Non-Clairvoyant Scheduling

Rajeev Motwani[†] *Steven Phillips*[†] *Eric Torng*[†‡]

Department of Computer Science
Stanford University
Stanford, CA 94305-2140

Abstract

Virtually all research in scheduling theory has been concerned with *clairvoyant scheduling* where it is assumed that the characteristics of a job (in particular, its execution time, release time and dependence on other jobs) are known a priori. This assumption is invalid for scheduling problems that arise in time-sharing operating systems where the scheduler must provide fast turnaround for processes being generated by the users without any knowledge of the future behavior of these processes.

We study *preemptive, non-clairvoyant* scheduling schemes where the scheduler has no knowledge of the jobs' characteristics. We develop a model for evaluating scheduling strategies for single and multi-processor systems. This model compares the non-clairvoyant scheduler against the optimal clairvoyant scheduler, and it takes into account various issues such as release times, execution time, preemption cost, and the inter-dependence between jobs. Within this model we study some standard scheduling algorithms described in the systems literature, and we provide some theoretical justification for their effectiveness in practice by presenting some randomized and deterministic upper and lower bounds.

[†]Supported by NSF Grant CCR-9010517, NSF Young Investigator Award CCR-9357849, and grants from Mitsubishi Corporation and OTL.

[‡]Supported by NDSEG Fellowship.

1. Introduction

We consider the problem of preemptively scheduling jobs with unspecified resource requirements so as to minimize the *average waiting time* of jobs in the system. This *non-clairvoyant* scheduling problem is the task typically faced by an operating system in a time-sharing environment where as many users as possible should get fast responses from the system. (A *clairvoyant* scheduling algorithm needs to know the jobs' characteristics in advance, and this is typically the class of algorithms considered in classical scheduling theory [14].) While we focus on non-clairvoyance, we also address other issues relevant to operating system scheduling such as the cost of preemption, scheduling for multiple processors, and inter-dependence of jobs.

Even a cursory examination of the literature on operating system scheduling [5, 7, 13, 15, 21, 27, 28] reveals that the designers of operating systems find the standard scheduling theory to be of little relevance to this problem. In fact, systems researchers have often complained about the inapplicability of results in classical scheduling theory to their problems [28]. They have resorted to analyzing scheduling strategies using empirical testing or probabilistic analysis [13]. We provide a theoretical evaluation of scheduling strategies for operating systems without the use of such probabilistic assumptions.

Standard worst-case analysis provides little insight since for each deterministic scheduling strategy, one can construct a scheduling problem on which that strategy performs poorly. We use the method of *competitive analysis* pioneered by Sleator and Tarjan [24] to study *preemptive non-clairvoyant scheduling strategies* using the *average waiting/idle time* of a job as the performance measure. The performance of a non-clairvoyant scheduler is compared to that of an optimal clairvoyant scheduler for each set of input jobs. We provide lower and upper bounds for randomized and deterministic schedulers, and our analysis may help explain the superiority of some commonly used scheduling strategies such as *Round Robin* and *multi-level adaptive feedback policy* [8, 11].

There has been some recent work on non-clairvoyant scheduling [23, 10] that has focused on non-preemptive batch processing where the task is to minimize the *makespan* (i.e. the length) of the schedule. However, preemptions and the waiting-time measure are essential for modeling *interactive time-sharing systems* which rely on preemptive scheduling to give each user the view that they have sole access to a fast "virtual" machine. In addition non-clairvoyant scheduling should not be confused with the form of online scheduling studied by Graham [12] and Bartal *et al.* [2] where release times are unknown but the running time of a job is provided as soon as it appears in the system.

In Section 1.1 we develop our model and the related notation. Our results are described in detail in Section 1.2. A preliminary version of this paper appears as [20].

1.1. The Model and Preliminaries

We first define a generic scheduling problem; refer to the survey article by Lawler, Lenstra, Kan and Shmoys [14] for further details. A scheduling problem of *size* n consists of a collection of n independent jobs $J = \{J_1, J_2, \ldots, J_n\}$. Each job J_i has an *execution time*

x_i and a *release time* r_i, the time at which job J_i is first available for processing. We are interested in *preemptive scheduling* where the execution of any job can be suspended at any time and resumed later from the point of preemption. We assume that a context switch takes no time (except in the section where we try to minimize the number of preemptions). In general, the jobs are to be processed on a collection of p identical processors $P = \{P_1, P_2, \ldots, P_p\}$.

A *schedule* is an assignment to each job of one or more time intervals on one or more processors with the total time assigned to job J_i equaling x_i. No two time intervals may overlap on any processor, and there should be no overlap between any two time intervals assigned to any job. Our goal is to find schedules which minimize one of the following objective functions.

Definition 1: *The* **completion time** c_i *of a job* J_i *is the time at which it completes its execution. The* **total completion time** *is* $C = \sum_{i=1}^{n} c_i$*, and the* **average completion time** *is* $\widehat{C} = \frac{C}{n}$.

Definition 2: *The* **waiting time** w_i *of a job* J_i *is the amount of time which elapses between its release and completion, i.e.* $w_i = c_i - r_i$*. The* **total waiting time** *is* $W = \sum_{i=1}^{n} w_i$*, and the* **average waiting time** *is* $\widehat{W} = W/n$.

Definition 3: *The* **idle time** v_i *of a job* J_i *is the amount of time which elapses between its release and completion when it is not being processed, i.e.* $v_i = w_i - x_i$*. The* **total idle time** *is* $V = \sum_{i=1}^{n} v_i$*, and the* **average idle time** *is* $\widehat{V} = V/n$.

We classify scheduling problems into two types. A scheduling problem is said to be **static** if all the release times are 0, i.e. all the jobs are available for execution at the start of the schedule. An **dynamic** scheduling problem allows arbitrary (non-negative) release times. For static scheduling, there is no difference between the completion time and the waiting time of a job.

A **clairvoyant scheduling algorithm** may use knowledge of the jobs' execution times to assign time intervals to jobs. A **non-clairvoyant scheduling algorithm** assigns time intervals to jobs without knowing the execution times of jobs that have not yet terminated. We will show that the optimal clairvoyant algorithm only needs to know the processing times of jobs currently in the system, even in the case of dynamic scheduling; no knowledge of the existence, release time, or length of jobs to be released in the future is necessary.

For a scheduling problem J of size n, let $\widehat{W}_A(J)$ denote the average waiting time with respect to the schedule produced by an algorithm A. The **optimal clairvoyant algorithm** OPT minimizes $\widehat{W}(J)$ for each J. The **performance ratio** of a non-clairvoyant algorithm A is defined as

$$R_A(n) = \sup_{J:|J|=n} \frac{\widehat{W}_A(J)}{\widehat{W}_{OPT}(J)}$$

A non-clairvoyant scheduling algorithm is said to $f(n)$**-competitive** if $R_A(n) \leq f(n)$. The algorithm is said to be **competitive** if there exists a constant k for which it is k-competitive. If we measure the performance ratio in terms of the average idle time, we will refer to it as the **idle-performance ratio**, denoted $R_A^{\text{idle}}(n)$.

Fact 1: *For any scheduling algorithm* A, $R_A(n) \leq R_A^{\text{idle}}(n)$

We assume that the smallest possible quantum of time that can be assigned by a processor to a job is negligibly small compared to the length of the jobs being scheduled. This allows us to work with the notion of continuous time. In particular, the definition of Round Robin will make use of this feature. Clearly, such an assumption can affect our bounds by only a negligibly small amount.

1.2. Main Results

In Section 2, we consider static scheduling on a single processor. We show that *Round Robin* (one of the most commonly used algorithms in practice) has a performance ratio of $2 - \frac{2}{n+1}$ which is optimal for deterministic, non-clairvoyant algorithms. For randomized algorithms, the lower bound on the performance ratio changes only in the lower order term, and a randomized variant of Round Robin achieves this lower bound. These results easily extend to the measure of idle-performance ratio. Finally, we show that the lower bounds remain unchanged even when the jobs are of bounded sizes (i.e. the ratio of the largest to the smallest execution time is bounded by some small constant). This is an interesting case since, in practice, fast turnaround is essential only for relatively small jobs. It is surprising that a non-clairvoyant scheduler cannot use either randomness or the restriction on the job sizes to obtain a better performance ratio. In Section 3, we extend some of these results to multi-processor scheduling.

In section 4, we search for algorithms which not only have good performance ratios but also minimize the number of preemptions performed. Until this point, we have ignored this issue because modern operating systems can perform the "context switch" in negligible time. Thus, for many applications, the cost of a preemption is negligible. However, for many computationally intensive applications, preemptions can be costly because the job execution is slowed significantly while the cache is restored after each preemption.

A fundamental question is: what is the trade-off between the performance ratio and the number of preemptions? We first define how we measure how many preemptions an algorithm performs as a function of the length of a job. We show that any scheduling strategy that performs a sub-logarithmic number of preemptions for a job of length x has performance ratio $\Omega(n)$. Next we observe that Round Robin performs $\Omega(x)$ preemptions for a job of length x. In practice, the number of preemptions can be reduced by using variants of a scheme called the *multi-level adaptive feedback policy* [7, 8, 11, 21, 27]. We prove that some very simple versions of this algorithm, the *geometric* algorithms, have near-optimal performance ratios while using only a logarithmic number of preemptions. A randomized member of the geometric algorithm family is shown to have essentially the same performance ratio as Round Robin. This may help explain why the multi-level adaptive feedback policy performs well in practice [8, 11].

Section 5 is devoted to dynamic scheduling. We show that clairvoyance can make a huge difference in a scheduler's ability to keep a system responsive. Any deterministic non-clairvoyant algorithm has performance ratio $\Omega(n^{1/3})$. while any randomized, non-clairvoyant algorithm has performance ratio $\Omega(\log n)$. The proofs of these lower bounds use jobs of unbounded length. We also study the performance of algorithms for the natural case where the job lengths are bounded. If the ratio of the longest to the shortest job is bounded by

k, then any deterministic non-clairvoyant algorithm has performance ratio at least k (for large n), and a trivial algorithm achieves this bound.

