Simulation of switching architectures for Optical Packet Switched network

Elena Segundo Martín

Thesis Advisors: **Franco Callegati, Walter Cerroni**

Dipartimento di Elettronica, Informatica e Sistemistica
Univesità di Bologna, Facoltà di Ingegneria

September 2010

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

In recent years, the Internet traffic has grown following an exponential behaviour as figure shown in [1] and it is expected to continue in future. One of the reasons is the key role assumed by information technologies. Internet users are consuming more bandwidth than ever. In particular, a real revolution involved the way people can communicate, leaded by two major factors: the huge success of mobile communications and the birth of new forms of information to be exchanged. People can communicate every time and everywhere, exchanging e-mails, browsing hypertexts, buying things online, accessing all kinds of information in huge databases, downloading music, movies and other multimedia contents.

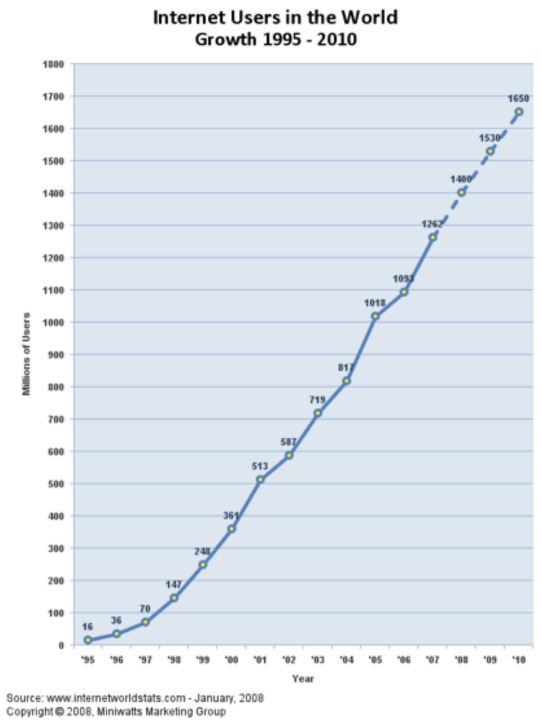


Figure 1. Internet traffic growth prediction

This rapidly rising eagerness for network bandwidth stimulates new advances in optical technologies. Optical networks offer fast, readily available bandwidth at reduced costs. They are ideal for broad range of Internet applications, including high-quality video delivery for entertainment purposes.

Advances in lightwave communications technology over the past several years created the opportunity to share the enormous optical bandwidth among multiple users in local, metropolitan, and wide area networks.

1.2 Goal and related work

A key component of optical networks is the switching node that allows routing the data flows along the correct destinations. The aim of this project is to implement a generic optical burst/packet switching node architecture, describing contention resolution techniques and trying to define an optimal solution. Optical networking paradigms based on statistical multiplexing, such as Optical Burst Switching or Optical Packet Switching, require the adoption of suitable contention resolution mechanisms at the optical nodes due to packets/bursts attempting to get access to the shared output channel at the same time. In the most general case of multi-fiber output links, contentions can be solved by exploiting different domains –namely space, wavelength and time- and by applying an optimal scheduling policy.

The simulator developed in this thesis will study two different scheduling policies: delay oriented, which tries to schedule packets with the smaller delay available, and gap oriented which tries to minimize the void remaining before the packet scheduled. In both cases void filling and no void filling algorithms will be considered. Void filling algorithms are those that allow scheduling a packet in a void even when other packets have been scheduled later in the same output channel, while non void filling algorithms only allow scheduling packets into the final void of each channel.

The project includes a study of the impact of a Limited Wavelength Conversion and analyzes its repercussion on cost and power consumption, depending on the number and the type of converters used. By simulation, we try to define some guidelines to minimize the contribution to the total cost and consumption due to the converters.

This thesis will describe contention resolution techniques that have been studied at the Electronics Department from the University of Bologna in Chapter 2. In chapter 3, the need of an evaluation of cost of an optical burst/packet switching node is explained. Afterwards, chapter 4 focuses on the simulator that allows obtaining the simulation results that are shown in chapter 5. And finally, chapter 6 presents some conclusions.

Chapter 2

Scheduler design for optical burst/packet switching

Switching takes place on the nodes of the network and its objective is to allow the transmission of information from the source node to the destination node. In classical networks there are two main switching techniques: circuit switching and packet switching. The first one finds its typical application in telephony, where each communication is associated to a circuit and the second one in Internet, where data are broken down into packets. Switching concepts can be used in optical networks. The main techniques in the field of optical switching [3] are:

- Optical circuit switching
- Optical burst switching
- Optical packet switching
- Optical code switching

All these techniques use WDM technology (Wavelength Division Multiplexing), exploit the wavelength multiplexing and can be implemented in dynamic mode, by using optical switches associated with a control plan to allow rapid reconfiguration of the network. However, implementation of optimal solutions requires more complex and faster hardware.

Although the different techniques available, this thesis focuses on Optical Burst Switching (OBS) and Optical Packet Switching (OPS).

2.1 OBS/OPS Scheduling

Optical Packet Switching [2] is based on the same statistical multiplexing concepts as its electronic counterpart, resulting in a more efficient resource utilization than the optical circuit switched networks. On the other hand, Optical Burst Switching aims at improving wavelength utilization with respect to circuit switched optical networks while adopting more feasible dynamic wavelength management techniques, resulting in a medium term solution. The basic idea behind OBS is to set up a wavelength path through the network on the fly reserved to a large data flow, called burst. The burst is transferred transparently along this path, which is reset at the end of the transfer.

The main differences between OPS and OBS are:

- In OBS control information for each burst is typically transmitted in advance on a separate channel and in OPS, it is inserted in header in front of the packet.
- Typically no buffering is implemented at any intermediate node in OBS.
- Bursts are built as aggregates of packets, thus a larger time-scale is considered.

2.2 Contention resolution in optical burst/packet switching nodes

Optical networking techniques require the adoption of suitable contention resolution mechanisms at the optical nodes. The main problem arises that packets attempt to get access to the shared switching and transmission resources at the same time. In optical technologies queuing techniques cannot be used as in electronic networks, due to the lack of optical RAMs. Exploiting three domains [4] can solve this problem:

- Space, transmitting packets on different wavelengths of the same fiber at the same time.
- Wavelength, transmitting packets on different wavelengths. If the output wavelength is different from the input one, converters are needed.
- Time, using delay lines as an optical buffer, in order to delay packet transmission.

In this case, the chosen solution is based on algorithms being able to proactively schedule the transmission of the packets in order to avoid contention as much as possible. The algorithms calculate the exact time when the transmission of the packet must success.

2.3 Model of a switching node architecture

The general architecture of an optical packet switch is presented in figure 2 [5], showing the main functional blocks:

- Input interfaces, used to demultiplex the wavelengths on incoming fibers, to synchronize packets and to tap some optical power, in order to extract the header
- Optical space switch, a switching matrix used to physically interconnect the input and output ports
- Delay line buffer, used to solve contentions in the time domain
- Output interfaces, used to insert a new header and to multiplex wavelengths back to the fiber
- Electronic control logic, used to perform header processing and routing table lookup

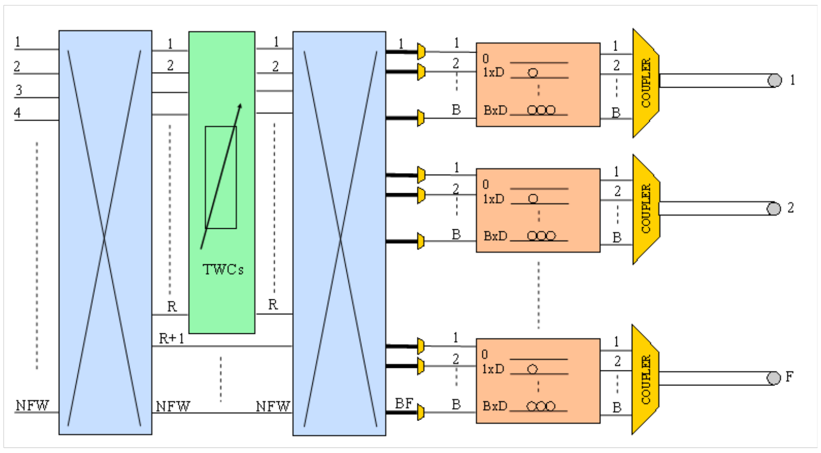


Figure 2. Structure of an optical switching node

When a burst/packet arrives at the node, the scheduler must select three parameters: the output fiber, the output wavelength and the time when the packet will be transmitted, and therefore the delay required if any.

In electronic packet switching, contention resolution in the time domain is based on a store-and-forward scheme, where contending packets are stored in a queue as long as the output port is available again. Optical queuing may be realized with a number of fiber coils of given length used as delay lines, it takes some time for a packet injected inside a long fiber trunk to come out at the other side. This optical buffer is not completely equivalent to a conventional queue. The main difference is that, while a packet may be stored as long as needed into an electronic memory, it cannot stay within the optical buffer for longer than an amount of time given by the propagation delay inside the delay line utilized. In case such delay is not long enough, either another contention resolution is performed or the packet is lost.

An optical buffer is made by *B* fiber delay lines with different lengths providing delays which are multiple of a basic delay unit *D*, also called *buffer granularity*. In the *B* fiber delay lines, the zero-delay is included, so the packet can be delayed from *D* to (*B*-1)·*D* or not be delayed.

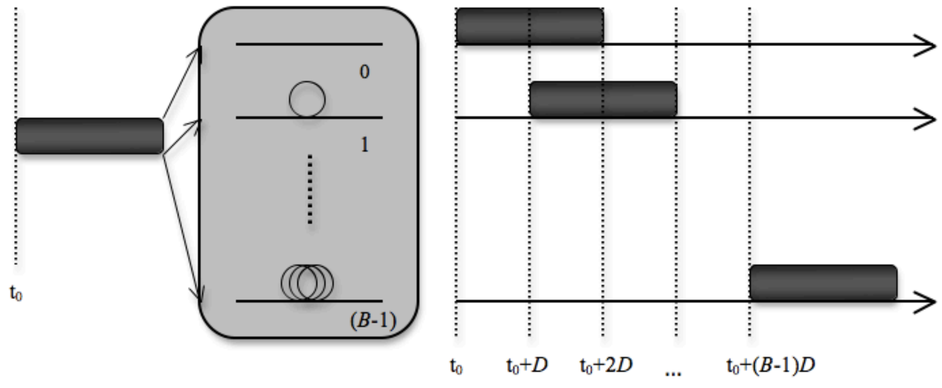


Figure 3. An example of optical fiber delay lines buffer

2.4 Key parameters for contention resolution

The key parameters that should be taken into account to design the node architecture are the following:

- The number of fibers per output link: *F*
- The number of wavelengths per fiber: *W*
- The wavelength conversion capability, in terms of the number of wavelengths per conversion waveband: *L*
- The number of available wavelength converters per output interface: *R*
- The number of delays: *B* and the delay unit: *D*
- The type of scheduling algorithm that are explained in the next point: *D-type VF, D-type NoVF, G-type VF and G-type NoVF*

These parameters determine the dimensions of the scheduling space *S*, with cardinality:

- $|S| = B \cdot W \cdot F$ in case of Full Wavelength Conversion

Or

- $|S'| = B \cdot L \cdot F$ in case of Limited Wavelength Conversion

The term *channel* is normally used in this work, refers to each wavelength in each fiber, so there are $C = W \cdot F$ or $C = L \cdot F$ different channels in both cases.

