Scheduling Accesses to Shared Memory by a Bus Arbiter: the $k-$client Problem Revisited

Eli Almog

Scheduling Accesses to Shared Memory by a Bus Arbiter: the $k-$client Problem Revisited

Research thesis
submitted in partial fulfillment of
the requirements for the degree of
Master of Science in Computer Science

Eli Almog

Submitted to the Senate of the Technion—Israel Institute of Technology
Tevet 5761 Haifa January 2001

The research thesis was done under the supervision of
Doctor Hadas Shachnai in the Department of Computer Science.

I wish to express my deepest gratitude and appreciation
to Dr. Hadas Shachnai
for her excellent guidance which made this work possible.

The generous financial support of
the Technion—Israel Institute of Technology
is greatly acknowledged.

Contents

List of Figures

Abstract

Multiprocessors are becoming a common type of modern computers. We focus on the *bus* architecture that is used, e.g., in multiprocessor PCs and in commercial systems. In bus architecture a *shared* bus is used to connect processors and memories. All communication, and in particular memory accesses, is done through the bus: an arbiter grants control on the bus to at most one processor, at any time. In this work we consider a *dynamic* bus architecture, in which the arbiter can connect the processors through the bus to *several* banks of memory.

Our optimization problem can be stated as follows. Assume that k processors access m memory banks through a single bus. In order to fulfill an access request for bank i, the bus has to be connected to that bank; the bus can be connected at any time to at most one bank of memory. The cost of servicing a request to access bank i, $1 \leq i \leq m$, is λ, if the bus is connected to bank i, and μ, if the bus is connected to some bank $j \neq i$; in this case $\mu - \lambda > 0$ is a *setup* cost, or the cost of switching the bus from bank j to bank i. Given a sequence of memory access requests, we need to determine the order by which the requests will be serviced, so as to minimize one of the following objective functions:

- *makespan* – the overall time until all the requests in the sequence have been completed
- *total completion time (TCT)* – sum of the completion times of all the processors
- *maximum response time* – the maximal time that elapses between services of consecutive requests of processor i, $1 \leq i \leq k$.

The problem of minimizing the makespan is known to be NP-complete. In this work we give a proof of hardness also for the TCT problem. Using the $k-$client model introduced by Alborzi et al. (1997), we study algorithms that schedule accesses to the memory banks in an offline and online fashion. In the offline case we show, that using dynamic programming our problem can be solved optimally for a large set of instances, for each of the above objective functions. For general inputs we give approximation algorithms, for the makespan and the TCT measures.

In the online case, we show a lower bound of $\Omega(\mu/\lambda)$ on the competitive ratio of any deterministic algorithm, for each of the above measures. We also show that the bound is tight for all measures, by describing algorithms whose competitive ratios are $O(\mu/\lambda)$.

Finally, we examine the performance of the algorithms studied in this work in real system environment, via simulation experiments.

List of Symbols

TCT	Total Completion Time
MRT	Maximum Response Time
FCFS	First Come First Served
SCS	Shortest Common Supersequence
DP	Dynamic Programming
LTSP	Loading Time Scheduling Problem
RR	Round-Robin algorithm
GRR	Greedy Round-Robin algorithm
GERR	Greedy Enhanced Round-Robin algorithm
SEQ	Sequential algorithm
GSEQ	Greedy Sequential algorithm
GCSEQ	Greedy Cautious Sequential algorithm
SCF	Shortest Client First algorithm
SPTF	Shortest Processing Time First algorithm
Gr	Greedy algorithm
k	number of clients
M	number of memory banks
n_i	number of requests in the chain of client i
λ	the processing time (service time without setup)
μ	service time with setup
$r_i(j)$	memory bank accessed by client i in its j-th request
$t(r)$	the point in time in which the service of request r is completed
$t_s(r)$	the time that it takes to the arbiter to service a request r
c_i	completion time of client i
$\hat{c}_i$	the minimal time required for servicing all the requests of client i, when this client is alone in the system
x_i	the time that it takes to service the requests of client i
$D^{A}_{i,j}$	the time allocated by algorithm $\mathcal{A}$ to client i while the service of client j has not been completed

$P_{i,j}^{A}$ .... the amount by which clients i and j delay the completion times of each other

$\mathcal{A}_{ms}(I)$ .... the makespan obtained by an algorithm $\mathcal{A}$

$OPT_{ms}(I)$ .... the makespan obtained by an optimal algorithm, OPT

$\mathcal{A}_{tct}(I)$ .... the total completion time obtained by an algorithm $\mathcal{A}$

$OPT_{tct}(I)$ .... the total completion time obtained by an optimal algorithm, OPT

$\mathcal{A}_{mrt}(I)$ .... the maximum response time obtained by an algorithm $\mathcal{A}$

$OPT_{mrt}(I)$ ...... the maximum response time obtained by an optimal algorithm, OPT

Chapter 1

Introduction

1.1 Background - Accessing Shared Memory in Bus Oriented Architectures

The decreasing costs of standard processing units have led to the development of many parallel applications in the past two decades. Moreover, computer manufacturers have realized, that efficient connection of several standard processors on the same machine can achieve a cost/performance ratio that is better than this ratio for a machine having a single high-speed processing unit: such fast components are often more expensive, and less reliable. These two factors, among others, have pushed forward multiprocessors, to become a common type of modern computers.

The architecture in which the processors are connected, can significantly affect the performance and scalability of a multiprocessor. In this work we focus on the *bus-oriented* (for short, *bus*) architecture, that is common in multiprocessor PCs, and in commercial systems [2, 14]. In a bus architecture, all the components communicate through a single data *bus*. The bus is used for communication between processors – and between processors and a shared memory. The centralized *arbiter* is a hardware device which acts as administrator of the bus: the arbiter receives requests to transmit data over the bus; it grants control on the bus to at most one processor, at any time. Generally, the bus is a bottleneck in such architecture. The nature of the bus overloading problem is described in [12].

1.2 Problem Statement

In this work we examine a model of bus architecture, in which the arbiter can connect *dynamically* to a shared-memory, which consists of several *banks*, through a single bus. In addition, we examine various scheduling schemes for handling memory access requests. Traditional single bus arbiters commonly use a *First*

Come First Served (FCFS) schedule.

Our scheduling problem can be stated as follows. Assume that k processors access m memory banks through a single bus; the bus can be connected to any memory bank, and in order to fulfill an access request for bank i, a bus has to be connected to that bank; the bus can be connected at any time to at most one bank of memory. Given a sequence of memory access requests, the arbiter has to decide which of the pending requests to service next. The cost of servicing a request r to bank i, $1 \leq i \leq m$, is λ, if the bus is connected to this bank, and μ otherwise; in this case $\mu - \lambda > 0$ is a *setup* cost, or the cost of switching a bus from another bank to bank i.

Generally, there are two approaches to writing data to the memory: *synchronous* (blocking) and *asynchronous* (non-blocking). In the synchronous approach, whenever a processor sends a write request to the arbiter, it has to wait until the data is written to the memory; in the asynchronous approach, as soon as the request is sent to the arbiter, the processor can continue its execution.

In this work we adopt the first approach. We study algorithms that schedule memory access requests in an *offline* and *online* fashion, that is, upon arrival of a request r, the arbiter does not know the arrival times and the addresses (given as memory banks) of future requests. However, the scheduler knows that the processing time for any request is λ, and that setup time is $\mu - \lambda$[1]. In such a setting, performance bounds for any algorithm will be derived by using *competitive analysis* [4]: we will measure the performance of online algorithms against the optimum offline algorithm, that knows all the sequence of requests in advance.

The following optimization goals will be used for evaluating the performance of algorithms for a given sequence of requests.

- *Makespan* – the overall time until all the requests in the sequence have been completed.
- *Total Completion Time* – sum of the completion times of all the processors.
- *Maximum Response Time* - the maximal response time of any request in the sequence.

1.3 The $k-$Client Problem

The $k-$*client* problem introduced by [1] will be used in our study of the scheduling problem, under the synchronous approach. In the basic $k-$client problem, there is one server , k clients, and a metric space (e.g. a plane or a line). Each client generates a sequence of requests in the metric space, and a request is serviced the moment the server (which moves at a constant speed) reaches the location

[1]In terms of online scheduling we call such a scheduler *clairvoyant*[13].

of that request. In the basic model it is assumed, that the processing time of any request is zero; thus, the service time of request i is determined by the time required for moving the server from its previous location to the location of request i. A variation of the basic problem in which the requests have non-zero processing times[2] was also considered in [1]. We use the following assumptions for the $k-$client model in the present work:

- The same non-zero processing time for all requests.
- The requests are located on a clique.
- The movement of the server from one vertex to another is called *setup time*. Because of the clique topology, the setup times are the same for all requests.

1.4 Related Work

Alborzi et al. studied in [1] online algorithms for the $k-$client problem. For the case where all processing times are equal to zero (i.e., the cost of any schedule is determined by the number of setups), the paper gives a lower bound of $\frac{\lg k}{2}+1$ on the competitive ratio of deterministic online algorithms, for both the makespan and the total completion time measures; algorithms whose competitive ratio is $(2k-1)$ are presented for both measures. These lower and upper bounds hold for a general topology. When $k=2$ the lower bounds improve to $\frac{25}{9}$ and 3 for the makespan and the TCT measure, respectively. A lower bound of $1+\Theta(\lg\lg k)$ was derived for any randomized online algorithm, already for the case of a clique topology, i.e., a clique of (k+1) vertices (denoted by K_{k+1}).

For the MRT measure the paper [1] refers to the case, where the topology is a $line(\triangle)$, i.e., a discrete line segment of length $\triangle$, in which the distance between neighboring points is one. For this topology the paper gives a lower bound of $\Omega(\sqrt[3]{\triangle})$ on the competitive ratio of any deterministic online algorithm; the paper presents algorithm whose competitive ratio is $\Theta(\triangle)$.

The paper [1] considers also the $k-$client problem with non-zero processing times. Assuming that requests may have *different* processing times and the scheduler is *non-clairvoyant*, i.e., the processing time of a given request is known only when its processing is completed, the paper shows that any online deterministic algorithm has a competitive ratio at least k for the TCT problem. For the online makespan problem the paper considers a class of algorithms called *non-skipping*[3]. It is shown, that any non-skipping algorithm $\mathcal{A}$, that has a competitive ratio c

[2] Formal definitions are given in Section 2.2.

[3] An algorithm $\mathcal{A}$ is non-skipping if the server always services a request it is in position to serve, that is, the server never skips a request that can be serviced without incurring setup cost.

when the processing times are zero, has competitive ratio *at least* c for arbitrary processing times.

A problem related to our makespan problem is the *Shortest Common Supersequence (SCS)*[7]. Here we are given a collection of strings over a fixed alphabet, Σ, and the goal is to find a shortest common supersequence, such that all the given strings appear as subsequences in the common supersequence. An instance of our makespan problem can be described in terms of the SCS problem as follows: each sequence of requests of client i, $1 \leq i \leq k$, is a *string*; the alphabet Σ is the set of banks $1, \ldots, M$. Indeed, a schedule is a supersequence that includes all the strings as subsequences. Note however, that the objective function in our makespan problem *differs* from the objective function of the SCS in how it treats the case where the same letter appears consequently in a string. (In the makespan problem, repetition of the same letter do not "count", since we attempt to minimize the total number of setups). Yet, as we indicate below, some of the results obtained for the SCS apply to the makespan problem.

Maier studied in [11] the SCS problem and showed that the problem is NP-complete when $|\Sigma| \geq 5$. Jiang and Li derived stronger hardness results in [9]; they also gave algorithm that produce solutions close to the optimal when the strings are random. A theoretical and empirical study of approximation algorithms for SCS with various degrees of *greediness* is given by Foulser et al. in [5]. The paper [5] also presents a *dynamic programming (DP)* algorithm for the SCS, when the number of strings is some fixed constant (see also in [8]). This gives a DP algorithm for our makespan problem, when the number of clients, k, is some constant. Finally, an approximation algorithm of Fraser and Irving [6] for the SCS problem yields a $\frac{k+3}{4}$ approximation algorithm for our makespan problem.

Bhatia et al. studied in [3] the *loading time scheduling problem (LTSP)*, where n tasks need to be scheduled on a set of machines; the tasks are given as vertices of a directed acyclic graph: an edge (i, j) implies that task i has to be completed before the execution of task j can be started. For each task there is a processing time; also, when we perform a set of tasks consequently on the same machine, the first of these tasks incurs a *loading* time. At any time, only one task can be processed; each task comes with a specified subset of machines by which it can be processed. Our goal is to select for each task one machine on which it will be processed, and an order of *non-preemptive*[4] execution for the tasks such that the makespan is minimized.

