i.e., at least $d + d - 1 = 2d - 1$ colors are required to color $G(d)$. By Lemma 4.2 we know that **HDF**, being a greedy algorithm, requires at most $2d - 1$ colors to color $G(d)$. It follows that there exists a sequence of choices for which **HDF** uses $2d - 1$ colors to color $G(d)$. □

To summarize, by Lemma 4.2 we have shown that any greedy heuristic, and hence **HDF**, uses at most $2d - 1$ colors to color a bipartite graph, and by Theorem 4.1 we have shown that for any d we can construct an example for which **HDF** actually uses $2d-1$ colors. We can use Lemma 4.2 to also obtain the time complexity of **HDF**.

Theorem 4.2 **HDF** *takes time $O((m+n)d)$ to solve $Unit - SimpleDTS$, and produces a schedule of length at most $2 - \frac{1}{d}$ times the optimal length.*

proof. For the time complexity, note that *Sort-by-degree()* takes time $O(n+d)$, if a bucket sort is used. For each color, each edge is examined at most once. Thus each color (i.e., iteration of the while loop) takes time $O(m + n + d)$ [132]. Using Lemma 4.2, the total time is thus $O((m + n + d)(2d - 1)) = O((m + n)d)$. The performance guarantee follows from Lemma 4.2 also. □

4.1.2 The $Unit - DTS$ Problem

The program for implementing the **HDF** heuristic for handling the situation where a color may be used to color at most $k \leq n$ edges is a slight modification of the program given above. We add the following line,

```
11.5    if $|E'| = k$, break
```

That is, after every edge is added to E', the program checks if $|E'| = k$, and if so, does not add any more edges to E' for the current color. We call this program **MHDF** for convenience.

The analysis of **MHDF** is slightly more complicated than for **HDF**. In particular, we are not able to prove a bound that is tight for all inputs.

Lemma 4.4 **MHDF** *produces a coloring using at most* $\lfloor m/k \rfloor + (2d-1)$ *colors for a graph of* n *vertices,* m *edges, and degree* d*, if at most* $k \leq n$ *edges may be colored with a single color.*

proof. Suppose **HDF** was used on the graph instead of **MHDF**. In the worst case it uses $2d-1$ colors, with color i coloring m_i edges, and $\sum_{i=1}^{2d-1} m_i = m$. If **MHDF** uses the same sequence of choices, coloring m_i edges may require up to $\lfloor m_i/k \rfloor + 1$ colors. The result follows. □

Note that this bound is exact for the graph $m = n = 5$, $d = 1$, $k = 2$. On the other hand, it is not tight for the class of graphs with $k = n$ and $k = 1$. In the former case, **MHDF** is simply **HDF**, so at most $2d-1$ colors are needed. In the latter case, exactly $m/k = m$ colors are needed.

4.2 The Highest Combined Degree (HCDF) Heuristic

An obvious modification to the **HDF** heuristic is to give preference not necessarily to vertices of highest degree, but to edges whose end vertices have the highest combined degree. Somalwar [132] experimentally investigated this Highest-Combined-Degree-First heuristic, **HCDF**, and found that it gave optimal or near-optimal solutions for input graphs generated randomly. In this section we find the time complexity and performance guarantee of **HCDF**, for both the $Unit - SimpleDTS$ and $SimpleDTS$ problems. This turns out to be a simple extension to the analysis for **HDF**.

Our first observation is that **HCDF** is also a special case of the greedy heuristic. We thus obtain the following result.

Theorem 4.3 **HCDF** *takes time* $O((m+d)d)$ *to solve* $Unit - SimpleDTS$*, and produces a schedule of length at most* $2 - \frac{1}{d}$ *times the optimal length.*

proof. The performance guarantee follows from Lemma 4.2 since **HCDF** is also a greedy heuristic. For the time complexity, we observe that the edges can be sorted using a bucket sort, with buckets in the range 1 to $2d$, and for every color, each edge is examined at most once. Therefore every color takes time $O(m + d)$, and there are at most $2d - 1$ colors used. □

We can also show that for certain graphs exactly $2d - 1$ colors will be used by modifying the construction used for **HDF**.

Theorem 4.4 *For any positive integer d, there exists a bipartite graph of degree d and a sequence of choices made by* **HCDF** *such that it uses $2d - 1$ colors to color the graph.*

proof. Construct $G(d)$ as described for Theorem 4.2. We will construct a new tree $T(d)$ by modifying $G(d)$; initially set $T(d) = G(d)$. We call an edge *critical* in $G(d)$ or $T(d)$ if the combined degree of its end vertices is $2d$. For all $i \in \{1, ..., d\}$, edge $rc(i, d)$ in $G(d)$ has at least one critical vertex; for each such edge, ensure the corresponding edge in $T(d)$ has both end vertices critical by adding appropriate edges and vertices if necessary. Now the sequence of edge-colorings used by **HDF** to color $rc(i, d)$, $i \in \{1, ..., d\}$, can be used by **HCDF**. Thus it takes $d - 1$ colors before the root of $T(d)$ is colored, and an additional d colors to color all the edges incident to the root. Since **HCDF** is greedy, by Lemma 4.2 all other edges can be colored using $2d - 1$ colors. □

Finally, we call the algorithm **HCDF** modified to color at most $k \leq n$ edges with the same color the Modified-HCDF algorithm, **MHCDF**. For this algorithm applied to the $Unit - DTS$ problem, we can obtain a result similar to **HDF**.

Lemma 4.5 **MHCDF** *produces a coloring using at most $\lfloor m/k \rfloor + (2d - 1)$ colors for a graph of n vertices, m edges, and degree d, if at most $k \leq n$ edges may be colored with a single color.*

proof. Similar to Lemma 4.4. □

4.3 Comparison of experimental and theoretical results

Somalwar [132] has experimentally evaluated the performance of **HDF** and **HCDF** on instances of the $Unit-DTS$ problem He compared their behavior to that of the exact algorithm **A**, which is a special case of **A2** for unit-weight edges implemented by Somalwar [132]. We also compare their behavior to the performance guarantees derived in this chapter.

Somalwar performed this experimentation in the context of the parallel I/O application, i.e., scheduling parallel I/O operations for a shared-bus multiprocessor system. For this context, certain parameters of the input graph $G = (A, B, E)$ were fixed. In particular, it was chosen that $n_1 = |A| = 16$, $n_2 = |B| = 64$, $m = |E|$ varied from 100 to 1000, and k varied from 4 to 16. Edges were generated using a pseudo-random generator.

Somalwar [132] found that in his experiments both **HDF** and **HCDF** produced schedules that are almost always of the optimal length. In that sense, they perform much better on average, for this set of experiments, than their performance guarantees predict for their worst case behavior. Although these results seem surprisingly good, there are some recent related experimental results to support them. Moret [105] found that for maximum cardinality bipartite matching, using a simple greedy heuristic similar to **HDF** delivered an optimal solution at least 99% of the time. Since the basic operation of edge-coloring unit-weight graphs can be regarded as repeated matching, these results are consistent with Somalwar's observations.

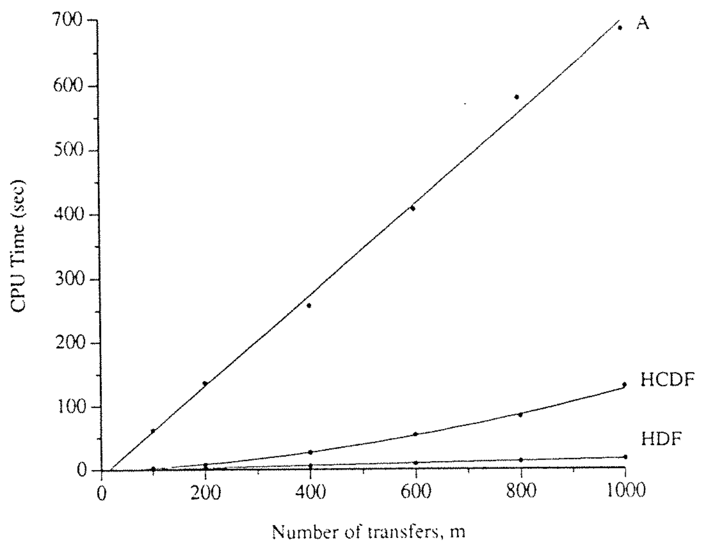

Figure 4.2: CPU time versus number of transfers for k = *4*

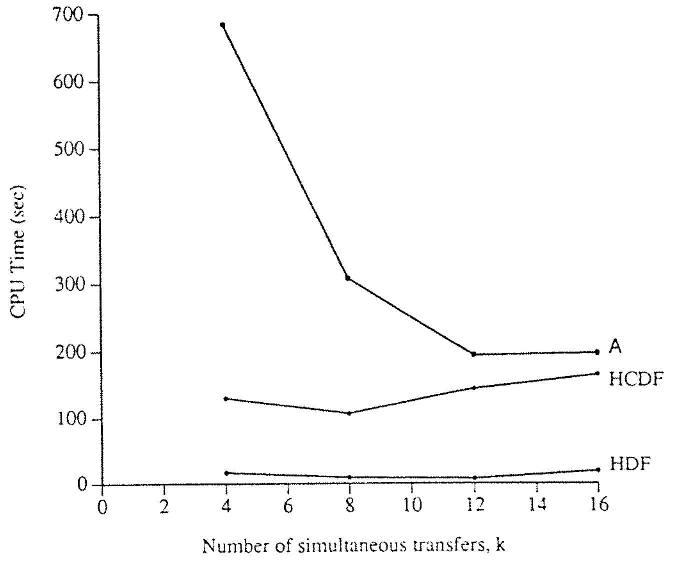

Figure 4.3: CPU time versus number of simultaneous transfers possible for m = *1000*

In Fig. 4.2 and Fig. 4.3 the execution time of **A**, **HDF** and **HCDF** is compared, as m and k are varied respectively. It can be seen that **HDF** executes in at most 10% of the time required for **A2** for this set of experiments. On the other hand, **HCDF** may take up as much as 80% of the time required by **A2**. Since both heuristics almost always produce optimal schedules, this set of experiments indicates that **HDF** is to be preferred over **HCDF** and **A2**, unless it is essential to produce optimal schedules, in which case **A2** should be used.

4.4 Discussion

4.4.1 Previous related work

The literature on exact edge-coloring of bipartite graphs with unit-weight edges was discussed in the previous chapter. To our knowledge, there has been no previous work on analysis of approximate edge-coloring of bipartite graphs with unit-weight edges.

