack problem, one of the NP-hard combinatorial optimization problems solvable in practice. It admits both pseudo-polynomial-time exact solutions and fully-polynomial-time approximation schemes; a wide range of solvers exist [28]. We implemented three kinds of solvers.

1. The very simple classic greedy heuristic works well for many DSSP instances.
2. "Dynamic programming (DP) by profits" is one of the simplest optimal knapsack algorithms.
3. A solver based on commercial mixed integer programming (MIP) software also yields optimal solutions and handles more constraints than a knapsack solver.

We discuss these approaches below in terms of their solution quality and resource demands. Because all three approaches yield solutions in under one second for our DSSP instances, we do not present timing results.

The greedy knapsack algorithm considers jobs in nonincreasing order of $R : T_1$ ratio, selecting jobs as long as doing so does not violate the capacity constraint of Equation 2. This algorithm can yield selections that are arbitrarily far from optimal, but a slight extension can guarantee a solution that is 1/2 as good as optimal; further extension improves the solution to 3/4 of optimal [28, p. 34]. The asymptotic time and memory requirements of greedy selection are dominated by sorting jobs. DP by profits is only slightly more sophisticated than the extended greedy algorithm, but it guarantees an optimal solution. Our straightforward implementation requires $O(J\sum_{j=1}^{J} R_j)$ time and memory; more complex implementations can reduce both requirements. DP by profits can also be converted to a fully-polynomial-time approximation scheme simply by scaling down completion rewards [28, p. 41]. A limitation of DP by profits and other simple knapsack solvers is that they cannot easily address elaborate side constraints.

General integer programming produces the same optimal solutions as DP by profits for the basic knapsack problem, but can also handle a wide range of side constraints. A MIP selector is also easy to modify when requirements change, as often occurs in production environments. Our MIP solver is implemented in GAMS/CPLEX [24] and is intended for production use. It addresses several complex side constraints specific to animation rendering that we do not discuss here. General-purpose MIP solvers employ algorithms with exponential worst-case asymptotic time and memory complexity, but in practice they often perform quite well.

3.2 Phase 2: Task Dispatching

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 them is available. The end result is a schedule that contains idle time due only to precedence constraints. For nonpreemptive scheduling problems like DSSP, such a schedule is *not* in general optimal [38]. For online DSSP, however, task lengths L_i are not known in advance and therefore we cannot compute a final schedule offline. We must use a runtime dispatcher of some kind, and non-delay dispatchers are simple and perform well in practice.

Given an idle processor and several runnable tasks, a *dispatcher policy* chooses one of the tasks. We implemented and empirically evaluated over two dozen dispatcher policies. In this paper we restrict attention to a handful of the best performers.

RANDOM: choose a runnable task at random

FIRST: visit *jobs* in job-ID order and choose the first runnable task encountered

PRIORITY: choose a runnable task from the highest priority *job*

STCPU (shortest total CPU time): choose a runnable task from the *job* with minimal $T_1(j)$

LCPF (longest critical path first): choose a runnable task from the *job* with maximal $T_\infty(j)$

CPA (critical path algorithm): choose the runnable *task* with the greatest weight as defined below

CPA (also known as HLFET [32, 1]) computes for each task a weight equal to the length of the longest path *from that task* to an exit node in the DAG of task precedence constraints. It then chooses tasks whose weight is greatest. Note that all of the above policies except RANDOM and CPA first choose a *job* and then choose a runnable task from it. We find that policies like STCPU and LCPF that take into account the properties of jobs usually outperform policies that consider task properties alone. Finally, we evaluated variants such as "choose maximal $T_1(j)$" and "choose minimal $T_\infty(j)$"; these do not perform well.

LCPF is a new dispatcher policy that takes advantage of the disconnected precedence DAG of DSSP. In the special case where each job contains exactly one task, LCPF is identical to the well-known *longest job first* (LJF) policy. However, when jobs contain multiple tasks, LJF is no longer well defined. To the best of our knowledge, LCPF has not been described or evaluated previously.

Note that CPA uses properties of *tasks* to make dispatching decisions while LCPF uses critical path length, which is a property of *jobs*. To build intuition for the difference, consider the two jobs of Figure 1. Given these jobs and a single processor, LCPF will choose tasks from job 1 until it is finished before choosing any task from job 2. By contrast, CPA will alternate between tasks from both jobs, and both jobs will finish at roughly the same time. Another important practical difference is that CPA requires estimates of task execution times L_i for *each task*, whereas LCPF requires only an estimate of critical path length $T_\infty(j)$ for *each job*. In practice, e.g. at DreamWorks, the latter are more readily available.

4. ANALYSIS

This section presents worst-case performance analysis of our two-phase scheduling methods for DSSP with variable task execution times and unit rewards. Our first method, MAXK, is an offline algorithm that requires knowing task execution times L_i for task dispatching simulation. Our second method uses a greedy selection algorithm with *any* dispatching policy, including those that do not require task execution times. We show that the second method achieves the *same* approximation bound as MAXK, even though it requires far less information and is more widely applicable. The theoretical results of this section shed further light on what characteristics make DSSP difficult.

Our solutions can be applied to fully general DSSP instances, but our approximation bounds apply only to unit rewards; Theorem 1 shows that approximation is hard in the general case. Throughout this section we assume that $R_j = 1$ for all jobs j.

