Fault-Tolerant Real-Time Scheduling

Bala Kalyanasundaram[*] *Kirk Pruhs*[†]

October 31, 1997

Abstract

We use competitive analysis to study how to best use redundancy to achieve fault-tolerance in online real-time scheduling. We show that the optimal way to use spatial redundancy depends on a complex interaction of the benefits, execution times, release times, and latest start times of the jobs. We give a randomized online algorithm whose competitive ratio is $O(\log\Phi\log\Delta\frac{\log^2 n\log m}{\log\log m})$ for transient faults. Here n is the number of jobs, m is the number of processors, Φ is the ratio of maximum value density of a job to the minimum value density of a job, and Δ the ratio of the longest possible execution time to the shortest possible execution time. We show that this bound is close to optimal by giving an $\Omega(\frac{\log\Delta\Phi}{\log\log m}(\frac{\log m}{\log\log m})^2)$ lower bound on the competitive ratio of any randomized algorithm. In the case of permanent faults, there is a randomized online algorithm that has a competitive ratio of $O(\log\Phi\log\Delta\frac{\log m}{\log\log m})$. We also show a lower bound of $\Omega(\frac{\log\Delta\Phi}{\log\log m}\frac{\log m}{\log\log m})$ on the competitive ratio for interval scheduling with permanent faults.

1 Introduction

We use competitive analysis to study how to best utilize redundancy to achieve fault tolerance in the setting of online real-time scheduling. In an online scheduling environment, the jobs arrive over time and must be scheduled without knowledge of future jobs. In real-time scheduling each job has a deadline, and a job must be finished by its deadline to be of full use to the system. Fault-tolerance is achieved by having multiple processors performing the same job so that if any one of these processors does not fault, the job will be completed. The question we examine is, "How should a fixed number of processors be allocated among the jobs?". We show that with job deadlines and laxity, the optimal online allocation depends on a complex interaction of various characteristics of the jobs, including release time, length and latest start times.

[*]Department of Computer Science, University of Pittsburgh, Pittsburgh, PA 15260. E-Mail: kalyan@cs.pitt.edu

[†]Department of Computer Science, University of Pittsburgh, Pittsburgh, PA 15260. E-Mail: kirk@cs.pitt.edu

1.1 Problem Statement

We adopt best-effort firm real-time scheduling, a standard model of real-time computation [1, 2, 5, 7, 8, 11], as our model of real-time computation. The setting is a collection $P_1, \ldots P_m$ of unit speed processors. The scheduler sees over time a collection $J_1, \ldots J_n$ of jobs. Each job J_i has a *release time* r_i, a processing time x_i, a deadline d_i, and a benefit b_i. Unless we state otherwise, we number the jobs in increasing order of release times, that is, $r_i < r_j$ if $i < j$, The scheduler is unaware of J_i until time r_i, at which time the scheduler additionally learns x_i, d_i, and b_i. Thus the scheduler may schedule J_i any time after time r_i. If some processor completes J_i by time d_i then the scheduler receives a benefit of b_i, otherwise no benefit is gained from this job. Note that we allow many processors to run a job J_i simultaneously, and that these processors need not have begun execution of J_i at the same time. The problem is to maximize the sum of the benefits of the jobs completed before their deadlines by at least one processor.

Interval scheduling is a special case of best effort firm real-time scheduling where the *laxity* $l_i = d_i - x_i - r_i$ of each job J_i is equal to zero. The *value density* v_i of J_i is b_i/x_i. We assume that the minimum execution time and minimum value density of any job are 1, that the maximum value density is Φ, and that the maximum execution time is Δ. We assume that Φ and Δ are known a priori. We say an instance is *static* if all release times are 0. The *latest start time* s_i of a job J_i is $d_i - x_i$.

One gets different models depending on the type of preemption allowed. The three standard options are nonpreemptive, preemptive, and preemptive with restart. A scheduling algorithm is *nonpreemptive* if a processor P_j may not interrupt a job J_i once it has begun execution of J_i, is *preemptive* if it is allowed to instantaneously abandon J_i and later restart J_i from the point of suspension, and is *preemptive with restart* if it can abandon J_i but must restart J_i from the beginning if it wants to return to J_i.

In this paper we consider only fail-stop faults. A processor P_j experiences a *fail-stop fault* during a period T of time if P_j is not capable of performing any work during the period T, that is, it is as if the processor was switched off during T. Thus, we do not consider malicious faults. A *permanent fault* is one from which the processor never recovers, otherwise, the fault is *transient.* A *checkpointing system* may instantaneously save the state of a job J_i, which is called a checkpoint, and then if J_i should experience a fault, J_i can be restarted from the most recent checkpoint. A *noncheckpointing system* must restart a job J_i from the beginning if J_i should experience a fault. For example, a file transfer may be checkpointing at the system level, but (as ftp users are painfully aware) restart at the users level. Note that the ability of the system to checkpoint is essentially the same as the ability to preempt. So we are left with three reasonable models of a fault-tolerant version of best-effort firm real-time scheduling: 1) preemptive and checkpointing, 2) nonpreemptive and noncheckpointing, and 3) preemptive with restart and noncheckpointing.

We now define the competitive ratio for a maximization problem. The competitive ratio of a randomized algorithm $\mathcal{A}$ on an instance I is the ratio of the optimal off-line benefit achievable for I over the expected benefit achieved by $\mathcal{A}$ on input I. By convention, the competitive ratio of $\mathcal{A}$ on I is 1 if the optimal benefit for I is 0. The competitive ratio of a randomized algorithm on a problem is is the supremum over all instances I of the competitive ratio for A on I. The competitive ratio can be viewed as the payoff to a game played between two players, the online player and the adversary. The adversary specifies the input and services that input optimally. We assume an oblivious adversary, that is, the adversary may not modify I in response to the outcome of a random event in $\mathcal{A}$. We assume that processor faults are not caused by the particular job running on that processor. That is, the faults are specified in I by a list with entries of the form (P_i, t, α), meaning that processor P_i is faulty between time t and time $t + \alpha$, independent of the jobs assigned to P_i.

In section 4, we show that the competitive ratio of every deterministic algorithm is linear in the number of processors for a preemptive with restart and noncheckpointing system. We also show, in section 4, that the competitive ratio of every randomized algorithm for a preemptive and checkpointing system, and for a nonpreemptive and noncheckpointing system, is linear in the number of jobs. Until then, we devote our attention to randomized algorithms for systems where the scheduler is preemptive with restart and noncheckpointing.

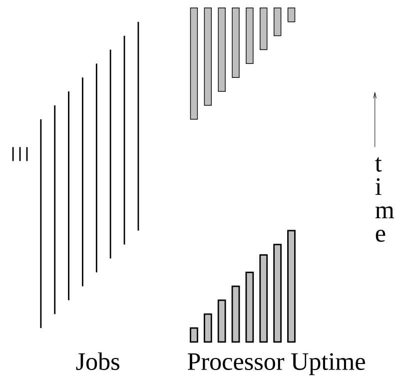

Figure 1: A Hard Instance for Monotone Algorithms

For preemptive with restart schedulers, figure 1 shows an instance of interval scheduling that is particularly hard for many natural algorithms. Jobs are represented as vertical lines with the length of the line being the execution time and the bottom of the line representing the release time. Processors are represented by vertical columns that are darkened when the processor is faulty. The benefit of every job is 1. From any particular processor's point of view, after recovering from a fault at time t, the processor should intuitively prefer jobs with earliest deadline among those released after t since these jobs are least likely to be affected by future faults. Yet if there are few faults, you don't want every processor picking the early deadline jobs or there would not be enough distinct jobs to be competitive. So the most natural algorithms would have some monotone nonincreasing probability distribution on the jobs ranked by increasing deadline. By making the number of short jobs $\Theta(\sqrt{m})$, and the number of long jobs m, this instance shows that such algorithms will have competitive ratio $\Omega(\sqrt{m})$.