The only related work that may be relevant deals with approximate edge-coloring of general graphs and multigraphs. Holyer [70] showed that determining whether a general graph can be colored in d or $d+1$ colors (the *classification problem* [41]) is NP-complete. A consequence of his result was that, unless P = NP, there does not exist an approximation algorithm for edge coloring a general multigraph using at most $(4/3-\epsilon)D$ colors, for any $\epsilon > 0$, where $D \in \{d, d+1\}$ is the minimum number of colors required. This would seem to imply that for general multigraphs finding a provably good approximation algorithm is difficult. However, an algorithm using no more than $(4/3)D$ colors, and running in time $O(m(n + D))$ was developed [69]. This algorithm uses an "interchange approach" as its basis: for each edge, check if some "simple" recoloring of the colored edges would eliminate the need for an additional color. As the authors say, "in order to prove better bounds, the 'simple' recolorings become more complicated" [69].

We note three deficiencies with this interchange heuristic. Firstly, it does not consider the practical constraint of a limited number of simultaneous data transfers, i.e., $k \leq n$. For our applications, this constraint corresponds to the realistic situation of limited bus bandwidth or switching capacity being available in the system. Secondly, although this heuristic has a better performance guarantee than those analyzed in this chapter, its time complexity is slightly worse, unless $d = n$. Thirdly, since the **HDF** and **HCDF** heuristics do not perform any backtracking or recoloring, they are likely to have smaller constants for their time complexity, and are likely to be simpler to implement, than the interchange heuristic.

To our knowledge, there has been no previous published experimental evaluation of approximation algorithms for edge-coloring bipartite graphs other than Somalwar [132]. The only unpublished results we are aware of are those by Moret [105], which provide evidence to support some of Somalwar's results.

4.4.2 Conclusions and further work

Approximate algorithms for edge coloring bipartite graphs are very attractive from a practical standpoint, particularly if they are to be used as the basis of scheduling algorithms that are executed very frequently in a data transfer application. While Somalwar [132] has suggested some simple greedy heuristics that seem to perform extremely well when evaluated experimentally, there was no analysis of the time complexity or the performance guarantee for these heuristics.

Our contribution in this chapter has been to prove the time complexity and performance guarantees for the heuristics proposed by Somalwar. The heuristics apply to the data transfer problem DTS when restricted to unit-length transfers. We have shown that the heuristics generate schedules less than twice the length of the optimal schedule, in the worst case, and take at most time $O((m+n)d)$ to execute, where n is the number of vertices, m the number of edges and d the degree of the input graph.

Our result enables us to compare the performance of Somalwar's heuristics with more sophisticated "interchange" heuristics [69]. The interchange heuristics are more general than those of Somalwar as they are applicable to general graphs as well as multigraphs. They also provide a better performance guarantee for only a slight increase in time complexity. However, we observe that for our applications, the Somalwar heuristics may be preferable as the applications are restricted to bipartite graphs, and the heuristics allow the case when a limit is placed on the number of simultaneous data transfers allowed.

We also surmise that the interchange heuristics have larger time constants, worse average-case execution time, and are harder to implement, than the simple greedy heuristics of Somalwar.

For future work, we suggest two questions. The first question is to experimentally and theoretically compare the interchange heuristics and Somalwar's greedy heuristic, and determine the range of parameters for which each is suitable. The comparison could include theoretical average-case analysis (e.g. [46]) as well as careful experimental evaluation. The second question is to consider the possibility of using parallel algorithms for scheduling; these may be especially suitable for the parallel I/O application. The results of Karloff and Shmoys [86] provide a starting point in this direction.

Chapter 5

Scheduling in Hierarchical Architectures

We have so far discussed scheduling of data transfers in system architectures which have a rather simple, although common, structure. The architecture of the *DTS* and the *SimpleDTS* problems assume that there is a direct dedicated link between every sender and every receiver; constraints arise in the number of links that may be used simultaneously, and in the capacity of each sender and receiver to engage in at most one transfer at any given time. In the context of the parallel I/O application, this architecture corresponds to the commercially succesful and popular class of shared-bus multiprocessors, in which processors and disks are connected by a set of parallel buses. For the communications application, it corresponds to the case of scheduling transfers through a single TDM switch, which can be viewed as a single multiplexer feeding a single demultiplexer.

In this chapter we consider more complex architectures that do not assume a direct dedicated link between every sender and every receiver. In particular, we consider a system where the data transfers must pass through a hierarchically arranged set of communication paths, which form a communication tree (see Fig. 5.1). We also generalize the architecture to allow arbitrary capacities, drawn from the set of positive integers, for each link in the architecture. Note that this also does away with the restriction that a sender or receiver can engage in at most one transfer in any given time. We call this architecture the tree architecture.

In section 5.1 we define the tree architecture formally in our model and specify the data transfer scheduling problem that we are interested in. In section 5.2 we show that this problem has application in three different areas: parallel I/O, switching systems, and file transfer in computer networks. The tree architecture is also of theoretical interest: we surmise that if the architecture is made more complex than a tree, optimal preemptive scheduling of integer-length transfers without precedence constraints cannot be done in polynomial time.

We then develop, in section 5.3, the outline of an algorithm to optimally solve the scheduling problem, and in section 5.4 we show how the algorithm can be designed to obtain a solution in time $O(Cn^4)$, where n is the number of senders and receivers, and C is the average number of transfers a sender or receiver can engage in at any one time. This algorithm, which we call the **Tree** scheduling algorithm, has been presented in [77, 125]. In general, **Tree** is a generalization and improvement in time complexity over previous algorithms [24, 11, 96, 20, 136] for this class of problems.

We have implemented the **Tree** algorithm. In section 5.5 we report the results of an experimental evaluation of the behavior of the algorithm for random input instances. Finally, in section 5.6 we discuss previous related work, and end with some conclusions.

5.1 Definition of the problem

We define the generalized data transfer scheduling problem *TreeDTS*, which differs from *DTS* in allowing tree-structured interconnection networks of the type shown in Fig. 5.1. The interconnection network is defined formally as *AG* below.

$TreeDTS = (PG, AG, RG, f, Preempt)$

where $Preempt = true$, f is makespan, and, PG, AG, and RG are defined as follows.

$PG = (T, Ep, Lp)$ with $|T| = m$, $Ep = \{\}$, and $Lp(t) \geq 0$ for all $t \in T$ are the task lengths.

$AG = (R, Ea, La)$ and (see Fig. 5.1) $RT = \{SUSER, RUSER, MUX, DMUX, NULL\}$; $|R| = n + n' + 1$, n is the number of resources of type $SUSER$ or $RUSER$, n' is the number of type MUX and $DMUX$, and there is one resource of type $NULL$. Ea is a directed tree whose root is the resource of type $NULL$. The root has a left subtree MT called the multiplexer subtree with interior nodes of type MUX, leaves of type $SUSER$, and arcs directed towards the root. (The definition of the right subtree DT follows by analogy).

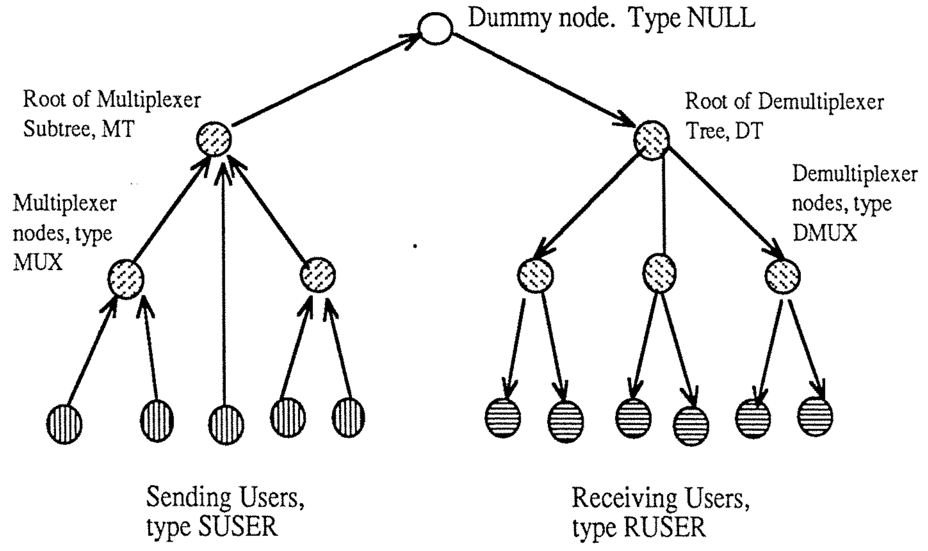
Dummy node. Type NULL
Root of Multiplexer Subtree, MT
Root of Demultiplexer Tree, DT
Multiplexer nodes, type MUX
Demultiplexer nodes, type DMUX
Sending Users, type SUSER
Receiving Users, type RUSER

Figure 5.1: Model of a tree-structured architecture

Now $La(r) = 0$ if $r \in R$, and $La(e)$ is the capacity of arcs e (in packets per second) for $e \in Ea$. We assume all interior vertices have degree at least 3 so as to avoid degenerate trees. We also assume that for vertices in MT the sum of $La(e)$ for incoming arcs e is at least the value of $La(e')$ for the single outgoing arc e'. There is an analogous assumption for DT, while for the root the capacity of the incoming arc equals that of the outgoing arc.

$RG = (R, Er, Lr)$ is a bipartite graph, where Er is a set of arcs from vertices of type $SUSER$ (senders) to those of type $RUSER$ (receivers) representing the data transfer operations to be scheduled, and Lr is a bijection from Er to T. Note that since there is a unique path betwen each sender-receiver pair, only the assignment of tasks to senders and receivers is shown in RG, the assignment of other resource types being left implicit.

5.2 Three Practical Applications

By casting the extended data transfer scheduling problem in our model, we see that it is a generalization of problems studied for three applications: parallel I/O, satellite switching, and network file transfers [80, 81]. Thus the results we derive in this chapter are available to all three applications. We describe these applications below.

Application 1: Parallel I/O in multiprocessor systems

In this chapter we consider cases of the I/O scheduling problem in which we do not assume a direct dedicated link from every processor to every memory. In addition, a processor or memory is not limited to engaging in at most one transfer at any given time. This problem is applicable to I/O scheduling in a variety of tree-structured parallel computer architectures, as well as interconnection networks such as KYKLOS [102].

Parallel database machines have been built that have tree-structured architectures, such as the VLSI tree machine of Song [133] and the relational database machine REPT [127]. The VLSI tree machine consists of two mirrored binary trees connecting a common set of leaves. The root of the top (called "circle") tree receives data and commands from the external host and broadcasts them down to the leaves ("square nodes"). The leaves perform the data manipulations in parallel and deliver results via the bottom ("triangle") tree. The interior nodes of the bottom tree combine results from the leaves before transferring them to the external host via the root of the bottom tree.

