UNIVERSITY OF ALBERTA

SPARE CAPACITY DESIGN OF ATM VP-BASED RESTORABLE NETWORKS

By

YONG ZHENG ©

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfilment of the requirements for the degree of Master of Science

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

EDMONTON, ALBERTA

FALL 1997

0-612-22697-2

Dedicated to my parents Siming Zheng and Qiaoai Zhang

and my wife Tao Weng

Abstract

This thesis studies the capacity placement problem in ATM VP-based restorable networks. Previous work on this problem has been heuristic in nature and / or has treated the ATM spare capacity design problem with exact methods but in a manner that is essentially the same as for STM path restorable networks. In this thesis, we develop an optimization approach which lets us exploit the inherently statistical nature of the traffic in ATM in capacity planning for restoration. Oversubscription factors are defined as the ratio of total VP bandwidth allocation after restoration to the total installed capacity of the span. An oversubscription larger than 1.0 is one of the unique properties of ATM networks. There are two parts in this thesis. The first part is oriented towards capacity planning that would permit controlled oversubscription of bandwidth. Three integer program formulations are developed to achieve optimal capacity planning with a controlled oversubscription. Results show that significant capacity savings can be obtained relative to STM if ATM restoration is allowed even a modest restoration-induced oversubscription of bandwidth on surviving spans. Then the objective of the second part is to give quantitative guidelines towards determining a realistic oversubscription factor based on the resultant overload implication at the cell level. The overload is the worst case oversubscription after restoration in ATM networks. Simulations are completed with two traffic models: on/off fluid model and auto-regressive model. We find that the tolerable overload depends on many factors, such as the class and traffic model. In conclusion, a restorable ATM network planning framework is proposed to exploit the intrinsic differences between ATM and STM transport.

Acknowledgments

I would like to express my sincere thanks to my supervisor, Dr. Wayne D. Grover for his interest, advice, support and patience throughout my graduate program.

I would like to thank my parents SiMing Zheng and Qiaoai Zhang and my wife Tao Weng for their love and support during the work of this thesis.

I would also like to thank these people of TRLabs for providing such a supportive environment in which to carry out my research: Dr. Rainer Irashko, Dr. Mike MacGregor, Robert Hang and Dave Morley for their constructive suggestions, instructions and careful reading in this work. I thank Dr. Bruce Cockburn and Dr. Ehab Elmallah for reviewing this thesis.

I would like to thank those people of TRLabs for providing such a friendly environment: Songsong Sun, Sing Cheng, Endy Yeung, Robert Hang, Huy Nguyen, Demetrios Stamatelakis and Danny Li.

Table of Contents

List of Figures

List of Tables

List of Abbreviations

ABR	Available Bit Rate
AR	Auto Regressive
ATM	Asynchronous Transfer Mode
BER	Bit Error Rate
B-ISDN	Broadband Integrated Services Digital Network
CAC	Call Admission Control
CBR	Constant Bit Rate
CLR	Cell Loss Ratio
DCS	Digital Crossconnect System
IP	Integer Program
ITU	International Telecommunications Union
ksp	k-successively-shortest link-disjoint paths
LP	Linear Program
MIP	Mixed Linear Program
NNI	Network Network Interface
OAM	Operation, Administration and Management
OC	Optical Carrier
QoS	Quality of Service
STM	Synchronous Transfer Mode
UBI	Unspecific Bit Rate
UNI	User Network Interface
VBR	Variable Bit Rate
VC	Virtual Circuit
VCC	Virtual Circuit Connection
VCI	Virtual Circuit Identifier
VP	Virtual Path
VPC	Virtual Path Connection
VPI	Virtual Path Identifier

Chapter 1. ATM Restoration Problem

1.1 Restoration Problem

The requirements of today's telecommunication networks are changing rapidly with the introduction of high-capacity transmission links, the increased amount of data and voice traffic, and the role of telecommunications as a major part of the world's infrastructure. A failure occurring in such a large network can result in a huge loss of bandwidth, loss of service to users, and loss of revenue to operating companies. To ensure service continuity, service providers have increased their efforts to avoid failures if possible and restore network failures quickly. At the centre of these efforts lies a challenging question: how can service providers ensure affordable service continuity? In modern telecommunications practice, this is formally called the *restoration problem*. Because of the huge economic impact made by network failures, fast and automated restoration has become an essential adjunct to the deployment of large scale telecommunication networks [1, 3].

Restoration is normally achieved by rapid (within 2 seconds [2]) and accurate rerouting of affected traffic over a set of replacement paths through the spare transmission capacity in the network. These restoration paths should have enough capacity to restore the failed traffic. Also, these paths need to be link disjoint. The restoration problem is significantly different from the well-studied packet routing and call routing problems and, therefore, presents demanding real time computational challenges [1].

An important network restoration objective is to reduce network redundancy. Network redundancy is defined as the ratio of spare to working capacity in a network [1]. Networks with large redundancies are less economical because the spare capacity required to protect them is expensive. Minimizing the capacity requirements of the networks while maintaining the ability to restore the most common types of failures quickly, is a key objective when solving the restoration problem.

1.2 STM Restoration Schemes

STM network restoration schemes are generally classified into two types: centralized and distributed. For centralized schemes, all restoration routes are precomputed by a central controller, and then downloaded to all DCS machines in the network. In the case of a failure, affected nodes implement predetermined restoration routes and switch lost traffic to those routes. *Distributed schemes*, notably the self-healing algorithm [1], establish a set of replacement paths without centralized control. Distributed algorithms rely on the autonomous and independent action of all nodes. When a transmission span failure occurs, restoration messages are exchanged between nodes to restore the paths containing the failed span. The replacement connection is found by the network elements and rerouted depending on network resources available at the time of the failure. The primary advantage of distributed restoration is fast restoration and robustness compared with other restoration schemes [2]. Regardless of whether a network uses centralized or distributed restoration to restore a failure, sufficient spare capacity must exist to accommodate the restoration routes.

The restoration routes and the associated capacity design problem may involve either span restoration or path restoration. *Span restoration* re-routes failed working traffic over a set

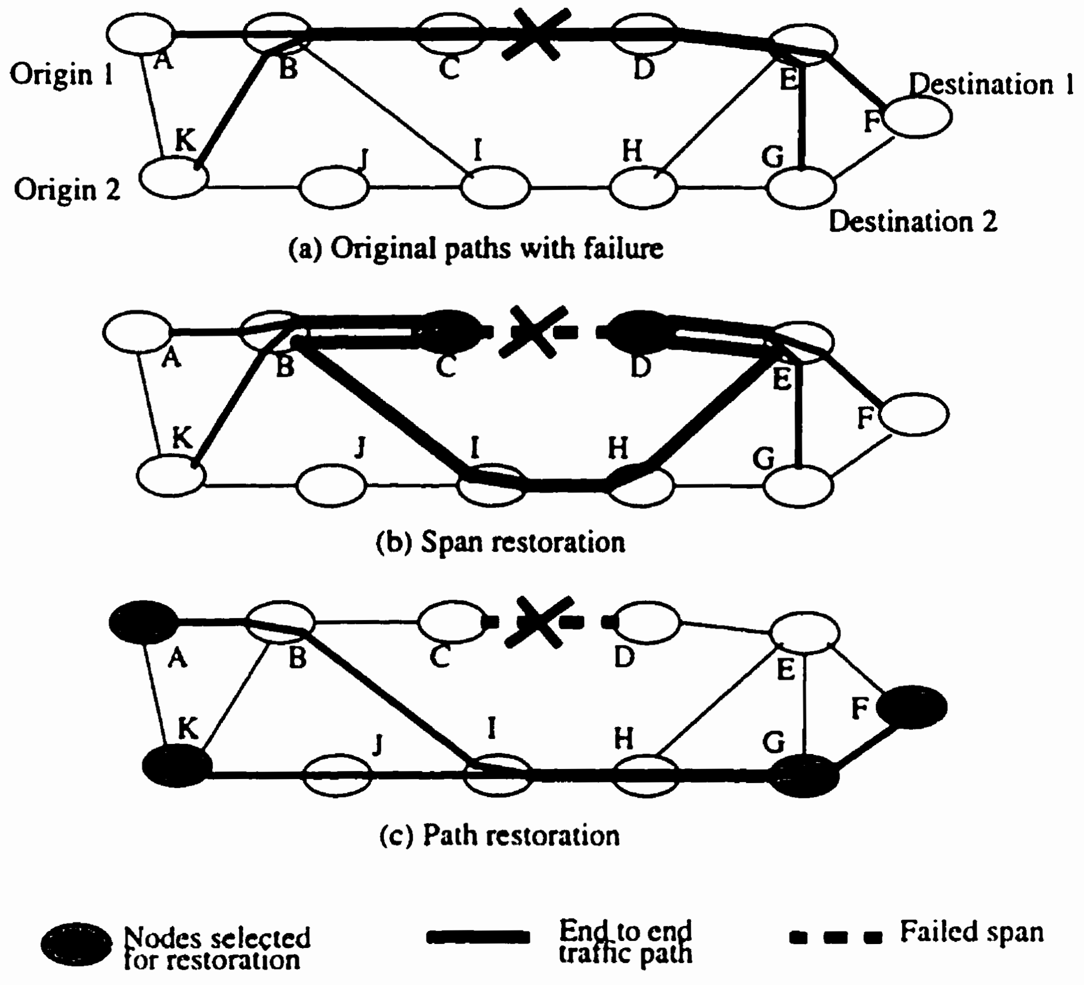


FIGURE 1. Span and Path Restoration

of replacement paths between the two end nodes of a failed span. *Path restoration* re-routes failed working traffic over a set of replacement paths between each source and destination affected by a failure. Figure 1 shows an example in which the failure of span CD affects two working routes: A to F and K to G. Span restoration finds replacement path segments between nodes C and D, whereas path restoration finds end-to-end replacement paths for the demand pairs A-F and K-G. Path restoration can be more capacity efficient than span restoration because it spreads the replacement paths over a larger portion of the network, increasing the alternatives available for making efficient use of network's spare capacity. However, path restoration is more complex to implement than span restoration because it may involve finding replacement paths for several source-destination pairs instead of only one node pair.

1.3 ATM Technology

Asynchronous Transfer Mode (ATM) is the transport technology of the B-ISDN network proposed by the International Telecommunications Union (ITU). ATM defines the switching, multiplexing and transmitting of information over B-ISDN networks. Instead of reserving time slots as in STM networks, information in ATM is packetized and placed in short 53-byte cells that are multiplexed and transmitted asynchronously on the transmission medium. ATM networks can support a variety of services (e.g., telephone, image, video and data), with a guaranteed *Quality of Service* (QoS).

A *Virtual Circuit Connection* (VCC) in ATM is analogous to a virtual circuit in data networks, such as an X.25 or a frame relay logical connection. The VCC is the basic unit of switching in ATM networks. After a VCC is set up between two nodes in the network, a variable rate, full-duplex stream of cells maintains a connection. For ATM, a second sub-layer of processing has been introduced to deal with the concept of a virtual path. A *Virtual Path Connection* (VPC) is a bundle of VCCs that have the same endpoints, e.g., switching systems, LAN gateways, etc. Thus, cells flowing over all of the VCCs in a single VPC may be switched together. Because VP switching is inherently more efficient than VC switching [4], it is advantageous to switch a cell in a VP. This Virtual Path concept was developed in response to a trend in high-speed networking in which the control cost of the network is becoming increasingly high in proportion to the overall network costs. The Virtual Path technique helps reduce the control cost by grouping connections sharing common paths through the network into a single unit. Network management actions can then be applied to a small number of groups of connections instead of a large number of individual connections.

The packets in an ATM network are called *cells.* The length of an ATM cell is 53 bytes, consisting of a 48-byte information field and a 5-byte header. Two of the fields defined in the header are the *Virtual Path Identifier* (VPI) and the *Virtual Circuit Identifier* (VCI). VPI is a 12-bit field (8-bit in UNI), and the VCI is a 16-bit field that together define the routing information of a cell. As with any other packet-switching network, routing of cells is performed at every node for each arriving cell.

A difference between STM and ATM networks is that ATM uses *statistical multiplexing*. Statistical multiplexing is a scheme that multiplexes traffic based on the strong law of large numbers [5]. This law states that for a number of uncorrelated flows, the bandwidth necessary to satisfy the needs for all of the flows stays nearly constant, even though the amount of traffic in individual flows can vary. The reason for this is that at any given moment a few applications could be increasing their traffic while other applications could be reducing their traffic. According to the strong law of large numbers, these changes roughly balance each other out. Compared with STM deterministic multiplexing, ATM statistical multiplexing can offer an improvement in bandwidth utilization [5].

1.4 ATM Restoration

Recently algorithms for ATM network restoration have been studied actively and the VP has become the focus of ATM restoration [4, 9, 10]. When a VP is restored, all VCs inside the VP are restored automatically. There is no need to restore individual VCs. Because a VP can support up to 65536 VCs (recall that there are 16 bits in VCI field), restoration at the VP level can greatly reduce the network management burden.

Another unique factor of VP-based restoration is that it is basically a path restoration technique instead of a span restoration technique shown in Figure 1 because a VP normally traverses several spans. As discussed in Section 1.2, path restoration has a significant advantage in terms of capacity efficiency compared with span restoration.

VP-based restoration is distinct from STM path restoration because the path establishment and bandwidth assignment of a VP are defined relatively independently. The route is defined in the VPI table of the ATM switch, while bandwidth is logically defined and managed in the database of an ATM switch. In fact a VP route can be established without defining its bandwidth, i.e., a zero bandwidth VP may be established. This unique property makes it possible to establish multiple backup VPs that share the spare capacity of a span. In Figure 2, for example, two backup VPs *i* and *j* are established for restoration using span *ab*. The bandwidth requirement of each backup VP is 5 units. The routes of backup VPs are set before a failure. In the case of a failure, bandwidth is allocated to the needed backup VPs. If VP *i* fails, the backup VP for VP *i* uses the spare capacity on to restore the traffic in VP *i*. If VP *i* and VP *j* are mutually independent, we only need to allocate 5 units of spare capacity to span *ab* instead of 10 units. The sharing of spare capacity on a span allows spare capacity to be more efficiently utilized [4].

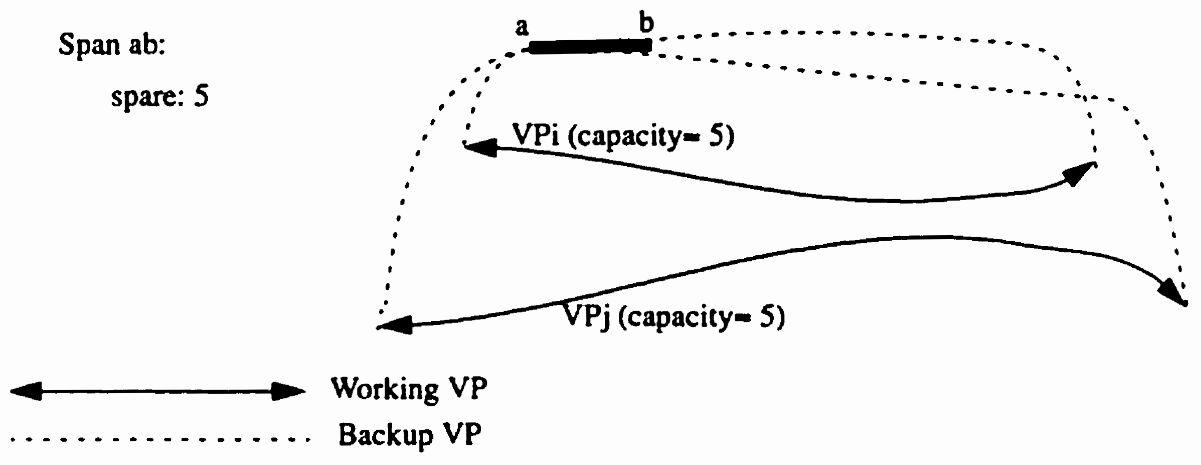


FIGURE 2. Spare Capacity Sharing in ATM networks

Compared with STM restoration, ATM restoration is more difficult due to the following reasons [3]. First, the average number of virtual paths accommodated in a link may be much larger. The maximum number of VPs that can be normally handled by an ATM switch is 4096 (12 bits in the VCI field in the ATM cell header for at a NNI). Thus the number of restoration paths may be tremendously larger than that in STM networks. This also makes a distributed restoration scheme difficult to implement. By any type of flooding method, the network may easily become saturated with route searching messages. The second difficulty comes from ATM's traffic characteristics. ATM networks support a variety of traffic types such as CBR, VBR, ABR and UBR. These traffic types have their own quality of service requirements. For example, real time applications are very sensitive to cell delay variance, while data communications are more sensitive to cell loss. Consequently, ATM restoration must consider more factors than only bandwidth requirement as in STM restoration. The third difficulty is that ATM uses statistical multiplexing. In contrast to the traffic flow in STM networks, there is no fixed bandwidth associated with traffic flows in ATM networks. Therefore, if some traffic flow is induced on a link, without some engineered control and network design, the result could be a severe overload on that link, an issue overlooked by some ATM restoration schemes.

In a previously proposed ATM restoration scheme [4], a backup VP is pre-assigned for each working VP. In the restoration process, traffic in a working VP is normally restored to only one backup VP. This no-splitable property can prevent VCs inside the VP from re-routing to different routes, which increases the complexity of restoration significantly. In the backup VP, there is no cell flow in normal operations. The nodes terminating the working VP are also the nodes terminating the backup VP. When a failure occurs, the *Sender* node (downstream side terminating node of the failed VP) detects the VP failure and sends a restoration message along the backup route. The Sender node then switches the failed VP to the backup VP. Each node that receives the restoration message captures the appropriate bandwidth on the links, and retransmits the message to the next node on the backup route. When the *Chooser* node (upstream side terminating node of the failed VP) finally receives the restoration message, it switches traffic from the failed VP to the backup VP. This completes, at least functionally, the restoration process for the failed VP. There are, however, issues of bandwidth coordination to make this simple scheme perform adequately.

1.5 Problem Introduction

Several spare capacity placement algorithms for ATM VP-based restoration design have been proposed [4, 10]. They may be classified into two general categories. The first category of ATM capacity placement algorithms are basically heuristic in nature. In the algorithm proposed by NTT [4], the shortest route is first set as the initial backup route for each working VP. Then the algorithm substitutes an alternative backup route for each working VP one at a time and the spare capacity is calculated with this modified set of backup routes. If a smaller total amount of spare capacity is achieved using this substitution, the alternate backup VP is kept. Every VP backup is tested in this manner to find which backup VP routes require less spare capacity given the current state of spare bandwidth allocations already placed for previously decided backup VP routes. This process is repeated until no improvement can be made. In the resultant design, the spare capacity of a span is forced by the largest capacity VP whose backup route traverses it. This procedure is conceptually heuristic.

The second category of ATM capacity placement algorithms [6], treats the ATM capacity design problem with exact methods but in a manner that is essentially the same as for STM path restorable networks [8]. This means that the spare capacity plan aims to support all restoration demands with an exact match of restoration bandwidth to failed working VP bandwidth. This approach is certainly a valid and defensible basis for planning a practical ATM network today. However, one can observe that this treats the ATM spare capacity problem as essentially equivalent to STM planning in that failed VPs are rerouted over backup VPs of exactly equal bandwidth allocation regardless of actual VP utilization. This is analogous to STM type restoration of STS-n signals as integral entities regardless of their actual payload fill. There is no way to take signal fill into account in STM restoration: each signal unit must be replaced exactly or all services borne on the affected transport signals experience hard outage. This hard outage aspect of STM does not pertain to ATM because ATM uses statistical multiplexing, however, two or more VPs of a unit bandwidth allocation could technically be re-routed for restoration and converge on the same link of unit spare bandwidth. Both VPs are functionally or logically rerouted as required, but the link bandwidth is oversubscribed at this point. Thus, there may be an overload effect in the link, i.e., the services in both VPs may undergo a degradation on QoS. This degradation, though conceivably severe, but, unlike STM, is soft and continuous, a degradation that arises if the replacement bandwidth is not an exact match to the failed working bandwidth. Moreover, the actual degradation that occurs depends on the VP utilization at failure time. If utilizations are low, then the oversubscription of bandwidth on restoration may not cause QoS to degrade below acceptable service levels.

ATM restoration planning could (if we wish to consider it) exploit a domain that is not available to STM. This planning would allow us to contemplate bandwidth planning that does not support strictly perfect replacement of each VP's initial bandwidth allocation. While not dismissing or minimizing the potential impact on service, which could be severe if oversubscription effects are uncontrolled, it could be of value to at least inspect the trade-off between network capacity requirements in dependence on a limited designed-in allowance for bandwidth oversubscription upon restoration. Specifically, our interest will be in recognizing the inherently statistical nature of the traffic flows in ATM and formulating the backup VP design process to permit a controlled maximum amount of convergent flow oversubscription on spans during restoration. A partial analogy for this line of thinking is found in the airline business: most flights are slightly overbooked as part of an overall optimum economic policy for revenue maximization. Most often, the overbooking is unseen to users as some passengers almost always do not show up. Similarly in an ATM network, could we not slightly (or even aggressively) overbook the restoration capacity we design into the network? Unless a failure occurs right when working VP utilizations are simultaneously at their peaks, the slight overbooking of restoration capacity may be unnoticed by customers. Indeed, if the trade-off of net capacity versus tolerable oversubscription is steep, and/or if mechanisms can be built in to also prioritize VPs when restoration-induced congestion is manifest, then ATM networks with a controlled degree designed-in bandwidth oversubscription upon restoration may well be part of an economically optimum overall strategy.

1.6 Outline of Thesis

The preceding discussions introduced the general problem of this research, we now proceed in detail as follows: Chapter 2 presents different types of original integer program formulations and heuristic algorithms for the capacity placement problem in ATM networks. First, we will quantitatively define the oversubscription factor based on the traffic nature in backup VP-based ATM restoration. Then, we will show that the oversubscription effect can be very severe in some prior work, illustrating that the spare capacity placement problem must be considered carefully. Three Integer Program (IP) formulations will be given for (a) minimum spare capacity with respect to design peak oversubscription, (b) minimum oversubscription with given spare capacity, and (c) minimum total capacity with respect to design peak oversubscription. In addition to these

IP formulations, two simpler algorithms are also presented to calculate reasonably tight upper and lower bounds on the required spare capacity. We will then use these formulations to study the effect of spare capacity saving with various degrees of oversubscription allowed in ATM restorable network planning. From the results we will find *oversubscription* can benefit the capacity saving in ATM networks.

Chapter 3, thereafter, presents a study to address the logical next question of what the tolerable overload factor can be based on the related cell-level overload implications. We study cell-level overload from the viewpoint of cell loss ratio degradation. The idea is to determine a realistic level of cell-level overload effects and cell-level performance degradation due to merging restoration flows. This work is oriented toward the cell level dependence on traffic types and number of VPs in merging restoration flows. Our aim is to produce quantitative guidelines on the tolerable oversubscription value with which to design a given backup VP-based restorable ATM network. Two different traffic models will be used in the respective simulations: an on/off fluid model and an auto-regressive model.

Chapter 4 is a concluding discussion which proposes and discusses a new overall framework for ATM backup VP capacity design based on the ideas and results of the previous two chapters. Finally a summary of the whole thesis will be given.

Chapter 2. Spare Capacity Placement

2.1 Logical View of VP-based ATM Restoration

To understand the traffic on an ATM span after restoration, a logical view of a span j is illustrated in Figure 3. Assume that span *j* has a total installed bandwidth allocation that is based on its nominal working load and some reservation of spare capacity for restoration. Unlike in an STM network, these working and spare bandwidth allocations are not necessarily distinct integral transmission sub-units, e.g. DS3s or STS-1s. Rather, each link's total bandwidth is viewed as having been planned as two allocations from the total bandwidth present. In case of another span *i* failure, all VPs going through span *i* are re-routed to other spans. During restoration, some working VPs on the failed span *i* may use span *j* in their backup routes. These backup VPs were logically present on span *j* prior to failure, but then consumed no bandwidth. Only upon failure does the re-directed cell

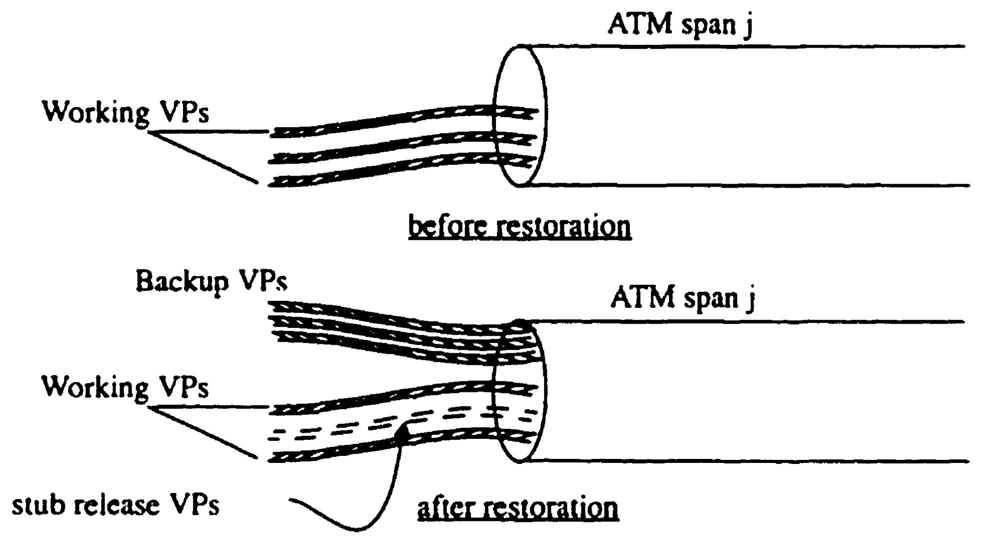


FIGURE 3. Logical View of VP-based ATM Restoration

stream appear in each backup VP. Additionally, part of the restoration reaction of the network may give span *j* a ***reduction*** in cell traffic: this occurs if one or more of the working VPs on span *j* is also affected by the failure event, either upstream or downstream of span *j* on the path of these working VPs. This is known as stub release here and was previously used in a similar content in [8]. Therefore, surviving span *j* may see both a disappearance of cell flows from some of its working VPs and a sudden onset of new cell streams for activated backup VPs that traverse it.

In general, there are three types of traffic on an ATM span after restoration:

1. traffic from working VPs, the undisturbed traffic already existing on the span,
2. traffic from stub release VPs, the traffic rerouted away from the span, and
3. traffic from backup VPs, the traffic choosing the span in their backup routes.

2.2 Oversubscription Factor

We can now quantitatively define the restoration induced bandwidth oversubscription factor. The oversubscription factor $X_{j,i}$ of ***a span j in response to failure of another span i*** is defined as the ratio of total VP bandwidth allocation after restoration to the total installed capacity of the span. This can be expressed as follows:

$$X_{j,i} \equiv \frac{W_j - Rs_{j,i} + Rr_{j,i}}{W_j + S_j} \tag{1}$$

where

> $Rr_{j,i}$ is the total allocated bandwidth of VPs on span i whose backup route crosses span j.
>
> $Rs_{j,i}$ is the total allocated bandwidth of VPs which disappear from span j because they traverse the failed span i which happens to be either upstream or downstream of span j. (This is called the ***stub release traffic.***)
>
> W_j is the total allocated bandwidth of working VPs on span j before failure.
>
> S_j is the total spare bandwidth allocation on span j.

Note that $X_{j,i}$ is based on *allocated* bandwidths of VPs throughout, but not the actual traffic. The term "overload" can more precisely describe the ratio of actual induced traffic to the link bandwidth. The actual cell-level overload that occurs depends on the actual utilization of each VP involved, not the bandwidth allocations to the VPs. Therefore, if each factor in the numerator were to be multiplied by a known cell-level utilization factor, a true overload measure results. However, for planning purposes, the worst case overload is obviously the same as the oversubscription factor, thus we continue to refer to $X_{j,i}$ as the restoration-induced oversubscription factor since a value X_{tol} will represent the designed in maximum oversubscription of bandwidth and hence the maximum tolerable cell-level overload that could occur in the network as designed.

It can be appreciated that $X_{j,i} \leq 1.0 \quad \forall (j, i)$ is a basic property of STM restoration because this implies that the total bandwidth of paths available for replacement of failed transport signals is always equal to or greater than the failed bandwidth. In STM, there is no option of 'partly' replacing one or more failed STS-n signals. Either each is replaced exactly by a matching restoration path or all services borne by the given STS-n experience immediate total outage. In ATM, however, the concept $X_{j,i} > 1.0$ is definitely conceivable and technically meaningful. As argued, it simply means that span *j*'s total bandwidth is technically oversubscribed when span *i* fails. Unlike STM, this is a state of 'partial' restoration in which all services may be affected to a degree in terms of cell loss and delay. However, no service is immediately terminated or disconnected because the restoration path bandwidths do not exactly match the pre-failure bandwidths. Whether cell-level performance exceeds QoS requirements under $X_{j,i} > 1.0$ will depend on the actual VP utilizations and traffic parameters at the time of failure.

2.3 Prior Work Involving Uncontrolled Oversubscription Effects

A heuristic algorithm of capacity placement in ATM backup VP-based restoration was previously proposed by NTT in [4]. The main problem of this method, described below, is that while every working VP is assigned a backup VP route to yield a near-minimum in total backup capacity allocations, there is no designed-in control to coordinate the backup VPs with respect to oversubscription arising from the set of working VPs that are cut by the same physical failures. The result is that while a logical replacement route exists to functionally replace each failed working VP, the total cell-level traffic impinging on other network spans is uncontrolled.

In the algorithm described in [4], hereafter called *KST-Alg*, the shortest route is first set as the initial backup route for each working VP. Then the algorithm substitutes an alternative backup route for one working VP. The spare capacity is calculated with this set of backup routes. If a smaller total amount of spare capacity is achieved by using this substitution, the substitution is kept. Every VP is tested in this manner to find which of all its possible disjoint alternate routes requires less spare capacity given the current state of spare bandwidth allocations already placed for previously decided backup VP routes. This process is repeated until no improvement can be made. In the resultant design, the spare capacity of a span is forced by the largest VP whose backup route traverses it.

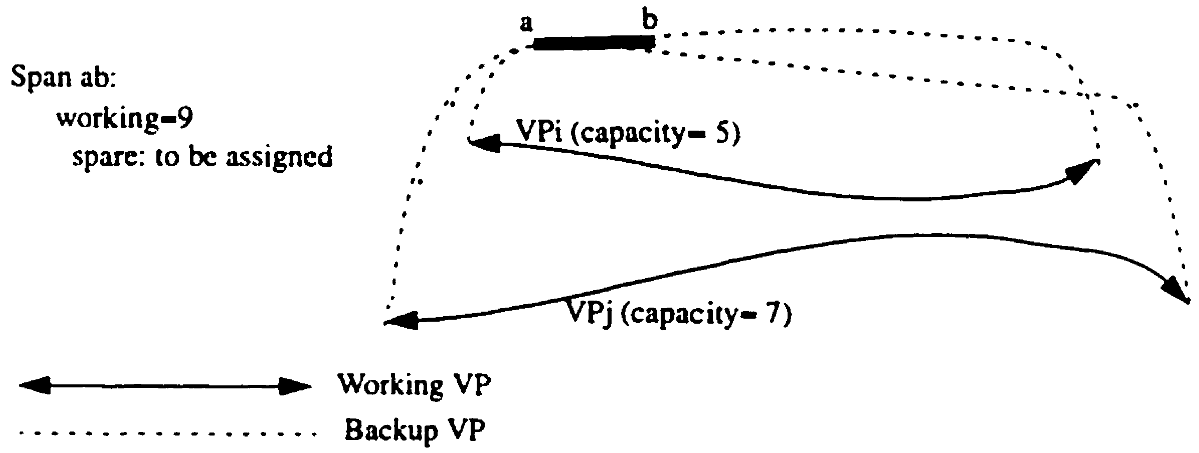


FIGURE 4. ***KST-Alg*** **Backup VP Capacity Allocation (Ideal Situation)**

Figure 4 illustrates the capacity minimization principle and how it results in uncontrolled oversubscription. In the example, span *ab* serves on the backup routes for both VP *i* (capacity 5) and VP *j* (capacity 7). Assume *KST-Alg* has first considered VP *j* and, accordingly, then assigns span *ab* a spare capacity of 7 units. Once span *ab* has 7 units of restoration capacity assigned to it, *KST-Alg* will later realize that it can efficiently route the backup VP for working VP *i* over span *ab* as well because more than enough capacity is already reserved on *ab* to serve VP *i* which needs only 5 units of bandwidth. This reuse of span *ab* in the example assumes that *KST-Alg* also finds that the rest of the backup VP route for VP *i* is suitably efficient on other spans as well. *KST-Alg* chooses a complete set of backup VPs which are efficient in this sense of re-use of capacity.

Thus, functionally speaking, a logical backup VP is planned for each working VP. Such backup VPs would be fully adequate if one VP fails at a time. What is missing, however, is consideration that if VP *i* and VP *j* happen to share the same physical span, for instance, *xy*, then in case of its failure, VP *i* and VP *j* will be re-routed simultaneously onto backup VPs which traverse span *ab* as illustrated in Figure 5. Therefore, omitting any 'stub

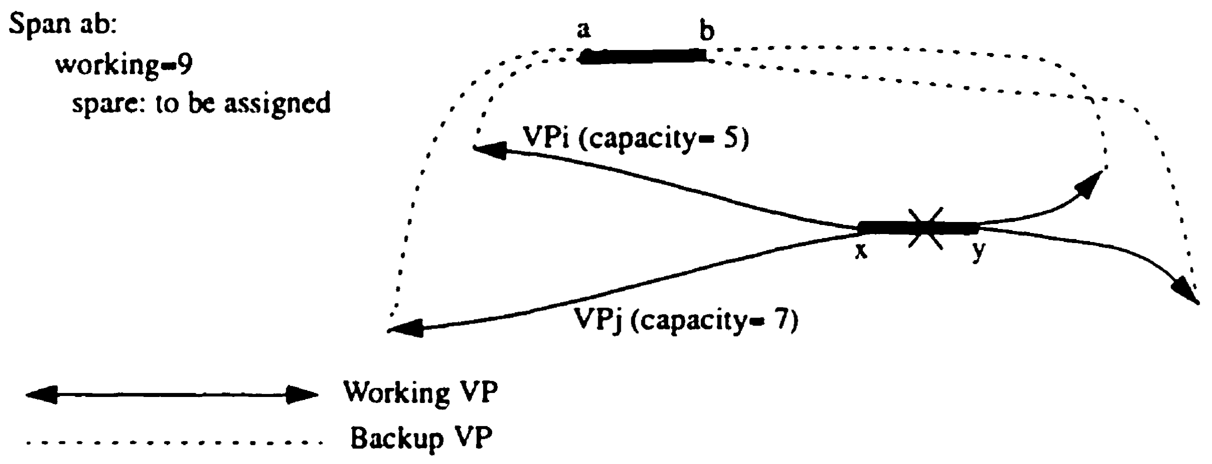


FIGURE 5. *KST-Alg* Backup VP Capacity Allocation with Problems

release' effects for the example, and assuming a W_j of 9 units on *ab*, the result of span cut *xy* is a restoration-induced total allocation of 9 +5+7 = 21 units of capacity on span *ab*, which only has a total capacity allocation of 9+7 = 16 units. Thus the restoration induced oversubscription factor $X_{ab_{xy}}$ is 1.31 = 21/16. What is missing, then, are considerations on how to coordinate the set of backup VPs from each physical span failure as a simultaneously instantiated group of backup VPs. The reported capacity results given by KST-Alg are extremely attractive and widely publicized predictions of very low spare capacity levels. In what follows, we will see that these low levels will be accompanied by essentially uncontrolled restoration oversubscription on the surviving spans.

2.4 Implementation of *KST-Alg*

In this section, *KST-Alg* was implemented to reproduce and to test the predictions of very low spare capacity and to validate our concerns of uncontrolled restoration induced oversubscription effects. The programs are written in the C language and presented in Appendix C. In the rest of this section, we first discuss the working VP routing method, and then move on to explain how to use *KST-Alg* to design the backup VP route for each working VP. Next, the oversubscription factor is calculated. Finally, we give a validation to our calculation.

2.4.1 Working VP Routing for *KST-Alg*

Given a network topology and working demands, we use the shortest path method to design the working VP routes and working capacity on each span. For every demand, we find the shortest path between the end node pair, set up a VP along the path and place the demand on all traversing spans. If there are several 'equal' distance shortest paths, we set up one VP on each, and split the demand into these VPs equally. Summing all VPs requirements on a span, we get the working capacity.