4.1 The MAXK Approximation Algorithm

MAXK is an offline algorithm that computes the maximum value K such that the K jobs with the highest $R : T_1$ ratio can be completed by the deadline. Given a value K, MAXK simply simulates dispatching all tasks of the selected jobs to check whether all of them complete by the deadline; linear or binary search can be used to find the maximal value of K. The value of K may depend on the dispatcher policy used; it is possible to evaluate several policies and choose the one that maximizes K. Regardless of the dispatcher policy used, the simulation must use task lengths L_i to create a schedule that takes precedence constraints into account.

We now prove that MAXK computes near-optimal schedules for unweighted DSSP. Our approximation bounds depend on the maximum critical path length over all jobs, denoted $T_\infty^{\max}$. They are stated in terms of the aggregate reward obtained by the optimal algorithm, i.e., the number of jobs completed, denoted OPT.

THEOREM 3. *Algorithm* MAXK *can schedule at least* $\max\left\{\left(1-\frac{T_\infty^{\max}}{D}\left(1-\frac{1}{P}\right)\right)OPT-1, OPT-(P-1)\right\}$ *jobs.*

PROOF. Suppose that MAXK selects k jobs. Then for the first $k+1$ jobs, a final non-delay schedule results in some tasks finishing after deadline D. We consider a schedule that results from dispatching all of the tasks from the first $(k+1)$ jobs, and bound its total idle time.

We construct a precedence chain of tasks $\tau_p \prec \tau_{p-1} \prec \cdots \prec \tau_1$ from the schedule with the following property: whenever there is an idle processor, another processor is executing one of the tasks in this chain. This technique applies to an arbitrary task dependency DAG [18]. We show how to construct such a precedence chain below.

Let τ_1 denote a task which exceeds the deadline D, and let $[t_1, t_1']$ denote the time interval for this task to run. Now look at the first time unit $[t, t+1]$ with $t+1 \le t_1$ when any of the processors is idle.

If there is no such value t, $\{\tau_1\}$ is the chain of tasks required and we are done. If such a t exists, then at time slot t, τ_1 is not runnable. This implies that some other task τ_2 which τ_1 depends on finishes execution after time t. Let $[t_2, t_2']$ denote the execution time interval for task τ_2, then $t_2' \geq t+1$. By the construction of t, there is no idle processor between $[t+1, t_1]$. Because $t+1 \leq t_2'$, there is no idle processor between $[t_2', t_1]$. Similarly, look at the first idle time for any of the processors before τ_2 starts execution. This way, we can construct a task precedence chain $\tau_p \prec \tau_{p-1} \prec \cdots \prec \tau_1$. Let $[t_i, t_i']$ denote the execution time interval for task τ_i. Then there is no idle space between $[0, t_p]$, and there is no idle space between $[t_{i+1}', t_i]$, for any index $1 \leq i < p$.

Let L denote the sum of task lengths for all the tasks in the above task dependency chain. Then the total idle space in the time interval $[0, D]$ is bounded by $(P-1)L$. Because $L \leq T_\infty^{\max}$, therefore

$$\sum_{j=1}^{k+1} T_1(j) > PD - (P-1)T_\infty^{\max}. \tag{3}$$

Because jobs have unit completion rewards in the unweighted case, MAXK sorts jobs in order of increasing total processing-time demand, i.e., $T_1(1) \leq T_1(2) \leq \cdots \leq T_1(J)$. Equation 3, together with the fact that $T_1(j)$'s are sorted ascending, implies that

$$\sum_{j=1}^{(k+1)\cdot\lambda} T_1(j) > PD, \quad \text{for} \quad \lambda = \frac{PD}{PD - (P-1)T_\infty^{\max}}.$$

It is clear that $\mathrm{OPT} < (k+1)\cdot\lambda$. This implies that

$$k > \frac{1}{\lambda}\mathrm{OPT} - 1 = \left(1 - \frac{T_\infty^{\max}}{D}\left(1 - \frac{1}{P}\right)\right)\mathrm{OPT} - 1. \tag{4}$$

Recall that the total idle space in the time interval $[0, D]$ is bounded by $(P-1)L$, where L is the sum of task lengths for all the tasks in the task dependency chain constructed above. Because the jobs are sorted by increasing T_1, $T_1(k+1) \geq L$. If the algorithm selects $(P-1)$ extra jobs, then the total processing time will exceed PD. In other words, if we select the first $k+1+(P-1)$ jobs, their total processing time will exceed the total system capacity PD. The optimal algorithm selects a set of jobs with total capacity not exceeding PD, thus we have

$$\mathrm{OPT} < k+1+(P-1), \quad \text{i.e.} \quad k \geq \mathrm{OPT} - (P-1). \tag{5}$$

Theorem 3 now follows by combining Equations 4 and 5 together. □

The theorem implies that MAXK is close to optimal if $T_\infty^{\max}$ is relatively small compared to D. It also establishes another performance bound, which is independent of maximal critical path length, and it shows that MAXK gives a good approximate solution to offline unweighted DSSP if the number of processors is small relative to the number of requests that can be completed by an optimal algorithm. This is likely to be true if average job processing demand $\sum_j T_1(j)/J$ is small relative to global deadline D.

4.2 General Two-Phase Solutions

Our next result shows that two-phase solutions that do *not* require individual task lengths also guarantee good results in the unweighted case if $T_\infty^{\max}$ is relatively small compared to D. In fact, ignorance of task lengths does not worsen our scheduling solution in the worst case if we have $T_1(j)$.