1.2 Our Results

In sections 2 and 3, we consider preemptive with restart and noncheckpointing systems that experience only fail-stop faults. In section 2, we give a randomized online algorithm CLASSIFY that has a competitive ratio of $O(\log\Phi\log\Delta\frac{\log^2 n\log m}{\log\log m})$. In section 3, we give an $\Omega(\frac{\log\Phi\Delta}{\log\log m}(\frac{\log m}{\log\log m})^2)$ lower bound on the competitive ratio of any randomized algorithm.

In the case of permanent faults, We give a randomized online algorithm with a competitive ratio that is $O(\log\Phi\log\Delta\frac{\log m}{\log\log m})$. We show that this upper bound is reasonably tight by giving a lower bound of $\Omega(\frac{\log\Phi\Delta}{\log\log m}\frac{\log m}{\log\log m})$ on the competitive ratio for interval scheduling with permanent faults.

These results show that the optimal way for a online scheduler to use redundancy is in a real-time environment is nontrivial. This is in contrast to fault-tolerant scheduling of nonreal-time jobs, where it is known that the best way for an online scheduler to use redundancy is to devote an equal number of processors to each equal valued job [4]. These results also show that for real-time scheduling it is the case that transient faults are significantly more damaging than permanent faults to the performance of the online scheduler. For scheduling jobs without deadlines, it was known that transient and permanent faults are essentially equally damaging [4].

1.3 Previous Results

We first survey results for online best-effort firm real-time scheduling without faults. All of the following results are for a single processor with preemption unless stated otherwise. Let Λ be the ratio of the largest benefit of a job to the minimum benefit of a job. There are 4-competitive algorithms for the cases $\Phi = 1$ or $\Delta = 1$, and this is optimal for deterministic algorithms [2, 11]. The optimal competitive ratio is $\Theta(\Phi)$ [2, 7]. In [2] a 2-competitive algorithm for interval scheduling is given for the case $\Phi = 1$ and $m = 2$. This immediately yields a 2-competitive randomized algorithm for interval scheduling for $\Phi = 1$ and $m = 1$. A 3/2 lower bound on the competitive ratio for this case is given in [6]. In [8] a deterministic algorithm for multiple processors is given with competitive ratio $O(\log\Phi)$ for large m, and this is shown to be optimal. In [6] a randomized algorithm is given with competitive ratio $O(\min(\log\Phi, \log\Delta))$, and this is shown to be optimal. In [1], a nonconstant deterministic lower bound for the case that $\Lambda = 1$ is given.

For nonpreemptive scheduling with $\Phi = 1$, there is an $O(\log\Delta)$-competitive randomized algorithm and this is optimal [3, 9].

Faults completely change the nature of these scheduling problems. There seems to be little research that uses competitive analysis to study fault-tolerance. In [4] fault-tolerant scheduling of jobs without deadlines is studied, and optimal deterministic and randomized algorithms are given for online problems involving minimizing the make-span and minimizing the average flow time. In each case, the optimal way for the online scheduler to use redundancy was essentially to devote an equal number of processors to each job.

1.4 Preliminaries

All logarithms in this paper are base 2. For convenience, we assume that no pair of events (job releases, job deadlines, faults, fault recoveries, job completions, and latest start times) occur at the same time. This allows us to associate an unique event with a particular time, thus simplifying the notation. To see that there is no loss of generality consider two events. If these two events are release times then they may be ordered arbitrarily. If one event is the completion time of a job J_j, and the other event is the fault of a processor P_i running J_j, then either P_i can finish J_j or it can't. If P_i can finish J_j, then we consider the competition time to occur before the fault, otherwise, we consider the fault to occur before the completion time. Other cases can be treated similarly.

2 Upper Bounds

We consider the problem where the scheduler is preemptive with restart and noncheck-pointing. We give a randomized online algorithm CLASSIFY with a competitive ratio of $O(\log \Phi \log \Delta \frac{\log^2 n \log m}{\log \log m})$. We assume that Δ and Φ are integer powers of 2, otherwise, we can round them up to the next integer power of 2.

2.1 Reduction to One Round Problem

At the top level CLASSIFY can be viewed as a reduction to a restricted version of the general problem, which we call the One Round Problem.

Algorithm CLASSIFY: The processors together initially select, uniformly at random, an integer $s \in \{0, 1, 2\}$, an integer $a \in [0, \log \Delta]$, an integer $b \in [0, \log \Phi]$, and a bit $w \in \{0, 1\}$. CLASSIFY will then ignore all jobs J_k with either $x_k \notin [2^{a-1}, 2^a]$, or $v_k \notin [2^{b-1}, 2^b]$. Hence, the benefit b_i of any job J_i, considered by CLASSIFY will lie within the range $[2^{a-1}2^{b-1}, 2^a 2^b]$.

The execution of the algorithm is divided into rounds. Round r, $r \geq 0$, begins at time $(3r + s)2^{a-1}$ and ends at time $(3r + s + 3)2^{a-1}$. At the end of a round, all jobs will be preempted. Let $\mathcal{J}_r$ be the collection of jobs not finished by some processor before the beginning of round r, and that are released before time $t = (3r + s + 1)2^{a-1}$. If $w = 0$ then then each processor P_i calls an algorithm DELAY on input $\mathcal{J}_r$ at the start of round r, and each time after P_i recovers from a fault during round r. If $w = 1$ then each processor P_i calls an algorithm NO-DELAY on input $\mathcal{J}_r$ at the start of round r, and each time after P_i recovers from a fault during round r. When we describe the algorithms NO-DELAY and DELAY we will deal with the fact that $\mathcal{J}_r$ is not fully known the time of the invocation of these algorithms.

End Algorithm

This naturally leads us to consider the following problem.

One Round Problem: We assume that the length of each job is in the range $[1, 2]$, and that the benefit of each job is 1. All jobs are released before time 1. To accrue a benefit from a job J_i, the scheduler must begin executing J_i before time 1 (the reason for this condition will be explained in the proof of theorem 2).

We call a job J_k *nondelayed* if the adversary finished J_k at time $r_k + x_k$, and *delayed* if the adversary finishes J_k at some time later than $r_k + x_k$. Note that jobs not finished by the adversary are neither delay nor nondelayed. We say a randomized algorithm A is c-competitive against nondelayed (respectively delayed) jobs if the expected benefit that A accrues from all the jobs is at least $\frac{1}{c}$ of the benefit that the adversary accrues from nondelayed (respectively delayed) jobs. In the One Round Problem, a time t, $t \leq 1$, is a *wake-up time* for a processor P_i if either $t = 0$, or P_i just recovered from a fault at time t. The *latest wake-up time* w_i of a processor P_i is its latest wake-up time before time 1.

Lemma 1 For every instance of the One Round Problem, there is an optimal schedule such that for any job J_j that is completed on a processor P_i, it is that case that P_i began executing J_j at either time w_i or time r_j.

Proof: If this wasn't true for a processor P_i and job J_j, one could move the time that J_j was scheduled on P_i back in time until $\max(r_j, w_i)$, and still maintain feasibility and optimality. Recall that for the One Round Problem, a scheduler can only gain benefit from jobs that it begins executing before time 1. ■

Theorem 2 If for every instance I of the One Round Problem it is the case that DELAY is c-competitive against the delayed jobs, and NO-DELAY is c-competitive against nondelayed jobs, then the competitive ratio of CLASSIFY will be at most $24c \cdot (1 + \log \Delta)(1 + \log \Phi)$.

Proof: The competitive ratio of CLASSIFY is at most the maximum over all $a \in [0, \log \Delta]$, and $b \in [0, \log \Phi]$, of the ratio of the expected benefit accrued by CLASSIFY from jobs J_k with $x_k \in [2^{a-1}, 2^a]$, and $v_k \in [2^{b-1}, 2^b]$, to the benefit accrued by the adversary on all such jobs. Thus assume that all jobs J_k now satisfy $x_k \in [2^{a-1}, 2^a]$, and $v_k \in [2^{b-1}, 2^b]$. By scaling the lengths by 2^{a-1} we may assume all jobs have lengths between 1 and 2. By scaling the benefits by 2^{b-1}, the competitive ratio that we obtain by assuming that all jobs have unit benefit will be off by at most a factor of two.