2.4.2 Backup VP Routing Assignment for *KST-Alg*

First, a subset of all distinct backup routes for every VP is found, i.e., a subset of all span disjoint routes between the end nodes of the working route. Considering that the number of distinct routes in a network of S spans is $O(2^S)$, the number we use to test the result typically has to be restricted in practice. Because of the properties of different networks, we find that hop-limited or distance-limited methods typically used are not realistic. For example, with a given hop limit, there may be thousands of backup routes for some VPs, while no routes can be found for other VPs in some networks. Consequently, a k-successive shortest distinct routes method is adopted to generate a useful set of distinct backup route options for each VP. We select several shortest eligible routes from this subset in the results presented subsequently.

It is interesting to note that in some situations, there are no backup route options at all. For example, in the sample network in Figure 6, if a VP (the dotted line) terminates at nodes A and B, then there are no span disjoint backup routes between them even though this network is 2 or more connected throughout. To avoid this situation, we change the working VP design as needed if such a situation is encountered. A VP with the shortest

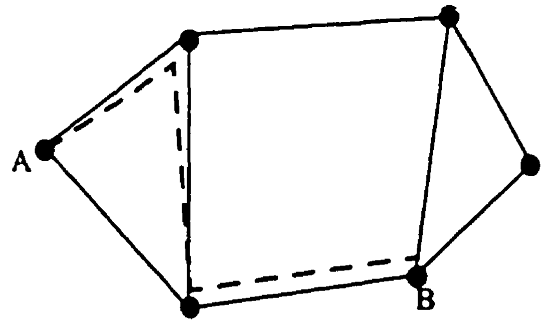


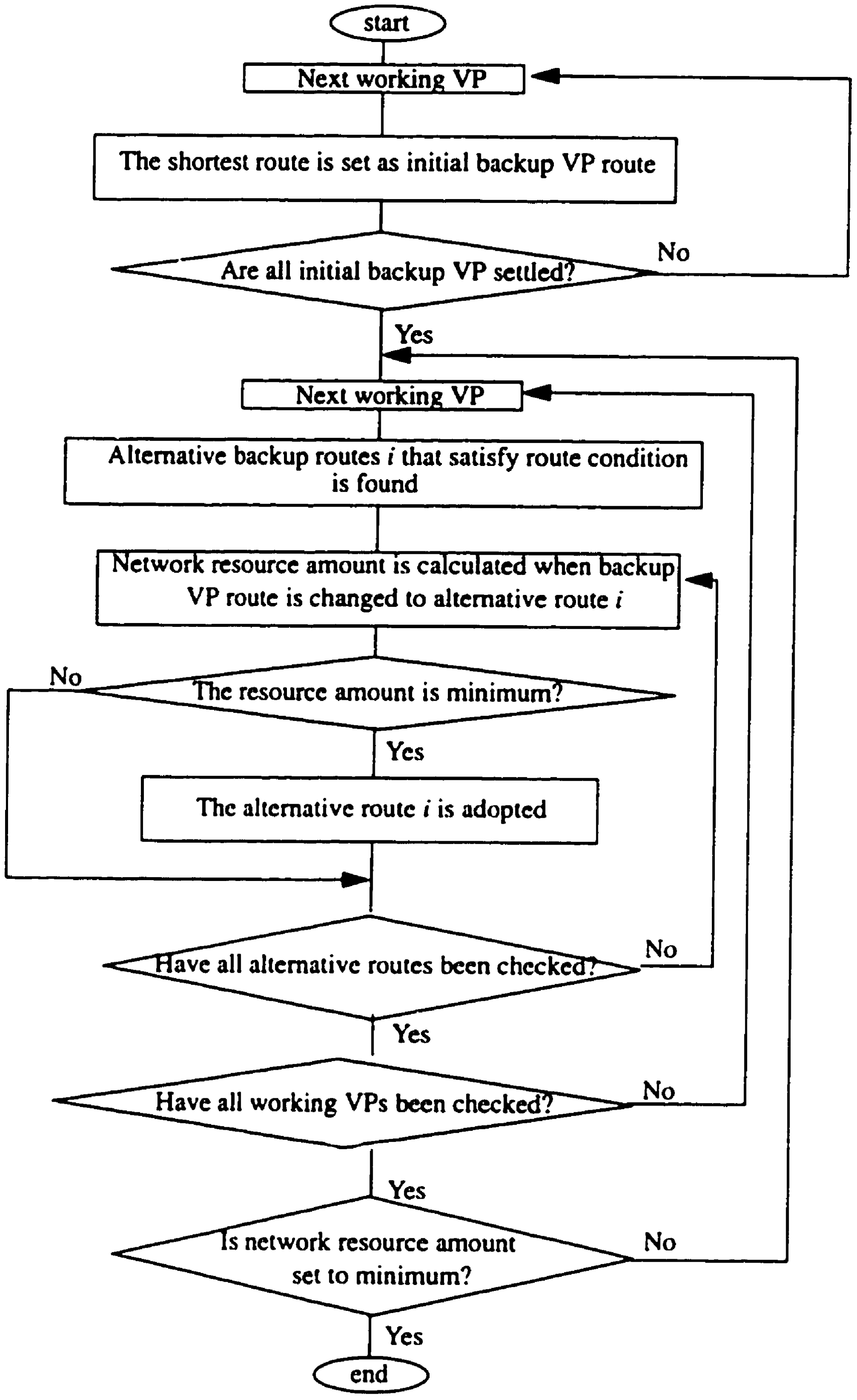
FIGURE 6. No Backup Routes in Working VP Design

path is checked to see if it has any backup path. If not, we choose another route for this VP which, although it is not the shortest path, has a span disjoint backup VP. In this case working & backup VPs are always possible to find as long as a cycle exists in the network graph between their end nodes.

With these preliminaries completed, *KST-Alg* is then applied to choose the best backup route set to minimize the total spare capacity. In the iteration for one VP, we substitute its backup route with all alternatives. The spare capacity of a span is forced by the largest VP whose backup route traverses the span. If one route set leads to less total spare capacity, this substitution is kept. This process is repeated until no improvement can be made by substituting backup routes for all VPs. A block diagram of KST-Alg implementation is illustrated in Figure 7 on page 22.

2.4.3 Calculation of Oversubscription Factors

After designing backup VPs with *KST-Alg*, we check the oversubscription factors on all other spans for each possible physical span cut. Oversubscription data is obtained by actual rerouting experiments on each designed network based on all possible span cuts. Each affected working VP is rerouted to its designed backup VP. We then apply all traffic (including working, stub release and backup) in the definition in Equation 1 to obtain the restoration-induced oversubscription on all spans for each span cut. For a network containing S spans, each of *S-1* other spans has an oversubscription factor upon a span failure. Therefore, the total data set of oversubscription factors is an *(S-1)* by S matrix.



start
Next working VP
The shortest route is set as initial backup VP route
Are all initial backup VP settled?
No
Yes
Next working VP
Alternative backup routes *i* that satisfy route condition is found
Network resource amount is calculated when backup VP route is changed to alternative route *i*
The resource amount is minimum?
No
Yes
The alternative route *i* is adopted
Have all alternative routes been checked?
No
Yes
Have all working VPs been checked?
No
Yes
Is network resource amount set to minimum?
No
Yes
end

FIGURE 7. ***KST-Alg*** **Implementation Block Diagram**

2.4.4 Validation of Implementation

The implementation of *KST-Alg* was verified by checking some program results with manual calculations.

In the backup design, the spare capacity of a span is forced by the largest capacity VP whose backup route traverses the span. In the process, we save chosen backup routes into a file. Later, we check the file and compare it with our manual calculation. For one span, we find all backup routes which use it, then verify if the spare capacity is equal to the largest backup VP. One of the five sample networks introduced in Section 2.5.1 on page 25, Net 3, was chosen to validate the results. For example, in Net 3, span 8 is picked. We find the following VP's which use this span. Their capacities are shown in Table 1:

TABLE 1. Backup VP Design Validation for Span 8 of Net 3

VP	capacity	VP	capacity	VP	capacity	VP	capacity
19	30.0	55	17.0	72	37.0	75	4.0
76	28.0	*78*	*39.0*	79	5.0	81	4.0
84	7.0	85	10.0	87	26.0	89	4.0
90	30.0	91	6.0				

Apparently, the largest VP in Table 1 is VP 78 and its capacity is 39.0. This is exactly the spare capacity which we get from the program. Other cases were also used to validate the program and we always ended with the same results.

In the oversubscription factor validation, to calculate the oversubscription on span j upon a span i cut, we considered the following VPs which use:

(a) span i as working route and span j as backup route;

(b) both span i and j as working route;

(c) span j as working route, regardless of span i.

Then these items are put in the oversubscription factor definition.

Item (a) is restoration traffic in Equation 1, item (b) is stub release traffic and item (c) is original working traffic on span j. After determining (a), (b), and (c), we put them in Equation 1 to calculate the overload factor. For example, in Net 3, the oversubscription factor of span 6 upon span 7 cut is considered. VP 37 (whose capacity is 4.0) traverses span 6 and its backup route traverses span 7. Thus, span 7 has 4.0 extra units of traffic after restoration. Table 2 summarizes all VPs which use span 7 as backup route and span 6 as working route.

TABLE 2. Oversubscription Factor Validation

VP	capacity	VP	capacity	VP	capacity	VP	capacity
37	4.0	38	6.0	46	441.0	72	37.0
75	4.0	76	28.0	77	95.0	78	39.0
81	4.0	82	82.0	84	7.0	84	7.0
85	10.0	87	26.0	89	4.0	90	30.0
91	6.0						

The sum of all traffic is 828. Next we find stub release traffic. In this case, only VP 55 (whose capacity is 17) uses both span 6 and 7 in its working route. It uses span 8 as backup. Thus, this traffic is taken from span 7 upon failing span 6. Span 7 has a working traffic 684. Its working and spare capacity are 684 and 441 respectively. Given these values we can get an oversubscription factor using Equation 1 of

$\frac{684 - 17 + 828}{684 + 441} = 1.3288$, which is what we get from the program.

2.5 Results and Analysis of KST-Alg

2.5.1 Network Investigated

Five networks and demand matrices previously studied for STM restoration [8] were used to test *KST-Alg*. The characteristics are detailed in Table 3. *Smallnet* is a test network which has a uniform point-to-point demand matrix with two demand units between all node pairs. Net 1 is a U.S. metropolitan area model (also known as the "Bellcore" study network). Net 2 is a metropolitan area model in the Telus system, Canada. Net 3 and 4 topologies and demands are based on European and US interexchange networks, respectively.

TABLE 3. Test Network Characteristics

network	# node	# span	# demand pairs	# Working VPs
Smallnet	10	22	45	79
Net1	15	28	67	68
Net2	20	31	153	153
Net3	30	59	263	271
Net4	53	79	347	418

2.5.2 Results

Table 4 shows the spare capacity requirement with *KST-Alg* and the data for the consequent oversubscription effects. The *spare capacity* expressed in percentage represents the ratio of distance-weighted spare capacity to the distance-weighted working capacity. Average oversubscription is the mean oversubscription value of all cases where oversubscription factors are greater than or equal to 1.0. Maximum oversubscription is the case with the largest oversubscription factor.

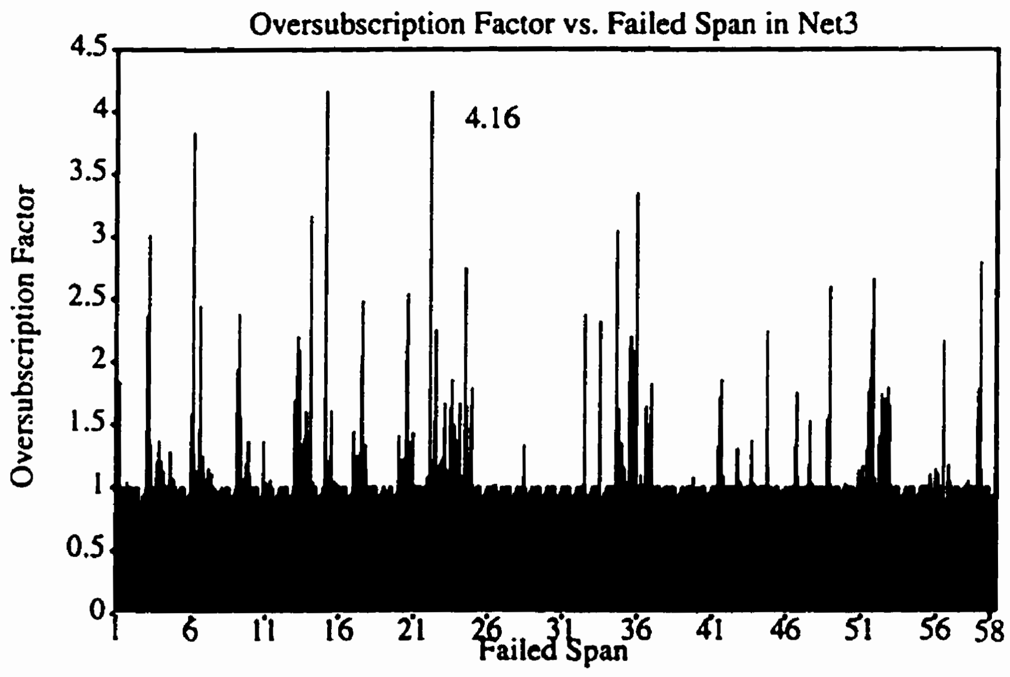
TABLE 4. Spare Capacity and Oversubscription in Designs with *KST-Alg*

Network	KST-Alg	Average Oversubscription	Maximum Oversubscription
Smallnet	28.6%	1.14	1.42
Net1	51.7%	1.39	3.00
Net2	54.3%	1.37	3.32
Net3	31.2%	1.37	4.16
Net4	38.6%	1.46	10.00

The table shows that *KST-Alg* generates a low redundancy. Smallnet has about 29% sparing. In metropolitan networks, sparing is up to 55%, while in long haul networks, sparing is only up to 39%.

These spare capacity predictions are indeed much lower than those required by STM networks [8]. This finding has gained much industry attention contributing to a general notion that ATM-based restoration will require very much less capacity than STM-based restoration. It is important, however, to note that these particularly low spare capacity levels are accompanied by significant and strictly uncontrolled oversubscription effects on surviving spans. With the levels of oversubscription reaching 3 to 10 times nominal traffic load, cell loss and cell delay in ATM networks would very likely be intolerable for many applications [6].

Figure 8 illustrates the detailed oversubscription factor analysis of Net3. The upper diagram shows the oversubscription overall surviving spans when a span is cut. Its structure is as follows: for each span considered as the failure span *i*, on the x-axis, the *(S-*



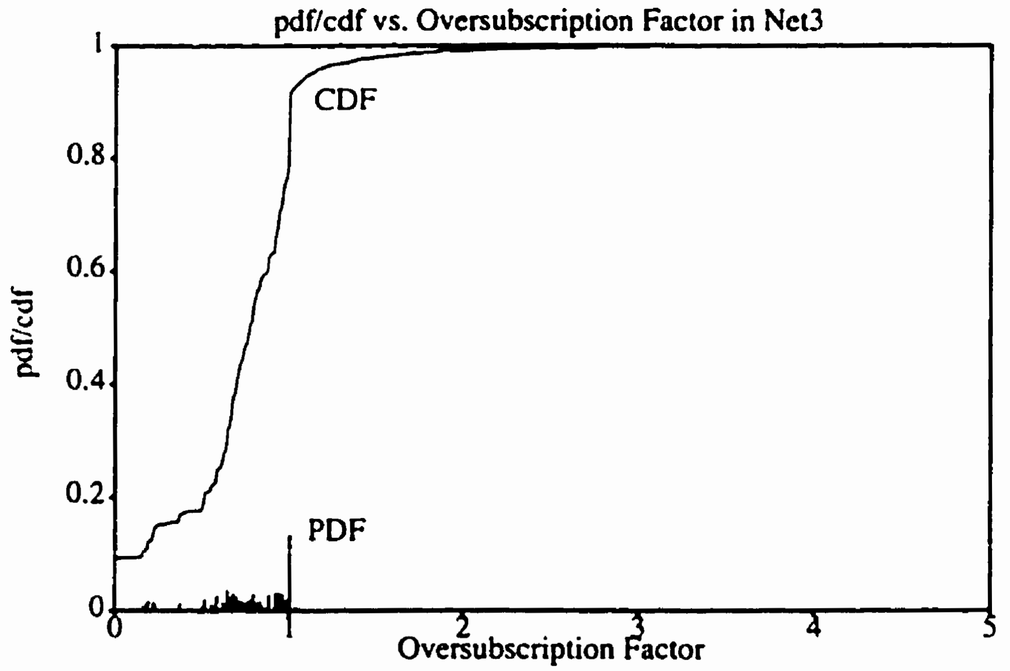


FIGURE 8. Oversubscription Factor Analysis of Net3

1) $X_{j,i}$ values experienced by other spans are plotted left to right with a vertical line for each value. Therefore, in the figure, the x-axis is labelled with the failure span names and in the fine scale the oversubscription factors of all the surviving spans are shown. For example, in a network of 10 spans, there would be ten clusters of nine $X_{j,i}$ values displayed side by side to form the plot. The lower plot shows the probability density and cumulative probability density functions of oversubscription values. These two diagrams demonstrate that a large number of oversubscription cases is involved in *KST-Alg* design. The largest oversubscription is about 4.16 in Net3.

2.6 IP-1: Minimum Spare Capacity with Design Limits on Oversubscription

The capacity savings implication in *KST-Alg* relative to STM networks is very attractive, but the uncontrolled oversubscription implications are probably unacceptable in practice. Our aim now is to formulate optimal capacity allocation methods that will still gain as much ATM-related capacity savings as may be safely possible given an allowed maximum of restoration-induced oversubscription. In Appendix C, the C-language programs, were which written to generate the Integer Program tableaus, are listed.

The first IP formulation optimizes the spare capacity placement of a restorable ATM network given a peak allowable oversubscription factor in the network. The objective function is:

$$\text{Minimize:} \quad \left\{ \sum_{j=1}^{S} c_j \cdot s_j \right\} \qquad (2)$$

Subject to:

1. Sparing is sufficient to keep restoration oversubscription below the design limit, X_{tol}, for all failures:

$$(X_{j,i} \leq X_{tol}) \quad \forall (i,j) \in S \ \ (i \neq j) \tag{3}$$

2. Backup VPs are sufficient to meet the target restoration levels $(R^{r,q})$ for all working VPs:

$$\left|f_k^{r,q}\right| = \left|g^{r,q}\right| \alpha_k^{r,q} \cdot R^{r,q} \qquad \forall k \in P^{r,q} \ \ \forall (r,q) \tag{4}$$

3. Only one backup VP can be used for each working VP, i.e. VP flows are not split:

$$\sum_{k \in P^{r,q}} \alpha_k^{r,q} = 1 \qquad \forall (r,q) \tag{5}$$

where $\alpha_k^{r,q} = 1$ if the k^{th} route for backup of $g^{r,q}$ is chosen, otherwise $\alpha_k^{r,q} = 0$.

The traffic in a working VP can not be split in restoration, i.e. only one backup route to restore a working VP. This requirement is the general industrial practice because splitting a VP entails a rearrangement of all individual VCs inside the VP and can be potentially very complicated.

The definition of variables is as follows:

C_j = the cost of span j per unit bandwidth (the length of a span may be included here)

W_j = the working capacity bandwidth allocation on span j

S_j = the spare capacity bandwidth allocation on span j

S – the set of all spans in the network

$g^{r,q}$ – the working VP on route q for demand pair r

$\beta^{r,q}$ – the portion of demand traffic going through working VP $g^{r,q}$

$P^{r,q}$ – the set of all distinct backup VP routes eligible for restoration of working VP $g^{r,q}$

$f_k^{r,q}$ – the k^{th} backup VP of the working VP $g^{r,q}$

$\alpha^{r,q}$ – the portion of restoration traffic going through backup VP $f^{r,q}$

$|g^{r,q}|$ – the bandwidth of the working VP $g^{r,q}$

$|f^{r,q}|$ – the bandwidth of the backup VP $f^{r,q}$

$\zeta_i^{r,q}$ – 1 if the route of working VP $g^{r,q}$ crosses span i, otherwise 0

$\delta_{k,j}^{r,q}$ – 1 if the k^{th} route available for backup of $g^{r,q}$ crosses span j

$R^{r,q}$ – the target restorability level of working VP $g^{r,q}$ (1.0 used here)

D – total number of non-zero demand pairs in the demand matrix

d^r – number of demand units between end-node pair r

Q^r – total number of working routes available to satisfy the demand between node pair r

The main output variables are S_j, the spare capacity bandwidth allocation on all spans. Also obtained in the solution is the set of values $\left|f_k^{r,q}\right|$ which are the total bandwidth used on restoration route k for working VP $g^{r,q}$. The $f_k^{r,q}$ information effectively details the restoration plan for the whole network which accompanies the optimal spare capacity values. The $\left|f_k^{r,q}\right|$ values stipulate for the *qth* working VP serving part of the total demand on relation *r*, which of the *k* possible routes for its backup VP is actually used in the design. To implement Constraint 1 on $X_{j,i}$, the oversubscription level on span *j* in response to the failure of span *i*, the above variables are substituted for $X_{j,i}$ as shown below:

$$X_{j,i} = \left(W_j - \underbrace{\left\{ \sum_{(r,q)} \left|g^{r,q}\right| \zeta_i^{r,q} \zeta_j^{r,q} \right\}}_{\substack{\text{stub release traffic lost} \\ \text{from span j after failure} \\ \text{of span i}}} + \underbrace{\left\{ \sum_{(r,q)} \sum_{k \in P^{r,q}} \left|f_k^{r,q}\right| \delta_{k,j}^{r,q} \zeta_i^{r,q} \right\}}_{\substack{\text{restoration traffic cross-} \\ \text{ing span } j \text{ after failure} \\ \text{of span } i}} \right) / (W_j + S_j) \qquad (6)$$

Because the number of constraints and variables is large, it is generally very time-consuming and memory-consuming to solve this mixed integer formulation. Several methods have been taken to accelerate the solving process. First, $f_k^{r,q}$ is eliminated by substituting Equation 4 for Equation 6, because the former states the relation between $f_k^{r,q}$ and $\alpha_k^{r,q}$. Thus we can only use $\alpha_k^{r,q}$ in the final IP formulation. This eliminates half of the variables and constraints and greatly decreases the complexity. Second, several features of the MIP solver program are taken advantage of, such as the method of branch-

and-bound [23], setting the priority of $\alpha_k^{r,q}$ according to its traffic volume, i.e. larger capacity VPs are decided first. Because larger VPs dominate the spare capacity of a span, if larger VPs are chosen first we do not have to significantly adjust the spare capacity to accommodate smaller VPs. Additionally, because only one route can be chosen to restore the failure VP, we set the branch parameter to 'up' which causes the selected branching variable to be set to one, and then forces all the rest of the variables in the constraint to zero, which eliminates all the infeasibilities in that constraint [23]. These techniques greatly improve the speed of solving IP programs.

2.7 IP-2: Minimum Peak Oversubscription with Given Spare Capacity

The second formulation applies to the case where an existing set of spare capacity allocations has been given and the problem is to find a set of backup VP allocations that results in the smallest peak oversubscription of a restorable ATM network working within the given pattern and amounts of available spare capacity. All the variables are the same as in IP-1, but S_j is now an input instead of an output. This formulation can be used in general to minimize the maximum impact of restoration in situations where there is not enough spare capacity for complete restoration without oversubscription.

The objective is:

$$\text{Minimize:} \quad \{ \; max(X_{j,i}) \qquad \forall (i,j) \in S \;\; (i \neq j) \; \} \qquad (7)$$

where $max(X_{j,i})$ is the peak restoration-induced oversubscription resulting over all spans for all span cuts from the assignment of backup VP routes with given spare capacity allocations. $X_{j,i}$ is given by Equation 6. In order to achieve this minimum of maximum value, which is not a linear formulation directly, a new variable Y is added. Y is defined as the maximum value of $X_{j,i}$. Therefore the objective is changed to minimize Y and the following new constraints are added:

$$X_{j,i} \leq Y \tag{8}$$

The objective is subject to the following constraints:

1. Backup VPs are sufficient to meet the target restoration for all working VPs:

$$f_k^{r,q} = \left|g^{r,q}\right| \alpha_k^{r,q} \cdot R^{r,q} \qquad \forall k \in P^{r,q} \;\; \forall (r,q) \tag{9}$$

2. Only one backup VP can be used for each working VP, i.e. VP flows are not split:

$$\sum_{k \in P^{r,q}} \alpha_k^{r,q} = 1 \qquad \forall (r,q) \tag{10}$$

where $\alpha_k^{r,q} = 1$ if the k^{th} route for backup of $g^{r,q}$ is chosen, otherwise $\alpha_k^{r,q} = 0$.

2.8 IP-3: Minimum Total Capacity with Design Limits on Oversubscription

The constraint system IP-1 presented in Section 2.6 is adequate for VP restorable designs without jointly considering the routing of working demands before a failure. IP-1, however, can be extended to simultaneously optimize the working VP routes *and* the backup spare capacity placement. An IP formulation which minimizes the sum of working and spare capacity must not only determine the spare capacity per span and the routing of all backup VPs, but also the working capacity per span and the routing of all working VPs. By adding the following two constraints to the IP-1 formulation presented previously, the solution will include the values of W_j, and $g^{r,q}$ which will now minimize the *total* capacity-cost required in a path restorable network.

The objective function now becomes:

$$\text{Minimize:} \quad \left\{ \sum_{j=1}^{S} c_j \cdot (W_j + S_j) \right\} \tag{11}$$

Subject to:

1. The total capacity on the working VPs allocated to node pair r can carry all the requirement of traffic demand relation r:

$$\left|g_k^{r,q}\right| = \beta^{r,q} d^r \qquad \forall q \in D^r, \forall r \in D \tag{12}$$

2. Span j's working capacity is sufficient to meet the pre-failure demands of all working VPs which cross it:

$$(w_j) - \left(\sum_{r=1}^{D} \sum_{q=1}^{Q^r} \zeta_j^{r,q} \cdot \left|g^{r,q}\right| \right) = 0 \qquad \forall j \in S \tag{13}$$

3. Only one working VP can be used for each demand pair.

$$\sum_{k \in P^{r,q}} \beta^{r,q} = 1 \qquad \forall (r,q) \tag{14}$$

where $\beta_k^{r,q} = 1$ if working VP $g^{r,q}$ is used to carry the demand r, otherwise $\beta_k^{r,q} = 0$.

Here we require that there is only one working VP for one demand pair. If there is no restriction of the working routes for each demand pair, there are too many working VPs in the network. This inevitably increases the network management burden.

4. Sparing is sufficient to keep restoration oversubscription below the design limit, X_{tol}, for all failures:

$$(X_{j,i} \le X_{tol}) \quad \forall (i,j) \in S \ \ (i \ne j) \tag{15}$$

5. Backup VPs are sufficient to meet the target restoration levels *($R^{r,q}$)* for all working VPs:

$$\left|f_k^{r,q}\right| = \left|g^{r,q}\right| \alpha_k^{r,q} \cdot R^{r,q} \qquad \forall k \in P^{r,q} \ \ \forall (r,q) \tag{16}$$

6. Only one backup VP can be used for its working VP, i.e. VP flows are not split:

$$\sum_{k \in P^{r,q}} \alpha_k^{r,q} = \beta^{r,q} \qquad \forall (r,q) \tag{17}$$

where $\alpha_k^{r,q} = 1$ if the k^{th} route for backup of $g^{r,q}$ is chosen, otherwise $\alpha_k^{r,q} = 0$.

2.9 Related Bounds for Spare Capacity

In addition to *KST-Alg* and the IP formulations above, two more simple algorithms are presented here to calculate reasonably tight upper and lower bounds on the required spare capacity of a backup VP-based restorable ATM network. These bounds also provide a check on the IP-based results to follow. They may also be useful as relatively quick procedures yielding fairly tight bounds on the sparing requirements of a given network and working path VP routings in advance of detailed optimization. The lower bounding procedure in particular may be useful to rapidly generate starting point designs for large networks, with the IP -based optimization used subsequently to reach a final complete design.

2.9.1 Upper Bound Algorithm

The upper bounding algorithm is based on *KST-Alg* with a simple modification to strictly eliminate any restoration induced oversubscription. In *KST-Alg*, the spare capacity of a span is set as the largest VP requirement. If several VPs simultaneously fail upon one span cut, the network may suffer from very high oversubscription. The spare capacity on each span in this upper bounding algorithm is set to the *sum* of all the working VP capacities that traverse it, rather than the maximum of such values. For example, in Figure 5 on page 18, the upper bounding algorithm derived from *KST-Alg* says that span *ab* needs 7+5 = 12 units of capacity, rather than *max(7,5)* =7 units as *KST-Alg* does. This results in an over-provisioned design with a guaranteed maximum oversubscription factor of 1.0.

2.9.2 Lower Bound Algorithm

The lower bounding algorithm is based on IP-1 with the constraint in Equation 5 relaxed to allow real valued α. This converts the Mixed Integer Program as presented into a real-valued Linear Program (LP) which can be solved much more quickly in general. While serving as an LP relaxation of an IP problem, this MIP also represents a class of restoration system where VPs would be arbitrarily decomposable for restoration rerouting.

This can be more concretely represented by letting individual VCs in a VP take different routes in restoration. Thus, the LP formulation would assume that we are to use several backup VPs to handle the total flow of each working VP. The sparing achieved is thus a lower bound for the practical case where only one backup VP is available to restore each working VP.

2.10 Results with IP Formulations and Bounding Algorithms

2.10.1 Test Networks

The networks used in testing *KST-Alg* are now also used in the IP and bounding algorithms. In addition, two more networks used in other ATM restoration works [6] are used here. One is the metropolitan area network of Toronto, Canada while the other one is a US long-haul network. Note that the demand matrices for these two models are artificially set and, therefore, are clearly unrealistic. The demands were assumed to be the same between any two node pair.

2.10.2 Results of Spare Capacity

Table 5 summarizes the results of using the four capacity design and bounding algorithms.

TABLE 5. Spare Capacity Requirement Comparison

Network	*KST-Alg*	IP-1 @$X_{tol}=1.0$	upper bound	lower bound @$X_{tol}=1.0$
Smallnet	28.6%	N/A[a]	39.1%	24.9%
Net1	51.7%	74.8%	78.4%	71.4%
Net2	54.3%	82.6%	88.5%	76.9%
Net3	31.2%	81.5%	86.9%	78.7%
Net4	38.6%	92.0%	92.9%	91.4%
Toronto	14.2%	53.7%	56.4%	49.2%
US	5.4%	60.9%	64.2%	60.4%

a. Solving IP-1 for Smallnet is extremely complicated. No result can be achieved within reasonable time.

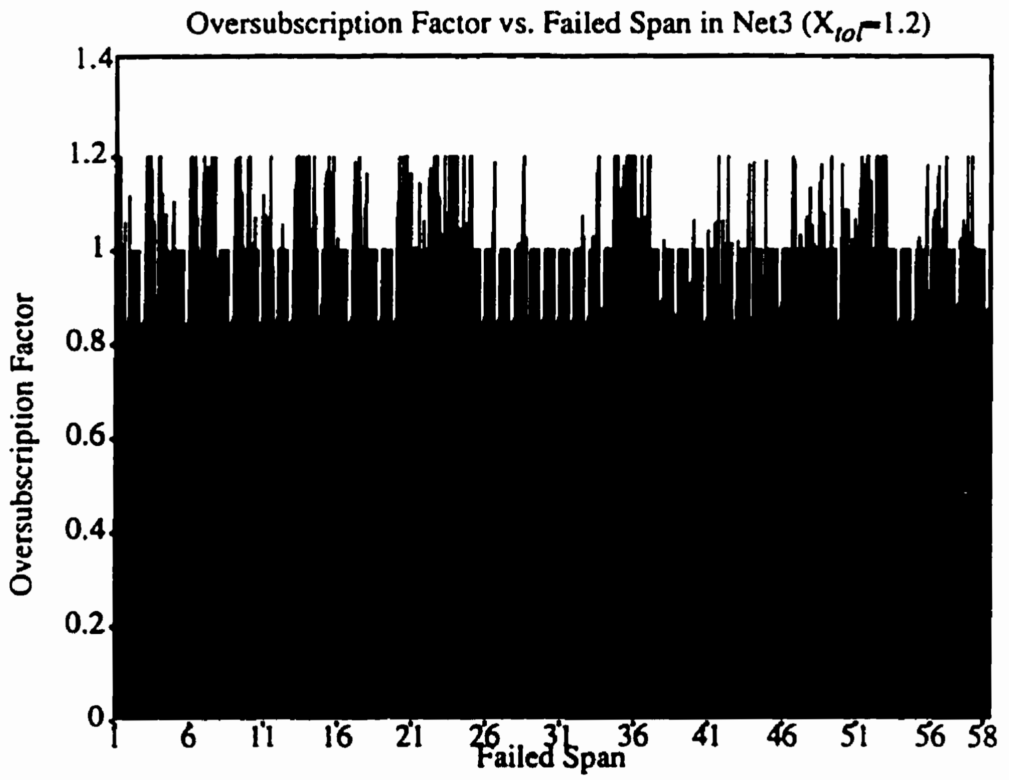
The results with IP-1 are based on an allowable oversubscription factor X_{tol} of 1.0. *KST-Alg* has the minimum spare capacity but has severe and frequent oversubscription cases. IP-1 falls between two bounding algorithms which provide a quick check and potential starting points for the exact problem. The required spare capacity for IP-1 increases greatly compared with *KST-Alg* in order to eliminate the oversubscription effects. In metropolitan networks such as Net1 and Net2, spare capacity found in IP-1 increases about 60% from *KST-Alg*, while in long haul networks such as Net3 and Net4 the capacity increases about 100%. It is also noted that the spare capacity requirement for Toronto and US networks given by KST-Alg is very low due to the fact that the demand matrix is uniform and that there is one unit of demand for every node pair. Therefore, in each span, the spare capacity is at most one demand and shared by several VPs. In these highly shared capacity networks the required spare capacity is very low.

Another interesting result is the lower bound on spare capacity when we decompose the VP and distribute the traffic over several routes. It is not surprising that it achieves a better result than IP-1, because traffic is shared by more spans and spare capacity is shared to a greater extent. However, we find that there is only a small reduction of the spare capacity by 1% to 5% from IP-1. Less than 1% capacity is saved especially in Net4. This implies that we do not have very much room to save the spare capacity, even if some elegant mechanisms were to be used to decompose and distribute the VP traffic at the VC level. Spliting the VP traffic can be potentially very complicated, would not likely provide any big advantage.

Figure 9 shows the oversubscription analysis in Net3 when using IP-1 with the tolerable oversubscription factor at 1.2 as an illustrated test case. The $X_{j,i}$ data in each is obtained from separate programs that conducted restoration experiments for each span failure using the assigned backup VP routes in each design. The structure of this figure is the same as that in Figure 8 on page 27. Thus, the tight clamp on $X_{j,i}$ values of 1.2 in this figure validates IP-1 for its intended properties. The *IP-1* design at $X_{tol}=1.2$ has about 50% more spare capacity than the *KST-Alg* design and 30% spare capacity less than the equivalent STM design. This capacity saving benefit compared with $X_{tol}=1.0$ is discussed in detail in the next section.