Note that our offline makespan problem is a special case of the LTSP, in which the precedence graph of the tasks (= requests) is a set of *chains*, all tasks have equal processing times, λ, and each task can run on a single machine (= the bank to which the corresponding request has to access). The results in [3] imply that

- the problem of finding the minimal makespan is NP-complete and MAX SNP-hard for $M \geq 4$

[4] i.e., with no interruption.

- for any constant δ, there does not exist a polynomial time algorithm that has an approximation factor of $(\log k)^\delta$, unless NP$\subseteq$DTIME$(k^{O(\log\log k)})$
- there does not exist a polynomial time M^α-approximation algorithm for some constant α, unless P=NP.

Also, an $M-$approximation algorithm for LTSP yields an $M-$approximation for our makespan problem.

1.5 Main Results

The following are the main results of this work.

- For the offline case
  - We show that the total completion time problem is NP complete.
  - We give an $O(\min(k,\ M))-$approximation algorithm for the makespan problem.
  - We give an $O\left(\min\left(k,\ M+\frac{(k-M)\lambda}{\mu}\right)\right)-$approximation algorithm for the TCT problem.
  - We develop dynamic programming algorithms for the TCT and the MRT measures.
- In the online case, lower bounds of $\Omega(\frac{\mu}{\lambda})$ are shown for all three cost functions. For each measure, we also show that the lower bound is tight by introducing online algorithms that achieve the upper bound of $O(\frac{\mu}{\lambda})$.
- We present experimental results, which compare the performance of the offline and online algorithms studied in this work, on input instances that simulate a real system environment.

Our proofs of the lower bounds on the competitive ratios of deterministic algorithm for the online makespan and the TCT problems (Theorems 4.1 and 5.1) draw some ideas from the proofs of lower bounds for the basic $k-$client problem [1]. The proof technique used for showing the upper bound on the competitive ratio of the Round-Robin (RR) algorithm (Theorems 5.2 and 5.4) is similar to the technique used in [13] for bounding the competitive ratio of RR in non-clairvoyant scheduling.

1.6 Organization of this Work

In Chapter 2 we give some definitions and notation. Chapter 3 discusses the offline version of our problem and presents our results for the makespan (Section 3.1) the TCT (Section 3.2) and the MRT measure (Section 3.3).

In Chapter 4 we describe the results for the online makespan problem. Chapter 5 deals with the online TCT problem, and Chapter 6 deals with the online MRT problem.

In Chapter 7 we present experimental results obtained for algorithms for the TCT and the MRT measures. Finally, in Chapter 8 we summarize the contribution of this work and discuss some directions for future work.

Chapter 2

Preliminaries

2.1 Optimization Problem and Performance Bounds

We start with the definition of *optimization problem* which is a cost minimization. An optimization problem $\mathcal{P}$ consists of a set $\mathcal{I}$ of inputs and a cost function $\mathcal{C}$. For any input I there is a set of feasible outputs $F(I)$, and associated with each feasible solution O in $F(I)$ is a positive real, $C(I, O)$, representing the cost of the output O with respect to the input I.

Given any legal input I, an algorithm $\mathcal{A}$ for an optimization problem $\mathcal{P}$ computes a feasible output $\mathcal{A}[I] \in F(I)$. The cost associated with the feasible output is denoted by $\mathcal{A}(I) = C(I, \mathcal{A}[I])$. An optimal algorithm OPT satisfies for all legal inputs I,

$$OPT(I) = \min_{O \in F(I)} C(I, O).$$

An algorithm $\mathcal{A}$ is an *r-approximation algorithm* for a minimization problem $\mathcal{P}$ if for any input I,

$$\mathcal{A}(I) \leq r \cdot OPT(I).$$

Optimization problems in which the input is accepted in an online manner and in which the output must be produced online are called *online problems*. Nevertheless, even for intrinsically online problems, we could hypothesize an offline algorithm with clairvoyant abilities.

An online algorithm $\mathcal{A}$ is c-competitive if there is a constant α such that for all input sequences I,

$$\mathcal{A}(I) \leq c \cdot OPT(I) + \alpha.$$

If $\mathcal{A}$ is c-competitive, we say that $\mathcal{A}$ attains a competitive ratio c. An algorithm is called competitive if it attains a constant competitive ratio c. The infimum over the set of all values c such that $\mathcal{A}$ is c-competitive is called the competitive ratio of $\mathcal{A}$, $\mathcal{R}(\mathcal{A})$. Formally,

$$\mathcal{R}(\mathcal{A}) = \inf_{c}\{c : \ \mathcal{A} \ is \ c - competitive\}.$$

2.2 Definitions and Notation

There are k clients (processors) and m memory banks. Every client $i, 1 \leq i \leq k$ has a sequence of n_i access requests to memory banks; we denote by n_{max} the length of the sequence with maximal number of requests, i.e., $n_{max} = \max_{1 \leq i \leq k} n_i$. For any $1 \leq i \leq k,\ 1 \leq j \leq n_i, \quad r_i(j) \in \{1, \ldots, M\}$ is the memory bank accessed by client i in its j-th request. There is a single arbiter that services client requests (i.e., connects the bus to the requested memory bank). Thus, only one request can be serviced by the arbiter at any time. No preemptions are allowed while servicing a request. The $(j+1)$-th request of client i is released when the j-th request of this client has been serviced. Thus, the arbiter always chooses from a *frontier* which consists of at most k requests.

The point in time in which the service of a request r is completed under a given schedule, $\mathcal{S}$, is denoted by $t(r, \mathcal{S})$. For short, when the schedule is given, we use the notation $t(r)$; $t_s(r, \mathcal{S})$ is the time that it takes the arbiter to service a request r under a schedule $\mathcal{S}$ (for short, $t_s(r)$); $t_s(r)$ can get one of two values, λ or μ, where $\mu > \lambda$. When the service of the request r does not require to switch the bus from one memory bank to another, the service time of r equals to λ, i.e., the required *processing time* for r. Otherwise, servicing the request r involves a *setup time*, therefore $t_s(r)$ equals to μ (the setup time is $(\mu - \lambda)$).

For any $1 \leq s, i \leq k,\ 1 \leq p \leq n_s$ and $1 \leq j \leq n_i$, if the request $r_s(p)$ is serviced immediately after $r_i(j)$ then:

$$t_s(r_s(p)) = \begin{cases} \lambda & r_s(p) = r_i(j) \\ \mu & \text{otherwise} \end{cases}$$

Note that we can represent the memory banks by a graph $G = (V, E)$, where the vertex v_i corresponds to the i-th memory bank, and there is an edge v_i, v_j of length μ for any $1 \leq i, j \leq M$.In addition, since in our model, a request that does not requires switching to another memory bank is processed in λ time units, we add in each vertex $v_i,\ 1 \leq i \leq M$ a self-loop of length λ. Thus, in terms of the $k-$ client problem, we handle the case where k clients generate request in a set of vertices that form a *clique* of size M.

We define $c_i(\mathcal{S}) = c_i$ as the completion time of client i under the schedule $\mathcal{S}$, i.e., $c_i = t(r_i(n_i))$. $\hat{c}_i$ is the minimal time required for servicing all the requests of client i, if this client were alone in the system.

2.3 Performance measures

In this work we consider several performance measures.

- The *makespan* of a given schedule is $\sum_{1 \leq i \leq k} \sum_{1 \leq j \leq n_i} t_s(r_i(j))$. In words, it measures the total time spent by the arbiter in order to fulfill all the requests to memory banks. This cost function is generally used in the k-server problem to measure the total distance that the server has to move to service all the requests. We define $\mathcal{A}_{ms}(I)$ to be the makespan obtained by an algorithm $\mathcal{A}$ for an input sequence I. In particular, $OPT_{ms}(I)$ is the makespan obtained by an optimal algorithm, OPT.

- The *total completion time (TCT)* cost function will be defined as $\sum_{1 \leq i \leq k} c_i$. We define $\mathcal{A}_{tct}(I)$ to be the total completion time obtained by an algorithm $\mathcal{A}$ for an input sequence I. In particular, $OPT_{tct}(I)$ is the total completion time obtained by an optimal algorithm, OPT.

- The *maximum response time (MRT)* of a given schedule is the maximum time that any request waits from the moment it arrives till the moment the arbiter starts to process it (including the setup time, if needed). For convenience, assume that each client i has a request 0, and that $t(r_i(0)) = 0$, then MRT is defined as

$$\max_{1 \leq i \leq k,\ 1 \leq j \leq n_i} \{t(r_i(j)) - t(r_i(j-1)) - \lambda\}.$$

  We define $\mathcal{A}_{mrt}(I)$ to be the maximum response time obtained by an algorithm $\mathcal{A}$ for an input sequence I. In particular, $OPT_{mrt}(I)$ is the maximum response time obtained by an optimal algorithm, OPT.

Chapter 3

The Offline $k-$client Problem

3.1 Makespan

W.l.o.g. we assume in the discussion of the makespan problem that consecutive requests of client i are to *different* memory banks, otherwise we can transform the input I to an input I' in which there are no consecutive requests to the same bank by client i. We get that

$$OPT_{ms}(I) = OPT_{ms}(I') + l\lambda$$

and

$$\mathcal{A}_{ms}(I) = \mathcal{A}_{ms}(I') + l\lambda$$

for some $l \geq 1$. Since we are concerned with upper bounds on the performance ratio of our algorithms, we get that

$$\frac{\mathcal{A}_{ms}(I)}{OPT_{ms}(I)} = \frac{\mathcal{A}_{ms}(I') + l\lambda}{OPT_{ms}(I') + l\lambda} \leq \frac{\mathcal{A}_{ms}(I')}{OPT_{ms}(I')}$$

Hence, we can use the input I' for obtaining upper bounds.

3.1.1 Approximation Algorithm

Consider the Greedy Round-Robin (GRR) algorithm, which is a variant of Round-Robin (RR). In each phase $j, 1 \leq j \leq n_{max}$ GRR groups the j-th requests of all the clients according to the memory banks they refer to. Then, GRR randomly chooses a group of unserviced requests and handles the requests in this group sequentially. The j-th phase terminates when GRR completes handling the j-th requests of all clients. Clearly, the maximum number of setups that GRR incurs in each phase is $\min(M, k)$.

Theorem 3.1 *The algorithm GRR has the approximation ratio* $\min(M, k)$.

Proof: The proof follows from the fact that the minimal number of setups under OPT is at least n_{max}, while GRR incurs at most $\min(M, k)n_{max}$ setups. □

Now we show that OPT has a "greedy" nature, i.e., after servicing a request to any bank m_i, OPT continues to service all possible requests to that bank, before passing to other banks. The proof is immediate once we see that an algorithm that minimizes the makespan also minimizes the number of setups and vice versa. Thus, OPT decision policy may be described as a sequence $m_{i_1}, \ldots, m_{i_f}$, where m_{i_j}, $j \geq 1$ is an index of memory bank, $1 \leq m_{i_j} \leq M$, f is the number of setups that OPT had to perform and the order in this sequence defines the order of execution of *batches* of requests for the same bank. Clearly, knowing the number of setups OPT had to pay, f, get that the makespan is $OPT_{ms}(I) = \lambda k \sum_{i=1}^{k} n_i + (\mu - \lambda)f$.

Trying to obtain a better ratio than M, we examined the next question: If the number of memory banks is at most as the number of clients, what will be an upper bound on the number of the setups for the input sequence, where each client has only 2 requests? Clearly, achieving the upper bound $B \leq 2M - 2$ would allow us to obtain an approximation ratio lower than $M - 1$ for any input sequence, by just using an algorithm that achieves an approximation ratio B for first two requests of all the clients, then for third and forth requests of all the clients and so on. However, an input sequence that we found shows that the upper bound on the number of setups for an input sequence where each client has two requests is $2M - 1$. An example of such an input sequence with $k = 20$ and $M = 5$, is presented in Figure 3.1 (the requests of each client i, $1 \leq i \leq k$ are given in the columns).

clients	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
	1	2	3	4	2	3	4	5	1	3	4	5	1	2	4	5	1	2	3	5
	5	5	5	5	1	1	1	1	2	2	2	2	3	3	3	3	4	4	4	4

Figure 3.1: Worst case for $k = 20, M = 5$

The case where $M = 2$ can be solved polynomially by the greedy algorithm, that will start servicing any input instance I from any of the clients that have the maximal number of requests in I. Note that in this case the length of the optimal scheduling will be either n_{max} or $n_{max} + 1$.