Finally, in Section 6, we consider a version of the dining/drinking philosophers problem where some pairs of jobs cannot be executed simultaneously due to a conflict in their resource requirements. We demonstrate a competitive strategy for minimizing the makespan and exhibit nearly tight lower bounds. We leave open the problem of finding a good scheduler for minimizing average completion time for jobs with dependencies.

2. Static Scheduling for Single Processors

In this section we analyze the situation where all jobs are released at time 0. In this case, total completion time, average completion time, total waiting time, and average waiting time are all equivalent measures for comparing scheduling algorithms, i.e. the performance ratio of a scheduling algorithm is identical with respect to all four performance measures. For ease of description, we will use the total completion time performance measure instead of the average waiting time performance measure until we study dynamic scheduling.

In the first subsection, we describe the optimal clairvoyant algorithm OPT for static scheduling. In the next two subsections, we prove matching upper and lower bounds on the performance ratios of deterministic and randomized non-clairvoyant algorithms for this problem. In the final subsection, we prove surprising lower bounds for the case where job sizes are bounded.

2.1. The Optimal Clairvoyant Algorithm

Before we can determine the performance ratios of non-clairvoyant algorithms, we first must describe the optimal clairvoyant algorithm OPT.

The completion time of a specific job can be broken down into two components: the running time of the individual job itself and the time it is delayed by the execution of other jobs before it finishes. To make the second term more tractable, we define the pairwise delay of two jobs J_i and J_j with respect to an algorithm A as follows:

Definition 4: *$D^A_{i,j}$ is the time allocated by algorithm A to job J_i before job J_j completes. In other words, this is the amount of time by which job J_i delays job J_j.*

Definition 5: *The* pairwise *delay $P^A_{i,j} = D^A_{i,j} + D^A_{j,i}$ is the amount by which two jobs J_i and J_j delay each other.*

We can now write the total completion time for an algorithm A on an instance J as

$$C_A(J) = \sum_{i=1}^{n} x_i \quad + \quad \sum_{1 \leq i < j \leq n} P^A_{i,j}$$

Since the first term is identical for all algorithms, OPT must minimize the second term. The minimum value of $P^A_{i,j}$ for any pair of jobs J_i and J_j is $\min(x_i, x_j)$, and this is achieved if the shorter of two jobs J_i and J_j is executed completely before the longer job begins execution. Hence OPT runs the jobs sequentially in non-decreasing order of job length. Therefore, for $x_1 \le x_2 \le \cdots \le x_n$,

$$\begin{aligned} C_{OPT} &= \sum_{i=1}^{n} x_i + \sum_{1 \le i < j \le n} min(x_i, x_j) && (1) \\ &= \sum_{i=1}^{n} x_i + \sum_{x=1}^{n} (n-i)x_i && (2) \\ &= \sum_{i=1}^{n} (n-i+1)x_i && (3) \end{aligned}$$

This is a special case of Smith's rule [25] for scheduling weighted jobs, and it is sometimes referred to as the SPT (shortest processing time first) algorithm [14].

2.2. Deterministic Algorithms

We now show that the best possible performance ratio achievable by a *deterministic* algorithm is $2 - \frac{2}{n+1}$ and that it is achieved by the well-known Round Robin (RR) algorithm.

Theorem 1: *For the static scheduling problem, every deterministic, non-clairvoyant algorithm has* $R_A(n) \ge 2 - \frac{2}{n+1}$ *and* $R_A^{idle}(n) \ge 2$.

Proof: Fix any deterministic non-clairvoyant algorithm A and execute the schedule produced by A for 1 time unit assuming no job finishes. Let e_i equal the amount of time A spent executing job J_i. The instance J is defined by choosing x_i to be $e_i + \epsilon$ for arbitrarily small ϵ. Each job runs after time 1 for a negligible amount of time, so has completion time 1. Summing over the n jobs gives

$$C_A(J) = n.$$

Let us now compare this to the completion time of these jobs when scheduled by OPT. Rename the jobs so that $x_1 \le x_2 \le \cdots \le x_n$; Equation 3 gives us

$$C_{OPT}(J) = (n-i+1)\sum_{i=1}^{n} x_i.$$

This expression is maximized (subject to the constraints that $\Sigma_{i=1}^{n} x_i = 1$ and that the x_i are in non-decreasing order) when $x_1 = x_2 = \ldots = x_n = \frac{1}{n}$. Hence $C_{OPT}(J) \le \frac{n+1}{2}$, so $R_A(J) \ge \frac{n}{\frac{n+1}{2}} = 2 - \frac{2}{n+1}$. The same proof ignoring the sum of execution times yields the idle-competitive lower bound. □

Notice that the worst case for OPT in the above proof is when the algorithm A divides each unit of processing time equally between all currently uncompleted jobs. This is exactly the algorithm which is referred to as the Round Robin algorithm in the literature on operating systems scheduling [5, 7, 21, 27].

Definition 6: *The Round Robin (RR) algorithm is a non-clairvoyant scheduling algorithm which at all times ensures that each uncompleted job has received an equal amount of processing time.*

Recall that in practice the RR algorithm cycles through the list of jobs, giving each job τ units of processing time in turn. For small enough τ, this is equivalent to the above definition.

We have seen that for the case where all the jobs have the same running time, RR achieves the optimal performance ratio of $2 - \frac{2}{n+1}$. In fact, this is the worst case scenario for RR which gives us a tight bound for the deterministic performance ratio.

Theorem 2: *For the static scheduling problem, Round Robin (RR) is exactly 2-idle-competitive and $(2 - \frac{2}{n+1})$-competitive.*

Proof: Fix a scheduling instance J. At any point in time, RR ensures that all active jobs have received an identical amount of processing time, which means that all pairs of active jobs have delayed each other by the same amount of time. Moreover, this is true for all pairs of jobs when the first one of the two jobs completes. Therefore, for any pair of jobs J_i and J_j, $P_{i,j}^{RR} = 2 \times \min(x_i, x_j) = 2 \times P_{i,j}^{OPT}$, so RR is exactly 2-idle-competitive.

Rename the jobs so that $x_1 \leq x_2 \leq \cdots \leq x_n$. Totalling the pairwise delays and the total running times gives

$$C_{RR}(J) = 2\Sigma_{i=1}^{n}(n-i)x_i + \Sigma_{i=1}^{n} x_i$$

and

$$C_{OPT}(J) = \Sigma_{i=1}^{n}(n-i)x_i + \Sigma_{i=1}^{n} x_i.$$

For a fixed value of $\Sigma_{i=1}^{n} x_i$, the ratio $C_{RR}(J)/C_{OPT}(J)$ is maximized when $\Sigma_{i=1}^{n}(n-i)x_i$ is maximized which is when the n jobs are all of equal length. We have already observed that in the case where all the jobs are of equal length, RR's performance matches the lower bound. Therefore, RR is $(2 - \frac{2}{n+1})$-competitive. $\Box$

2.3. Randomized Algorithms

A natural question is whether randomization allows us to improve upon the lower bound of 2 on the performance ratio. We show that there is a simple randomized algorithm which improves very slightly upon RR, but its performance ratio is still asymptotically 2. We then prove a matching lower bound on the performance ratio of any randomized algorithm.

Definition 7: *The algorithm $RAND$ behaves as follows on an instance of size n. With probability $1 - \frac{2}{n+3}$, it acts like RR. With probability $\frac{2}{n+3}$, it randomly orders the jobs and runs each job to completion in that order.*

We will refer to the run-to-completion algorithm as RTC.

Theorem 3: *$RAND$ is $(2 - \frac{4}{n+3})$-competitive.*

Proof: Fix a scheduling instance J with job sizes $x_1, \ldots, x_n$. As shown earlier, the total completion time for RR is the following.

$$C_{RR}(J) = \sum_{i=1}^{n} x_i \quad + \quad 2 \sum_{1 \leq i < j \leq n} \min\{x_i, x_j\}$$

For randomly ordered jobs, the run-to-completion algorithm has the following expected total completion time. This is because $P_{i,j}^{RTC}$ is equally likely to be x_i or x_j.

$$E[C_{RTC}(J)] = \sum_{i=1}^{n} x_i \quad + \sum_{1\leq i<j\leq n} \frac{x_i + x_j}{2} \quad = \quad \frac{n+1}{2}\sum_{i=1}^{n} x_i$$

Thus when $RAND$ runs RR with probability $1 - \frac{2}{n+3}$, and run to completion with the remaining probability, the expected total completion time is

$$\begin{aligned} C_{RAND}(J) &= \left(1 - \frac{2}{n+3} + \frac{n+1}{n+3}\right)\sum_{i=1}^{n} x_i \quad + \quad 2\left(1 - \frac{2}{n+3}\right) \sum_{1\leq i<j\leq n} \min\{x_i, x_j\} \\ &= \left(2 - \frac{4}{n+3}\right)\left(\sum_{i=1}^{n} x_i \quad + \sum_{1\leq i<j\leq n} \min\{x_i, x_j\}\right) \\ &= \left(2 - \frac{4}{n+3}\right) \; C_{OPT}(J) \end{aligned}$$

$\square$

We now prove a matching lower bound for randomized algorithms.

Theorem 4: *Any randomized algorithm has* $R_A(n) \geq 2 - \frac{4}{n+3}$ *and* $R_A^{idle}(n) \geq 2$.

Proof: We use Yao's [30] method (a variant of the von-Neumann minimax principle) by first proving a lower bound on the expected performance ratio of any deterministic algorithm on problem instances chosen from a specific probability distribution, then showing that the same lower bound holds for all randomized algorithms.

Let the n jobs have sizes which are independently chosen from the exponential distribution. The probability that a job has size greater than t is $G[t] = e^{-t}$. Fix a non-idling, deterministic algorithm A and consider two jobs J_i and J_j. Let $i(t)$ denote the amount of time A has spent on J_i after a total of t time units have been spent on jobs J_i and J_j assuming neither J_i nor J_j has completed yet. Clearly $0 \leq i(t) \leq t$.