Consider some arbitrary round r. The expected aggregate benefit earned by CLASSIFY during round r is $\frac{1}{2c}$ of the delayed jobs completed by adversary in round r and not completed by CLASSIFY in an earlier round, plus $\frac{1}{2c}$ of the nondelayed jobs completed by adversary in round r and not completed by CLASSIFY in an earlier round. We first allocate half of the benefit from a job completed by *Classify* to the round that the adversary completed the job. Hence, in an amortized sense, the expected aggregate benefit earned by CLASSIFY during round r is $\frac{1}{4c}$ of the delayed jobs completed by adversary in round r, plus $\frac{1}{4c}$ of the nondelayed jobs completed by adversary in round r. We now note that each job's release time falls within the first unit of time of some round with probability $\frac{1}{3}$. Hence, we get that the expected benefit accrued by CLASSIFY is at least $\frac{1}{12c}$ of the benefit accrued by the adversary if jobs have unit benefit. ∎

We thus now want to analyze the algorithms DELAY and NO-DELAY for the One Round Problem.

2.2 The One Round Problem

In this subsection we restrict our attention solely to the One Round Problem. Both the algorithm DELAY and NO-DELAY can be viewed as reductions to the case of agreeable jobs, which we will define later. Our algorithm P1R for handling agreeable jobs relies on a solution to the Selection Problem, which we now introduce.

2.2.1 The Selection Problem

Selection Problem: Let p be a probability distribution on the positive integers such that for all $i \geq 1$, $p(i) \geq p(i+1)$, where $p(x)$ is the probability that the random variable is equal to x. Let $\prec$ be an arbitrary total ordering among a collection $\mathcal{J}$ of jobs that are released online over time. Assume that the online player can effectively decide the $\prec$ order between any two pair of jobs. At any time the online player may hold no job, or one job. So if the online player is holding job J_i when job J_j is released, the online player must permanently discard one of J_i or J_j. After the last job is released, the online player's goal is to, be holding the kth $\prec$-smallest job since its release time, with probability at least $p(k)$.

We now present an online strategy DISTRIBUTE for solving the Selection Problem.

Algorithm DISTRIBUTE: Initially the online player is holding no jobs. Consider the time just after the uth job J_u arrives. Assume that J_u is the rth $\prec$-smallest job at this time. We consider several cases. First assume that the online player is holding job J_i when job J_u arrives. If $J_i \prec J_u$ then the online player continues to hold J_i. Otherwise assume that J_i was the sth $\prec$- smallest job before J_u was released, and is now the $(s+1)$st $\prec$-smallest job. Note $r \leq s$. In this case the online player continues to hold J_i with probability $p(s+1)/p(s)$, and switches to J_u with probability $1 - p(s+1)/p(s)$. If the online player is holding no job when J_j arrives then the online player switches to holding J_u with probability $p(u)/(1 - \sum_{k=1}^{u-1} p(k))$.

End Algorithm

Lemma 3 The algorithm DISTRIBUTE solves the Selection Problem.

Proof: We prove by induction on u that after the release of the uth job J_u, the kth $\prec$-smallest job is held with probability $p(k)$, and that no job is held with probability $1 - \sum_{k=1}^{u} p(k)$. We consider the release of the uth job J_u, and adopt the notation in the algorithm description above. Let J_i be some job released before J_u. If $J_i \prec J_u$, the probability that J_i is held, as well as its relative ranking, remain unchanged. If $J_u \prec J_i$, then the probability that J_i remains held is the probability it was held before J_u was released, times the probability that the online player continues to hold J_i. The former probability is $p(s)$ by induction, and the latter is $p(s+1)/p(s)$ by the description of the algorithm. Thus the probability that J_i continues to be held is $p(s+1)$ as required. The probability that J_u is held after it is released is

$$[1 - \sum_{k=1}^{u-1} p(k)][p(u)/(1 - \sum_{k=1}^{u-1} p(k))] + \sum_{k=r}^{u-1} p(k)(1 - \frac{p(k+1)}{p(k)})$$

The first term is the probability that the online player was holding no job and then switched to J_u, and the second term is the probability that the online player switched from a $\prec$-larger job to J_u. This simplifies to

$$p(u) + \sum_{k=r}^{u-1}[p(k) - p(k+1)]$$

The sum $\sum_{k=r}^{u-1}[p(k) - p(k+1)]$ is a telescoping series that simplifies to $p(r) - p(u)$. Thus we have that J_u is held with probability $p(r)$ as required. The probability that the online player is holding no job after J_u is released is the probability that it was holding no job before J_u was released, times the probability that it didn't switch when J_u is released. By induction the first probability is $1 - \sum_{k=1}^{u-1} p(k)$, and by the description of the algorithm the second probability is $(1 - p(u)/(1 - \sum_{k=1}^{u-1} p(k))$ The product is thus $1 - \sum_{k=1}^{u} p(k)$ as required. ∎

In our applications of the Selection Problem we will always be using one fixed probability distribution. So from here on we adopt the following distribution.

Definition 4
a) Let c be the largest integer such that $c^c \leq m$.
b) For $1 \leq j \leq m$, let $p(j) = \frac{1}{c^k(c+2)}$ where $k(\geq 0)$ is the largest integer satisfying $c^{k-1} < j$.
c) For integer $j > m$, $p(j) = 0$.

Lemma 5
a) For all $i \geq 0$, $p(i) \geq p(i+1)$.
b) $\sum_{i=1}^{\infty} p(i) \leq 1$.
c) $c = \Theta(\frac{\log m}{\log \log m})$.

Proof: The first statement is obvious. Consider the second statement. Note that $p(1) = \frac{1}{c+2}$, and for $j \in [c^i + 1, \min(c^{i+1}, m)]$, $0 \leq i \leq c$, it is the case that $p(j) = \frac{1}{c^{i+1}(c+2)}$. Hence all i, with $0 \leq i \leq c$, it is the case that

$$\sum_{j=c^i+1}^{\min(c^{i+1},m)} p(j) \leq c^{i+1} \frac{1}{c^{i+1}(c+2)} = \frac{1}{c+2}$$

The claim is then immediate. The third statement can be verified by noting that $c^c \leq m \leq (c+1)^{c+1}$, and taking the logarithms of both sides. ■

2.2.2 Agreeable Jobs

We now define what we mean by agreeable jobs, give an algorithm P1R, and show that P1R is $O(\frac{\log m}{\log\log m})$ competitive if all jobs are agreeable over the times when processors are selecting jobs.

Definition 6
a) The *earliest completion time* $e_k(t)$ of a job J_k relative to time t is $\max(t, r_k) + x_k$ if $t + x_k \leq d_k$. Otherwise, $e_k(t)$ is $+\infty$.
b) We say $J_i \prec_t J_j$ if and only if $e_i(t) \leq e_j(t)$.
c) We say that collection of jobs $\mathcal{J}$ is $\prec$*-invariant over the interval* $[a, b]$ if for all $s, t \in [a, b]$ and for all $J_i, J_j \in \mathcal{J}$ it is the case that $J_i \prec_s J_j$ if and only if $J_i \prec_t J_j$.
d) A collection of jobs $\mathcal{J}$ is *agreeable* over a time period $[a, b]$ if for each job $J_i \in \mathcal{J}$ it is the case that $s_i \geq b$, and $\mathcal{J}$ is $\prec$-invariant over $[a, b]$.

Algorithm P1R($\mathcal{J}$): Assume that processor P_i called this algorithm at time t. The processor P_i then uses the algorithm DISTRIBUTE on the jobs in $\mathcal{J}$ using the $\prec_t$ ordering. We assume that it is possible for the online algorithm to determine whether a job J_i is in $\mathcal{J}$ at the time r_i of J_i's release.

End Algorithm

Intuitively, P1R($\mathcal{J}$) guarantees that at all times, that P_i has been running the job $J_k \in \mathcal{J}$ with the jth earliest completion time relative to time t exclusively since time $\max(t, r_k)$ with probability at least $p(j)$. Lemma 7 is required to facilitate our analysis of P1R in theorem 10.

Lemma 7 Let J_j be some fixed job. Assume that a collection $\mathcal{P} = \{P_1, \ldots P_m\}$ of processors independently at random decide to select J_j. Let α_i be a 0/1 random variable equal to 1 if and only if P_i selected J_j. Let $q_i = E[\alpha_i]$ be the probability that processor P_i, selected J_j. Then the probability that at least one processor selects J_j is $\min(1, \frac{1}{2}\sum_{i=1}^m q_i)$.