3.1.2 Optimal Algorithms

Theorem 3.2 *The makespan problem can be solved optimally in polynomial time when k is some fixed constant.*

Proof: We show that the makespan problem can be solved recursively using dynamic programming. We define $C(j_1, \ldots, j_k, p)$ to be the minimal makespan for handling the first j_i requests of client i, $1 \leq i \leq k$, where p is the index of the client that was serviced last. The recursive formula for finding $C(j_1, \ldots, j_k, p)$, $1 \leq p \leq k$, is given by

$$C(j_1, \ldots, j_k, p) = \min_{1 \leq s \leq k} [C(j'_1, \ldots, j'_k, s) + t_s(r_p(j_p))] \tag{3.1}$$

where,

$$j'_i = \begin{cases} j_i - 1 & \text{if } i = p \\ j_i & \text{otherwise} \end{cases} \tag{3.2}$$

The minimum makespan is obtained by computing $\min_{1 \leq p \leq k} C(n_1, \ldots, n_k, p)$. Using (3.1), the cost of each configuration is calculated in $O(k)$ steps; the possible number of configurations is $O(kn^k_{max})$. Hence, the overall running time of the algorithm is $O(k^2 n^k_{max})$. Clearly, when we have the cost (or overall completion time) calculated for all possible configurations, we can find the order of servicing client requests that leads to the desired result. This can be done by maintaining the indices of the minimal configurations while computing the minimum makespan. We initialize the algorithm by taking $C(j_1, \ldots, j_k, p) = \mu$, $1 \leq p \leq k$

where,

$$j_i = \begin{cases} 1 & \text{if } i = p \\ 0 & \text{otherwise} \end{cases} \tag{3.3}$$

In order to keep tracking of legal configurations only, we take $C(j_1, \ldots, j_k, p) = \infty$, if at least one of the indices is negative or $j_i = 0 \; \forall \; 1 \leq i \leq k$.

The above algorithm runs in the time that is polynomial in n_{max}, but exponential in k, therefore it is suitable for the case where k is some fixed constant. □

We now show, that the makespan problem is solvable in polynomial time also when k is accepted as part of the input; however, we require that M and n_{max} are fixed constants.

Theorem 3.3 *The makespan problem can be solved optimally in polynomial time when M and n_{max} are some fixed constants.*

Proof: Let $c = n_{max} \geq 1$ be some constant. If $M \geq k$, then we get the result of Theorem 3.2. Thus we assume that $M < k$. Note, that for any input I,

$$\begin{aligned} OPT_{ms}(I) &\leq GRR_{ms}(I) \\ &\leq \min(M, k) n_{max} \\ &= Mc \end{aligned}$$

Hence, an optimal schedule can be found in $O(kcM^{Mc})$ steps, by using an exhaustive search over all the possible sequences of memory banks, whose length is Mc. The number of such sequences is M^{Mc}. The feasibility of a given sequence can be checked in $O(kc)$ steps, by scheduling the client requests in the prescribed order (i.e., taking all the available requests for bank j when this bank appears in the sequence). Therefore, we get that the overall complexity of finding an optimal schedule in $O(kcM^{Mc})$. $\square$

3.2 Total Completion Time

3.2.1 Hardness Result

We now give a proof of hardness for the TCT problem.

Theorem 3.4 *The following problem is NP-complete*

Input: A set of k clients; client i, $1 \leq i \leq k$, has n_i requests, and M memory banks; an integer $w > 0$.

Question: Can all the requests be scheduled such that $\sum_{i=1}^{k} c_i \leq w$?

Proof: The proof is by reduction from the makespan problem. Given an instance, I, of the makespan problem, with k clients and M memory banks, we construct an instance I', with $k' = k_1 + k_2$ clients, where $k_1 = k$, $k_2 = \mu k^2 n_{max}$ and $M' = M + k_2(n_{max} + 1)$ memory banks. We may assume that $\mu \in \mathbf{N}$ is some small constant, since the makespan problem is known to be NP-complete even for the case where $\mu = 1$, $\lambda = 0$.

Each of the first k clients has n_i requests to any of the first M memory banks, as in the original instance, I; for $1 \leq j \leq k_2$, client $(k + j)$ has a chain of $n_{max} + 1$ requests: the l-th request $1 \leq l \leq n_{max} + 1$ is to access the bank $(M + (j - 1)(n_{max} + 1) + l)$.
Let

$$w' = (k + k_2)w + \mu \sum_{i=1}^{k_2} i(n_{max} + 1).$$

In the following we show that $OPT_{ms}(I) \leq w$ iff $OPT_{tct}(I') \leq w'$.

Claim 3.1 *If $OPT_{ms}(I) \leq w$ then $OPT_{tct}(I') \leq w'$.*

Proof: If $OPT_{ms}(I) \leq w$, then we can schedule first all the requests of clients $1, \ldots, k$. These clients will complete by time w. Then, we will process sequentially the requests of clients $k + 1, \ldots, k'$. The total completion time of the schedule is bounded by

$$(k+k_2)w + \mu \sum_{i=1}^{k_2} i(n_{max}+1) = w'.$$

□

Claim 3.2 *If* $OPT_{ms}(I) > w$ *then* $OPT_{tct}(I') > w'$.

Proof: Note that since we need $\mu(n_{max}+1)$ time units to complete servicing each of the clients $k+1 \le j \le k'$, in any optimal schedule the requests of clients $1, \ldots, k$ are serviced before handling any of the "new" clients. More specifically, suppose that in some optimal schedule, we start handling the first request of client j, $k+1 \le j \le k'$ before servicing the last request of client i, $1 \le i \le k$; then, by interchanging the execution order of the two requests, the total completion time of the schedule can only decrease.

Now, suppose that

$$OPT_{ms}(I) > w \tag{3.4}$$

Then, by the above discussion, we get that

$$\begin{aligned} OPT_{tct}(I) &\ge w + 1 + k_2(w+1) + \mu \sum_{i=1}^{k_2} i(n_{max}+1) \\ &= w + 1 + k_2 w + \mu k^2 n_{max} + \mu \sum_{i=1}^{k_2} i(n_{max}+1). \end{aligned}$$

Observe, that $OPT_{ms}(I) \le \mu k n_{max}$. Thus, from (3.4) we have

$$OPT_{tct} \ge w + 1 + k_2 w + kw + \mu \sum_{i=1}^{k_2} i(n_{max}+1) > w'.$$

□

Clearly, the reduction is polynomial in k, n_1, $\ldots$, n_k. This completes the proof. □

3.2.2 An Optimal Algorithm

Theorem 3.5 *The TCT problem can be solved optimally in polynomial time when* k *is a fixed constant.*

Proof: We now present a dynamic programming algorithm which finds an optimal solution in polynomial time when the number of clients, k is some fixed constant. The algorithm is similar to the dynamic programming algorithm given for the makespan problem in Section 3.1.2. Denote by $C(j_1, \ldots, j_k, p)$ the minimal total completion time after handling the first j_i requests of client i, $1 \le i \le k$, where client p was serviced last.

Let $f(j'_1, \ldots, j'_k) = k - \sum_{i=1}^{k} 1(j'_i, n_i)$, where

$$1(j'_i, n_i) = \begin{cases} 1 & j'_i = n_i \\ 0 & \text{otherwise} \end{cases}$$

$f(j'_1, \ldots, j'_k)$ denotes the number of clients whose requests have not been completed yet. Then

$$C(j_1, \ldots, j_k, p) = \min_{1 \leq s \leq k} [C(j'_1, \ldots, j'_k, s) + t_s(r_p(j_p)) f(j'_1, \ldots, j'_k)] \tag{3.5}$$

where j'_i is as given in (3.2).

Using (3.5), the cost of each configuration is calculated in $O(k)$ steps; the possible number of configurations is $O(kn^k_{max})$. Hence, the overall running time of the algorithm is $O(k^2 n^k_{max})$. □

3.2.3 Approximation Algorithms

We now consider the more general case where k can be accepted as part of the input. Assume first that there are no consecutive requests of any client to the same memory bank. For this case we derive for the algorithm GRR a performance ratio of $\min(M + \frac{(k-M)\lambda}{\mu},\ k)$.

Theorem 3.6 *If for any $1 \leq i \leq k$, $r_i(j) \neq r_i(j+1)$, $\forall\ 1 \leq j \leq n_{i-1}$, then GRR achieves the approximation ratio $\min(M + \frac{(k-M)\lambda}{\mu},\ k)$ for the TCT problem.*

Proof: Note that since $r_i(j) \neq r_i(j+1)\ \forall\ 1 \leq j \leq n_{i-1}$, $\hat{c}_i = \mu n_i$. Thus, for any instance, I,

$$OPT_{tct}(I) \geq \mu \sum_{i=1}^{k} n_i \tag{3.6}$$

Now, for a given instance, I, we sort the n_is and renumber the clients such that $n_1 \leq n_2 \leq \cdots \leq n_k$. Recall, that GRR operates in phases; in each phase, j, the j-th requests of all clients are serviced. We group phases into stages. The first stage consists of phases $1, \ldots, n_1$, i.e., until we complete handling all the requests of client 1. The contribution of this stage to the TCT of GRR is at most

$$\mu \min(M, k) n_1 k + \lambda \max(k - M, 0) n_1 k \tag{3.7}$$

Indeed, in each phase of stage 1 we have at most $\min(M, k)$ setups, and at most $\max(k - M, 0)$ requests whose services incur only the processing time, λ. This contribution is multiplied by the number of non-completed clients, which equals to k.

Similarly, stage i ends when we complete handling the remaining requests of client i, $2 \leq i \leq k$. Thus, the overall contribution of stage i to the TCT of GRR is at most

$$\mu \min(M,k)(n_i - n_{i-1})(k-i+1) + \lambda \max(k-M,0)(n_i - n_{i-1})(k-i+1) \quad (3.8)$$

From (3.7) and (3.8) we get that

$$\begin{aligned} GRR_{tct}(I) &\leq (\mu \min(M,k) + \lambda \max(k-M,0))\,[kn_1 + \sum_{i=2}^{k}(n_i - n_{i-1})(k-i+1)] \\ &= [\mu \min(M,k) + \lambda \max(k-M,0)] \sum_{i=1}^{k} n_i \end{aligned}$$

Thus,

$$\begin{aligned} \frac{GRR_{tct}(I)}{OPT_{tct}(I)} &\leq \min(M,k) + \frac{\lambda}{\mu}\max(k-M,0) \\ &= \min(M + \frac{(k-M)\lambda}{\mu},\ k) \end{aligned}$$

□

We now introduce the *Shortest Client First algorithm (SCF)* that achieves a $\frac{k+1}{2}$ approximation ratio. The algorithm SCF mimics the operation of the algorithm Shortest Processing Time First (SPTF) [10] that is known to be the optimal for scheduling a set of independent jobs, so as to minimize the total completion time. The algorithm SPTF sorts a given set of k jobs in non-increasing order by their execution times, and then the jobs are scheduled sequentially (i.e., each job is run to completion, uninterrupted). Indeed, the intuition in to schedule first short jobs, so as to minimize the delay caused to the jobs that are still waiting to be scheduled. Here, we consider each client as a "job".

The algorithm SCF first finds the execution time of each job, that is, the time required for processing all the requests of client i, $1 \leq i \leq k$; then, we sort the list of clients in non-increasing order by the processing time of the corresponding "jobs". Finally, the clients are serviced sequentially by the list, such that all the requests of client i are completed before we turn to service client $(i+1)$. A formal description is given below.

The SCF Algorithm

1. For every $1 \leq i \leq k$ find $\hat{c}_i$.
2. Sort the client list in non-decreasing order by the $\hat{c}_i$'s. Renumber the clients according to their positions in this sorted list, such that

$$\hat{c}_1 \leq \hat{c}_2 \leq \cdots \leq \hat{c}_k \quad (3.9)$$

3. Schedule the clients sequentially using the sorted list, i.e., start servicing the request of client $(i+1)$ once all the requests of client i have been handled.

The running time of SCF is dominated by the sorting phase, which requires $O(k \lg k)$ steps.