The probability that both jobs J_i and J_j are still running after exactly t time units have been allocated to the two jobs is exactly $G[i(t)]G[t - i(t)] = e^{-t}$. We conclude that

$$E[P_{i,j}^{A}] = \int_0^\infty G[i(t)]G[t - i(t)]\mathrm{d}t = \int_0^\infty e^{-t}\mathrm{d}t = 1$$

Thus for any algorithm A, the expected total completion time can be computed as follows.

$$E[C_A] = \sum_{i=1}^{n} E[x_i] \quad + \sum_{1\leq i<j\leq n} E[P_{i,j}^{A}] \quad = \quad n + \binom{n}{2}$$

Note that A's total completion time could be greater if A ever idles the processor while unfinished jobs are available.

For the optimal clairvoyant algorithm OPT, the pairwise delay of two jobs J_i and J_j can be computed as follows.

$$E[P_{i,j}^{OPT}] = E[\min\{x_i, x_j\}] = \int_0^\infty G[t]^2\mathrm{d}t = \int_0^\infty e^{-2t}\mathrm{d}t = \frac{1}{2}$$

Hence by linearity of expectation,

$$E[C_{OPT}] = \sum_{i=1}^{n} E[x_i] \quad + \sum_{1\le i<j\le n} E[P_{i,j}^{OPT}] \quad = \quad n + \frac{1}{2}\binom{n}{2}$$

Therefore

$$\frac{E[C_A]}{E[C_{OPT}]} \ge \frac{n+\binom{n}{2}}{n+\frac{1}{2}\binom{n}{2}} = 2 - \frac{4}{n+3}$$

To see that this implies the lower bound for all randomized algorithms, consider a randomized algorithm B, which can be defined as a probability distribution on deterministic algorithms A. Let the random variable I be drawn from the distribution on scheduling instances described above. We have

$$\frac{E_{A,I}[C_A]}{E_I[C_{OPT}]} \ge 2 - \frac{4}{n+3}$$

Therefore

$$\frac{E_I[E_A[C_A]]}{E_I[C_{OPT}]} \ge 2 - \frac{4}{n+3}$$

so there must be some I for which

$$\frac{E_A[C_A]}{C_{OPT}} \ge 2 - \frac{4}{n+3}$$

and hence $R_B(n) \ge 2 - \frac{4}{n+3}$. Again, the same proof holds for the idle-competitive lower bound when the sum of the execution times term is ignored. □

2.4. Bounded Job Sizes

The lower bounds in the previous section use instances with arbitrarily large jobs. In some practical situations we might know in advance that the size of all jobs is within a specific range. Surprisingly, a non-clairvoyant scheduler cannot use this extra information to get a better performance ratio. Theorem 1 and a slightly weaker version of Theorem 4 still hold even when the job sizes are restricted to be within fixed bounds.

Theorem 5: *For static scheduling, any deterministic, non-clairvoyant algorithm A has $R_A(n) \ge 2 - \frac{2}{n+1}$ and $R_A^{\text{idle}}(n) \ge 2$, even when all jobs have length between 1 and 5.*

Proof: Let A be a deterministic non-clairvoyant algorithm, and let k be the maximum job size ($k = 5$ in the statement of the theorem, though we leave it variable for now). We run A on n jobs for t time units. Let e_i equal the amount of time A spent executing job i. Define x_i as follows:

$$x_i = \begin{cases} e_i + \epsilon & \text{if } 1 \le e_i < k \\ k & \text{if } e_i = k \\ 1 & \text{otherwise} \end{cases}$$

with ϵ arbitrarily small. The time t is chosen so that $\sum_{1\le i\le n} e_i = \alpha n k$. The parameter α will be determined later.

Note that the best case for A is when at most one job has $0 < e_i < 1$ (since if $0 < e_i \le e_j < 1$, then modifying A so that the time spent on job i is moved to job j, until $e_i = 0$ or

$e_j = 1$, cannot increase the non-clairvoyant cost, and the clairvoyant cost remains the same). Finally, we can assume that all the e_i such that $1 \le e_i < k$ are equal (as this is the worst case for the clairvoyant scheduler, and doesn't affect A's cost).

To summarize, the n jobs are now partitioned into b jobs with $e_i = k$, $p-1$ jobs with $e_i = 0$, one job with $e_i = \epsilon$ ($0 \le \epsilon < 1$), and $n-b-p$ jobs with $e_i = h$ ($1 \le h < k$), and $bk + (n-b-p)h + \epsilon = \alpha nk$. The non-clairvoyant total completion time for the jobs $\{x_i\}$ is minimized if the jobs with $e_i = k$ were executed first, and the jobs with $e_i = 0$ are executed last. In this case the non-clairvoyant cost is

$$\begin{aligned} C_A &= k\frac{b(2n-b+1)}{2} + h(n-b-p)(n-b) + \frac{p(p-1)}{2} + (1-\epsilon)p & (4) \\ &\ge k\frac{b(2n-b+1)}{2} + k(\alpha n - b)(n-b) + \frac{p(p-1)}{2}, & (5) \end{aligned}$$

while the clairvoyant cost for the jobs $\{x_i\}$ is

$$\begin{aligned} C_{OPT} &= k\frac{b(b+1)}{2} + h\frac{(n-b-p)(n-p+b+1)}{2} + \frac{p(2n-p+1)}{2} & (6) \\ &\le k\frac{b(b+1)}{2} + k\frac{(\alpha n-b)(n-p+b+1)}{2} + \frac{p(2n-p+1)}{2} & (7) \end{aligned}$$

The inequality 5 follows from $\epsilon < 1$, while inequality 7 follows from $\epsilon \ge 0$. Now consider the expression $\frac{C_A}{C_{OPT}} - \frac{2n}{n+1}$. It is sufficient for our purposes to find values for k and α so that for all legal b and p, this expression is non-negative. Its sign is the sign of

$$g(\alpha, k, n, b, p) = (n+1)C_A - 2nC_{OPT}.$$

Substituting $\alpha = 2/5, k = 5$, we get

$$g(2/5, 5, n, b, p) = (5b + 5b^2 + bn + 5b^2n + 2bn^2 + p - np - 10bnp + p^2 + 3np^2)/2 - (n+1)p. \quad (8)$$

Note that $n \ge 5b/2$, so

$$2g(2/5, 5, n, b, p) \ge n(10b^2 - 10bp + 3p^2 - 2p). \quad (9)$$

Minimizing this expression with respect to b gives $n(-2p + p^2/2)$, which is non-negative for integral values of p other than 1,2 and 3. Hence we need only to check the cases $p = 1, 2$, or 3. Substituting these values into the above equations (8) and (9) gives expressions which are positive at all integral values of b. □

Note that $k = 5$ was chosen to make the above proof tractable; it is not the smallest possible value for which the above theorem holds. However, the technique used in the proof breaks down for $k = 4$.

We can also show that when the maximum ratio between job sizes is bounded, no randomized algorithm can have a competitive ratio much better than 2.

Theorem 6: *There is a function f such that for $\epsilon > 0$, any randomized algorithm A has $R_A(n) \ge 2 - \epsilon - o(1)$ when the maximum job ratio is at most $f(\epsilon)$.*

Proof: Assume that algorithm A has $R_A(n) < 2 - \epsilon - o(1)$ whenever the maximum job ratio is bounded. We show that this assumption implies the existence of an algorithm A' for the unbounded case with a performance ratio better than $2 - \frac{4}{n+3}$, the lower bound from Theorem 4. Consider the distribution D on static scheduling instances considered there: n jobs are drawn independently from the exponential distribution. Define the distribution D' by taking an instance from D, then rounding the jobs to fit into the interval $[\delta, k]$.

Assume that the algorithm A has performance ratio better than $2 - \epsilon + o(1)$ on D'. Consider the algorithm A' which runs algorithm A assuming that all jobs have size at most k, then afterwards finishes all the jobs with size greater than k. Clearly

$$E_D[C_{A'}] \leq E_{D'}[C_A] + n^2 e^{-k}(k+1),$$

where the last term is an overestimate of the contribution of the jobs larger than k to the total completion time. In addition,

$$E_D[C_{OPT}] \geq E_{D'}[C_{OPT}] - n^2 \delta (1 - e^{-\delta}),$$

where the last term is an overestimate of the component of the completion time saved by OPT by not having jobs smaller than δ rounded up to δ. Since $E_D[C_{OPT}] \geq n^2/4$, for large k and small δ the performance ratio of A' on D is at most $2 - \epsilon' + o(1)$ for some constant ϵ'. This contradicts Theorem 4. □

3. Static Multi-Processor Scheduling

We now consider static scheduling for multiple identical processors. In practice, there may be communication costs associated with transferring a job from one processor to another, but we assume that any job can be assigned to any processor instantaneously. We define the performance ratio for a scheduling problem involving p processors as follows; the notion of competitiveness generalizes in the obvious manner.

Definition 8: *Let $C_A(p, J)$ denote the total completion time of an instance J when scheduled by the algorithm A on p processors. The performance ratio of a non-clairvoyant algorithm for static scheduling is*

$$R_A(p, n) = \sup_{J:|J|=n} \frac{C_A(p, J)}{C_{OPT}(p, J)}$$

The results and algorithms from the static, deterministic, single processor model generalize to the static, deterministic, multi-processor model. For example, McNaughton's [19] results imply that OPT still obeys the SPT (shortest processing time) rule of Conway, Maxwell and Miller [6]; this rule is to assign the jobs in order of non-decreasing processing time to the earliest available processor. We prove that Round Robin still is the best possible deterministic, non-clairvoyant algorithm for multi-processor scheduling. Observe that the performance ratio we derive is identical to the single processor case when $p = 1$. We will assume that $n > p$, since otherwise RR and OPT are identical.

Theorem 7: *For all $1 \leq p \leq n$, the Round Robin algorithm for static scheduling on p processors is $\left(2 - \frac{2p}{n+p}\right)$-competitive and 2-idle-competitive.*

Proof: We start by defining a partition of the jobs in J into groups of consecutive jobs. Note that the jobs are now being numbered in the order of non-increasing processing times.