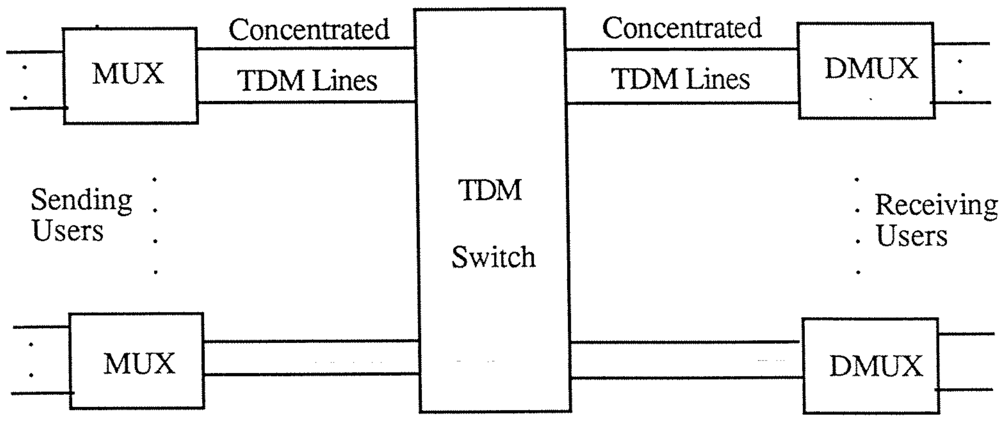

Figure 5.2: SS/TDMA hierarchical switching system

Application 2: Hierarchical switching systems

Hierarchical time division multiplexed (TDM) switching systems have been proposed [39] that connect sending users via a bank of multiplexers, followed by a TDM switch, followed by a bank of demultiplexers to receiving users, as in Fig. 5.2. Hence, the switch has three stages. Advantages of the hierarchical structure are that fewer switches may be needed to serve the user population; trunking efficiency may be increased due to the fact that end users have access to multiple input links; and modular growth is possible if additional lines and multiplexers/demultiplexers are required.

The hierarchical switching systems we consider in the remainder of this paper have an arbitrary number of stages, but are required to conform to a tree topology. A special case of our switching systems includes the one in Fig. 5.2

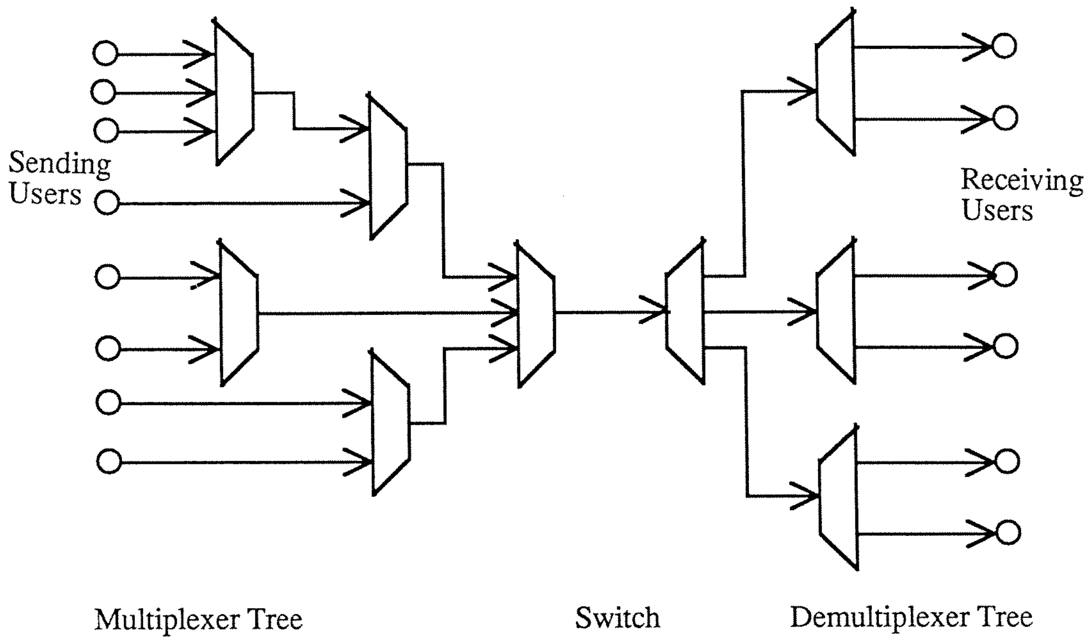

Figure 5.3: Hierarchical switching system

but where the middle $n \times n$ switch can transport at most $k \leq n$ packets per slot.

Note that the central TDM switch can also be regarded as a multiplexer feediing a demultiplexer. Thus the switching system can be regarded as being composed of only two elements, as shown in Fig. 5.3. We will use this model for the system architecture in the rest of this chapter.

Application 3: File transfers in computer networks

Consider a communications network where each node has several ports that can simultaneously transfer files, file transmission can be preempted, and the maximum number of simultaneous transfers at any time in the network is fixed. Each user is connected to a single network node and assigned a unique port on that node for file transfers. Such a network can be modeled as a hierarchical switching system, or a specialized tree-structured architecture (a special case of the architecture shown in Fig. 5.1).

5.3 The Tree scheduling algorithm

In order to develop an algorithm for solving *TreeDTS* optimally, we first introduce some notation and some definitions. We then introduce the notion of critical transfers and develop the outline of an algorithm.

5.3.1 Basic definitions and notation

The sets of senders and receivers are denoted V_S and V_R, and the sets of multiplexers and demultiplexers are denoted V_M and V_D. The architecture graph is sometimes called a system graph in this section, and is shown in Fig. 5.4. It is denoted (V, E) where $V = V_S \cup V_M \cup V_D \cup V_R$ and the directed links E are as follows. The set $E = E_M \cup E_D \cup \{[x_M, x_D]\}$, where $x_M \in V_M$, $x_D \in V_D$, $(V_S \cup V_M, E_M)$ denotes the multiplexer tree with root node x_M, and $(V_R \cup V_D, E_D)$ denotes the demultiplexer tree with root node x_D. In addition, the links of $(V_S \cup V_M, E_M)$ are directed towards x_M, and the links of $(V_R \cup V_D, E_D)$ are directed away from x_D. Note that a multiplexer node $v \in V_M$ (resp., demultiplexer node $v \in V_D$) is one with a single outgoing (resp., incoming) link and at least two incoming (resp., outgoing) links. Let $n = |V_S| + |V_R|$.

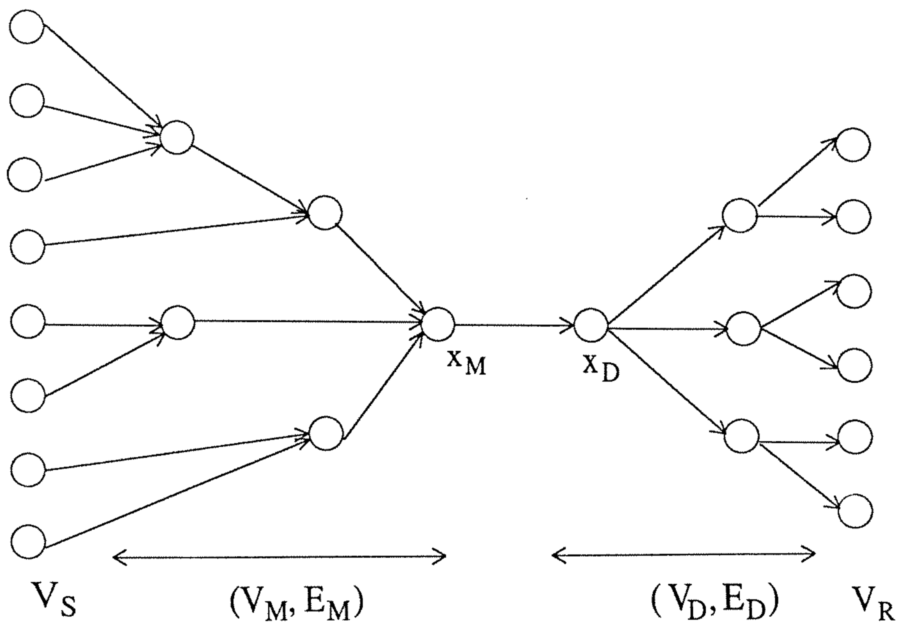

Figure 5.4: Switching system graph (V, E)

For each link $e \in E$, the positive integer $c(e)$ is the number of transfers that can be carried on simultaneously over that link; in terms of the satellite switching application, it is the number of packets that can be delivered over the link in one time slot. For each node $v \in V$, let $I(v)$ and $O(v)$ be the sets of incoming and outgoing links of v, respectively. We assume for a multiplexer node $v \in V_M$, $\sum_{e \in I(v)} c(e) \geq \sum_{e \in O(v)} c(e)$; and for a demultiplexer node $v \in V_D$, $\sum_{e \in I(v)} c(e) \leq \sum_{e \in O(v)} c(e)$. For a user node $v \in V_S \cup V_R$, let $C(v)$ be the capacity $c(e)$ of the single link e incident to v.

Note that $|V| \leq 2n$, since $(V_S \cup V_M, E_M)$ and $(V_R \cup V_D, E_D)$ are trees and each node in V_M (resp., V_D) has at least two incoming (resp., outgoing links) links.

We now introduce some specializations of the definitions given in Chapter 2, in order to simplify the discussion of the solution to *TreeDTS*. Let $E_{SR} = \{[u,v] : u \in V_S, v \in V_R\}$. We call any nonnegative integer matrix $r = (r(e) : e \in E_{SR})$ a *traffic matrix*. The interpretation of $r([u,v])$ is the number of packets to be transferred through the system from user u to user v; in effect it captures the resource graph of the problem. If r is a traffic matrix then let $t_r = (t_r(e) : e \in E)$, where $t_r(e)$ is the number of packets to be transferred over link e with respect to r. More formally, t_r satisfies the following: if e is the outgoing (resp., incoming) link of node $v \in V_S$ (resp., $v \in V_R$) then $t_r(e) = \sum_{u \in V_R} r([v,u])$ (resp., $\sum_{u \in V_S} r([u,v])$); and for all nodes $v \in V_M \cup V_D$, $\sum_{e \in I(v)} t_r(e) = \sum_{e \in O(v)} t_r(e)$. Clearly, t_r can be calculated from r in $O(n^2)$ time.

A traffic matrix r is called a *feasible transfer* if for all $e \in E$, $t_r(e) \leq c(e)$. A *schedule* is a sequence $s = (d_i, r_i : i = 1, ..., m)$, where $d_i > 0$ is integer, and r_i is a feasible transfer. The *length* of the schedule is $\sum_{i=1}^{m} d_i$, the scalar d_i is called a *duration*, and m is called the *switching complexity*. A schedule $s = (d_i, r_i : i = 1, ..., m)$ is said to *satisfy* a traffic matrix r if $\sum_{i=1}^{m} d_i r_i = r$.