Theorem 3.7 *The SCF algorithm yields a $\frac{k+1}{2}$ approximation ratio for the TCT problem.*

Proof: Note that for any instance, I,

$$SCF_{tct}(I) = k\hat{c}_1 + (k-1)\hat{c}_2 + \cdots + \hat{c}_k.$$

Suppose that the TCT of some optimal algorithm, OPT, is given by

$$OPT_{tct}(I) = c_1 + \cdots + c_k.$$

Clearly, $c_i \geq \hat{c}_i$. Hence,

$$\frac{SCF_{tct}(I)}{OPT_{tct}(I)} = \frac{k\hat{c}_1 + (k-1)\hat{c}_2 + \cdots + \hat{c}_k}{c_1 + c_2 + \cdots + c_k} \tag{3.10}$$

$$\leq \frac{k\hat{c}_1 + (k-1)\hat{c}_2 + \cdots + \hat{c}_k}{\hat{c}_1 + \hat{c}_2 + \cdots + \hat{c}_k}. \tag{3.11}$$

Let $\sum_{i=1}^{k} \hat{c}_i = R$ for some $R \geq 1$, then from (3.11), the numerator is maximized when $\hat{c}_i = R/k$. Hence,

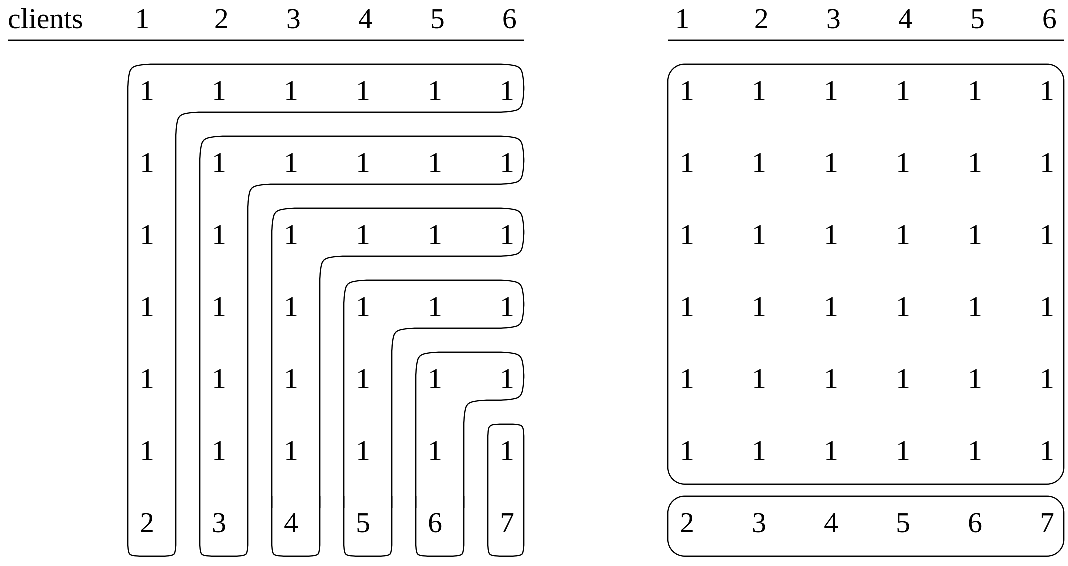
$$\frac{SCF_{tct}(I)}{OPT_{tct}(I)} \leq \frac{\sum_{j=1}^{k} (k-j+1)\frac{R}{k}}{R} = \frac{k+1}{2}.$$

□

3.3 Maximal Response Time

In this section we present a dynamic programming algorithm for solving optimally the MRT problem, when k is a fixed constant. We first consider a natural greedy algorithm, and show that it can perform poorly, even if $\lambda = 0$.

Example 3.4 *Consider an instance I in which $k = 6$ and $M = 7$, as given in Figure 3.2. The greedy algorithm, Gr, handles all the requests to bank 1 before proceeding to the other requests. Thus, we get that $Gr_{mrt}(I) = k\mu = 6\mu$. Under an optimal algorithm we get $OPT_{mrt}(I) = 2\mu$. The following algorithm, which*



(a.) (b.)

Figure 3.2: (a) An Optimal algorithm (b) The Greedy algorithm

achieves the optimum, operates in phases. In phase i, $1 \leq i \leq 6$ the algorithm schedules as a sequence all the remaining requests in row i (starting with the rightmost request) and all the remaining requests in column i. Thus, for example, in the first phase, we schedule the first requests of all clients (starting from $r_6(1)$) and then complete handling the requests of client number 1.

Note that the above example distinguishes the MRT measure from the makespan and TCT measures discussed earlier: for these measures, if $\lambda = 0$, *greediness* is a desired property of an optimal algorithm, i.e., any optimal algorithm for the makespan or the TCT problem handles all the available requests to the same bank, before it switches to a different bank.

We now show that MRT can be solved optimally using dynamic programming.

Theorem 3.8 *The MRT problem can be solved optimally in polynomial time when k is a fixed constant.*

Proof: Consider the following dynamic programming algorithm. Denote by $C((j_1,\ l_1,\ m_1),\ldots,(j_k,\ l_k,\ m_k),p)$ the minimal MRT for handling the first j_i requests of client i, $1 \leq i \leq k$, such that a request of client p was serviced last, and the time since the j_i-th request of client i was serviced is $(l_i\lambda + m_i\mu)$, for some $m_i,\ l_i \geq 0$. Let $N = \sum_{i=1}^{k} n_i$.

The recursion for finding $C((j_1,\ l_1,\ m_1),\ldots,(j_k,\ l_k,\ m_k),p)$ is given by

$$C((j_1,\ l_1,\ m_1),\ldots,(j_p,\ 0,\ 0),\ldots,(j_k,\ l_k,\ m_k),p) = \tag{3.12}$$
$$\min_{\substack{1\le s\le k\\ 1\le l'_p\le N\\ 1\le m'_p\le N}} \left[max\left\{C((j'_1,\ l'_1,\ m'_1),\ldots,(j_p-1,\ l'_p,\ m'_p),\ldots,(j'_k,\ l'_k,\ m'_k),s), \max_{1\le i\le k}(l_i\lambda+m_i\mu)\right\}\right]$$

where for $i \neq p$,

$$(j'_i,\ l'_i,\ m'_i) = \begin{cases} (j_i,\ l_i-1,\ m_i) & \text{if } t_s(r_p(j_p)) = \lambda \\ (j_i,\ l_i,\ m_i-1) & \text{if } t_s(r_p(j_p)) = \mu \end{cases}$$

The optimal solution for a given instance, I, is found by taking

$$\min_{\substack{1\le p\le k\\ 1\le l'_i\le N\\ 1\le m'_i\le N}} C((n_1,\ l_1,\ m_1),\ldots,(n_k,\ l_k,\ m_k),p).$$

We initialize the algorithm by taking

$$C((j_1,\ l_1,\ m_1),\ldots,(j_k,\ l_k,\ m_k),p) = \mu,\ 1\le p\le k$$

where,

$$(j_i,\ l_i,\ m_i) = \begin{cases} (1,\ 0,\ 0) & \text{if } i=p \\ (0,\ 0,\ 1) & \text{otherwise} \end{cases}$$

In order to consider legal configurations only, we define

$$C((j_1,\ l_1,\ m_1),\ldots,(j_k,\ l_k,\ m_k),p) = \infty,$$

if at least one of the indices is negative or $j_i = 0\ \forall\ 1\le i\le k$. Using (3.12), calculating the cost for each configuration requires $O(N^2k)$ steps, and since the possible number of configurations is $O(n_{max}^k N^{2k})$, the overall running time of the algorithm is $O(kn_{max}^k N^{2(k+1)})$. Hence, we get a polynomial time optimal algorithm, when k is a fixed constant. $\square$

Chapter 4

The Online Makespan Problem

The trivial upper bound on the competitive ratio of any deterministic algorithm is μ/λ. Indeed, for servicing any request r, the minimal cost incurred by an optimal algorithm is λ, and the maximal cost incurred by any algorithm is μ. In the following we derive a matching lower bound on the competitive ratio of any deterministic algorithm for the makespan problem.

Theorem 4.1 *Any online deterministic algorithm for the makespan problem has a competitive ratio* $\Omega\left(\frac{\mu}{\lambda}\right)$.

Proof: We show that for any online algorithm $\mathcal{A}$, there exists an instance I which consists of $(\frac{\lg k}{2}+1)k$ requests, $k=2^l$, such that the online algorithm has to pay μ in order to service each request; the optimal offline algorithm pays only k times μ and services the remaining requests at the cost of λ for each. Therefore, for arbitrary k, the competitive ratio of online algorithm can be lower bounded by

$$\frac{\dfrac{\lg k}{2}k\mu}{\dfrac{\lg k}{2}k\lambda + k\mu} = \frac{\dfrac{\lg k}{2}\mu}{\dfrac{\lg k}{2}\lambda + \mu}$$

The instance I is constructed as follows. Initially, each client, i, has a single request to the i-th memory bank, $1 \le i \le k$, $k = 2^l$ for some integer $l \ge 1$. The instance I is constructed in $l = \lg k$ phases. For the first phase, assume, w.l.o.g., that $\mathcal{A}$ handles first the requests $r_{\frac{k}{2}+1}(1), \ldots, r_k(1)$. Then, for each of the remaining clients we add a second request, such that $r_i(2) = r_{i+\frac{k}{2}}(1)$. This completes the first phase. Now, the algorithm $\mathcal{A}$ has an instance with $\frac{k}{2}$ clients, each having 2 requests and each request is to a *different* bank. Suppose that $\mathcal{A}$ completes processing first the requests of client $\frac{k}{4}+1, \ldots, \frac{k}{2}$; now we add to the sequence of requests of client i, $1 \le i \le \frac{k}{4}$, all the requests of client $i + \frac{k}{4}$.

More generally, at the beginning of phase j, $1 \le j \le \lg k$, $\mathcal{A}$ has $\frac{k}{2^{j-1}}$ clients, whose requests have not been completed. Once $\mathcal{A}$ completes processing the requests of $k/2^j$ clients, we add the sequence of requests of each of those clients to one of the $k/2^j$ remaining clients. Consequently, at any time $\mathcal{A}$ faces a set of requests to *distinct* banks (see Example 4.1). The total number of requests in I is

$$1 \cdot \frac{k}{2} + 2 \cdot \frac{k}{4} + 4 \cdot \frac{k}{8} + \cdots + \frac{k}{2} \cdot 1 + k = k + \frac{k}{2} \lg k.$$

Hence

$$\mathcal{A}_{ms}(I) = (k + \frac{k}{2} \lg k)\mu.$$

Note that from the construction of I, client 1 has k requests (to k distinct banks). An optimal offline algorithm can incur the cost $k\mu + \frac{k}{2} \lg k\lambda$ by handling sequentially the requests of client 1; whenever there are pending request to the bank specified by $r_1(j)$, these requests will be processed, before OPT turns to handle $r_1(j+1)$.

To complete the proof we use the next claim adopted from [1].

Claim 4.1 *[1] If $r_1(j) = r_i(p)$ for some $i \neq 1$, then by the time OPT processes $r_1(j)$, OPT has processed the requests $r_i(1), \ldots, r_i(p-1)$.*

¿From Claim 4.1 we get that OPT indeed can complete handling all the requests in I by using exactly k setups. Hence,

$$\frac{\mathcal{A}_{ms}(I)}{OPT_{ms}(I)} = \frac{(k + \frac{k}{2} \lg k)\mu}{k\mu + \frac{k}{2} \lg k\lambda}.$$

For sufficiently large value of k we get statement of the theorem. □

Example 4.1 *For $1 \leq i \leq s$, $r_i(1) = 1$. Suppose that $\mathcal{A}$ is an online deterministic algorithm. W.l.o.g., assume that $\mathcal{A}$ services first $r_5(1)$, $r_6(1)$, $r_7(1)$ and $r_8(1)$. Then we add to each of the clients $1 \leq i \leq 4$ a second request, such that $r_i(2) = i + 4$.*

Now, suppose that $r_3(2)$, $r_4(2)$ are processed by $\mathcal{A}$ before $r_1(2)$, $r_2(2)$, then now we add two requests to each of the clients 1, 2 as follows:

$$\begin{array}{lllllllll} r_1(3) & = & r_3(1) & = & 3; & r_1(4) & = & r_3(2) & = & 4 \\ r_2(3) & = & r_4(1) & = & 4; & r_2(4) & = & r_4(2) & = & 8 \end{array}$$

Finally, assume that $r_2(4)$ is processed by $\mathcal{A}$ before $r_1(4)$, then we add for client 1 four requests, by copying the four requests of client 2. Thus,

$$r_1(5) \;=\; 2; \;\; r_1(6) \;=\; 4; \;\; r_1(7) \;=\; 6; \;\; r_1(8) \;=\; 8\;.$$