Definition 9: *Let J be any instance of size n with the execution times of the jobs being such that $x_1 \geq x_2 \geq \ldots \geq x_n$. For $1 \leq k \leq \lceil n/p \rceil$, define the group $G(k)$ of jobs to be*

$$G(k) = \{J_i \,|\, (k-1)p < i \leq kp\}$$

All groups, except possibly the last one, consist of p consecutive jobs.

We claim the following for the non-preemptive schedule produced by OPT – the set of p jobs which are the k'th last jobs to be scheduled on the p processors is exactly the set of jobs in $G(k)$. In particular, the set $G(1)$ consists of the last job to be scheduled on each of the p processors. To verify this, number the processors from 1 through p, in the order in which OPT first assigns a job to them. An easy inductive argument shows that the SPT algorithm cycles through the processors in this order, giving each the smallest length job currently unscheduled.

A job in group $G(k)$ has exactly $k-1$ jobs scheduled after it on its processor. Thus, it contributes to the completion time of exactly k jobs. We obtain the following expression for the total completion time of the algorithm OPT.

$$C_{OPT}(p, J) = \sum_{k=1}^{\lceil n/p \rceil} \sum_{J_i \in G(k)} k x_i$$

Consider now the total completion time of RR. Let c_i denote the completion time of the i'th largest job.

Lemma 1: *For the Round Robin algorithm, $c_i = c_{i+1} + (i/p)(x_i - x_{i+1})$ for $i \geq p$ while $c_i = c_{p+1} + x_i - x_{p+1}$ for $i < p$.*

Proof: For $i \geq p$, at the time of completion of the job J_{i+1}, all i remaining jobs have received exactly x_{i+1} processing. By the time J_i completes, each of the i jobs $J_1 \cdots J_i$ has received an additional $x_i - x_{i+1}$ processing time which means a total of $i(x_i - x_{i+1})$ work is done between time c_{i+1} and time c_i. Since there are p processors, this work can be done in $(i/p)(x_i - x_{i+1})$ time and the claim follows. Similar arguments show that for $i < p$, $c_i = c_{p+1} + x_i - x_{p+1}$. □

Eliminating the recurrence, we get for $i \geq p$

$$c_i = \frac{i}{p} x_i + \frac{1}{p} \sum_{j=i+1}^{n} x_j$$

and for $i < p$

$$c_i = x_i + \frac{1}{p} \sum_{j=p+1}^{n} x_j.$$

A simple calculation now establishes the following expression for the total completion time of RR.

$$C_{RR}(p, J) = \sum_{i=1}^{p} x_i \quad + \quad \sum_{i=p+1}^{n} \frac{2i-1}{p} x_i$$

We claim that the ratio of C_{RR} to C_{OPT} is maximized when all the jobs in the same group have the same processing times. To justify this, observe that for OPT, the coefficient of x_i's contribution to the total completion time is determined solely by its group index. On the other hand, for RR, the coefficient is smaller for the longer jobs within a specific group. Thus, reducing value of each x_i to be the same as the smallest processing time of any job in its group can only increase the ratio. By a similar argument, it can be seen that the worst-case ratio is achieved when n is a multiple of p.

Assume now that each group contains exactly p jobs of the same length. Let y_k denote the (common) processing time requirement of any job in the group $G(k)$. Then, a simple calculation shows that

$$C_{OPT} = \sum_{k=1}^{n/p} pky_k$$

$$C_{RR} = \sum_{k=1}^{n/p} p(2k-1)y_k$$

Once again, the ratio is maximized when all the y_k are equal, implying that the ratio is exactly $2 - \frac{2p}{n+p}$. The idle-competitive ratio can also be derived from the above expressions. □

A generalization of the lower bound argument for single processor scheduling now shows that RR is the optimal non-clairvoyant algorithm.

Theorem 8: *For static scheduling on p processors, any deterministic, non-clairvoyant algorithm A has* $R_A(p,n) \geq 2 - \frac{2p}{n+p}$ *and* $R_A^{\text{idle}}(p,n) \geq 2$.

Proof: As before, we use an adversary strategy. Fix any deterministic non-clairvoyant algorithm A, and let it run for one time unit, ensuring that no job completes during that time. Let e_i denote the amount of processing received by job i, and make $x_i = e_i + \epsilon$ for a arbitrarily small ϵ. Clearly, the total completion time for A is n, since all jobs finish immediately after time 1.

The total completion time for the optimal clairvoyant algorithm is

$$C_{OPT}(p,J) = \sum_{k=1}^{\lceil n/p \rceil} \sum_{J_i \in G(k)} kx_i$$

This expression is maximized, subject to the constraint that all $e_i \geq 0$ and $\sum_i e_i = p$, when the value of each e_i is equal to p/n. In that case, the competitive ratio of A is exactly $2 - \frac{2p}{n+p}$. A similar argument yields the bound on the idle-competitive ratio. □

4. Minimizing Preemptions

In this section, we study algorithms which minimize both the waiting time of a job and the number of preemptions performed on a job. Until now, we have ignored the cost of performing preemptions. This is a valid abstraction for many applications because modern operating systems can perform a "context switch" in negligible time. However, for some applications, the cost of a preemption is noticeable. For example, when a job has been preempted, the contents of the cache are not restored when it resumes execution. For many computationally intensive jobs, the time required to rebuild the cache is significant. Until the cache is rebuilt, the execution of the job can be slowed significantly.

The fundamental question we study in this section is: what is the trade-off between the performance ratio of an algorithm and the number of preemptions performed by the algorithm? We first define how we measure how many preemptions an algorithm performs. We establish a minimum number of preemptions p that any competitive algorithm must perform on a given job in the worst-case. We then observe that while round robin is optimal with respect to waiting time, it may perform $O(2^p)$ preemptions on some jobs. Finally, we introduce the class of *geometric algorithms*. Each member of this family of algorithms has constant performance ratio and performs at most $O(p)$ preemptions on any job.

This is particularly interesting because the family of geometric algorithms is closely related to a commonly used scheduling strategy, *multi-level adaptive feedback policy* (MLAF) [7, 8, 11, 21, 27]. Although we don't consider some issues (such as the difference between I/O bound and computationally intensive jobs) relevant to the efficiency of MLAF, when a set of computationally intensive jobs is released simultaneously, MLAF reduces to a member of the geometric algorithm family. This helps explain why MLAF performs so well in practice [8, 11].

4.1. Measuring the Number of Preemptions

Earlier, we assumed that the smallest job was much larger than the quantum of time assigned to any job during any preemptive schedule. We now assume that the smallest job is at least as large as the smallest quantum of time assigned to any job during a preemptive schedule and define this minimum job size to be 1.

We measure the number of preemptions performed by an algorithm as a function of the length of a job. For an algorithm A, a job length x, and a problem instance J, we define $PM_A^J(x)$ to be the maximum number of preemptions A performs on any of job of length x in J. If J contains no jobs of length x, $PM_A^J(x) = 0$. We then define

$$PM_A(x) = \sup_J PM_A^J(x).$$

4.2. Number of Preemptions versus Performance Ratio

The following theorem shows that no algorithm that performs a sub-logarithmic number of preemptions can have a constant performance ratio.

Theorem 9: *For any deterministic, non-clairvoyant algorithm A, if $PM_A(x) = o(\log x)$, then $R_A(n) = \Omega(n)$.*

Proof: Fix any deterministic, non-clairvoyant scheduling strategy A for which $PM_A(x) = o(\log x)$. We will show that there exists an input instance J' with one long job and $n-1$ much shorter jobs where A finishes the long job before any of the short jobs. Suppose the long job is $l \geq n^2$ times as long as the second longest job which has length p. Rename the jobs so that $x_1 \leq x_2 \cdots \leq x_n$. $C_A(J') \geq nlp$ since the long job finishes before any of shorter jobs. Note we are only considering the running time of the long job and the amount of time it delays each of the shorter jobs. $C_{OPT}(J') = \sum_{i=1}^{n} x_i + \sum_{1 \leq i < j \leq n} P_{i,j}^{A}$. This quantity is maximized if we assume each short job has length p. With this assumption, $C_{OPT}(J') = (n-1)p + lp + (n-1)\frac{n}{2}p \leq 2lp$. The inequality holds due to the assumption that $l \geq n^2$. Therefore, $R_A(n) \geq \frac{C_A(J')}{C_{OPT}(J')} \geq \frac{nlp}{2lp} = \frac{n}{2}$.

We will construct J' by first describing an instance J and showing that J can be modified into J'. Fix the number of jobs in J to be n. Choose m so that $m^{\frac{1}{n}} \geq 2n^2$. Since $PM_A(x) = o(\log x)$, $\log_m x > PM_A(x)$ for all $x \geq x_0$ for some value of x_0. This means that if some job J_i runs for $x > x_0$ time, the number of preemptions A performs on J_i is at most $\log_m x$.

Define J to be n jobs each of length x_0. Execute the schedule produced by A on J until time t when the first job terminates. Since $PM_A(x)$ is monotonic, we can bound the number of times each job has been preempted by $\log_m x_0$. Therefore, at time t, the total number of times all jobs have been preempted is at most $n \log_m x_0 \leq \log_{2n^2} x_0$. Hence there is some time t' ($t' \leq t$) when a job preemption occurs at least $2n^2$ times as late as all previous job preemptions. Thus at time t', some job has been executed for at least n^2 times as much time as the second largest job.

We create the modified instance J' as follows. Let e_i be the amount of execution time each job J_i has received before time t'. The job J_i that received the most execution time will be defined to have a running time of exactly e_i. All the other jobs J_i are defined to have running times $x_i = e_i + \epsilon$ with ϵ arbitrarily small. Thus we have created an input instance J' where the performance ratio is at least $\frac{n}{2}$. $\square$

Now consider the RR algorithm which has an optimal performance ratio.