THEOREM 4. *The two-phase scheduling method using a greedy knapsack selector with* $r = 1 - (1 - 1/P)(T_\infty^{\max}/D)$ *and* any *non-delay dispatcher completes* $(1 - \frac{T_\infty^{\max}}{D}(1 - \frac{1}{P}))OPT - 1$ *jobs before the deadline. If all jobs have critical path less than* $D/2$, *then the algorithm can schedule at least* $\frac{1}{2}OPT - 1$ *jobs.*

PROOF. Suppose that the selection algorithm chooses the first k jobs. Then we have the following:

$$\sum_{j=1}^{k} T_1(j) \leq rPD = PD - (P-1)T_\infty^{\max} \tag{6}$$

$$\sum_{j=1}^{k+1} T_1(j) > rPD = PD - (P-1)T_\infty^{\max} \tag{7}$$

We claim that the algorithm is able to schedule the first k jobs successfully using any non-delay scheduling policy. Otherwise, there exists an instance where a non-delay scheduling policy results in a final schedule for the first k jobs such that the time to finish all the tasks exceeds D. Consider the schedule for this dispatching instance. As in the proof of Theorem 3, we can argue that the total idle space for the time interval $[0, D]$ is bounded by $(P-1)T_\infty^{\max}$. Therefore,

$$\sum_{j=1}^{k} T_1(j) > PD - (P-1)T_\infty^{\max}. \tag{8}$$

Equations 6 and 8 contradict each other. Therefore, the algorithm is guaranteed to schedule the first k jobs.

Because jobs have unit completion rewards in the unweighted case, the greedy knapsack selector also sorts jobs in order of increasing total processing-time demand, i.e., $T_1(1) \leq T_1(2) \leq \cdots \leq T_1(J)$. Equation 7 is identical to Equation 3 in the proof of Theorem 3. Thus an identical argument shows that

$$k > \left(1 - \frac{T_\infty^{\max}}{D}(1 - \frac{1}{P})\right)\mathrm{OPT} - 1 > \left(1 - \frac{T_\infty^{\max}}{D}\right)\mathrm{OPT} - 1.$$

Our algorithm schedules k jobs, and it is bounded by the above inequality. If $T_\infty^{\max} \leq D/2$, $(1 - T_\infty^{\max}/D) \geq 1/2$, then $k > \frac{1}{2}\mathrm{OPT} - 1$. □

Theorem 4 implies that any two-phase solution (with a proper selection parameter r) completes at least half as many jobs as an optimal algorithm if $T_\infty^{\max} \leq D/2$. If $T_\infty^{\max} = D/10$, then it achieves at least 90% of the optimal value. As $T_\infty^{\max}/D$ goes to 0, its performance approaches that of the optimal algorithm. The lower bounds of Theorem 4 are tight, in the sense that we can construct pathological examples to achieve them. Note that this result does not assume any particular dispatcher policy, i.e., we have shown that when critical paths are short it is *impossible* for a non-delay dispatcher to perform poorly.

The bound of Theorem 4 depends on maximum critical path length $T_\infty^{\max}$ for all the jobs; it is very weak when $T_\infty^{\max}/D$ is close to 1. Critical paths in real workloads can be long (see [47], workload characterization), so we must evaluate two-phase schedulers empirically in order to understand performance in practice. The experimental results of Section 5 show that when $T_\infty^{\max}$ is high our theoretical bounds are pessimistic, that our two-phase approach yields good schedules in practice, that dispatcher policies differ dramatically in performance, and that our method outperforms existing practices substantially.

5. EXPERIMENTAL RESULTS

In this section, we evaluate both job selection and task dispatching empirically. We use traces derived from LSF scheduler logs from a cluster of 1,000 CPUs to drive our simulation experiments.

These logs were collected from the cluster as it was used by DreamWorks to render part of the feature film *Shrek 2* between 15 February and 10 April 2004. Each of the 500 machines in the 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. 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 cancelled them. Our final trace contains 56 nights, 2,388 jobs, and 280,011 tasks. The jobs have a median CPL T_∞ of 3.4 hours, although on most (75%) of nights, there is at least one job that cannot be completed within a 13 hour time window. A more complete characterization of this workload is in our companion paper [47].

In Section 4, we proved that differences among dispatcher policies are small when completion rewards are identical for all jobs and critical paths are short relative to the global deadline. In our animation workload, however, completion rewards vary and critical paths can be long. We explore these cases here. This section addresses five questions.

1. Is a greedy knapsack selector adequate?
2. Do dispatcher policies differ in performance?
3. How well do selector/dispatcher pairs optimize our objective function?
4. Does selection parameter r require tuning?
5. How does $T_\infty^{\max}$ affect dispatcher performance?

All of our experiments use two parameters from the DreamWorks cluster where our traces were collected: $P = 1,000$ processors and deadline $D = 13$ hours, because the nightly rendering shift typically runs from 8 p.m. to 9 a.m. We first discuss how we assign completion rewards to jobs so that we can compare the aggregate reward of different solutions.

5.1 Reward Assignment

Our traces contain only ordinal priorities, and so we must assign R_j based on job priorities or other job properties. We report results for three reward functions: 1) linear: $R_j \equiv 500 - \text{priority}_j$; 2) size-dependent: $R_j \equiv T_1(j)$; and 3) banded:

$$R_j \equiv \begin{cases} 100,000 & \text{if priority} < 100 \\ 1,000 & \text{if priority} < 200 \\ 10 & \text{if priority} < 300 \\ 1 & \text{otherwise} \end{cases}$$

Linear rewards essentially use the priority as the reward (DreamWorks uses low priority values to indicate greater importance). Banded rewards group jobs into crude importance categories, e.g., "must do," "good to do," and "if there's time." Size-dependent rewards mean that value is directly proportional to processor time spent.