The resulting instance is depicted in Figure 4.1. Note, that an optimal offline algorithm incurs the cost

$$OPT_{ms}(I) = 8\mu + 12\lambda.$$

This is done by handling the requests in I by the order they appear in sequence of client 1; for each bank we handle all the pending requests before switching to a different bank. On the other hand, $\mathcal{A}$ incurs the cost

$$\mathcal{A}_{ms}(I) = 20\mu.$$

Hence,

$$\frac{\mathcal{A}_{ms}(I)}{OPT_{ms}(I)} = \frac{20\mu}{8\mu + 12\lambda} = \Omega\left(\frac{\mu}{\lambda}\right).$$

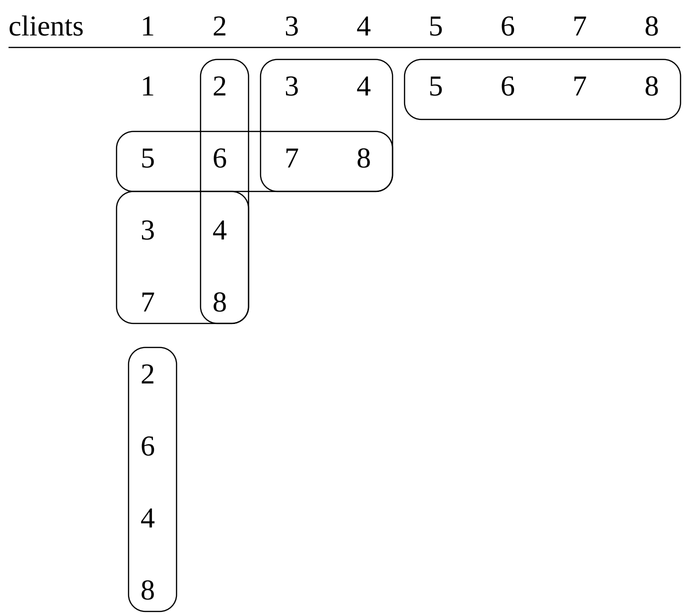


Figure 4.1: Example of an input instance in the proof of Theorem 4.1.

Chapter 5

The Online TCT Problem

The TCT cost function is somewhat harder to tackle, compared to the makespan. Indeed, when analyzing algorithms for the makespan problem, we could assume, that any "reasonable" algorithm (and in particular an optimal one), will possess the *greediness* property, that is, any optimal algorithm for the makespan problem schedules first all the existing requests for some bank i, before moving to another bank. This would minimize the number of setups, which is a main factor in the makespan cost function. However, when our objective function is the TCT, reducing the number of setups may result in non-optimal schedule.

We note, that any algorithm for the TCT problem has to minimize *two* factors:

- the number of setups, and
- the number of interruptions while servicing the requests of specific client.

The parameters which determine, whether one factor should be preferred over the other are λ (is $\lambda = 0$?), the ratio $\frac{\mu}{\lambda}$ (when $\lambda \neq 0$) and the number of requests for each client.

Obviously, when $\lambda = 0$ any optimal algorithm for the TCT problem is greedy. Now consider the case where $\lambda > 0$. Suppose that we start scheduling the client requests, and the last request of client i serviced so far is $r_{i_1}(j_1)$, $1 \leq j_1 < n_{i_1}$, i.e., the service of this client has not been completed. Assume, that next we service $r_{i_2}(j_2)$, where $r_{i_1}(j_1) = r_{i_2}(j_2)$. Indeed, servicing $r_{i_2}(j_2)$ does not incur a setup cost. yet, by interrupting the service of client i_1, we increase its completion time.

In two special cases, it is easy to determine whether an optimal algorithm should be greedy:

- When $\mu > k\lambda$, we note that since the increase caused in the total completion time from each "greedy step" is at most $k\lambda$, [1]we get that greediness is a desired property.

[1]This is due to the fact that the completion time of each client that has not completed yet is increased by λ; the maximal number of such clients is k.

- When $\mu \approx \lambda$ it is always preferable to service the clients sequentially, i.e., to complete servicing all the requests of client i before turning to service client $(i+1)$.

In this section we handle separately the case where $\mu = \lambda$ and the case where $\mu > \lambda$. We first introduce some notation that will be used in studying the online TCT problem. Given an algorithm $\mathcal{A}$, we denote by $x_i = x_i(\mathcal{A})$ is the time that it takes to service the requests of client i under the algorithm $\mathcal{A}$, i.e., $x_i = \sum_{j=1}^{n_i} t_s(r_i(j))$. For the case where $\mu = \lambda = 1$, $x_i = n_i$. $D^{\mathcal{A}}_{i,j}$ is the time allocated by algorithm $\mathcal{A}$ to client i while the service of client j has not been completed. The delay $P^{\mathcal{A}}_{i,j} = D^{\mathcal{A}}_{i,j} + D^{\mathcal{A}}_{j,i}$ is the amount by which clients i and j delay the completion time of each other.

5.1 The General Case

5.1.1 Lower Bound

Theorem 5.1 *If M and k can be arbitrarily large, the competitive ratio of any online deterministic algorithm for the TCT problem is $\Omega(\frac{\mu}{\lambda})$.*

Proof: Following the steps of the proof of Theorem 2.4 in [1], let $N = \lfloor \frac{M}{k} \rfloor$. We construct a clique of size Nk. The clique consists of N sub-cliques of size k each. The adversary generates requests for each sub-clique, using the "critically alive" game strategy described in [1]. For obtaining the lower bound, we use the next two claims adopted from [1].

Claim 5.1 *Any online deterministic algorithm, $\mathcal{A}$, incurs at least the cost $(N - 1 + \frac{k+1}{2})\mu k$ for servicing the requests generated by the adversary during the "critically alive" game.*

Claim 5.2 *The number of requests serviced by an optimal offline algorithm during the game for each client is at most $N - 1 + k^2$, and the maximal cost for servicing requests that incur setups is $\frac{k\mu(N-1+k^2)}{\lg k/2}$.*

¿From Claim 5.2 we get that for the generated instance, I,

$$OPT_{tct}(I) \le \frac{N-1+k^2}{\frac{1}{2}\lg k}\mu k + (N-1+k^2)\lambda k.$$

The bound is derived by taking the ratio

$$\frac{\mathcal{A}_{tct}(I)}{OPT_{tct}(I)} = \frac{(N-1+\frac{k+1}{2})\mu k}{\frac{N-1+k^2}{\frac{1}{2}\lg k}\mu k + (N-1+k^2)\lambda k}$$

$$= \frac{\frac{1}{2}(2N-1+k)\mu \lg k}{(N-1+k^2)\mu + \frac{1}{2}(N-1+k^2)\lambda \lg k}$$

For sufficiently large values of M and k this yields the statement of the theorem. $\square$

5.1.2 Upper Bound

We now show that the lower bound in Theorem 5.1 is tight to within a constant factor. Specially, we show that the competitive ratio of RR is $O(\frac{\mu}{\lambda})$. First, we lower bound $OPT_{tct}(I)$ for any instance I. The total completion time for OPT on any input instance I is

$$OPT_{tct}(I) = \sum_{i=1}^{k} x_i + \sum_{1 \le i < j \le k} P_{i,j}^{OPT}. \tag{5.1}$$

For convenience we renumber the clients such that $n_1 \le n_2 \cdots \le n_k$. To obtain the lower bound we minimize the two sums in the right-hand side of (5.1). Clearly, the left sum cannot be less than $\lambda \sum_{i=1}^{k} n_i$, and the right sum is always at least $\lambda \sum_{1 \le i < j \le k} \min(n_i, n_j)$.

$$\begin{aligned}
OPT_{tct}(I) &\ge \lambda \sum_{i=1}^{k} n_i + \lambda \sum_{1 \le i < j \le k} \min(n_i, n_j) && (5.2)\\
&= \lambda \sum_{i=1}^{k} n_i + \lambda \sum_{i=1}^{k} (k-i) n_i && (5.3)\\
&= \lambda \sum_{i=1}^{k} (k-i+1) n_i && (5.4)
\end{aligned}$$

Round-Robin (RR) is a well known scheduling algorithm in various operating systems. Generally, RR can be used to guarantee fairness while servicing the requests of different clients. RR also achieves a good competitive ratio with respect to the TCT measure.

Theorem 5.2 *For any k and $\mu \ge \lambda$, RR is within a constant factor from the optimal in the set of online deterministic algorithms for the TCT problem.*

Proof: The total completion time of RR for an input instance I is

$$RR_{tct}(I) = \sum_{i=1}^{k} x_i + \sum_{1 \le i < j \le k} P_{i,j}^{RR}$$

Note that RR guarantees that for any pair of clients, i and j, $P_{i,j}^{RR} \leq 2\mu \min(n_i, n_j)$. Hence,

$$RR_{tct}(I) \leq \mu \sum_{i=1}^{k} n_i + 2\mu \sum_{i=1}^{k} (k-i)n_i.$$

Now, let $c = \sum_{i=1}^{k} n_i$ and $y = \sum_{i=1}^{k}(k-i)n_i$. We define

$$f(y) = \frac{c+2y}{c+y}.$$

Note that for any $c \geq 1$ $f'(y) > 0$, thus $f(y)$ is maximized when y is maximal, that is, when $n_1 = n_2 \cdots = n_k$. Hence, for any I, let $c = c(I) = \sum_{i=1}^{k} n_i$

$$\begin{aligned}\frac{RR_{tct}(I)}{OPT_{tct}(I)} &\leq \frac{\mu}{\lambda} f(y) \\ &\leq \frac{\mu}{\lambda} \frac{c + 2\sum_{i=1}^{k}(k-i)\frac{c}{k}}{c + \sum_{i=1}^{k}(k-i)\frac{c}{k}} = \frac{\mu}{\lambda}\left(2 - \frac{2}{k+1}\right). \qquad (5.5)\end{aligned}$$

¿From Theorem 5.1, we have that any deterministic online algorithm has competitive ratio $\Omega\left(\frac{\mu}{\lambda}\right)$. This completes the proof. □

5.2 The Case where $\mu = \lambda$

When $\mu = \lambda$ the competitive ratio of any online algorithm is determined by the *number* of requests generated by the adversary for each client. W.l.o.g we assume that $\lambda = 1$. In this case, there is an optimal offline algorithm which services the clients sequentially. This follows from the fact that with no setup costs, our problem reduces to the problem of scheduling a set of k independent jobs: the chain of requests of client i, of length n_i, can be represented by job J_i, whose execution time is n_i; all the jobs are ready for execution at time 0. As mentioned in Section 3.2.3, the SPTF algorithm minimizes the total completion time. Hence, we get that for any instance I

$$OPT_{tct}(I) = \sum_{i=1}^{k}(k-i+1)n_i \qquad (5.6)$$

We now derive a lower bound on the competitive ratio of any online deterministic algorithm in this case.

Theorem 5.3 *Let $\mu = \lambda = 1$, then for any $k \geq 1$, the competitive ratio of any online deterministic algorithm for the TCT problem is at least $2 - \frac{2}{k+1}$.*

Proof: Given a deterministic algorithm $\mathcal{A}$, adversary initially generates requests until $\mathcal{A}$ has serviced $x = kp$ requests, for some $p \in N$. Let l_i be the number of request serviced last for client i, $1 \leq i \leq k$, then $n_i = l_i + 1$. Hence, we get that

$$\begin{aligned}\mathcal{A}_{tct}(I) &= xk + k + (k-1) + \cdots + 1 \\ &= xk + \frac{k(k+1)}{2}.\end{aligned}$$

From (5.6) and the fact that $\sum n_i = x + k$ we conclude that $OPT_{tct}(I)$ is maximized when

$$n_1 = n_2 = \cdots = n_k = \frac{x}{k} + 1.$$

Therefore,

$$\begin{aligned}\frac{\mathcal{A}_{tct}(I)}{OPT_{tct}(I)} &= \frac{xk + \frac{k(k+1)}{2}}{\sum_{i=1}^{k}(k-i+1)n_i} \\ &\leq \frac{xk + \frac{k(k+1)}{2}}{\left(\frac{x}{k}+1\right)\frac{k(k+1)}{2}} \\ &= \frac{2kx}{(x+k)(k+1)} + \frac{k}{x+k} \\ &\equiv f(x).\end{aligned}$$

Note, that $f'(x) = \frac{k^2-k}{(x+k)^2(k+1)} > 0$ for any $k \geq 1$, hence $f(x)$ is monotonically increases as x grows large. Taking $x \to \infty$ we get that

$$\frac{\mathcal{A}_{tct}(I)}{OPT_{tct}(I)} = \lim_{x\to\infty} \frac{2kx}{(x+k)(k+1)} + \frac{k}{x+k} = 2 - \frac{2}{k+1}.$$

□

Theorem 5.4 *The Round-Robin algorithm is exactly* $2 - \frac{2}{k+1}$ *-competitive.*

Proof: The proof follows from Theorem 5.2, when taking $\mu = \lambda = 1$. □

Chapter 6

The Online MRT Problem

6.1 Some Restricted Versions of the Problem

Before analyzing a general problem we examine two restricted versions:

- $\mu = \lambda$
- $M = 1$

For the case where $\mu = \lambda$, we note that for any instance, I, $OPT_{mrt}(I) \geq (k-1)\lambda$. For this case we show that RR is an optimal online algorithm.