2.10.3 IP-2 Results

Since *KST-Alg* allocates the least total spare capacity, it is of interest to see how low the peak oversubscription can be capped within this sparing if *IP-2* is applied to the *KST-Alg*



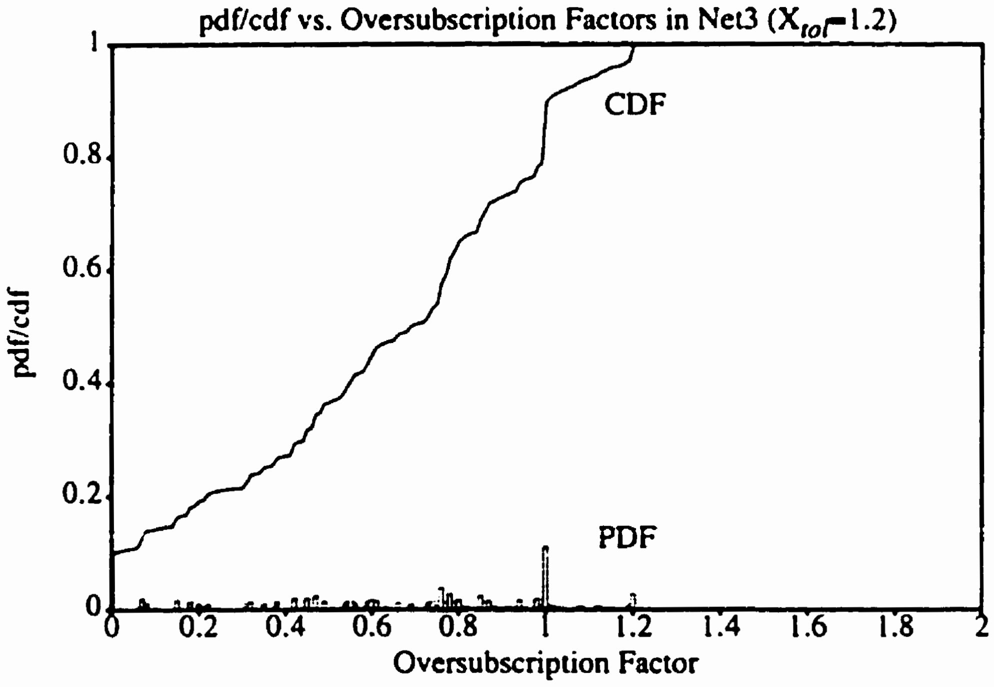


FIGURE 9. Oversubscription Analysis of Net3 with X_{tol} Set to 1.2

spare capacity design to improve the coordination of backup VP assignments to reduce the peak oversubscription factor. Figure 10 shows this result for Net-3 which illustrates the application of *IP-2* to improve on the worst-case oversubscription of the *KST-Alg* spare capacity design but with exactly the same spare capacity placement that *KST-Alg* placed in the first instance. By rearranging the backup VP assignments, *IP-2* manages to reduce the peak oversubscription of the *KST-Alg* design to 3.04 from 4.16 (in Figure 8 on page 27) while retaining 31% spare capacity. The side effect of reallocating backup VPs to reduce the peak oversubscription is that there are more individual overload cases. When we squeeze the maximum oversubscription down by applying IP-2, the restoration flows are distributed more extensively over all spans and then more spans suffer from oversubscription effects although the peak factor is lowered.

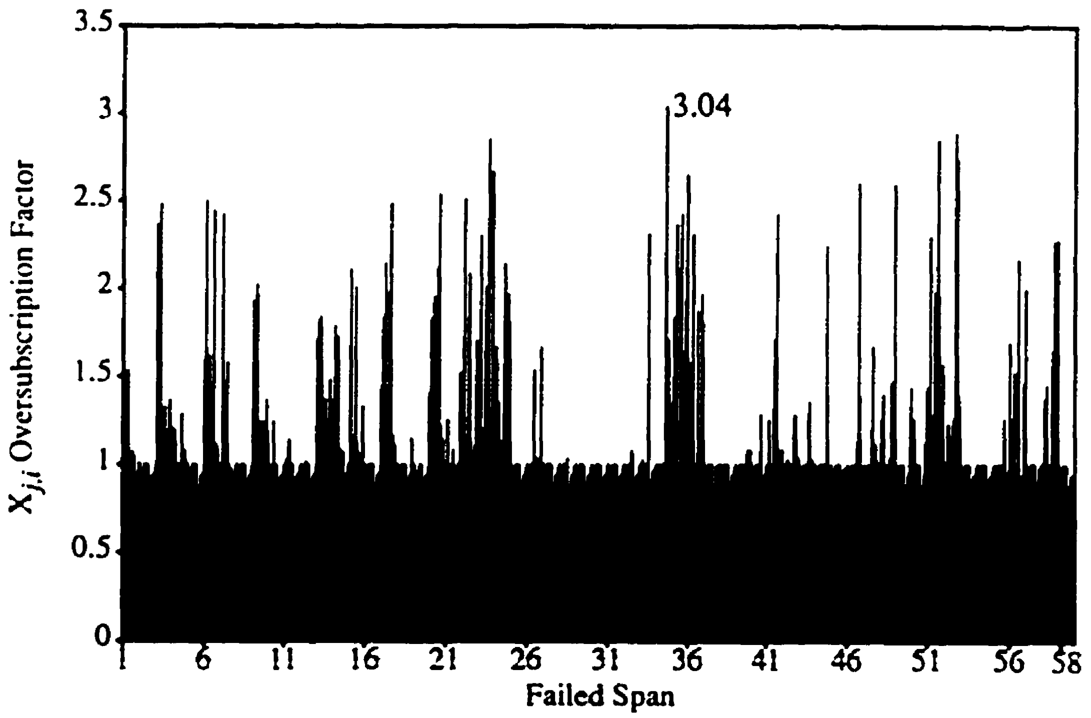


FIGURE 10. Oversubscription Factors in IP-2 Design for Net-3 with Sparing from *KST-Alg* (31% spare capacity)

2.10.4 IP-3 Results

Before presenting the results of IP-3, a simple complexity analysis must be given of the IP-1 formulation as related to IP-3. Suppose a network has D demand pairs, W possible working routes for a demand and B possible backup routes for each working VP. The complexity of IP-1 is $O\left(B^D\right)$ because each working VP (or demand) has B choices and the total number of working VP is D. The complexity of IP-3 is $O\left(W^D B^D\right)$ because for every combination of working VP arrangement the complexity is the same as IP-1. There are $O\left(W^D\right)$ working VP arrangement combinations in total. From this analysis, we can deduce that it is hard to obtain completed IP-3 runs on networks of any significant size. Therefore, only three results of the tested networks are given here. In Table 6, the capacity shows the working, spare and total capacity in the tested networks. Note that the total capacity is normalized to the case of IP-1 and shown as a percentage. We find that total capacity saved can be up to 10%.

TABLE 6. Total Capacity Requirement Comparison

Network	IP-1 @$X_{tol}=1.0$		IP-3 @$X_{tol}=1.0$	
Net4	100%		97.3%	
	W=702758	S=646108	W=737197	S=575858
Toronto	100%		90.6%	
	W=81200	S=43566	W=88300	S=24700
US	100%		96.3%	
	W=1832900	S=1115500	W=1958500	S=880300

When the IP is allowed to jointly optimize the placement of working and spare capacity in a network, it chooses working paths which are coordinated with the network restoration process. This means that demands may sometimes be routed via paths longer than the shortest path. This increases the working capacity in the network, but more spare capacity can potentially be saved. For example, in the Toronto network, the working capacity in IP-3 is 7100 units more than that in IP-1. On the other hand, the spare capacity is 18866 less. Overall, the total required may be reduced if working and spare capacity are jointly minimized.

2.11 Spare Capacity versus Tolerable Oversubscription Design Trade-off

Using IP-1 it is possible to explore how the total spare capacity of the network responds to increasing X_{tol}. Table 7 summarizes the designs for each of our test networks for X_{tol} ranging up to 2.0. For comparative presentation, all spare capacity totals are normalized to that of $X_{tol}=1.0$ case for each network. The total spare capacity decreases rather quickly as the design tolerance for restoration-induced oversubscription increases. With 10% design maximum oversubscription of bandwidth on restoration ($X_{tol} = 1.1$), spare capacity is reduced by a range of 17% to 23%. At a more aggressive $X_{tol} = 1.5$, a full 60% to 70% reduction of the spare capacity is obtained.

TABLE 7. Spare Capacity Requirement vs. Allowable Oversubscription Factor

Design X_{tol}	Net1	Net 2	Net 3	Net 4	Toronto	US
1.00	100%	100%	100%	100%	100%	100%
1.05	91.5%	90.0%	90.0%	90.1%	87.5%	87.7%
1.10	82.9%	82.3%	80.6%	81.1%	76.3%	79.0%
1.15	75.0%	75.5%	76.9%	72.7%	66.7%	65.9%
1.20	68.0%	69.0%	65.9%	65.4%	57.3%	57.1%
1.25	62.0%	65.0%	59.3%	58.3%	50.4%	49.1%
1.30	57.3%	58.9%	53.6%	52.1%	40.5%	42.6%
1.40	48.6%	49.9%	43.7%	40.9%	30.4%	32.1%
1.50	40.9%	43.3%	34.9%	31.1%	23.0%	24.7%
1.75	26.5%	30.4%	22.7%	13.8%	7.1%	11.3%
2.00	17.3%	23.2%	14.2%	5.4%	1.9%	5.5%

X_{tol} is, however, the strict peak oversubscription level that we will tolerate in the IP-1 designs. This maximum $X_{j,i}=X_{tol}$ may occur for only one specific combination of failure span and restoration span in the design. It is, therefore, worth inspecting the number of spans that actually experience a given level of oversubscription within a design tolerance of X_{tol}. Figure 11 considers this in terms of the 90th percentile of *actual* oversubscription levels experienced by spans over all span cuts versus the design X_{tol}. The data shows, for example, that at $X_{tol} = 1.4$, 90% of the spans *actually* experience oversubscription no greater than 1.06, 1.08, 1.21 and 1.28, 1.33, 1.36 in Nets 3, 4, 2, 1, US and Toronto respectively. This adds to the expectation that fairly significant capacity savings could be possible in practise without severe restoration-induced side-effects, through judicious choice of X_{tol} as a parameter for the basic design of the network [10].

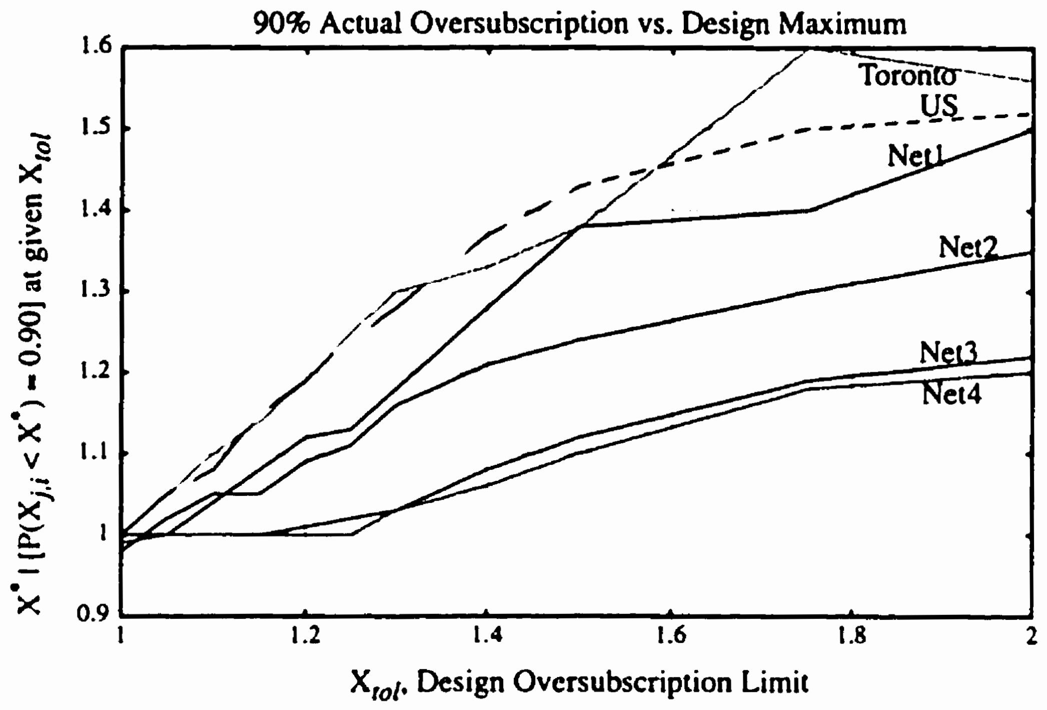


FIGURE 11. 90th Percentile Actual Oversubscription vs. Design Maximum

2.12 Comments on Tolerable Oversubscription

Here, we discuss the important issue of the oversubscription factor which is tolerable in network planning. The maximum acceptable level of restoration-induced oversubscription would depend on whether worst or average case VP utilizations and traffic statistics are assumed for determining such a guideline. It may also be in part a policy or business issue; if there is to be strictly no degradation on restoration, then $max(X_{j,i}) = X_{tol} = 1.0$ and the network restoration planning is equivalent to STM (i.e., perfect bandwidth replacement). In a network that is lightly loaded in terms of cell level utilization of the installed bandwidth, some $X_{j,i}>1$ could clearly be tolerated before QoS guarantees are affected greatly. An alternate business point of view might be that all VPs should be allowed to suffer to a degree during a network restoration event. The QoS impact also depends on the time of the failure relative to the busy period and the equipment provisioning interval. At the time of an actual failure, each surviving span would assess its actual cell-level utilization after allowing enough time for backup VP switching to occur. It would then either do nothing, in which case utilizations were low enough to provide restoration for all services, or it would mark the lower priority VPs traversing it with a throttling indication to be acted upon either by the VP sources themselves or neighboring switches. All this considered, a relatively high X_{tol} might actually be practical. In practise the aggressiveness of each network provider in designing ATM restorable networks would be expected to vary in this regard. Some quantitative guidelines as to the acceptable X_{tol} will be obtained from sub-studies of the theoretical queuing delay and cell loss increase effects for different merging traffic types discussed in the next chapter. What is useful at this stage, however, is to provide a design formulation that would allow us to explore the capacity savings that are obtainable in ATM restoration depending on the maximum restoration-induced overload factor that is considered admissible.

2.13 Conclusion

Based on a logical view of traffic in backup VP-based ATM restoration, the restoration induced flow convergence oversubscription factor wa quantitatively defined. The oversubscription factor was defined as the ratio of presumed bandwidth for restoration to the actual link bandwidth allocation. The technical property of an oversubscription larger than 1.0 is one of the unique properties in ATM networks.

The spare capacity placement algorithm *KST-Alg* proposed by NTT was implemented to test it for the restoration induced oversubscription effects. The spare capacity requirement produced by *KST-Alg* is indeed much lower than required by STM networks. It is important to note that these particularly low spare capacity levels are accompanied by significant and strictly uncontrolled oversubscription effects on surviving spans. With very high levels of oversubscription, cell loss and cell delay in ATM networks would very likely be intolerable for many applications.

The capacity savings relative to STM networks are very attractive, but the uncontrolled oversubscription implications are probably unacceptable in practise. Therefore, we formulated optimal capacity allocation methods that will still gain as much ATM-related capacity savings as safely possible by giving us a controlling input on the maximum extent of the restoration-induced oversubscription. The first IP formulation optimizes the spare capacity placement of a restorable ATM network given a peak allowable oversubscription factor in the network. The second formulation applies to the case where an existing set of spare capacity allocations has been given; the problem, then, is to find a set of backup VP allocations that results in the smallest maximum oversubscription factor. The third formulation tries to minimize the total (working + sparing) capacity of a restorable ATM network with a given design peak oversubscription factor. In addition to these IP formulations, two simpler algorithms were presented to calculate reasonably tight upper and lower bounds on the required spare capacity of a backup VP-based restorable ATM network.

The results obtained indicated that the total spare capacity decreases rather quickly as the design tolerance for restoration-induced oversubscription increases. With a 10% design maximum oversubscription of bandwidth on restoration ($X_{tol} = 1.1$), spare capacity is reduced by a range of 17% to 23%. At a more aggressive $X_{tol} = 1.5$, a full 60% to 70% reduction of the spare capacity is obtained. This suggests that significant capacity savings can be obtained relative to STM if ATM restoration is allowed even modest restoration-induced oversubscription of bandwidth on surviving spans.

The tolerable oversubscription factor depends on several considerations. In part, this factor is a network operation policy. An aggressive network operator may use a large oversubscription to save more valuable capacity in the network. As the logical next step in this study, a guideline governing the choice of oversubscription factors will be given based on the theoretical study of a queuing model of traffic. In the next chapter, some simulations are run to find what X_{tol} might actually be feasible.

Chapter 3. Tolerable Overload Assessment

3.1 Tolerable Overload Assessment Method

In the previous section, we found that a fairly significant amount of capacity can be saved if even a modest oversubscription factor is allowed in the restoration spare capacity placement design. The oversubscription factor $X_{i,j}$ is only for network planning purposes because whether a real failure causes an actual overload of traffic and a QoS degradation depends on several factors. If VPs are not simultaneously utilized at their peak levels, an actual cell-level overload may not occur. Only in the worst case where all VPs are fully loaded, does the oversubscription factor indicate the actual overload in the network. The overload O can be defined as following:

$$O \equiv f(X_{i,j}, U, T, P) \tag{18}$$

Where

O is the tolerable overload of the network,

$X_{i,j}$ is the oversubscription factors in the network,

U is the utilization factors of VPs in the possible failure cases.

T is the traffic type, e.g., CBR, VBR, ABR, UBR, in VPs, and

P is the traffic parameters, e.g., rate, burstiness.

Here we can find that lots of factors are involved to decide the tolerable overload. The overload illustrates the ability of allowing overload traffic for an given traffic.

The next question facing network designers/operators is, then, what is a reasonable theoretical level of overload. Because spare capacity saving increases with a large overload factor, we should use as large an overload factor as possible. Conversely, since overload inevitably degrades network performance, it should not be arbitrarily large. A design method should be provided which identifies the largest overload factor that does not degrade QoS beyond a manageable level. We therefore note that:

1. The network traffic types affect the tolerable overload factor greatly.
2. Different networks have their own characteristics.
3. Some networks mainly carry bursty traffic, while others mainly carry continuous traffic.

The tolerable overload, of course, is not the same in different networks.

In order to get the tolerable overload factor, let us first recall the definition of overload in Section 2.2 on page 14. It is defined as the ratio of allocated traffic to the link bandwidth. As we know, traffic multiplexing in STM networks is deterministic. On the contrary, traffic in ATM networks is statistically multiplexed. There is no easy way to get the traffic volume directly. To overcome this problem, we borrow the concept of *Equivalent Bandwidth* from *Call Admission Control* (CAC) algorithms [19]. Equivalent bandwidth is defined as the effective bandwidth requirement of connections multiplexed into one link which meets the required QoS. When a new traffic source is added, the network can decide to accept this new connection based on its equivalent bandwidth and available bandwidth. Note that equivalent bandwidth only depends on the traffic source. The link bandwidth into which the traffic is induced is not relevant here. Equivalent bandwidth of the traffic shows the required bandwidth to accommodate the traffic.

To illustrate how to apply the concept of equivalent bandwidth to the overload calculation, consider a group of ATM traffic sources that are aggregated into a link. The traffic volume, or the equivalent bandwidth of the traffic, is 110Mbps. If we route this traffic to a link whose capacity is 110Mbps, it is obvious that QoS is normal and no performance degradation occurs. In this case, the overload factor is 1.0. However if we route this traffic to a link whose capacity is 100Mbps, more cells are lost and the delay increases due to the queuing overflow. In this case, the overload factor is 1.1 (= 110Mbps/100Mbps). If the capacity of the link is less, it is expected that overload will be high and the QoS degradation will be more severe, as illustrated in Figure 12. Our overall objective in planning ATM network with oversubscription is to find the largest overload factor so that QoS is still at a tolerable level. This largest overload value, then, is called the tolerable overload.

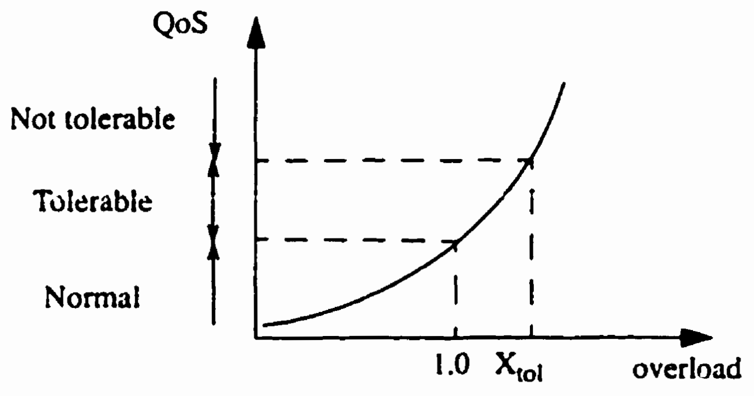


FIGURE 12. QoS versus Restoration Induced Overload Factor

To gain some idea of the tolerable overload in practical circumstances, a group of traffic sources and a finite queuing buffer are given. We first calculate the equivalent bandwidth for those traffic sources when QoS is normal. Then we calculate the bandwidth when QoS is still tolerable. Using the definition of overload, the tolerable overload factor for this group of traffic sources is obtained by dividing the equivalent bandwidth by the least tolerable bandwidth. How to calculate the bandwidth requirement to ensure a given QoS becomes the new problem. As there is no analytical method to get the equivalent bandwidth and least bandwidth [19], we have to use extensive simulations to get these two values.

To study the effect of traffic characteristics on the overload factor, we will change individual system description variables, such as source utilization and buffer size, while keeping all other factors constant. Then, we analyze how the tolerable overload factor responds to these individual factors. For example, to study the effect of source utilization, we use a group of traffic sources. The overload factors are calculated in several cases using utilization of every source equal to 0.1, 0.2, up to 0.9. Analyzing the result, we can find how the overload factor changes with the source utilization. Using this method, we can get the effect of other traffic descriptors.

3.2 On/Off Fluid Traffic Model

Several models of ATM traffic have been under active research. Here we use the on/off fluid model due to its simplicity and adequacy [11]-[13]. This model is also used in equivalent bandwidth based call admission control schemes [11], as illustrated in

Figure 13. In the on/off fluid model, there is a continuous alternation of active and idle periods. In active periods, the source constantly transmits at its peak bit rate. In idle periods, no cell arrivals occur. The durations of two periods are exponentially distributed (i.e., in poisson fashion). Such a source model has the advantages of being both simple and flexible as it can be used to either represent connections ranging from bursty to continuous bit streams or approximate more complex sources.

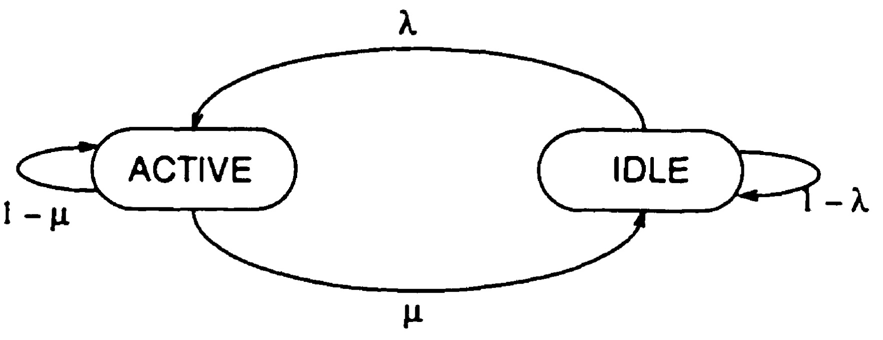


FIGURE 13. State Diagram of an On/Off Fluid ATM Traffic Model

The on/off fluid model uses three traffic descriptors:

1. R, the peak bit rate, which is the bit rate in active mode,

2. ρ, utilization, which is the percentage of time in active mode, and

3. b, mean burst length, which is the mean length of the active mode.

Other descriptors can be derived from the above.

m: mean bit rate, $m = R\rho$;

μ: transition rate out of active state. $\mu = 1/\rho$;

λ: transition rate out of idle state. $\lambda = \rho/(b(1-\rho))$. The mean idle length is

$1/\lambda$, i.e., $(b(1-\rho)/\rho)$.

3.3 Equivalent Bandwidth

Besides the parameters of the traffic model, the following factors are also involved in the equivalent bandwidth calculation:

1. c, the link bandwidth,
2. B, size of the finite buffer, and
3. ε, cell loss ratio (CLR).

Consider n traffic sources (R_i, ρ_i, b_i) $i = 1 \ldots n$ being multiplexed into a link with a finite queue (size B) and a link bandwidth c. In general, let Z be a random variable denoting the aggregate bit rate of all sources. Then, the dynamics of the queue in the system are defined as follows:

1. If $Z < c$ and
   a. the buffer is empty, then
      it remains empty;
   b. the buffer is not empty, then
      its content decreases at a constant rate of $c - Z$.
2. If $Z = c$, then
   the buffer content does not change.

3. If $Z > c$ and

a. the buffer is not full, then

the buffer content increases at a constant rate of $Z - c$;

b. the buffer is full, then

the buffer is still full, the cells are lost at a constant rate of $Z - c$.

The equivalent bandwidth $\hat{c}$ is defined as the minimum link bandwidth c, where the probability of cell loss is less than some desired Cell Loss Ratio ε.

The analytical derivation of equivalent bandwidth for several mutually independent identical sources is summarized in Appendix A. A detailed analysis can be found in [13] and [19].

In general, the distribution of the buffer content is of the form:

$$F(x) = \sum_{i=0}^{N} a_i \Phi_i e^{z_i x} \qquad (19)$$

where the z_i and Φ_i are, respectively, generalized eigenvalues and eigenvectors associated with the solution of the differential equation satisfied by the stationary probabilities of the system, and the a_i's are coefficients determined from boundary conditions [19].

The distribution of $F(x)$ is completely determined from the values of the associated eigenvalues, eigenvectors, and corresponding coefficients. There are no explicit expressions for these quantities, so they must be determined numerically.

An important aspect of this problem is numerical stability. The inevitable errors incurred during numerical integration, no matter how small, are liable to excite the unstable modes and lead to solutions of $F(x)$ that blow up.

The analysis in previous papers [11]-[14] has shown that the most important factors affecting equivalent bandwidth are the ratio of peak bit rate to link bandwidth, traffic source utilization, and mean burst length.

3.4 Cell Loss Ratio Consideration

Cell Loss Ratio (CLR) is an important factor in the calculation of equivalent bandwidth. It is obvious that equivalent bandwidth is larger if the CLR requirement becomes more stringent. In this section, we discuss CLR in normal working conditions and a tolerable CLR to allow in a restored network state. In this simulation, we only consider CLR due to the queuing buffer overflow (i.e., bit error rate effects are ignored).

In a real network, several factors are involved in the final system overall CLR, such as the BER of the transmission media, losses in the ATM concentrator/switch, and software. Combining all these factors, the overall CLR can be obtained which is dominated by the worst factor. In our simulation, CLR is only due to queuing saturation. It is expected that our CLR is of the same order of magnitude as the overall CLR. Therefore, our CLR should be neither unnecessarily low nor too high.

Assumed values for some of the of major factors affecting the overall CLR include below 10^{-11} for the BER of fibre optics, 10^{-9} for the end-to-end objective of the underlying physical layer DS-3 circuits, 10^{-10} for ATM concentrator or switch, and 10^{-7} for queuing of application level software [17]. Different values are adopted in their simulations for equivalent bandwidth, such as the most stringent CLR service objective of 10^{-10} in [11], more relaxed value of 10^{-5} in [12]-[14], 10^{-6} [11] and 10^{-7} in [18]. In [11], the authors studied the effect of the required cell loss probability ranging from 10^{-9} to 10^{-3}. These numbers give us a general view of CLR requirement due to buffer saturation. In our simulation, 10^{-9} is used as the nominal working objective to calculate the equivalent bandwidth. In case of overload, a tolerable CLR of 10^{-5} is assumed. This is consistent with previous research [13, 16, 15, 20].

Let c_{-9} and c_{-5} denote the equivalent bandwidth when CLR is 10^{-9} and 10^{-5} respectively. The physical explanation is that a traffic volume c_{-9} is routed to a link whose nominal design capacity is c_{-5}. Accordingly the overload is:

$$O = \frac{c_{-9}}{c_{-5}} \tag{20}$$

3.5 Simulation Design

The system model used in our simulation consists of a group of traffic sources, a finite queue and a transmission link. Cells arrive asynchronously to the queue from the sources.

They are multiplexed on a FIFO basis and transmitted out onto the link. The finite queue is served by the link.

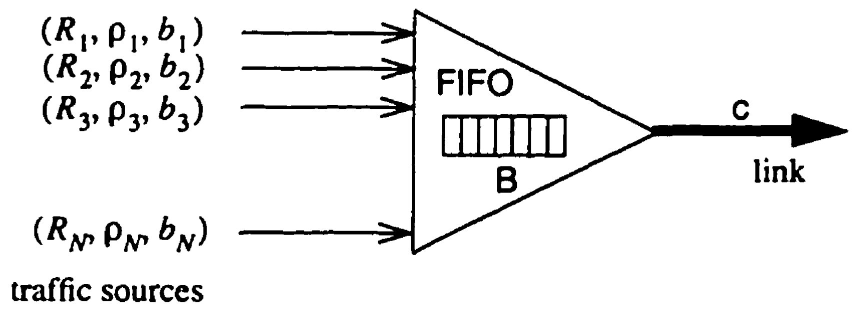


FIGURE 14. Simulation Queuing Model

It should be emphasized that this simulation can only obtain the CLR if a link capacity is given. The opposite is not feasible by running only one simulation (i.e., to obtain the equivalent bandwidth with a given CLR requirement). Consequently, to get the bandwidth requirement for a given CLR, this simulation has to be run several times with a set of differing link capacity values. The capacity of the nearest CLR is regarded as the bandwidth required for that given CLR. Based on this mechanism, we get the equivalent bandwidth c_{-9} and bandwidth c_{-5} with the target CLR of 10^{-9} and 10^{-5}. The corresponding overload factor is calculated using Equation 20.

The traffic sources are mutually independent. Each source is a continuous alternation of active and idle periods. The length of the active period is a poisson process whose mean value is b, whereas the length of idle period is a poisson distribution whose mean value is $b(1-\rho)/\rho$. Within one period, the traffic is constant. It is either at its peak bit rate or zero depending on its state.

In this simulation, an event-driven model is also used. The simulation is not driven by "*timer*", but by "*events*". An event is any transition of any traffic source. The transition can either move from idle to active or from active to idle. Between any two adjacent events, the state of all traffic sources is unchanged. During these events the aggregated traffic rate of all sources is constant. A series of transition events is generated for all sources over the whole simulation time. The length of active and idle periods conforms to the respective poisson process mean values. After verifying all transition events and traffic in all periods, we can calculate the buffer content, total traffic, and cell losses according to the analysis in Section 3.3.

The observation window of the whole simulation is set large enough to hold at least 200,000 transitions of any source. Having tried several values for the window size, we have found that this number is large enough to make the results reasonably stable. The simulation process was repeated several times with different random number seeds and the average overload factor which raises CLR from 10^{-9} to 10^{-5} was used as the final result.

This simulation program is written using the C language in a UNIX environment. Its correctness was checked by a separate Matlab implementation which is, of course, much slower.

3.6 Results

3.6.1 Simulation Parameters

The parameters for the on/off fluid model used in this simulation were adopted from [11]. In [11], several *Call Admission Control* (CAC) schemes are compared based on the same set of traffic descriptors. Traffic models are characterized by three descriptors: peak rate, utilization and mean burst length. The peak rate is normalized to a reference link bandwidth. The burst length is in the unit of time intervals. A three-element vector (R, ρ, b) is used to represent them. For example, (0.08, 16%, 72) denotes a traffic model whose peak rate is 0.08, utilization is 16% and mean burst length is 72. In the simulation in [11] and our simulation, two basic classes of traffic are used, (0.05, 20%, 80) class 1 and (0.1, 20%, 50) class 2. The numbers of class 1 and 2 traffic sources are 70 and 35, respectively. In order to calculate the overload factor for our purposes, the buffer size normally chosen is 100. Using these values as basic parameters, we can change any one of them to analyze the effect on the tolerable overload factor. We will discuss the implications of the following results in Section 3.9. It is worthwhile to reiterate that the tolerable overload factor is determined for a traffic with a normal CLR of 10^{-9} and a CLR of 10^{-5} during restoration. Before going on, it should be noted that tolerable overload factor inherently shows the tolerance for a newly induced traffic of an existing traffic. For the existing traffic, if we can induce more traffic along with it before degrading the QoS significantly, the tolerable overload factor is high.

3.6.2 Effect of Peak Rate

To study the effect of peak rate, using 70 traffic sources whose utilization is 20% and mean burst length is 80, we varied the peak rate from 0.05 to 0.11. In Figure 15, we find that by increasing peak rate, the tolerable overload factor increases from 1.070 at peak rate 0.05 to 1.103 at peak rate 0.11. When the peak rate is low, it is expected that we may only induce a low peak rate traffic. If a higher peak rate traffic is induced, CLR can easily become intolerable. On the other hand, if the peak rate is high, the tolerance of high peak rate traffic is increased. Consequently, the tolerable overload factor is high. Therefore, for network planning, if the peak rate of the network traffic is high, a high tolerable overload factor should be chosen. On the other hand, if the network traffic has a low bit rate, a low

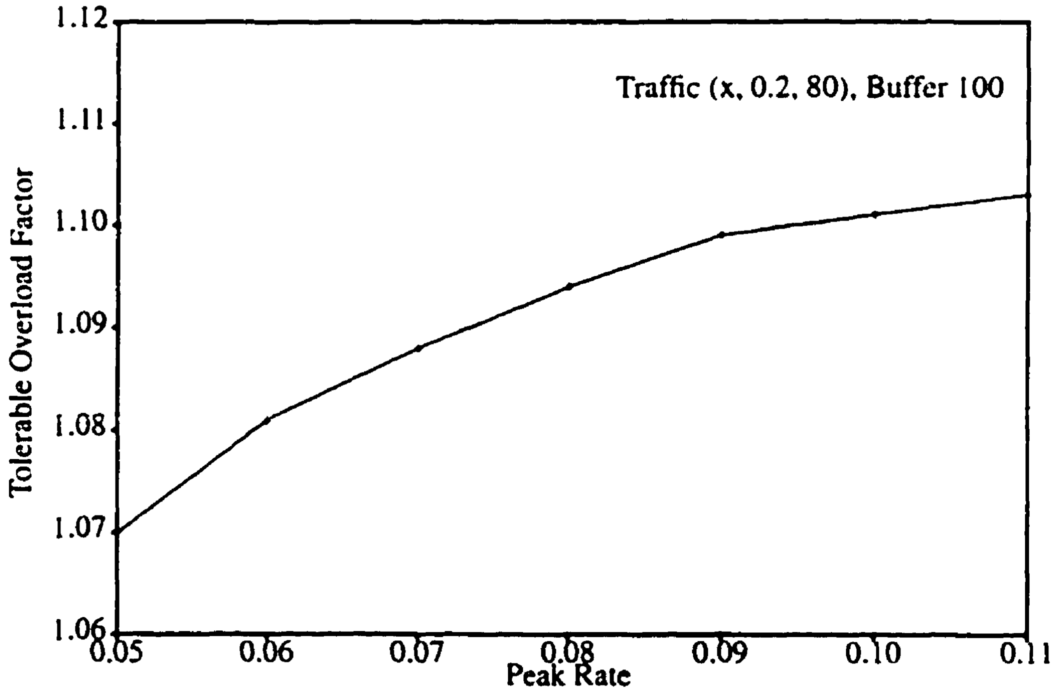


FIGURE 15. Tolerable Overload Factor versus Peak Rate of Sources

tolerable overload factor should be used. For example, an overload of 1.19 could be used with peak rate of 0.10, while an overload of 1.08 could be used with peak rate of 0.06.

3.6.3 Effect of Utilization

Next we change the utilization for two classes of traffic. For class 1, whose peak rate is 0.05 and mean burst length is 80, and class 2, whose peak rate is 0.1 and mean burst length is 50, the utilization was varied from 10% to 90%. In Figure 16, it is shown that when the

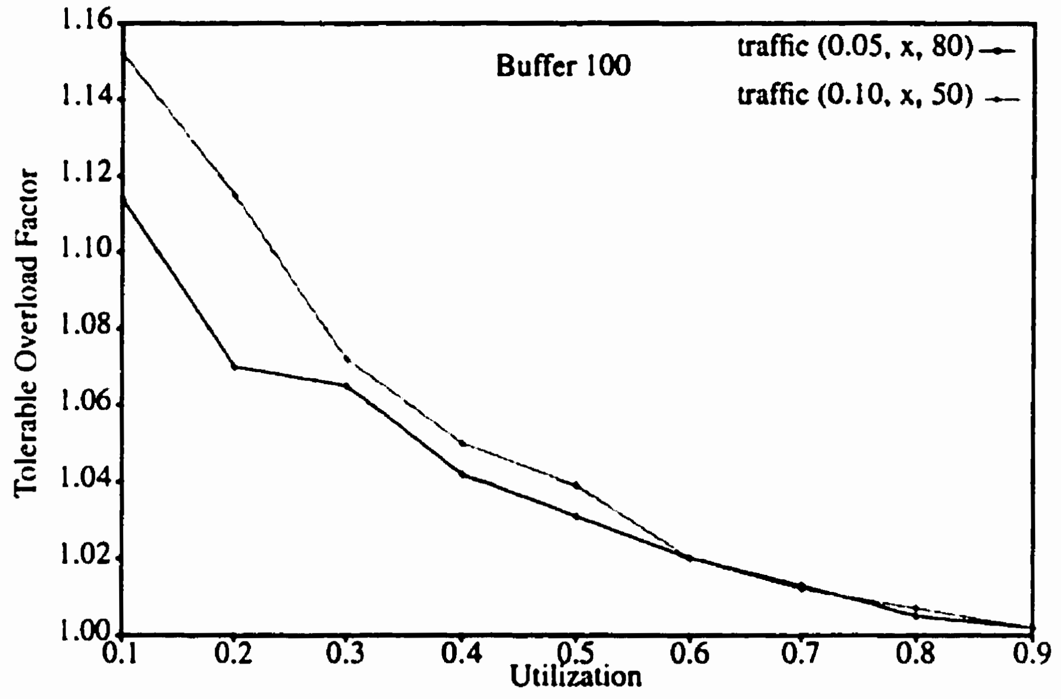


FIGURE 16. Tolerable Overload Factor versus Source Utilization

utilization is increased from 10% to 90%, the overload factor decreases from 1.114 and 1.152 to 1.002. When the utilization increases to 100% (i.e., a constant rate traffic), we can expect the tolerable overload factor to approach 1.0, the reason being that with low utilization, and thus more idle periods in the network, we can induce more traffic. Consequently a high tolerable overload factor may be used for ATM network planning.