2.5 Channel and Delay Selection algorithms (CDS)

Once the output channel is known, there are three parameters that the scheduler should be able to select, the output fiber, the output wavelength and the delay. The way to find them depends on the algorithm, which are explained in 2.5.2. First of all, it would be useful introduce some concepts briefly.

2.5.1 Concept of Gap

Optical hardware issues pose a number of constraints to the scheduling problem. The most obvious one is time constraints related to the discrete characteristics of the FDLs, as packets can only be delayed to a finite subset of delays. Therefore, voids or gaps will appear between scheduled packets. As a consequence, efficiency can be significantly reduced because of this discrete nature. On the other hand, depending on the nature of wavelength converters available, a packet may not be transmitted to any wavelength in a given fiber.

2.5.2 Algorithms: D-type VF, D-type NoVF, G-type VF and G-type NoVF

Depending on the performance parameter to be optimized (delay, loss probability, computational complexity) several scheduling choices are presented, leading to two different classifications of scheduling algorithms:

- Delay-oriented algorithms (D-type), aimed at transmitting the packet with the smallest possible delay.
- Gap-oriented algorithms (G-type), aimed at transmitting the packet with the smallest possible void between packets (and increasing the channel efficiency).

Or

- Without void filling (noVF), the burst/packet can only be scheduled in every wavelength after the last burst/packet scheduled previously.
- With void filling (VF), the burst/packet can be scheduled in every wavelength after the last one and between two bursts/packets scheduled previously.

In this thesis the four possible combinations will be analyzed: G-type noVF, D-type noVF, G-type VF and D-type VF [8].

Chapter 3

Evaluation of the cost of optical burst/packet switching nodes

This chapter provides an explanation about node costs and it focuses on the impact of limited optical wavelength conversion. One of the objectives of this thesis is to analyze converter costs, depending on the type and the number of these devices.

3.1 Node Cost Evaluation

Some previous studies [5][6] propose frameworks for evaluating the cost of a switching matrix. It is not easy because the technology required to build OPS/OBS nodes is still partially immature and detailed data from telecom manufacturers are not available.

These works focus on three cost components:

- **CAPEX** considers the cost of both the optical components and the production process.
- **Footprint** is a measure of the physical space occupied by the switching fabric.
- **Power consumption**, the level of power dissipated by a node, evaluated as being proportional to the number of active devices.

In this thesis, the impact of a limited optical wavelength conversion is studied. The goal is to identify the optimal number of converters in order to provide performance similar to those of a switching matrix node with an infinite number of converters. We try to compare the cost and power consumption of these devices depending on the configuration of the node.

3.2 Impact of limited optical wavelength conversion

Wavelength converter represents a fundamental component of the switching node architecture. Nevertheless, wavelength conversion requires expensive devices. The overall cost is related to the number of converters required.

Converters are fixed, each λ_{in} corresponds to a specific λ_{out}, or tunable devices, meaning that given λ_{in} the single converter may provide a certain range of λ_{out} that depends on λ_{in}. It depends on whether the LASER used as pump in the converters is tunable or not.

There is another conversion classification. If λ_{out} can be any wavelength, we call this Full-range Wavelength Conversion (FWC), that provides maximum flexibility in the access to the switching resources. If λ_{out} is somehow limited, we call this Limited-range Wavelength Conversion (LWC). Full-range converters are generally more complex and expensive.

In this work we assume that converters with a tunable LASER are FWC and if theLASER is not tunable the conversion is from a specific wavelength to another specific one.

Chapter 4

The simulator structure

The overall aim of this project is to develop a modular simulator, able to provide an evaluation of the performance of the optical packet switch. The simulator extends an existing one, by optimizing the management of the data structures and by introducing limited wavelength conversion. The event generation is very similar to the one seen in previous works, with some new features added. In particular, voids management has been improved in order to allow faster simulations, thus allowing for more detailed results thanks to the possibility to simulate a larger number of packets .

4.1 Scheduling problem formulation

The scheduler in the OPS node has to decide the output fiber, the output wavelength and the instant when an arriving packet will be transmitted. This chapter presents the simulator that performs these functions. The first thing that it requires is to know the *scheduling points* in every fiber and wavelength, all the instants when the packet could be transmitted. In an ideal scheduling matrix, the total number of scheduling points at time t_a is:

$$di\ (t_a)=t_a+d_i \tag{1}$$

where i=0,...,B-1 and t_a is the instant when actual data will arrive at the configured switching matrix. d_0 is the zero-delay, the packet is transmitted at the instant when it arrives considering the switching matrix propagation time negligible.

All these points are candidates if another packet is not supposed to be transmitted on that moment but only in case of unlimited wavelength conversion. If conversion is limited, besides these conditions, before considering a point as a candidate it must be proved that there is a free converter if necessary. For the time being we assume an unlimited wavelength conversion. In point 4.3, the changes to have an optical packet switching with a limited conversion are explained.

Finally, the *scheduling time window* is defined as

$$d_B\ (t_a) = d_B\ \text{-}1\ (t_a) + L_M \tag{2}$$

where L_M is the maximum length of the packet.

The voids or gaps on each output channel are arranged in a logical list, for a total of C lists per output interface. Each gap in the list is represented by an element including the void starting time b and ending time e. Lists are ordered chronologically. When the channel is absolutely free (no packets have been scheduled) the list includes a single void filling the whole scheduling time window.

Therefore, assuming K voids are present on a given channel $c \in [1:C]$, the corresponding list looks like $V_c = \{(b_{c1},e_{c1}),..., (b_{ck},e_{ck}),..., (b_{cK},e_{cK})\}$, where b_{ck} is the beginning gap and e_{ck} the ending gap. Gaps in the same channel will be consecutive and non-overlapping.

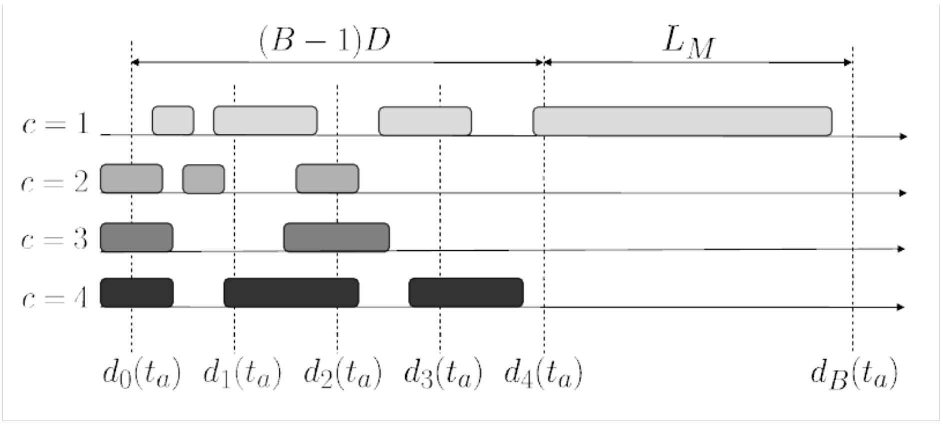


Figure 4. **Example of buffer status at time t_a**

Figure 4 is an example of buffer status at time t_a [7]. In this case B = 5, C = 4 and L_M = 3D (with D = 1). The gap lists for every channel are the following:

$$V_1 = \{(0, 0.2), (0.6, 0.8), (1.8, 2.4), (3.3, 3.9), (6.8, 7.0)\}$$

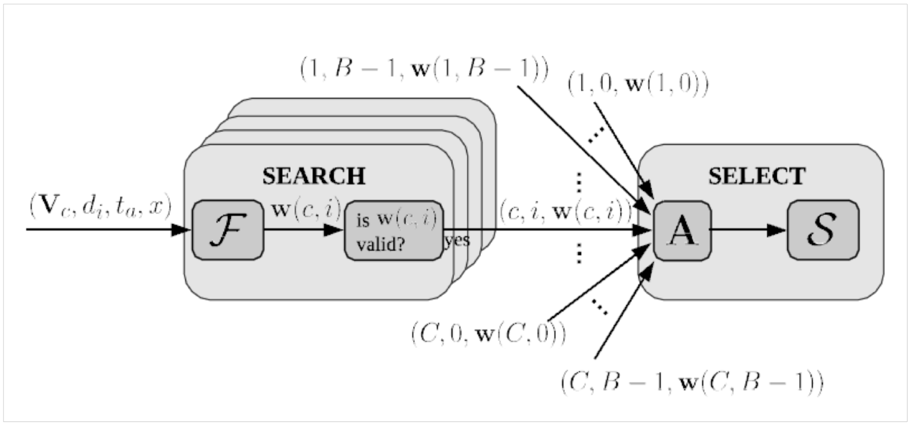
$$V_2 = \{(0.3, 0.5), (0.9, 1.6), (2.2, 7.0)\}$$

$$V_3 = \{(0.4, 1.5), (2.5, 7.0)\}$$

$$V_4 = \{(0.4, 0.9), (2.2, 2.7), (3.8, 7.0)\}$$

The simulator must choose the correct point where the packet should be scheduled. The scheduling points are founded by checking channels availability at every $d_i(t_a)$ and if the packet fits on suitable voids. Finally, the optimal point is chosen based on the policy of the CDS algorithm selected.

This exhaustive search would require high computational complexity and time consumption if it is done sequentially, especially when there is a large number of output channels. Therefore, the search problem is decomposed into a series of tasks that can be executed in parallel.



$(1, B-1, \mathbf{w}(1, B-1))$
$(1, 0, \mathbf{w}(1, 0))$
SEARCH
$(\mathbf{V}_c, d_i, t_a, x)$
$\mathcal{F}$
$\mathbf{w}(c, i)$
is $\mathbf{w}(c, i)$ valid?
yes
$(c, i, \mathbf{w}(c, i))$
SELECT
A
$\mathcal{S}$
$(C, 0, \mathbf{w}(C, 0))$
$(C, B-1, \mathbf{w}(C, B-1))$

Figure 5. Schematic of the scheduling logical sub-blocks

The figure 5 shows the logical blocks that execute this process. Firstly, the *Search function* finds the entire candidate scheduling points, which satisfy the conditions for every channel, and using the *Select function*, the simulator chooses the best one. Then, the following subsections explain in greater detail how these two blocks work.

4.1.1 The Search function

The objective is to find scheduling points, applying a function $F\ (V_c, d_i, t_a, x)$ that checks, for every possible insertion point d_i, with $i = 0 \dots B\text{-}1$, if is suitable to accommodate the incoming packet (packet length x) consulting the lists V_c. Collect all the suitable points into a temporary vector $A(t_a, x)$.