Observation 6.1 *If $\lambda = \mu$ then RR is 1-competitive.*

Proof: For any instance, I, RR services at most $k-1$ requests of different clients between any two consecutive services of some client. There are no setups, therefore, $RR_{mrt}(I) \leq (k-1)\lambda$. □

If $M = 1$ either OPT or any deterministic algorithm $\mathcal{A}$ performs exactly one setup for any instance, I, when processing the first request. Thus, obviously, $OPT_{mrt}(I) = (k-2)\lambda + \mu$. We now show that RR is an optimal online algorithm in this case as well.

Observation 6.2 *If $M = 1$ then RR is 1-competitive.*

Proof: RR performs at most $k-1$ different client requests between any two consecutive requests of some specific client. The only setup that should be made is before processing the first request in the schedule, thus the maximal response time is at most $(k-2)\lambda + \mu$, and the claim follows. □

6.2 Analysis for the General Case

Theorem 6.1 *For sufficiently large k the competitive ratio of any deterministic online algorithm for the MRT problem is $\Omega\left(\frac{\mu}{\lambda}\right)$.*

Proof: Let $\mathcal{A}$ be an online deterministic algorithm. Consider the following instance, I. The first request of any client $1 \leq i \leq k$ is to bank 1, i.e., $r_i(1) = 1$. While there exists a client for which $\mathcal{A}$ has not serviced k requests, any request generated by adversary, for any client, is to bank 1. Note that $\mathcal{A}$ has to complete servicing the first k requests of all clients within finite number of steps, otherwise $\mathcal{A}$ is not competitive.

Suppose that $\mathcal{A}$ completes the service of k requests for all clients for the first time after servicing $r_{\hat{i}}(k)$, for some $1 \leq \hat{i} \leq k$. The adversary will generate the next request for client i, $1 \leq i \leq k$ to bank $(i+1)$.

Note that at this point, each client $i \neq \hat{i}$ has two requests: the first request is to bank 1 and the second request is to bank $(i+1)$ (client $\hat{i}$ has only one request, to bank $\hat{i}+1$).

Claim 6.1 $\mathcal{A}_{mrt}(I) \geq \frac{k}{2}\mu - \lambda$.

Proof: Suppose that $\mathcal{A}$ completes servicing $k/2$ clients. The response time of the remaining clients at this point is at least μ. We handle two cases separately:

- Among the $k/2$ remaining clients there exists a client with unserviced request to bank 1; Clearly, the MRT of this client is at least $\frac{k}{2}\mu$ (the lower bound on the service time of first $k/2$ clients).
- The requests for bank 1 of all the remaining clients have been serviced. In this case, one of the remaining clients will have to wait to be serviced for another $\frac{k}{2}\mu - \lambda$ time units.

This gives the statement of the claim. □

Now, consider the operation of an optimal offline algorithm, OPT, on the instance I. For convenience, we renumber the clients in non-decreasing order by n_i's, such that $n_1 \leq n_2 \cdots \leq n_k$. Note that $n_1 = k+1$, since there exists a client[1] whose total number of requests is $(k+1)$.

OPT services I in phases, as follows. In the first phase it services $r_i(1)$, for all $1 \leq i \leq k$; then it completes servicing the requests of client 1. Thus, we get an MRT of at most $(2k-3)\lambda + 2\mu$. In phase j, for $j \geq 2$, OPT uses RR to service the requests, until the remaining number of unserviced requests for client j is k; then, OPT completes servicing the requests of this client. Note that since

[1]The number of requests of the client initially numbered as $\hat{i}$ is exactly $(k+1)$

in each phase j OPT incurs at most two setups (when servicing the first request in phase j, and when servicing the last request of client j) we get the

$$OPT_{mrt}(I) \leq (2k-3)\lambda + 2\mu.$$

Thus, if k is sufficiently large, the competitive ratio of $\mathcal{A}$ is

$$\frac{\mathcal{A}_{mrt}(I)}{OPT_{mrt}(I)} \geq \frac{\frac{k}{2}\mu - \lambda}{(2k-3)\lambda + 2\mu} = \Omega\left(\frac{\mu}{\lambda}\right).$$

□

An example of an input instance in the proof of Theorem 6.1 is given in Figure 6.1.

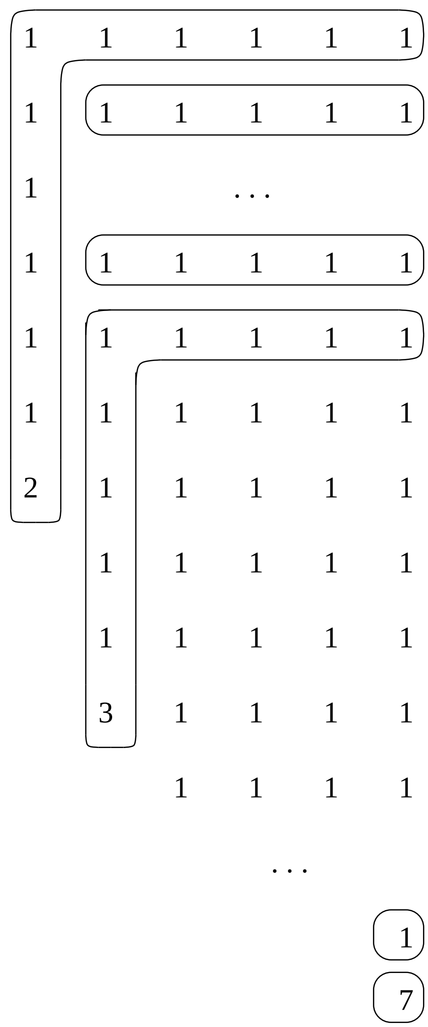


Figure 6.1: Example of an input instance in the proof of Theorem 6.1 ($k = 6$)

Theorem 6.2 *The Round-Robin algorithm has a competitive ratio $O(\frac{\mu}{\lambda})$ for the MRT problem.*

Proof: MRT of RR is at most $k\mu - \lambda$, while the MRT of any optimal algorithm is at least $(k-1)\lambda$. This completes the proof. $\square$

Chapter 7

Experimental Results

In this section we examine the performance of algorithms for the TCT and MRT over several sets of input sequences[1]. For simulating a real system environment we modeled the generation of access requests to the banks $1, \ldots, M$ as a Markov process, that is, the probability that the j-th request of client i is to some bank $1 \le s \le M$ depends only on the $(j-1)$-th request of client i. Formally, for each client i, $1 \le i \le k$, we generated an $M \times M$ matrix, B_i; each entry in B_i, $B_i[s,t]$, is the probability that the next request of client i is to bank t, given that the last request was to bank s, and $\sum_{t=1}^{M} B_i[s,t] = 1$ for all $1 \le s \le M$.

Each row of B_i was generated by taking random permutation of a set of probabilities computed from a geometric series, with some parameter $0 < q < 1$. Specifically, given the parameter q, our basic set of probabilities was $\{p_1, \ldots, p_M\}$, where

$$p_i = \frac{q^i}{\sum_{i=1}^{M} q^i}, \quad 1 \le i \le M.$$

Now, let $\pi_s = (\pi_{s,1}, \ldots, \pi_{s,M})$ be a random permutation of the indices $\{1, \ldots, M\}$ selected for row s, then $B_i[s,t] = p_{\pi_{s,t}}, \quad 1 \le t \le M$. We call q the *locality parameter* of the system, since it indicates to which subset of banks the i-th client tends to generate requests after accessing bank j. Indeed, as q approaches 1, we get that the distribution on the accesses to the banks becomes close to the Uniform distribution, while smaller values indicate that after accessing some bank, s, client i will access almost surely the bank t, for some $1 \le s,t \le M$. Thus, the size of the subset of banks that can be accessed by client i in each request becomes large as q gets larger. Unless specified otherwise we used in our experimental study the value $q = 0.7$. Each result was obtained as an average over ten tests.

[1] Experimental results for the makespan measure can be found in [5].

7.1 Results for the TCT Problem

The optimal algorithm for TCT was implemented using the dynamic programming algorithm described in the proof of Theorem 3.2.

The following algorithms were implemented in our experiments for the TCT problem

- Round-Robin (RR)
- Greedy Round-Robin (GRR)
- Greedy Enhanced Round-Robin (GERR) - This algorithm was proposed in [5] as an online algorithm for plan merging. In each phase $j, 1 \leq j \leq n_{max}$ GERR groups the requests from the current frontier according to the memory banks they refer to. Then, GERR chooses a group with the maximal number of unserviced requests and handles the requests in this group sequentially. For each client, $1 \leq i \leq k$, when GERR completes servicing the current request of client i, it inspects the next request of client i; if this new request is to a bank s, that was not accessed in phase j, this request is added to the set of requests for bank s (that will be serviced later on in this phase). Thus, GERR attempts to group the maximal number of available requests to each of the banks. Clearly, in the worst case GERR performs the same as GRR, but we expect GERR to perform better on the average.
- SEQ - Services the clients in sequential order by their numbers.
- Greedy SEQ (GSEQ) - A variant of SEQ which services the clients sequentially, except that after servicing a request of client i to access a bank s, $1 \leq s \leq M$, GSEQ services all the available requests for bank s; then, GSEQ turns to service the next request of client i.
- Greedy Cautious SEQ (GCSEQ) - Unlike the GSEQ algorithm, GCSEQ does not always service all the requests in the current frontier to the current memory bank. It first calculates the *range of greediness* - R, i.e., given an order of sequential execution of the client request chains, if the number of clients whose requests are still unserviced is $\hat{k}$, the algorithm defines the value $|R| = \hat{k} - \lceil \frac{\hat{k}\lambda}{\mu} \rceil$. The computed size defines the range of the clients indices, starting from the index of the current client, in which the algorithm is greedy, i.e., does not skip requests to the same memory bank. The idea behind the algorithm is simple: greediness does not always lead to a better TCT.
- Shortest Client First (SCF)

We now turn to describe our results, as given in Figures 7.1-7.5. We used in the experiments the following set of base parameters: $\mu = 10$ and $\lambda = 1$; the number of banks was set to $M = 5$; the number of clients was $k = 50$, and the maximal number of requests was $n_{max} = 150$.

Figure 7.1 shows the TCT of the various algorithms vs. n_{max}. Due to the computational complexity involved in the implementation of the optimal algorithm, OPT (see in Theorem 3.5) we used in this experiment a small number of clients, i.e., $k = 5$. We note that GSEQ and GCSEQ performed very similarly: this is due to the values of the input parameters, which are relatively small here. The graphs for the two algorithms show that for any value of n_{max}, TCT for GCSEQ and GERR is approximately 1.27 times the optimal. We also note that for any value of n_{max} TCT for RR is no more than 2.15 times the optimal. This result shows that RR performs much better on the average than its upper bound of $\left(2 - \frac{2}{k+1}\right) \frac{\mu}{\lambda}$ computed in (5.5). Note that RR achieved the worst performance among the tested algorithms. Like RR , SCF performs better than its upper bound, $\frac{k+1}{2}$, stated in Theorem 3.7, and is no more than 1.32 times the optimal for any value of n_{max}. GRR, whose approximation ratio $\min(M + \frac{(k-M)\lambda}{\mu}, k)$ was shown in Theorem 3.6, had a TCT that is no more than 1.66 times the optimal, for any value of n_{max}.