In network planning, if the network is loaded with low utilization traffic, we can use a high tolerable overload factor which would reduce the spare capacity in the network. In the case where the utilization is 100%, the tolerable overload factor is close to 1.0.

3.6.4 Effect of Mean Burst Length

Next we change the mean burst length for two classes of traffic. For class 1 whose peak rate is 0.05 and utilization is 20%, and class 2 whose peak rate is 0.1 and utilization is 20%, the mean burst length was varied from 30 to 100. In Figure 17, we find that with the increase of mean burst length, the overload factor increases from 1.027 to 1.082 and 1.068 to 1.160 respectively. When the burst length is high, the idle burst length is also high. This implies that a long idle period so that more traffic can be induced into the network. For network planning, if the network traffic has a long burst length and the idle burst length is also high, we may use a high tolerable overload factor. Otherwise, a low tolerable overload factor may be used.

3.6.5 Effect of Buffer Size

Next we change the buffer size for two classes of traffic. For class 1, whose peak rate is 0.05, utilization is 20% and mean burst length is 80, and class 2, whose peak rate is 0.1, utilization is 20%, mean burst length is 50, the buffer size is varied from 30 to 100. In Figure 18, as the buffer size is increased, the overload factor decreases from 1.119 to

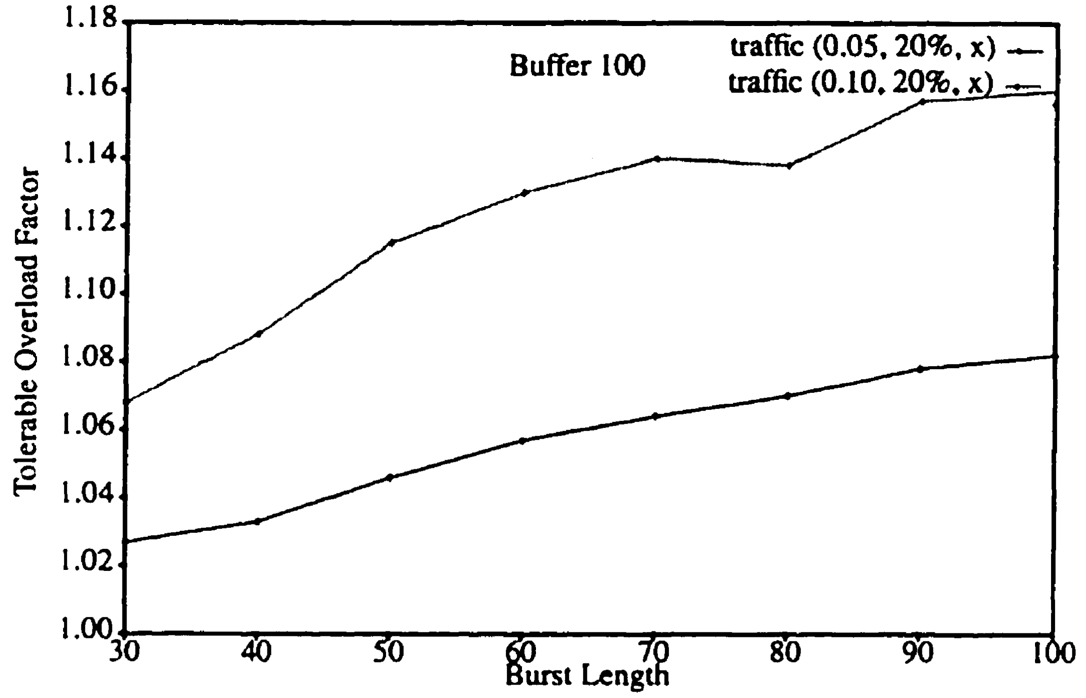


FIGURE 17. Tolerable Overload Factor versus Burst Length of Sources

1.070 and 1.173 to 1.115 respectively. This is consistent with the change of burst length. A buffer size increase is equivalent to a decrease in burst length. It is worthwhile to note the difference between $C_{.5}$ and $C_{.9}$ and tolerable overload which is their ratio. If the buffer size is larger, the equivalent bandwidth is smaller but the tolerable overload is also smaller as shown in Figure 18. This is because the traffic becomes smooth if the buffer is larger. If the buffer is infinite, the traffic is constant and almost no overload is allowed. Therefore, the tolerance to overload, expressed as a multiplier of the baseline traffic is more limited because the system is more efficient in the first place, if the buffer is large. For network planning, if we have the same network traffic but the buffer size inside ATM switches is enlarged, a low tolerable overload factor should be used. If the buffer size becomes smaller, a higher tolerable overload factor may be used.

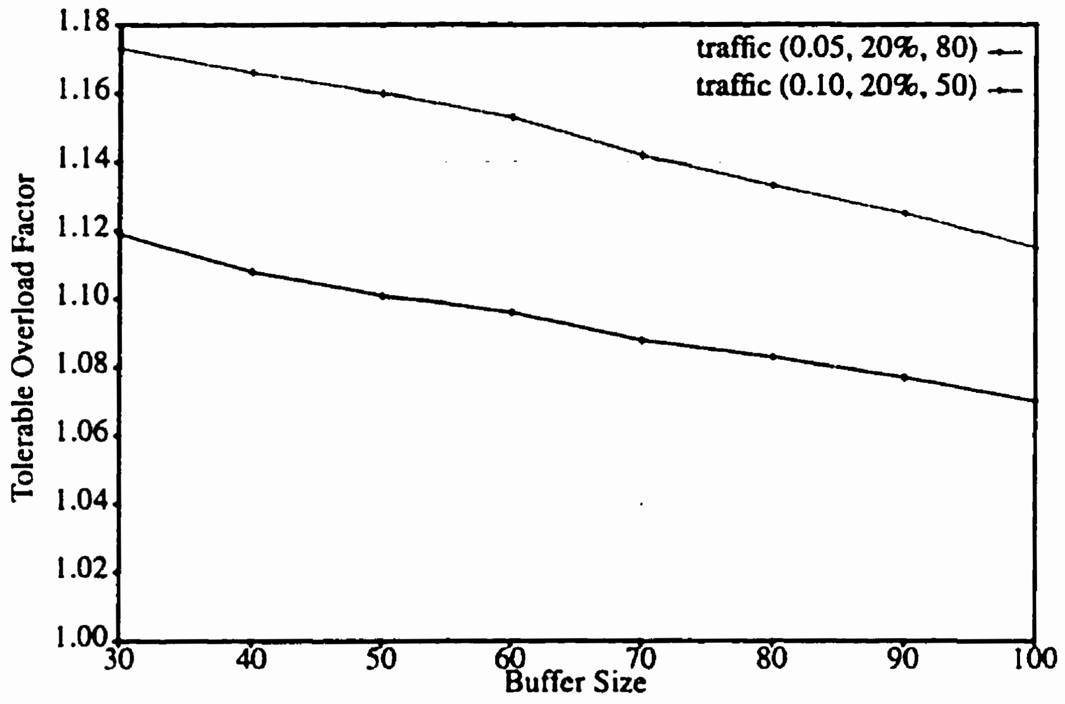


FIGURE 18. Tolerable Overload Factor versus Buffer Size of Sources

3.6.6 Effect of Number of Sources

The effect of the number of traffic sources on the overload factor is investigated here. The parameters of traffic types used in the simulation is summarized in Table 8. Each row represents one simulation case. For example, row 1 shows the source type (peak of 0.05, utilization of 20% and burst of 80) with number of sources ranging from 10 to 70. The

"Number of Sources" column shows the range of traffic sources. The "Source Traffic Type" column shows the characteristics of the traffic used in the simulation.

TABLE 8. Tolerable Overload Factor vs. Number of Sources

Number of Sources		Source Traffic Type		
from	**to**	**peak**	**utilization**	**burst**
10	70	0.05	20%	80
10	70	0.05	10%	80
5	35	0.1	20%	50
5	35	0.1	10%	50
60	70	0.05	20%	80

Figure 19 shows the fourth row of the results. In this graph, the peak rate of the traffic

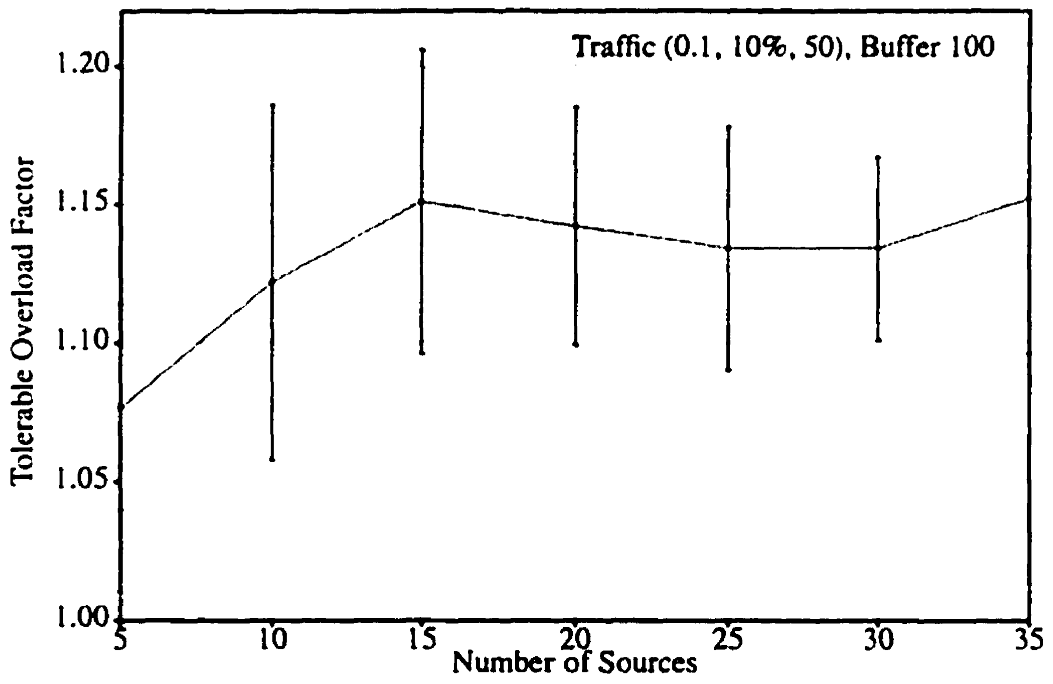
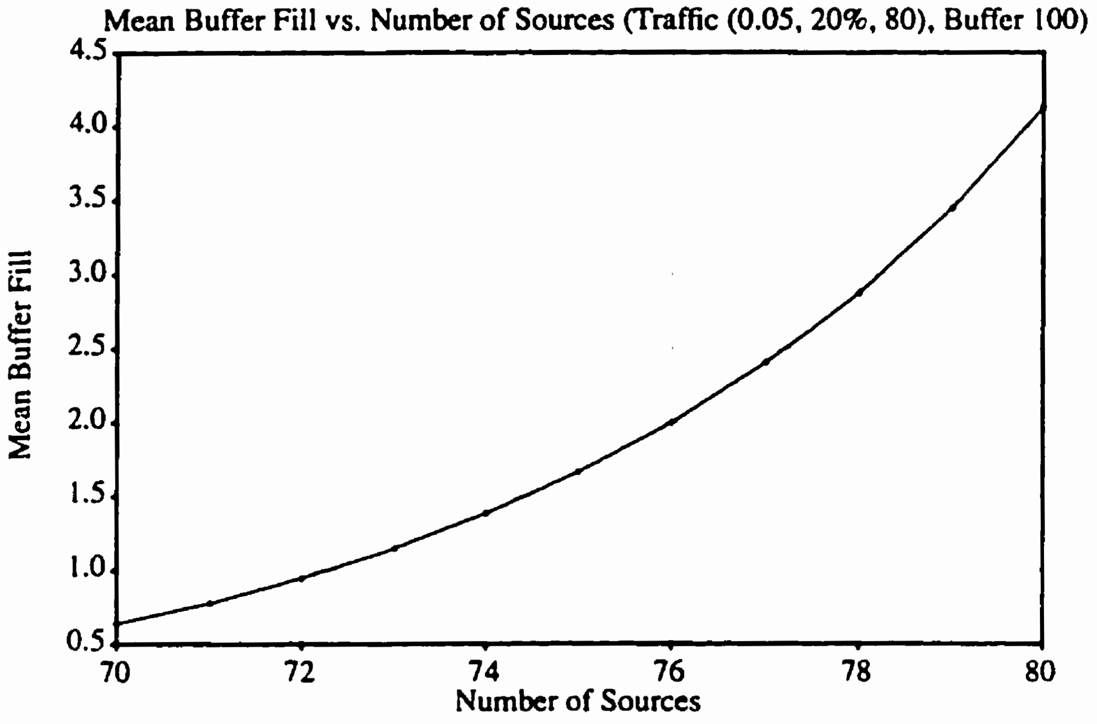


FIGURE 19. Tolerable Overload Factor versus Number of Sources

sources is 0.1, utilization is 10%, and mean burst length is 50. The buffer B is 100. In this graph, we show not only the mean value of tolerable overload factors, but also the standard deviation of the tolerable overload factors with different initial random seeds. From this example and the other four sets of results, we find that the tolerable overload factors is not significantly affected by the number of sources compared with other factors. This implies, then, that in network planning, the tolerable overload factor is independent of the number of traffic sources, and relies more on the other factors, such as source utilization and source peak rate. From our analysis, the tolerable overload is likely to decrease with a large number of sources because the aggregated traffic becomes more constant.

3.7 Buffer Fill Study

In this phase, we study the buffer fill changes during simulation with respect to the system descriptions. Because the buffer fill is needed to compute the cell delay and delay variance, we record it to show overload effects on QoS. The mean buffer fill and cell loss ratio are studied when the number of sources changes. The traffic descriptors used in this simulation are peak rate 0.05, utilization 20%, and burst length 80. The nominal number of sources are 60 and 39. The respective link capacity is 0.95 and 0.55. The numbers of sources change from 60 to 70 and from 39 to 46. Results are illustrated in Figure 20 and Figure 21 respectively. When the number of sources increases from 60 to 68 in Figure 20, the CLR increases from 10^{-7} to 10^{-4}. Meanwhile, the overload increases to about 1.11, and the mean buffer fill increases from 0.7 to 2.9, showing that the mean buffer fill increases when the overload factor increases due to more traffic induced. All traffic will suffer from a longer delay in the buffer as well as a larger cell loss ratio. This result is also



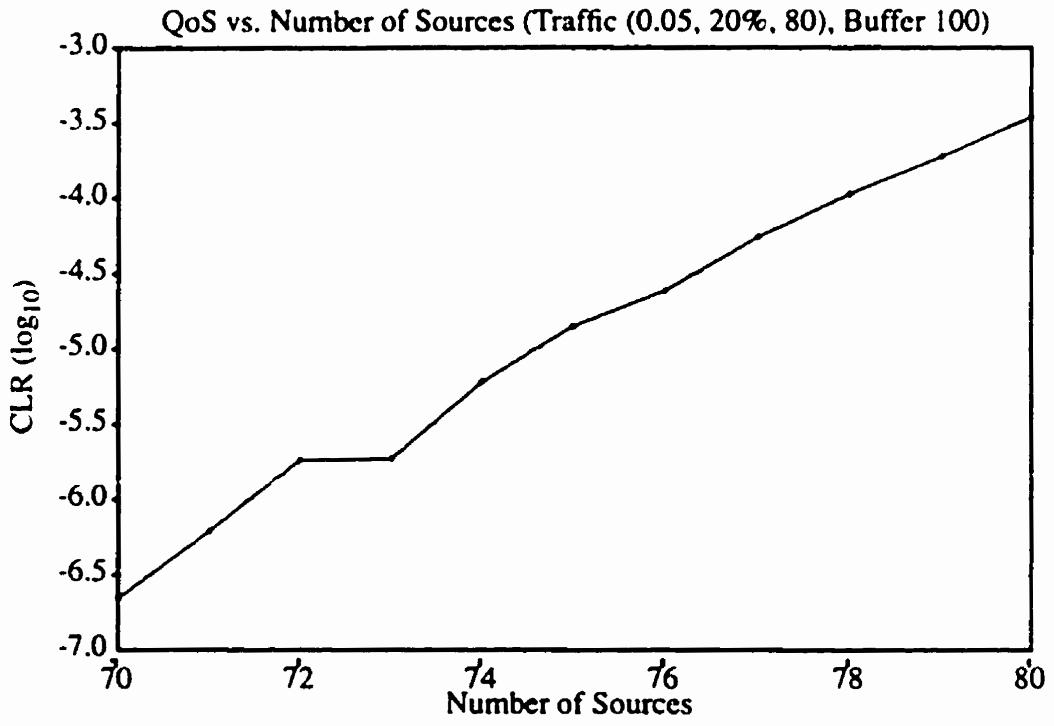
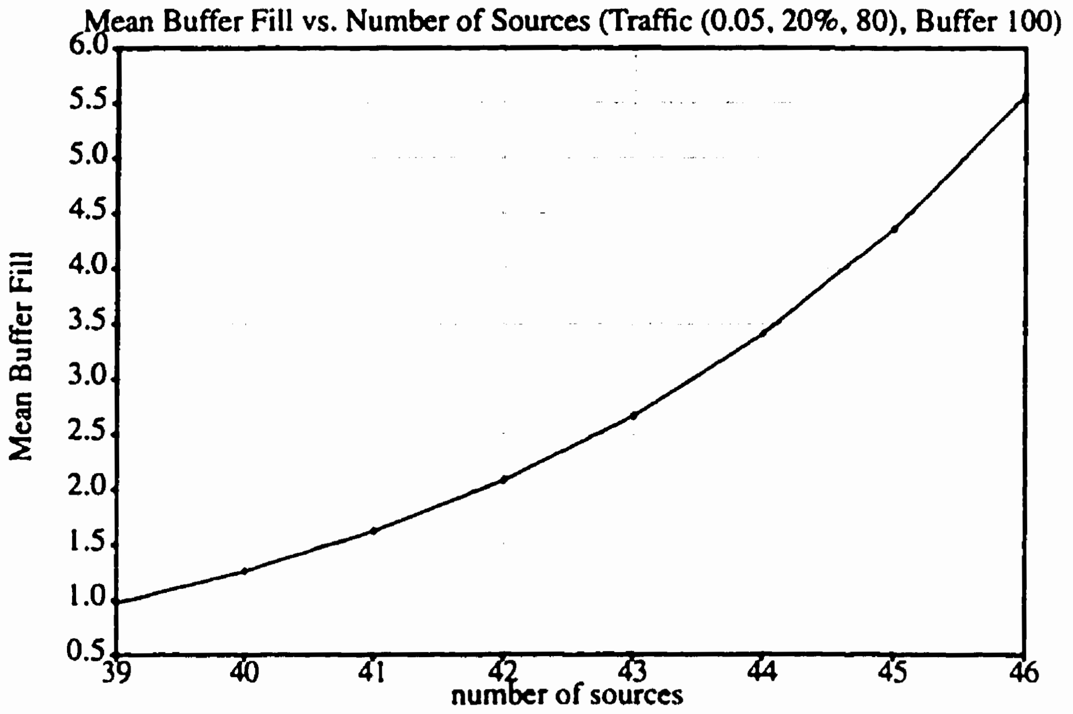


FIGURE 20. Mean Buffer fill and CLR versus Number of Sources (Traffic 1)



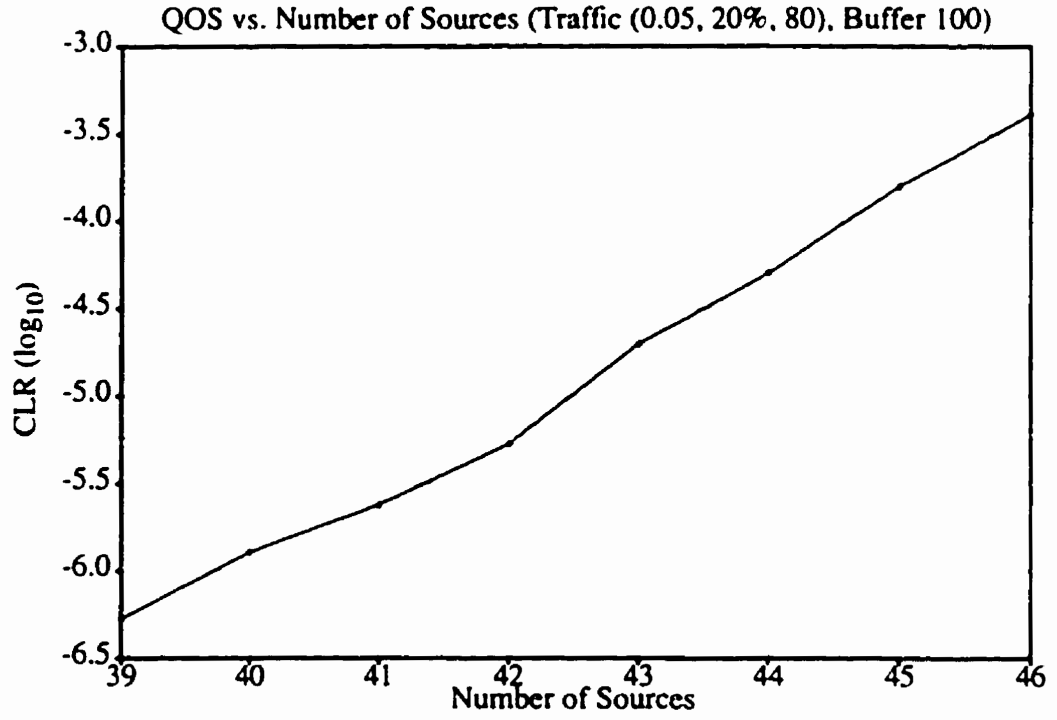


FIGURE 21. Mean Buffer fill and CLR versus Number of Sources (Traffic 2)

confirmed in Figure 21. In this figure, when the number increases from 39 to 44, the CLR increases from 10^{-6} to 10^{-4}. Meanwhile, the overload factor increases to about 1.13, and the mean buffer fill increases from 0.7 to 3.3. From these analyses, we find that the cell delay as well as the cell loss ratio increases when the overload occurs after the restoration induced flow convergence. Because the sensitivity of cell delay is different with different traffic types, appropriate methods should be used in ATM switches to decrease the delay of delay-sensitive traffic. For example, different buffer sizes may be employed for different types of traffic.

3.8 Auto-Regressive Traffic Model

In the previous study, we used the on/off fluid model to simulate the ATM traffic. To supplement this research, the *auto-regressive (AR)* model was also used to study the overload factor. A very different model from the on/off fluid model, this model, having an exponential autocovariance, is well suited to model the variable bit rate (VBR) video traffic [21].

In the AR model, within a frame *n*, traffic is generated at a constant bit rate $\lambda(n)$. A first-order autoregressive process $\lambda(n)$ can be expressed recursively as follows:

$$\lambda(n) = a\lambda(n-1) + bw(n) \tag{21}$$

where *w(n)* is a Gaussian random variable and *a* and *b* are constant coefficients. Assuming that *w(n)* has mean η and variance *1*, the bit rate of the current frame is calculated from the bit rate of the last frame adjusted by a weight and a Gaussian random variable. Assume

$|a| < 1$, and the process achieves steady state with large n [21]. The steady-state average $E(\lambda)$ and discrete autocovariance $C(n)$ are as follows [21]:

$$E(\lambda) = \frac{b}{1-a}\eta \tag{22}$$

$$C(n) = \frac{b^2}{1-a^2}a^n \tag{23}$$

This is an elegant mathematical model that has been formed to match real-world traffic. There is no physical explanation of these parameters. The values of a, b, and η are calculated by matching the average bit rate $E(\lambda)$ and discrete autocovariance $C(n)$ measured from real world traffic. For the experimental data in [21], we have the following values:

$$a \approx 0.8781 \qquad b \approx 0.1108 \qquad \eta \approx 0.572 \tag{24}$$

$$E(\lambda) = 0.52 \qquad C(n) = 0.0536 \times \left(e^{-0.13}\right)^n \tag{25}$$

Next, we investigate the impact of the traffic source number on the overload factor. The number of sources is changed from 60 to 70 and 10 to 20, as recorded in Figure 22 and Figure 23. In this graph, we give the mean value and standard deviation of the overload factors across with different seeds. We find the tolerable overload factor is around 1.015 and 1.025 respectively, regardless of the number of sources.

3.9 Guidelines for Tolerable Overload Factor

Determining the tolerable overload factor is a very complicated issue in ATM network restoration. The traffic characteristics in a real network at the time of restoration will affect

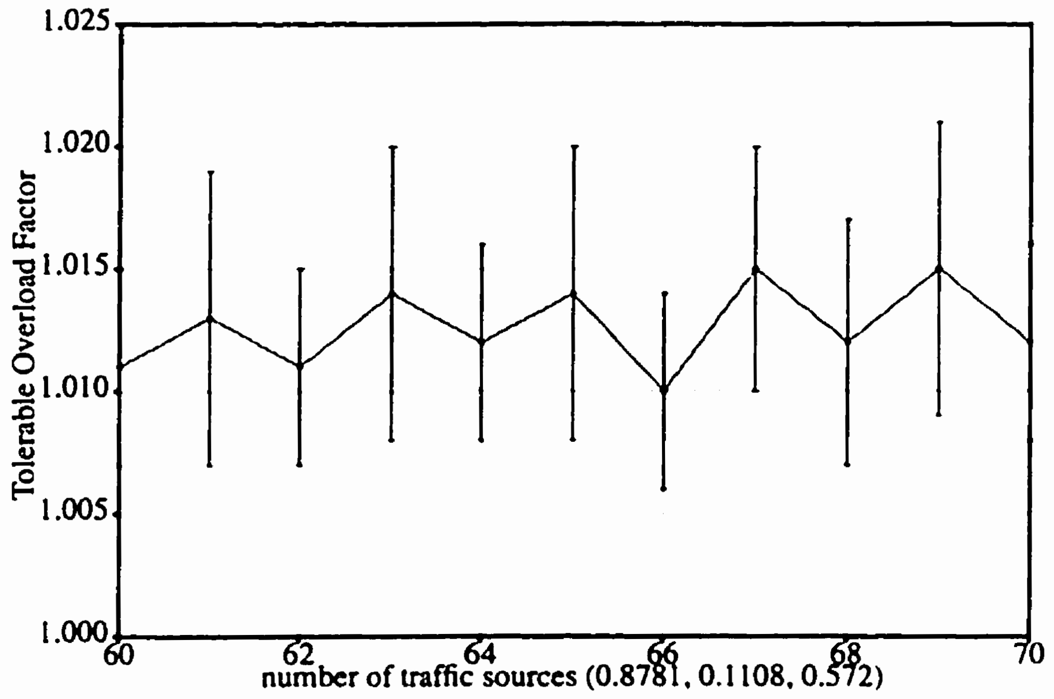


FIGURE 22. Overload Factor versus Number of Sources for AR Traffic Model (with buffer size of 1000)

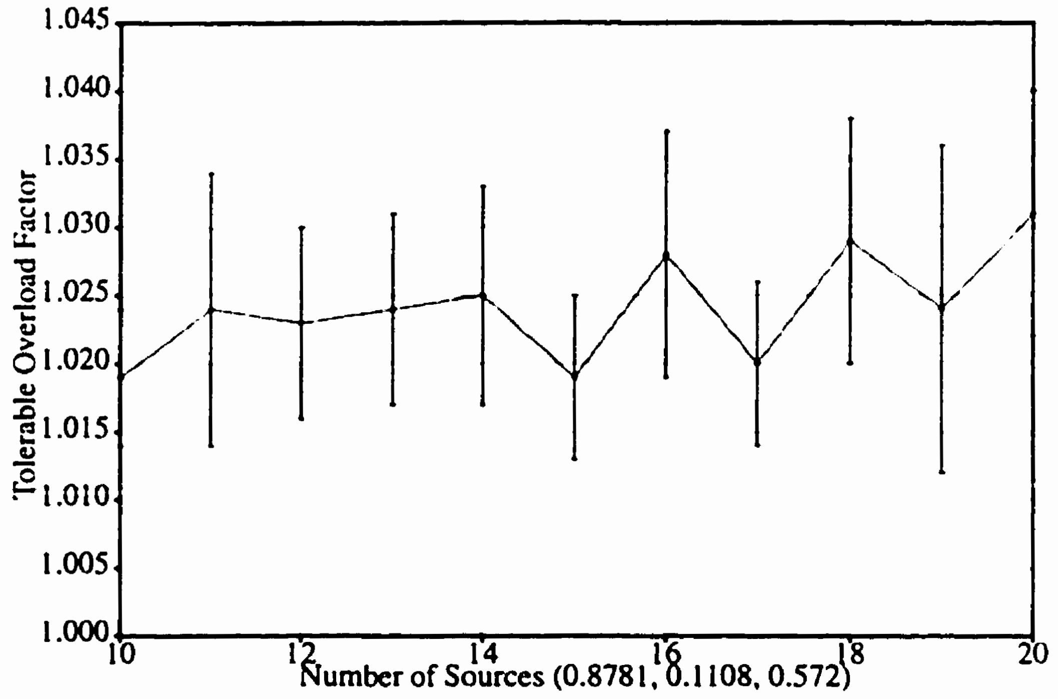


FIGURE 23. Overload Factor versus Number of Sources for AR Traffic Model (with buffer size of 1000)

the tolerable overload factor. From extensive simulations, however, general guidelines of a realistic expectation for a tolerable design overload factor can be determined, a guideline that would not degrade the QoS significantly (i.e., if CLR was initially less than 10^{-9}, it would not rise to more than 10^{-5}).

Among the factors, the traffic source model is the most important. For *Constant Bit Rate* (CBR) traffic (e.g., uncompressed voice traffic), the tolerable overload factor is almost 1.0. When the traffic is more bursty, the tolerable overload becomes larger, because more traffic can be induced in the idle periods. For AR model based *Variable Bit Rate* (VBR) traffic, the factor is approximately 1.01 to 1.03, while in on-off fluid model based VBR traffic, the factor is approximately 1.05 to 1.15. Even for the same type of VBR traffic, the detailed traffic characteristics can greatly affect the overload factor. The following table summarizes the simulation results.

TABLE 9. Tolerable Overload Assessment Factors

Factor Trend		**Tolerable Overload Trend**
Peak Bit Rate	Δ	Δ
Utilization	Δ	∇
Burst Length	Δ	Δ
Buffer Size	Δ	∇
Number of Sources	Δ	–

With the increase of peak bit rate and burst length, the tolerable overload becomes larger. With the increase of utilization and buffer size, the overload factor decreases. The number of sources does not significantly affect the tolerable overload. From our analysis, the tolerable overload is likely to decrease with a larger number of sources because overall traffic becomes more constant.

It is anticipated that for *Unspecified Bit Rate* (UBR) and *Available Bit Rate* (ABR) traffic, the tolerable overload factor can be very large because the UBR/ABR traffic is designed to increase network utilization. UBR/ABR traffic sources are designed to be very adaptive to the network load. If the network is not busy, UBR/ABR traffic sources generate more traffic into the network. If there is failure and network becomes congested, UBR/ABR sources decrease the traffic generation speed accordingly. Only minimum performance levels are guaranteed for UBR traffic and nothing is guaranteed for ABR traffic [22], therefore they can be expected to tolerate large overload.

It should be emphasized that this discussion only gives a general range for factors of tolerable overload. The practical value of an overload factor for a particular network can not always be determined exactly. Network operators must study the traffic nature in their networks comprehensively, and carry out extensive simulations to get a practical tolerable overload factor for their particular network. As a general guideline, an overload factor of 1.1 is acceptable.

3.10 Conclusion

In this chapter, a general guideline of the tolerable overload assessment was given and the concept of Equivalent Bandwidth in Call Admission Control was used. To assess the overload tolerance of a group of traffic sources, the equivalent bandwidth of the traffic must be obtained. Then, if this traffic is carried on a link whose capacity is smaller than its equivalent bandwidth, the resulting QoS inevitably degrades. The smaller the link capacity, the more severe the QoS degradation. The overload obtained by dividing equivalent bandwidth by the link capacity at the critical QoS level is regarded as the tolerable overload for this group of traffic. In our simulation, "tolerable" is defined as the cases when CLR goes from 10^{-9} to 10^{-5}.

Extensive simulations were done to analyze how traffic and network factors affect the tolerable overload. For CBR traffic, any overload is almost unacceptable. While at the other extreme, UBR/ABR traffic has a large overload factor. Overload for VBR traffic is found between the two extremes. For on/off traffic model based traffic, the tolerable overload factor is approximately 1.05 to 1.15 depending on the traffic characteristics, while for Auto-Regressive model based traffic, the value is approximately 1.02.

Network operators should analyze the traffic nature of their network comprehensively and do extensive simulations to determine the practical tolerable overload factor for their particular networks.

Chapter 4. Concluding Discussion

4.1 Comparative Overview of ATM and STM Restoration Designs

ATM VP-based restoration is inherently similar to STM path-restoration if the ATM network design case is approached on a pure VP replacement bandwidth basis. For any given network, two steps are involved in the design of backup path for STM and ATM based restoration. These are working Path/VP design and backup Path/VP design.

In the working VP design, VPs can have equal or unequal bandwidth and each demand pair can have one/multiple VPs. In the second step, each working VP can either have single equal-sized backup VP or can split the VP over multiple routes. Also, the bandwidth replacement scheme can be perfect or more aggressive where the oversubscription factor is greater than 1.0.

With the combination of these factors, we get the comparative overview of ATM versus STM design cases in Table 10. In cases 1 and 2, each demand pair has one working VP with a bandwidth replacement that is perfect. These two cases are equivalent to the STM path restoration with stub release. In case 3, the bandwidth for each VP is variable and a backup VP cannot be split to several routes. This is equivalent to STM path restoration with single backup route constraint. In case 4, each demand has several working routes. This is equivalent to having multiple pseudo-demand pairs for each demand pair in STM path restoration, i.e., one demand divided in several working routes are treated as several demands in restoration. In case 5, bandwidth is not perfectly replaced and a flow oversubscription is allowed.

TABLE 10. Comparative Overview of ATM versus STM Capacity Design Cases

Case	ATM Restoration Model				Relation to STM Capacity Formulation
	Band-width per VP	# Working Routes per Demand Pair	Working VP's Splitable for Restoration	Perfect Bandwidth Replace-ment	
1	constant	one	no	yes	STM path restoration with stub release
2	variable	one	yes	yes	
3	variable	one	no	yes	STM with stub release plus single backup route constraint
4	variable	multiple	no	yes	STM with stub release, and single backup VP route & multiple "pseudo-demand" pairs
5	- any of above -			no	as per above with flow oversubscription con-straint

4.2 ATM Restoration Design Methodology

Designing for controlled convergence of restoration flows is a proposed approach which would let the network planner mediate a controlled trade-off of temporary post-restoration ATM performance for significantly reduced network capacity. The benefit of the proposed design framework is that it allows a network operator to first determine an acceptable restoration stress level and then to design exactly for that grade of restoration performance with a known minimum of total capacity for restoration. This design approach contributes to recognizing and enabling the exploitation of the intrinsic differences between ATM and STM transport methods from a restoration viewpoint.