Lemma 2: *$PM_{RR}(x) = \Omega(x)$.*

Proof: On an instance with 2 jobs of size x, the number of preemptions performed by RR increases linearly with the job length x. $\square$

We can modify RR to reduce the number of preemptions performed by increasing the time slice offered to each job in each successive round. If we increase the time slice in each round by a constant additive factor (i.e. increasing the time slice by 1 unit each round), this only reduces the number of preemptions performed on a job from $\Omega(x)$ to $O(\sqrt{x})$. However, if we increase the time slice in each round by a constant multiplicative factor (e.g. doubling the time slice each round), this reduces the number of preemptions performed on a job from $\Omega(x)$ to $O(\log\ x)$, the best we can hope for in any competitive algorithm. We call any algorithm which increases the time slice in each round by a multiplicative factor a *geometric algorithm.*

4.3. Deterministic Geometric Algorithms

We first study the kind of geometric algorithms which are used in practice – deterministic geometric algorithms. For any $r > 1$, the geometric algorithm G_r is defined as follows.

Definition 10: *G_r works in rounds. In the k'th round ($k \geq 0$), G_r cycles through the jobs giving each job a time slice of t_k units of processing time. The sum of time slices until the end of the k'th round is r^k. If a job completes within a time slice, G_r immediately moves on to the next job.*

Note that the time slices satisfy $t_0 = 1, t_1 = r - 1$, and $t_k = rt_{k-1}$ for $k > 1$.

RR and RTC fit naturally around the family of geometric algorithms G_r with RR corresponding to G_1 and RTC corresponding to G_∞.

Theorem 10: $PM_{G_r}(x) = O(\log_r x)$.

Proof: Since the time slice each job receives is multiplied by a factor of r each round, a job length x will be preempted at most $\lceil \log_r x \rceil$ times. □

Theorem 11: *For any fixed ratio $r > 1$, $R^{idle}_{G_r}(n) = (1 + r)$. Therefore G_r is $(1 + r)$-competitive.*

Proof: We consider only the pairwise delay component of total completion time. Consider the pair of jobs J_i and J_j with $x_i \leq x_j$. Since G_r never spends more than rx_i time executing job J_j before finishing job J_i, we have $P^{G_r}_{i,j} \leq (1+r)x_i = (1+r)\min\{x_i, x_j\} = (1+r)P^{OPT}_{i,j}$. It follows that $R^{idle}_{G_r}(n) \leq (1 + r)$.

To see that this upper bound is tight, consider the instance where the n job sizes are $1+\epsilon$, $r + \epsilon$, $r^2 + \epsilon$, …, $r^{n-1} + \epsilon$ with ϵ arbitrarily small. If, in each round, G_r executes the remaining jobs in non-increasing order of execution time, the pairwise delay for each pair of jobs J_i and J_j, $x_i < x_j$, will be $(1 + r)x_i$ whereas for OPT the pairwise delay will only be x_i. The upper bound on the performance ratio of G_r follows from Fact 1. □

The upper bound on the performance ratio may not be tight because we have ignored the sum of execution times term in the the total completion time. In the example in the proof, the sum of the execution times is significant when compared to the pairwise delays. To find a good lower bound on the performance ratio, we need to find an instance which maximizes the pairwise delays between jobs of different sizes relative to the pairwise delays between jobs of roughly equal size while keeping the sum of the execution times insignificant.

The instances we have found that provide the best lower bound for $R_{G_r}(n)$ are the following:

Definition 11: *Choose $m \gg k$ with both m and k very large. Let $J^i_{r,s}$ to be $\frac{m}{s^i}$ jobs of size $r^i + \epsilon$. Define $\mathrm{J_{r,s}} = \cup_{i=0}^{k-1} J^i_{r,s}$.*

We can prove that G_r is asymptotically $\Theta(\sqrt{r})$-competitive on $J_{r,\sqrt{r}}$ and that $J_{r,\sqrt{r}}$ is the worst example for G_r within the class $J_{r,s}$.

Theorem 12: *For any ratio $r > 1$, $lim_{n\to\infty} R_{G_r}(J_{r,\sqrt{r}}) = 2\sqrt{r}$, so $lim_{n\to\infty} R_{G_r}(n) \geq 2\sqrt{r}$.*

Proof: We can break OPT's total completion time for $J_{r,\sqrt{r}}$ into three components: JE, the sum of the execution times of the jobs; PS, the pairwise delays of jobs in the same group; and, PD, the pairwise contributions of jobs in different groups. These terms, ignoring ϵ, can be expressed as follows.

$$
\begin{aligned}
JE &= \sum_{i=0}^{k-1} r^i \frac{m}{\sqrt{r^i}} = m \sum_{i=0}^{k-1} \sqrt{r^i} = m \frac{\sqrt{r^k}-1}{\sqrt{r}-1} \\
PS &= \sum_{i=0}^{k-1} \frac{r^i}{2} \left(\frac{m}{\sqrt{r^i}}\right)^2 = \frac{m^2}{2} \sum_{i=0}^{k-1} 1 = \frac{km^2}{2} \\
PD &= \sum_{i=0}^{k-1} r^i \frac{m}{\sqrt{r^i}} \left(\sum_{j=i+1}^{k-1} \frac{m}{\sqrt{r^j}} \right) \\
&= m \sum_{i=0}^{k-1} \sqrt{r^i} \left(\sum_{j=0}^{k-1} \frac{m}{\sqrt{r^j}} - \sum_{j=0}^{i} \frac{m}{\sqrt{r^j}} \right) \\
&= m \sum_{i=0}^{k-1} \sqrt{r^i} \frac{m}{\sqrt{r}-1} \left(\sqrt{r^{-i}} - \sqrt{r^{-k}} \right) \\
&= \frac{m^2}{\sqrt{r}-1} \sum_{i=0}^{k-1} 1 - \frac{m^2}{\sqrt{r}-1} \sum_{i=0}^{k-1} \sqrt{r^{\frac{i}{k-1}}} \\
&= k\frac{m^2}{\sqrt{r}-1} - \frac{m^2}{(\sqrt{r}-1)\sqrt{r^{k-1}}\frac{\sqrt{r^k}-1}{\sqrt{r}-1}} \\
&= \frac{km^2}{\sqrt{r}-1} - \frac{m^2\sqrt{r}}{r+1-2\sqrt{r}} + O(m^2).
\end{aligned}
$$

The dominant terms for large k, large m, and $m \gg r^k$ are the $\frac{km^2}{2}$ term of PS and the $\frac{km^2}{\sqrt{r}-1}$ term of PD. This means that the total completion time for $J_{r,\sqrt{r}}$ asymptotically is $km^2(\frac{1}{\sqrt{r}-1}+\frac{1}{2})$. Similarly, we can break down G_r's total completion time into the same three components where its JE component equals OPT's JE component, its PS component is twice the PS component of OPT, and its PD component is $(1+r)$ times the PD component of OPT. The same two terms of the PS and PD components will be dominant for G_r which means the total completion time for G_r will asymptotically be $km^2\left(\frac{r+1}{\sqrt{r}-1}+1\right)$. Thus the performance ratio of G_r on this instance is asymptotically $2\sqrt{r}$, so $\lim_{n\to\infty} R_{G_r}(n) \geq 2\sqrt{r}$. $\square$

The following theorem shows that the choice $s = \sqrt{r}$ is optimal in the above theorem.

Theorem 13: *$C_{G_r}(J_{r,s})/C_{OPT}(J_{r,s})$ is maximized by choosing $s = \sqrt{r}$.*

Proof: Break down the total completion time of OPT and G_r into the three components JE, PS, and PD. We first show that $C_{G_r}(J_{r,s})/C_{OPT}(J_{r,s})$ is maximized when $\frac{PD}{PS}$ is maximized. Then we will show that $\frac{PD}{PS}$ is maximized at $s = \sqrt{r}$ when $s \geq \sqrt{r}$. Finally, we will show that $\frac{PD}{PS}$ is maximized at $s = \sqrt{r}$ when $s \leq \sqrt{r}$.

When $s \neq \sqrt{r}$, JE is insignificant for both OPT and G_r if we choose m and k large enough. The dominant term of the PD component for OPT is

$$\frac{m^2}{s-1}\frac{\left(\frac{r}{s^2}\right)^k - 1}{\frac{r}{s^2}-1} - \frac{m^2}{s-1}s\frac{\left(\frac{r}{s^2}\right)^k}{\frac{r}{s}-1}$$

while the dominant term of the PS component for OPT is

$$\frac{m^2}{2}\frac{\left(\frac{r}{s^2}\right)^k - 1}{\frac{r}{s^2}-1}$$

Since the PD term for G_r equals the PD term for OPT multiplied by $1+r$ whereas the PS term for G_r equals the PS term for OPT multiplied by 2, $C_{G_r}(J_{r,s})/C_{OPT}(J_{r,s})$ is maximized by the choice of s which maximizes $\frac{PD}{PS}$.

If we ignore the negative half of the PD term, $\frac{PD}{PS} = \frac{2}{s-1}$ which is strictly decreasing for $s > 1$. Furthermore, for $s = \sqrt{r}$, $\frac{PD}{PS} = \frac{2}{\sqrt{r}-1} = \frac{2}{s-1}$. Therefore, $\frac{PD}{PS}$ is maximized by some $s \leq \sqrt{r}$.

For $s < \sqrt{r}$ and large k, simplify the PS and PD expressions by ignoring the -1 terms from the numerators of both expressions. $\frac{PD}{PS}$ now reduces to $\frac{2s}{r-s}$ which is strictly increasing between 1 and $\sqrt{r}$, and for $s = \sqrt{r}$, $\frac{PD}{PS} = \frac{2}{\sqrt{r}-1} = \frac{2\sqrt{r}}{r-\sqrt{r}} = \frac{2s}{r-s}$. Therefore, $\frac{PD}{PS}$ is maximized when $s = \sqrt{r}$ which means $\frac{C_{G_r}(J_{r,s})}{C_{OPT}(J_{r,s})}$ is maximized when $s = \sqrt{r}$. $\square$

4.4. Randomized Geometric Algorithms

We now describe a version of the geometric algorithm where the initial time slice is randomly chosen. This randomization avoids the worst-case behavior of G_r which is achieved by having the job's execution times synchronized with the time allocated to them during the various rounds. We feel the analysis of this algorithm provides real insight into why MLAF works well in practice as it accurately models real life situations where there is not an aggressive adversary choosing job lengths synchronized with the machine's quantum.