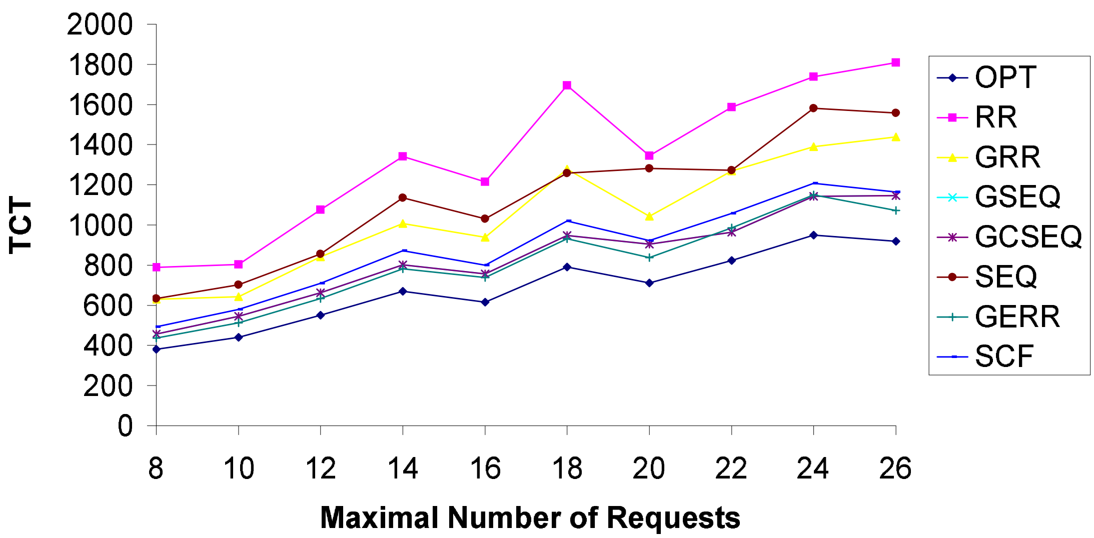


Figure 7.1: Total completion time vs. n_{max} ($k = 5$).

Figure 7.2 presents the results of the same experiment, only that OPT was not implemented. Thus, we could use a large number of clients. Here, we observe that GERR and GCSEQ perform better than the other algorithms. We also note the linear dependence of TCT of the various algorithms on n_{max}.

In Figure 7.3 we examine the TCT of the algorithms for different values of M. (The number of clients was reduced to $k = 5$, and $n_{max} = 25$, due to the

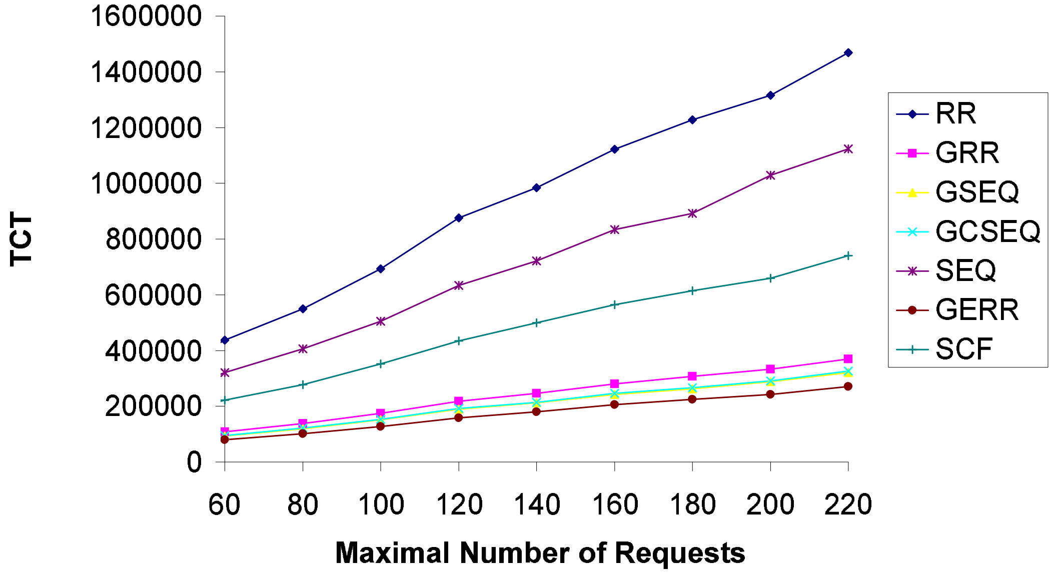


Figure 7.2: Total completion time vs. n_{max}.

implementation of OPT). We note that when $M > k$, SCF performs better than the greedy algorithms. Among the online algorithms GERR, GSEQ and GCSEQ achieved the best performance. RR, SCF and GRR obtain the same maximum performance ratio compared to the optimal algorithm as in Figure 7.1. Again, we get that the performance of these algorithms on the average is much better than the upper bounds.

Figure 7.4 shows the dependence of TCT on the locality parameter of the system. We note that TCT only slightly depends on q: the relative performance of the algorithms was not affected when the value of q was changed. This holds also for larger number of clients, and larger value of n_{max}, as shown in Figure 7.5. Finally, in Figure 7.6 we examine the performance of the algorithms for different numbers of clients. The graphs show that the TCT of the algorithms is approximately linear in k^2. As in the above experiments, we got that GERR, GCSEQ and GSEQ achieved the best performance.

In our experimental study of the TCT problem we found that GERR achieves the best performance among the tested algorithms. It should be mentioned, though, that "greediness" is more advantageous when the number of memory banks is small, relative to the number of clients. When the number of memory banks becomes larger, other algorithms that schedule short clients first (e.g. SCF) perform better.

In Figures 7.1, 7.3, 7.4 and 7.5 we find some unexpected interruptions in the monotonicity of the curves. For the results shown in Figures 7.1, 7.3 and 7.4 a main factors in this "anomaly" may be the small values of k and n_{max} that increased the effect of randomness in choosing n_i and $r_{i,j}$, $1 \leq i \leq k$, $1 \leq j \leq$

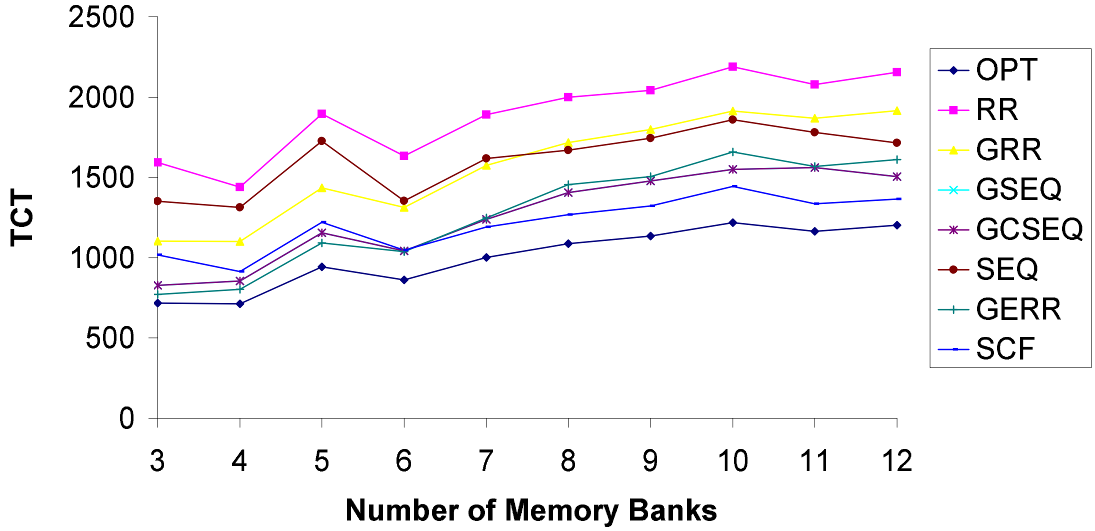


Figure 7.3: Total completion time vs. M ($k = 5$, $n_{max} = 25$).

n_i on the results. We note that when k and n_{max} are large, this "anomaly" disappears. An exception to this appears in the curves shown in Figure 7.5. A possible explanation to this may be the insufficient number of tests for this specific case, where the value of the locality parameter has little effect on the behavior of the algorithms.

7.2 Results for the MRT Problem

For the MRT measure we implemented the following algorithms:

- Round-Robin (RR)
- Greedy Round-Robin (GRR)
- Random - In each phase $j, 1 \leq j \leq n_{max}$ Random groups the j-th requests of all the clients. Then, it randomly chooses an unserviced request in this set and handles it. The j-th phase terminates when Random completes handling the j-th requests of all clients.

Due to the computational complexity involved in the implementation of the optimal algorithm (see in Theorem 3.8) we have not implemented OPT in this set of experiments (in which both k and n_{max} were large)

In Figure 7.7 we present the MRT of the algorithms vs. the ratio μ/λ. We observe that when μ/λ is small RR performs better than GRR, but GRR is better when $\mu/\lambda > 2$. The Random algorithm performs poorly for any ratio of μ/λ.

Figure 7.8 shows MRT vs. the maximal number of requests. We get that MRT of the algorithms is not affected by the value of n_{max}. Finally, Figure 7.9

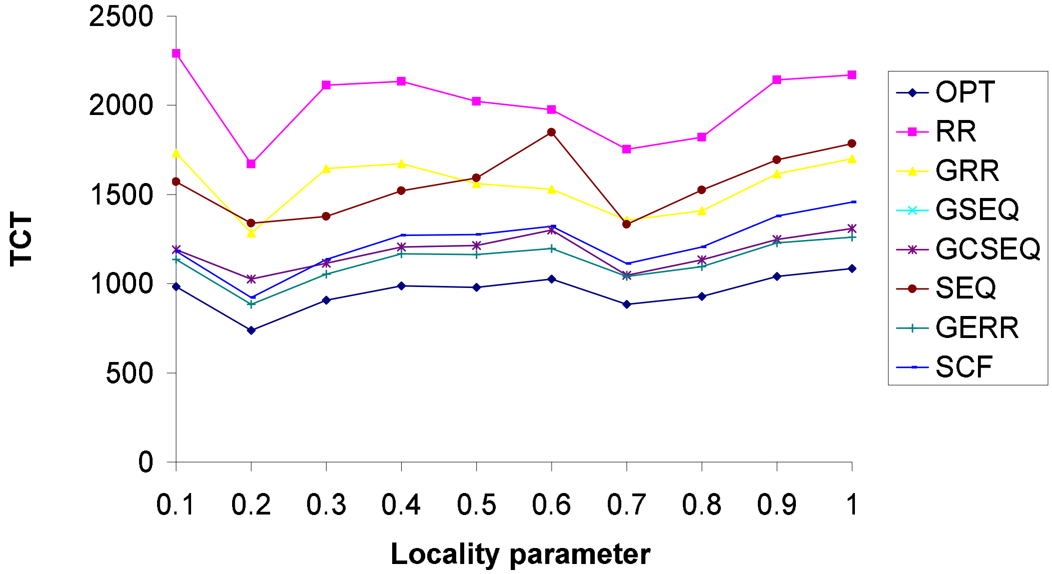


Figure 7.4: Total completion time vs. locality parameter($k = 5,\ n_{max} = 25$).

shows that the advantage of GRR over RR and Random becomes clearer when the number of clients becomes larger, relative to number of memory banks.

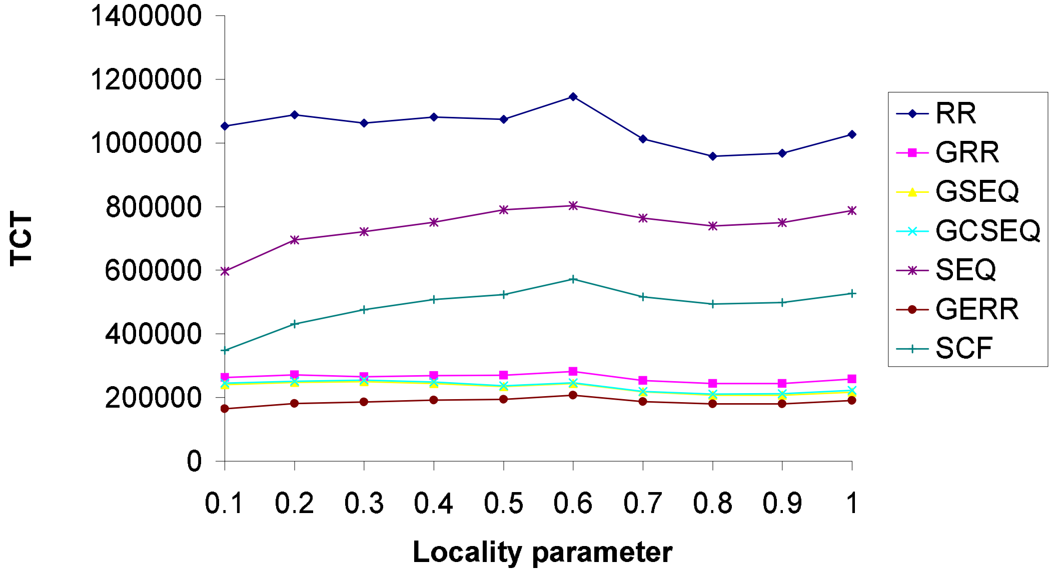


Figure 7.5: Total completion time vs. locality parameter.