In addition, when designing a network with acceptable restoration-induced oversubscription of bandwidth, the potential reduction in QoS could be minimized by a restoration oriented priority congestion control scheme. In this approach, the network spare capacity design could be based on a reasonably aggressive X_{tol} value to obtain significant capacity savings; then, at the time of an actual failure, each surviving span would assess its actual cell-level utilization after allowing enough time for backup VP switching. In case utilizations were high, ATM switches would mark the lower priority VPs traversing them with a throttling indication to be acted upon either by either the VP sources themselves or neighboring switches. This gives several attractive properties: despite the number of logical VPs traversing the span after restoration, all VPs will inherently enjoy transparent continuation of service if actual conditions permit restoration of all VPs. On the other hand, if the net cell-level utilization does constitute a sufficient overload, then priority VPs can be restored selectively without QoS reduction by throttling lower priority service class VPs. In this way the benefits of ATM capacity design to exploit restoration-induced oversubscription of bandwidth can be pursued with a protective mechanism to ensure QoS for selected services while still granting all services restoration on a best-effort basis whenever actual network circumstances permit.

Based on this study, the proposed framework for ATM backup VP capacity design would allow network operators to determine both the traffic assumptions they wish to adopt and the acceptable QoS impacts during an assumed busy-hour restoration event. Through

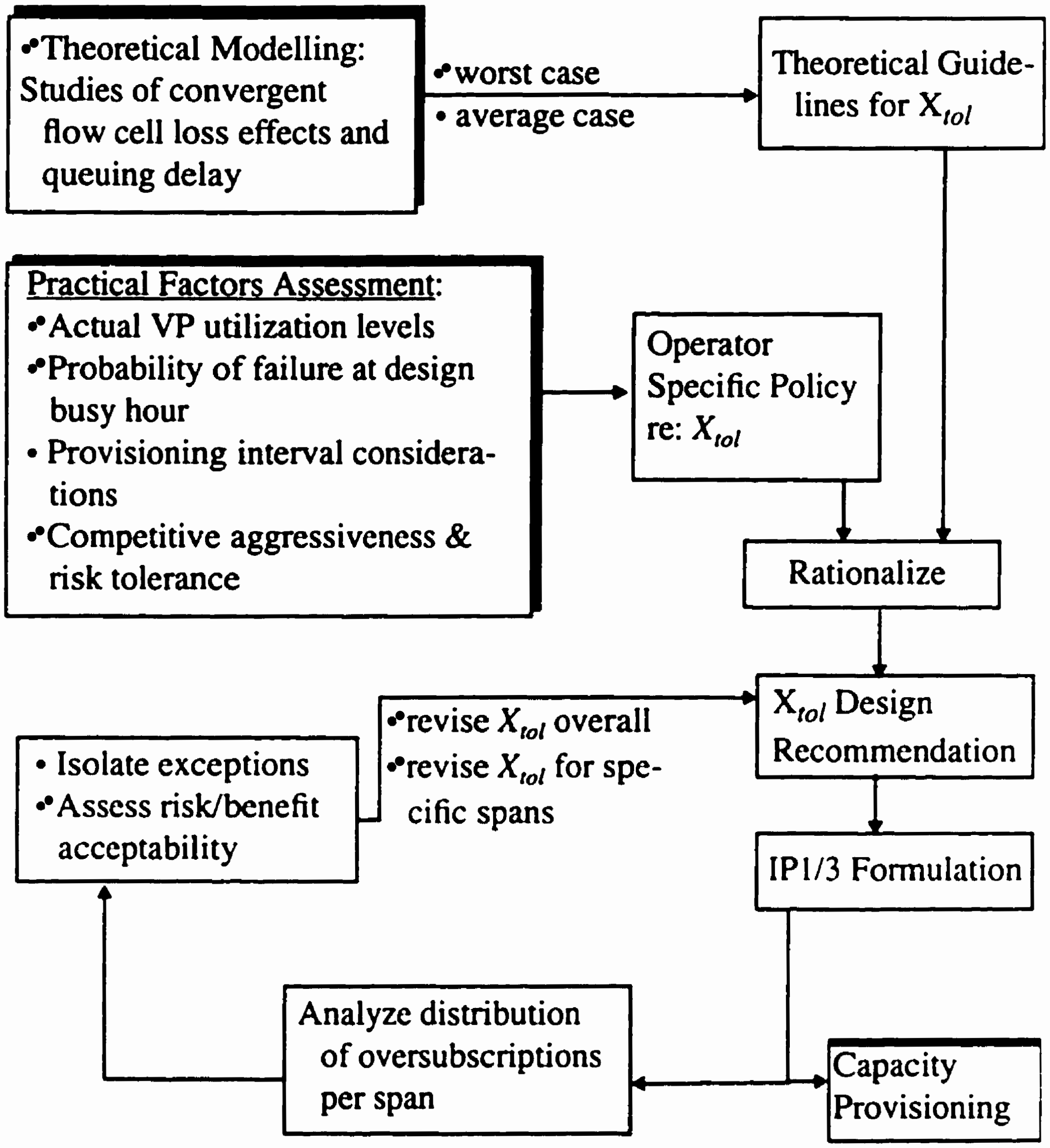

•Theoretical Modelling: Studies of convergent flow cell loss effects and queuing delay
•worst case
• average case
Theoretical Guide-lines for X_{tol}
Practical Factors Assessment:
•Actual VP utilization levels
•Probability of failure at design busy hour
• Provisioning interval considera-tions
•Competitive aggressiveness & risk tolerance
Operator Specific Policy re: X_{tol}
Rationalize
X_{tol} Design Recommendation
•revise X_{tol} overall
•revise X_{tol} for spe-cific spans
• Isolate exceptions
•Assess risk/benefit acceptability
IP1/3 Formulation
Analyze distribution of oversubscriptions per span
Capacity Provisioning

FIGURE 24. ATM Restoration Design Methodology

theoretical study of the queuing model, a guideline of X_{tol} should first be determined. In addition, some practical operation considerations contributing to X_{tol} should be investigated. These include actual VP utilization levels, probability of failure at design busy hour, provisioning interval considerations, competitive aggressiveness and risk tolerance. For example, if busy hour effects are not coincident in the network (i.e., the VP utilization are not simultaneously at their peak), the network can tolerate a large oversubscription. All these factors lead to an X_{tol} recommendation. Once X_{tol} is determined, *IP-1/3* can realize the corresponding minimum capacity restorable network. By analyzing the actual oversubscription cases in the network in detail, the exact X_{tol} can be determined. This new value is once again put in the *IP-1/3* formulation. Through iterating several times, we finally achieve a satisfied network specific capacity provisioning. The overall process is illustrated in Figure 24.

If the capacity placement has been given in a network, IP-2 can be used to optimize the restoration VP routing to achieve the minimum peak oversubscription in the network.

4.3 Summary

In chapter 2, the restoration induced flow convergence oversubscription factor was quantitatively defined based on the logical view of the traffic characteristics in backup VP based ATM restoration. An allowable oversubscription larger than 1.0 is one of the unique properties in ATM networks. Next a heuristic algorithm proposed in a literature was implemented to verify the severe restoration induced oversubscription if we do not consider the design carefully. The implications of uncontrolled oversubscription will likely be unacceptable in practise. Therefore, we formulate optimal capacity allocation methods that will still gain as much ATM-related capacity savings as safely possible by giving us a controlled input on the maximum extent of the restoration-induced oversubscription. The first IP formulation optimizes the spare capacity placement of a restorable ATM network given a peak allowable oversubscription factor in the network. The second formulation applies to the case where an existing set of spare capacity allocations has been given. The problem is to find a set of backup VP allocations that results in the smallest maximum oversubscription factor. The third formulation tries to minimize the total capacity (working + sparing) of a restorable ATM network with a given design peak oversubscription factor. In addition to these three IP formulations, two simpler algorithms were also presented to calculate reasonably tight upper and lower bounds on the required spare capacity of a backup VP-based restorable ATM network. Our results showed that the total spare capacity decreases rather quickly as the design tolerance for restoration-induced oversubscription increases. This suggests that significant capacity savings can be obtained relative to STM if ATM restoration is allowed even modest restoration-induced oversubscription of bandwidth on surviving spans.

Tolerable oversubscription factors depend on several considerations. Partly it is a network operation policy. An aggressive network operator can use a large oversubscription to save more valuable capacity in the network. In chapter 3, the theoretical study of a queuing model of traffic gave a guideline for the oversubscription factor. Extensive simulations were done to analyze how traffic and network factors affect the tolerable overload. For CBR traffic, the overload is almost intolerable. On the other extreme, ABR traffic has a large overload factor, while the overload for VBR traffic is between CBR and ARB traffic tolerable overloads. For on/off model based traffic, the factor is approximately 1.05 to 1.15 depending on the traffic characteristics, while for auto-regressive model based traffic, the tolerable overload is approximately 1.02.

It is highly recommended that network operators analyze the nature of their traffic comprehensively and do extensive simulations to determine what is the practical tolerable overload value in their network.

In chapter 4, we gave a new framework for ATM backup VP capacity design. Network operators would determine the traffic assumptions they wish to adopt and the acceptable QoS impacts during an assumed busy-hour restoration event. This leads to a tolerable oversubscription factor recommendation. Integer programs can realize the corresponding minimum capacity restorable network. By analyzing the actual oversubscription cases in the network in detail, we can revise the tolerable oversubscription factor recommendation. This new value is put in the IP formulation again. We can achieve a satisfactory network specific capacity provisioning by iterating several times as shown in Figure 24.

4.4 Future Work

In this research, we studied the spare capacity placement problem in ATM restorable networks. We found that a significant amount of spare capacity can be saved if even a modest level of oversubscription is allowed in ATM restorable networks. This work shows one of the basic distinctions between ATM and STM restorable networks: oversubscription can be larger than 1.0 in ATM networks. As we know, ATM networks are more complicated than STM networks. ATM restoration requires further research to determine how the restoration mechanism is implemented using OAM cells/messages to detect, notify, and re-route failed VP. Another area of research requires using real-time simulation to study the effect of failed VP traffic on working VPs. The question of whether restoration should be done of the STM OC level or the ATM VP level to achieve the largest benefit in the network planning is must also be answered.

Bibliography

[1] W.D. Grover, "Distributed Restoration of the Transport Network", *Book Chapter 11, Telecommunications Network Management into the 21st Century: Techniques, Standards, Technologies and Applications*, IEEE Press, 1994, pp. 337-417.

[2] W.D. Grover, "The Self-healing Network: A fast distributed restoration technique for networks using digital cross-connect machines", *Proceeding of IEEE Global Communications Conference*, 1987, pp. 1090-1095.

[3] T.H. Wu, "Emerging Technologies for Fiber Network Survivability", *IEEE communication magazine*, February, 1995, pp. 58-74.

[4] R. Kawamura, K. Sato, and I. Tokizawa, "Self-healing ATM Networks Based on Virtual Path Concept", *IEEE Journal on selected areas in communications*. Vol. 12, No. 1, 1994, pp. 120-127.

[5] C. Partridge, "Gigabit Networking", Addison-Wesley, 1994.

[6] Y. Zheng, W.D. Grover, M. MacGregor, "Dependence of network capacity requirements on the allowable flow convergence overloads in ATM backup VP restoration", Electronics Letters, Vol. 33, No. 5, February 27, 1997. pp.362-363.

[7] Y. Xiong, L. Mason, "Restoration strategies and spare capacity requirements in self-healing ATM Networks", *Infocom 97*, Kobe, Japan, April, 1997.

[8] R. R. Iraschko, M.H. MacGregor, W.D.Grover, "Optimal Capacity Placement for Path Restoration in Mesh Survivable Networks", *IEEE ICC'96*, June 1996, pp.1568-1574.

[9] W.D. Grover, V. Rawat, M. MacGregor, "A Fast Heuristic Principle for Spare Capacity Placement in Mesh-Restorable SONET / SDH Transport Networks", *Electronics Letters*, Vol. 33, No. 3, January 30, 1997. pp.195-196.

[10] Y. Zheng, W.D. Grover, M. MacGregor, "Broadband Network Design with Controlled Exploitation of Flow Convergence Overloads in ATM VP-based Restoration", Canadian Conference on Broadband Research, April 16-17, 1997, Ottawa, Canada.

[11] H.G. Perros, K.M.Elsayed, "Call Admission Control Schemes: A Review", *IEEE Communication Magazine*, November, 1996, pp. 82-91.

[12] R. Guerin, H. Ahmadi, M. Naghshineh, "Equivalent Capacity and its Application to Bandwidth Allocation in High-Speed Networks", *IEEE JSAC*, November, 1991, pp. 968-81.

[13] R.O. Onvural, "Asynchronous Transfer Mode Networks, Performance Issues", *Artech House*, 1994, pp. 119-33.

[14] G. Gallassi, G. Rigolio, L. Verri, "Resource Management and Dimensioning in ATM Networks", *IEEE Network Magazine*, May, 1990, pp. 8-17.

[15] F. Vakil, H, Saito, "On Congestion Control in ATM Networks", *IEEE LTS*, August, 1991, pp. 55-65.

[16] J.W. Roberts, "Variable-Bit-Rate Traffic Control in B-ISDN", *IEEE Communications Magazine*, September, 1991, pp. 50-56.

[17] C.A. Cooper, K.I. Park, "Toward a Broadband Congestion Control Strategy", *IEEE Network Magazine*, May, 1990, pp. 18-23.

[18] H. Saito, "Call Admission Control in an ATM Network Using Upper Bound of Cell Loss Probability", *IEEE Transactions on Communications*, vol. 40, 1992, pp. 1512-21.

[19] D. Anick, D. Mitra, M.M. Sondhi, "Stochastic Theory of a Data-Handling System with Multiple Sources", *Bell Sys. Tech. J.* vol. 61, 1982, pp. 1871-94.

[20] M. Decina and T. Toniatti, "On Bandwidth Allocation to Bursty Virtual Connections in ATM Networks", *Proc. ICC'90*, paper 318.6.1, pp. 844-851.

[21] B. Maglaris, D. Anastassiou, P. Sen, G. Karlsson, and J. Robbins, "Performance Models of Statistical Multiplexing in Packet Video Communications", *IEEE Transactions on Communications*, Vol. 36, No. 7, July, 198, pp. 834-844.

[22] K. Schulz, M. Incollingo, and H. Uhrig, "Taking Advantage of ATM Services and Tariffs: The Importance of Transport Layer Dynamic Rate Adaptation", *IEEE Network*, Vol. 11, No. 2, March/April, 1997, pp. 10-17.

[23] "CPLEX Manual", CPLEX Optimization Corp. 1995

Appendix A: Abstract of Equivalent Bandwidth Calculation

The following abstract of equivalent bandwidth calculation is based on the analysis in [19].

Suppose there are N ***mutually independent identical sources.*** The unit of time is selected to be the average "on" period. Within this unit of time, the average "off" period is denoted by $1/\lambda$.

Let $P_i(t, x)$, $(0 \le i \le N, t \ge 0, x \ge 0)$ denote the stationary probability that at time t, i sources are on and the buffer content does not exceed x. If at time t the number of on sources equals i, two elementary events can take place during the next interval Δt, i.e., a source can turn on or turn off. Since the "on" and "off" periods are exponentially distributed, the probabilities are $(N-i)\lambda\Delta t$ and $i\Delta t$ respectively. Compound events have probabilities $O\left(\Delta t^2\right)$. The probability of no change is $1 - \{(N-i)\lambda + i\}\Delta t + O\left(\Delta t^2\right)$.

Now,

$$\begin{aligned} P_i(t+\Delta t, x) = {} & \{N-(i-1)\}\lambda\Delta t P_{i-1}(t,x) + (i+1)\Delta t P_{i+1}(t,x) \\ & [1 - \{(N-i)\lambda + i\}\Delta t] P_i(t, x-(i-c)\Delta t) + O\left(\Delta t^2\right) \end{aligned} \tag{26}$$

Passing to the limit $\Delta t \to 0$, yields the following partial differential equations:

$$\frac{\partial P_i}{\partial t} + (i-c)\frac{\partial P_i}{\partial x} = \{N-i+1\}\lambda P_{i-1} - \{(N-i)\lambda + i\}P_i + (i+1)P_{i+1} \tag{27}$$

Let $F_i(x)$ be the equilibrium probability that i sources are on and buffer content does not exceed x,

$$F_i(x) = \lim_{t \to \infty} P_i(t, x) \tag{28}$$

Therefore, we obtain, for $i \in [0, N]$,

$$(i-c)\frac{\partial F_i}{\partial x} = (N-i+1)\lambda F_{i-1} - \{(N-i)\lambda + i\} F_i + (i+1) F_{i+1} \tag{29}$$

Equation 29 can be rewritten in matrix notation as:

$$D\frac{\partial}{\partial x}F(x) = MF(x) \qquad x \geq 0 \tag{30}$$

where $D = diag\{-c, 1-c, 2-c, \ldots, N-c\}$ and

$$M = \begin{bmatrix} -N\lambda & 1 & & & \\ N\lambda & -\{(N-1)\lambda+1\} & 2 & & \\ & (N-1)\lambda & -\{(N-2)\lambda+2\} & 3 & \\ & & & & \\ & & & 2\lambda \quad -(\lambda+(N-1)) & N \\ & & & \lambda & -N \end{bmatrix}$$

By solving Equation 30, we get

$$F(x) = \sum_{i=0}^{N} a_i \Phi_i e^{z_i x} \tag{31}$$

where the z_i and Φ_i are, respectively, generalized eigenvalues and eigenvectors associated with the solution of the differential equation satisfied by the stationary probabilities of the system, and the a_i's are coefficients determined from boundary conditions.

Let

$$G(x) \equiv Pr(buffercontent > x) = 1 - \sum_{i=0}^{N} F(x) \qquad (32)$$

We refer $G(x)$ as the probability of overflow beyond x.

The distribution of $F(x)$ is completely determined from the values of the associated eigenvalues, eigenvectors, and corresponding coefficients. There are no explicit expressions for these quantities, which must then be determined numerically.

An important aspect of this problem is numerical stability. The inevitable errors, no matter how small, incurred during numerical integration are liable to excite the unstable modes and lead to solutions that blow up.

Appendix B: Test Networks Topology and Demand Matrics Files

In this appendix, the network topology and demand matrics of all seven tested networks are listed. The topology files are in SNIF format. The detail description of SNIF format can be found in TRLabs internal technical report. Briefly, it includes four description lines starting with: Date, File Name, Network, Program. Then the positions of network nodes are listed. Finally is the span description, which includes span tag, end nodes, distance, working capacity and spare capacity of the span. The demand matrix is relatively simple. It first indicates the number of demands and then list each demand between two node pair.

- **SmallNet SNIF file**

```
Date: Jan 30, 1995
File Name: SmallNet.sniff
Network: SmallNet
Program:
#
Node       Xcoord     Ycoord
0          -1         1
1          1          1
2          -1.5       0
3          -.5        .5
4          .5         .5
5          1.5        0
6          -.5        -.5
7          .5         -.5
8          -1         -1
9          1          -1
#
Span       NodeA      NodeB      Distance Working Spare
1          0          1          1          8          0
2          0          2          1          7          0
3          0          3          1          7          0
4          1          3          1          5          0
5          1          4          1          6          0
6          1          5          1          9          0
7          2          3          1          7          0
8          3          6          1          4          0
9          3          7          1          9          0
10         3          4          1          6          0
11         4          6          1          7          0
12         4          7          1          5          0
13         4          5          1          6          0
14         2          8          1          10         0
15         2          6          1          4          0
16         6          8          1          5          0
17         6          7          1          4          0
18         7          8          1          7          0
19         7          9          1          7          0
20         5          7          1          6          0
21         8          9          1          6          0
22         5          9          1          7          0
```

- **SmallNet Demand File**

The demand matrix has 45 demands. One unit of demand exists between every node pair uniformly.

- **Net1 Snif File**

```
Date:
File Name: bellcore.sniff
Network: bellcore
Program:

Node    Xcoord          Ycoord
0       35              84
1       76              77
2       42              76
3       56              69
4       57              27
5       76              51
6       44              54
7       73              36
8       44              43
9       28              44
10      17              33
11      73              18
12      39              18
13      76              62
14      27              63

Span    NodeA   NodeB   DistanceWorking Spare
1       0       1       9               24          0
2       0       2       6               8       0
3       0       14      21              12      0
4       1       2       14              32      0
5       1       3       6               40      0
6       1       13      11              48      0
7       2       14      16              16      0
8       3       13      8               24      0
9       3       14      11              48      0
10      4       7       7               28      0
11      4       11      7               0       0
12      5       6       5               8       0
13      5       7       7               110     0
14      5       13      10              74      0
15      6       7       11              4       0
16      6       8       11              8       0
17      6       9       11              58      0
18      6       13      8               58      0
19      7       8       1               124     0
20      7       10      7               250     0
21      7       11      13              44      0
22      9       10      5               262     0
23      9       12      23              0       0
24      9       14      10              168     0
25      10      12      13              84      0
26      10      14      19              0       0
27      11      12      8               8       0
28      13      14      11              48      0
```

- **Net1 Demand File**

```
67
0       1       12
0       2       8
0       4       4
0       6       4
0       10      4
0       13      4
0       14      8
1       2       16
1       3       12
```

```
1         4         4
1         5         4
1         6         8
1         8         4
1         9         4
1         10        4
1         13        8
1         14        12
2         3         8
2         6         4
2         10        4
2         13        4
2         14        12
3         5         4
3         6         4
3         7         4
3         8         4
3         9         4
3         10        8
3         13        8
3         14        16
4         9         12
4         10        8
5         6         8
5         7         8
5         10        52
5         11        4
5         12        4
5         13        8
5         14        16
6         8         8
6         9         8
6         10        32
6         11        4
6         12        4
6         13        8
6         14        16
7         10        24
7         12        16
8         9         12
8         10        40
8         11        8
8         12        16
8         13        12
8         14        28
9         10        40
9         11        4
9         12        4
9         13        4
9         14        16
10        11        20
10        12        36
10        13        20
10        14        88
11        12        8
11        14        4
12        14        4
13        14        16
```

- **Net2 Snif File**

```
Date: May 27, 1994
File Name: Telus.nwk
Network: netF
Program:
#
Node      Xcoord    Ycoord
0         100       100
1         0         0
2         0         80
3         50        80
4         70        80
5         0         100
6         0         20
7         20        50
8         100       0
9         30        80
10        70        100
11        0         60
```

```
12        20        25
13        60        60
14        0         40
15        40        60
16        50        0
17        75        25
18        20        0
19        100       60
#
Span      NodeA     NodeB     Distance  Working  Spare
1         0          4         80       264         0
2         0          10        110      14          0
3         0          19        45       123         0
4         1          6         40       19          0
5         1          12        65       108         0
6         1          18        64       11          0
7         2          5         95       110         0
8         2          9         83       219         0
9         3          4         30       41          0
10        3          9         23       204         0
11        3          10        76       66          0
12        4          10        76       70          0
13        4          13        26       407         0
14        5          10        118      16          0
15        6          14        30       19          0
16        7          11        51       39          0
17        7          12        72       141         0
18        7          15        20       179         0
19        8          13        76       159         0
20        8          17        76       44          0
21        8          19        95       44          0
22        9          13        30       299         0
23        9          15        20       146         0
24        11         15        33       204         0
25        12         13        110      423         0
26        12         16        60       178         0
27        12         17        108      26          0
28        13         15        20       334         0
29        13         17        122      116         0
30        14         17        80       19          0
31        16         18        65       70          0
```

- **Net2 Demand File**

```
153
0         1         3
0         2         3
0         3         3
0         4         26
0         5         2
0         7         3
0         8         6
0         9         4
0         10        3
0         11        5
0         12        5
0         13        102
0         15        4
0         16        4
0         17        4
0         18        2
0         19        4
1         2         3
1         3         3
1         4         9
1         5         3
1         7         2
1         8         4
1         9         2
1         10        3
1         11        5
1         12        23
1         13        46
1         15        3
1         16        3
1         17        2
1         18        2
1         19        4
2         3         8
```

2	4	4
2	5	11
2	7	2
2	8	4
2	9	9
2	10	5
2	11	6
2	12	6
2	13	55
2	15	4
2	16	3
2	17	2
2	18	2
2	19	4
3	4	12
3	5	5
3	7	2
3	8	4
3	9	14
3	10	22
3	11	8
3	12	7
3	13	85
3	15	5
3	16	3
3	17	2
3	18	2
3	19	4
4	5	2
4	7	3
4	8	7
4	9	4
4	10	6
4	11	8
4	12	7
4	13	84
4	15	4
4	16	4
4	17	2
4	18	2
4	19	2
5	7	2
5	8	8
5	9	4
5	10	4
5	11	10
5	12	4
5	13	58
5	15	3
5	16	3
5	17	2
5	18	2
5	19	3
7	8	4
7	9	4
7	10	4
7	11	8
7	12	14
7	13	53
7	15	7
7	16	2
7	17	3
7	18	2
7	19	2
8	9	4
8	10	5
8	11	8
8	12	10
8	13	110
8	15	5
8	16	4
8	17	5
8	18	3
8	19	20
9	10	5
9	11	8
9	12	8
9	13	72
9	15	8
9	16	2

```
9          17         3
9          18         2
9          19         3
10         11         6
10         12         7
10         13         54
10         15         4
10         16         3
10         17         3
10         18         2
10         19         6
11         12         13
11         13         99
11         15         27
11         16         8
11         17         5
11         18         5
11         19         14
12         13         251
12         15         10
12         16         19
12         17         9
12         18         7
12         19         8
13         15         107
13         16         41
13         17         91
13         18         37
13         19         55
15         16         4
15         17         3
15         18         3
15         19         4
16         17         3
16         18         2
16         19         4
17         18         2
17         19         6
18         19         4
```

- **Net3 Snif File**

```
Date: June 23, 1994
File Name: British_long_haul.snif
Network: British Telecom study network
Program: None

Node      Xcoord   Ycoord
0         27662 58733
1         36839 59453
2         40193 51952
3         59028 51473
4         45300 48681
5         36839 45327
6         26544 44810
7         31358 32213
8         39315 39740
9         53040 41815
10        60304 44847
11        64071 37623
12        50965 35670
13        62094 28058
14        53441 30961
15        48094 28886
16        40193 25933
17        32545 19089
18        40272 16914
19        48094 18512
20        51893 19353
21        57272 23060
22        57512 17315
23        61578 24480
24        65811 23221
25        65650 17716
26        59652 14602
27        47375 10850
28        47454 15240
29        33568 12448
```

```
Span NodeA NodeB Distance Working Spare

1 0 1 10 958 0
2 0 2 98 0 0
3 0 6 21 2016 0
4 1 3 75 813 0
5 1 2 166 0 0
6 3 10 34 1279 0
7 2 4 42 684 0
8 2 3 88 0 0
9 5 6 78 2008 0
10 6 7 56 80 0
11 2 6 45 448 0
12 4 5 50 139 0
13 5 8 50 2468 0
14 4 10 10 343 0
15 4 9 45 961 0
16 9 10 99 0 0
17 10 11 23 1254 0
18 9 11 56 50 0
19 11 12 89 0 0
20 11 13 45 1315 0
21 7 9 91 119 0
22 9 12 10 1062 0
23 8 12 10 1061 0
24 12 14 10 878 0
25 8 14 34 0 0
26 8 16 56 63 0
27 7 17 134 0 0
28 7 16 122 102 0
29 16 17 111 0 0
30 15 16 102 0 0
31 16 18 104 0 0
32 14 15 106 22 0
33 15 20 67 219 0
34 13 23 45 202 0
35 13 21 23 1759 0
36 13 14 11 955 0
37 23 24 100 11 0
38 18 19 100 4 0
39 18 27 66 56 0
40 18 29 67 27 0
41 17 18 29 167 0
42 17 29 44 191 0
43 28 29 46 224 0
44 26 27 32 90 0
45 25 26 77 7 0
46 22 27 67 205 0
47 21 24 79 142 0
48 22 25 55 188 0
49 19 28 23 117 0
50 19 20 11 763 0
51 8 17 45 668 0
52 8 19 66 1783 0
53 8 9 99 0 0
54 7 8 100 0 0
55 22 28 10 231 0
56 20 21 10 1053 0
57 24 25 34 13 0
58 21 22 30 483 0
59 20 22 40 21 0
```

- **Net3 Demand File**

```
263
0 1 114
0 2 4
0 3 3
0 4 85
0 5 139
0 6 101
0 7 2
0 8 182
0 9 2
0 10 50
0 11 5
0 12 63
0 13 37
0 14 3
```

```
0 15 2
0 16 10
0 17 37
0 18 11
0 19 350
0 20 30
0 21 4
0 22 7
0 23 2
0 24 5
0 25 3
0 26 13
0 27 11
0 28 10
0 29 7
1 2 37
1 3 63
1 4 102
1 5 142
1 6 106
1 7 27
1 8 189
1 9 3
1 10 4
1 11 6
1 12 52
1 13 8
1 14 10
1 15 47
1 16 10
1 17 14
1 18 30
1 19 441
1 20 31
1 21 11
1 23 59
1 24 20
1 25 3
1 26 30
1 28 49
1 29 81
2 3 17
2 5 5
2 6 34
2 7 10
2 8 37
2 9 6
2 10 45
2 12 5
2 17 26
2 18 59
2 20 11
2 21 8
2 22 3
2 23 6
2 25 15
2 28 5
2 29 73
3 4 37
3 5 10
3 6 38
3 8 4
3 9 28
3 10 95
3 11 39
3 12 5
3 13 43
3 14 4
3 15 82
3 17 8
3 19 7
3 21 10
3 22 60
3 23 26
3 24 55
3 26 4
3 28 30
3 29 6
4 5 117
4 6 100
```

4	8	152
4	9	17
4	11	29
4	13	94
4	14	5
4	15	8
4	16	36
4	17	6
4	19	207
4	23	21
4	24	5
4	29	8
5	6	124
5	8	200
5	9	5
5	10	7
5	12	65
5	13	24
5	16	10
5	17	61
5	19	486
5	25	6
5	28	8
6	8	155
6	9	8
6	11	8
6	16	21
6	17	15
6	19	217
6	22	13
6	23	23
6	24	8
6	27	19
7	11	31
7	12	10
7	15	8
7	16	61
7	22	11
7	25	59
8	10	63
8	11	1
8	16	9
8	19	557
8	21	55
8	22	35
8	24	27
8	28	73
9	14	1
9	15	1
9	17	28
9	18	1
9	20	14
9	21	1
9	22	1
9	24	1
9	25	1
9	26	1
9	28	1
9	29	2
10	11	28
10	12	2
10	13	1
10	14	11
10	15	54
10	16	2
10	17	2
10	19	2
10	20	1
10	21	37
10	22	23
10	23	1
10	24	2
10	26	2
11	12	31
11	13	2
11	14	46
11	15	15
11	16	2
11	19	31
11	21	2

11	23	4
11	25	57
11	26	2
11	27	2
12	13	4
12	14	1
12	15	3
12	17	36
12	18	3
12	19	1
12	20	1
12	21	1
12	22	2
12	23	22
12	25	5
12	26	3
12	29	4
13	14	76
13	17	3
13	18	4
13	19	23
13	20	4
13	21	4
13	28	1
13	29	3
14	17	4
14	18	3
14	20	6
14	21	6
14	24	2
14	27	5
15	19	1
15	20	1
15	21	1
15	24	1
15	27	7
15	29	10
16	21	2
16	22	1
16	29	3
17	18	1
17	19	82
17	20	2
17	21	3
17	22	7
17	26	2
17	27	7
18	21	2
18	22	2
18	23	1
18	24	2
18	26	4
18	28	5
18	29	20
19	20	2
19	21	15
19	23	12
19	24	1
19	25	10
20	22	1
20	24	9
20	25	10
20	26	2
20	27	1
20	28	6
21	22	2
21	23	17
21	24	4
21	25	1
21	27	28
21	28	1
22	23	7
22	24	1
22	25	2
22	26	12
22	28	2
23	24	1
23	25	10
23	27	1
24	25	1

```
24  28  1
25  27  7
25  28  9
25  29  5
26  27  7
26  28  1
27  28  8
27  29  96
28  29  96
```

- **Net4 Snif File**

```
Date:
File Name: US_Long_haul
Network:
Program:

Node      Xcoord           Ycoord
0         167.751000       43.902400
1         193.225000       67.479700
2         175.881000       55.284600
3         191.599000       95.122000
4         149.051000       94.579900
5         18.157200        99.187000
6         18.970200        49.864500
7         80.216800        85.094900
8         15.447200        7.317070
9         9.214090         139.024000
10        139.566000       173.984000
11        7.046070         78.319800
12        164.499000       63.956600
13        126.287000       37.398400
14        68.834700        176.152000
15        66.124700        75.067800
16        149.051000       117.886000
17        169.106000       102.439000
18        164.770000       121.680000
19        162.602000       80.758800
20        81.571800        127.642000
21        46.612500        127.642000
22        15.718200        18.699200
23        47.425500        106.504000
24        63.956600        100.271000
25        51.761500        85.636900
26        88.888900        189.431000
27        104.878000       66.395700
28        117.886000       78.048800
29        110.569000       150.678000
30        106.233000       85.365900
31        78.048800        71.002700
32        71.273700        189.702000
33        122.764000       60.433600
34        111.653000       37.398400
35        136.856000       59.349600
36        153.930000       151.491000
37        70.189700        43.902400
38        149.593000       132.249000
39        33.333300        173.442000
40        140.108000       188.618000
41        28.726300        82.926800
42        107.317000       173.713000
43        123.306000       189.160000
44        39.837400        6.775070
45        39.837400        48.238500
46        52.303500        176.152000
47        94.579900        66.937700
48        162.331000       7.317070
49        79.674800        109.756000
50        42.547400        75.880800
51        104.336000       49.322500
52        30.000000        45.000000

Span      NodeA     NodeB     DistanceWorking Spare
1         0         2         5.00000029       0
2         0         1         76.0000000         0
3         1         2         46.00000039        0
4         1         12        209.00000030         0
5         20        38        299.00000066         0
```

```
6          3          4          168.00000036         0
7          4          12         214.00000047         0
8          4          19         16.00000022        0
9          5          41         91.0000008         0
10         1          48         349.00000051         0
11         6          45         36.00000023        0
12         6          52         36.0000000         0
13         7          15         29.00000039        0
14         7          49         1.00000059       0
15         8          22         12.0000005         0
16         8          44         30.0000000         0
17         12         35         250.00000059         0
18         9          39         2560.00000062          0
19         10         40         114.0000004          0
20         10         43         35.0000004         0
21         36         38         1480.00000073          0
22         9          21         585.00000065         0
23         11         22         118.00000084         0
24         11         41         79.00000068        0
25         5          11         100.00000014         0
26         0          13         414.00000025         0
27         13         34         305.0000000          0
28         14         26         23.0000002         0
29         14         32         25.00000028        0
30         14         42         32.00000028        0
31         14         46         53.00000014        0
32         15         37         148.0000008          0
33         15         49         31.0000000         0
34         15         50         7.00000038       0
35         16         17         48.00000015        0
36         16         18         28.0000008         0
37         16         20         210.00000037         0
38         4          16         1011.00000041          0
39         16         38         243.00000044         0
40         17         18         20.0000002         0
41         3          19         721.0000000          0
42         10         36         79.00000037        0
43         20         21         373.00000081         0
44         13         35         35.00000021        0
45         28         49         88.00000089        0
46         22         44         30.0000000         0
47         23         24         8.0000000        0
48         23         25         2.00000022       0
49         23         49         19.00000034        0
50         23         50         36.00000012        0
51         24         25         1.00000018       0
52         27         33         112.00000074         0
53         27         47         10.00000067        0
54         27         51         14.0000009         0
55         28         30         43.0000002         0
56         28         31         76.00000094        0
57         29         36         80.00000029        0
58         10         29         89.0000008         0
59         29         46         348.00000026         0
60         22         31         513.00000073         0
61         30         31         76.00000015        0
62         30         49         88.0000009         0
63         31         47         71.00000085        0
64         33         34         12.0000009         0
65         33         35         207.00000068         0
66         37         45         139.0000009          0
67         22         37         740.00000019         0
68         39         46         79.00000015        0
69         40         43         98.0000001         0
70         9          41         173.00000091         0
71         10         42         377.00000022         0
72         45         50         166.00000029         0
73         26         42         49.0000003         0
74         34         47         146.0000000          0
75         47         51         14.00000018        0
76         22         48         653.00000050         0
77         45         52         7.0000000        0
78         21         48         309.00000045         0
79         32         39         75.00000028        0
```