Definition 12: *For any $r > 1$, the randomized geometric algorithm RG_r works in rounds, giving each uncompleted job an equal time slice. The jobs are randomly permuted once, and this ordering is used in each round. The time slice of each round is determined as follows; choose a value t using some probability distribution on $[1, r]$.*

> **Round 0:** *Each job gets $\frac{t}{r-1}$ time units.*
>
> **Round i:** *Each job gets tr^{i-1} time units ($i \geq 1$).*

Round 0 exists only to simplify analysis by insuring that the sum of the lengths of rounds $0 \ldots i$ is $\frac{t}{r-1}r^i$ for each $i \geq 0$. We assume that each job has an execution time of at least $\frac{1}{r-1}$ time units (rescaling the minimum job size from 1 to $\frac{1}{r-1}$ for convenience).

We need to specify the distribution used to choose t. The following lemma shows that there is a unique choice of a best possible distribution.

Lemma 3: *For the algorithm RG_r, let $m(x,y)$ denote the supremum over job lengths x and y of $\frac{E[P_{x,y}]}{\min\{x,y\}}$. Then $m(x,y)$ is minimized by the choice of the probability distribution given by the density function $f(t) = \frac{1}{t \log r}$.*

Proof: Let the distribution on t be given by an arbitrary cumulative distribution function G on $[1,r]$. Assume that G is differentiable, with corresponding density function g. This is not a restrictive assumption since any probability distribution on an interval can be approximated arbitrarily well by a differentiable distribution function.

The maximum of $\frac{E[P_{x,y}]}{\min\{x,y\}}$ must clearly be attained for some pair $\{x,y\}$, the ratio of whose lengths is r. Without loss of generality, assume that the two jobs x and y have execution times $\frac{r\alpha}{r-1}$ and $\frac{r^2\alpha}{r-1}$ for some α satisfying $1 \le \alpha \le r$. Let X_g^α be the random variable whose value is the position in $[\alpha, r\alpha)$ of the end of a time slice. (Exactly one round (either round 1 or round 2) has a time slice that ends in this interval.) Because of the random ordering of the jobs, the delay (Definition 4) $D_{y,x}^{RG_r} = \frac{1}{2}(X_g^\alpha + \frac{1}{r}X_g^\alpha)$. Therefore

$$E[P_{x,y}] = \frac{r\alpha}{r-1} + \frac{1}{2}\left(1 + \frac{1}{r}\right) E[X_g^\alpha]$$

so we have

$$\frac{E[P_{x,y}]}{\min\{x,y\}} = 1 + \left(\frac{1}{2} + \frac{1}{2r}\right)\frac{E[X_g^\alpha]}{\alpha}\frac{r-1}{r} \tag{10}$$

To find the supremum of this quantity we need to find the supremum of $E[X_g^\alpha]/\alpha$. In fact, it can be easily verified that for the function f defined in the statement of the lemma, $E[X_f^\alpha]/\alpha$ is independent of α. Hence to show that

$$\sup\ E[X_g^\alpha]/\alpha \ge \sup\ E[X_f^\alpha]/\alpha,$$

we only need to show that for some α, $E[X_g^\alpha] \ge E[X_f^\alpha]$. The rest of the proof is devoted to this aim.

Note that

$$\begin{aligned}
\frac{r-1}{r}E[X_g^\alpha] &= \int_\alpha^r t g(t)\mathrm{d}t + \int_r^{r\alpha} t\frac{g(t/r)}{r}\mathrm{d}t \\
&= \int_\alpha^r t g(t)\mathrm{d}t + \int_1^{\alpha} r\,u\,g(u)du \quad \text{(putting } u = t/r\text{)} \\
&= \int_1^r t g(t)\mathrm{d}t + (r-1)\int_1^{\alpha} t g(t)\mathrm{d}t
\end{aligned}$$

Consider the function $h(t) = g(t) - f(t)$. Define $q(\alpha) = \frac{r-1}{r}(E[X_g^\alpha] - E[X_f^\alpha])$, and define $C(\beta) = \int_1^\beta t\,h(t)\mathrm{d}t$. We have

$$q(\alpha) = C(r) + (r-1)C(\alpha)$$

and we need to show that for some $\alpha \in [1,r]$, $q(\alpha) \ge 0$.

Assume that $C(r) < 0$, otherwise we are done. We will further assume that there are a finite number of regions $(l_1, r_1), \ldots, (l_k, r_k)$, with the l_i strictly increasing, inside which h is non-zero. By uniform continuity we can suitably approximate h by such a function.

Suppose that h is negative inside some (l_i, r_i) and positive inside (l_{i+1}, r_{i+1}). We can change g by moving probability from the interval (l_{i+1}, r_{i+1}) to the interval (l_i, r_i) until h is zero in one of the two intervals. Clearly $C_h(r)$ has decreased. Also, because $C_h(\alpha)$ must be maximized at r_j for some interval (l_j, r_j) on which h is positive, and we did not move probability across any such points, it is also true that $\max_\alpha C(\alpha)$ has also decreased. Hence $\max_\alpha q(\alpha)$ has decreased.

We can continue this process so long as there are a pair of consecutive intervals such that h is negative in the leftmost of the two and positive in the right interval. Thus, we will end up with a situation in which there exists a point α such that h is non-negative to the left of α, and non-positive to the right. Because $\int_1^r h(t)\mathrm{d}t = 0$, it is easy to see that $C(\alpha) \geq \frac{-1}{r-1}C(r)$ as required. $\square$

The next theorem determines the idle-performance ratio of this "best possible" randomized geometric algorithm.

Theorem 14: *For any $r > 1$, the randomized algorithm RG_r using the density function $f(t) = \frac{1}{t \log r}$ has performance ratio $c(r) = 1 + \frac{(r+1)(r-1)}{2r \log r}$, and for no other distribution does RG_r have a lower idle-performance ratio.*

Proof: The value of $c(r)$ is obtained by calculating $E[X_f^\alpha]$ in (10). It follows from the definition of $P_{x,y}$ that using the density function f, RG_r is $c(r)$-idle-competitive. For any other distribution, fix a pair (x, y) of job lengths that maximizes $\frac{E[P_{x,y}]}{\min\{x,y\}}$. Then by the above lemma, RG_r is at least $c(r)$-idle-competitive when given the jobs x, y. $\square$

Notice that there is a tradeoff between performance ratio and the number of preemptions. As r increases, the performance ratio increases but the number of preemptions decreases. Notice also that for small values of r, $c(r)$ is close to the optimal value of 2 (see Theorem 4, Section 2.3). For instance, when $r = 2$, $c(r) = 2.08$.

5. Dynamic Scheduling

We now turn our attention to the problem of *dynamic* non-clairvoyant scheduling. The n jobs can be released at arbitrary times, and the non-clairvoyant algorithm is not aware of the existence of a job until it is released. The measure of performance for the dynamic situation is the average waiting time of the jobs (note that this is no longer the same as the average completion time). The dynamic scheduling problem can model the practical situation where jobs can be interrupted for I/O. In our context, I/O interrupts merely divide a job into a set of smaller jobs with varying release times.

Clearly, the static model we explored in the previous sections is a special case of the dynamic model. It is reasonable to hope that the algorithms which work well in the static model will carry over with little or no loss of competitiveness. Unfortunately, any deterministic non-clairvoyant algorithm must perform very poorly when compared to the optimal clairvoyant algorithm. The situation may be better for randomized non-clairvoyant algorithms, but it is still not possible to be competitive. These facts (and their proof) show that clairvoyance can make a very large difference for a scheduler trying to keep a system responsive under heavy loads.

The clairvoyant algorithm OPT obeys the following generalization of Smith's rule [14]: *the job being executed at any time is the one which requires the smallest amount of processing to complete.* Note that OPT requires no knowledge about future jobs.

We now show that the non-clairvoyant algorithms of the previous sections are no longer competitive in this more general setting. We start with RR; in the dynamic setting, RR ensures that during any time interval where there are no completions or releases, each active job receives an equal share of the processing time.

Theorem 15: *For the dynamic scheduling problem of size n, RR has performance ratio $\Omega(\frac{n}{\log n})$.*

Proof: Let H_k denote the k'th harmonic number. Consider the set of n jobs with the following release and execution times.

- At time 0, two jobs of length 1 are released.
- For $1 \leq k \leq n-2$, at time H_k, a job of length $\frac{1}{k+1}$ is released.

First consider the performance of OPT on this instance. OPT sets aside one of the first two jobs and schedules the remaining jobs in a non-preemptive manner. In other words, each job (except the one set aside) is scheduled as soon as it is released and finishes just as the next job is released. Finally, the initially set aside job is scheduled. This one job has waiting time $1 + H_{n-1}$ while every other job has waiting time equal to its length. Therefore the total waiting time for the n jobs is $2H_{n-1} + 1$.

RR, on the other hand, does not complete any job until time $H_{n-1} + 1$. To see this, observe that when the job of size $\frac{1}{k}$ enters the system, the k jobs that have already been released require exactly the same amount of processing time, i.e. $\frac{1}{k}$, to complete. A straight-forward calculation shows that RR has a total waiting time of $2n + H_{n-1} - 1$. Thus the performance ratio of RR on this instance is $\Omega(\frac{n}{\log n})$. $\square$

While Round Robin has a very large performance ratio (no algorithm which does not idle the processor can have a performance ratio worse than n), the following theorem shows that every deterministic, non-clairvoyant algorithm has a large performance ratio.

Theorem 16: *For the dynamic scheduling problem of size n, any deterministic non-clairvoyant algorithm has performance ratio $\Omega(n^{\frac{1}{3}})$.*

Proof: We present an adversary strategy which works in two stages. At time 0, the adversary releases k jobs and lets the algorithm A schedule them for k time units. The adversary ensures that the length of each job is large enough so that no job is completed during that period. Let e_i be the amount of processing time allocated to job i during the first k time units. Without loss of generality, assume that the jobs are ordered so that $e_i \geq e_j$ if $i < j$; clearly, $e_1 \geq 1$. The adversary now chooses the execution times of the k jobs such that $x_i = e_i + \frac{1}{k-1}$. Observe that by this time OPT would have completed all these jobs except the largest one, viz. job 1 while the algorithm A has completed no jobs, and each job requires $\frac{1}{k-1}$ time units of processing to complete.