Note that a lower bound to the minimum length is $L(r) = \max_{e \in E} \lceil \frac{t_r(e)}{c(e)} \rceil$. In section 5.3.2, we provide the outline of an algorithm that finds a schedule of length $L(r)$ that satisfies r. Thus, the schedule has minimum length. In section 5.4, a time complexity analysis of a fast design for this algorithm is given.

For the rest of this chapter we will use the following notation. Suppose $f = (f(x) : x \in X)$, $g = (g(x) : x \in X)$, and X is some set. Then $f \leq$ (resp., $=, \geq$) g is interpreted as $f(x) \leq$ (resp., $=, \geq$) $g(x)$ for all $x \in X$.

5.3.2 The scheduling algorithm

In this section we first define a critical transfer. Informally, for any time slot, a critical transfer is the minimum transfer required to ensure that the lower bound of the schedule length decreases by one unit; a sequence of critical transfers will thus result in an optimum schedule. In the first two lemmas we show the condition a transfer must satisfy in order to be a critical transfer. In the third lemma we show that a critical transfer exists for our system, leading to an algorithm for solving *TreeDTS*.

Def. A feasible transfer g is called *critical* with respect to a traffic matrix h if $g \leq h$ and $L(h-g) = L(h) - 1$.

We also define $b_h = (b_h(e) : e \in E)$ to be a function of traffic matrix h such that $b_h(e) = \max\{0, t_h(e) - (L(h)-1)c(e)\}$.

Lemma 5.1 *Suppose h is a traffic matrix, g is a feasible transfer, and $d > 0$ is an integer such that $h \geq dg$. Then $L(h-dg) = L(h) - d$ if and only if $L(h-dg) \leq L(h) - d$.*

Proof. Since g is a feasible transfer, $t_h(e) - dt_g(e) \geq t_h(e) - dc(e)$. Therefore,

$$\begin{aligned} L(h-dg) &= \max_{e\in E}\lceil \tfrac{t_h(e)-dt_g(e)}{c(e)} \rceil \\ &\geq \max_{e\in E}\lceil \tfrac{t_h(e)}{c(e)} \rceil - d \\ &= L(h) - d. \end{aligned}$$

Since $L(h - dg) \geq L(h) - d$, the lemma is implied. □

Lemma 5.2 *Suppose h is a traffic matrix. A feasible transfer $g \leq h$ is critical with respect to h if and only if $t_g \geq b_h$.*

Proof. By definition, g is critical if and only if $L(h - g) = L(h) - 1$. From Lemma 5.1, we know $L(h-g) = L(h)-1$ if and only if $L(h-g) \leq L(h)-1$. The last inequality is equivalent to $\frac{t_{h-g}(e)}{c(e)} \leq L(h) - 1$ for all $e \in E$, because $L(h) - 1$ is integer valued. Next note that $\frac{t_{h-g}(e)}{c(e)} \leq L(h) - 1$ is equivalent to $t_g(e) \geq t_h(e) - (L(h)-1)c(e)$. Therefore, g is critical with respect to h if and only if for all $e \in E$, $t_g(e) \geq t_h(e) - (L(h) - 1)c(e)$. The lemma is implied since $t_g(e) \geq 0$ for all $e \in E$. □

For the next lemma, and the remainder of this section, we consider another network (V, E^*) (see Fig. 5.5), where the nodes are V, the directed links are $E^* = E \cup E_{RS}$, and $E_{RS} = \{[u,v] : u \in V_R, v \in V_S\}$. (Notice that the links in E_{RS} are oriented from receiver nodes to sender nodes.) A nonnegative vector $f = (f(e) : e \in E^*)$ is called a *flow*. For each $v \in V$, let $I^*(v)$ and $O^*(v)$ be the set of incoming and outgoing links, respectively, for the node v. A flow f is *node-conserved* if for all $v \in V$, $\sum_{e \in I^*(v)} f(e) = \sum_{e \in O^*(v)} f(e)$.

Def. For any traffic matrix h, let the *network link lower bound* $b_h^* = (b_h^*(e) : e \in E^*)$ and *network link capacity* $c_h^* = (c_h^*(e) : e \in E^*)$, such that

$$b_h^*(e) = \begin{cases} b_h(e), & \text{if } e \in E; \\ 0, & \text{if } e \in E_{RS}. \end{cases}$$

$$c_h^*(e) = \begin{cases} c(e), & \text{if } e \in E; \\ \min\{h([v,u]), C(u), C(v)\}, & \text{if } [u,v] = e \in E_{RS}. \end{cases}$$

Note that b_h^* and c_h^* are integer and $c_h^* \geq b_h^*$. Also, for the special case in [24] when "speed-up" is not allowed, we would replace $h([v,u])$ by 1 in the definition of c_h^* above; this change leads to minor modifications in the proofs below which are left to the reader.

Lemma 5.3 *For any traffic matrix h such that $L(h) \geq 1$, there is a critical feasible transfer g.*

Proof. The proof has three steps. We define a flow f and observe that it is node-conserved, show that $b_h^* \leq f \leq c_h^*$, and then use the corresponding integer-valued flow to define a traffic matrix. Let f be a flow such that

$$f(e) = \begin{cases} \frac{h([v,u])}{L(h)}, & \text{if } [u,v] = e \in E_{RS}; \\ \frac{t_h(e)}{L(h)}, & \text{if } e \in E. \end{cases}$$

Then f is node conserved. Next we show that $b_h^* \leq f \leq c_h^*$. Suppose $e = [u,v] \in E_{RS}$. Let e' and e'' be the outgoing link of v and the incoming link of u, respectively. (Notice that here we use the property that leaves of a tree have only one parent). Then $L(h) \geq \max\{\frac{t_h(e')}{c(e')}, \frac{t_h(e'')}{c(e'')}, 1\} \geq \max\{\frac{h([v,u])}{C(v)}, \frac{h([v,u])}{C(u)}, 1\}$; and

$$b_h^*(e) = 0 \leq f(e) = \frac{h([v,u])}{L(h)}$$

$$\leq \min\{C(u), C(v), h([v,u])\} \leq c_h^*(e).$$

```
Algorithm Tree.
Input: Traffic matrix $r$.
Output: Schedule $s = (d_i, r_i : i = 1, ..., m)$
    that satisfies $r$.

Let $m = 0$ and $r' = r$.
while $r' \neq 0$ do
    Let $m = m + 1$.
    Note that $r' = r - \sum_{i=1}^{m-1} d_i r_i$.
    Compute a critical feasible transfer
        $r_m$ of $r'$.
    Let $d_m > 0$ be the maximum value $d$ such
        that $r' \geq dr_m$ and
        $L(r' - dr_m) = L(r') - d$.
    $r' = r' - dr_m$
end
```

Table 5.1: The **Tree** algorithm

Suppose $e \in E$. Then $f(e) = \frac{t_h(e)}{L(h)} \leq \frac{L(h)c(e)}{L(h)} = c_h^*(e)$. Also, note that $\frac{t_h(e)}{c(e)} \leq L(h)$; $t_h(e) \leq L(h)c(e)$; $t_h(e)(1 - L(h)) \geq -(L(h) - 1)L(h)c(e)$; $t_h(e) \geq t_h(e)L(h) -(L(h) - 1)L(h)c(e)$; and $\frac{t_h(e)}{L(h)} \geq t_h(e) -(L(h) - 1)c(e)$. Thus, $f(e) \geq b_h(e)$. We are done verifying $b_h^* \leq f \leq c_h^*$.

From the *Integrality Theorem for Flows* (see [122]) we know that since there is a node-conserved flow f such that $b_h^* \leq f \leq c_h^*$, then there is a node-conserved *integer-valued* flow f' such that $b_h^* \leq f' \leq c_h^*$. Let g be a traffic matrix such that $g([u, v]) = f'([v, u])$. Then $t_g(e) = f'(e)$ for all $e \in E$. Note that g is a feasible transfer because $t_g(e) = f'(e) \leq c(e)$; and $g \leq h$ because $g([u, v]) = f'([v, u]) \leq h([u, v])$. Finally, g is critical since $f' \geq b_h^*$ and Lemma 5.2. $\square$

Def. Let $R_i = r - \sum_{j=1}^{i-1} d_j r_j$.

Theorem 5.1 *Algorithm* **Tree** *solves TreeDTS.*

Proof. Lemma 5.3 implies that a critical feasible transfer r_i can always be found for a traffic matrix R_i. Also, the value d_i is greater than zero, because r_i is a critical feasible transfer for R_i. Therefore, the sequence R_1, R_2,... is decreasing in value and the algorithm will eventually terminate with the schedule s. Since $\sum_{i=1}^{m} d_i = \sum_{i=1}^{m}[L(R_i) - L(R_{i+1})]$, the length of s is $L(r)$ and is optimal. □

For the rest of the section we discuss the computation of d_i, which is the maximum value d such that $R_i \geq dr_i$ and $L(R_i - dr_i) = L(R_i) - d$. The inequality $R_i \geq dr_i$ is equivalent to $d \leq \min_{e \in E} \lfloor \frac{t_{R_i}(e)}{t_{r_i}(e)} \rfloor$. From Lemma 5.1, $L(R_i - dr_i) = L(R_i) - d$ is equivalent to $\lceil \frac{t_{R_i}(e) - dt_{r_i}(e)}{c(e)} \rceil \leq L(R_i) - d$ for all $e \in E$. Since $L(R_i) - d$ is integer, the last inequality is equivalent to $\frac{t_{R_i}(e) - dt_{r_i}(e)}{c(e)} \leq L(R_i) - d$, which in turn is equivalent to

$$d \leq \beta_{R_i,r_i}(e)$$

$$= \begin{cases} \frac{L(R_i)c(e) - t_{R_i}(e)}{c(e) - t_{r_i}(e)} & \text{if } c(e) > t_{r_i}(e); \\ \infty, & \text{otherwise.} \end{cases}$$

$$= \begin{cases} \frac{[L(r) - \sum_{j=1}^{i-1} d_j]c(e) - t_{R_i}(e)}{c(e) - t_{r_i}(e)}, & \text{if } c(e) > t_{r_i}(e); \\ \infty, & \text{otherwise.} \end{cases}$$

Therefore, $d_i = \min\{\min_{e \in E} \beta_{R_i,r_i}(e), \min_{e \in E_{SR}} \lfloor \frac{t_{R_i}(e)}{t_{r_i}(e)} \rfloor\}$.