First, this function calculates a vector with two components, for every channel c and delay i:

$$F\ (V_c, d_i, t_a, x) = (H(V_c, d_i, t_a, x), T(V_c, d_i, t_a, x)) \quad (3)$$

The first component is also called *Hails* and corresponds to the gap that would be left before if the packet is scheduled in channel c at delay i. And the second one is called *Tails* and is the gap that would be left after the packet. The values are calculated with:

$$H(\mathrm{V_c}, \mathrm{d_i}, t_a, x) = d_i(t_a)\text{-}b_{ck} \quad (4)$$

$$T(V_c, d_i, t_a, x) = e_{ck} - d_i(ta) - x \quad (5)$$

if $\quad e_{ck} > d_i(t_a) \quad$ and $\quad b_{ck} < d_i(t_a) + x \quad$ (6)(7)

otherwise $\quad H(\mathrm{V}_c, d_i, t_a, x) = T(\mathrm{V}_c, d_i, t_a, x) = -\infty \quad$ (8)

In the example of figure 4, the corresponding values are in the following table 1. The length of the incoming packet is $x = 0.3$, $t_a = 0$ and the results are:

$\mathbf{w}(c,i)$	$i=0$	$i=1$	$i=2$	$i=3$	$i=4$
$c=1$	$(0,-0.1)$	$-\infty$	$(0.2,0.1)$	$-\infty$	$-\infty$
$c=2$	$-\infty$	$(0.1,0.3)$	$(-0.2,4.7)$	$(0.8,3.7)$	$(1.8,2.7)$
$c=3$	$-\infty$	$(0.6,0.2)$	$-\infty$	$(0.5,3.7)$	$(1.5,2.7)$
$c=4$	$-\infty$	$-\infty$	$(-0.2,0.4)$	$-\infty$	$(0.2,2.7)$

Table 1. Values assumed by ($H(V_c, d_i, t_a, x)$, $T(V_c, d_i, t_a, x)$) for the reference example of Figure 4

Therefore, before including a scheduling point into the list of possible values the following validity test will be used:

$$(H(c,i),T(c,i)) \in A(t_a,x) \quad \text{if} \qquad H(c,i)\geq 0 \text{ and } T(c,i)\geq 0 \tag{9}$$

Which, applied to the previous example, will provide a vector $A(t_a, x)$ with elements, those with two non-negative components.

Finally, algorithms that do not use void filling need only to maintain the horizon information for each channel. This means that lists V_c at time t_a for the example above would be:

$V_1 = \{(6.8, 7.0)\}$ $V_2 = \{(2.2, 7.0)\}$ $V_3 = \{(2.5, 7.0)\}$ $V_4 = \{(3.8, 7.0)\}$

and the rest of values shown in table XXX would be all $-\infty$.

4.1.2 The Select function

This block chooses the best scheduling point out of the vector $A(t_a, x)$ using the select function

$$S(A(t_a, x)) = (c_0, i_0) \tag{10}$$

that depends on the algorithm adopted.

G-type Algorithms.

The objective is to find the minimum gap that would be created between the incoming packet and the preceding one.

In case many scheduling points provide the same residual gap, the one with smaller delay will be selected. In case there are still more than one result, it can be chosen randomly.

D-type Algorithms.

In this type of algorithms, the function chooses the scheduling point with minimum delay, measuring the distance between each potential scheduling point and the end of the scheduling time window $d_B(t_a)$, reduced by the residual gap measured by $H(c, i)$

$$D(c,i)=dB\ (ta)-di\ (ta)-H(c,i) \tag{11}$$

Then, the optimal scheduling point is such that

$$D(c_0, i_0) = \max \quad D(c, i) \tag{12}$$

In case several channels provide the smallest delay, the one with smaller residual gap will be chosen. Finally if still many channels provide the same solution, a random choice can be made.

VF and NoVF Algorithms.

The formulation for VF and NoVF are similar. The best scheduling point must satisfy the same conditions in both cases, the difference is that in NoVF algorithms only the gaps after the last packets scheduled in every channel can be candidates and in VF algorithms all the gaps included on $A(t_a, x)$ are valid.

Moreover, in this thesis G-type VF algorithm is only minimizing the gap before the packet scheduled (Head), but not taking into account the void remaining after the packet (Tail). This can lead to lower channel efficiency.

As an illustration, figure 6 shows the best scheduling points depending on the algorithm selected.

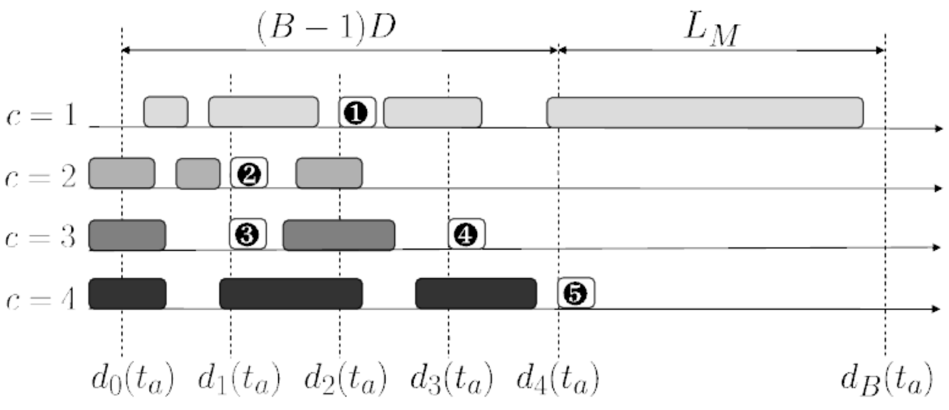


Figure 6. Possible scheduling solutions for the example on Fig. 4 assuming an incoming packet with *x* = 0,3: (1) G-type VF with minimum ending gap or with best fit; (2) G-type VF with minimum starting gap, D- type VF with minimum starting gap; (3) D-type VF with minimum ending gap; (4) D-type noVF; (5) G-type noVF.

4.2 Gap control and data updating

Another important point of the simulator is the gap management that must be constantly updated. The following structure represents a gap:

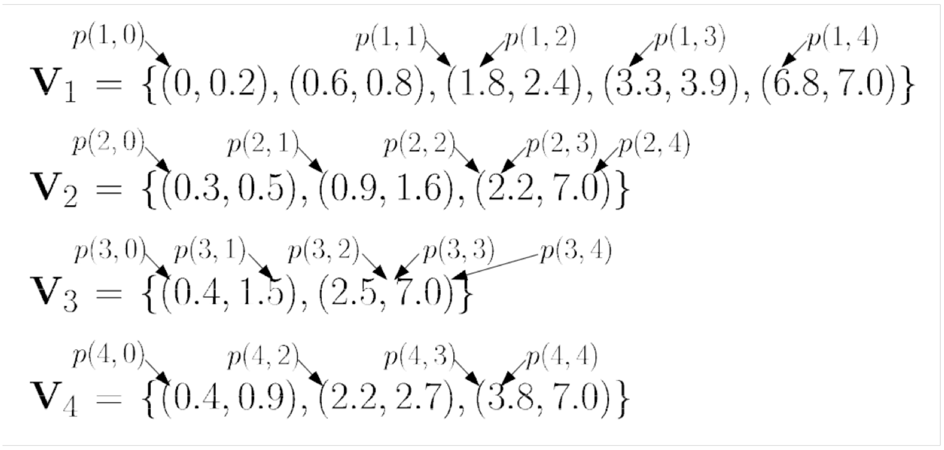
```
struct gap {
  double tstart;     /* time the gap starts */
  double tend;       /* time the gap ends */
  gap   *prev;       /* pointer to previous gap */
  gap   *next;       /* pointer to next gap */
};
```

In order to keep track of the gaps in every channel, two arrays of pointers are used.

```
struct gap *lista_gap[NMAXOUT][NMAXFIBPEROUT][NMAXWDM];

struct gap *gp[NMAXOUT][NMAXFIBPEROUT][NMAXWDM][MAXFDL];
```

The first array has pointers to the first gap in every outwavelength of every fiber of every output port sorted by time. And the second one has pointers to the gap in every channel and delay. In our example, the pointers would be the following:


p(1,0)
p(1,1)
p(1,2)
p(1,3)
p(1,4)
V1 = {(0, 0.2), (0.6, 0.8), (1.8, 2.4), (3.3, 3.9), (6.8, 7.0)}
p(2,0)
p(2,1)
p(2,2)
p(2,3)
p(2,4)
V2 = {(0.3, 0.5), (0.9, 1.6), (2.2, 7.0)}
p(3,0)
p(3,1)
p(3,2)
p(3,3)
p(3,4)
V3 = {(0.4, 1.5), (2.5, 7.0)}
p(4,0)
p(4,2)
p(4,3)
p(4,4)
V4 = {(0.4, 0.9), (2.2, 2.7), (3.8, 7.0)}

When a packet is scheduled, new gaps are created. Then, the gap list must be updated just moving to the gap where the packet have been placed and checking the place that occupies the packet in the gap:

- In the middle of the gap: Two new gaps will appear, before and after the packet.
- At the beginning of the gap. New gap after the packet.
- At the end of the gap. New gap before the packet.
- Filling the whole gap. It is enough to link the previous and the next gap.

The function of the simulator that performs all this steps is the *update_gaplist*:

```
int update_gaplist(double tempo, double duration, int out, int fib, int wl, int delay)
```

After current time has been updated, it is also important to change the pointers to gaps in every channel and delay. In this work, a new *update_gp* function has been created, in order to manage pointers in a faster and effective way.

```
void update_gp(double tempo){

 struct gap *g;
 g=(struct gap *)malloc(sizeof(struct gap));
 int t;

 for(i=0;i<numout;i++) {
        for(j=0;j<numfib;j++) {
            for(k=0;k<numwdm;k++) {
                    for(l=0;l<numfdl;l++) {

                            g=gp[i][j][k][l];
                            t=l;
                            while(g==NULL) {g=gp[i][j][k][t+1];
                                              t++;}

                            while(((g->tstart<tempo+l*T)&&(g->tend<tempo+l*T))){g=g->next;}

                            if(g->tstart<tempo+l*T)               gp[i][j][k][l]=g;
                            else if(g->tstart<tempo+(l+1)*T)      gp[i][j][k][l]=g;
                            else if(l==(numfdl-1))                gp[i][j][k][l]=g;
                            else                                  gp[i][j][k][l]=NULL;
```

```
                                if(l+1<numfdl){
                                        if((gp[i][j][k][l+1]!=NULL)&&(gp[i][j][k][l]!=NULL)){
                                                        if(gp[i][j][k][l+1]->tstart<gp[i][j][k][l]->tstart){
                                                                gp[i][j][k][l+1]=gp[i][j][k][l];
                                                        }
                                        }
                                }
                        }
                }
            }
    }
}
```

Instead of deleting the whole list and return to find all the first gaps in every channel and delay, this function uses old pointers. It assumes that the new pointed gap must be the same or subsequent to the previously pointed. In this way, having to check all the previous gaps is avoided. If the first gap satisfies the conditions, the pointer remains the same. Otherwise, the following gaps will be checked in order to find the correct one. The efficiency is even better as time goes on. In addition, if a pointer has been updated and its value has changed, the pointer of the next delay is initialized with this value because the previous gaps are not candidates. This process is repeated for every delay in every channel.

Limitation of wavelength conversion

The implementation of wavelength conversion in the switching matrix impacts on the algorithm. In order to control the number of converters available, we create a matrix (Table 2) which has the indexes of the input/output wavelength in rows and columns. This matrix is initialized with the total number of converters available to move from a given input to a given output wavelength the matrix is called.

```
num_conv[NMAXWDM+1][NMAXWDM+1];
```

There are our types of converters as explained in Table 2.