5.2 Selection

The naïve classic greedy knapsack algorithm can yield solutions arbitrarily far from optimal; straightforward extensions guarantee approximation bounds of only 1/2 or 3/4 [28]. But does an optimal solver (e.g., our MIP or DP by profits) significantly outperform the greedy algorithm for animation rendering job selection? We compared greedy and optimal selectors in terms of how well they solve the selection problem alone (Equations 1 and 2 in Section 3.1). Our test used eighteen nights for which the total processing demands of all jobs exceeded $PD = 13,000$ CPU-hours by at least 10%. For linear and banded completion rewards, the optimal solution is at most 0.25% better than greedy. For size-dependent rewards, the optimal solution is up to 3% better. For the workload studied, greedy selection works well unless we face side constraints that only the MIP solver can address.

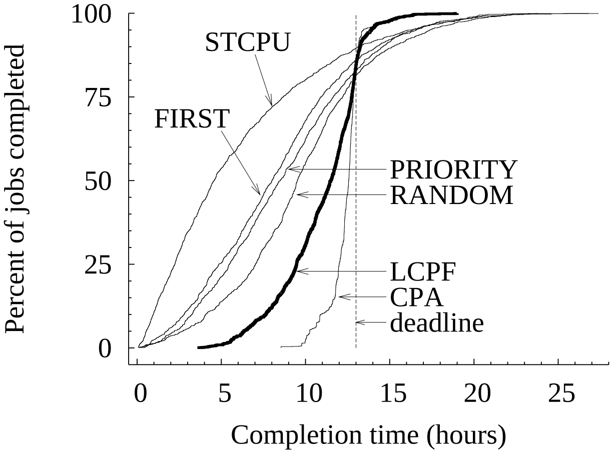

Figure 3: Job completion times.

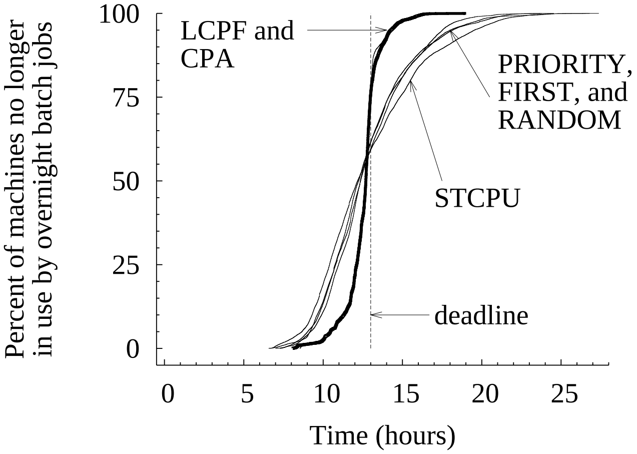

Figure 4: Times at which processors cease work.

5.3 Dispatching

In this section, we compare dispatcher behavior by a variety of measures. We implemented a simulator that supports over twenty dispatcher policies, including those described in Section 3.2. We used traces from 32 nights whose submitted jobs required at least 13,000 CPU-hours of processing time. For each night, we used a greedy selector to reduce the total CPU demand to less than 13,000 CPU-hours. After selection, a total of 1,311 jobs remained.

Figure 3 shows the distribution of job completion times for six dispatcher policies. Job completion time profiles differ dramatically across these six policies. STCPU begins to complete jobs almost immediately and has finished half of the jobs after only 4.6 hours. By contrast, CPA completes *no* jobs for more than eight hours. After ten hours, STCPU has completed over 80% of all jobs, PRIORITY has finished 62.5%, LCPF has completed 30.7%, and CPA has finished only 2.4%. LCPF and CPA overtake the other policies shortly *after* the deadline. However, by tuning selection parameter r we can shift all of the distributions to the left: When selection uses a lower value of r, all policies complete more jobs by

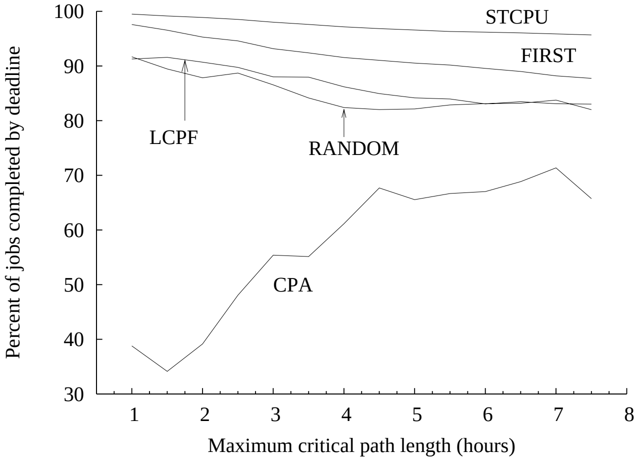

Figure 5: Shorter critical paths help all policies except CPA.

the deadline and we can adjust *when* LCPF overtakes the other policies. For our animation workload, $r \approx 0.9$ yields good results. LCPF and CPA furthermore perform much better than the others in terms of makespan, the time to finish the last job. The worst makespan for both LCPF and CPA is 19 hours; for PRIORITY it is over 23 hours and for STCPU it is 26.8 hours.