5.4 A time efficient design

In this section we discuss time efficient implementations of Algorithm **Tree** and the time complexity. First we prove a bound on the sum of the link

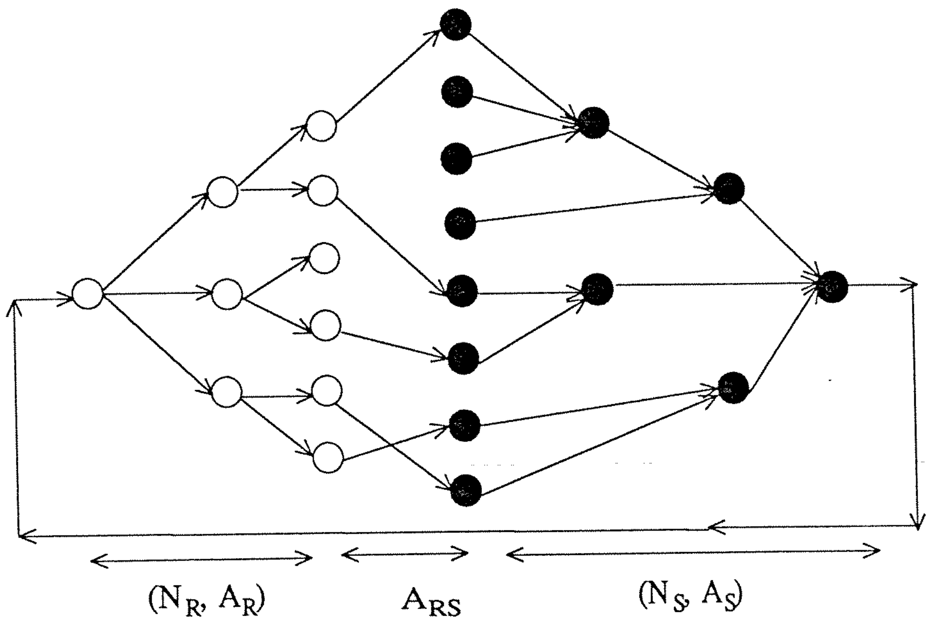

Figure 5.5: Network model $G = (V, E^*)$

capacities in the network (V^*, E^*), and a bound on the switching complexity m of Algorithm **Tree**. These bounds are used in the calculation of the time complexity of our implementation of Algorithm **Tree**.

Lemma 5.4 *Let* $C = \frac{1}{n}\sum_{v \in V_S \cup V_R} C(v)$ *and* $c^* = (c^*(e) : e \in E^*)$ *such that*

$$c^*(e) = \begin{cases} c(e), & \text{if } e \in E; \\ \min\{C(v), C(u)\}, & \text{if } e = [u,v] \in E_{RS}. \end{cases}$$

Then $\sum_{e \in E^*} c^*(e)$ *is* $O(n^2C)$.

Proof. We know $\sum_{e \in E_{RS}} c^*(e) \le \sum_{[u,v] \in E_{RS}} [C(u) + C(v)] \le n \sum_{v \in V_R \cup V_S} C(v)$ $= n^2C$. Now consider the directed spanning tree $(V' \cup \{x_D\}, E')$, where V' $= V_S \cup V_M$and $E' = E_M \cup \{[x_M, x_D]\}$. Let $c' = (c'(e) : e \in E')$, where $c'(e) =$ $C(v)$ if $v \in V_S$ and e is the outgoing link of v, and if $v \in V_M$ then $\sum_{e \in I(v)} c'(e)$

$= \sum_{e\in O(v)} c'(e)$. Since $\sum_{e\in I(v)} c(e) \geq \sum_{e\in O(v)} c(e)$ for each $v \in V_M$, it follows that $c' \geq c^*$. Let p_v be the length of the path in (V', E') from node $v \in V'$ to node x_D. Then $\sum_{e\in E'} c^*(e) \leq \sum_{e\in E'} c'(e) = \sum_{v\in V_S} p_v C(v)$. Due to the fact that each node $v \in V_M$ has at least two incoming links and (V', E') is a tree, we know $p_u \leq |V_S|$ for any $u \in V'$. Therefore, $\sum_{e\in E'} c^*(e) \leq |V_S| \sum_{v\in V_S} C(v)$ $\leq n \sum_{v\in V_S} C(v)$. Similarly,

$$\sum_{e\in E_D \cup \{[x_M, x_D]\}} c^*(e) \leq n \sum_{v\in V_R} C(v).$$

Thus, $\sum_{e\in E} c^*(e) \leq n \sum_{v\in V_S \cup V_R} C(v)$ and we are done. □

Lemma 5.5 *Suppose h is a traffic matrix such that $L(h) \geq 1$. Let f be an integer, node-conserved flow and g be the traffic matrix such that $g([u, v]) = f([v, u])$. Then g is a critical feasible transfer for h if and only if $b_h^* \leq f \leq c_h^*$.*

Proof. Note that for $e \in E$, $f(e) = t_g(e)$. Next, note that $f \leq c_h^*$ if and only if both g is a feasible transfer and $g \leq h$. Hence, Lemma 5.2 implies that g is a critical feasible transfer for h if and only if $b_h^* \leq f \leq c_h^*$. □

Let f_i be the node-conserved flow such that

$$f_i([u, v]) = \begin{cases} r_i([v, u]), & \text{if } [u, v] \in E_{RS}; \\ t_{r_i}([u, v]), & \text{if } [u, v] \in E. \end{cases}$$

Lemma 5.6 *The switching complexity m of Algorithm* **Tree** *is $O(n^2 C)$.*

Proof. Let the residual traffic at iteration i after a transfer r_i of duration d time slots be $r^d = R_i - dr_i$. From Lemma 5.5, r_i is a critical feasible transfer for r^d if and only if $b^*_{r^d} \leq f_i \leq c^*_{r^d}$. Then

$$d_i = 1 + \max\{d \geq 0 : b^*_{r^d} \leq f_i \leq c^*_{r^d}\}.$$

Therefore, for each $i < m$, at least one of the two inequalities must hold: $b^*_{R_i} \neq b^*_{R_{i+1}}$ or $c^*_{R_i} \neq c^*_{R_{i+1}}$, i.e., either the lower bound or the upper bound of the flow changes.

Next note that $c^*_{R_i} \geq c^*_{R_{i+1}}$ for $i < m$, because $R_i \geq R_{i+1}$. Since $R_{i+1} = R_i - d_i r_i$, if $e \in E$ then

$$b^*_{R_{i+1}}(e) = \max\{0, t_{R_{i+1}}(e) - (L(R_{i+1}) - 1)c(e)\}$$

$$= \max\{0, t_{R_i}(e) - d_i t_{r_i}(e) - (L(R_i) - d_i - 1)c(e)\}$$

$$= \max\{0, t_{R_i}(e) - (L(R_i) - 1)c(e) + (c(e) - t_{r_i}(e))d_i\}$$

$$\geq b^*_{R_i}(e).$$

If $e \in E_{RS}$ then $b^*_{R_{i+1}}(e) = 0 = b^*_{R_i}(e)$.

Thus, $b^*_{R_1} \leq b^*_{R_2} \leq \dots \leq b^*_{R_{m+1}}$, $c^*_{R_1} \geq c^*_{R_2} \geq \dots \geq c^*_{R_{m+1}}$, and $b^*_{R_i} \neq b^*_{R_{i+1}}$ or $c^*_{R_i} \neq c^*_{R_{i+1}}$. This implies $m \leq m + \sum_{e \in E^*}(b^*_{R_1}(e) + c^*_{R_{m+1}}(e)) \leq \sum_{e \in E^*}(b^*_{R_{m+1}}(e) + c^*_{R_1}(e))$, which is $O(n^2 C)$ from Lemma 5.4. □

Theorem 5.2 *Algorithm* Tree *can be designed so that it has a time complexity $O(n^4 C)$.*

Proof. Each iteration i of Algorithm **Tree** requires computing r_i, d_i, and updating r'. The implementation of the algorithm we consider will compute r_i by constructing an integer, node-conserved flow f_i such that $b^*_{R_i} \leq f_i \leq c^*_{R_i}$. By Lemma 5.5, the traffic matrix r_i, where $r_i([u,v]) = f_i([v,u])$, is a critical feasible transfer for R_i.

To construct f_{i+1} we consider an implementation that uses f_i, where $f_0 = (0 : e \in E^*)$. Let f'_{i+1} be a flow, that may not be node-conserving, such that

$$f'_{i+1}(e) = \min\{c^*_{R_{i+1}}(e), \max\{f_i(e), b^*_{R_{i+1}}(e)\}\}.$$

Thus, $b^*_{R_{i+1}} \leq f'_{i+1} \leq c^*_{R_{i+1}}$. For each node v, let the surplus flow $\delta_{i+1}(v)$ $= \sum_{e \in I^*(v)} f'_{i+1}(e) - \sum_{e \in O^*(v)} f'_{i+1}(e)$. Let f''_{i+1} be a flow such that $f''_{i+1} = f'_{i+1}$. The flow f''_{i+1} can be modified by the *Feasible Distribution Algorithm* in Rockafellar [122] so that f''_{i+1} becomes integer, node-conserved, and $b^*_{R_{i+1}} \leq f''_{i+1} \leq c^*_{R_{i+1}}$. Then $f_{i+1} = f''_{i+1}$. The Feasible Distribution Algorithm will modify f by applying at most $\frac{1}{2}\sum_{v \in V} |\delta_{i+1}(v)|$ integer flow augmentations, similar to flow augmentations of the Ford-Fulkerson Labeling Algorithm [42]. Each flow augmentation will take $O(n^2)$ time. Therefore, the transformation of f''_{i+1} will take $O(\frac{1}{2}\sum_{v \in V} |\delta_{i+1}(v)| n^2)$ time. Thus, the time to compute $f_1, f_2, ..., f_m$ is $O(Dn^2)$, where $D = \sum_{i=1}^{m} \sum_{v \in V} |\delta_i(v)|$.