- **Net4 Demand File**

```
347
0          1          11
```

0	2	2
0	13	7
0	16	1
0	33	1
0	48	2
0	47	3
1	2	9
1	3	3
1	5	1
1	48	6
1	6	1
1	7	1
1	38	5
1	9	2
1	10	1
1	21	3
1	13	5
1	12	1
1	14	1
1	15	1
1	16	5
1	17	1
1	19	4
1	20	1
1	22	2
1	39	2
1	28	1
1	30	1
1	31	2
1	33	1
1	36	1
1	29	1
1	47	1
1	49	2
1	50	1
1	41	1
1	51	2
2	3	1
2	38	1
2	9	1
2	13	1
2	17	1
2	19	1
2	21	1
2	16	1
2	22	1
2	31	1
2	33	1
2	47	1
2	49	1
20	38	21
16	20	8
18	20	2
0	17	1
3	4	1
3	6	1
3	7	1
3	38	3
3	9	1
3	10	1
3	13	1
3	14	1
3	16	2
3	19	7
3	20	1
3	22	1
3	24	2
3	39	1
3	31	2
3	33	1
3	36	1
3	47	1
3	49	2
3	48	1
4	12	1
4	19	1
4	16	7
5	9	3
5	10	1
5	11	5

5	14	1
5	16	1
5	20	1
5	22	3
5	48	2
5	31	1
5	49	1
5	50	1
5	41	1
6	7	1
6	9	1
6	10	1
6	15	1
6	50	5
6	22	2
6	23	2
6	31	1
6	45	3
6	37	1
6	49	3
7	9	1
7	10	1
7	15	1
7	20	1
7	19	1
7	49	6
7	22	1
7	23	2
7	24	1
7	25	1
7	28	2
7	30	1
7	31	1
7	38	1
7	39	1
7	45	1
7	37	1
7	47	1
7	50	1
7	51	1
8	48	2
8	22	3
12	35	7
9	20	3
20	39	5
10	20	2
17	20	3
20	36	3
20	21	7
20	23	1
20	28	1
20	48	1
9	39	45
9	10	2
9	21	10
9	13	1
9	14	2
9	15	1
9	16	3
9	17	2
9	18	1
9	38	7
9	36	5
9	49	4
9	22	7
9	23	1
9	24	1
9	28	1
9	48	5
9	31	5
9	30	1
9	33	2
9	29	1
9	42	1
9	37	1
9	47	1
9	50	1
9	41	21
9	51	2
10	14	5

10	16	2
10	17	1
10	38	4
10	36	11
10	39	7
10	19	1
10	22	1
10	31	2
10	40	4
10	29	7
10	42	9
10	43	4
10	49	2
10	50	1
10	41	1
10	51	1
36	38	10
0	34	2
11	22	9
11	41	3
11	48	1
12	33	1
13	19	1
13	20	1
13	31	2
13	33	2
13	35	1
13	47	1
13	49	1
13	51	1
13	48	1
14	16	1
14	38	2
14	36	3
14	20	1
14	22	1
14	26	1
14	39	7
14	31	1
14	29	1
14	42	8
14	46	2
14	49	1
14	41	1
15	22	1
15	23	1
15	24	2
15	50	2
15	37	2
15	49	3
21	48	5
22	48	36
16	17	3
16	18	2
16	38	5
16	19	2
16	36	3
16	21	1
16	22	1
16	39	2
16	31	1
16	49	1
16	41	1
17	18	1
17	38	3
17	21	1
22	45	2
24	45	1
31	45	1
45	50	8
37	45	3
45	49	2
18	38	2
19	20	2
20	22	2
20	31	2
20	33	1
20	49	2
20	41	2
19	31	1

19	38	1
10	46	1
21	22	1
21	31	1
21	36	1
21	38	1
23	28	2
28	49	5
28	31	6
28	50	2
28	51	2
4	21	1
4	31	1
22	24	1
22	39	2
22	28	1
22	31	8
22	33	1
22	36	1
22	37	13
22	38	1
22	49	6
22	50	2
22	41	3
22	51	1
23	24	2
23	25	2
23	30	1
23	31	2
23	37	2
23	39	1
23	50	3
23	42	1
23	47	2
23	49	5
23	41	1
23	51	1
24	28	1
24	31	1
24	49	4
24	50	1
24	51	1
25	49	1
26	42	3
27	33	1
27	47	1
27	51	2
28	30	2
28	33	1
28	38	1
28	39	1
28	47	1
28	48	1
29	36	5
29	46	1
22	47	1
30	31	2
30	33	1
30	36	1
30	38	1
30	47	1
30	49	1
30	50	1
30	51	1
30	48	1
31	33	2
31	36	1
31	39	1
31	42	1
31	37	1
31	47	7
31	49	3
31	50	1
31	41	1
31	51	4
31	48	1
33	34	1
33	47	2
33	49	1
33	51	1

```
34     47     4
36     39     2
36     42     1
36     49     2
37     49     1
37     50     1
38     49     1
29     39     1
39     42     4
39     46     1
38     39     1
39     49     1
40     43     1
26     46     1
42     46     1
47     49     1
47     50     2
47     51     3
49     50     3
41     49     1
49     51     1
48     49     1
50     51     1
34     51     1
34     48     1
48     51     1
```

- **Toronto Snif File**

The following is the SNIF file for toronto metro network used in [6]. To make the result comparable to that in [6], only the first 15 nodes and respective spans are used in the IP formulations.

```
Date:     March    27,      1997
File      Name:    toronto.snif
Network:toronto metro
Program:
#
Node      Xcoord   Ycoord
0         1        1
1         0        0
2         0        80
3         50       80
4         70       80
5         0        100
6         0        20
7         20       50
8         100      0
9         30       80
10        70       100
11        0        60
12        20       25
13        60       60
14        0        40
15        40       60
16        50       0
17        75       25
18        20       0
19        100      60
20        110      60
21        120      60
22        130      60
23        140      60
24        150      60
#
Span      NodeA    NodeB    Dist     Working Spare
1         0        1        5        0       0
2         0        3        3        0       0
3         0        4        4        0       0
4         1        2        4.5      0       0
5         1        4        3.3      0       0
```

```
6       1       5       2.5     0       0
7       2       5       2.3     0       0
8       2       6       2       0       0
9       3       4       3.5     0       0
10      3       9       4.3     0       0
11      4       7       8.6     0       0
12      4       8       4       0       0
13      4       10      5.5     0       0
14      4       11      4       0       0
15      5       6       2       0       0
16      5       8       3.5     0       0
17      6       8       3.5     0       0
18      6       12      5.7     0       0
19      6       16      7       0       0
20      7       9       3       0       0
21      7       13      5       0       0
22      8       12      2       0       0
23      8       16      4.3     0       0
24      9       10      4       0       0
25      9       13      3       0       0
26      10      11      3       0       0
27      10      13      6.7     0       0
28      10      14      4       0       0
29      11      12      3.5     0       0
30      11      14      3.5     0       0
31      12      14      4       0       0
32      12      15      2.5     0       0
33      13      14      9.4     0       0
34      13      17      5.3     0       0
35      13      18      7       0       0
36      14      15      3       0       0
37      14      18      3       0       0
38      14      19      3       0       0
39      15      16      2       0       0
40      15      19      6.2     0       0
41      15      24      2       0       0
42      16      24      2.5     0       0
43      17      18      3       0       0
44      17      20      2.5     0       0
45      18      19      2       0       0
46      18      20      4.1     0       0
47      19      20      3.4     0       0
48      19      22      2       0       0
49      19      23      3.5     0       0
50      19      21      6       0       0
51      20      21      4       0       0
52      21      22      4.7     0       0
53      22      23      3       0       0
54      23      24      2.5     0       0
55      19      24      5.4     0       0
```

- **Toronto Demand File**

The demand matrix has 300 demands. One unit of demand exists between every node pair uniformly.

•• **US Snif File**

The following is the SNIF file for Toronto metro network used in [6]. To make the result comparable to that in [6], only the first 20 nodes and respective spans are used in the IP formulations.

```
Date:   March   27,     1997
File    Name:   us.mif
```

```
Network:us long haul
Program:

Node     Xcoord   Ycoord
0        1        1
1        0        0
2        0        80
3        50       80
4        70       80
5        0        100
6        0        20
7        20       50
8        100      0
9        30       80
10       70       100
11       0        60
12       20       25
13       60       60
14       0        40
15       40       60
16       50       0
17       75       25
18       20       0
19       100      60
20       110      60
21       120      60
22       130      60
23       140      60
24       150      60
25       160      60
26       170      60
27       180      60
#
Span     NodeA    NodeB    Dist     Working  Spare
1        0        1        48       0        0
2        0        3        38       0        0
3        1        2        33       0        0
4        1        4        33       0        0
5        2        4        42       0        0
6        2        7        40       0        0
7        3        5        40       0        0
8        3        6        40       0        0
9        4        6        23       0        0
10       5        8        32       0        0
11       6        7        62       0        0
12       6        10       53       0        0
13       7        12       58       0        0
14       8        9        21       0        0
15       8        10       16       0        0
16       9        13       15       0        0
17       10       11       20       0        0
18       10       13       35       0        0
19       11       12       46       0        0
20       12       15       20       0        0
21       12       17       60       0        0
22       13       14       17       0        0
23       13       16       15       0        0
24       13       22       21       0        0
25       14       16       18       0        0
26       15       18       32       0        0
27       16       17       32       0        0
28       16       19       36       0        0
29       16       22       16       0        0
30       17       18       29       0        0
31       17       19       18       0        0
32       18       20       27       0        0
33       18       21       33       0        0
34       19       20       25       0        0
35       19       24       20       0        0
36       20       21       17       0        0
37       22       23       22       0        0
38       22       25       20       0        0
39       22       27       18       0        0
40       23       24       21       0        0
41       23       26       26       0        0
42       23       27       11       0        0
43       25       26       21       0        0
44       26       27       15       0        0
45       11       14       25       0        0
```

- **US Demand File**

The demand matrix has 378 demands. One unit of demand exists between every node pair uniformly.

Appendix C: KST-Alg, IP Formulation and Bounding Program

- **File structure**

All files related to capacity planning algorithms are grouped in one directory. Under this directory, several modules in their respective sub-directories are used implement the algorithms. They are listed as follows:

1. snif: supporting library. Snif is the network topology description format used throughout this research. This module includes the procedures to read and write snif files.

2. route: supporting library. This module includes the procedures to read and write the VP file, which includes the information of ATM VP working routes, backup routes. And in this module there are procedure to find the k-shortest path according to the criteria of number limit of route, hop limit and distance limit.

3. kway: utility program. This module includes the program to generate, working capacity for each span, the working VP routes and possible backup routes set.

4. vpbk: utility program. This module includes the program to implement KST-Alg and upper bound algorithms.

5. oversubscription: utility program. This module includes the program to calculate the oversubscription factors of a particular capacity planning for a given network.

6. ip: utility program. This module includes the program to generate the IP-1, IP-2, IP-3 and lower bound program. Because some parts of constraints in these IP formulations are very similar, the programs are grouped into one file with macro 'IF' to generate different pars for each formulation.

7. txt2snif: utility program. This module includes the program to read the result from the IP solving program, CPLEX, and write the result to a VP description file.

8. test: This module includes the script to generate the capacity planning for each tested sample network.

Only some important programs which would help the understanding of the algorithms are listed in this appendix, i.e., vpbk, ip and oversubscription modules.

- **KST-Alg and upper bound algorithm implementation**

Makefile: ./vpbk/Makefile

```
NI = ../../include
NS = ../../lib
BIN= ../../bin
CC = gcc
CFLAGS = -I$(NI) -L$(NS) -g -Wall
LINTFLAGS= -I$(NI)

all:    vpbk vpspan

vpspan:         vpbk.o  spancut.o
        $(CC) $(CFLAGS) -o vpspan vpbk.o  spancut.o -lns -lm
        cp vpspan $(BIN)

vpbk:           vpbk.o  singlevp.o
        $(CC) $(CFLAGS) -o vpbk vpbk.o  singlevp.o -lns -lm
        cp vpbk $(BIN)


# delete all executables
clean:
        rm -f *.o core vpbk vpspan *BAK

indent:
        indent -i4 vpbk.c
        indent -i4 spancut.c
        indent -i4 singlevp.c

report:*.c Makefile Readme
        enscript *.c Makefile Readme
        echo > report
```

Main module: ./vpbk/vpbk.c

```
#include <stdio.h>
#include <stdlib.h>
#include <malloc.h>
#include <values.h>
#include <string.h>

#include <snif.h>
#include "route.h"

/* function prototypes */

extern int      method;

extern void     niceme();

extern float
calculate_capacity(VP * netVPs, int numVPs, int *backupVP,
                   float *spareCapacity, ROUTE ** all_routes,
                   SPAN * netSpans, int numSpans);


int
main(int argc, char *argv[])
{

    NODE            *netNodes;/* all the nodes in the network */
    SPAN            *netSpans;/* all the spans in the network */
    ADJLIST         *adj;/* (heads of) adjacency lists */
    VP              *netVPs;/* all VPs in the network */

    int             *tag2node, *node2tag, *tag2span, *span2tag;

    int              numNodes, numSpans, numVPs;
    ROUTE           **all_routes;
    int              *num_route;
    float            *BkSpare, bestCapacity, totalworking;
    int              *BkRoute, updateVP;

    int               i, j;

    char              stamp[1024];

    if (argc != 6) {
        printf("Usage: vpbk sniffFile vpFile routeFile outFile rvp-
File\n"
                 "\tsniffFile\tnetwork file in snif format.\n"
                 "\tvpFile\tworking vp capacity and route.\n"
                 "\trouteFile\tpossible restoration route.\n"
                 "\toutFile\tsnif file filled with spare capacity\n"
                 "\trvpFile\trestoration vp capacity and route\n");
        exit(1);
    }
    StartSNIF(argv[1]);

    StartVP(argv[2]);

    StartROUTE(argv[3]);


    /* search for the best route */
    BkSpare = (float *) malloc(numSpans * sizeof(float));
    BkRoute = (int *) malloc(numVPs * sizeof(int));

    for (i = 0; i < numVPs; i++)
        BkRoute[i] = (num_route[i] == 0) ? -1 : 0;
    bestCapacity = MAXFLOAT;

    for (i = 0, updateVP = -1; i != updateVP; i = (i + 1) % numVPs) {
        int             best = BkRoute[i];
        printf("%s: considering VP %d path %d\n",
                (method == 0 ? "single" : "spancut"),
                netVPs[i].tag, netVPs[i].path);
        for (j = 0; j < num_route[i]; j++) {
            float           temp;
            BkRoute[i] = j;

            niceme();
            temp = calculate_capacity(netVPs, numVPs, BkRoute, BkS-
pare,
                                      all_routes, netSpans, numSpans);
            if (temp < bestCapacity) {
                bestCapacity = temp;
                updateVP = i;
                best = j;
                printf("%s: got best route %d to %f\n",
                    (method == 0 ? "single" : "spancut"), j, bestCa-
pacity);
            }
        }
        BkRoute[i] = best;
    }


   bestCapacity = calculate_capacity(netVPs, numVPs, BkRoute, BkS-
pare,
                                      all_routes, netSpans, numSpans);


   /* record the restoration selection to file for performance
analysis */
   ChoiceWriteFile(argv[3],
                  (method == 0 ? ".single.choice" : ".span-
cut.choice"),
                  netVPs, numVPs, BkRoute);
    /* record the restoration info to file */
    RVPWriteFile(argv[5], netVPs, numVPs,
                all_routes, num_route, BkRoute,
                tag2span, tag2node, span2tag, node2tag);

    for (i = 0, totalworking = 0; i < numSpans; i++) {
       totalworking += netSpans[i].working * netSpans[i].distance;
       netSpans[i].spare = BkSpare[i];
    }

    sprintf(stamp,
           "backup VP using %s\n"
           "#network topology: %s\n"
           "#VP description: %s\n"
           "#backup route: %s\n"
           "#backup VP file: %s\n"
           "#DISTANCE WEIGHTED working = %1.2f, sparing = %1.2f, "
           "redundency =  %1.2f%%\n",
           (method == 0 ? "single cut" : "span cut"),
           argv[1], argv[2], argv[3], argv[5],
           totalworking, bestCapacity, bestCapacity * 100 / total-
 working);

    SNIFProgramStamp(stamp);
    SNIFWriteNetFile(argv[4], netNodes, netSpans, numNodes, num-
 Spans,
                    node2tag, tag2node, span2tag, tag2span);

    printf("%s", stamp);

    /* free up memory */
    free(BkRoute);
    free(BkSpare);

    StopROUTE;

    StopVP;

    StopSNIF;

    return (0);
}
```

KST-Alg capacity calculation module: ./vpbk/singlevp.c

```
#include <stdio.h>
#include <malloc.h>

#include <snif.h>
#include "route.h"

/* only consider each VP cut every time */

/*
 * backupVP is a [0 .. numVPs-1] array to record the backup route
 index of
 * each working VP, so the backupVP[i] is the ith working VP's
 backup route
 * index in the all_route table. so all_route[i][ backupVP[i] ] is
 the actual
 * backup route. ^^^^^^^^^^^^ all backup routes for ith working VP
 *
 */
/* spareCapacity is a [0 .. numSpans-1] array to record the spare
 capacity */

int             method = 0;/* single VP backup */
```

```
float
calculate_capacity(VP * netVPs, int numVPs, int *backupVP,
                   float *spareCapacity, ROUTE ** all_routes,
                   SPAN * netSpans, int numSpans)
{
    int              i, j, span;
    float            temp = 0;


    /* initial span spare capacity  */
    for (i = numSpans - 1; i >= 0; i--)
       spareCapacity[i] = 0;

    /* try all backup VPs */
    for (i = numVPs - 1; i >= 0; i--) {
       /* skip the un-restorable VP */
       if (backupVP[i] == -1)
           continue;
       /* try spans of that backup VP */
       for (j = (all_routes[i][backupVP[i]]).num - 1; j >= 0; j--) {
           span = (all_routes[i][backupVP[i]]).span[j];
           if (spareCapacity[span] < netVPs[i].capacity) {
              spareCapacity[span] = netVPs[i].capacity;
           }
       }
    }

    /* add up total capacity together */
    for (i = numSpans - 1; i >= 0; i--)
       temp += spareCapacity[i] * netSpans[i].distance;
    return (temp);

}
```

Upper bound capacity calculation module: ./vpbk/spancut.c

```
#include <stdio.h>
#include <stdlib.h>
#include <malloc.h>

#include <snif.h>
#include "route.h"

/* consider span cut each time */

/*
 * backupVP is a [0 .. numVPs-1] array to record the backup route
 index of
 * each working VP, so the backupVP[i] is the ith working VP's
 backup route
 * index in the all_route table. so all_route[i][ backupVP[i] ] is
 the actual
 * backup route. ^^^^^^^^^^^^ all backup routes for ith working VP
 */
/* spareCapacity is a [0 .. numSpans-1] array to record the spare
 capacity */

int              method = 1;/* span cut backup */

float
calculate_capacity(VP * netVPs, int numVPs, int *backupVP,
                   float *spareCapacity, ROUTE ** all_routes,
                   SPAN * netSpans, int numSpans)
{
    int              i, j, l, m;
    float            temp = 0;
    float           *tempCapa;
    tempCapa - (float *) malloc(numSpans * sizeof(float));

    for (i - numSpans - 1; i >- 0; i--)
       spareCapacity[i] - 0;

    /* try to cut all spans */
    for (m - numSpans - 1; m >- 0; m--) {
       /* init spare capacity */
       for (i - numSpans - 1; i >- 0; i--)
           tempCapa[i] - 0;

       /* try all backup VPs through this span */
       for (i - 0; i < numVPs; i++) {
           for (l - netVPs[i].num - 1; l >- 0; l--) {
              /*
               * if the original route goes through this span, or
 say,
               * working span is cut
               */
              if (netVPs[i].span[l] -- m) {
                  /* add capacity to all spans of backup VP */
                  for (j - (all_routes[i][backupVP[i]]).num - 1; j
 >- 0; j--) {
                      /* add capacity to all backup route spans */
#ifdef  DEBUG
                      printf("add: cut span %d, cut VP %d, spare
 span %d, capa %f\n",
                             m, i, (all_routes[i][back-
 upVP[i]]).span[j], netVPs[i].capacity);

#endif  DEBUG
                      tempCapa[(all_routes[i][back-
 upVP[i]]).span[j]] +- netVPs[i].capacity;
                  }
                  /* stub release all other segments of the working
 VP */
                  for (j - netVPs[i].num - 1; j >- 0; j--) {
                      int          span - netVPs[i].span[j];
                      if (span != m)
#ifdef  DEBUG
                          printf("sub: cut span %d, cut VP %d, spare
 span %d, capa %f\n",
                                 m, i, span, netVPs[i].capacity);
#endif
                      tempCapa[span] -- netVPs[i].capacity;
                  }
                  break;

              }
           }

       }


       /* compare this span cut with others */
       for (i - numSpans - 1; i >- 0; i--)
           spareCapacity[i] - max(spareCapacity[i], tempCapa[i]);

    }


    /* add up total capacity together */
    for (i - numSpans - 1, temp - 0; i >- 0; i--)
       temp +- spareCapacity[i] ** netSpans[i].distance;
    free(tempCapa);
    return (temp);

}
```

•• **IP-1, IP-2, IP-3 and lower bounding algorithms implementation**

Because these four programs are very similar, they are grouped into one single file. There is a conditional compile for each algorithm. TSPARE, TOVER, TTOTAL and TVCSPLIT are symbals for IP-1, IP-2, IP-3 and lower bound algorithms, respectively.

Makefile: ./ip/Makefile

```
NI = ../../include
NS = ../../lib
BIN= ../../bin
CC = gcc
CFLAGS = -I$(NI) -L$(NS) -g -Wall
LINTFLAGS= -I$(NI)

all:    vpipt vpips vpipv vpipo

vpipv:          vpipv.o
        $(CC) $(CFLAGS) -o vpipv vpipv.o -lns -lm
        cp vpipv $(BIN)

vpips:          vpips.o
        $(CC) $(CFLAGS) -o vpips vpips.o -lns -lm
        cp vpips $(BIN)

vpipo:          vpipo.o
        $(CC) $(CFLAGS) -o vpipo vpipo.o -lns -lm
        cp vpipo $(BIN)

vpipt:          vpipt.o
        $(CC) $(CFLAGS) -o vpipt vpipt.o -lns -lm
        cp vpipt $(BIN)

vpipv.o:      vpip.c
        $(CC) $(CFLAGS) -DTVCSPLIT -c -o vpipv.o vpip.c

vpips.o:      vpip.c
        $(CC) $(CFLAGS) -DTSPARE -c -o vpips.o vpip.c

vpipo.o:      vpip.c
        $(CC) $(CFLAGS) -DTOVER -c -o vpipo.o vpip.c

vpipt.o:      vpip.c
        $(CC) $(CFLAGS) -DTTOTAL -c -o vpipt.o vpip.c


# delete all executables
clean:
        rm -f *.o core vpipv vpips vpipo *BAK

indent:
        indent -i4 vpip.c


report: *.c Makefile Readme
        enscript *.c Makefile Readme
        echo >report
```

main module: ./ip/vpip.c

```
#include <stdio.h>
#include <stdlib.h>
#include <values.h>
#include <malloc.h>
#include <string.h>
```

```
#include <snif.h>
#include <route.h>


/*
 * Four possible executive files
 *
 *
 * #define TOVER get the minimum OVER with given sparing
 *
 * #define TSPARE get the minimum SPARE capacity with an upper limit
 of overload
 *
 * #define TTOTAL get the minimum TOTAL working and sparing capacity
 with an upper limit
 * of overload
 *
 * #define TVCSPLIT get the minimum SPARE capacity with an upper
 limit of
 * overload BUT instead of using a IP program, a LP program is gen-
 erated.
 * (only int d....... varibles are not included)
 *
 * One of these should be defined in the Makefile
 */

/*
 * variable format:
 * w123: working capacity for span # 123
 * s123: sparing capacity for span # 123
 * gv12p3: traffic in working VP for demand #12 routing path 3
 * cv12p3: coefficient factor, 0 for no traffic, 1 for yes
 * fv12p3r456: traffic in VP #12 path 3 at backup route #456
 * dv12p3r456: coefficient factor, 0 for no traffic, 1 for yes
 */



int
main(int argc, char *argv[])
{

    NODE            *netNodes;/* all the nodes in the network */
    SPAN            *netSpans;/* all the spans in the network */
    ADJLIST         *adj;/* (heads of) adjacency lists */
    VP              *netVPs;/* all VPs in the network */

    int             *tag2node, *node2tag, *tag2span, *span2tag;

    int              numNodes, numSpans, numVPs;

    ROUTE          **all_routes;
    int             *num_route;

    FILE            *fp;

    int              i, j, k, l, m;


    float            rightside;
    float            flagrest;

#ifndefTVCSPLIT
    char            *filename;
    char            *suffix = ".ord";
#endif TVCSPLIT

#ifndefTOVER
    float            totalcapa;
    float            overload = 1.0;
```

```
#endif TOVER

#ifdef TTOTAL
    float   *demand;
#endif



#ifdef TOVER

    if (argc != 5) {
       printf("Usage: vpip snifFile vpFile routeFile outipFile\n"
              "\tsnifFile\tnetwork file in snif format.\n"
              "\tvpFile\tworking vp capacity and route.\n"
              "\trouteFile\tpossible restoration route.\n"
              "\toutipFile\trestoration IP formulation file.\n");
       exit(1);
    }
#else  TOVER

    if ((argc != 5) && (argc != 6)) {
       printf("Usage: vpip snifFile vpFile routeFile outipFile
 [overload]\n"
              "\tsnifFile\tnetwork file in snif format.\n"
              "\tvpFile\tworking vp capacity and route.\n"
              "\trouteFile\tpossible restoration route.\n"
              "\toutipFile\trestoration IP formulation file.\n"
              "\toverload\toverload factor (should >= 1.0) default
 1.0\n");
       exit(1);
    }
    if (argc == 6)
       overload = atof(argv[5]);

#endif TOVER

    StartSNIF(argv[1]);

    StartVP(argv[2]);

    StartROUTE(argv[3]);


    if ((fp = fopen(argv[4], "w")) == NULL) {
       fprintf(stderr, "fail to open the IP formulation file:
 %s\n", argv[4]);
       exit(1);
    }

#ifdef TTOTAL
    /* get the capacity for each demand pair */
    demand = malloc(numVPs * sizeof (float));
    for (i = 0; i < numVPs; i++) {
       demand[i] = 0.0;
    }
    for (i = 0; i < numVPs; i++) {
       demand[netVPs[i].tag] += netVPs[i].capacity;
    }
#endif TTOTAL


    /* write header */

    /* write objective */
#ifdef TOVER
    fprintf(fp, "- overload");
#else  TOVER
#ifdef TTOTAL
    for (i = 0; i < numSpans; i++) {
       fprintf(fp, "- %g w%ld ", netSpans[i].distance,
 span2tag[i]);
```

```
    }
#endif TTOTAL
    for (i = 0; i < numSpans; i++) {
        fprintf(fp, "- %g s%ld ", netSpans[i].distance,
 span2tag[i]);
    }
#endif TOVER
    fprintf(fp, ";\n\n");



#ifdef TTOTAL

    /* write the constraint 6, radio button of working VPs for each
 demand */

    printf("writing constraint 6...\n");

    for (i = 0, m = netVPs[0].tag; i < numVPs; i++) {
        if (m != netVPs[i].tag) {
            fprintf(fp, " = 1;\n");
            m = netVPs[i].tag;
        }
        fprintf(fp, "+ cv%ldp%ld ", netVPs[i].tag, netVPs[i].path);
    }
    fprintf(fp, " = 1;\n\n\n");


    /* write the constraint 4, working VP is coeffient (0 or 1) of
 each demand */

#ifdef verbose
    printf("writing constraint 4...\n");

    for (i = 0; i < numVPs; i++) {
        fprintf(fp, "%g cv%ldp%ld - gv%ldp%ld = 0;\n",
                demand[netVPs[i].tag], netVPs[i].tag, netVPs[i].path,
                        netVPs[i].tag, netVPs[i].path);
    }

#endif verbose

    /* write constraint 5 in IP-3, for working capacity > demand */

    printf("writing constraint 5...\n");

    fprintf(fp, "\n\n\n");
    for (i = 0; i < numSpans; i++) {
        for (k = 0; k < numVPs; k++) {
            for (l = 0; l < netVPs[k].num; l++) {
                if (i == netVPs[k].span[l]) {
#ifdef verbose
                    fprintf(fp, "+ gv%ldp%ld ", netVPs[k].tag,
 netVPs[k].path);
#else  verbose
                    fprintf(fp, "+ %g cv%ldp%ld ",
                        demand[netVPs[k].tag], netVPs[k].tag,
 netVPs[k].path);
#endif verbose
                    break;
                }
            }
        }
        fprintf(fp, "- w%ld = 0.0;\n\n", span2tag[i]);
    }

#endif TTOTAL

    /* write the constraint 3, radio button for all possible backup
 route */
```

```
    printf("writing constraint 3...\n");

    for (i = 0; i < numVPs; i++) {
       for (j = 0; j < num_route[i]; j++) {
           fprintf(fp, "+ dv%ldp%ldr%03d ", netVPs[i].tag,
 netVPs[i].path, j);
       }
#ifdef TTOTAL
       fprintf(fp, " - cv%ldp%ld = 0;\n", netVPs[i].tag,
 netVPs[i].path);
#else  TTOTAL
       fprintf(fp, " = 1;\n");
#endif
    }
    fprintf(fp, "\n\n");


    /* write constraint 1, actual overload < design peak */

    printf("writing constraint 1...\n");
    fflush(stdout);

    /* try to cut span i */
    for (i = 0; i < numSpans; i++) {
       printf("formulating overload in case of failure span %d\n",
 span2tag[i]);
       /* try to formulate overload of span j */
       for (j = 0; j < numSpans; j++) {
           if (i == j) {
              /* don't care about span j overload in case of span i
 failure */
              continue;
           }
           /* overload of span i in case of failure span i */
           rightside = 0.0;
           flagrest = FALSE;
           for (k = 0; k < numVPs; k++) {
              /* check if VP k is cut in case of span i failure */
              for (l = 0; l < netVPs[k].num; l++) {
                  if (i == netVPs[k].span[l])
                     break;
              }
              if (l == netVPs[k].num)
                  continue;
              /*
               * check if the VP k went through span j, if yes, it
 is a
               * stub traffic and re-routed elsewhere
               */
              for (l = 0; l < netVPs[k].num; l++) {
                  if (j == netVPs[k].span[l])
                     break;
              }
              if (l != netVPs[k].num) {
#ifdef TTOTAL
#ifdef verbose
                  fprintf(fp, "- gv%ldp%ld ", netVPs[k].tag,
 netVPs[k].path);
#else  verbose
                  fprintf(fp, "- %g cv%ldp%ld ",
                            demand[netVPs[k].tag], netVPs[k].tag,
 netVPs[k].path);
#endif verbose
#else  TTOTAL
                  rightside += netVPs[k].capacity;
#endif TTOTAL
                  continue;
              }
              /* check if the backup route go through span j */
              for (l = 0; l < num_route[k]; l++) {
                  for (m = 0; m < all_routes[k][l].num; m++) {
                    if (j == all_routes[k][l].span[m]) {
                        flagrest = TRUE;
#ifdef  verbose
                        fprintf(fp, "+ fv%ldp%ldr%03d ",
                                netVPs[k].tag, netVPs[k].path, l);
#else   verbose
                        fprintf(fp, "+ %g dv%ldp%ldr%03d ",
#ifdef TTOTAL
                                    demand[netVPs[k].tag],
#else   TTOTAL
                                    netVPs[k].capacity,
#endif TTOTAL
                                netVPs[k].tag, netVPs[k].path, l);
#endif verbose
                        break;
                    }
                }
            }               /* try next possible route */

        }                   /* try next VP */

        if (flagrest == TRUE) {
            /*
             ** finish all backup route, now write working and
 stub
             * traffic
             */

            /* write overload factor */
#ifdef  TOVER
            rightside -= netSpans[j].working;
            fprintf(fp, "- %g overload ",
                    netSpans[j].working + netSpans[j].spare);
#else   TOVER
#ifdef  TTOTAL
            if (overload != 1.0) {
                fprintf(fp, "- %g w%ld ",
                    overload - 1.0, span2tag[j]);
            }
            fprintf(fp, "- %g s%ld ",
                    overload, span2tag[j]);
#else   TTOTAL
            fprintf(fp, "- %g s%ld",
                    overload, span2tag[j]);
            rightside += (overload - 1.0) ** netSpans[j].working;
#endif  TTOTAL
#endif  TOVER

            fprintf(fp, " <= %g;\n\n", rightside);
        }
    }                       /* try next overload span */
  }                         /* try next cut span */


   /* write the constraint 2, backup route is sufficient to support
 working VP */
#ifdef verbose
    printf("writing constraint 2...\n");
    fflush(stdout);

    for (i = 0; i < numVPs; i++) {
        for (j = 0; j < num_route[i]; j++) {
            fprintf(fp, "fv%ldp%ldr%03d ", netVPs[i].tag,
 netVPs[i].path, j);
            fprintf(fp, " - %g dv%ldp%ldr%03d = 0;\n",
#ifdef  TTOTAL
                    demand[netVPs[i].tag],
#else   TTOTAL
                    netVPs[i].capacity,
#endif  TTOTAL
                    netVPs[i].tag, netVPs[i].path, j);
        }
    }
    fprintf(fp, "\n\n");
#endif  verbose

    /* write bounds for link capacity, just for quick solving */
#ifdef TOVER
    fprintf(fp, "overload <= 100.0;\n");
#else  TOVER
    for (i = 0, totalcapa = 0.1; i < numVPs; i++) {
        totalcapa += netVPs[i].capacity;
    }
    for (i = 0; i < numSpans; i++) {
#ifdef TTOTAL
        fprintf(fp, " w%ld <= %g;\n", span2tag[i], totalcapa);
#endif TTOTAL
        fprintf(fp, " s%ld <= %g;\n", span2tag[i], totalcapa);
    }
#endif TOVER

    /* write constraint for exclusive coefficient <= 1 */
#if     0
#ifndef TVCSPLIT
    fprintf(fp, "\n\n");
    for (i = 0; i < numVPs; i++) {
#ifdef TTOTAL
        fprintf(fp, "cv%ldp%ld <= 1;\n", netVPs[i].tag,
 netVPs[i].path);
#endif TTOTAL
        for (j = 0; j < num_route[i]; j++) {
            fprintf(fp, "dv%ldp%ldr%03d <= 1;\n",
                    netVPs[i].tag, netVPs[i].path, j);
        }
    }
    fprintf(fp, "\n\n");
#endif TVCSPLIT
#endif 0

    /* write integers for exclusive coefficient */
#ifndef TVCSPLIT
    fprintf(fp, "\n\nint ");
    for (i = 0; i < numVPs; i++) {
#ifdef TTOTAL
        fprintf(fp, "cv%ldp%ld ", netVPs[i].tag, netVPs[i].path);
#endif TTOTAL
        for (j = 0; j < num_route[i]; j++) {
            fprintf(fp, "dv%ldp%ldr%03d ",
                    netVPs[i].tag, netVPs[i].path, j);
        }
    }
    fprintf(fp, ";\n\n");
#endif TVCSPLIT


    /* write end */
    fclose(fp);

#ifndef TVCSPLIT
    /* write order file for all binary variables */

    /* record the restoration selection to file for performance
 analysis */
    i = strlen(argv[4]) + strlen(suffix) + 1;
    if ((filename = (char *) malloc(i * sizeof(char))) == NULL) {
        fprintf(stderr, "fail to allocate memory for choice file
 name\n");
        exit(1);
    }
    strcpy(filename, argv[4]);
    strcat(filename, suffix);

    if ((fp = fopen(filename, "w")) == NULL) {
        fprintf(stderr, "fail to open the order file: %s\n", file-
name);
        exit(1);
    }
    free(filename);

    fprintf(fp, "NAME\n");

    for (i = 0; i < numVPs; i++) {
#ifdef TTOTAL
        fprintf(fp, " UP cv%ldp%ld                              ",
                netVPs[i].tag, netVPs[i].path);
        fprintf(fp, "%d\n",
                (int) (demand[netVPs[i].tag] + 1));
        for (j = 0; j < num_route[i]; j++) {
            fprintf(fp, " UP dv%ldp%ldr%03d                         ",
                    netVPs[i].tag, netVPs[i].path, j);
            fprintf(fp, "%d\n",
                    (int) (netVPs[i].capacity / 10 + 1));
        }
#else   TTOTAL
        for (j = 0; j < num_route[i]; j++) {
            fprintf(fp, " UP dv%ldp%ldr%03d                         ",
                    netVPs[i].tag, netVPs[i].path, j);
            fprintf(fp, "%d\n",
                    (int) (netVPs[i].capacity / 10 + 1));
        }
#endif TTOTAL
    }

    fprintf(fp, "ENDATA\n");

    fclose(fp);

#endif TVCSPLIT

    /* free up memory */

    StopROUTE;

    StopVP;

    StopSNIF;

    return (0);

}
```