$\lambda_{in}/\lambda_{out}$	**0**	**1**	**2**	**…**	***W*-1**	***W***
0		N_{01}	N_{02}	…	$N_{0\ W-1}$	N_{0F}
1	N_{10}		N_{12}	…	$N_{1\ W-1}$	N_{1F}
2	N_{20}	N_{21}		…	$N_{2\ W-1}$	N_{2F}
…	…	…	…	…	…	…
***W*-1**	$N_{W-1\ 0}$	$N_{W-1\ 1}$	$N_{W-1\ 2}$	…		$N_{W-1\ F}$
W	N_{F0}	N_{F1}	N_{F2}	…	$N_{F\ W-1}$	N_{FF}

No conv	No conversion is needed
Nconv1	Number of converters that can convert between specific wavelengths
Nconv2	Number of converters that can convert a specific wavelength to all of them
Nconv3	Number of converters that can convert any wavelength to a specific one
Nconv4	Number of converters Full-to-Full, that can convert between all wavelengths

Table 2. Appearance of matrix *num_conv* created in order to control the number of converters available

We can assume that type 1 converters are fixed, because given λ_{in}, the single converter may provide a certain λ_{out}. In all other cases the converters are tunable, given λ_{in}, the single converter may provide a certain range of λ_{out} that depends on λ_{in}.

Another classification refers to the range of λ_{out} of a converter. There are Full-range conversion and Limited-range conversion. According to this, types 1 and 3 are Limited-range converters, and types 2 and 4 Full-range converters.

This thesis focuses on type 1, Specific-to-Specific wavelength converters and type 4, Full-to-Full wavelength converters.

When the number of converters is limited, to find a channel and delay available for a packet is not enough to be transmitted. If a wavelength conversion is necessary, a converter capable of carrying out the conversion must be free. In order to avoid this problem, the *Select function* has been modified. Instead of applying the *Search function* for every output wavelength and fiber to find matrix $A(t_a, x)$, the search is made only for those output wavelengths for which conversion is not necessary or if there is at least a converter availabl. *Search_conv* is the function that handles this check. It searches first the converters which can run fewer combinations of conversion, then it searches the converters that are more flexible.

```
int search_conv(int inwdm, int outwdm) {

  int c=0; // It remains = 0 if no conversion is needed

  if(inwdm!=outwdm) {
     if(num_conv[inwdm][outwdm]>0) { // N in->out case 1
        c=1;
     } else if(num_conv[inwdm][numwdm]>0) { // N in->F case 2

        c=2;
     } else if(num_conv[numwdm][outwdm]>0) { // N F->out case 3
        c=3;
     } else if(num_conv[numwdm][numwdm]>0) { // N F->F case 4
        c=4;
     } else { // No converters available
        c=-1;
     }
  }
  return c;
}
```

The type of converter for every possible output channel is stored in a matrix:

```
conv_type[i][j] = search_conv((int)lista_eventi->lung_onda,j);
```

Then, if a converter is available, the *Search function* is called to find the scheduling points for that channel, fiber-wavelength. Otherwise, the corresponding values of $A(t_a, x)$ become $-\infty$. The use of limited number of converters reduces the scheduling choices, the algorithm may schedule a packet to a subset of all the scheduling points that an unlimited wavelength conversion could choose.

Once the output channel is chosen, it should be a way to simulate that the converter becomes occupied and, after a while, it is set free. For this purpose there are two new functions:

```
void sel_conv(int conv_t, int inwdm, int outwdm) {

  switch(conv_t) {
  case 1:
    num_conv[inwdm][outwdm]--;
    break;
  case 2:
    num_conv[inwdm][numwdm]--;
    break;

  case 3:
    num_conv[numwdm][outwdm]--;
    break;

  case 4:
    num_conv[numwdm][numwdm]--;
    break;
  }
}
```

This function reduces the number of converters depending on the output channel where the packet has been scheduled. Then, a new event is created, in the same way that new packets arrival, simulating the time the packet will leave thus freeing the converter.

Meanwhile, the simulator is able to attend other events. When the time of freeing the converter comes, it will be necessary to call the following function that increases the corresponding number of converters.

```
void set_free_conv(int conv_t, int outwdm){

        if(conv_t==1){ // N in->out case 1
                num_conv[lista_eventi->input][outwdm]++;
        }else if(conv_t==2){ // N in->F case 2
                num_conv[lista_eventi->input][numwdm]++;
        }else if(conv_t==3){ // N F->out case 3
                num_conv[numwdm][outwdm]++;
        }else if(conv_t==4){ // N F->F case 4
                num_conv[numwdm][numwdm]++;
        }
}
```

4.3 Packet arrivals generation and event management

In case of Limited Wavelength Conversion, there are two types of events. We use the word *event* to refer to a new packet arrival and to setting free a converter. When these events are created, as explained below, theyare arranged in a chronological order in an event list:

```
    struct evento *lista_eventi;        /* pointer to the first element of the event list sorted by time*/
```

```
        struct evento {              /* structure representing an event */
                    double time;     /* time of arrival */
                    double lung;     /* packet length */
                    short input;     /* input where the packet appears */
                    short fiber;     /* fiber of the input/output */
                    short lung_onda; /* wavelength of arrival */
                    short tipo;      /* type of event */
                    evento *punt;    /* pointer to the next event */
        };
```

The function responsible for event management is *manage_events:*

```
void manage_events(){
        int testvar=0;
        //manage events while list isn't NULL
        while(lista_eventi!=NULL){

            switch(lista_eventi->tipo) {
            case ARRIVAL:
                        tempo=lista_eventi->time;
                         update_gp(tempo);          /*Update gap pointers to new time*/
                                    /* Next arrival generation and insertion in the event list */
                        if(pac_sim<pacch) {
                          generator_mm1(lista_eventi->input, lista_eventi->fiber,
                                    lista_eventi->lung_onda,&tprossimo,&lpacch);
                          insert_event(tprossimo,lpacch,lista_eventi->input,lista_eventi->fiber,
                                    lista_eventi->lung_onda,ARRIVAL);
                          pac_sim++;

                                    /* output generation and wavelength selection*/
                        out=randinteger(numout);

                        if (sel_fib_wdm(out,tempo,lista_eventi->lung,&rit_ass,&tau,&fib,&lout)==0){
                                    /*selects delay, fiber & wavelength */

                                    sel_conv(conv_type[fib][lout],(int)lista_eventi->lung_onda,lout);
                                    double conv_time;
                                    conv_time= lista_eventi->lung; //conversion time in seconds

                                    insert_event(tempo+conv_time,lista_eventi->lung,
                                                lista_eventi->lung_onda,conv_type[fib][lout],lout,FREE_CONV);

                                    update_gaplist(tempo,lista_eventi->lung, out, fib, lout, rit_ass);
                                    tfreeout[out][fib][lout]=tempo+(double)rit_ass*T+lista_eventi->lung;
                        }else {   pacch_persi++;}

                        eliminate_event();
                        break;

            case FREE_CONV:
                        set_free_conv((int)lista_eventi->fiber, (int)lista_eventi->lung_onda);
                        eliminate_event();
                        break;

         }
        }
}
```

At *arrival event*, the gap pointers are updated, a new packet arrival is generated and inserts in the event list. Then a random generator selects the output and the select function finds the best channel to schedule the packet. If the select is not successful the packet is lost.

At *free_conv* event, as explained already, the converter is set free.

The arrival generation starts when the main function produces the first arrivals for every input calling the *first_arrivals()* function. Input traffic is made of variable length data payloads, arriving as a Poisson process. The function *generate_first_pack (double *l)* returns the length of the first packet generated:

```
void generate_first_pack(double *l) {
 /* generates length of the first packet at time t=0*/

 double length;
 int lung_in_byte;

 length=-log(1-ran2(&seed))*lungmedia;
 lung_in_byte=(int)(length*GIGABITSEC/8+1);
 *l=length;
}
```

Then, *insert_first_arrival(double length,short in,short fib, short wl)* creates the first event structure and inserts it in the event list *lista_event* by means of the function *insert_first_arrival(double length,short in,short fib, short wl)*. A similar procedure is used to generate arrivals for the remaining inputs, but a different function is used to insert them into the list of events (*insert_event(double tempo, double lunghezza,short in, short fib, short l_onda,short tipo_ev)*). This functions checks the time of arrival and inserts the event into the list, chronologically ordered.

After the first packets have been generated, the function *manage_events()* is called from the main. This function, which will be analyzed later in this thesis, checks if there are still packets to be scheduled and, in case the number of generated packets is lower than the number of total packets to simulate, uses the function *generator_mm1(int ingress, int fiber, int l_onda, double *prox_arrivo, double *lun)* to generate an interarrival time and an exponentially distributed length for the next packet. Then, these data is used to create an event, and the function *insert_event(double tempo, double lunghezza,short in, short fib, short l_onda,short tipo_ev)* allocates it into the correct position from the event list.

Chapter 5
Simulation results

In this chapter the simulation results are presented. They are referred to a generic optical packet switching node with either Limited or Unlimited Wavelength Conversion. The following input parameters allow us to design the node architecture:

- The number of input/output ports *I/O*
- The number of fiber per output link *F*
- The number of wavelengths available in each fiber *W*
- The number of delays *B* and the delay unit D
- The traffic payload per wavelength
- The average payload size
- The chosen scheduling algorithm: either D or G-type, with or without void filling

5.1 Unlimited Wavelength Conversion

The first simulations refer to the case of a node with unlimited wavelength conversion capability, giving a large enough value to the number of converters. So a packet will never be lost due to lack of converters.

In order to compare different cases, the cardinality is kept constant at $|S| = B \cdot W \cdot F = 64$ has been used in all simulations. As we have considered the case of a single fiber per link, the scheduling space can be expressed as $|S| = B \cdot W$.

The traffic load per wavelength is 0.8 and the average payload size is chosen, depending on the single channel bit-rate, in order to have an average transmission time in the order of 1 μs. The results show the packet loss probability as a function of the buffer delay unit *D* normalized to the average payload length (1000 in this study). It is clear that *D* is a key parameter in the buffer design, and it must be accurately chosen.

5.1.1 No Void Filling algorithms

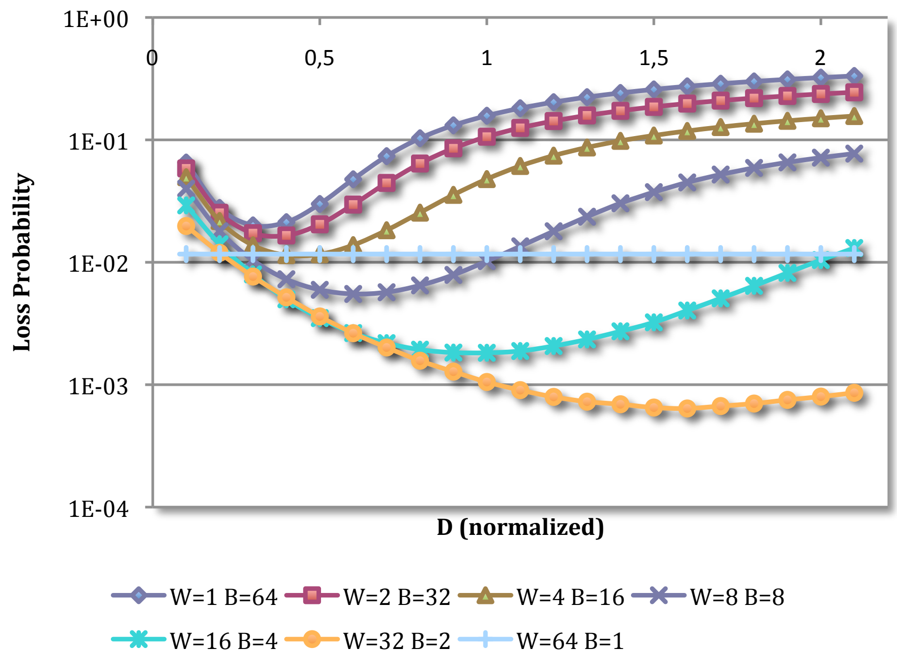


Figure 7. Packet loss probability as a function of *D* for a switch with D-type NoVF algorithm, |*S*|=64, *F*=1, 5.000.000 of packets and *R*=500.