Proof: If $\sum_{i=1}^m q_i \geq 1$ then we scale the probabilities down proportionally so that $\sum_{i=1}^m q_i = 1$. The probability that at least one processor selects J_j is $E[\max(\alpha_1, \ldots, \alpha_m)]$. We inductively show that $\gamma_k = E[\max(\alpha_1, \ldots, \alpha_k)]$ is at least $\min(1, \frac{1}{2}\sum_{i=1}^k q_i)$. The base case $k = 0$

is immediate. Now consider some arbitrary k. If $\gamma_{k-1} \geq \frac{1}{2}$ then the inductive claim follows immediately since $\gamma_k \geq \gamma_{k-1}$. Now assume that $\gamma_{k-1} < \frac{1}{2}$. Note that

$$\gamma_k = \gamma_{k-1} + (1-\gamma_{k-1})q_k = \gamma_{k-1} + q_k - \gamma_{k-1}q_k \geq \gamma_{k-1} + q_k - \frac{q_k}{2} \geq \frac{1}{2}\sum_{i=1}^{k} q_i$$

The second inequality follows since by induction $\gamma_{k-1} \geq \frac{1}{2}\sum_{i=1}^{k-1} q_i$. ■

In preparation for our analysis of P1R we need to define the notions of early jobs and late processors. Let $\mathcal{P}$ be a collection of processors with latest wake-up times in the range $[a,b] \subseteq [0,1]$, and let $\mathcal{J}$ be a set of jobs that are agreeable over $[a,b]$. Assume that each processor in $\mathcal{P}$ runs P1R($\mathcal{J}$) after each of its wake-up times in $[a,b]$. Let $OPT(\mathcal{J})$ and $OPT(\mathcal{P})$ be the jobs and processors involved in the optimal assignment OPT of jobs in $\mathcal{J}$ to processors in $\mathcal{P}$. Let $x = \mid OPT(\mathcal{J}) \mid$ be the optimal benefit accrued from jobs in $\mathcal{J}$. We consider two cases.

The first case is if $x \geq 6$. Let g be the largest integer such that $c^g + \frac{x}{3} + 2 \leq x$ (such a g must exit for sufficiently large c, or equivalently for sufficiently large m). We now rank the jobs in $OPT(\mathcal{J})$ by their earliest completion times relative to time a. We say a job J_i is *early* if its ranking is in the range $[(c^g+1),(c^g+\lceil\frac{x}{6}\rceil)]$. We say a processor P_j is *late* if the adversary finished a job J_i on P_j, and J_i's ranking is in the range $[(c^g+\lceil\frac{x}{6}\rceil),(c^g+\lfloor\frac{x}{3}\rfloor+2)]$.

The second case is if $x < 6$. In this case we let $g = 0$, we define the one *early* job to be the job $J_i \in OPT(\mathcal{J})$, with the earliest completion time relative to time a, and we define the one late processor to be the processor that finished J_i in OPT.

In either case note that there are at least $\frac{x}{6}$ early jobs, and at least $\frac{x}{6}$ late processors. Let $P(i,j)$ be the probability that, in the online schedule, P_i processed J_j exclusively since time $\max(r_j, w_i)$. We note the following facts.

Fact 8 By agreeability, each late processor P_i can complete every early job J_j if P_i were to start running J_j at time $\max(r_j, w_i)$.

Fact 9 By the definition of the probability distribution in definition 4 , and the definition of g, it is the case that for each late P_i, and each early J_j, $P(i,j) \geq \frac{1}{c^g(c+2)}$. Furthermore, by the definition of g, $P(i,j) = \Omega(\frac{1}{xc})$.

Theorem 10 Let $\mathcal{P}$ be a collection of processors with latest wake-up times in the range $[a,b] \subseteq [0,1]$, and let $\mathcal{J}$ be a set of jobs that are agreeable over $[a,b]$. If each processor in $\mathcal{P}$ runs P1R($\mathcal{J}$) after each of its wake-up times in $[a,b]$, then the expected total online benefit from jobs in $\mathcal{J}$ is $\Omega(\frac{\log\log m}{\log m})$ times the optimal that one could obtain by running jobs in $\mathcal{J}$ on the processors in $\mathcal{P}$.

Proof: Let $OPT(\mathcal{J})$ and $OPT(\mathcal{P})$ be the jobs and processors involved in the optimal assignment OPT of jobs in $\mathcal{J}$ to processors in $\mathcal{P}$. Let $x = \mid OPT(\mathcal{J}) \mid$ be the optimal benefit accrued from jobs in $\mathcal{J}$. By fact 9, for each late P_i and each early J_j, it is the case that $P(i,j) = \Omega(\frac{1}{xc})$. Since there are $\Omega(x)$ late processors, we have for each early J_j that

$$\sum_{\text{late } P_i} P(i,j) = \Omega\left(\frac{1}{c}\right)$$

We now apply lemma 7 to get that the expected benefit gained by the online algorithm from J_j is $\Omega(\frac{1}{c})$. Since there are $\Omega(x)$ early jobs, the expected benefit gained by the online algorithm is $\Omega(\frac{x}{c})$. Recall that $c = \Theta(\frac{\log\log m}{\log m})$ by lemma 5. ■

Corollary 11 The competitive ratio of P1R in the case of permanent faults is $O(\frac{\log m}{\log\log m})$.

Proof: This follows since the latest wake-up time for each processor in each round is 0. ■

Jobs need not be agreeable when processor wake up at different times, as is the case with transient faults. The following example illustrates this. Consider a job J_1, with $x_1 = 1.5$, $r_1 = 0$, and $d_1 = 3$, and a job J_2, with $x_2 = 1$, $r_2 = .75$ and $d_2 = 2$. The set $\{J_1, J_2\}$ of jobs is not agreeable during the interval $[0, 1]$.

2.2.3 Time Trees

To handle nonagreeable jobs we need to introduce the notion of time trees. Intuitively, time trees will be used to assign jobs to a small number of sets in such a way that all the jobs in one set will be agreeable relative to the latest wake-up times of the processors running jobs from that set.

Let $S = \{s(1), \ldots, s(\mu)\}$ be a some arbitrary collection of points in time such that $s(0) < s(1) < s(2) < \ldots < s(\mu)$. The value of μ is initially unknown to the online player, and the value of $s(i)$ may not be known until time $s(i)$. We define $E_S(a,b)$, $0 \leq b \leq \lceil \log_2(\mu+1) \rceil$ and $0 \leq a \leq \lceil \mu/2^b \rceil$, to be the set $\{s(i) : a2^b \leq i < \min(\mu, (a+1)2^b)\}$. One can think of $E_S(a,b)$ as being the ath node of height b in a tree $\mathcal{T}$ where each node is a subset of S. All the leaves are at depth $\lceil \log_2(\mu+1) \rceil$, and the ith leaf from the left at this depth is the singleton set $\{s(i)\}$. Each interior node is union of its children. Define the *time tree* of S, denoted time(S) to be the collection of all $E_S(a,b)$'s.

We now inductively define the *Forward Time Cover* $FTC_S(k)$, $1 \leq k \leq \mu$. We call this the forward time cover of $s(k)$. Let $b_\alpha b_{\alpha-1}, \ldots, b_0$ be the binary representation of k. Let i be the position of the rightmost 1, that is, $b_i = 1$ and $b_j = 0$ for all $j < i$. Then if $k + 2^i - 1 < \mu$

$$FTC_S(k) = \{\{s(k), s(k+1), \ldots, s(k+2^i-1)\}\} \cup FTC_S(k+2^i)$$

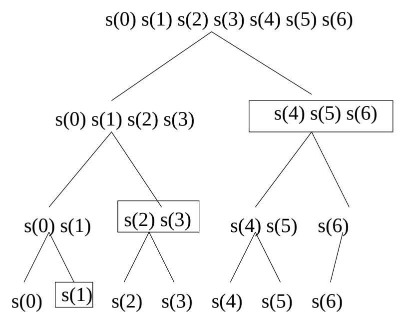

Figure 2: A Time Tree for $\mu = 6$

otherwise

$$FTC_S(k) = \{\{s(k), s(k+1), \ldots, s(\mu)\}\}$$

By convention, $FTC_S(0) = \{S\}$. Another way to think about $FTC_S(k)$ is that it is the minimum cardinality subset of time(S) whose union is $\{s(k), \ldots, s(\mu)\}$. Figure 2 shows the time tree for $\mu = 6$, with then elements of

$$FTC_S(1) = \{\{s(1)\}, \{s(2), s(3)\}, \{s(4), s(5), s(6)\}\}$$

surrounded by rectangles.