- **Oversubscription calculation module**

  Makefile: ./oversubscription/Makefile

```
NI = ../../include
NS = ../../lib
BIN= ../../bin
CC = gcc
CFLAGS = -I$(NI) -L$(NS) -g -Wall

all:    overload

overload:       overload.o
        $(CC) $(CFLAGS) -o overload overload.o -L$(NS) -lns -lm
        cp overload $(BIN)

# delete all executables
```

```
clean:
        rm -f *.o core overload *BAK

indent:
        indent -i4 overload.c

report: *.c Makefile Readme
        enscript *.c Makefile Readme
        echo > report
```

Main module: ./oversubscription/overload.c

```
/* oversubscription calculation utility program, the result of this
 file can be read by gnuplot */

#include <stdio.h>
#include <assert.h>
#include <stdlib.h>
#include <math.h>
#include <malloc.h>
#include <alloca.h>
#include <values.h>

#include <snif.h>
#include <route.h>

#define MAXOVER10
#defineFACTOR 100

#define max(a, b)       ((a>b)? (a) : (b))
#define min(a, b)       ((a<b)? (a) : (b))

int
main(int argc, char *argv[])
{

    NODE            *netNodes;/* all the nodes in the network */
    SPAN            *netSpans;/* all the spans in the network */
    ADJLIST         *adj;/* (heads of) adjacency lists */
    VP              *netVPs;/* all VPs in the network */

    int             *tag2node, *node2tag, *tag2span, *span2tag;

    int              numNodes, numSpans, numVPs;

    int              i, j, l, m, totalover = 0;

    float           *temp, overaver = 0.0;
    float            overmin = MAXFLOAT, overmax = -MAXFLOAT;

    int             *hist, ind, total, numhist;

    FILE            *fp;

    if (argc != 5) {
        printf("Usage: overload sniffFile vpFile rawoutFile pdfout-
File\n"
               "\tsniffFile\tsnif file describing network struc-
ture.\n"
               "\tvpFile\tworking and backup vp capacity and
route.\n"
               "\trawoutFile\tfile containing raw overload fac-
tors.\n"
               "\tpdfoutFile\tfile containing overload PDF and
CDF.\n");
        exit(1);
    }
    StartSNIF(argv[1]);

    StartVP(argv[2]);
```

```
#ifdef DEBUG
    for (i = 0; i < numVPs; i++)
        printf("%d %d %d %d %1.2f\n", netVPs[i].tag,
 netVPs[i].source,
                netVPs[i].target, netVPs[i].path, netVPs[i].capac-
 ity);
#endif DEBUG


    /* record the restoration info to file */
    if ((fp = fopen(argv[3], "w")) == NULL) {
        fprintf(stderr, "fail to open overload raw data file %s.\n",
 argv[3]);
        exit(1);
    }
    /* write header of overload factors file */
    fprintf(fp, "#Title:\toverload factors\n");
    fprintf(fp, "#Snif:\t%s\n", argv[1]);
    fprintf(fp, "#VP:\t%s\n", argv[2]);
    fprintf(fp, "#Span #:\t%d\n", numSpans);
    fprintf(fp, "#Comment:\trow - cut span, column - overload
 span\n\n");
#if     0
    for (i = 0; i < numSpans; i++) {
        fprintf(fp, "\t%d", span2tag[i]);
    }
    fprintf(fp, "\n");
#endif

    temp = (float *) malloc(numSpans * sizeof(float));

    hist = (int *) malloc((MAXOVER * FACTOR + 1) * sizeof(int));
    for (i = 0; i <= MAXOVER * FACTOR; i++)
        hist[i] = 0;

    /* try to cut every span */
    for (m = 0; m < numSpans; m++) {
#ifdef DEBUG
        printf("Now considering span cut %d\n", span2tag[m]);
#endif DEBUG

        /* init spare capacity */
        for (i = numSpans - 1; i >= 0; i--)
            temp[i] = 0;

        /* try all backup VPs through this span */
        for (i = 0; i < numVPs; i++) {
            for (l = netVPs[i].wnum - 1; l >= 0; l--) {
                /*
                 * if the original route goes through this span, or
 say,
                 * working span is cut
                 */
                if (netVPs[i].wspan[l] == m) {
#ifdef DEBUG
                    printf("VP %d path %d found tranversing span
 %d\n",
                            netVPs[i].tag, netVPs[i].path,
 span2tag[m]);
#endif DEBUG
                    /* add capacity to all spans of backup VP */
                    for (j = netVPs[i].num - 1; j >= 0; j--) {
                        int              span = netVPs[i].span[j];
                        /* add capacity to all backup route spans */
#ifdef DEBUG
                        printf("add: cut span %d, cut VP %d, spare
 span %d, capa %f\n",
                                m, i, span2tag[span], netVPs[i].capac-
 ity);
#endif DEBUG
                        temp[span] += netVPs[i].capacity;
                    }
                    /* stub release all other segments of the working
 VP */
                    for (j = netVPs[i].wnum - 1; j >= 0; j--) {
                        int             span = netVPs[i].wspan[j];
                        if (span != m) {
#ifdef DEBUG
                            printf("sub: cut span %d, cut VP %d, spare
 span %d, capa %f\n",
                                    m, i, span2tag[span],
 netVPs[i].capacity);
#endif
                            temp[span] -= netVPs[i].capacity;
                        }
                    }
                    break;
                }
            }
        }

        for (i = 0; i < numSpans; i++) {
            if (i == m) {
                continue;
            }
            temp[i] = max(temp[i], 0.0);
            temp[i] = (netSpans[i].working + netSpans[i].spare) ==
 0.0 ?
                1.0 : ((temp[i] + netSpans[i].working) /
                        (netSpans[i].working + netSpans[i].spare));
            overmin = min(overmin, temp[i]);
            overmax = max(overmax, temp[i]);
            if (temp[i] > 1.0) {
                totalover++;
                overaver += temp[i];
            }
            fprintf(fp, "%f\t%f\n", (span2tag[m] + 1.0 * i / num-
 Spans), temp[i]);

            ind = min((int) (0.99999+temp[i] * FACTOR), MAXOVER *
 FACTOR);
            hist[ind] += 1;

        }
    }

    fprintf(fp, "#\n#\n");
    fprintf(fp, "#max:\t%f\n", overmax);
    fprintf(fp, "#num of overload:\t%d\n", totalover);
   fprintf(fp, "#average:\t%f\n", (totalover == 0) ? 0.0 : overaver
 / totalover);

    if (overmax > MAXOVER)
        fprintf(stderr,
                "Err: MAXOVER 10 is less than actual maximum over-
 load.\n");
    numhist = (int) (overmax +.99999) * FACTOR;

    for (i = 0, total = 0; i <= numhist; i++) {
        total += hist[i];
        if (total >= (.9 * numSpans * (numSpans - 1)) )  {
            fprintf(fp, "#90%% actual overload is: %1.2f\n",
                        (float) i / FACTOR);
            break;
        }
    }
    fprintf(fp, "#\n#\n");

    fclose(fp);
    /* writing PDF file */
    if ((fp = fopen(argv[4], "w")) == NULL) {
        fprintf(stderr, "fail to open the overload pdf file: %s\n",
argv[4]);
        exit(1);
    }

    for (i = 0, total = 0; i <= numhist; i++) {
        fprintf(fp, "%1.2f\t", (float) i / FACTOR);
        fprintf(fp, "%1.3f\t", (float) hist[i] / numSpans / (num-
Spans - 1));
        total += hist[i];
        fprintf(fp, "%1.3f\n", (float) total / numSpans / (numSpans
- 1));
    }
    fprintf(fp, "\n");

    fclose(fp);

    free(temp);
    free(hist);

    /* free up memory */
    StopVP;

    StopSNIF;

    return (0);
}
```

- **Sample network *TINY* and the resualtant IP-1, IP-2, IP-3**

tiny.snif:

```
Date:   February 2, 1995

File Name:  TINY

Network:   Very tiny sample network

Program: Test Network Working=509 , Spare=532

Node    Xcoord          Ycoord
1       47.00000077.000000
2       34.00000083.000000
3       22.00000076.000000
4       20.00000050.000000
5       43.00000054.000000
6       85.00000074.000000

Span    NodeA   NodeB   DistanceWorkingSpare
1       1       2       20.00   74      53
2       1       3       30.00   71      74
3       1       4       70.00   71      68
4       1       5       35.00   53      71
5       1       6       50.00   55      48
6       2       3       20.00   53      74
7       3       5       40.00   16      18
8       4       5       35.00   68      71
9       5       6       55.00   48      55
```

tiny.dmd: demand matrix

```
8
1 4 2.23
1 5 3.55
2 4 6.98
2 5 3.41
2 6 7.92
```

```
3 4 1.37
3 6 3.44
4 6 5.01
```

IP-1:

```
- 20 s1 - 30 s2 - 70 s3 - 35 s4 - 50 s5 - 20 s6 - 40 s7 - 35 s8 - 55
 s9 ;

+ dv0p0r000  = 1;
+ dv0p1r000 + dv0p1r001 + dv0p1r002 + dv0p1r003  = 1;
+ dv1p0r000 + dv1p0r001 + dv1p0r002 + dv1p0r003  = 1;
+ dv2p0r000 + dv2p0r001  = 1;
+ dv2p1r000 + dv2p1r001 + dv2p1r002  = 1;
+ dv3p0r000 + dv3p0r001 + dv3p0r002  = 1;
+ dv4p0r000 + dv4p0r001 + dv4p0r002  = 1;
+ dv5p0r000 + dv5p0r001  = 1;
+ dv6p0r000 + dv6p0r001 + dv6p0r002  = 1;
+ dv7p0r000  = 1;


+ 3.49 dv2p0r000 + 3.49 dv2p1r001 + 3.49 dv2p1r002 + 3.41 dv3p0r001
 + 3.41 dv3p0r002 + 7.92 dv4p0r001 + 7.92 dv4p0r002 - 1 s2 <= 0;

+ 3.49 dv2p0r000 + 3.49 dv2p0r001 + 3.41 dv3p0r002 + 7.92 dv4p0r002
 - 1 s3 <= 3.49;

+ 3.49 dv2p1r001 + 7.92 dv4p0r001 - 1 s4 <= 6.9;

+ 3.49 dv2p0r001 + 3.49 dv2p1r002 + 3.41 dv3p0r001 - 1 s5 <= 7.92;

+ 3.49 dv2p0r000 + 3.49 dv2p0r001 + 3.49 dv2p1r000 + 3.49 dv2p1r001
 + 3.49 dv2p1r002 + 3.41 dv3p0r000 + 3.41 dv3p0r001 + 3.41 dv3p0r002
 + 7.92 dv4p0r000 + 7.92 dv4p0r001 + 7.92 dv4p0r002 - 1 s6 <= 0;

+ 3.49 dv2p0r001 + 3.49 dv2p1r000 + 3.41 dv3p0r000 + 7.92 dv4p0r000
 - 1 s7 <= 0;

+ 3.49 dv2p1r000 + 3.49 dv2p1r001 + 3.49 dv2p1r002 + 3.41 dv3p0r002
 + 7.92 dv4p0r002 - 1 s8 <= 3.49;

+ 3.49 dv2p0r001 + 3.49 dv2p1r002 + 3.41 dv3p0r001 + 7.92 dv4p0r000
 + 7.92 dv4p0r001 + 7.92 dv4p0r002 - 1 s9 <= 0;

+ 3.44 dv6p0r001 + 3.44 dv6p0r002 - 1 s1 <= 0;

+ 3.44 dv6p0r002 - 1 s3 <= 0;

+ 3.44 dv6p0r001 - 1 s4 <= 0;

+ 3.44 dv6p0r001 + 3.44 dv6p0r002 - 1 s6 <= 0;

+ 3.44 dv6p0r000 - 1 s7 <= 0;

+ 3.44 dv6p0r002 - 1 s8 <= 0;

+ 3.44 dv6p0r000 + 3.44 dv6p0r001 + 3.44 dv6p0r002 - 1 s9 <= 0;

+ 1.115 dv0p1r002 - 1 s1 <= 3.49;

+ 1.115 dv0p1r001 + 3.49 dv2p1r001 + 3.49 dv2p1r002 - 1 s2 <= 0;

+ 1.115 dv0p1r000 + 3.49 dv2p1r001 - 1 s4 <= 0;

+ 1.115 dv0p1r003 + 3.49 dv2p1r002 - 1 s5 <= 0;

+ 1.115 dv0p1r002 + 3.49 dv2p1r000 + 3.49 dv2p1r001 + 3.49 dv2p1r002
 - 1 s6 <= 0;

+ 1.115 dv0p1r001 + 1.115 dv0p1r002 + 3.49 dv2p1r000 - 1 s7 <= 0;
+ 1.115 dv0p1r000 + 1.115 dv0p1r001 + 1.115 dv0p1r002 + 1.115
  dv0p1r003 + 3.49 dv2p1r000 + 3.49 dv2p1r001 + 3.49 dv2p1r002 - 1 s8
  <= 0;

+ 1.115 dv0p1r003 + 3.49 dv2p1r002 - 1 s9 <= 0;

+ 3.55 dv1p0r001 - 1 s1 <= 6.9;

+ 3.55 dv1p0r000 + 3.49 dv2p0r000 + 3.41 dv3p0r001 + 3.41 dv3p0r002
  - 1 s2 <= 0;

+ 1.115 dv0p0r000 + 3.55 dv1p0r003 + 3.49 dv2p0r000 + 3.49 dv2p0r001
  + 3.41 dv3p0r002 - 1 s3 <= 0;

+ 3.55 dv1p0r002 + 3.49 dv2p0r001 + 3.41 dv3p0r001 - 1 s5 <= 0;

+ 3.55 dv1p0r001 + 3.49 dv2p0r000 + 3.49 dv2p0r001 + 3.41 dv3p0r000
  + 3.41 dv3p0r001 + 3.41 dv3p0r002 - 1 s6 <= 0;

+ 3.55 dv1p0r000 + 3.55 dv1p0r001 + 3.49 dv2p0r001 + 3.41 dv3p0r000
  - 1 s7 <= 0;

+ 3.55 dv1p0r003 + 3.41 dv3p0r002 - 1 s8 <= 4.605;

+ 3.55 dv1p0r002 + 3.49 dv2p0r001 + 3.41 dv3p0r001 - 1 s9 <= 0;

+ 3.44 dv6p0r001 + 3.44 dv6p0r002 - 1 s1 <= 7.92;

+ 7.92 dv4p0r001 + 7.92 dv4p0r002 - 1 s2 <= 3.44;

+ 7.92 dv4p0r002 + 3.44 dv6p0r002 - 1 s3 <= 0;

+ 7.92 dv4p0r001 + 3.44 dv6p0r001 - 1 s4 <= 0;

+ 7.92 dv4p0r000 + 7.92 dv4p0r001 + 7.92 dv4p0r002 + 3.44 dv6p0r001
  + 3.44 dv6p0r002 - 1 s6 <= 0;

+ 7.92 dv4p0r000 + 3.44 dv6p0r000 - 1 s7 <= 0;

+ 7.92 dv4p0r002 + 3.44 dv6p0r002 - 1 s8 <= 0;

+ 7.92 dv4p0r000 + 7.92 dv4p0r001 + 7.92 dv4p0r002 + 3.44 dv6p0r000
  + 3.44 dv6p0r001 + 3.44 dv6p0r002 - 1 s9 <= 0;

+ 1.37 dv5p0r001 - 1 s1 <= 0;

+ 1.37 dv5p0r000 - 1 s2 <= 0;

+ 1.37 dv5p0r000 + 1.37 dv5p0r001 - 1 s3 <= 0;

+ 1.37 dv5p0r001 - 1 s6 <= 0;

+ 1.37 dv5p0r001 - 1 s1 <= 3.49;

+ 3.49 dv2p0r000 + 1.37 dv5p0r000 - 1 s2 <= 0;

+ 1.115 dv0p0r000 + 3.49 dv2p0r000 + 3.49 dv2p0r001 + 1.37 dv5p0r000
  + 1.37 dv5p0r001 + 5.01 dv7p0r000 - 1 s3 <= 0;

+ 3.49 dv2p0r001 + 5.01 dv7p0r000 - 1 s5 <= 0;

+ 3.49 dv2p0r000 + 3.49 dv2p0r001 + 1.37 dv5p0r001 - 1 s6 <= 0;

+ 3.49 dv2p0r001 - 1 s7 <= 1.37;

+ 3.49 dv2p0r001 - 1 s9 <= 5.01;

+ 5.01 dv7p0r000 - 1 s3 <= 0;

+ 5.01 dv7p0r000 - 1 s5 <= 0;
```

```
   s1 <- 34.01;
   s2 <- 34.01;
   s3 <- 34.01;
   s4 <- 34.01;
   s5 <- 34.01;
   s6 <- 34.01;
   s7 <- 34.01;
   s8 <- 34.01;
   s9 <- 34.01;


  int dv0p0r000 dv0p1r000 dv0p1r001 dv0p1r002 dv0p1r003 dv1p0r000
   dv1p0r001 dv1p0r002 dv1p0r003 dv2p0r000 dv2p0r001 dv2p1r000
   dv2p1r001 dv2p1r002 dv3p0r000 dv3p0r001 dv3p0r002 dv4p0r000
   dv4p0r001 dv4p0r002 dv5p0r000 dv5p0r001 dv6p0r000 dv6p0r001
   dv6p0r002 dv7p0r000 ;
```

IP-2:

```
  - overload;

  + dv0p0r000  = 1;
  + dv0p1r000 + dv0p1r001 + dv0p1r002 + dv0p1r003  = 1;
  + dv1p0r000 + dv1p0r001 + dv1p0r002 + dv1p0r003  = 1;
  + dv2p0r000 + dv2p0r001  = 1;
  + dv2p1r000 + dv2p1r001 + dv2p1r002  = 1;
  + dv3p0r000 + dv3p0r001 + dv3p0r002  = 1;
  + dv4p0r000 + dv4p0r001 + dv4p0r002  = 1;
  + dv5p0r000 + dv5p0r001  = 1;
  + dv6p0r000 + dv6p0r001 + dv6p0r002  = 1;
  + dv7p0r000  = 1;


  + 3.49 dv2p0r000 + 3.49 dv2p1r001 + 3.49 dv2p1r002 + 3.41 dv3p0r001
   + 3.41 dv3p0r002 + 7.92 dv4p0r001 + 7.92 dv4p0r002 - 4.81 overload
   <- -3.44;

  + 3.49 dv2p0r000 + 3.49 dv2p0r001 + 3.41 dv3p0r002 + 7.92 dv4p0r002
   - 9.615 overload  <- -1.115;

  + 3.49 dv2p1r001 + 7.92 dv4p0r001 - 11.565 overload  <- -4.665;

  + 3.49 dv2p0r001 + 3.49 dv2p1r002 + 3.41 dv3p0r001 - 16.37 overload
   <- -3.44;

  + 3.49 dv2p0r000 + 3.49 dv2p0r001 + 3.49 dv2p1r000 + 3.49 dv2p1r001
   + 3.49 dv2p1r002 + 3.41 dv3p0r000 + 3.41 dv3p0r001 + 3.41 dv3p0r002
   + 7.92 dv4p0r000 + 7.92 dv4p0r001 + 7.92 dv4p0r002 - 7.92 overload
   <- 0;

  + 3.49 dv2p0r001 + 3.49 dv2p1r000 + 3.41 dv3p0r000 + 7.92 dv4p0r000
   - 9.29 overload  <- -1.37;

  + 3.49 dv2p1r000 + 3.49 dv2p1r001 + 3.49 dv2p1r002 + 3.41 dv3p0r002
   + 7.92 dv4p0r002 - 14.475 overload  <- -7.495;

  + 3.49 dv2p0r001 + 3.49 dv2p1r002 + 3.41 dv3p0r001 + 7.92 dv4p0r000
   + 7.92 dv4p0r001 + 7.92 dv4p0r002 - 12.93 overload  <- -5.01;

  + 3.44 dv6p0r001 + 3.44 dv6p0r002 - 18.31 overload  <- -18.31;

  + 3.44 dv6p0r002 - 9.615 overload  <- -4.605;

  + 3.44 dv6p0r001 - 11.565 overload  <- -11.565;

  + 3.44 dv6p0r001 + 3.44 dv6p0r002 - 7.92 overload  <- 0;

  + 3.44 dv6p0r000 - 9.29 overload  <- -1.37;
```

```
+ 3.44 dv6p0r002 - 14.475 overload  <- -10.985;

+ 3.44 dv6p0r000 + 3.44 dv6p0r001 + 3.44 dv6p0r002 - 12.93 overload
 <- -5.01;

+ 1.115 dv0p1r002 - 18.31 overload  <- -14.82;

+ 1.115 dv0p1r001 + 3.49 dv2p1r001 + 3.49 dv2p1r002 - 4.81 overload
 <- -3.44;

+ 1.115 dv0p1r000 + 3.49 dv2p1r001 - 11.565 overload  <- -11.565;

+ 1.115 dv0p1r003 + 3.49 dv2p1r002 - 16.37 overload  <- -11.36;

+ 1.115 dv0p1r002 + 3.49 dv2p1r000 + 3.49 dv2p1r001 + 3.49 dv2p1r002
 - 7.92 overload  <- 0;

+ 1.115 dv0p1r001 + 1.115 dv0p1r002 + 3.49 dv2p1r000 - 9.29 overload
 <- -1.37;

+ 1.115 dv0p1r000 + 1.115 dv0p1r001 + 1.115 dv0p1r002 + 1.115
 dv0p1r003 + 3.49 dv2p1r000 + 3.49 dv2p1r001 + 3.49 dv2p1r002 -
 14.475 overload  <- -10.985;

+ 1.115 dv0p1r003 + 3.49 dv2p1r002 - 12.93 overload  <- -5.01;

+ 3.55 dv1p0r001 - 18.31 overload  <- -11.41;

+ 3.55 dv1p0r000 + 3.49 dv2p0r000 + 3.41 dv3p0r001 + 3.41 dv3p0r002
 - 4.81 overload  <- -3.44;

+ 1.115 dv0p0r000 + 3.55 dv1p0r003 + 3.49 dv2p0r000 + 3.49 dv2p0r001
 + 3.41 dv3p0r002 - 9.615 overload  <- -4.605;

+ 3.55 dv1p0r002 + 3.49 dv2p0r001 + 3.41 dv3p0r001 - 16.37 overload
 <- -11.36;

+ 3.55 dv1p0r001 + 3.49 dv2p0r000 + 3.49 dv2p0r001 + 3.41 dv3p0r000
 + 3.41 dv3p0r001 + 3.41 dv3p0r002 - 7.92 overload  <- 0;

+ 3.55 dv1p0r000 + 3.55 dv1p0r001 + 3.49 dv2p0r001 + 3.41 dv3p0r000
 - 9.29 overload  <- -1.37;

+ 3.55 dv1p0r003 + 3.41 dv3p0r002 - 14.475 overload  <- -6.38;

+ 3.55 dv1p0r002 + 3.49 dv2p0r001 + 3.41 dv3p0r001 - 12.93 overload
 <- -5.01;

+ 3.44 dv6p0r001 + 3.44 dv6p0r002 - 18.31 overload  <- -10.39;

+ 7.92 dv4p0r001 + 7.92 dv4p0r002 - 4.81 overload  <- 0;

+ 7.92 dv4p0r002 + 3.44 dv6p0r002 - 9.615 overload  <- -4.605;

+ 7.92 dv4p0r001 + 3.44 dv6p0r001 - 11.565 overload  <- -11.565;

+ 7.92 dv4p0r000 + 7.92 dv4p0r001 + 7.92 dv4p0r002 + 3.44 dv6p0r001
 + 3.44 dv6p0r002 - 7.92 overload  <- 0;

+ 7.92 dv4p0r000 + 3.44 dv6p0r000 - 9.29 overload  <- -1.37;

+ 7.92 dv4p0r002 + 3.44 dv6p0r002 - 14.475 overload  <- -10.985;

+ 7.92 dv4p0r000 + 7.92 dv4p0r001 + 7.92 dv4p0r002 + 3.44 dv6p0r000
 + 3.44 dv6p0r001 + 3.44 dv6p0r002 - 12.93 overload  <- -5.01;

+ 1.37 dv5p0r001 - 18.31 overload  <- -18.31;

+ 1.37 dv5p0r000 - 4.81 overload  <- -3.44;

+ 1.37 dv5p0r000 + 1.37 dv5p0r001 - 9.615 overload  <- -4.605;
+ 1.37 dv5p0r001 - 7.92 overload  <= 0;

+ 1.37 dv5p0r001 - 18.31 overload  <= -14.82;

+ 3.49 dv2p0r000 + 1.37 dv5p0r000 - 4.81 overload  <= -3.44;

+ 1.115 dv0p0r000 + 3.49 dv2p0r000 + 3.49 dv2p0r001 + 1.37 dv5p0r000
  + 1.37 dv5p0r001 + 5.01 dv7p0r000 - 9.615 overload  <= -4.605;

+ 3.49 dv2p0r001 + 5.01 dv7p0r000 - 16.37 overload  <= -11.36;

+ 3.49 dv2p0r000 + 3.49 dv2p0r001 + 1.37 dv5p0r001 - 7.92 overload
  <= 0;

+ 3.49 dv2p0r001 - 9.29 overload  <= 0;

+ 3.49 dv2p0r001 - 12.93 overload  <= 0;

+ 5.01 dv7p0r000 - 9.615 overload  <= -4.605;

+ 5.01 dv7p0r000 - 16.37 overload  <= -11.36;

overload <= 100.0;


int dv0p0r000 dv0p1r000 dv0p1r001 dv0p1r002 dv0p1r003 dv1p0r000
  dv1p0r001 dv1p0r002 dv1p0r003 dv2p0r000 dv2p0r001 dv2p1r000
  dv2p1r001 dv2p1r002 dv3p0r000 dv3p0r001 dv3p0r002 dv4p0r000
  dv4p0r001 dv4p0r002 dv5p0r000 dv5p0r001 dv6p0r000 dv6p0r001
  dv6p0r002 dv7p0r000 ;
```