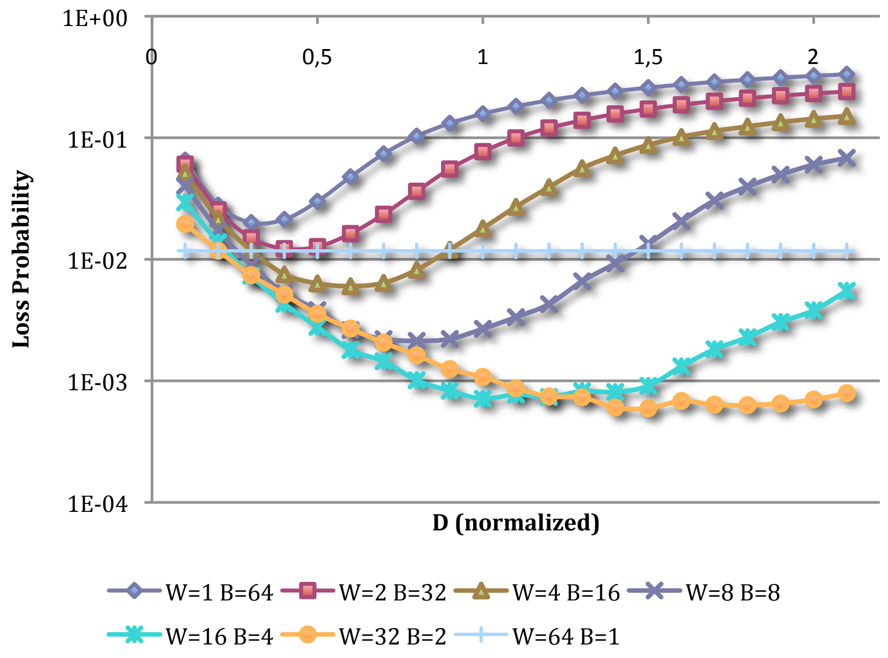


Figure 8. Packet loss probability as a function of *D* for a switch with G-type NoVF algorithm, |*S*|=64, *F*=1, 5.000.000 of packets and *R*=500.

The figures demonstrate that large values of W are preferable to a high value of B.

Nonetheless these results show that a little buffering in the time domain is still important, since the case with no delay (B = 1) is not the best. In this case, the loss probability is independent of the change of delay unit D. This constant value corresponds to the theoretical result of calculating the Erlang's B formula for 64 servers and 51.2 Erlangs of offered traffic (0.8 Erlang per wavelength).

The optimal performance of both D and G algorithms is W = 32, B = 2.

5.1.2 Void Filling algorithms

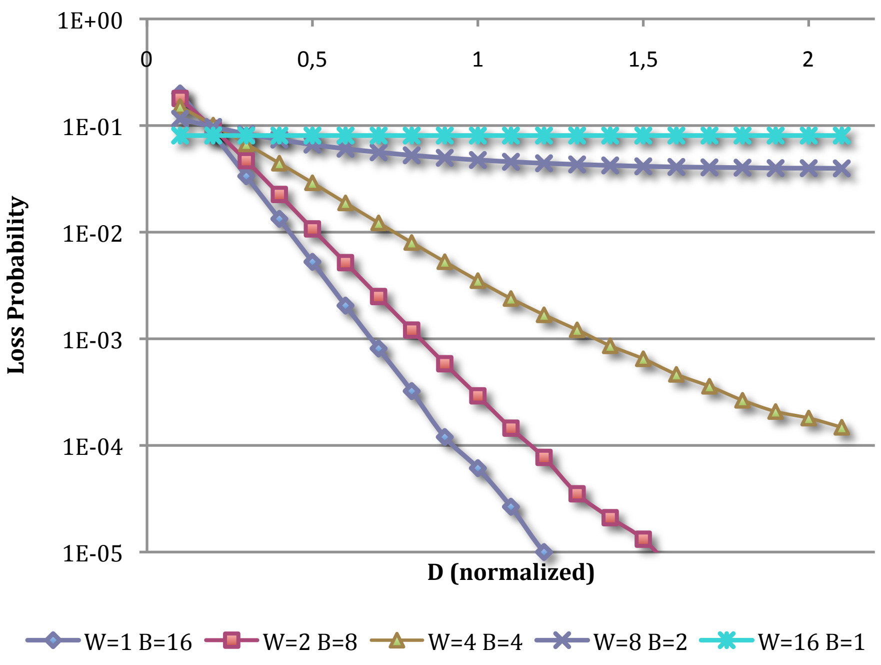


Figure 9. Packet loss probability as a function of *D* for a switch with D-type VF algorithm, |*S*|=64, *F*=1, 5.000.000 of packets and *R*=500.

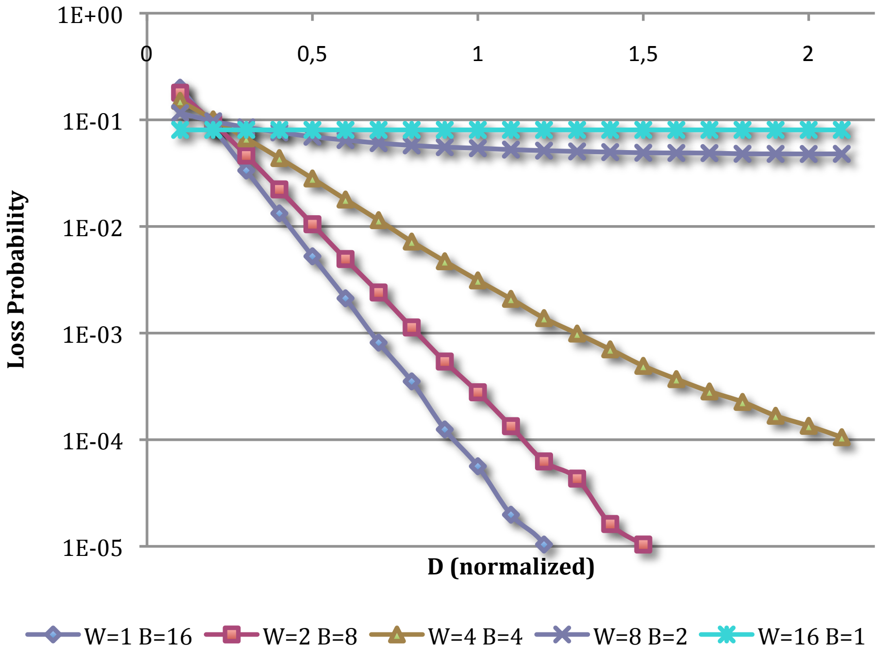


Figure 10. Packet loss probability as a function of *D* for a switch with G-type VF algorithm, |*S*|=64, *F*=1, 5.000.000 of packets and *R*=500.

Results with VF are very similar in both D and G algorithms. The main difference that can be found when comparing them to noVF simulations is that, in this case, loss probability keeps decreasing when delay unit becomes bigger. There is not an optimal value *D*, the bigger it is the smaller is the loss probability.

5.2 Limited Wavelength Conversion

First, we have considered three cases, with |*S'*| = 16 and D-type NoVF algorithm, where only one type of converter is used in order to find the optimal number of converters of each type. The following graphics show the results:

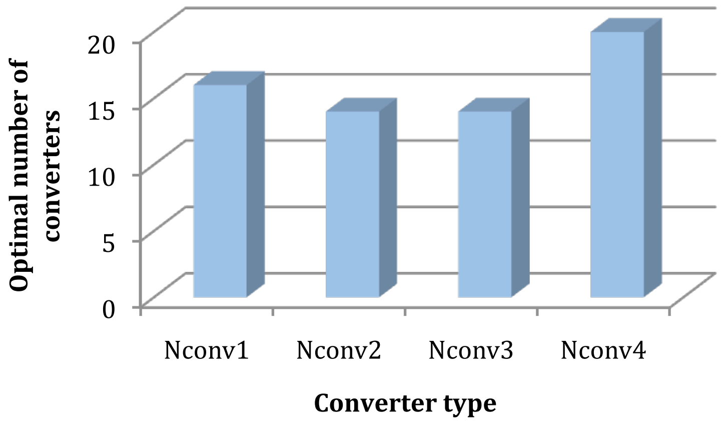


Figure 11. Optimal number of each type of converters to achieve minimum packet loss probability in case of D-type NoVF algorithm, *F*=4, *W*=2, *B*=8, 5.000.000 packets

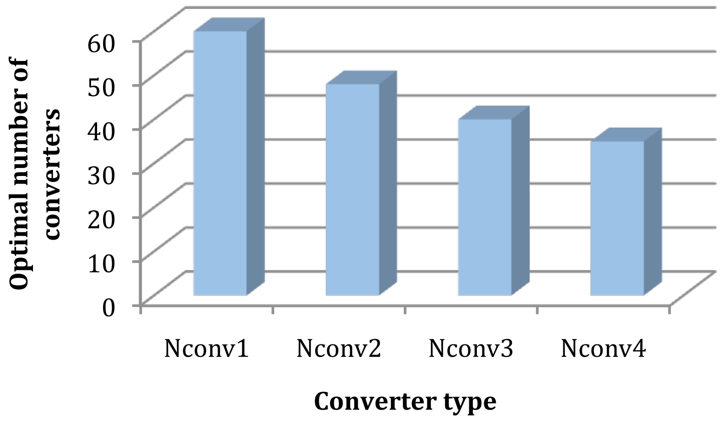


Figure 12. Optimal number of each type of converters to achieve minimum packet loss probability in case of D-type NoVF algorithm, *F*=4, *W*=4, *B*=4, 5.000.000 packets

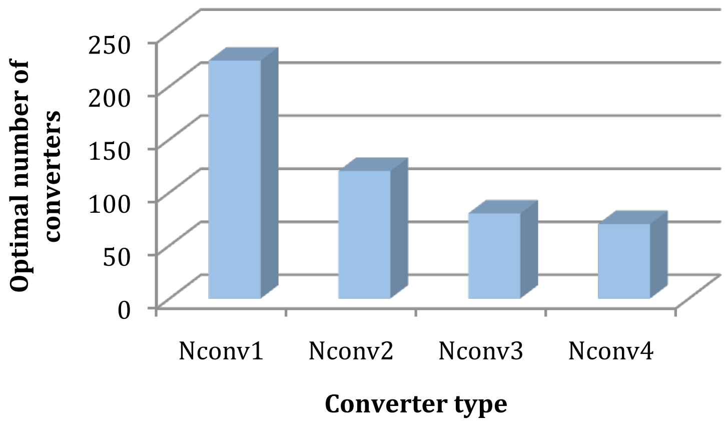


Figure 13. Optimal number of each type of converters to achieve minimum packet loss probability in case of D-type NoVF algorithm, *F*=4, *W*=8, *B*=2, 5.000.000 packets

A higher number of converters Full-to-Full are required in case of large values of *W* and for large values of *B*, since it is necessary to convert many packets from wavelength to another. Graphs prove that converters of type 1 and 4, Specific-to-Specific and Full-to-Full respectively, lead to the best results. They represent the most extreme cases and from now on, so as to simplify, we work with these two types.