We similarly define the *backward time cover* $BTC_S(k)$ of $s(k)$, $0 \leq k \leq \mu$. Let $b_\alpha b_{\alpha-1}, \ldots, b_0$ be the binary representation of k, with $b_\alpha = 0$. Let i be the position of the rightmost 0, that is, $b_i = 0$ and $b_j = 1$ for all $j < i$. Then if $k - 2^i + 1 > 0$ then

$$BTC_S(k) = \{\{s(k - 2^i + 1), \ldots, s(k)\}\} \cup BTC_S(k - 2^i)$$

otherwise

$$BTC_S(k) = \{\{s(0), \ldots, s(k)\}\}$$

$BTC_S(k)$ is the minimum cardinality subset of time(S) whose union is $\{s(0), \ldots, s(k)\}$.

The following basic facts are readily observable.

Fact 12 For all k, $0 \leq k \leq \mu$, for all $A \in FTC_S(k)$ it is the case that $A \in \text{time}(S)$, and for all $B \in BTC_S(k)$ it is the case that $B \in \text{time}(S)$.

Fact 13 A forward time cover can be constructed online in the following sense. Let A be an element of of $FTC_S(k)$, and let the smallest element of A be $s(a)$ and the largest be $s(b)$. Then the online algorithm can determine at time $s(a)$ that the first member of A is $s(a)$, and can determine at time $s(b)$ that the last member of A is $s(b)$. Thus for each $s(i)$ one can determine at time $s(i)$ whether $s(i)$ is in A or not.

Fact 14 The cardinality of each forward time cover and each backward time cover is at most $2 + \log_2 \mu$.

Fact 15 For each time $s(i)$, there exist at most $2 + \log_2 \mu$ different $A \in \mathrm{time}(S)$ such that $s(i) \in A$.

Definition 16 For a collection S of release times, let Release(S) be the set of jobs released at those times in S. For a collection S of latest start times, let Latest(S) be the collection of jobs whose latest start times are in S.

2.2.4 No Delay

In this section we present an algorithm No-Delay that takes as input a collection $\mathcal{J}$ of jobs. No-Delay ignores the existence of all jobs not in $\mathcal{J}$. Hence, in this section we assume that $\mathcal{J}$ is the totality of all the jobs. No-Delay attempts to be competitive against all nondelayed jobs. We thus assume that the laxity of each job $J_i \in \mathcal{J}$ is zero.

Algorithm No-Delay($\mathcal{J}$): We assume that online algorithm is able to determine whether a job J_j is in $\mathcal{J}$ at time r_j. Assume that processor P_i invoked this procedure at time t. Let $R = \{r_0 \ldots, r_n\}$ be the collection of release times of jobs in $\mathcal{J}$. Let k be the least integer such that $t < r_k$ (if there is no such k, then P_i idles for the duration of the algorithm since it can not finish any job.). P_i idles until time r_k.

Intuitively, the algorithm wants to select a C uniformly at random from $FTC_R(k)$, and have P_i call P1R(Release(A)). This can be accomplished in the following manner. Let A be the first element of $FTC_R(k)$. Note that $r_k \in A$. At time r_k, P_i calls P1R(Release(A)). Now consider a time r_j that is the smallest time in some element $B \in FTC_R(k)$. Note that by fact 13 the online algorithm can determine online when a new element of the forward time cover begins. Assume that B is the sth element chronologically in $FTC_R(k)$. P_i preempts the job that it is currently running and calls P1R(Release(B)) with probability $\frac{1}{s}$. And with probability $1 - \frac{1}{s}$ continues running the job it was previously running.

End Algorithm

Lemma 17 Let $\mathcal{P}$ be a collection of processors that selected an $A \in \mathrm{time}(R)$ while running No-Delay. Then the jobs in Release(A) are agreeable over the interval of wake-up times of the processors in $\mathcal{P}$.

Proof: For each $P_i \in \mathcal{P}$ and each $J_j \in \mathrm{Release}(A)$ it is the case that $w_i < r_j$. Hence, the ranking of jobs by their earliest completion times over the interval of wake-up times of the processors in $\mathcal{P}$ is the same as ranking the jobs by their deadlines. The claim follows since the deadlines of the jobs are independent of the wake-up times of the processors. ■

Lemma 18 NO-DELAY is $O(\frac{\log n \log m}{\log\log m})$ competitive against nondelayed jobs in the One Round Problem.

Proof: Let $\tau = 2 + \log n$. Assume that the adversary finished a job J_j on a processor P_i, and let $r_{k-1} < w_i < r_k$. We define the *representative* of P_i to be the element of $FTC_R(k)$ that contains r_j. Now consider an $A \in \text{time}(R)$. Let $\mathcal{P}_A$ be the set of x processors that have A as their representative. Note that by lemma 17 the jobs in A are agreeable over the interval of wake-up times of the processors in $\mathcal{P}_A$. Let $\mathcal{J}_A$ be the x jobs in Release(A) with earliest completion relative to the wake-up time of any processor in $\mathcal{P}_A$. Note that the processors in $\mathcal{P}_A$ can collectively complete all of the jobs in $\mathcal{J}_A$. For each $J_j \in \mathcal{J}_A$ and each $P_i \in \mathcal{P}_A$, let $q_{i,j} = \frac{1}{\tau x c}$. Recall that $c = \Theta(\frac{\log m}{\log\log m})$. Each $P_i \in \mathcal{P}_A$ selected A with probability at least $\frac{1}{\tau}$. Thus by fact 9, each of the late processors in $\mathcal{P}_A$ ran each of the early jobs in $\mathcal{J}_A$ with probability $\Omega(q_{i,j})$. There are at least $\frac{x}{6}$ late processors in $\mathcal{P}_A$, and at least $\frac{x}{6}$ early jobs in $\mathcal{J}_A$. By fact 8 each late processor can complete each early job. Note that early jobs and late processors here are defined relative to $\mathcal{J}_A$.

Now for all $J_j \in \mathcal{J}$ it is the case that

$$\sum_{P_i \in \mathcal{P}_A} q_{i,j} = O(\frac{1}{c})$$

since J_j appears in at most τ elements of time(R). Hence, by lemma 7 the benefit accrued from each J_j is $\Omega(\sum_{P_i \in \mathcal{P}_A} q_{i,j})$.

Now note that

$$\sum_{P_i \in \mathcal{P}_A} \sum_{J_i \in \mathcal{J}_A} q_{i,j} \geq (\frac{x}{6})(\frac{x}{6})(\frac{1}{\tau x c}) = \Omega(\frac{x}{\tau c})$$

The result then follows by summing over all $A \in \text{time}(R)$. ∎

2.2.5 Delay

In this section we present an algorithm DELAY that takes as input a collection $\mathcal{J}$ of jobs. NO-DELAY ignores the existence of all jobs not in $\mathcal{J}$. Hence, in this section we assume that $\mathcal{J}$ is the totality of all the jobs. NO-DELAY attempts to be competitive against delayed jobs.

Algorithm DELAY$(\mathcal{J})$: We assume that online algorithm is able to determine whether a job J_j is in $\mathcal{J}$ at time r_j. Assume that processor P_i called this procedure at time t. Let $R = \{r_0 \ldots, r_n\}$ be the collection of release times of jobs in $\mathcal{J}$. Let k be the greatest integer such that $r_k < t$ (If there no such k then P_i idles for the duration of the algorithm since it can run all the jobs in $\mathcal{J}$ at their release times.)