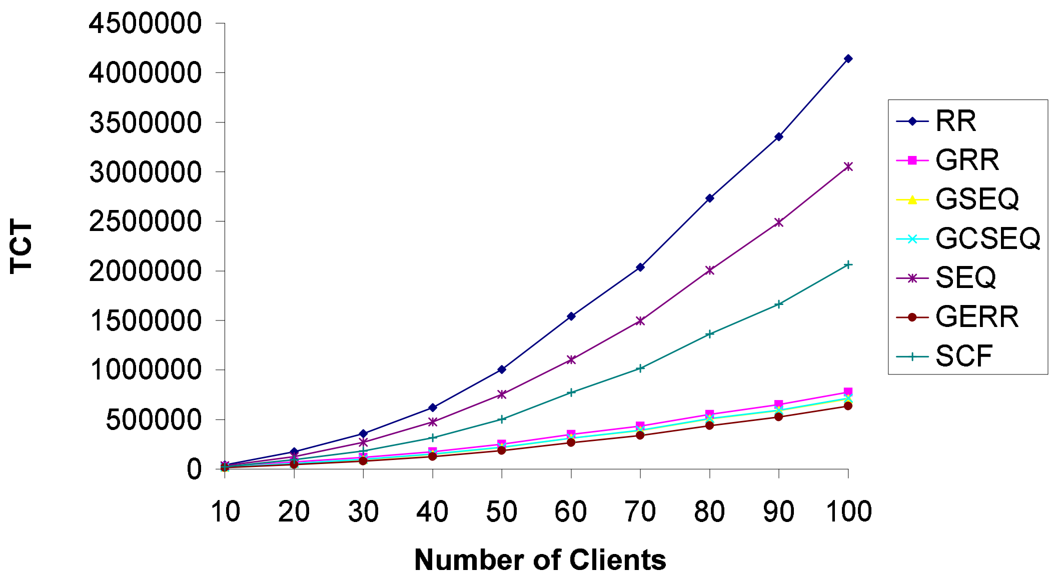


Figure 7.6: Total completion time vs. k.

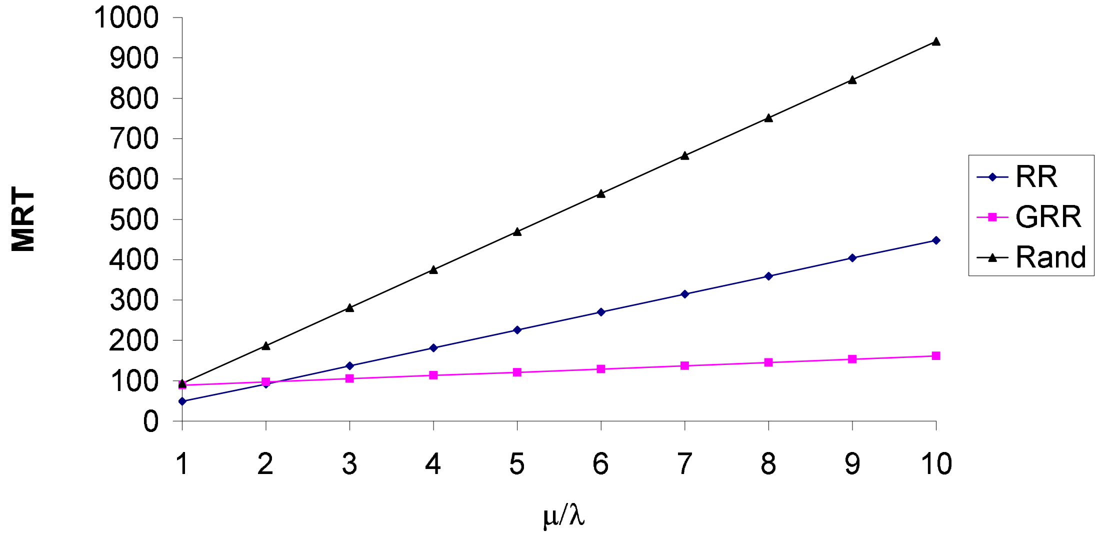


Figure 7.7: Maximal Response Time vs. the ratio $\frac{\mu}{\lambda}$.

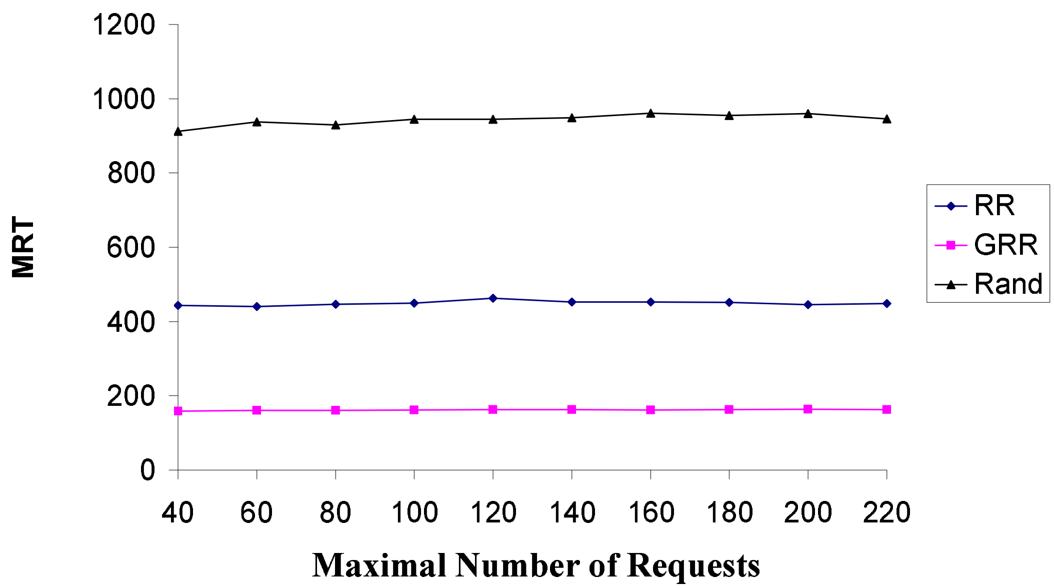


Figure 7.8: Maximal Response Time vs. n_{max}.

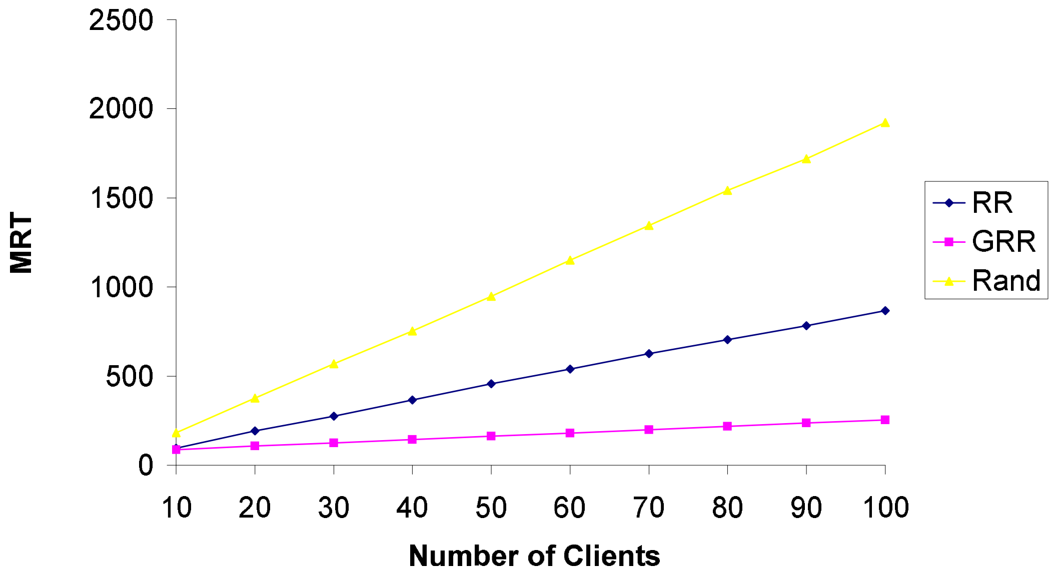


Figure 7.9: Maximal Response Time vs. k.

Chapter 8

Summary

8.1 The Contribution of this Work

We studied the problem of scheduling multiple clients by a bus arbiter. We examined the offline and the online problem using three performance measures: makespan, TCT and MRT.

- For the offline problem
  - We showed that the TCT problem is NP-complete.
  - Dynamic programming algorithms were presented for all performance measures.
  - We presented approximation algorithms for the makespan and the TCT problems.
- For the online problem
  - We derived a lower bound of $\Omega(\frac{\mu}{\lambda})$ on the competitive ratio of online deterministic algorithms for all three measures.
  - We described online algorithms whose competitive ratios are $O(\frac{\mu}{\lambda})$.

8.2 Conclusions from the Experimental Study

- Greedy algorithms (e.g. GERR and GSEQ) often achieve better results for both the TCT and the MRT performance measures, compared to non-greedy algorithms.
- For the TCT measure the performance of RR, GRR and the SCF algorithms on the average is much better than their worst case upper bounds.
- As it can be seen in Figure 7.3, the performance of greedy algorithms for TCT is getting worse, when the number of memory banks becomes larger.

- Our experimental study shows that GERR outperforms any other algorithm w.r.t the TCT measure; GRR achieved the best performance for the MRT measure.
- In generating request sequences for the clients (excluding the usage of locality parameter in some of the experiments) we have not used access patterns, that is, sub-sequences of accesses which may appear repeatedly in the sequences of specific clients. For inputs that contain such patterns, our algorithms can be tuned to initially learn these patterns, and later employ this knowledge in servicing the clients. This would yield improved performance.

8.3 Open Problems

Several interesting problems remain open:

- We showed that the TCT problem is NP-complete, but the question: "is the TCT problem MAX-SNP hard (as the makespan problem)?" is still open.
- Hardness of the offline MRT problem.
- We studied the problem in a single bus architecture. The multi-bus model appears to be a natural extension of our study.
- Better approximation algorithms for the offline makespan and TCT problems (reducing the gap between the upper and lower bounds for the makespan problem).

Bibliography

[1] H. Alborzi, E. Torng, P. Uthaisombut and S. Wagner, *The k-client Problem*, in Proceedings of the Eighth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), New Orleans, 1997, pp. 73–82.

[2] P.A. Bernstein, "Sequoia: A Fault-Tolerant Tightly Coupled Multiprocessor for Transaction Processing", *Computer*, 21:2, 1988, pp. 37–45.

[3] R. Bhatia, S. Khuller, J. Naor, *The Loading Time Scheduling problem*, 36th IEEE Conference on Foundations of Computer Science, Milwaukee, Wisconsin, 1995, pp. 72–81.

[4] A. Borodin and R. El-Yaniv, *Online Computation and Competitive Analysis*, Cambridge University Press, 1998.

[5] D. E. Foulser, M. Li and Q. Yang, *A Theory of Plan Merging*, Artificial Intelligence, 57, 1992, pp. 143–181.

[6] C. B. Fraser and R. W. Irving , *Approximation algorithms for the shortest common supersequence*, Nordic J. Comp. 2, 1995, pp. 303–325.

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

[8] S.Y. Itoga, *The String Merging Problem*, BIT, 21, 1981, pp.20–30.

[9] T. Jiang and M. Li, *On the Approximation of Shortest Common Supersequences and Longest Common Subsequences*, SIAM Journal on Computing, 24(5), October 1995, pp. 1122–1139.

[10] E. Lawler, J. Lenstra, A. Rinnooy Kan and D. Shmoys, *Sequencing and Scheduling: Algorithms and Complexity*, Handbooks in Operations Research and Management Science, Vol. 4: *Logistics of Production and Inventory*, 1990.

[11] D. Maier, *The Complexity of Some Problems on Subsequences and Supersequences*, Jornal of the Association for Computing Machinery, Vol. 25, No. 2, April 1978, pp. 322–336.

[12] M. Milenkovic, *Operating Systems Concepts and Design*, McGraw-HILL, INC., 1992.

[13] R. Motwani, S. Phillips and E. Torng, *Non-Clairvoyant Scheduling*, Fourth ACM-SIAM Symp. on Discrete Algorithms, 1993, pp. 422–431.

[14] S. Thakkar, P. Gifford and G. Fielland, *The Balance Multiprocessor System*, IEEE Micro, 8:1, 1988, pp. 57–69.