For the cases W=2 B=8 and W=8 B=2, we studied the cost and consumption of these devices.

5.2.1 Power Consumption

The objective of these simulations is to find the probability of a converter being used, calculated as the average amount of time any converter is converting a packet. The goal is to understand if it is possible to have a significant saving in used power if converters are turned off when they are not used.

For each simulation has been used only one type of converters, both Full-to-Full converters and those that only can convert from an specific wavelength to another specific one, always using optimal amounts. There is a difference between power consumption of these two types of converters. Wavelength converter consumption depends mainly on laser power required and the control electronics. Consumption due to a single converter is [3]:

$$P_{\text{converter}} = P_{\text{LASER}} + P_{\text{management electr.}} \tag{13}$$

It is clear that a tunable laser consumes more than a fixed wavelength laser. So a Full-to-Full converter that needs a tunable laser, dissipate more power than those that only can convert from an specific wavelength to another specific one.

$$P_{\text{LASER tunable}} > P_{\text{LASER fixed}} \tag{14}$$

Therefore,

$$P_{\text{Full-to-Full converter}} > P_{\text{Specific-to-Specific converter}} \tag{15}$$

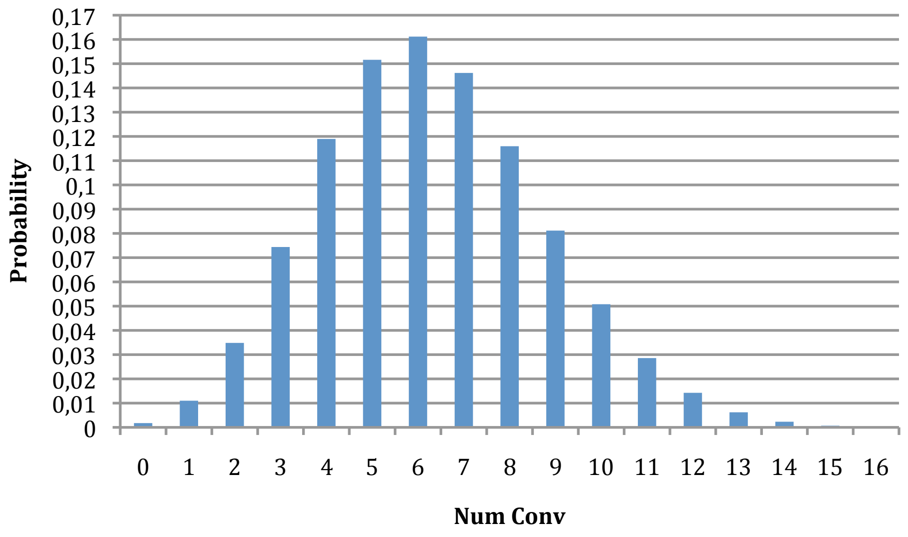


Figure 14. **Number of converters occupied probability in case of D-type NoVF algorithm, *F*=4, *W*=2, *B*=8, 5.000.000 packets and 16 Specific-to-Specific converters.**

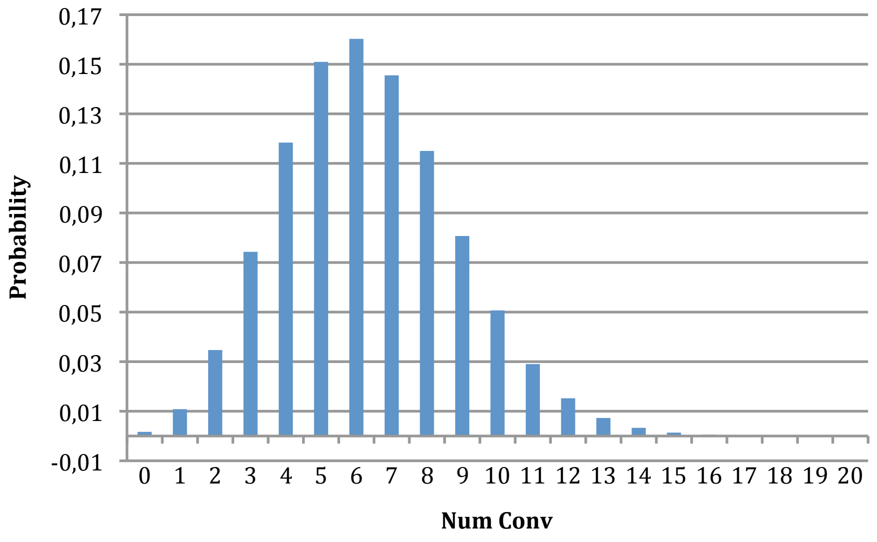


Figure 15. **Number of converters occupied probability in case of D-type NoVF algorithm, *F*=4, *W*=2, *B*=8, 5.000.000 packets and 20 Full-to-Full converters.**

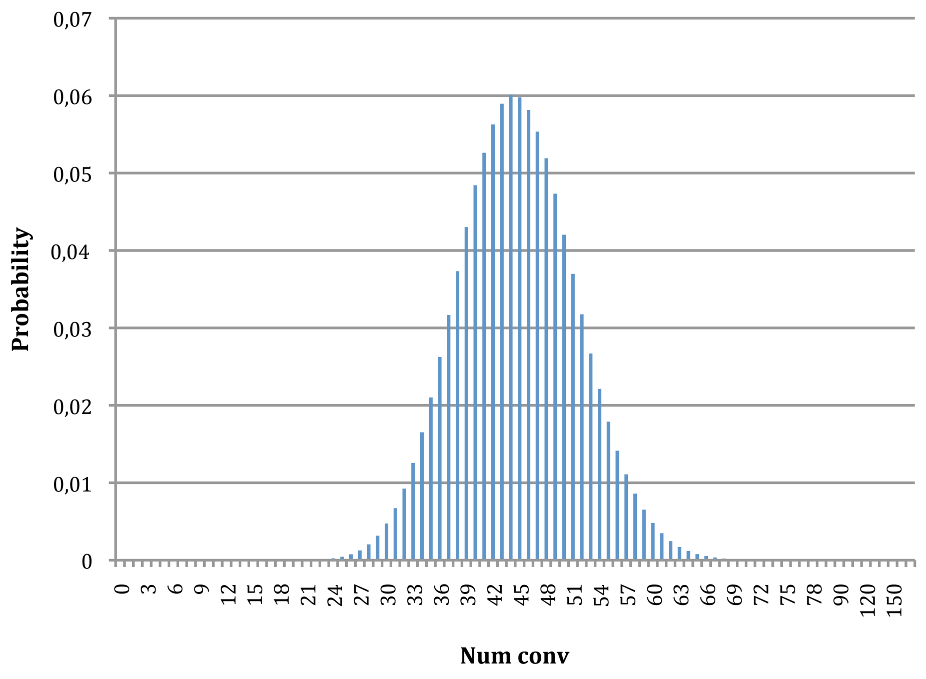


Figure 16. Number of converters occupied probability in case of D-type NoVF algorithm, *F*=4, *W*=8, *B*=2, 5.000.000 packets and 224 Specific-to-Specific converters (150 onwards is always 0) .

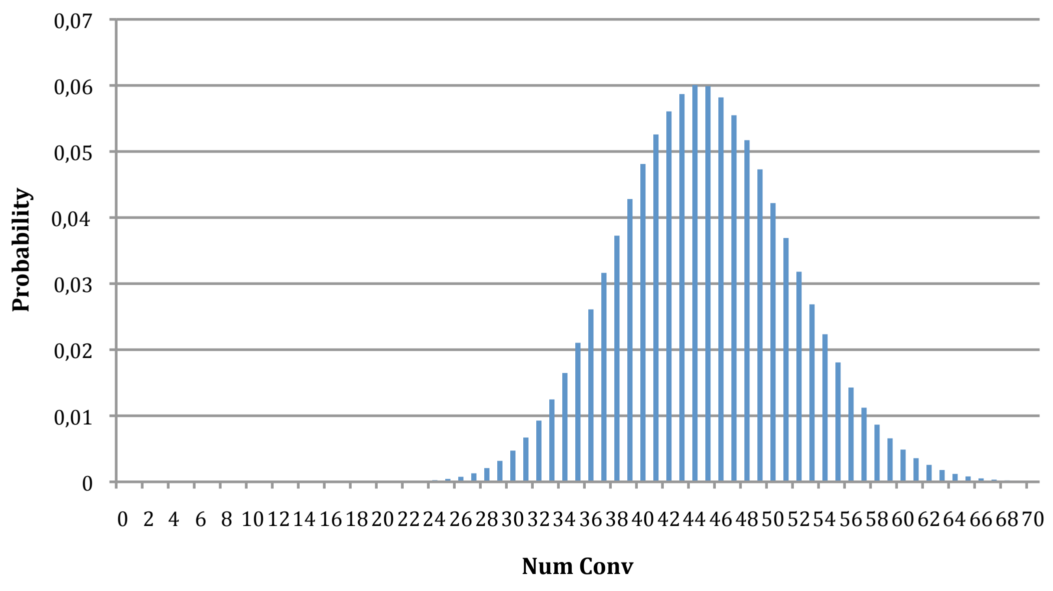


Figure 17. Number of converters occupied probability in case of D-type NoVF algorithm, *F*=4, *W*=8, *B*=2, 5.000.000 packets and 60 Full-to-Full converters.

Something important is that all graphics have a Gaussian behaviour. In addition, this behaviour depends on the traffic conditions and not on the type or number of converters as long as they are enough to provide the "amount of conversion" needed. In case *W*=2 *B*=8, it is more probable that 6 devices are running and if *W*=8 *B*=2 the number is 44.

At this point, it is easy to approximate the average power consumption as the consumption of the converters weighted but the probability that a certain number of them is used, to calculate the average power consumption.

$$P_{avg} = \sum_{i=0 \ldots Nconv} \left(i \cdot prob_i \cdot P_{device} \right) \qquad (16)$$

The results are:

	Average Power Consumption	
	By turning off converters	**Converters always working**
***W*=2 *B*=8 Full-to-Full converters**	$6{,}340863368 \cdot P_{Full\text{-}to\text{-}Full}$	$20 \cdot P_{Full\text{-}to\text{-}Full}$
***W*=2 *B*=8 Specific-to-Specific converters**	$6{,}305124708 \cdot P_{Specific\text{-}to\text{-}Specific}$	$16 \cdot P_{Specific\text{-}to\text{-}Specific}$
***W*=8 *B*=2 Full-to-Full converters**	$44{,}76024082 \cdot P_{Full\text{-}to\text{-}Full}$	$70 \cdot P_{Full\text{-}to\text{-}Full}$
***W*=8 *B*=2 Specific-to-Specific converters**	$44{,}7409878 \cdot P_{Specific\text{-}to\text{-}Specific}$	$224 \cdot P_{Specific\text{-}to\text{-}Specific}$

Table 3. Comparison of Average Power consumption depending on weather or not are turned off

It is obviously better to turn off converters when are not being used, obtaining energy savings from 20% to 65%.