P_i first uniformly at random selects an $A \in BTC_R(k)$. Let $S = \{\sigma(0), \ldots, \sigma(\mu)\}$ be the collection of latest start times of jobs in Release(A). Let j be the least integer such that $t < \sigma(j)$ (if there is no such j then P_i idles since it is unable to complete a job in Release(A)).

The processor P_i now uniformly at random selects an element $B \in FTC_S(j)$. Note that S is known to the online algorithm at time t so $FTC_S(j)$ can be fully constructed at time t. Finally P_i calls P1R(Latest(B)).

End Algorithm

Lemma 19 Let $\mathcal{P}$ be a collection of processors that selected an $A \in \text{time}(R)$ while running No-Delay, and then selected a B in time tree of the latest start times of the jobs in Release(A). Then the jobs in Latest(B) are agreeable over the interval of wake-up times of the processors in $\mathcal{P}$.

Proof: For each $P_i \in \mathcal{P}$ and each $J_j \in$ Latest(B) it is the case that $r_j < w_i < s_j$. Hence, the ranking of jobs by their earliest completion times over the interval of wake-up times of the processors in $\mathcal{P}$ is the same as ranking the jobs by their lengths. The claim follows since the lengths of the jobs are independent of the wake-up times of the processors. ∎

Lemma 20 Delay is $O(\frac{\log^2 n \log m}{\log\log m})$ competitive against delayed jobs in the One Round Problem.

Proof: Let $\tau = 2 + \log n$. Assume that the adversary finished a job J_j on a processor P_i, and assume $r_{k-1} < w_i < r_k$. Recall that w_i is the latest wake-up time of processor P_i. Let A be the element of $FTC_R(k)$ that contains r_j. Let $S = \{\sigma(0), \ldots, \sigma(\mu)\}$ be the collection of latest start times of jobs in Release(A), and assume $\sigma(\ell-1) < w_i < \sigma(\ell)$. We define the *representative* of J_j to be the element of $FTC_S(\ell)$ that contains s_j, the latest start time of J_j. Now consider an arbitrary representative B. Let $\mathcal{P}_B$ be the x processors that have B as their representative. Note that by lemma 19 the jobs in B are agreeable over the interval of wake-up times of the processors in $\mathcal{P}_B$. Let $\mathcal{J}_B$ be the x jobs in Latest(B) with earliest completion relative to the wake-up time of any processor in $\mathcal{P}_B$. Note that the processors in $\mathcal{P}_B$ can collectively complete all of the jobs in $\mathcal{J}_B$. For each $J_j \in \mathcal{J}_B$ and each $P_i \in \mathcal{P}_B$, let $q_{i,j} = \frac{1}{\tau^2 xc}$. Each $P_i \in \mathcal{P}_B$ selected B with probability at least $\frac{1}{\tau^2}$. Thus by fact 9, each of the at least $\frac{x}{6}$ late processors in $\mathcal{P}_B$ ran each of the at least $\frac{x}{6}$ early jobs in $\mathcal{J}_B$ with probability $\Omega(q_{i,j})$. By fact 8 each late processor can complete each early job. Once again note that earliness and lateness are defined relative to Latest(B).

Now for all $J_j \in \mathcal{J}$ it is the case that

$$\sum_{P_i \in \mathcal{P}} q_{i,j} = O(\frac{1}{c})$$

since J_j appears in at most τ^2 representatives. Hence, by lemma 7 the benefit accrued from each J_j is $\Omega(\sum_{P_i \in \mathcal{P}} q_{i,j})$.

Now note that

$$\sum_{P_i \in \mathcal{P}_B} \sum_{J_i \in \mathcal{J}_B} q_{i,j} \geq (\frac{x}{6})(\frac{x}{6})(\frac{1}{\tau^2 x c}) = \Omega(\frac{x}{\tau^2 c})$$

The result then follows by summing over all representatives. ■

3 Lower Bounds

We give lower bounds on the competitive ratio of any online scheduler that is preemptive with restart and noncheckpointing.

3.1 Unit Benefit and Almost Equal Length Case

We begin with the case that all jobs have equal benefit and almost equal length.

Lemma 21 The competitive ratio of every online randomized scheduler, which is preemptive with restart and noncheckpointing, is $\Omega(\frac{\log m}{\log \log m})$, even if all faults are permanent, and all jobs have unit benefit, $\Theta(1)$ length, and zero laxity.

Proof: Let m be an integer such that there is another integer c such that $\sum_{i=0}^{c-1} c^i = \frac{m}{2}$. Notice that $c = \Theta(\frac{\log m}{\log \log m})$. Given the online scheduling algorithm $\mathcal{A}$, we construct an instance $\mathcal{I}$ consisting of m jobs $J_1, \ldots, J_m$ that satisfy the following:

- The release times form an increasing sequence, that is, $r_i < r_j$ for all $i < j$.
- The deadlines form an increasing sequence, that is, $d_i < d_j$ for $i < j$.
- Each job has zero laxity.
- All deadlines are after the release of the last job, that is, $r_m < d_1$.

We conceptually partition the jobs into groups $G_0, \ldots G_{c-1}$, where G_i contains the c^i jobs with earliest deadline that are not in $G_0, \ldots, G_{i-1}$.

Let t be some time after r_m but before d_1. Let $p_{i,a}$ be the probability that the online algorithm was running a job from G_a on P_i at time t. For each processor P_i, it must be the case that at least $c/2$ of the G_a's satisfy $p_{i,a} \leq 2/c$. Hence, there is a group G_b for which there exists a collection $\mathcal{P}$ of at least $m/2$ processors such that for all $P_i \in \mathcal{P}$ it is the case that $p_{i,b} \leq 2/c$. The adversary will fault all processors not in $\mathcal{P}$ at time t. The processors in $\mathcal{P}$ will all fault immediately after the latest deadline of a job in G_b.

The total expected benefit earned by $\mathcal{A}$ from jobs in groups $G_0, \ldots, G_{b-1}$ is at most $\sum_{i=0}^{b-1} c^i = \frac{c^b}{(c-1)}$. The total expected benefit earned by $\mathcal{A}$ from jobs in G_b is at most

$\frac{2c^b}{c} = 2c^{b-1}$. The online algorithm can not finish any job from a group G_i where $i > b$. The result then follows since the adversary could earn a benefit of c^b by finishing a job from G_b on each processor in $\mathcal{P}$. ■

The following lemma shows that transient faults are more difficult for the online scheduler in the worst case.

Lemma 22 The competitive ratio of every online randomized scheduler, which is preemptive with restart and noncheckpointing, is $\Omega(\frac{\log^2 m}{\log^2 \log m})$ even if all jobs have unit benefit, $\Theta(1)$ length, and zero laxity.

Proof: Let c be an integer that is divisible by 8, and for which there is an integer k satisfying $16c = k^2$. Given the online scheduling algorithm $\mathcal{A}$, we construct an instance $\mathcal{I}$ consisting of jobs partitioned in k groups $G_1, \ldots, G_k$ satisfying:

- There are exactly c^i jobs in each group G_i.
- The laxity of each job is 0.
- All jobs in each G_i have same length.
- For any two jobs J_j and J_k in the same group G_i, if $r_j < r_k$ then $d_j < d_k$.
- The deadline of each job in each G_i is before the deadline of every job in G_{i+1}.
- There are c jobs in G_{i+1} released between the release of any pair of consecutively released jobs in each G_i.
- All jobs are released before the deadline of any job.
- Exactly c jobs in G_{i+1} are released before any job in G_i is released.

An example of the lower bound instance, with $k = 3$ and $c = 2$, is show in figure 3.

Hence, the total number of jobs is $n = \sum_{i=1}^{k} c^i$. The total number of processors is $m = n \cdot c^k$. Note that by solving for c we get $c = \Theta(\frac{\log^2 m}{\log^2 \log m})$. All processors are initially faulty. Exactly n processors recover from their initial fault just before the release of each $J_i \in G_k$. Let t be some time after the release of every job, but before the deadline of any job. We say a processor P_i *selects* a job J_j if P_i is running J_j at time t.