A crucial difference between LCPF and CPA is that LCPF completes far more jobs *early in the night*, while both outperform STCPU late in the night. Completion time profiles are important in practice for animation rendering: quality control staff must review completed jobs and re-run those that yielded unsatisfactory results. CPA leaves the staff idle for most of the night shift and overworked near the deadline. It also leaves too little time to rerun failed jobs, which are not uncommon (see [47], workload characterization).

Figure 4 shows the distribution of times at which processors cease to serve tasks. LCPF and CPA utilize nearly all processors until the deadline, then release them all nearly simultaneously. An hour after the deadline, LCPF has released all but 7.2% of processors, whereas STCPU is still using 31.7% of them. LCPF and CPA produce tidy rectangular Gantt charts, whereas the other policies yield schedules with ragged right edges. Uniform processor release times are desirable in animation and other domains, because clusters that run batch jobs overnight serve interactive users during the day.

A final advantage of LCPF is that it achieves higher processor utilization than the other policies: idle time (shaded in Figure 2) is only 0.6% for LCPF but exceeds 3% for STCPU. Idleness during the 13 hour window before the deadline is 13.5% for STCPU and 10.5% for PRIORITY but only 5.5% for LCPF.

5.4 Selection and Dispatching Together

We also evaluated a complete two-phase scheduler (selector plus dispatcher) in terms of aggregate reward R_Σ. These tests used traces from the ten nights whose submitted jobs had the greatest total processor demand $\sum T_1(j)$. We assigned rewards to jobs using each of the three methods described in Section 5.1. We used our MIP to select jobs, varying selection parameter r to see how under- or over-selection impacts aggregate rewards. For simplicity, in this section we assume that exact clairvoyant (offline) knowledge of task execution times is available to dispatcher policies. Appendix A considers the case where imperfect estimates of critical path lengths and processor demand are available but task execution times are not.

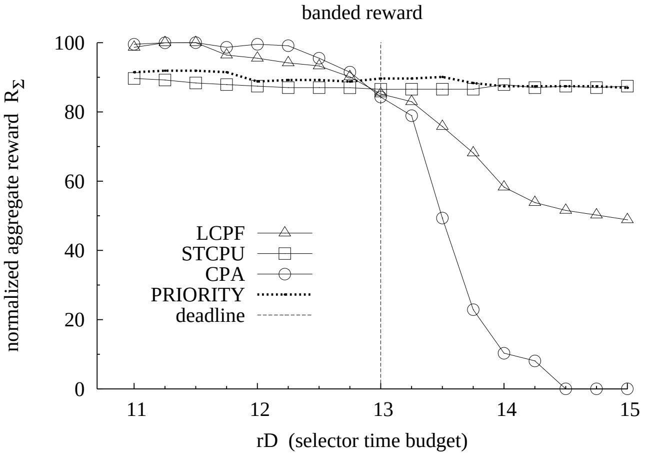

Figure 6: Banded rewards.

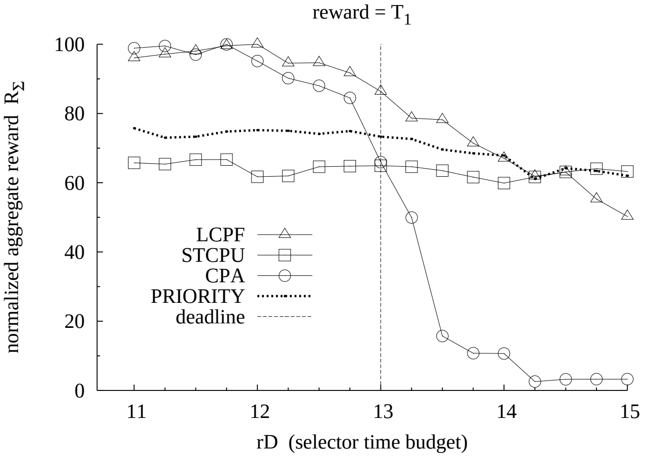

Figure 7: Size-dependent rewards.

Figures 6 and 7 present results for banded and size-dependent rewards, respectively (linear-reward results closely resemble Figure 6 and are presented in Appendix A). The horizontal axes show the time budget $r \cdot D$ used during selection. E.g., $x = 12$ hours corresponds to $r = \frac{12}{13}$ and means that the selector chose a subset of jobs with aggregate processing demand no greater than 12,000 CPU-hours.

Our results show that LCPF outperforms the other policies provided that r is tuned; $r \approx 0.9$ works well for our traces. CPA is comparable to LCPF if the selection parameter is well tuned, but it suffers far more than LCPF when r is poorly tuned. STCPU and PRIORITY are relatively insensitive to r but yield considerably lower aggregate reward than well-tuned LCPF. In terms of aggregate reward, LCPF outperforms PRIORITY by 9% for banded rewards and 32% for size-dependent rewards. Current practice in many production environments appears in Figures 6 and 7 as the PRIORITY data series at $x = 15$ hours; this approximates the case of "no selection + priority scheduling."

Theorems 3 and 4 show that MAXK, which uses knowledge of individual task lengths L_i, performs no better in the worst case than our two-phase solution, which requires only total processing demand $T_1(j)$ for each job. Our experiments show that CPA, which requires task lengths, performs no better on real inputs than LCPF, which requires only critical path lengths. In summary, our theoretical and empirical results point to the remarkable conclusion that *ignorance of individual task execution times does not make* DSSP *harder*.