Furthermore, for a given node configuration the only difference between using each type of converter lies in single device power consumption. Although the total converters number is clearly different depending on type, the average number of converters used is almost exactly the same, since it depends on the traffic profile and not on the type of converters.

Given that $P_{Full\text{-}to\text{-}Full\ converter}$ is higher then $P_{Specific\text{-}to\text{-}Specific\ converter}$, a conclusion that we could previously imagine is that a node with Full-to-Full converters consumes more than those made with Specific-to-Specific.

5.2.2 Cost

The cost of a converter of a generic optical packet switching node is given by [3]:

$$C_{\text{converter}} = C_{\text{LASER}} + C_{\text{interferometer}} + 3\cdot C_{\text{interconnexion}} \qquad (17)$$

This section tries to find out which option has a lower cost, using only one type of converter or mixing both. As in power consumption, we can make a cost comparison because a Full-to-Full converter is more expensive than a Specific-to-Specific one, because requires a tunable LASER. Likewise we can say that:

$$C_{\text{LASER tunable}} > C_{\text{LASER fixed}} \qquad (18)$$

Consequently,

$$C_{\text{Full-to-Full converter}} > C_{\text{Specific-to-Specific converter}} \qquad (19)$$

To combine both types of converters, we start with the optimal amount of one of them and none of the others. The first value decreases until zero as the second one increases. And then, we do the same but with the other optimal value.

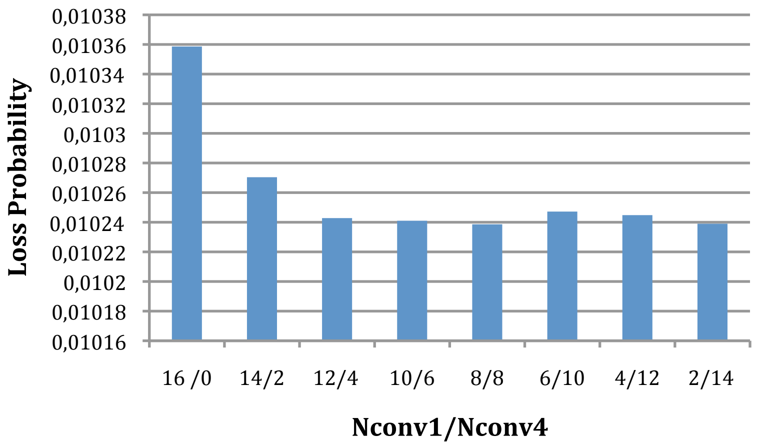


Figure 18. Packet loss probability mixing Full-to-Full and Specific-to-Specific converters, starting with S-to-S optimal number, and D-type NoVF algorithm, *F*=4, *W*=2, *B*=8.

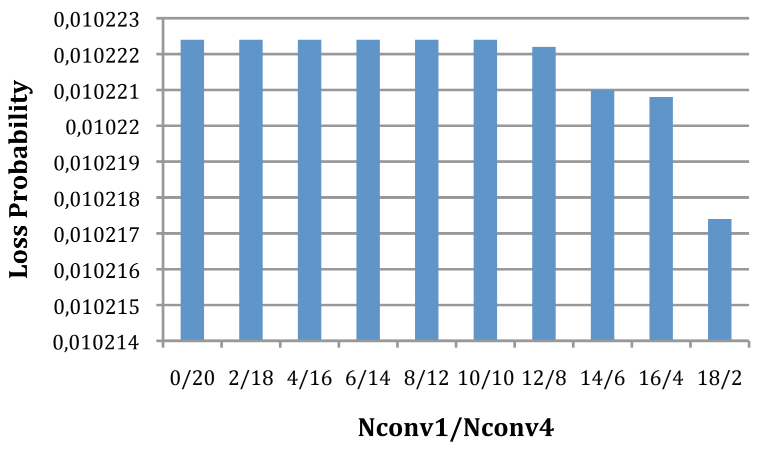


Figure 19. **Packet loss probability mixing Full-to-Full and Specific-to-Specific converters, starting with F-to-F optimal number, and D-type NoVF algorithm, *F*=4, *W*=2, *B*=8.**

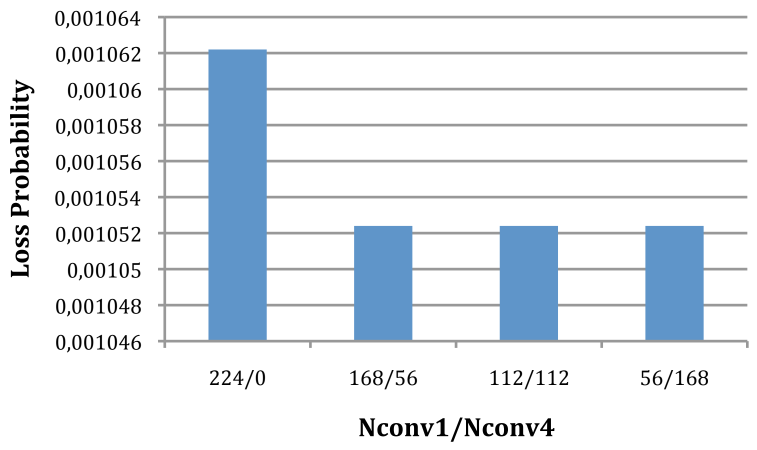


Figure 20. **Packet loss probability mixing Full-to-Full and Specific-to-Specific converters, starting with S-to-S optimal number, and D-type NoVF algorithm, *F*=4, *W*=8, *B*=2.**

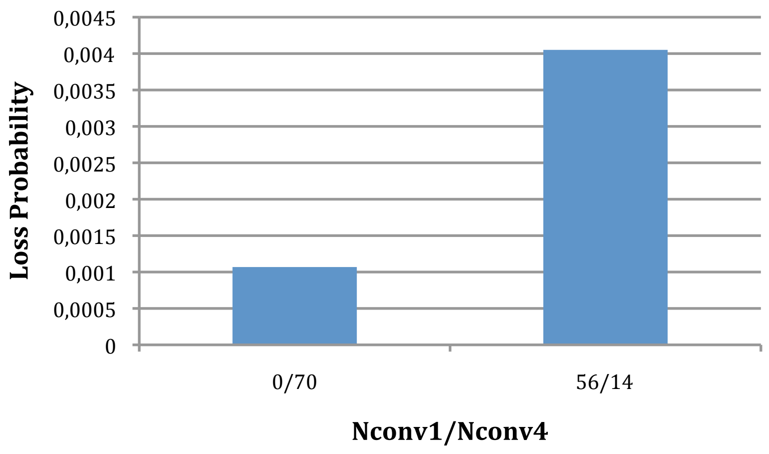


Figure 21. Packet loss probability mixing Full-to-Full and Specific-to-Specific converters, starting with F-to-F optimal number, and D-type NoVF algorithm, *F*=4, *W*=8, *B*=2.

We try to find the best combination for each configuration node, searching a minimum loss probability and the lower number of converters used to get it, especially of Full-to-Full converters because they have a higher cost. Table 4 shows the results:

		Loss Probability	Total Cost
W=2 *B*=8	$N_{\text{Specific-to-Specific}} = 14$ $N_{\text{Full-to-Full}} = 2$	0,0102704	$14 \cdot C_{\text{S-to-S}} + 2 \cdot C_{\text{F-to-F}}$
	$N_{\text{Specific-to-Specific}} = 16$ $N_{\text{Full-to-Full}} = 0$	0,0103586	$16 \cdot C_{\text{S-to-S}}$
	$N_{\text{Specific-to-Specific}} = 0$ $N_{\text{Full-to-Full}} = 20$	0,0102224	$20 \cdot C_{\text{F-to-F}}$
W=8 *B*=2	$N_{\text{Specific-to-Specific}} = 168$ $N_{\text{Full-to-Full}} = 56$	0,0010524	$168 \cdot C_{\text{S-to-S}} + 56 \cdot C_{\text{F-to-F}}$
	$N_{\text{Specific-to-Specific}} = 224$ $N_{\text{Full-to-Full}} = 0$	0,0010622	$224 \cdot C_{\text{S-to-S}}$
	$N_{\text{Specific-to-Specific}} = 0$ $N_{\text{Full-to-Full}} = 70$	0,0010686	$70 \cdot C_{\text{F-to-F}}$

Table 4. Cost of converters with optimal number of F-to-F, S-to-S converters and mixing both

It can be seen that the cost increases with the number of wavelengths used, that is intuitive since the more the wavelengths the more the conversions needed. Given that Full-to-Full converters are more expensive, the best option is to mix both types of converters. It is clear that it is cheaper to use only Specific-to-Specific converters, but this solution offers a higher loss probability.

Chapter 6

Conclusions

In this project we have studied contention resolution problem of an Optical Packet Switch. After analyzing results with Gap oriented and Delay oriented policies, we concluded that in Void Filling algorithms is better to have more delays than a big number of wavelengths, unlike in NoVF cases. Time domain becomes important when packets can be scheduled not only after the last packet. It is also worth noting that G-type offers better results when NoVF is used, while D-type becomes better with VF. This can be explained by the fact that, when packets can only be scheduled at the last gap (NoVF), channel efficiency is more important in order to have low loss probabilities. Although, when we use VF algorithms it is more significant to fill gaps with the smaller delay possible.

Moreover, the thesis focuses on an approximate evaluation of cost and power consumption due to wavelength converters. These devices are an important part of an optical switch, exploiting wavelength conversion for contention resolution. Results demonstrate that the probability function of having a certain number of converters occupied have a Gaussian behaviour. These graphics also show that it is possible to save power consumption by turning off converters when they are not being used. Finally, the best choice is mixing converters that can convert between two specific wavelengths and converters capable of converting to any wavelength in order to reduce cost.

Bibliography

[1] Internet world stats. *www.internetworldstats.com/emarketing*

[2] W, Cerroni. “DWDM Photonic Packet-Switched Networks” Alma Mater Studiorum, University of Bologna - Italy, 2002.

[3] F, Mignani. “Valutazione di Costo e Complessitá di Commutatori Ottici a Paccheto” Alma Mater Studiorum, University of Bologna - Italy, 2007-2008.

[4] F. Callegati, W. Cerroni, G. S. Pavani, “Key Parameters for Contention Resolution in Multi-Fiber Optical Burst/Packet Switching Nodes (Invited Paper)”, Proc. of 4th International Conference on Broadband Communications, Networks and Systems (IEEE Broadnets 2007), Raleigh, NC, 2007.

[5] F. Callegati, W. Cerroni. “A Framework for Evaluating the Cost of Optical Packet Switching Nodes”. DEIS, University of Bologna, Italy.

[6] A. Campi, F. Callegati, W. Cerroni. “Complexity/Performance Trade-off in Optical Packet Switches”. DEIS, University of Bologna, Italy, 2009.