IP-3:

```
- 20 w1 - 30 w2 - 70 w3 - 35 w4 - 50 w5 - 20 w6 - 40 w7 - 35 w8 - 55
  w9 - 20 s1 - 30 s2 - 70 s3 - 35 s4 - 50 s5 - 20 s6 - 40 s7 - 35 s8
  - 55 s9 ;

+ cv0p0 + cv0p1 + cv0p2  = 1;
+ cv1p0 + cv1p1 + cv1p2  = 1;
+ cv2p0 + cv2p1 + cv2p2  = 1;
+ cv3p0 + cv3p1 + cv3p2  = 1;
+ cv4p0 + cv4p1 + cv4p2  = 1;
+ cv5p0 + cv5p1 + cv5p2  = 1;
+ cv6p0 + cv6p1 + cv6p2  = 1;
+ cv7p0 + cv7p1 + cv7p2  = 1;



+ 3.55 cv1p2 + 6.98 cv2p0 + 6.98 cv2p1 + 3.41 cv3p0 + 7.92 cv4p0 +
  7.92 cv4p2 + 3.44 cv6p1 - w1 = 0.0;

+ 2.23 cv0p2 + 3.55 cv1p1 + 3.41 cv3p2 + 7.92 cv4p1 + 1.37 cv5p1 +
  1.37 cv5p2 + 3.44 cv6p0 - w2 = 0.0;

+ 2.23 cv0p1 + 6.98 cv2p1 + 1.37 cv5p2 + 5.01 cv7p2 - w3 = 0.0;

+ 2.23 cv0p0 + 3.55 cv1p0 + 6.98 cv2p0 + 3.41 cv3p0 + 3.41 cv3p2 +
  7.92 cv4p2 + 1.37 cv5p1 + 5.01 cv7p1 - w4 = 0.0;

+ 7.92 cv4p0 + 7.92 cv4p1 + 3.44 cv6p0 + 3.44 cv6p1 + 5.01 cv7p1 +
  5.01 cv7p2 - w5 = 0.0;

+ 3.55 cv1p2 + 6.98 cv2p2 + 3.41 cv3p1 + 3.41 cv3p2 + 7.92 cv4p1 +
  3.44 cv6p1 - w6 = 0.0;

+ 2.23 cv0p2 + 3.55 cv1p1 + 3.55 cv1p2 + 6.98 cv2p2 + 3.41 cv3p1 +
  1.37 cv5p0 + 3.44 cv6p2 - w7 = 0.0;
```

```
+ 2.23 cv0p0 + 2.23 cv0p2 + 6.98 cv2p0 + 6.98 cv2p2 + 1.37 cv5p0 +
 1.37 cv5p1 + 5.01 cv7p0 + 5.01 cv7p1 - w8 = 0.0;

+ 7.92 cv4p2 + 3.44 cv6p2 + 5.01 cv7p0 - w9 = 0.0;

+ dv0p0r000  - cv0p0 = 0;
+ dv0p1r000 + dv0p1r001 + dv0p1r002 + dv0p1r003  - cv0p1 = 0;
+ dv0p2r000  - cv0p2 = 0;
+ dv1p0r000 + dv1p0r001 + dv1p0r002 + dv1p0r003  - cv1p0 = 0;
+ dv1p1r000 + dv1p1r001 + dv1p1r002  - cv1p1 = 0;
+ dv1p2r000 + dv1p2r001 + dv1p2r002  - cv1p2 = 0;
+ dv2p0r000 + dv2p0r001  - cv2p0 = 0;
+ dv2p1r000 + dv2p1r001 + dv2p1r002  - cv2p1 = 0;
+ dv2p2r000  - cv2p2 = 0;
+ dv3p0r000 + dv3p0r001 + dv3p0r002  - cv3p0 = 0;
+ dv3p1r000 + dv3p1r001 + dv3p1r002  - cv3p1 = 0;
+ dv3p2r000 + dv3p2r001  - cv3p2 = 0;
+ dv4p0r000 + dv4p0r001 + dv4p0r002  - cv4p0 = 0;
+ dv4p1r000 + dv4p1r001  - cv4p1 = 0;
+ dv4p2r000 + dv4p2r001  - cv4p2 = 0;
+ dv5p0r000 + dv5p0r001  - cv5p0 = 0;
+ dv5p1r000 + dv5p1r001  - cv5p1 = 0;
+ dv5p2r000 + dv5p2r001 + dv5p2r002  - cv5p2 = 0;
+ dv6p0r000 + dv6p0r001 + dv6p0r002  - cv6p0 = 0;
+ dv6p1r000 + dv6p1r001 + dv6p1r002  - cv6p1 = 0;
+ dv6p2r000 + dv6p2r001  - cv6p2 = 0;
+ dv7p0r000  - cv7p0 = 0;
+ dv7p1r000 + dv7p1r001  - cv7p1 = 0;
+ dv7p2r000  - cv7p2 = 0;


+ 6.98 dv2p0r000 + 6.98 dv2p1r001 + 6.98 dv2p1r002 + 3.41 dv3p0r001
 + 3.41 dv3p0r002 + 7.92 dv4p0r001 + 7.92 dv4p0r002 + 7.92 dv4p2r000
 + 3.44 dv6p1r001 + 3.44 dv6p1r002 - 1 s2  <= 0;

+ 3.55 dv1p2r002 + 6.98 dv2p0r000 + 6.98 dv2p0r001 - 6.98 cv2p1 +
 3.41 dv3p0r002 + 7.92 dv4p0r002 + 7.92 dv4p2r001 + 3.44 dv6p1r002 -
 1 s3  <= 0;

+ 3.55 dv1p2r000 - 6.98 cv2p0 + 6.98 dv2p1r001 - 3.41 cv3p0 + 7.92
 dv4p0r001 - 7.92 cv4p2 + 3.44 dv6p1r001 - 1 s4  <= 0;

+ 3.55 dv1p2r001 + 6.98 dv2p0r001 + 6.98 dv2p1r002 + 3.41 dv3p0r001
 - 7.92 cv4p0 + 7.92 dv4p2r000 + 7.92 dv4p2r001 - 3.44 cv6p1 - 1 s5
 <= 0;

- 3.55 cv1p2 + 6.98 dv2p0r000 + 6.98 dv2p0r001 + 6.98 dv2p1r000 +
 6.98 dv2p1r001 + 6.98 dv2p1r002 + 3.41 dv3p0r000 + 3.41 dv3p0r001 +
 3.41 dv3p0r002 + 7.92 dv4p0r000 + 7.92 dv4p0r001 + 7.92 dv4p0r002 +
 7.92 dv4p2r000 + 7.92 dv4p2r001 - 3.44 cv6p1 - 1 s6  <= 0;

- 3.55 cv1p2 + 6.98 dv2p0r001 + 6.98 dv2p1r000 + 3.41 dv3p0r000 +
 7.92 dv4p0r000 + 7.92 dv4p2r001 + 3.44 dv6p1r000 - 1 s7  <= 0;

+ 3.55 dv1p2r002 - 6.98 cv2p0 + 6.98 dv2p1r000 + 6.98 dv2p1r001 +
 6.98 dv2p1r002 + 3.41 dv3p0r002 + 7.92 dv4p0r002 + 7.92 dv4p2r001 +
 3.44 dv6p1r002 - 1 s8  <= 0;

+ 3.55 dv1p2r001 + 6.98 dv2p0r001 + 6.98 dv2p1r002 + 3.41 dv3p0r001
 + 7.92 dv4p0r000 + 7.92 dv4p0r001 + 7.92 dv4p0r002 - 7.92 cv4p2 +
 3.44 dv6p1r000 + 3.44 dv6p1r001 + 3.44 dv6p1r002 - 1 s9  <= 0;

+ 3.41 dv3p2r000 + 3.41 dv3p2r001 + 7.92 dv4p1r000 + 7.92 dv4p1r001
 + 1.37 dv5p1r000 + 1.37 dv5p2r001 + 1.37 dv5p2r002 + 3.44 dv6p0r001
 + 3.44 dv6p0r002 - 1 s1  <= 0;

+ 2.23 dv0p2r000 + 3.55 dv1p1r002 + 3.41 dv3p2r001 + 7.92 dv4p1r001
 + 1.37 dv5p1r000 + 1.37 dv5p1r001 - 1.37 cv5p2 + 3.44 dv6p0r002 - 1
 s3  <= 0;
```

```
+ 3.55 dv1p1r000 - 3.41 cv3p2 + 7.92 dv4p1r000 - 1.37 cv5p1 + 1.37
 dv5p2r001 + 3.44 dv6p0r001 - 1 s4  <= 0;

+ 3.55 dv1p1r001 + 3.41 dv3p2r000 - 7.92 cv4p1 + 1.37 dv5p1r001 +
 1.37 dv5p2r002 - 3.44 cv6p0 - 1 s5  <= 0;

- 3.41 cv3p2 - 7.92 cv4p1 + 1.37 dv5p1r000 + 1.37 dv5p2r001 + 1.37
 dv5p2r002 + 3.44 dv6p0r001 + 3.44 dv6p0r002 - 1 s6  <= 0;

- 2.23 cv0p2 - 3.55 cv1p1 + 1.37 dv5p1r001 + 1.37 dv5p2r000 + 3.44
 dv6p0r000 - 1 s7  <= 0;

- 2.23 cv0p2 + 3.55 dv1p1r002 + 3.41 dv3p2r001 + 7.92 dv4p1r001 -
 1.37 cv5p1 + 1.37 dv5p2r000 + 1.37 dv5p2r001 + 1.37 dv5p2r002 +
 3.44 dv6p0r002 - 1 s8  <= 0;

+ 3.55 dv1p1r001 + 3.41 dv3p2r000 + 7.92 dv4p1r000 + 7.92 dv4p1r001
 + 1.37 dv5p1r001 + 1.37 dv5p2r002 + 3.44 dv6p0r000 + 3.44 dv6p0r001
 + 3.44 dv6p0r002 - 1 s9  <= 0;

+ 2.23 dv0p1r002 - 6.98 cv2p1 + 1.37 dv5p2r001 + 1.37 dv5p2r002 - 1
 s1  <= 0;

+ 2.23 dv0p1r001 + 6.98 dv2p1r001 + 6.98 dv2p1r002 - 1.37 cv5p2 - 1
 s2  <= 0;

+ 2.23 dv0p1r000 + 6.98 dv2p1r001 + 1.37 dv5p2r001 - 1 s4  <= 0;

+ 2.23 dv0p1r003 + 6.98 dv2p1r002 + 1.37 dv5p2r002 - 5.01 cv7p2 - 1
 s5  <= 0;

+ 2.23 dv0p1r002 + 6.98 dv2p1r000 + 6.98 dv2p1r001 + 6.98 dv2p1r002
 + 1.37 dv5p2r001 + 1.37 dv5p2r002 - 1 s6  <= 0;

+ 2.23 dv0p1r001 + 2.23 dv0p1r002 + 6.98 dv2p1r000 + 1.37 dv5p2r000
 - 1 s7  <= 0;

+ 2.23 dv0p1r000 + 2.23 dv0p1r001 + 2.23 dv0p1r002 + 2.23 dv0p1r003
 + 6.98 dv2p1r000 + 6.98 dv2p1r001 + 6.98 dv2p1r002 + 1.37 dv5p2r000
 + 1.37 dv5p2r001 + 1.37 dv5p2r002 + 5.01 dv7p2r000 - 1 s8  <= 0;

+ 2.23 dv0p1r003 + 6.98 dv2p1r002 + 1.37 dv5p2r002 + 5.01 dv7p2r000
 - 1 s9  <= 0;

+ 3.55 dv1p0r001 - 6.98 cv2p0 - 3.41 cv3p0 + 3.41 dv3p2r000 + 3.41
 dv3p2r001 - 7.92 cv4p2 + 1.37 dv5p1r000 + 5.01 dv7p1r001 - 1 s1  <=
 0;

+ 3.55 dv1p0r000 + 6.98 dv2p0r000 + 3.41 dv3p0r001 + 3.41 dv3p0r002
 - 3.41 cv3p2 + 7.92 dv4p2r000 - 1.37 cv5p1 + 5.01 dv7p1r000 - 1 s2
 <= 0;

+ 2.23 dv0p0r000 + 3.55 dv1p0r003 + 6.98 dv2p0r000 + 6.98 dv2p0r001
 + 3.41 dv3p0r002 + 3.41 dv3p2r001 + 7.92 dv4p2r001 + 1.37 dv5p1r000
 + 1.37 dv5p1r001 + 5.01 dv7p1r000 + 5.01 dv7p1r001 - 1 s3  <= 0;

+ 3.55 dv1p0r002 + 6.98 dv2p0r001 + 3.41 dv3p0r001 + 3.41 dv3p2r000
 + 7.92 dv4p2r000 + 7.92 dv4p2r001 + 1.37 dv5p1r001 - 5.01 cv7p1 - 1
 s5  <= 0;

+ 3.55 dv1p0r001 + 6.98 dv2p0r000 + 6.98 dv2p0r001 + 3.41 dv3p0r000
 + 3.41 dv3p0r001 + 3.41 dv3p0r002 - 3.41 cv3p2 + 7.92 dv4p2r000 +
 7.92 dv4p2r001 + 1.37 dv5p1r000 + 5.01 dv7p1r001 - 1 s6  <= 0;

+ 3.55 dv1p0r000 + 3.55 dv1p0r001 + 6.98 dv2p0r001 + 3.41 dv3p0r000
 + 7.92 dv4p2r001 + 1.37 dv5p1r001 + 5.01 dv7p1r000 + 5.01 dv7p1r001
 - 1 s7  <= 0;

- 2.23 cv0p0 + 3.55 dv1p0r003 - 6.98 cv2p0 + 3.41 dv3p0r002 + 3.41
 dv3p2r001 + 7.92 dv4p2r001 - 1.37 cv5p1 - 5.01 cv7p1 - 1 s8  <= 0;
```

```
+ 3.55 dv1p0r002 + 6.98 dv2p0r001 + 3.41 dv3p0r001 + 3.41 dv3p2r000
 - 7.92 cv4p2 + 1.37 dv5p1r001 + 5.01 dv7p1r000 + 5.01 dv7p1r001 - 1
 s9  <- 0;

- 7.92 cv4p0 + 7.92 dv4p1r000 + 7.92 dv4p1r001 + 3.44 dv6p0r001 +
 3.44 dv6p0r002 - 3.44 cv6p1 + 5.01 dv7p1r001 - 1 s1  <- 0;

+ 7.92 dv4p0r001 + 7.92 dv4p0r002 - 7.92 cv4p1 - 3.44 cv6p0 + 3.44
 dv6p1r001 + 3.44 dv6p1r002 + 5.01 dv7p1r000 - 1 s2  <- 0;

+ 7.92 dv4p0r002 + 7.92 dv4p1r001 + 3.44 dv6p0r002 + 3.44 dv6p1r002
 + 5.01 dv7p1r000 + 5.01 dv7p1r001 - 5.01 cv7p2 - 1 s3  <- 0;

+ 7.92 dv4p0r001 + 7.92 dv4p1r000 + 3.44 dv6p0r001 + 3.44 dv6p1r001
 - 5.01 cv7p1 - 1 s4  <- 0;

+ 7.92 dv4p0r000 + 7.92 dv4p0r001 + 7.92 dv4p0r002 - 7.92 cv4p1 +
 3.44 dv6p0r001 + 3.44 dv6p0r002 - 3.44 cv6p1 + 5.01 dv7p1r001 - 1
 s6  <- 0;

+ 7.92 dv4p0r000 + 3.44 dv6p0r000 + 3.44 dv6p1r000 + 5.01 dv7p1r000
 + 5.01 dv7p1r001 - 1 s7  <- 0;

+ 7.92 dv4p0r002 + 7.92 dv4p1r001 + 3.44 dv6p0r002 + 3.44 dv6p1r002
 - 5.01 cv7p1 + 5.01 dv7p2r000 - 1 s8  <- 0;

+ 7.92 dv4p0r000 + 7.92 dv4p0r001 + 7.92 dv4p0r002 + 7.92 dv4p1r000
 + 7.92 dv4p1r001 + 3.44 dv6p0r000 + 3.44 dv6p0r001 + 3.44 dv6p0r002
 + 3.44 dv6p1r000 + 3.44 dv6p1r001 + 3.44 dv6p1r002 + 5.01 dv7p1r000
 + 5.01 dv7p1r001 + 5.01 dv7p2r000 - 1 s9  <- 0;

- 3.55 cv1p2 + 6.98 dv2p2r000 + 3.41 dv3p1r000 + 3.41 dv3p1r001 +
 3.41 dv3p1r002 + 3.41 dv3p2r000 + 3.41 dv3p2r001 + 7.92 dv4p1r000 +
 7.92 dv4p1r001 - 3.44 cv6p1 - 1 s1  <- 0;

- 3.41 cv3p2 - 7.92 cv4p1 + 3.44 dv6p1r001 + 3.44 dv6p1r002 - 1 s2
 <- 0;

+ 3.55 dv1p2r002 + 6.98 dv2p2r000 + 3.41 dv3p1r002 + 3.41 dv3p2r001
 + 7.92 dv4p1r001 + 3.44 dv6p1r002 - 1 s3  <- 0;

+ 3.55 dv1p2r000 + 3.41 dv3p1r000 - 3.41 cv3p2 + 7.92 dv4p1r000 +
 3.44 dv6p1r001 - 1 s4  <- 0;

+ 3.55 dv1p2r001 + 3.41 dv3p1r001 + 3.41 dv3p2r000 - 7.92 cv4p1 -
 3.44 cv6p1 - 1 s5  <- 0;

- 3.55 cv1p2 - 6.98 cv2p2 - 3.41 cv3p1 + 3.44 dv6p1r000 - 1 s7  <-
 0;

+ 3.55 dv1p2r002 - 6.98 cv2p2 + 3.41 dv3p1r002 + 3.41 dv3p2r001 +
 7.92 dv4p1r001 + 3.44 dv6p1r002 - 1 s8  <- 0;

+ 3.55 dv1p2r001 + 3.41 dv3p1r001 + 3.41 dv3p2r000 + 7.92 dv4p1r000
 + 7.92 dv4p1r001 + 3.44 dv6p1r000 + 3.44 dv6p1r001 + 3.44 dv6p1r002
 - 1 s9  <- 0;

- 3.55 cv1p2 + 6.98 dv2p2r000 + 3.41 dv3p1r000 + 3.41 dv3p1r001 +
 3.41 dv3p1r002 + 1.37 dv5p0r001 + 3.44 dv6p2r001 - 1 s1  <- 0;

- 2.23 cv0p2 - 3.55 cv1p1 + 1.37 dv5p0r000 + 3.44 dv6p2r000 - 1 s2
 <- 0;

+ 2.23 dv0p2r000 + 3.55 dv1p1r002 + 3.55 dv1p2r002 + 6.98 dv2p2r000
 + 3.41 dv3p1r002 + 1.37 dv5p0r000 + 1.37 dv5p0r001 - 1 s3  <- 0;

+ 3.55 dv1p1r000 + 3.55 dv1p2r000 + 3.41 dv3p1r000 - 1 s4  <- 0;

+ 3.55 dv1p1r001 + 3.55 dv1p2r001 + 3.41 dv3p1r001 + 3.44 dv6p2r000
 + 3.44 dv6p2r001 - 1 s5  <- 0;
```

```
- 3.55 cv1p2 - 6.98 cv2p2 - 3.41 cv3p1 + 1.37 dv5p0r001 + 3.44
 dv6p2r001 - 1 s6  <= 0;

- 2.23 cv0p2 + 3.55 dv1p1r002 + 3.55 dv1p2r002 - 6.98 cv2p2 + 3.41
 dv3p1r002 - 1.37 cv5p0 - 1 s8  <= 0;

+ 3.55 dv1p1r001 + 3.55 dv1p2r001 + 3.41 dv3p1r001 - 3.44 cv6p2 - 1
 s9  <= 0;

- 6.98 cv2p0 + 6.98 dv2p2r000 + 1.37 dv5p0r001 + 1.37 dv5p1r000 +
 5.01 dv7p1r001 - 1 s1  <= 0;

- 2.23 cv0p2 + 6.98 dv2p0r000 + 1.37 dv5p0r000 - 1.37 cv5p1 + 5.01
 dv7p1r000 - 1 s2  <= 0;

+ 2.23 dv0p0r000 + 2.23 dv0p2r000 + 6.98 dv2p0r000 + 6.98 dv2p0r001
 + 6.98 dv2p2r000 + 1.37 dv5p0r000 + 1.37 dv5p0r001 + 1.37 dv5p1r000
 + 1.37 dv5p1r001 + 5.01 dv7p0r000 + 5.01 dv7p1r000 + 5.01 dv7p1r001
 - 1 s3  <= 0;

- 2.23 cv0p0 - 6.98 cv2p0 - 1.37 cv5p1 - 5.01 cv7p1 + 6.98 dv2p0r001
 + 1.37 dv5p1r001 + 5.01 dv7p0r000 - 5.01 cv7p1 - 1 s5  <= 0;

+ 6.98 dv2p0r000 + 6.98 dv2p0r001 - 6.98 cv2p2 + 1.37 dv5p0r001 +
 1.37 dv5p1r000 + 5.01 dv7p1r001 - 1 s6  <= 0;

- 2.23 cv0p2 + 6.98 dv2p0r001 - 6.98 cv2p2 - 1.37 cv5p0 + 1.37
 dv5p1r001 + 5.01 dv7p1r000 + 5.01 dv7p1r001 - 1 s7  <= 0;

+ 6.98 dv2p0r001 + 1.37 dv5p1r001 - 5.01 cv7p0 + 5.01 dv7p1r000 +
 5.01 dv7p1r001 - 1 s9  <= 0;

- 7.92 cv4p2 + 3.44 dv6p2r001 - 1 s1  <= 0;

+ 7.92 dv4p2r000 + 3.44 dv6p2r000 - 1 s2  <= 0;

+ 7.92 dv4p2r001 + 5.01 dv7p0r000 - 1 s3  <= 0;

- 7.92 cv4p2 + 7.92 dv4p2r000 + 7.92 dv4p2r001 + 3.44 dv6p2r000 +
 3.44 dv6p2r001 + 5.01 dv7p0r000 - 1 s5  <= 0;

+ 7.92 dv4p2r000 + 7.92 dv4p2r001 + 3.44 dv6p2r001 - 1 s6  <= 0;

+ 7.92 dv4p2r001 - 3.44 cv6p2 - 1 s7  <= 0;

+ 7.92 dv4p2r001 - 5.01 cv7p0 - 1 s8  <= 0;

 w1 <= 34.01;
 s1 <= 34.01;
 w2 <= 34.01;
 s2 <= 34.01;
 w3 <= 34.01;
 s3 <= 34.01;
 w4 <= 34.01;
 s4 <= 34.01;
 w5 <= 34.01;
 s5 <= 34.01;
 w6 <= 34.01;
 s6 <= 34.01;
 w7 <= 34.01;
 s7 <= 34.01;
 w8 <= 34.01;
 s8 <= 34.01;
 w9 <= 34.01;
 s9 <= 34.01;


int cv0p0 dv0p0r000 cv0p1 dv0p1r000 dv0p1r001 dv0p1r002 dv0p1r003
 cv0p2 dv0p2r000 cv1p0 dv1p0r000 dv1p0r001 dv1p0r002 dv1p0r003
 cv1p1 dv1p1r000 dv1p1r001 dv1p1r002 cv1p2 dv1p2r000 dv1p2r001
 dv1p2r002 cv2p0 dv2p0r000 dv2p0r001 cv2p1 dv2p1r000 dv2p1r001
```

```
dv2p1r002 cv2p2 dv2p2r000 cv3p0 dv3p0r000 dv3p0r001 dv3p0r002
cv3p1 dv3p1r000 dv3p1r001 dv3p1r002 cv3p2 dv3p2r000 dv3p2r001
cv4p0 dv4p0r000 dv4p0r001 dv4p0r002 cv4p1 dv4p1r000 dv4p1r001
cv4p2 dv4p2r000 dv4p2r001 cv5p0 dv5p0r000 dv5p0r001 cv5p1
dv5p1r000 dv5p1r001 cv5p2 dv5p2r000 dv5p2r001 dv5p2r002 cv6p0
dv6p0r000 dv6p0r001 dv6p0r002 cv6p1 dv6p1r000 dv6p1r001 dv6p1r002
cv6p2 dv6p2r000 dv6p2r001 cv7p0 dv7p0r000 cv7p1 dv7p1r000
dv7p1r001 cv7p2 dv7p2r000 ;
```

- **Summary of IP-1 formulation size of tested networks**

In the following table, the number of variables and constraints used in all tested networks.

TABLE 11. IP Formulation Size for All Tested Networks

Network	# of constraints	# of variables
Smallnet	541	1332
Net1	610	738
Net2	2506	3077
Net3	2413	2811
Net4	3810	4152
Toronto	858	1339
US	1051	1994
tiny	60	35

Appendix D: Overload Assessment Simulation Program

- **Traffic generation module**

  This module can simulate both the on/off fluid model and AR model using a conditional compile option, IPP and AR respectively.

  Makefile: ./traffic/makefile

```
NI = ../../include
NS = ../../lib
CC = gcc
CFLAGS = -I$(NI) -L$(NS) -g -Wall
OPT=-O3

%.o:    %.c traffic.h
        $(CC) $(CFLAGS)  -O3 -c $*.c
        ar rcv $(NS)/libns.a $*.o
        ranlib $(NS)/libns.a

all:    $(NI)/traffic.h artraffic.o traffic.o

$(NI)/traffic.h: traffic.h
        cp traffic.h $(NI)

artraffic.o: traffic.c
        $(CC) $(CFLAGS)  -DAR -O3 -o $*.o -c traffic.c
        ar rcv $(NS)/libns.a $*.o
        ranlib $(NS)/libns.a

traffic.o: traffic.c
        $(CC) $(CFLAGS)  -DIPP -O3 -o $*.o -c traffic.c
        ar rcv $(NS)/libns.a $*.o
        ranlib $(NS)/libns.a

clean:
        rm -f *.o core *BAK

indent:
        indent -i4 traffic.h
        indent -i4 traffic.c

report: *.h *.c Makefile Readme
        enscript *.h *.c Makefile Readme
        echo >report
```

  Main module: ./traffic/traffic.c

```
#include <stdio.h>
#include <stdlib.h>
#include <malloc.h>
#include <assert.h>
#include <values.h>
#include <sys/times.h>
#include <sys/time.h>
#include <unistd.h>
#include <math.h>
#include <string.h>
#include "traffic.h"

#define LOGFILE "simulation.log"
```

```
/* state options */
#define IDLE     0
#define ACTIVE   (!IDLE)

#define SIMULPERIODS100000.0
#define SIMULHIST500000.0

#defineFRAMESPERSECOND30.0
#definePERIOD          (1000.0/FRAMESPERSECOND)
#defineDEFAULT         0.54           /* a little higher than mean
 value */

#defineLOWCLR          1e-5
#defineHIGHCLR         1e-9

#defineMAX(a, b)(((a)>(b))? (a) : (b))
#defineMIN(a, b)(((a)<(b))? (a) : (b))

#defineRANDOM()(drand48())

#define upround(x)( (double) ((int) (x * PRECISION + 1)) / PRECISION
 )
#define downround(x)( (double) ((int) (x * PRECISION)) / PRECISION )
#defineind(x)          ( (( (int) (x * PRECISION) ) > histtop) ?
 histtop : ( (int) (x * PRECISION) ) )
#define inc(k) ++k; if (k > histtop) k = histtop;
#define dec(k) --k; if (k < 0) k = 0;

#ifdef IPP
#definePRINTSOURCE{ \
        int ind; \
        printf ("current %f, until %f, next %d\n", lasttime, simul-
 time, next); \
        printf ("active #%d, rate %f, traffic %f\n", numactive,
 rateactive, totalcell); \
        for (ind = 0; ind < sourcenum; ind++) { \
            printf("s #%d: %s, t %f, p %f, u %f, b %f, i %f\n", \
            ind, (sources[ind].state == IDLE) ? "IDLE  " : "ACTIVE",
 sources[ind].endtime, \
            sources[ind].peak, sources[ind].utilization, \
            sources[ind].burst, sources[ind].idleburst); \
        } \
         if (totalcell != 0.0) \
         for (ind = 0; ind < queuenum; ind++) { \
             printf("q #%d: cont %f, lost %f, clr %f%%\n", \
             ind, queue[ind].bufcontent, queue[ind].totallost, \
                queue[ind].totallost/totalcell*100); \
         }  \
        }
#else  /* AR */
#definePRINTSOURCE{ \
        int ind; \
        printf ("\n\n\ncurrent %f, until %f, next %d\n", lasttime,
 simultime, next); \
        printf ("active #%d, rate %f, traffic %f\n", numactive,
 rateactive, totalcell); \
        for (ind = 0; ind < sourcenum; ind++) { \
            printf("s #%d: t %f, rate %f, a %f, b %f, mean %f\n", \
            ind, sources[ind].endtime, sources[ind].peak,
 sources[ind].a, \
            sources[ind].b, sources[ind].mean); \
        } \
         if (totalcell != 0.0) \
         for (ind = 0; ind < queuenum; ind++) { \
             printf("q #%d: cont %f, lost %f, clr %f%%\n", \
             ind, queue[ind].bufcontent, queue[ind].totallost, \
                queue[ind].totallost/totalcell*100); \
         } \
        }
#endif
```

```
struct imodel {
    int              tag;/* keep a tag, as the entries are sorted */
    double           peak;/* IPP: peak rate */
                              /* AR:  current */
    double           utilization;/* IPP: utilization, percentage of
 time of active */
    double           burst;/* IPP: mean length of active */
    double           idleburst;/* IPP: mean length of idle */
    int              state;/* IPP: current state */
    double     a, b, mean; /* AR:  parameters */
    double           endtime;/* end of simulation time of current
 state */
} *sources;

static char
            *
currentdate(void)
{
    struct timeval  tp;
    struct timezone tzp;

    gettimeofday(&tp, &tzp);
    return (asctime(localtime(&(tp.tv_sec))));
}


/* get current time and set as the seed of random list */
static long
SEEDRANDOM(long seed)
{
    struct tms      current;
    static time_t   lastseed;

    if (seed == 0l) {
       while ((seed = times(&current)) == lastseed)
           sleep(1);
    }
    lastseed = seed;
    srand48(seed);
    return (seed);
}


#definepoisson(mean)(-log(RANDOM()) * (mean))

#if    0
static double
poisson(double mean)
{
    return (-log(RANDOM()) * mean);
    /* the following is the conterpart test in MATLAB */
    /*
     * clear m = 80 a = [0.00000001:0.000001:1]; b= -log(a) * m;
 mean(b)
     * std(b)
     *
     */


    /* the following is for the integer poisson variable */
#if    0
    int             i;

    double          u, p, f;
    i = 0;
    f = p = exp(-mean);
    u = RANDOM();
    while (f <= u) {
        p *= (mean / (i + 1.0));
        f += p;
        i++;
    }
    return (i);
#endif
}

#endif


#ifdef AR

/* generate the rate of next period */

static double gaussian (double a, double d)
{
    static double t = 0.0;
    double x, v1, v2, r;
    if (t == 0.0) {
        do {
            v1 = 2.0 * RANDOM() - 1.0;
            v2 = 2.0 * RANDOM() - 1.0;
            r = v1 * v1 + v2 * v2;
        } while (r >= 1.0);
        r = sqrt((-2.0 * log(r)) / r);
        t = v2 * r;
        return (a + v1 * r * d);
    } else {
        x = t;
        t = 0.0;
        return (a + x * d);
    }
}

static double nextspeed(int k) {
    double next;

    next = sources[k].a * sources[k].peak + sources[k].b * gauss-
 ian(sources[k].mean, 1.0);
    return (next < 0.0) ? 0.0 : next;
}


#endif /* AR */

static long     seed[10] = {    0x7123456789abl, 0x987345abd301l,
                            0x3102845ba7dfl, 0xab7269e64083l,
                            0x197629ef2a8cl, 0x981234fc8321l,
                            0x9234985178cal, 0xabc837e91f38l,
                            0x1f87167ea981l, 0x23745798ceafl};

static double         *hist;
static  int    histtop;

#ifdef IPP
void
simul_hist(int trafnum, struct traffic traffic[], double capacity,
        int histnum, double histtotal[])
#else  /* AR */
void
ar_hist(int trafnum, struct traffic traffic[], double capacity,
         int histnum, double histtotal[])
#endif

{
    int               seednum;

    FILE            *fp;
    int              i, j;
    struct queue     queue;
    /* open log file */
    fp = fopen(LOGFILE, "a+");
    assert(fp);

    fprintf(fp, "Histogram analysis starts now...\n");

    fclose(fp);


    seednum = sizeof(seed) / sizeof(seed[0]);

    histtop = histnum;
    hist = (double *) malloc ((histtop+1) * sizeof(hist[0]));
    assert(hist);

    for (i = histtop; i >= 0; i--) {
       histtotal[i] = 0.0;
    }
    queue.capacity = capacity;

    /* run simulation with each seed */
    for (i = 0; i < seednum; i++) {
       /* simulation, ahha */
#ifdef IPP
       simul_cap2clr(seed[i], trafnum, traffic, 0, &queue);
#else  /* AR */
       ar_cap2clr(seed[i], trafnum, traffic, 0, &queue);
#endif
       for (j = histtop; j >= 0; j--) {
           histtotal[j] += hist[j];
       }
    }

    fp = fopen(LOGFILE, "a+");
    assert(fp);
    fprintf(fp, "\n\t Total result is as follows\n");
    for (j = histtop; j >= 0; j--) {
       histtotal[j] /= seednum;
       fprintf(fp, "\t%1.2f\t%e\n", 1.0*i/PRECISION, histtotal[j]);
    }

    free(hist);

    fprintf(fp, "Histogram analysis ends now...\n\n\n\n");
    fclose(fp);


}

#ifdef IPP
float
simul_overload(int trafnum, struct traffic traffic[],
               double capahigh, double capalow, double capastep,
               double bufsize)
#else  /* AR */
float
ar_overload(int trafnum, struct traffic traffic[],
               double capahigh, double capalow, double capastep,
               double bufsize)
#endif

{
    int             seednum;

    int             queuenum;
    struct queue   *queue;
    double          clrlowcapa, clrhighcapa;
    double          highest = 0.0, lowest = 100.0;
    float           overload;
    FILE            *fp;
    int              i, j;

    /* open log file */
    fp = fopen(LOGFILE, "a+");
    assert(fp);

    fprintf(fp, "Overload analysis starts now...\n");

    fclose(fp);


    seednum = sizeof(seed) / sizeof(seed[0]);

    queuenum = (capahigh - capalow) / capastep + 1;
    queue = (struct queue *) malloc(queuenum * sizeof(queue[0]));
    assert(queue);

    for (i = 0; i < queuenum; i++) {
       queue[i].capacity = capalow + i * capastep;
       queue[i].bufsize = bufsize;
    }

    /* run simulation with each seed */
    for (i = 0, overload = 0.0; i < seednum; i++) {
       /* simulation, ahha */
#ifdef  IPP
       simul_cap2clr(seed[i], trafnum, traffic, queuenum, queue);
#else   /* AR */
       ar_cap2clr(seed[i], trafnum, traffic, queuenum, queue);
#endif

       fp = fopen(LOGFILE, "a+");
       assert(fp);
       fprintf(fp, " analysis of last simulation. \n");

       if (queue[0].clr < LOWCLR) {
           fprintf(fp, " FATAL error, capacity low bound is too
 big\n");
           fclose(fp);
           free(queue);
           return (0.0);
       } else {
           if (queue[queuenum - 1].clr > HIGHCLR) {
              fprintf(fp, " FATAL error, capacity high bound is too
 small\n");
              fclose(fp);
              free(queue);
              return (0.0);
           } else {
              /* find the overload treshhold capacities */
              for (j = 0; j < queuenum; j++) {
                  if (queue[j].clr < LOWCLR)
                     break;
              }
              clrlowcapa = queue[j].capacity;
              lowest = MIN(lowest, queue[j - 1].capacity);

              for (; j < queuenum; j++) {
                  if (queue[j].clr < HIGHCLR)
                     break;
              }
              clrhighcapa = queue[j].capacity;
              highest = MAX(highest, clrhighcapa);

              /* get overload factor */
              overload += (clrhighcapa / clrlowcapa);

              fprintf(fp, " when clr = %e, capacity is %1.3f\n",
 LOWCLR, clrlowcapa);
              fprintf(fp, " when clr = %e, capacity is %1.3f\n",
 HIGHCLR, clrhighcapa);
                fprintf(fp, "  thus the overload is %1.3f\n\n\n",
 clrhighcapa / clrlowcapa);
                fclose(fp);
            }
        }

    }

    overload /= seednum;

    free(queue);

    fp = fopen(LOGFILE, "a+");
    assert(fp);
    fprintf(fp, " final overload is %1.3f\n", overload);
    fprintf(fp, " lowest is %1.3f, highest is %1.3f\n\n\n", lowest,
 highest);
    fprintf(fp, "Overload analysis ends now...\n\n\n\n");
    fclose(fp);


    return (overload);

}


#ifdef AR
int srccomp(const void *src1, const void *src2) {
    return ( ((struct imodel *) src1)->endtime - ((struct imodel *)
 src2)->endtime);
}
#endif


/* record history of queue if queuenum is 0 */

#ifdef IPP
void
simul_cap2clr(long seed,
               int trafnum, struct traffic traffic[],
               int queuenum, struct queue queue[])
#else  /* AR */
void
ar_cap2clr(long seed,
               int trafnum, struct traffic traffic[],
               int queuenum, struct queue queue[])
#endif

{
    int             i, j, m, k, next, g;
    int             sourcenum, numactive;
    double          rateactive;
    double         thistotal, t, ratet1, t1, totalcell;

    double          simultime = 0.0, lasttime, duration;
    double          current;
    FILE           *fp;
    char            hostname[16], domainname[16];

    extern          getdomainname(char *, int);

    /* open log file */
    fp = fopen(LOGFILE, "a+");
    assert(fp);

    /* write log file */
    gethostname(hostname, 16);
    getdomainname(domainname, 16);
    fprintf(fp, "\nStarting simulation in %s.%s at %s",
           hostname, domainname, currentdate());
    /* plant the seed */
    if (seed == 0l)
       fprintf(fp, " seed not specified.\n");
    seed = SEEDRANDOM(seed);
    fprintf(fp, " using seed 0x%lx", seed);

    /* calculate the number of individule sources */
    for (i = 0, sourcenum = 0; i < trafnum; i++) {
       for (j = 0; j < traffic[i].num; j++) {
           sourcenum += traffic[i].group[j].num;
       }
    }

    /* set up each source in the simulation context */
    sources = (struct imodel *) malloc(sourcenum * sizeof(struct
 imodel));
    assert(sources);

    fprintf(fp, "\n %d source(s) as following:\n", sourcenum);

    /* initialize each traffic */
    for (i = 0, k = 0, numactive = 0, rateactive = 0.0; i < trafnum;
 i++) {
       /* try each group */
       for (j = 0; j < traffic[i].num; j++) {
           /* try each source model */
           for (m = 0; m < traffic[i].group[j].num; m++, k++) {
              /* constant parameter */
              sources[k].tag = k;
#ifdef IPP
              sources[k].peak = traffic[i].group[j].model.peak;
              sources[k].utilization = traf-
 fic[i].group[j].model.utilization;
              sources[k].burst = traffic[i].group[j].model.burst;
              sources[k].idleburst = (sources[k].utilization ==
 0.0) ? MAXDOUBLE :
                     (sources[k].burst * (1 - sources[k].utilization)
 / sources[k].utilization);
              if ((sources[k].utilization != 0.0) &&
                     (simultime < sources[k].burst / sources[k].utili-
 zation))
                     simultime = sources[k].burst / sources[k].utili-
 zation;
              fprintf(fp, "\t%d: peak %f, util %f, burst %f, idle-
 burst %f\n",
                          k, sources[k].peak, sources[k].utilization,
                          sources[k].burst, sources[k].idleburst);
              /* initial state of variables */
              sources[k].state = (RANDOM() < sources[k].utiliza-
 tion) ? ACTIVE : IDLE;
              sources[k].endtime = poisson((sources[k].state ==
 ACTIVE) ? sources[k].burst :
                                                   sources[k].idleburst);
              if (sources[k].state == ACTIVE) {
                  numactive++;
                  rateactive += sources[k].peak;
              }
#else  /* AR */
              sources[k].a = traffic[i].group[j].model.a;
              sources[k].b = traffic[i].group[j].model.b;
              sources[k].mean = traffic[i].group[j].model.mean;
              fprintf(fp, "\t%d: a %f, b %f, mean %f\n",
                          k, sources[k].a, sources[k].b,
 sources[k].mean);
              /* initial state of variables */
              sources[k].peak = DEFAULT;
              sources[k].peak = nextspeed(k);
              sources[k].endtime = (RANDOM() * PERIOD);
              rateactive += sources[k].peak;
#endif
```

```
            }
        }
    }

#ifdef IPP
    simultime *= (queuenum == 0) ? SIMULHIST : SIMULPERIODS;
#else  /* AR */
    simultime = (queuenum == 0) ? PERIOD * 10000 : PERIOD * 50000;
#endif


    if (queuenum == 0) {
        fprintf(fp, "\n this is to record histogram when capacity is
 %f\n", queue[0].capacity);
    } else {
        fprintf(fp, "\n %d queue(s) as following:\n", queuenum);
        for (g = 0; g < queuenum; g++) {
            queue[g].clr = 0.0;
            fprintf(fp, "\t #%d: capacity %f, size %f\n",
                    g, queue[g].capacity, queue[g].bufsize);
        }
    }

    for (g = 0; g < queuenum; g++) {
        queue[g].totallost = 0.0;
        queue[g].bufcontent = 0.0;
    }
    if (queuenum == 0) {
        queue[0].bufcontent = 0.0;
        for (i = histtop; i >= 0; i--) {
            hist[i] = 0;
        }
    }
    totalcell = 0.0;
    lasttime = 0.0;

#ifdef AR
    /* sort */
    qsort(sources, sourcenum, sizeof(sources[0]), srccomp);
    next = -1;
#endif

    /* start real simulation */
    while (1) {


        /* find the next event time */
#ifdef IPP
        for (i = sourcenum - 1, next = 0, current = sources[0].end-
 time; i >= 1; i--) {
            if (current > sources[i].endtime) {
                next = i;
                current = sources[next].endtime;
            }
        }
#else  /* AR */
        if (++next >= sourcenum) {
            next = 0;
        }
#endif

        /* check simulation end */
        if ((lasttime >= simultime) && (sources[next].endtime !=
 lasttime))
            break;

        /* update buffer content and statistics until current event
 */
        duration = sources[next].endtime - lasttime;
        thistotal = rateactive * duration;
```

```
      if (queuenum == 0) {
          if (rateactive > queue[0].capacity) {
              /* rate is greater than capacity */
              t = (upround(queue[0].bufcontent) - queue[0].bufcon-
tent) /
                   (rateactive - queue[0].capacity);
              k = ind(queue[0].bufcontent);
              if (t >= duration) {
                   hist[k] += thistotal;
                   totalcell += thistotal;
              } else {
                   hist[k] += t * rateactive;
                   totalcell += t * rateactive;
                   inc(k);
                   t1 = 1.0 / PRECISION / (rateactive -
queue[0].capacity);
                   rate t1 = rateactive * t1;
                   t += t1;
                   while (t < duration) {
                       hist[k] += ratet1;
                       totalcell += ratet1;
                       inc(k);
                       t += t1;
                   }
                   hist[k] += (duration - t + t1) * rateactive;
                   totalcell += (duration - t + t1) * rateactive;
              }
          }
          if (rateactive == queue[0].capacity) {
              /* rate is equal to capacity */
              hist[ind(queue[0].bufcontent)] += thistotal;
              totalcell += thistotal;
          }
          if (rateactive < queue[0].capacity) {
              /* rate is less than capacity */
              t = ( queue[0].bufcontent - downround(queue[0].buf-
content) ) /
                   (queue[0].capacity - rateactive );
              k = ind(downround(queue[0].bufcontent));
              if (t >= duration) {
                   hist[k] += thistotal;
                   totalcell += thistotal;
              } else {
                   hist[k] += t * rateactive;
                   totalcell += t * rateactive;
                   dec(k);
                   t1 = 1.0 / PRECISION / (queue[0].capacity - rate-
active );
                   ratet1 = rateactive * t1;
                   t += t1;
                   while (t < duration) {
                       hist[k] += ratet1;
                       totalcell += ratet1;
                       dec(k);
                       t += t1;
                   }
                   hist[k] += (duration - t + t1) * rateactive;
                   totalcell += (duration - t + t1) * rateactive;
              }
          }
          /* update buffer content */
          queue[g].bufcontent -= duration * (queue[g].capacity -
rateactive);
          if (queue[g].bufcontent < 0) {
              queue[g].bufcontent = 0;
          }
      } else {
          totalcell += thistotal;
          for (g = queuenum - 1; g >= 0; g--) {
              queue[g].bufcontent -= duration * (queue[g].capacity
- rateactive);
```

```
                if (queue[g].bufcontent < 0) {
                    queue[g].bufcontent = 0;
                } else if (queue[g].bufcontent > queue[g].bufsize) {
                    queue[g].totallost += queue[g].bufcontent -
 queue[g].bufsize;
                    queue[g].bufcontent = queue[g].bufsize;
                }
            }
        }
        lasttime = sources[next].endtime;

        /* update the transient traffic */
#ifdef IPP
        if (sources[next].state == ACTIVE) {
            numactive--;
            rateactive -= sources[next].peak;
            sources[next].endtime += poisson(sources[next].idle-
 burst);
        } else {
            numactive++;
            rateactive += sources[next].peak;
            sources[next].endtime += poisson(sources[next].burst);
        }
        sources[next].state = !sources[next].state;
#else   /* AR */
        rateactive -= sources[next].peak;
        sources[next].peak = nextspeed(next);
        rateactive += sources[next].peak;
        sources[next].endtime += PERIOD;
#endif
    }


    /* gather clr */
    if (queuenum != 0) {
        fprintf(fp, "\n\t clr in this simulation\n");
        for (g = 0; g < queuenum; g++) {
            queue[g].clr = queue[g].totallost / totalcell;
            fprintf(fp, "\t  capacity %1.3f, %e\n",
                    queue[g].capacity, queue[g].totallost / total-
 cell);
        }
    } else {
        /* print bufcontent */
        fprintf(fp, "\n\t total traffic %f\n", totalcell);
        fprintf(fp, "\n\t buffer content histogram in this simula-
 tion\n");
        for (i = 0; i <= histtop; i++) {
            fprintf(fp, "\t%1.2f\t%e\t%f\n", 1.0*i/PRECISION,
 hist[i]/totalcell, hist[i]);
            hist[i] /= totalcell;
        }
    }

    free(sources);

    fprintf(fp, "\n\nEnding simulation %s\n", currentdate());
    fclose(fp);

    return;
}
```