5.5 Impact of Critical Path Length

To understand the impact of critical path length T_∞ on dispatcher performance, we randomly generated workloads with different $T_\infty^{\max}$. We randomly chose subsets of jobs from our trace that had $T_1(j)$ close to 13,000 CPU-hours and $T_\infty^{\max}$ less than a specified upper bound. We constructed 30 random subsets for each value of $T_\infty^{\max}$ between one hour and 7.5 hours in 15-minute increments.

Figure 5 shows how several dispatcher policies performed on these workloads. There are two striking features. First, CPA performs very poorly because it completes many jobs slightly *after* the deadline. Unlike the other policies shown, CPA is very sensitive to r and fails dramatically if this parameter is not tuned. (LCPF suffers for the same reason; however we have seen in Section 5.4 that it outperforms other policies with an appropriate r.) Second, the performance of all other dispatcher policies improves by 5–15% as $T_\infty^{\max}$ decreases. Like the theoretical results of Section 4, our empirical results demonstrate that short critical paths increase the number of jobs completed by the deadline. This result has direct practical implications: Tasks can often be parallelized, e.g., in animation rendering individual frames can be broken into "tiles" that are rendered separately. Parallelization typically complicates a computation and increases total processor demand; our results demonstrate benefits against which these costs can be weighed.

6. RELATED WORK

Scheduling is a basic research problem in both computer science and operations research. The space of problems is vast; [8, 38] provide good reviews. In this section we focus on non-preemptive multiprocessor scheduling without processor-sharing.

6.1 Minimizing Makespan

Much scheduling research focuses on minimizing makespan for tasks with arbitrary precedence constraints. Graham [18] showed that a simple priority-based heuristic called *list scheduling* has a worst case performance ratio of $2-1/P$. Brent [7] and Graham [19] extended list scheduling to arbitrary non-delay scheduling and proved similar results. Adam *et al.* empirically evaluated list scheduling algorithms for parallel programs and reported that CPA ("HLFET" in their terminology) yields better makespan than several alternatives for both real and synthetic workloads [1]. See Kwok and Ahmad [32] for a recent survey of static scheduling algorithms and Sgall [42] for online scheduling algorithms. Makespan minimization is also studied for a wide range of computation models, such as uniform machines [11, 9], unrelated machines [34], open shops [41], flow shops [20, 15] and job shops [15, 29].

The special case where each task has unit execution time has received extra attention. Lenstra & Kan [33] proved that it is NP-hard to approximate this special case within a ratio of $4/3$. It is an open problem whether we can get an upper bound better than 2. If $P=2$, makespan minimization is possible in polynomial time [36, 12]. It is an outstanding open problem [25] whether it is NP-hard for $P=3$. For multithreaded computations on parallel computers, normally a task represents a single machine instruction, thus unit execution time is a natural assumption. Blumofe and Leiserson [4, 5] and Gibbons *et al.* [2] analyze time- and space-efficient parallel scheduling algorithms for well-structured multithreaded computations. Their work extends Cilk, a provably time- and memory-efficient runtime thread scheduler [3].

6.2 Minimizing Mean Completion Time

Minimizing mean task completion time is an important objective in both systems and theory literature. The classic *shortest job first* heuristic is optimal in the offline case with no precedence constraints, and works well in many online scheduling systems. See [10] for a recent survey of theoretical results in this domain.

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 [44]. Kumar and Shorey [31] analyze mean response time for stochastic "fork-join" jobs, where fork-join jobs closely resemble the stage-structured jobs of DSSP. 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.

6.3 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.* [16] for an overview of theory and practice in this space, and Krallmann *et al.* [30] for a general framework for the design and evaluation of scheduling algorithms. Most work in this space empirically evaluates scheduling heuristics, such as backfilling [35], adaptive scheduling [23], and task grouping [43], to improve system utilization and throughput. Markov [37] described a two-stage scheduling strategy for Sun's Grid Engine, which 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.

6.4 Commercial Products

Open-source schedulers such as Condor manage resources, monitor jobs, and enforce precedence constraints [13]. Commercial products such as LSF additionally enforce fair-share constraints [39]. 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 per-job completion rewards of our framework are more expressive than ordinal job priorities and thus better suited to DSSP.

7. CONCLUSIONS

DSSP is a new class of scheduling problem for which two-phase decomposition is a promising approach. While we experiment only with an animation rendering workload, many other commercial applications have the same structure as DSSP and we hope to find traces for them as well and learn about their other requirements.

We presented a two-phase scheduling solution framework and a variety of algorithms for both the job selection and task dispatching phases. While deadline scheduling work traditionally focuses on offline solutions that require complete knowledge of task and job run times, we compared algorithms that require complete knowledge with algorithms that use only rough predictions of run times for entire jobs, not individual tasks. Both our theoretical results and our empirical comparisons of the algorithms reach the same surprising conclusion: our two-phase scheduling methods perform well even without detailed knowledge of individual task run times.

We conclude that an optimal selection algorithm combined with our new LCPF dispatching algorithm solve DSSP well. Our approach can satisfy a wide range of constraints while achieving high aggregate value. LCPF also performs very well according to several secondary criteria: like CPA it overtakes other policies in terms of job completions near the deadline, but it dominates CPA well before the deadline. LCPF also releases processors nearly simultaneously.