We say that a group G_i is *bad* if there exists a collection $\mathcal{P}$ of at least $3m/4$ processors such that each processor in $\mathcal{P}$ selects some job from G_i with probability at most $\frac{i}{2c}$. We claim there must be at least one bad group G_b. Otherwise, the total probability that the processors selected jobs would be at least

$$\sum_{i=1}^{k} \frac{m}{4} \cdot \frac{i}{2c} = \frac{mk(k+1)}{16c} > m$$

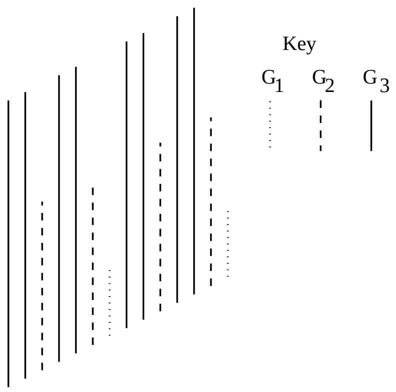

Figure 3: An Example of the Lower Bound Instance

which is a contradiction.

We now explain how the adversary can complete $\Theta(c^b)$ jobs in the group G_b while guaranteeing that the expected benefit accrued by $\mathcal{A}$ is $O(c^{b-1})$. The sum of benefits from jobs in $G_1, \ldots, G_{b-1}$ is at most $c^b/(c-1)$, and the adversary will ensure that that all processors fault on or before the latest deadline of a job in G_b. Hence, we can restrict our attention to jobs in G_b.

We partition the processors into sets $\mathcal{P}(x)$, $1 \leq x \leq c^b$. Let $\mathcal{P}(1)$ be the processors that recovered from their initial fault before the first job in G_b was released. The set $\mathcal{P}(x)$, $2 \leq x \leq c^b$, consists of the processors that recovered from their initial between the release of the $(x-1)$st and xth jobs in G_b. Note that all processors recover from their initial fault before the release of the last job in G_b. We say that a processor P_i is *bad* if the probability that P_i selected a job in G_b is at most $\frac{b}{2c}$. Let $\mathcal{BP}(x)$ be the bad processors in $\mathcal{P}(x)$. Since group G_b is bad, at least $3m/4$ processors are bad. We say that a collection $\mathcal{BP}(x)$ is *weak* if $|\mathcal{BP}(x)| \geq |\mathcal{P}(x)|/2$. Since the cardinalities of each of the $\mathcal{P}(x)$'s are the same, at least $\frac{c^b}{2}$ of the $\mathcal{BP}(x)$'s are weak. Let $\mathcal{Q} = \{\mathcal{BP}(0), \ldots, \mathcal{BP}(\frac{3c^b}{4})\}$. At least $c^b/4$ of the $\mathcal{BP}(x) \in \mathcal{Q}$ are weak.

All processors P_i not in some $\mathcal{BP}(x) \in \mathcal{Q}$ experience a permanent fault at time t. Note that each processor P_i in some $\mathcal{BP}(x) \in \mathcal{Q}$ can complete at least at least $c^b/4$ different jobs from G_b if P_i did not fault after time t.

We iteratively construct collections $\mathcal{K}_1, \mathcal{K}_2, \ldots, \mathcal{K}_f$ of jobs from G_b. The adversary will finish each job in each $\mathcal{K}_\ell$, while the online algorithm will only obtain an expected benefit of $O(\frac{|\mathcal{K}_\ell|}{c})$ from the jobs in each $\mathcal{K}_\ell$. The result then follows by noting that it will be the case that $\sum_{\ell=1}^{f} \mid \mathcal{K}_\ell \mid \geq c^b/4$.

Assume that we have already constructed $\mathcal{K}_1, \ldots, \mathcal{K}_{\ell-1}$. We now show how to construct $\mathcal{K}_\ell$. Let x be the smallest integer such that $\mathcal{BP}(x) \in \mathcal{Q}$, $\mathcal{BP}(x)$ is weak, and every processor in $\mathcal{BP}(x)$ recovered from its initial fault after the release of all jobs in $\mathcal{K}_{\ell-1}$ (if there is not such x then $f = \ell - 1$ and the construction is over). Consider the collection $\mathcal{L}$ of jobs in G_b that arrived after all processors in $\mathcal{BP}(x)$ recovered from their initial fault. We partition the $\sum_{j=1}^{b} \frac{c^j}{8}$ jobs with earliest deadlines in $\mathcal{L}$ into sets $\mathcal{H}_1(\ell), \ldots \mathcal{H}_b(\ell)$. The sets $\mathcal{H}_j(\ell)$ contains the $c^j/8$ jobs with earliest deadline that are not in $\mathcal{H}_1(\ell), \ldots, \mathcal{H}_{j-1}(\ell)$. Let $p_{i,a}$ be the probability that a processor $P_i \in \mathcal{BP}(x)$ selects some job in $\mathcal{H}_a(\ell)$. For each $P_i \in \mathcal{BP}(x)$ a counting argument shows that there are at least $\frac{b}{2}$ of the $\mathcal{H}_a(\ell)$'s such that $p_{i,a} \leq \frac{1}{c}$. Another counting argument shows that there is a group $\mathcal{H}_{z_\ell}(\ell)$ for which there exist at least $|\mathcal{BP}(x)/2|$ processors $P_i \in \mathcal{BP}(x)$ for which it is the case that $p_{i,z_\ell} \leq \frac{1}{c}$. Call a processor $P_i \in \mathcal{BP}(x)$ *low* if $p_{i,z_\ell} \leq \frac{1}{c}$. Let $\mathcal{K}_\ell = \mathcal{H}_{z_\ell}(\ell)$. Select an arbitrary collection $\mathcal{B}_\ell$ of $c^{z_\ell}/8$ low processors. The processors in $\mathcal{B}_\ell$ will experience a permanent fault just after the latest deadline of a job in $\mathcal{K}_\ell$.

All processors P_i not in some $\mathcal{B}_\ell$, $1 \leq \ell \leq f$, will experience a permanent fault at time t.

Consider some $\mathcal{K}_\ell$. The total benefit of jobs in $\mathcal{H}_1(\ell), \ldots, \mathcal{H}_{z_\ell - 1}(\ell)$ is at most

$$\sum_{i=1}^{z_\ell - 1} \frac{c^i}{8} \leq \frac{c^{z_\ell}}{8(c-1)}$$

The total expected benefit earned by $\mathcal{A}$ from jobs in $\mathcal{K}_\ell$ is at most $\frac{c^{z_\ell}}{8c}$. The adversary can earn a benefit of $c^{z_\ell}/8$ by finishing each job in $\mathcal{K}_\ell$ on the processors in $\mathcal{B}_\ell$. It is easy to see that it must be the case that $\sum_{\ell=1}^{f} \mid \mathcal{K}_\ell \mid \geq c^b/4$. ■

3.2 Arbitrary Benefit Case

We now turn our attention to the case that jobs may have arbitrary length and benefit. We first consider the following online game that will arise in our lower bound construction.

Cut-Off Game: Let c and ℓ be integers such that $c > 3$, $\ell \geq 4$, and ℓ is divisible by 4. The setting for this game consists of a processor randomly selecting a job from a collection of jobs with benefits $c^0, c^1, \ldots, c^\ell$. Let p_a be the with probability that the processor selected a job with benefit c^a. Simultaneously, the adversary selects an integer b satisfying $1 \leq b \leq \ell$. If $a < b$ the the online algorithm gets benefit c^a, otherwise, the online algorithm gets benefit 0.

The expected payoff to the processor is $W_b = \sum_{i=0}^{b-1} p_i c^i$. We call a b *bad* if $W_b \leq \frac{3c^b}{c\ell}$, otherwise b is *good*. We call a b *weak* if b is bad and $p_b \leq \frac{4}{\ell}$.

Lemma 23 In the above Cut-Off Game at least $\ell/4$ choices for b are weak.