After the first k time units, the adversary releases a job of length $\frac{1}{k-1}$ every $\frac{1}{k-1}$ time units apart for a total of k^2 time units. A total of $k^3 - k^2$ jobs are released in this phase. It is clear that for both algorithms, OPT and A, the best possible strategy is to immediately run to completion each job that is released during this second phase. Finally, both algorithms will finish off all the remaining jobs from the first phase.

During the first phase, the waiting time for both algorithms is $O(k^2)$ because there are only k jobs around for k time. For the algorithm A, $k+1$ jobs are present in the system at any time during the second phase while OPT only has two unfinished jobs at any time during the second phase. Thus OPT has total waiting time $O(k^2)$, while the algorithm A has total waiting time $\Omega(k^3)$. This implies a lower bound of $\Omega(k)$ on the performance ratio of A. Since $n = k^3 - k^2 + k < k^3$, $k > n^{\frac{1}{3}}$ completing the proof. □

The assumption that A is deterministic is crucial to the above proof. The adversary must decide upon the lengths of the jobs after observing the amount of processing received by each job. While we hold out hope for defeating such an adversary and obtaining a reasonably small performance ratio by the use of randomness, the following theorem shows that even randomized algorithms cannot be competitive.

Theorem 17: *For the dynamic scheduling problem of size n, any (randomized or deterministic) non-clairvoyant algorithm has performance ratio $\Omega(\log n)$.*

Proof: We use Yao's technique (see Theorem 4) and prove a lower bound on the performance ratio of deterministic algorithms on an dynamic scheduling instance drawn from a probability distribution.

Fix a non-idling non-clairvoyant scheduling strategy. The jobs are released in two phases. In the first phase, at time 0, k jobs are released whose sizes are independently drawn from the exponential distribution with mean 1 (if X is a random variable denoting the size of a job, then for $c \geq 0$, $\Pr[X > c] = e^{-c}$). The scheduling algorithm is then allowed to run for time $k - 2k^{3/4}$.

Let N_t be the expected number of jobs that are completed in t time units during the first phase. If the initial pool of jobs is infinite, then regardless of the scheduling strategy, N_t is Poisson with mean t. Therefore the variance $V(N_t) = t$, so with probability $1 - O(k^{-1/2})$, $N_t \leq t + t^{3/4}$. Hence with a finite initial pool of k jobs, if $t = k - 2k^{3/4}$ then with probability $1 - O(k^{-1/2})$, $k - N_t \geq k^{3/4}$. With probability $1 - O(k^{-1})$, at least a $\frac{1}{2e}$ fraction of the remaining jobs at time $t = k - 2k^{3/4}$ have remaining size at least 1 (since each remaining job has probability $1/e$ of having remaining size at least 1). Hence with high probability, at the end of the first stage, the non-clairvoyant algorithm has remaining $\Omega(k^{3/4})$ jobs of size at least 1.

The second phase consists of releasing a job of size 1 at each time unit, for a total of k^2 time units. Clearly the non-clairvoyant algorithm should execute these jobs before the large jobs it has remaining from the first phase, for a expected total waiting time of $\Omega(k^{2.75})$.

Consider now the clairvoyant algorithm: during the first phase it sets aside any job whose execution time is greater than b. It chooses b to be maximal such that all shorter jobs are finished within the first phase. We claim that with high probability, $b \geq \frac{\log k}{4}$. We make use of the Chernoff bound [4] on tails of the binomial distribution, i.e. for $0 \leq \epsilon \leq 1$, if $S_{k,p}$ is number of successes in k independent Bernoulli trials each having success probability p,

$$\Pr[|S_{k,p} - kp| \geq \epsilon kp] \leq 2e^{-\epsilon^2 kp/3}$$

Substituting $p = e^{-b}, b = \frac{\log k}{4}$, and $\epsilon = 1/2$ in this bound gives us that with probability $1 - O(1/k)$, there are at least $k^{3/4}/2$ jobs of size at least b released in the first phase. A variance argument shows that with probability $1 - O(k^{-1/2})$, the total size of the set of all k jobs is at most $k + k^{3/4}$. Hence the clairvoyant scheduler can set aside $k^{3/4}/b$ jobs of size at least b, and finish all other jobs in the first phase.

Thus after the first phase, with high probability the clairvoyant algorithm has only $O(k^{3/4}/\log k)$ jobs remaining. It executes the second phase jobs as they arrive, and lastly schedules the set-aside jobs. The expected total waiting time of the clairvoyant algorithm is $O(k^{2.75}/\log k)$. Since $n = k^2 + k$, the performance ratio of the non-clairvoyant algorithm is $\Omega(\log n)$. $\Box$

The above lower bounds use collections of jobs where the ratio between the shortest and longest job is unbounded. The following theorems give tight bounds when the ratio between the smallest and largest job is bounded by k, for some constant k. These theorems should be contrasted with Theorem 5 where it is shown that knowledge of a small bound on the job sizes does not improve the competitive ratio of deterministic, non-clairvoyant algorithms for static scheduling.

Theorem 18: *No deterministic, non-clairvoyant algorithm has performance ratio bounded away from k when the maximum ratio is k.*

Proof: We present a two-stage adversary strategy. At time 0, the adversary releases $m(k+1)$ jobs and lets the algorithm A schedule these jobs until m jobs have received at least $k-1$ units of processing time. Call this time s. If at any time before s one of these m jobs is processed for k time units, it is finished. Let e_i be the amount of processing time job i has received before time s. Define $x_i = \max\{k, e_i + 1\}$ The shortest jobs left at time s for A are the m jobs that have received at least $k-1$ units of processing time, so assume that A does the optimal thing and immediately finishes these m jobs, say by time t. A still has mk jobs left of size 1. OPT, on the other hand, by not giving any of these m longest jobs any processing time before time t would have finished all the shorter jobs leaving only these m jobs of length k at time t.

After time t, the adversary releases a job of length 1 every time unit for a time period of mk^4 time units resulting in the release of mk^4 further jobs. It is clear that the best possible strategy for both algorithms is to run to completion each job that is released during this second phase. Finally, both algorithms will finish off all the remaining jobs from the first phase.

To calculate the total waiting time of each algorithm, notice that for both algorithms, the waiting time during the second phase dominates the waiting time of the other phases. For the algorithm A, $mk+1$ jobs are present in the system during the second phase while for OPT, only $m+1$ jobs are present in the system. This leads to a waiting time of $m^2k^5(1+o(1))$ for A while OPT has a waiting time of $m^2k^4(1+o(1))$ which gives us the desired ratio of k. $\Box$

Theorem 19: *Run-to-completion is k-competitive if the largest job is at most k times as large as the smallest job.*

Proof: At any time, OPT and RTC have spent the same total time processing the set of jobs. In addition, RTC never sets aside a job that has received some processing time. Hence at any time, RTC has no more than k times as many jobs as OPT that are not currently being processed. Note that the total waiting time is the sum of the job execution times plus the integral over time of the number of jobs not currently being processed. Therefore RTC is k-competitive. □

6. Scheduling Dependent Jobs

We now consider scheduling jobs on multiple machines when there is some dependence between the jobs. Any multi-processor system will have certain resources which can only be made available to one job at a time. Two jobs which demand such a resource are constrained to have non-overlapping executions. Any scheduler for such a system must guarantee two properties: 1) *safety*: no two conflicting jobs may be executed simultaneously; and, 2) *liveness*: there is progress being made on completing the jobs. This problem has been abstracted into the well-known *dining philosophers* and *drinking philosophers* problems [9, 22, 3, 17]. Several interesting algorithms have been devised [18, 26, 3, 17, 1] to solve such problems in an "online" setting. Typically, the goal is to minimize the response time for a job in terms of the parameters of the job set.

We have obtained some preliminary results in the following model. An instance of the *distributed scheduling problem* is a graph where the vertices represent jobs, and two vertices are adjacent when the corresponding jobs are in conflict. The processing environment consists of an infinite number of processors, in which each job has an execution time and a release time, and a centralized scheduler assigns jobs to processors. A clairvoyant scheduler has complete knowledge of the jobs which will appear in the future including their conflict relations, execution times and release times. A non-clairvoyant scheduler has the complete information only about the jobs which have already been released, in particular it knows the conflict relations between the currently available jobs.

To focus on the issue of clairvoyance, we assume that the number of processors is infinite. Having a bounded number of processors can only make the non-clairvoyant algorithms more competitive. Unfortunately, our results apply only to the makespan measure of performance, in which the goal of a scheduler is to minimize the time until all the jobs have been completed.

In the *uniform distributed scheduling* problem each job has unit execution time, and jobs are released at integral times. We start by considering this special case which is closely related to the problem of online coloring of graphs [16, 29]. For this problem, the optimal non-clairvoyant algorithm, even with randomization, has a performance ratio asymptotic to 2. An algorithm that achieves this bound is the *CHROMATIC* algorithm.

Theorem 20: *For the uniform distributed scheduling problem, the performance ratio of any deterministic non-clairvoyant algorithm is at least* $2 - \frac{2}{n+1}$.

Proof: Let A be any non-clairvoyant algorithm, and consider the following adversary strategy. At time 0, the adversary releases m jobs $J_1, \ldots, J_m$ such that their conflict graph forms a clique. Then, for $1 \leq t \leq m-1$, exactly one job J_t^* is released at time t. The adversary arranges the conflict relations of these new jobs so as to guarantee the following invariant. The *clique invariant* asserts that at all times $0 \leq t \leq m-1$, the collection of jobs available to algorithm A have a conflict graph which is a clique. Given this invariant, the algorithm can execute exactly one job during each time interval $[t, t+1]$. We assume that the algorithm always executes at least one job at each time instant, otherwise the adversary argument can be easily adapted to make A's performance even worse.

We now show how the clique invariant can be maintained. Consider any optimal vertex coloring of the entire task graph. The adversary assigns conflict edges to J_t^* in such a way that it is adjacent to all the vertices except those in the color class of the job executed by A during the time interval $[t-1, t]$. Thus, the overall graph has chromatic number m. A simple induction argument establishes that at all times, the algorithm A has exactly one job from each of the m color classes, and these jobs form a clique.