Since $|\delta_{i+1}(v)| \leq \sum_{e \in I^*(v) \cup O^*(v)} |f'_{i+1}(e) - f_i(e)|$,

we have

$$D \leq \sum_{i=0}^{m-1} \sum_{v \in V} \sum_{e \in I^*(v) \cup O^*(v)} |f'_{i+1}(e) - f_i(e)|$$

$$\leq \sum_{i=0}^{m-1} 2 \sum_{e\in E^*} |f'_{i+1}(e) - f_i(e)|$$

$$= 2 \sum_{e\in E^*} \sum_{i=0}^{m-1} |f'_{i+1}(e) - f_i(e)|.$$

Note that, informally, either $f'_{i+1}(e)$ is increased as the lower bound increases to $b^*_{R_{i+1}}(e)$, or is decreased as the upper bound decreases, i.e.,

$$\begin{aligned} &|f'_{i+1}(e) - f_i(e)| \\ &= \max\{b^*_{R_{i+1}}(e) - f_i(e), 0\} \\ &\qquad + \max\{f_i(e) - c^*_{R_{i+1}}(e), 0\} \\ &\leq b^*_{R_{i+1}}(e) - b^*_{R_i}(e) + c^*_{R_i}(e) - c^*_{R_{i+1}}(e). \end{aligned}$$

Therefore, $D \leq \sum_{e\in E^*} c^*(e)$, and applying Lemma 5.4, D is $O(n^2C)$. This implies that the time it takes to compute $f_1, f_2, ..., f_{m+1}$ is $O(n^4C)$.

For each iteration i of Algorithm **Tree**, the time to compute d_i, $b^*_{R_i}$, and $c^*_{R_i}$ and to update r' is $O(n^2)$. By Lemma 5.6, the number of iterations is $O(n^2C)$. We can conclude that the time complexity of the algorithm is $O(n^4C)$. $\square$

5.5 Experimental evaluation

In this section we describe the results of an experimental evaluation of the performance of the **Tree** algorithm. since the algorithm always produces an optimal schedule, the key question is the time that it takes to produce that schedule.

We implemented the **Tree** algorithm as a C program, generally along the lines we have discussed in this chapter. The main difficulty was in implementing the routines to find augmenting paths to modify a flow so as to make it node-conserved. The shortest augmenting path subroutines were implemented using the algorithms sketched in Gibbons [58] as a guide.

The program implementing **Tree** was evaluated by measuring the CPU time it took to execute when presented with uniformly randomly generated graphs as inputs. Random graphs were generated for selected combinations of the input parameters using a pseudo-random number generator [94]. The programs were executed on a Sun Sparc 2 workstation running the SunOS$^{\text{TM}}$ Release 4.1.1 operating system after being compiled using the Sun Microsystems C compiler (bundled with SunOS Release 4.1.1), with Level 4 optimization enabled ("-O4" option). The data structures for the program fit in the 32 MB main memory of the system, and so the program does not perform any I/O in order to execute, except to read the input graph and print results.

Tree expt. 1. Effect of varying number of senders. The first experiment estimated the performance of **Tree** for architecture graphs in which both the multiplexer tree and the demultiplexer tree are complete balanced binary trees with unit capacities for all links, and all edges in the resource graph have unit weights. Thus the average capacity of the user links in the architecture graph is $C = 1$, the number of senders and receivers is the same, and if there are S senders there are $(S - 1)$ multiplexers, each with two incoming links and one outgoing link. Clearly S is constrained to be a power of 2, and once it is chosen the structure of the architecture graph, and the number of nodes in the resource graph, is completely fixed.

The only random variable remaining in the architecture graph is the capacity of the link from the root of the multiplexer tree to the root of the demultiplexer tree. The capacity of this link is set to be an integer drawn randomly from the interval $[2, S/2]$. The edges of the resource graph (i.e., the entries of the traffic matrix) were generated by a pseudo-random number generator [94]. Each edge exists with probability 0.5; if it exists, it carries unit traffic.

For each value of S, one hundred random input instances, or *batches*, were generated as described above, and the CPU time taken by **Tree** to generate a schedule for the entire one hundred batches, as reported by the C Shell "time" command, was recorded. The "time" command used has a resolution of 20 ms. Since all (except one) of the measurements are of times greater than 4 seconds, and many are of times of hundreds of seconds, this resolution is sufficient. The mean for each set of 100 measurements was calculated, plotted and curve-fitted as described in section 3.7.

In Fig. 5.6 the mean CPU time per batch taken by **Tree** as the number of senders (or receivers) S is increased, is shown. The curve-fit shown in the figure corresponds to the following equation and correlation coefficients:

$$Tree(n) = 2.78 \times 10^{-5}\ n^3 - 9.26 \times 10^{-4}\ n^2 - .055,$$

$$R3^2 = .99,\ R2^2 = .98,\ R1^2 = .89,\ R0^2 = 1.0$$

For the sake of completeness, some data on 'internal' measures of performance of **Tree** are summarized in Table 5.2. These are the average number

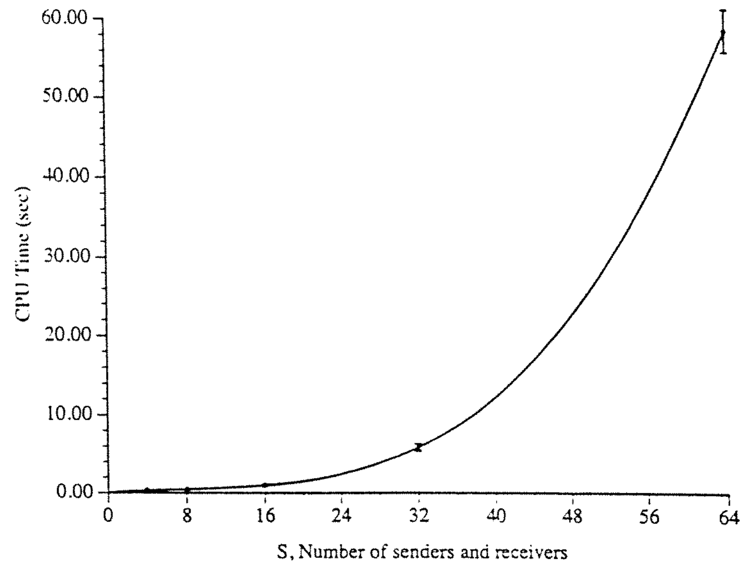

Figure 5.6: CPU time versus number of senders or receivers for unit-length transfers and complete binary tree architectures

of unit-length transfers to be scheduled per batch, the average optimal schedule length per batch in time slots, the average number of augmenting paths calculated in order to find the optimal schedule, and the average CPU time per batch.

Tree expt. 2. Effect of varying transfer lengths. In the second experiment, the architecture graph of the input instances to **Tree** is kept fixed and only the lengths of transfers are varied. The architecture graph is fixed to be a balanced binary tree with 64 senders and unit capacity links. Each edge in the resource graph exists with probability 0.5; it is labeled with the length of the transfer by a random number drawn from the interval [1, K], where K is the maximum transfer length and is varied from 2 to 1024. For each value of K, one hundred batches are generated randomly as described above, and the time for calculating the optimal schedule for the entire one hundred

Senders, S	Transfers	Makespan	Paths	CPU Time (sec)
8	7.99	5.0	7.46	.014
16	31.28	17.77	29.96	.050
32	127.94	68.34	123.97	.365
64	515.09	266.61	501.27	4.362

Table 5.2: Behavior of **Tree** as number of senders is varied for balanced binary trees of unit capacities and unit-length transfers

Maximum Transfer Length, K	Transfers	Traffic	Make-span	Switchings	Paths	CPU Time (sec)
1	515.09	515.09	266.61	266.61	501.27	4.362
2	515.09	771.88	401.76	348.69	505.78	5.242
16	515.09	4377.62	2290.68	486.37	513.54	6.709
128	515.09	33217	17402	511.35	514.86	6.982
512	515.09	132092	69207	514.21	515.06	6.999
1024	515.09	263929	138282	514.53	515.06	7.017

Table 5.3: Behavior of **Tree** as maximum transfer length is varied for balanced binary trees of unit capacities and 64 senders

batches is measured using the "time" command, as for Tree expt. 1. Figure 5.7 shows the mean CPU time taken to calculate the schedule, per batch, as K is varied.

Table 5.3 shows 'internal' measures of the behavior of **Tree** as K is varied, averaged per batch. Since transfers are no longer of unit length, the total traffic per batch does not equal the number of transfers (unlike Table 5.2), and the makespan does not equal the switching complexity. Thus these quantities are displayed separately.

Tree expt. 3. Effect of varying link capacities. In the third experiment,

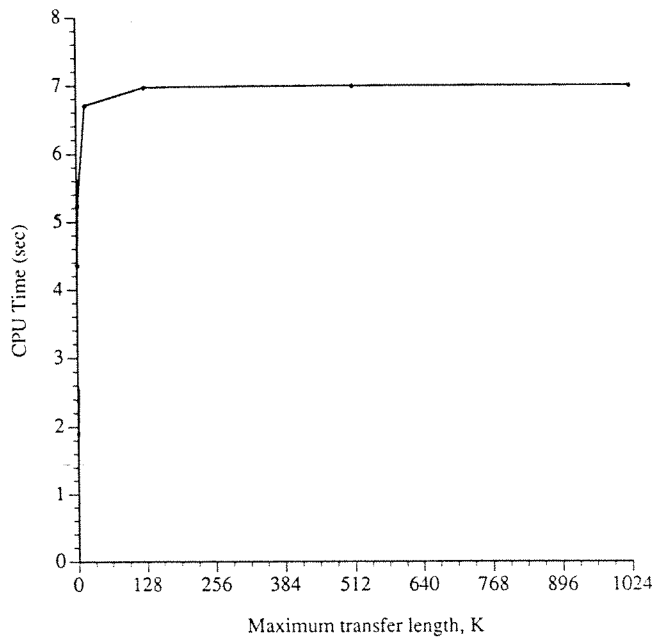

Figure 5.7: CPU time versus maximum transfer length for complete binary tree architectures with 64 senders

balanced binary trees were again considered, but the capacity of the user links, C, was varied as described below. The maximum transfer length was set at $K = 2$.

Recall that the architecture graph for *TreeDTS* contains sending users connected to a bank of multiplexers. In order to keep the structure of the graph 'sensible', it was assumed that the sum of the incoming link capacities to a multiplexer is at least equal to the outgoing link capacity (and the analogous assumption for demultiplexers). When $C = 1$, and the tree is not degenerate (i.e., the branching factor is at least two), this assumption is easily captured by generating input architecture graphs in which all links have unit capacity. For $C > 1$, we capture this assumption by setting the capacity of all user links to exactly C. The capacity of non-user links in the architecture graph is then generated as an integer drawn randomly from the interval $[1, C']$, where

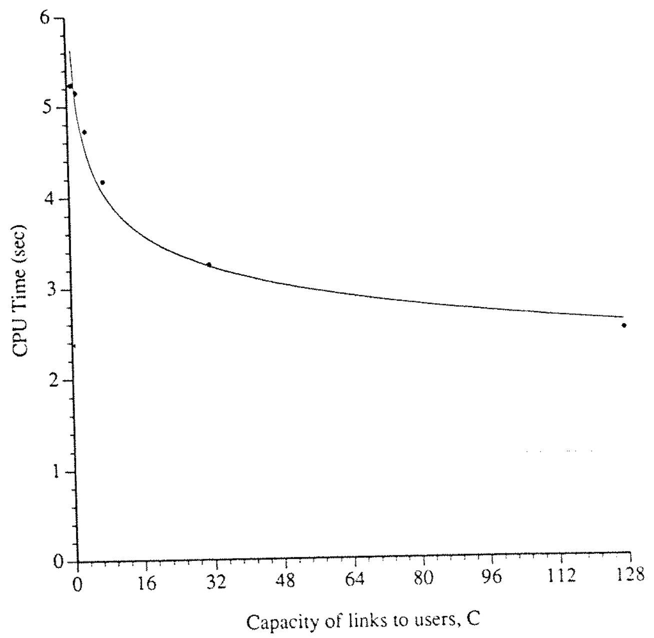

Figure 5.8: CPU time versus user link capacity for complete binary tree architectures with 64 senders

C' is the sum of the incomping links to a multiplexer (or outgoing links at a demultiplexer).