8. 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, 1974.

[2] Guy E. Blelloch, Phillip B. Gibbons, and Yossi Matias. Provably efficient scheduling for languages with fine-grained parallelism. *JACM*, 46(2):281–321, 1999.

[3] Robert D. Blumofe. *Executing Multithreaded Programs Efficiently*. PhD thesis, MIT, September 1995.

[4] Robert D. Blumofe and Charles E. Leiserson. Space-efficient scheduling of multithreaded computations. *SIAM J Comput*, 27(1):202–229, 1998.

[5] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. *JACM*, 46(5):720–748, September 1999.

[6] Abhijit Bose. Personal communication, September 2004.

[7] Brent. The parallel evaluation of general arithmetic expressions. *JACM*, 21(2):201–206, 1974.

[8] Peter Brucker. *Scheduling Algorithms*. Springer, 3rd edition, 2001.

[9] C. Chekuri and M. A. Bender. An efficient approximation algorithm for minimizing makespan on uniformly related machines. In *Proc 6th Conf on Integer Programming & Combinatorial Optimization (IPCO'98)*, pages 383–393. Springer LNCS 1412, 1998.

[10] Chandra Chekuri and Sanjeev Khanna. Approximation algorithms for minimizing average weighted completion time. In Joseph Leung, editor, *Handbook of Scheduling: Algorithms, Models, and Performance Analysis*. CRC Press, 2004.

[11] F. A. Chudak and D. B. Shamoys. Approximation algorithms for precedence-constrained scheduling problems on parallel machines that run at different speeds. *J Algorithms*, 30:323–343, 1999.

[12] E. Coffman and Ronald Graham. Optimal scheduling for two processor systems. *Acta Informatica*, pages 200–213, 1972.

[13] Directed acyclic graph manager (DAGMan) for Condor scheduler. `http://www.cs.wisc.edu/condor/dagman/`.

[14] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. In *OSDI*, December 2004.

[15] U. Feige and C. Scheideler. Improved bounds for acyclic job shop scheduling. In *STOC*, pages 624–633, 1998.

[16] Dror G. Feitelson, Larry Rudolph, Uwe Schwiegelshohn, Kenneth C. Sevcik, and Parkson Wong. Theory and practice in parallel job scheduling. In *Proceedings of JSSPP [26], LNCS 1291*, pages 1–34, 1997.

[17] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman, 1979.

[18] Ron Graham. Bounds for certain multiprocessor anomalies. *Bell Sys Tech J*, 45:1563–1581, 1966.

[19] Ronald Graham. Bounds on multiprocessing time anomalies. *SIAM J Appl Math*, 17:263–269, 1969.

[20] L. A. Hall. Approximability of flow shop scheduling. *Mathematical Programming*, 82:175–190, 1998.

[21] Les Hatton. The T experiments: Errors in scientific software. *IEEE Computational Sci & Eng*, pages 27–38, April 1997.

[22] Les Hatton and Andy Roberts. How accurate is scientific software? *IEEE Trans Software Eng*, 20(10):785–797, October 1994.

[23] 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.

[24] ILOG Corporation. *CPLEX and related software documentation*. `http://www.ilog.com`.

[25] David Johnson. The twelve open problems from [G&J]: Updates. *J of Algorithms*, 2(4):393–405, 1981.

[26] Workshops on Job Scheduling Strategies for Parallel Processing (JSSPP). `http://www.cs.huji.ac.il/~feit/parsched/index.html`. Proceedings are published in Springer LNCS series: `http://www.cs.huji.ac.il/~feit/parsched/lncs.html`.

[27] Richard M. Karp. Reducibility among combinatorial computations. In R. E. Miller and J. W. Thatcher, editors, *Complexity of Computer Computations*, pages 85–103. Plenum Press, 1972.

[28] Hans Kellerer, Ulrich Pferschy, and David Pisinger. *Knapsack Problems*. Springer, 2004.

[29] Maxim Sviridenko Klaus Jansen, Roberto Solis-Oba. Makespan minimization in job shops: A linear time approximation scheme. *SIAM J on Discr Math*, 16(2):288–300, 2003.

[30] Jochen Krallmann, Uwe Schwiegelshohn, and Ramin Yahyapour. On the design and evaluation of job scheduling algorithms. In *Proceedings of JSSPP [26], LNCS 1659*, pages 17–42, 1999.

[31] 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.

[32] 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.

[33] J. K. Lenstra and A. H. G. Rinnooy Kan. Complexity of scheduling under precedence constraints. *Operations Research*, 26(1):22–35, January 1978.

[34] J. K. Lenstra, D. B. Shmoys, and E. Tardos. Approximation algorithms for scheduling unrelated parallel machines. *Mathematical Programming*, 46:259–272, 1990.

[35] David A. Lifka. The ANL/IBM SP scheduling system. In *Proceedings of JSSPP [26] LNCS 949*, pages 295–303, 1995.

[36] T. Kasami M. Fuju and N. Ninomiya. Optimal sequence of two equivalent processors. *SIAM J Appl Math*, 17(3):784–789, 1971.

[37] 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.

[38] Michael Pinedo. *Scheduling: Theory, Algorithms, and Systems*. Prentice-Hall, 2nd edition, 2002.

[39] Platform Computing. LSF Scheduler. `http://www.platform.com/products/LSFfamily/`.

[40] Platform Computing. *Administering Platform LSF*, February 2003. Chapter 14.