We conclude that the total time required by the algorithm A is exactly $2m-1$, since a clique must be scheduled starting at time $m-1$. On the other hand, a clairvoyant algorithm can do much better. Let us call a color class as a level t color class if the largest release time of a job in that class is t. It is easy to verify that for all $t \leq m-1$, there are at least $t+1$ color classes of level at most t. The clairvoyant algorithm orders the color classes by increasing levels (breaking ties arbitrarily), and executes an entire color class at a time in that order. Clearly, it always executes an entire color class at a time, without ever being idle. Thus, it has a makespan of m. Since $n = 2m-1$, we have the desired result. □

We now show how this argument can be extended to allow for randomized schedulers.

Theorem 21: *For uniform distributed scheduling, no randomized non-clairvoyant algorithm is c-competitive for any constant $c < 2$.*

Proof: Now the adversary is not allowed to use the random choices made by the randomized algorithm A to determine the input instance. This precludes the trick of ensuring that the jobs J_t^* get the same color as the job executed in the time interval $[t-1, t]$. Instead, the adversary "guesses" the color of the job executed in the previous time interval by randomly choosing one of the m colors with the uniform distribution. Clearly, if it makes the right choice at each of the $m-1$ release times, we obtain the desired lower bound. The probability that it always makes the right choice is exactly $1/m^{m-1}$. We can boost this probability by using $O(m^m)$ independent task graphs which are provided concurrently. With high probability, on at least one of these task graphs the algorithm A will attain the competitive ratio of $2 - 2/m$. Since $n = O(m^m)$, we have the desired lower bound. □

We now show that there exists a clairvoyant algorithm which achieves the competitive ratio of 2. We refer to this algorithm as the *CHROMATIC* algorithm – at each time step it optimally colors the vertices of the residual task graph and schedules any one color class in parallel. Note that this algorithm is not computationally efficient, since coloring is NP-hard.

Theorem 22: *The $CHROMATIC$ algorithm is 2-competitive for the uniform distributed scheduling problem.*

Proof: Suppose that the last job to be released is released at time T. Let the chromatic number of the task graph be χ. Then, regardless of the jobs scheduled before time T, the residual graph at time T has chromatic number at most χ. It follows that the makespan of the $CHROMATIC$ algorithm is at most $T+\chi$. On the other hand, the optimal clairvoyant algorithm cannot finish all the jobs before time $T+1$. Moreover, the chromatic number of the entire task graph is a lower bound on the number of time units needed by OPT. Thus, we have that

$$C_{CHROMATIC} = T + \chi \leq (C_{OPT} - 1) + C_{OPT} \leq \left(2 - \frac{1}{C_{OPT}}\right) C_{OPT}$$

□

Consider now the general distributed scheduling problem where the jobs are allowed to have arbitrary execution and release times. The algorithm we analyze is a generalization of the $CHROMATIC$ algorithm and is referred to as the $GREEDY$ algorithm. At any time when no job is currently being executed, the $GREEDY$ algorithm considers the residual job graph and initiates the execution of those jobs which would have been executed first by an optimal algorithm.

Theorem 23: *The $GREEDY$ algorithm is 3-competitive.*

Proof: Let T be the release time of the last job to be released. Further, let E be the execution time of the job with the largest execution time. Suppose that the optimal clairvoyant algorithm has a makespan of M for the given task graph. Then we claim that the $GREEDY$ algorithm has a makespan of $T+E+M$. This is because the jobs being executed at time T will complete by time $T+E$, after which the residual task graph will be optimally scheduled by $GREEDY$. Moreover, the makespan of the optimal clairvoyant algorithm is at least as large as the larger of T and E. This gives an upper bound of 3 on the competitive ratio of $GREEDY$. □

The question of determining the optimal ratio for the general problem remains open.

7. Open Problems

We list here the most important unresolved problems in non-clairvoyant scheduling.

Dynamic scheduling upper bounds. Can one find an algorithm for dynamic scheduling with a $o(n)$ performance ratio? In particular, it would be desirable to find a randomized algorithm with a performance ratio close to the $O(\log n)$ lower bound. This would prove that randomization provides significant help for dynamic scheduling, in contrast to the case of static scheduling.

Dynamic scheduling model. In practice there is an important distinction between I/O bound and computationally intensive processes. Our model of dynamic scheduling could be extended by formalizing this distinction.

Multi-processor scheduling. The power of randomization in multi-processor scheduling has not been determined. Furthermore, it would be interesting to expand the model to include communication costs between processors.

Minimizing preemptions. It would be interesting to find the exact performance ratio of geometric algorithms.

Dependent jobs. The model should be extended to measure the average waiting time of jobs, not just the makespan of the schedule. The question of determining the optimal ratio for general distributed scheduling problem also remains open.

Acknowledgements

This work originated in discussions the first author had with Howard Karloff and Seffi Naor. We would like to thank them for their contributions. The second author is grateful for conversations with Anna Karlin. We would also like to thank Barun Chandra and Sundar Vishwanathan for sharing their observations with us. Finally, we are grateful to Amos Fiat for his help and encouragement.

References

[1] B. Awerbuch and M. Saks, A dining philosophers algorithm with polynomial response time, in: *Proc. of 30th ACM Symp. on Foundations of Computer Science* (1990) 65–74.

[2] Y. Bartal, A. Fiat, H. Karloff and R. Vohra, New Algorithms for an Ancient Scheduling Problem, in: *Proc. of 24th ACM Symp. on Theory of Computing* (1992) 51–58.

[3] K. Chandy and J. Misra, The drinking philosophers problem, *ACM TOPLAS* **6** (1984) 632–646.

[4] H. Chernoff, A Measure of Asymptotic Efficiency for Tests Based on the Sum of Observations, *Annals of Mathematical Statistics* **23** (1952) 493–509.

[5] E.G. Coffman, Jr., and P.J. Denning, *Operating Systems Theory* (Prentice-Hall, Englewood Cliffs, 1973).

[6] R.W. Conway, W.L. Maxwell and L.W. Miller, *Theory of Scheduling* (Addison-Wesley, Reading, 1967).

[7] H.M. Deitel, *An Introduction to Operating Systems* (Addison-Wesley, Reading, 1990).

[8] H.M. Deitel and M.S. Kogan, *The Design of OS/2* (Addison-Wesley, Reading, 1992).

[9] E.W. Dijkstra, Hierarchical ordering of sequential processes, *Acta Informatica* (1971) 115–138.

[10] A. Feldmann, J. Sgall and S-H. Teng, Dynamic Scheduling on Parallel Machines, in: *Proc. of 32nd IEEE Symp. on Foundations of Computer Science* (1991) 111–120.

[11] R.E. Goldberg and L.J. Kenah, *VAX/VMS Internals and Data Structures* (Digital Press, Bedford, 1991)

[12] R.L. Graham, Bounds on multiprocessing anomalies, *SIAM Journal of Applied Mathematics* **17** (1969) 263–269.

[13] L. Kleinrock, *Queuing Systems, Vol. I (Theory) and Vol. II (Computer Applications)* (John Wiley & Sons, New York, 1976).

[14] E.L. Lawler, J.K. Lenstra, A.H.G. Rinnooy Kan and D.B. Shmoys, Sequencing and Scheduling: Algorithms and Complexity, in: *Handbooks in Operations Research and Management Science, Vol. 4: Logistics of Production and Inventory* (1990).

[15] E.D. Lazowska, J. Zahorjan, G.C. Graham and K.C. Sevcik, *Quantitative System Performance* (Prentice-Hall, Englewood Cliffs, 1984).

[16] L. Lovasz, M. Saks and W.T. Trotter, An online graph coloring algorithm with sublinear performance ratio, *Discrete Mathematics* **75** (1989) 319–325.

[17] J. Lundelius and N. Lynch, Synthesis of efficient drinking philosophers algorithms, Unpublished manuscript, 1988.

[18] N. Lynch, Upper bounds for static resource allocation in a distributed system, *Journal of Computer and System Sciences* **23** (1981) 254–278.

[19] R. McNaughton, Scheduling with deadlines and loss functions, *Management Science* **6** (1959) 1–12.

[20] R. Motwani, S. Phillips and E. Torng, Non-Clairvoyant Scheduling, in: *Proc. of Fourth ACM-SIAM Symp. on Discrete Algorithms* (1993) 422–431.

[21] J.L. Peterson and A. Silberschatz, *Operating Systems Concepts* (Addison-Wesley, 1985).

[22] M. Rabin and D. Lehmann, On the advantages of free choice: a symmetric and fully distributed solution to the dining philosophers problem, in: *Proc. of 8th ACM Symp. POPL* (1981) 133–138.

[23] D.B. Shmoys, J. Wein and D.P. Williamson, Scheduling Parallel Machines On-line, in: *Proc. of 32nd IEEE Symp. on Foundations of Computer Science* (1991) 131–140.

[24] D.D. Sleator and R.E. Tarjan, Amortized efficiency of list update and paging rules, *CACM* **28** (1985) 202–208.

[25] W.E. Smith, Various optimizers for single-stage production, *Naval Res. Logist. Quart.*, **3** (1956) 59–66.

[26] E. Styer and G. Peterson, Improved algorithms for distributed resource allocation, in: *Proc. of 7th ACM Symp. on Principles of Distributed Computing* (1988) 105–116.

[27] A.S. Tanenbaum, *Modern Operating Systems* (Prentice-Hall, Englewood Cliffs, 1992).

[28] A.S. Tanenbaum and R. van Renesse, Distributed Operating Systems, *ACM Computing Surveys* **17** (1985) 419–470.

[29] S. Vishwanathan, Randomized online graph coloring, in: *Proc. of 31st IEEE Symp. on Foundations of Computer Science* (1990) 464–469.

[30] A.C.-C. Yao, Probabilistic computations: Towards a unified measure of complexity, in: *Proc. of 18th IEEE Symp. on Foundations of Computer Science* (1977) 222–227.