Figure 5.8 shows the mean CPU time per batch for generating an optimal schedule using Tree for 100 batches of randomly generated inputs as C is varied from 1 to 128. The curve is described by:

$$Tree(C) = 5.64C^{-0.16}, \qquad R^2 = .98$$

The internal performance measures of the algorithm are summarized in Table 5.4.

It is interesting to note that the execution time of the algorithm decreases as C is increased, if the architecture and resource graph (traffic) inputs are

User Link Capacity, C	Transfers	Traffic	Make-span	Switchings	Paths	CPU Time (sec)
1	515.09	515.09	266.61	266.61	501.27	4.362
2	515.09	771.88	383.23	335.20	505.88	5.172
4	515.09	771.88	337.21	295.48	506.24	4.729
8	515.09	771.88	284.32	249.04	512.69	4.181
32	515.09	771.88	172.73	151.0	541.20	3.251
128	515.09	771.88	78.56	73.42	573.88	2.498

Table 5.4: Behavior of **Tree** as user link capacity is varied for balanced binary trees of 64 senders

kept the same. Informally, the explanation for this is straightforward: as C is increased, more traffic can be transferred per time slot, so that the switching complexity (number of feasible transfers that need to be calculated) decreases. The multiplicative factor of C in the theoretical asymptotic time complexity arises from assuming that in the worst case, at each time slot, the capacity or lower bound of any link changes by at most one unit; then the number of times that feasible transfers have to be calculated is proportional to the maximum difference between capacity and lower bound for any link, which in turn is proportional to C. In order to bring the theoretical and experimental results in closer intuitive agreement, another method should be used to bound the switching complexity of the algorithm, i.e., the theoretical analysis should be sharpened.

5.6 Discussion

5.6.1 Previous related work

The previous work for scheduling data transfers in tree architectures has all been done for special cases of the *TreeDTS* problem. In addition, it has all been done in the context of specific application domains. The result is that the previous research has been reported in a wide variety of journals and conferences, using specialized notation and jargon, and with no cross-referencing of research across application areas. In the following we summarize some of this research using our scheduling model.

The ground-breaking work on data transfer scheduling was done by Coffman, Garey, Johnson and LaPaugh, in their 1985 paper which focused on the problem of non-preemptive file transfers in a distributed network [27]. The problem they address differs from *TreeDTS* in that $Preempt = false$, and general architecture graphs were considered. The paper is remarkable in defining a new and interesting class of problems and presenting a large number of results, including NP-completeness of various classes of problems, approximation algorithms for these problems with performance guarantees, polynomial-time algorithms for special cases, and distributed algorithms for some situations. The suggestions for future work mention the realistic situation of file transfers where intermediate network nodes may be used to forward files if the sender and receiver have no direct link. These and other variations of the problems mentioned in the paper have subsequently been addressed by several researchers (e.g., see [23, 142, 101] and references therein), and continue to receive attention.

Our work differs from that of Coffman et al [27] and their successors in that we are considering the situation where preemption is allowed. This assumption not only makes the scheduling problem easier, but is well-justified for many applications. For instance, in the case of parallel I/O, data is typically read in fixed-size blocks from magnetic media, allowing preemption between block boundaries. Similarly, in the case of satellite TDMA switching, data is transferred in fixed-size packets in order to time-share the medium using time-division multiplexing, and communications protocols often assume packetization of data in order to operate correctly. In the rest of this section, we consider preemptive scheduling only.

Eng and Acamopra [39] consider a special case of $TreeDTS$ for the satellite switching application. They place the following additional restrictions on the architecture graph:

1. The hierarchical switching system has three stages, i.e., AG is a tree with exactly four levels (a dummy root, and two subtrees of three levels each), and
2. arcs connected to leaves (users) have a capacity of exactly 1 packet per second, i.e., $C = 1$.

Eng and Acampora [39] provide necessary and sufficient conditions for the existence of an optimal schedule. Bonnucelli [11] and Liew [96] provide exact optimal scheduling algorithms of time complexity $O(n^5)$ under these conditions. Informally, the basic approach is, at every time slot, to identify critical transfers - those whose remaining traffic demand equals the optimal makespan calculated using Eng and Acampora's conditions. At every time slot as many packets as possible from these critical transfers are transferred - the number that can be sent is calculated using a max-flow algorithm. The resulting schedule will have the optimal makespan. We note that our algorithm will solve this restricted problem in time $O(n^4)$.

Choi and Hakimi [24] study a special case of $TreeDTS$ for the file transfer application. They place the following constraints on the architecture graph:

1. AG is a tree with exactly three levels, and
2. a) arcs of AG connected to leaves (users) have an arbitrary positive integer capacity equal to the number of ports, (this case is called *speed-up*), or

   b) arcs of AG connected to leaves (users) have capacity exactly 1 and (this case is called no speed-up)

In addition, preemption is allowed arbitrarily, i.e., packets may be of variable length. We shall show in a later chapter that this extension can be handled by a simple modification of $TreeDTS$.

Choi and Hakimi's problem generalizes that of Bonnucelli [11] and Liew [96] in allowing non-unit capacities and allowing variable-length packets, but is restricted in allowing only three levels in the AG. The term speed-up implies that multiple ports may be used to expedite the data transfer between a given sender-receiver pair. Thus allowing a vertex v to engage in upto $\alpha(v)$ transfers means that a transfer of length w between vertices u and v only requires time $w/(\min(\alpha(u), \alpha(v)))$.

Choi and Hakimi's method is interesting because it illustrates an alternative approach to the problem. They essentially use a generalized edge-coloring method in order to obtain optimal schedules. Their problem is a generalization of the *SimpleDTS* and *DTS* problems that we have solved using standard edge-colorings of bipartite graphs, but is a restriction of *TreeDTS*, for which we used more powerful network flow methods. Further, they handle the case where at most $k \leq n$ transfers may take place at any given time by using a procedure similar to the k-filling procedure we use for *DTS*. Thus not only their problem, but their solution method falls at a point in between the approaches that we have taken to solve the *DTS* and *TreeDTS* data transfer scheduling problems.

Under speed-up, Choi and Hakimi's algorithm runs in time $O(C_{sum}m^2)$, where C_{sum} is the sum of the link capacities at the user nodes and m is the number of data transfers, i.e., it runs in time $O(Cn^5)$. Without speed-up the algorithm runs in time $O(C_{sum}m^2 + C_{sum}^2 m)$, i.e., $O(Cn^5 + C^2n^4)$.

In the same paper, Choi and Hakimi [24] also consider the situation where a non-zero *switching time* is required in order to change from one switch configuration to another. They show that even a simple version of this problem is NP-complete, and design approximation algorithms.

Our formulation of the problem allows modeling of speed-up since all arcs connected to leaves have capacities equal to the number of ports. For the case without speed-up we modify the network model so that arcs representing the data transfers themselves (arcs from receiving users to sending users) have capacity 1. In either case our algorithm generalizes and provides a faster algorithm for the problem than the optimal algorithms of Choi and Hakimi [24], provided that preemption is allowed only at fixed packet length boundaries. As already mentioned, in a later chapter we will show that even for the latter case, the **Tree** optimal algorithm can be extended to generalize and have better time complexity than Choi and Hakimi's optimal algorithms.

The data transfer problem has continued to receive attention for the satellite switching system application. Chalasani and Varma [20] present an algorithm for the restricted system studied by Eng and Acampora [39, 11, 96] that takes time $O(n^2 \min(L, n^2) \min(k, n^{.5}))$, where L is the schedule length and k is the capacity of the central switch in the three-stage network. Thus their algorithm takes time $O(n^{4.5})$.

Chalasani and Varma [136] have also presented an algorithm whose basic idea is similar in spirit to the approach we have used to design **Tree**: instead of recalculating flows from scratch at each step of the algorithm, calculate only a modification to the flow. However, their algorithm again only applies to the three-stage unit-capacity network specified by Eng and Acampora [39], and runs in time $O(n^2 + cn)$, where c is the number of traffic units to be reassigned in the traffic matrix, i.e., it runs in time $O(Ln^2)$, where L is the schedule length. There are two interesting points in their approach, however. The first is that they use a procedure similar to k-filling to convert the pr-oblem for a three-stage hierarchical architecture to that for a single switch (the architecture graph of DTS). The second is that they use an algori-thm originally developed for routing traffic in a switching network to perform scheduling.

It is clear that the **Tree** algorithm solves more general cases of the optimal data transfer scheduling problem than those considered in [39, 11, 96, 20, 136]. It also solves more general cases of the problem than the polynomial-time solvable problem studied in [24]. In general, for arbitrary traffic matrices (and hence schedule lengths that may be greater than $O(n^2)$), **Tree** also performs better than previous algorithms [39, 24, 11, 96, 20, 136].

5.6.2 Conclusions and Further Work

We have defined the problem of data transfer scheduling in tree-structured architectures in our model, and shown that it generalizes previous work done in three different application areas: parallel I/O [132], satellite switching systems [39, 11, 96], and file transfers in computer networks [24]. The generalization comes in two forms: in allowing arbitrary tree topologies in the architecture, and in allowing arbitrary integer capacities for the communications links. We have developed an algorithm whose time complexity is $O(Cn^4)$, where C is the average capacity of the links connected to senders and receivers, and n is the number of senders and receivers. In general, this algorithm is faster than previous algorithms [24, 11, 96, 20, 136].

Our study has also been unique in that we have actually implemented the scheduling algorithm and investigated its behavior experimentally. For the case $C = 1$, unit transfer lengths, and complete binary tree architecture graphs, we have found that the CPU execution time of the algorithm grows, on average over randomly generated input instances, as $O(n^3)$ in our experiments, rather than the $O(n^4)$ worst-case theoretical behavior. As expected, the execution time of the algorithm is experimentally observed to be relatively insensitive to transfer lengths. However, we observed that, for a fixed architecture and traffic requirement, the execution time dropped as the user link capacity was increased. While we ofer an informal explanation for this result, it indicates that the theoretical analysis can be sharpened.