Proof: We claim that for all i, and for all $x \geq i$, if exactly i distinct values of b in the range $[1, x]$ are good, then $\sum_{j=1}^{x-1} p_j \geq \frac{2i}{\ell}$. The proof is by induction on i. The base case,

$i = 0$, follows trivially. Now assume that the statement holds for all $k < i$. To reach a contradiction, assume that the statement is not true for i. Let x be the minimum value for which there are i good $b \in [1, x]$, but $\sum_{j=1}^{x-1} p_j < \frac{2i}{\ell}$. Let these i values of b be $b_1, \ldots, b_i$ and assume $b_1 < b_2 < \ldots < b_i$. Note that $b_i = x$ by the minimality assumption on x. Therefore $p_{x-1} \leq \sum_{j=1}^{x-1} p_j < \frac{2i}{\ell}$. By induction,

$$\forall k \in [1, i-1] \quad \sum_{j=0}^{b_k - 1} p_j \geq \frac{2k}{\ell} \tag{1}$$

Consider the problem of maximizing

$$W_x = W_{b_i} = \sum_{j=0}^{x-1} p_j c^j$$

subject to the constraints in equation 1 and $p_{x-1} < \frac{2i}{\ell}$. Obviously one can do no better than $p_{b_1 - 1} = p_{b_2 - 1} = \ldots = p_{b_{i-1} - 1} = p_{x-1} = \frac{2}{\ell}$ and all other $p_k = 0$. Therefore,

$$W_x \leq \sum_{k=1}^{i} \frac{2}{\ell} c^{b_k - 1} \leq \sum_{j=1}^{x-1} \frac{2}{\ell} c^{j-1} \leq \frac{2}{\ell} \frac{c^x}{c-1} < \frac{3c^x}{c\ell}$$

The last inequality follows because $c > 3$. Therefore x is bad, which is a contradiction.

Now consider $x = \ell$. We can now conclude that at least $\frac{\ell}{2}$ of the $b \in [1, \ell]$ are bad. A simple counting argument shows that at least $\frac{\ell}{4}$ of these bad b's are also weak. ■

Lemma 24 The competitive ratio of every online randomized scheduler, which is preemptive with restart and noncheckpointing, is $\Omega(\frac{\log \Delta\Phi}{\log\log m} \frac{\log m}{\log\log m})$ even if all jobs have zero laxity, and all faults are permanent.

Proof: We consider instances where $\Lambda = \Delta\Phi$. Let $c = \Theta(\frac{\log m}{\log\log m})$. If $\Lambda = c^{O(1)}$ then the desired lower bound follows from lemma 21. Otherwise, let $c^\ell = \Lambda$ where $\ell = \Theta(\log\Lambda / \log\log m)$. The instance consists of a collection of jobs $J_{i,b}$, $0 \leq b < \ell$ and $0 \leq i < m$, satisfying

- The benefit of each job $J_{i,b}$ is c^b.
- The release time of each J_{i_1,b_1} occurs before the release time of each of J_{i_2,b_2} if $b_1 < b_2$.
- The deadline of each J_{i_1,b_1} occurs before the deadline of each of J_{i_2,b_2} if $b_1 < b_2$.
- All jobs are released before the deadline of any job.
- The collection G_b of the m jobs with benefit c^b satisfies all the constraints of the instance constructed in the proof of lemma 21.

By lemma 23, for each processor at least $\ell/4$ groups are weak. Hence, there exists a group G_b that is weak for some collection $\mathcal{P}$ of at least $m/4$ processors. All processors not in $\mathcal{P}$ fault for the first time after the release of the last job, but before the deadline of any job. The faulting pattern for the processors in $\mathcal{P}$ are determined by the lower bound adversary in the proof of lemma 21. Assume that the adversary completes α jobs from G_b. The expected benefit earned by the online algorithm from jobs in $G_1, \ldots, G_{b-1}$ is $O(\alpha c^b/c\ell)$ since b is bad. The expected benefit earned by the online algorithm from jobs in G_b is $O(\alpha c^b/c\ell)$ by lemma 21 and since b is also weak. No jobs from any G_i, with $i \geq b$, can be completed. ■

The proof of the following lemma is analogous to the proof of the preceding lemma.

Lemma 25 The competitive ratio of every online randomized scheduler, which is preemptive with restart and noncheckpointing, is $\Omega(\frac{\log \Delta\Phi}{\log\log m}(\frac{\log m}{\log\log m})^2)$, even if all jobs have zero laxity.

4 Lower Bounds for Alternate Models

Lemma 26 The competitive ratio of every deterministic algorithm in a preemptive with restart and noncheckpointing system is $\Omega(m)$.

Proof: Assume that we have m jobs of varying lengths, zero laxity, and unit benefit that arrive at time 0. Consider that case that some processor P_i is not running the shortest job just after time 0. In this case the adversary faults all processors except P_i right after time 0, and faults P_i just after the earliest completion time of the shortest job. Hence, the adversary receives positive benefit and the online algorithm receives no benefit. If every processor is running the shortest job, then the adversary faults no processor. The online algorithm receives a benefit of 1, and the adversary receives a benefit of m. ■

Lemma 27 The competitive ratio of every randomized algorithm in a system with checkpointing and preemption is $\Omega(n)$.

Proof: There is only one processor. Job J_i, $0 \leq i \leq n-1$, has $r_i = 0$, $x_i = 1 + i/n^2$, $d_i = 1 + 2i/n^2$, $l_i = i/n^2$, and $b_i = 1$. By having a transient fault from time $1/n^2$ to time $(i+1)/n^2$, and then having a permanent fault at time $1+2i/n^2$, the adversary can guarantee that only J_i can be finished. The lower bound follows since for any online algorithm, there is a job such that the algorithm runs the job with probability at most $1/n$. ■

Lemma 28 The competitive ratio of every randomized algorithm in a system with a no preemption and no checkpointing $\Omega(n)$.

Proof: There is only one processor. The construction proceeds in stages. Stage i, $0 \leq i \leq n-1$ begins at time i/n^2. We now describe stage i. A job J_i is released at time $r_i = i/n^2$, with $x_i = 1 - 2i/n^2$, $d_i = 1 - i/n^2$, and $b_i = 1$. we say that J_i is *bad* if J_i is being run just after time i/n^2 with probability at most $1/n$. If J_i is not bad then we proceed to stage $i+1$. Note that by stage $n-1$ there must be a bad job. Now assume that J_i is bad. J_i will be the last job released. The adversary permanently faults the processor just after the deadline for J_i. The desired lower bound then follows since the adversary finished job J_i, and no job other than J_i can be finished by any algorithm. ■

References

[1] S. Baruah, J. Harita, and N. Sharma, "On-line scheduling to maximize task completions", *IEEE Real-time Systems Symposium*, 1994.

[2] S. Baruah, G. Koren, D. Mao, B. Mishra, A. Raghunathan, L. Rosier, D. Shasha, and F. Wang, "On the competitiveness of on-line real-time task scheduling", *Journal of Real-Time Systems*, **4**, 124–144, 1992.

[3] S. Goldman, J. Parwatikar, S. Suri, "On-line scheduling with hard deadlines", *Workshop on Algorithms and Data Structures*, 258–271, 1997.

[4] B. Kalyanasundaram, and K. Pruhs, "Fault-tolerant scheduling", *ACM Symposium on Theory of Computing*, 115-124, 1994.

[5] B. Kalyanasundaram and K. Pruhs "Speed is as powerful as clairvoyance", *IEEE Foundations of Computer Science*, 214–223, 1995.

[6] B. Kalyanasundaram and K. Pruhs "Randomized algorithms for real-time scheduling with fault-tolerance", Technical report, Computer Science Deptartment, University of Pittsburgh, 1994.

[7] G. Koren, and D. Shasha, "D^{over}:An optimal on-line scheduling algorithm for overloaded real-time systems", IEEE Real-time Systems Symposium, 290–299, 1992.

[8] G. Koren and D. Shasha, "MOCA: A multiprocessor on-line competitive algorithm for real-time systems scheduling", *Theoretical Computer Science*, **128**, 75–97, 1994.

[9] R. Lipton, and A. Tomkins, "Online interval scheduling", *ACM/SIAM Symposium on Discrete Algorithms*, 302–311, 1994.

[10] J. Vytopil, *Formal techniques in real-time and fault-tolerant systems*, Kluwer Academic Publishers, 1993.

[11] G. Woeginger, "On-line scheduling of jobs with fixed start and end time", *Theoretical Computer Science*, **130**, 5–16, 1994..