[41] S. V. Sevastianov and G. J. Woeginger. Makespan minimization in open shops: A polynomial time approximation scheme. *Mathematical Programming*, 82:191–198, 1998.

[42] 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, 1998.

[43] 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.

[44] 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.

[45] Francis Wray. The parallel implementation of closely coupled numerical algorithms. In A. Adey, editor, *Parallel Processing in Engineering Applications*. Springer, 1990.

[46] Francis Wray. High performance numerically intensive applications on distributed memory parallel computers. In J. T. Devreese and P. E. Van Camp, editors, *Scientific Computing on Supercomputers*, volume III. Plenum, 1991.

[47] Yunhong Zhou, Terence Kelly, Janet Wiener, and Eric Anderson. An extended evaluation of two-phase scheduling methods for animation rendering. In *Proceedings of JSSPP [26] LNCS, Springer*, 2005, to appear.

APPENDIX

A. EXTENDED EMPIRICAL RESULTS

The empirical results presented in Section 5.4 consider the case where dispatcher policies have clairvoyant/offline knowledge of task execution times. In this appendix we consider the case where our scheduler instead must rely on estimates of computational demands that are available in the environment where our traces were collected. Specifically, our scheduler uses the estimates of critical path lengths and total CPU demand described in [47].

Figures 8 through 10 show results analogous to Figures 6 and 7 for the three reward functions described in Section 5.1. The data series labeled "actual" correspond to experiments in which both job selection and task dispatching exploit complete clairvoyant knowledge of task execution times. Those labeled "predicted" employ estimates of T_1 and T_∞ for job selection and dispatching. We cannot present "predicted" series for CPA because it requires estimates of individual task execution times, which are not available for our workload.

The results for banded and linear rewards (Figures 8 and 9) are similar. In both cases LCPF is relatively much less sensitive to a poorly tuned selection parameter *r* than CPA. Furthermore LCPF outperforms the other policies provided that the selection parameter is well tuned. For size-dependent rewards (Figure 10), LCPF outperforms both STCPU and PRIORITY in the "predicted" case *regardless* of *r*. LCPF delivers roughly 9% greater aggregate value than PRIORITY for size-dependent rewards, when the best performance of both policies are compared.

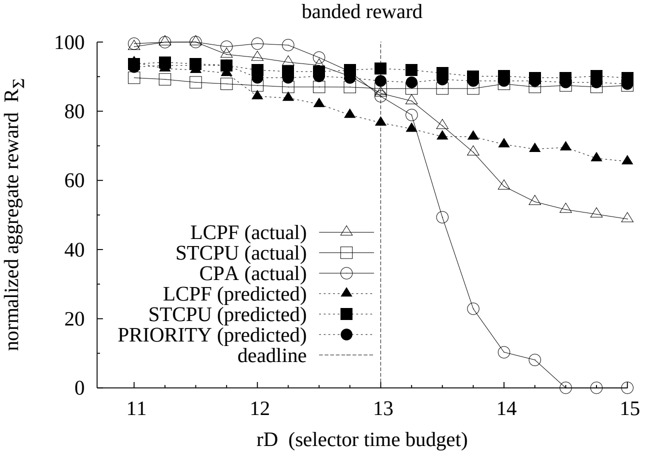

Figure 8: Banded reward.

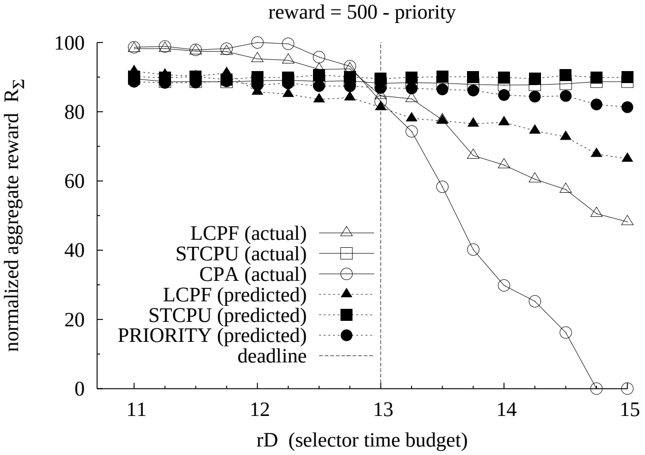

Figure 9: Linear reward.

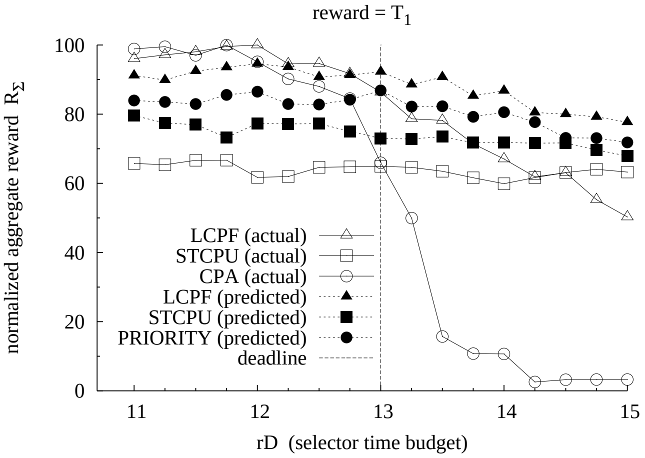

Figure 10: Size-dependent reward.