An obvious direction for extensions to this work is consider the use of heuristics in place of the optimal algorithm. We shall consider one such heuristic in Chapter 6. We have also considered the situation where data transfers can be preempted at arbitrary points, i.e., data packets can be of arbitrary length. This result is discussed in a later chapter.

For future work, we suggest two related questions. The first is whether the architecture graph can be generalized from a tree topology to an architecture with a higher connectivity. An example would be a hypercube network. Such an extension to our scheduling algorithms would be of significant practical interest. For instance, Ghosh and Agarwal [57] have proposed a hypercube network for I/O, in addition to the existing hypercube network for inter-processor communication, to overcome the parallel I/O bottleneck in hypercube multiprocessor architectures such as the Intel iPSC/2 and iPSC/860. An extension to higher-connectivity architectures would also be of theoretical interest in order to investigate whether polynomial-time or pseudo-polynomial time algorithms could be then be developed. So the second question that we suggest for future work is to determine the smallest set of extensions to the tree architecture defined in *TreeDTS* which still allow polynomial-time algorithms to be developed.

Chapter 6

A Fast Heuristic for Hierarchical Architectures

In the previous chapter we discussed an optimal algorithm for scheduling data transfers in tree structured architectures, and a theoretical as well as experimental evaluation of its performance. This optimal algorithm, **Tree**, thus applies to the *TreeDTS* scheduling problem, and has a time complexity of $O(Cn^4)$, where n is the number of users (or leaves in the architecture graph), and C is the average capacity of the links connected to the users. The experimental evaluation showed that, for the situations studied, the average execution time of the algorithm varied as n^3 rather than n^4 for random input graphs. While the theoretical worst-case time complexity is an improvement over previous algorithms in this area, and the experimentally measured average-case behavior gives us better performance than the worst-case behavior, the algorithm may still not be fast enough for some applications. Since the algorithm will be executed repeatedly in any realistic batch scheduling situation, any gain in speed will help improve overall system performance. This motivates the search for faster approximation algorithms to solve *TreeDTS*.

In this chapter we present an approximation algorithm based upon a simple greedy heuristic. In sec. 6.1 we describe the approximation algorithm, and in sec. 6.2 we describe an experimental evaluation of its performance, as well as a comparison of its performance with that of **Tree**. We end with a brief discussion.

6.1 A greedy heuristic

The basic idea of the greedy heuristic for *TreeDTS* is quite simple, and similar in spirit to the greedy heuristics for *SimpleDTS* and *DTS* discussed in Chapter 4. For each time slot, a set of transfers between sender-receiver pairs is chosen such that it is 'feasible', i.e., the resulting traffic on any link in the network does not exceed the capacity of that link. The "greedy" aspect of this approach lies in that as large a set as possible is chosen for each time slot; the "heuristic" aspect lies in that for each time slot, the transfers are examined once, in some order, and a transfer is added to the feasible set only if the resulting set will not violate the link capacity constraint. An algorithm based on this idea is described below.

Algorithm Greedy Tree Heuristic
Input: An instance of $TreeDTS = (PG, AG, RG, f, Preempt)$, where the bipartite resource graph is denoted $RG = (A, B, E)$.
Output: A schedule satisfying *TreeDTS*, represented as a set of feasible transfers for each time slot.

```
1. Assign some order $F = \langle e_1, e_2, ..., e_m \rangle$ to the edges of $E$
2. i := 0
```

```
3.  while $F \neq \{\}$ {
4.      $E' := \{\}$        /* Feasible transfers for this time slot */
5.      for each $e$ read in sequence from $F$ {
6.          if the traffic due to $\{e\} \cup E'$ for any link in $AG$ does not
            exceed that link's capacity {
7.              add $e$ to $E'$
8.              remove $e$ from $E$ and $F$
9.          }
10.     }
11.     i := i + 1
12.     Assign an order $F$ to the edges of $E$
13. }
```

Clearly, the performance of the heuristic will depend upon the ordering F of the edges in the resource graph. The bulk of the execution time of the algorithm will be spent in sorting the edges to obtain F, and in checking for each transfer and each time slot that the edge set E' represents a set of feasible transfers.

The heuristic that we have investigated simply orders the edges in E by the order in which they are presented to the scheduler, i.e., essentially a random ordering. Thus obtaining F from E takes no time. We call the resulting algorithm the *Greedy Random Assignment* (**GRA**) algorithm.

6.2 Experimental evaluation of the greedy heuristic

The **GRA** algorithm was implemented and evaluated experimentally in a manner similar to that described in 5.5 for **Tree**. In fact, many of the underlying routines in the implementation used were the same, as was the 'driver' program for parsing inputs and keeping statistics. Also, **GRA** was evaluated for the same set of input parameters as described for **Tree**, and using the same random number generator, seeds, and number of input graphs per experiment. Specifically, the three experiments described as **Tree expt. 1 - 3** in sec. 5.5 were repeated, with the **Tree** algorithm replaced by the **GRA** algorithm; the resulting experiments are called **GRA expt. 1 - 3**. There is one difference in the results obtained: since **GRA** is an approximation algorithm, there are two measures of peformance: the makespan of the schedule calculated, and the amount of time to needed to calculate the schedule. In the following, the figures presented for **Tree** in sec. 5.5 are shown again to allow easy comparison with those for **GRA**.

GRA expt. 1. Effect of varying nuumber of senders. The performance of **GRA** is compared with that of **Tree** for this experiment in Figures 6.1 and 6.2. From Fig. 6.1 we see that as the number of sending (or receiving) users grows, the savings in execution time obtained by using the heuristic increases. The curve fit is given by the equation:

$$GRA(n) = 7.89 \times 10^{-4}\, n^2 - 2.35 \times 10^{-2}\, n + 0.173,$$

$$R2^2 = .98,\ R1^2 = .90,\ R0^2 = .99$$

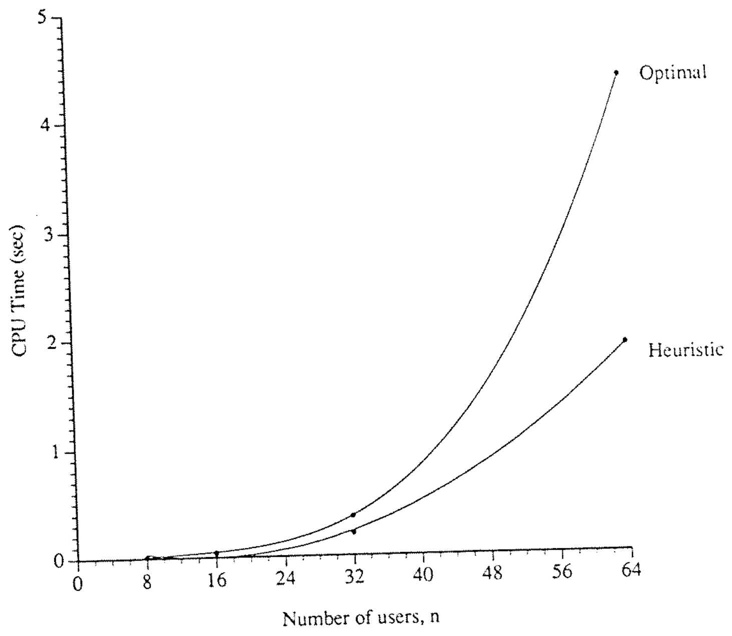

Figure 6.1: CPU time for **GRA** and **Tree** versus number of senders or receivers for unit-length transfers and complete binary tree architectures

Thus the increase in execution time of **GRA** for this experiment as the number of senders is increased is $O(n^2)$, as opposed to $O(n^3)$ for **Tree**. This improvement obviously comes for a price: the schedule obtained by **GRA** is, in general, not of minimum length. In Fig. 6.2 we consider the percentage increase in the schedule length obtained by **GRA**. Recall that for each value of n, one hundred batches were generated at random as inputs. For each batch, the percentage increase in schedule length was calculated. The figure shows the maximum and average values of the percentage increase over the one hundred batches. While the maximum percentage increase observed was almost 35%, the average penalty was only about 5% or less.

GRA expt. 2. Effect of varying transfer lengths. The execution time of **GRA** as the transfer length is varied is compared with that of **Tree** in Figure 6.3, and the percentage penalty paid in terms of schedule length is shown in

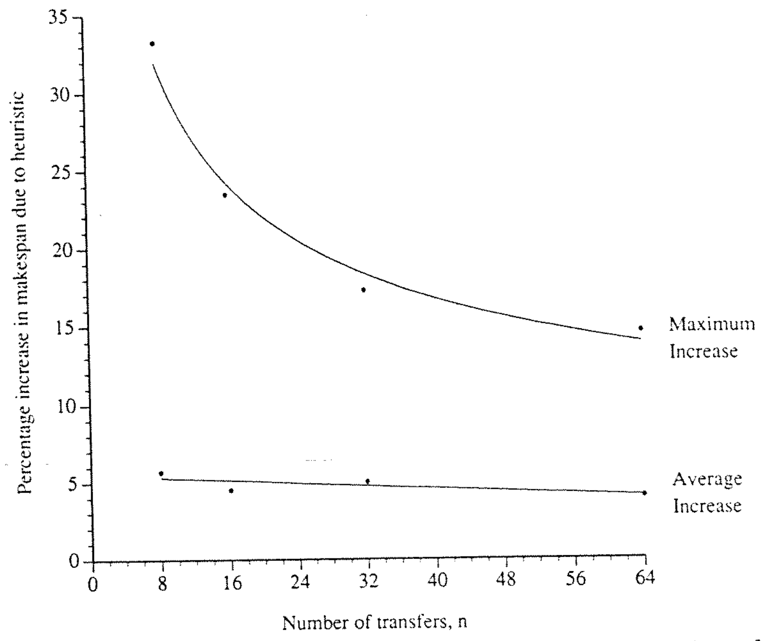

Figure 6.2: Maximum and average penalty paid for using the **GRA** heuristic instead of the **Tree** algorithm, versus number of senders or receivers for unit-length transfers and complete binary tree architectures

Fig. 6.4.

GRA expt. 3. Effect of varying capacities. The execution time of **GRA** as the user link capacities are varied is compared with that of **Tree** in Figure 6.5, and the percentage penalty paid in terms of schedule length is shown in Fig. 6.6. The curve-fit for the variation in execution time is given by the equation:

$$GRA(C) = 2.79C^{-0.25}, \qquad R^2 = .95$$