[7] F. Callegati, A. Campi, W. Cerroni. “Fast and Versatile Scheduler Design for Optical Burt/Packet Switching”

[8] Font A. “Simulation of an Optical Packet Switch Scheduler”. DEIS, University of Bologna, Italy, 2010

Appendix A
Codes of the Simulator

```
/* SIMULATION OF AN OPTICAL PACKET SWITCHING NODE SCHEDULER*/

#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <time.h>
#include <unistd.h>
#include <sys/time.h>

#define IM1 2147483563                         /* definition of constants used in ran2 */
#define IM2 2147483399
#define AM (1.0/IM1)
#define IMM1 (IM1-1)
#define IA1 40014
#define IA2 40692
#define IA2 40692
#define IQ1 53668
#define IQ2 52774
#define IR1 12211
#define IR2 3791
#define NTAB 32
#define NDIV (1+IMM1/NTAB)
#define EPS 1.2e-7
#define RNMX (1.0-EPS)

#define ARRIVAL 1                             /* type of event used */
#define FREE_CONV 2                           /* type of event used */


#define INFINIT 1000000                       /* infinite integer */
#define INFINITO 1e38                         /* infinite real */

#define LMAXMEDIAIP 5000                      /* maximum average length of arriving packets */
#define LMIN 32                               /* minimum average length of arriving packets, in
bytes */
#define GIGABITSEC 2.5e9                      /* transmission speed */

#define CONVERSIONSPEED 2.5e9                /* conversor speed of optical wavelength
                                             converters*/

#define FALSE 0
#define TRUE 1

#define NSTADI 1
#define MAXFDL 128                            /* maximum number of FDL */
#define MAXMAXRIT (MAXFDL-1)                  /* maximum delay (units) fornito dal buffer
                                             monostadio */

#define NMAXOUT 16                           /* maximum number of inputs and outputs */
#define NMAXFIBPEROUT 16                     /* maximum number of fibers per input/output */
#define NMAXWDM 64                           /* maximum number of wavelengths per fiber */
```

```
#define TOTFIBERS (NMAXFIBPEROUT*NMAXOUT) /* maximum number of input/output fibers */

#define G_VF 1                                  /* algorithm used */
#define G_NoVF 2
#define D_VF 3
#define D_NoVF 4

#define min_lungmedia (LMIN*8/GIGABITSEC)    /* min and max length of packets in seconds */
#define max_lungmedia (4*LMAXMEDIAIP*8/GIGABITSEC)

#define step_byte ((4*LMAXMEDIAIP)-LMIN)/300
#define STEP (step_byte*8)/GIGABITSEC
#define NSTEP  300


struct evento {             /* structure representing an event */
  double time;              /* time of arrival */
  double lung;              /* packet length */
  short input;              /* input where the packet appears */
  short fiber;              /* fiber of the input/output */
  short lung_onda;          /* wavelength of arrival */
  short tipo;               /* type of event */
  evento *punt;             /* pointer to the next event */
};

struct gap {                /* structure representing a gap */
  double tstart;            /* time the gap starts */
  double tend;              /* time the gap ends */
  gap    *prev;             /* pointer to previous gap */
  gap    *next;             /* pointer to next gap */
};

/* GLOBAL VARIABLES*/

/* --------------------------(INPUT VARIABLES)------------------------------------- */

int    pacch;               /* number of packets to be simulated */
double Tbyte;               /* granularity in bytes */
double carico;              /* input load */
double lungbyte;            /* average length in byte */
int    numout;              /* number of inputs/outputs */
int    numfib;              /* number of fibers per input/output */
int    numwdm;              /* number of wavelengths per fiber */
int    pol;                 /* indicates the chosen politic to choose output wavelength */
int    numfdl;              /* number of FDL */
int nconv1;                 /* number of conversors type 1-to-1 */
int nconv2;                 /* number of conversors type 1-to-N */
int nconv3;                 /* number of conversors type N-to-1 */
int nconv4;                 /* number of conversors type N-to-N */

/* ---------------------------(GENERAL USE VARIABLES)------------------------------ */

double lungmedia;           /* average length in seconds */
double T;                   /* granularity in seconds*/
double lambda;              /* average frequence of poisson arrivals*/
long   seed;                /* seed for the random number generator*/
int    MAXRIT;              /* maximum delay = numfdl-1 */
int    test;
int    lastwl=0;
```

```
/*------------------------- (STATE VARIABLES)---------------------------------- */

struct evento *lista_eventi;         /* pointer to the first element of the event list sorted by time*/
double tfreeout[NMAXOUT][NMAXFIBPEROUT][NMAXWDM];    /* array of times when output
channels are free*/
double uscita_mm1[NMAXOUT][NMAXFIBPEROUT][NMAXWDM];
double ting_mm1[NMAXOUT][NMAXFIBPEROUT][NMAXWDM];

double Heads[NMAXFIBPEROUT][NMAXWDM][MAXFDL];  /* Remaining gaps for every
channel(fiber&wave) and every delay */
double Tails[NMAXFIBPEROUT][NMAXWDM][MAXFDL];
double AUX[NMAXFIBPEROUT][NMAXWDM][MAXFDL];

int num_conv[NMAXWDM+1][NMAXWDM+1];              /*number of conversors of every type*/
int conv_type[NMAXFIBPEROUT][NMAXWDM];           /*type of conversor choosen for every fiber
and wavelength*/

struct gap *lista_gap[NMAXOUT][NMAXFIBPEROUT][NMAXWDM];              /* pointer to the first
gap in every out, fiber, wavelength, sorted by time */

struct gap *gp[NMAXOUT][NMAXFIBPEROUT][NMAXWDM][MAXFDL];      /*pointer to the gap in
every chanel and delay */

/*---------------------------- (CONTROL VARIABLES)---------------------------- */

double lpacch;
double tprossimo, tempo;
int pac_sim;                          /* number of simulated packets */
int rit_ass;                          /* assigned delay in T units*/
int i,j,k,l;

double tr_tot_off;
double tr_tot_smalt=0;

int out;                      /* chosen output */
int fib;                      /* chosen fiber */
int lout;                     /* chosen wavelength*/
double tau;                   /* variable containing, for each packet, added length*/
double tau_tot;

int pacch_senzaconv;
int pacch_persi;              /* number of lost packets */

FILE *f1;                     /*output files*/

struct timeval monitortime;

double tempo_conv1[100*(NMAXWDM-1)*(NMAXWDM)]; /*total time for every number of occupied
type 1 conversors*/
double tempo_conv4[500]; /* total time for every number of occupied type 4 conversors*/

double oldtime1;
double oldtime4;

int conv1_occupati;          /*total number of occupied type 1 converters*/
int conv4_occupati;          /*total number of occupied type 4 converters*/
```

```
/*--------------------FUNCTION CODES----------------------------------------*/

double ran2(long *idum) {            /* generates random double between 0 and 1 */

        int j;
        long k;
        static long idum2=123456789;
        static long iy=0;
        static long iv[NTAB];
        double temp;

        if (*idum <= 0) {
           if (-(*idum) < 1) *idum=1;
           else *idum = -(*idum);
           idum2=(*idum);
           for (j=NTAB+7;j>=0;j--) {
                       k=(*idum)/IQ1;
                       *idum=IA1*(*idum-k*IQ1)-k*IR1;
                       if (*idum < 0) *idum += IM1;
                       if (j < NTAB) iv[j] = *idum;
           }
           iy=iv[0];
        }
         k=(*idum)/IQ1;
         *idum=IA1*(*idum-k*IQ1)-k*IR1;
         if (*idum < 0) *idum += IM1;
         k=idum2/IQ2;
         idum2=IA2*(idum2-k*IQ2)-k*IR2;
         if (idum2 < 0) idum2 += IM2;
         j=iy/NDIV;
         iy=iv[j]-idum2;
         iv[j] = *idum;
         if (iy < 1) iy += IMM1;
         if ((temp=AM*iy) > RNMX) return RNMX;
         else return temp;
}

/*--------------------------------------------------------------------------*/
int randinteger (int temp) {         /* provides a random number between 0 and temp-1
                                     with a uniform probability distribution.*/

          double casuale;
          int out;
          float div;

          casuale=ran2(&seed);
          div=(float)1/temp;
          out=(int)(casuale/div);
          return out;
}

/*--------------------------------------------------------------------------*/

double calc_tau(int del,double t,double tout) {         /* calculates length of the next packet */

        double tau=0.0;
        if(del>0) tau=(double)del*T-(tout-t);
        return tau;
}

/*--------------------------------------------------------------------------*/

void open_files(){

        f1=fopen("results.txt","a");
}
/*--------------------------------------------------------------------------*/

void init_vars(){
        test=0;
        pacch_senzaconv=0;
        lpacch=0;
        tprossimo=0.0; tempo=0.0;
        tau_tot=0;
        lun_totale=0;
        pacc_ritardati=0;
        pacch_persi=0;

  /*initializes variables that memorize results */

        for(j=0;j<=MAXMAXRIT;j++) distr_rit[j]=0;
         for(i=0;i<numout;i++){
           for (j=0;j<numfib;j++){
             for(k=0;k<numwdm;k++) {
                     scelta_wdm[i][j][k]=0;
                     lun_tot_smaltita[i][j][k]=0;
                     lun_tot_eff_smalt[i][j][k]=0;
                     perdita_per_lam[i][j][k]=0;
           }}}
         for(j=0;j<=NSTEP;j++) lung_pacch_persi[j]=0;
         for(j=0;j<=NSTEP;j++) lung_pacch_generati[j]=0;

  /* initialize arrays and lists */

         for(i=0;i<numout;i++) {
              for(k=0;k<numfib;k++) {
                     for(j=0;j<numwdm;j++) {
                             ting_mm1[i][k][j]=0;     /* inizialize to zero the time od arrival af first packets
                                   to enter M/M/1 generator*/
                             tfreeout[i][k][j]=0;     /* void commutator */
                             lista_gap[i][k][j]=(struct gap *)malloc(sizeof(struct gap));
                             ista_gap[i][k][j]->tstart=0;   /* initializes gap lists*/
                             lista_gap[i][k][j]->tend=INFINITO;
                             lista_gap[i][k][j]->prev=NULL;
                             lista_gap[i][k][j]->next=NULL;
                             for (l=0;l<numfdl;l++){
                                     gp[i][k][j][l]=lista_gap[i][k][j];      /*assign gap pointers
                                                                             to the corresponding channel*/
                             }
                     }
              }
         }
         lista_eventi=NULL;       /* initializes event list as void*/

        for(i=0;i<numwdm;i++){
            num_conv[i][numwdm]=nconv2;
            num_conv[numwdm][i]=nconv3;

            for(j=0;j<numwdm;j++){
                        num_conv[i][j]=nconv1;
            }
        }

        num_conv[numwdm][numwdm]=nconv4;

        for(i=0;i<=(nconv1*(numwdm-1)*(numwdm));i++) tempo_conv1[i]=0;

        for(i=0;i<=nconv4;i++) tempo_conv4[i]=0;

         oldtime1=0;
         oldtime4=0;
         conv1_occupati=0;
         conv4_occupati=0;

}
/*---------------------------------------------------------------------------*/

void generate_first_pack(double *l) {          /* generates length of the first packet at time t=0*/

          double length;
          int lung_in_byte;

          length=-log(1-ran2(&seed))*lungmedia;
          lung_in_byte=(int)(length*GIGABITSEC/8+1);
          *l=length;
}

/*---------------------------------------------------------------------------*/

void insert_first_arrival(double length,short in,short fib, short wl) {

          lista_eventi=(struct evento *)malloc(sizeof(struct evento));
          lista_eventi->time=0.0;              /* insert first arrival */
          lista_eventi->lung=length;           /* in the event list, void before */
          lista_eventi->input=in;
          lista_eventi->fiber=fib;
          lista_eventi->lung_onda=wl;          /* next events will be insert in order */
          lista_eventi->tipo=ARRIVAL;
          lista_eventi->punt=NULL;

}
/*---------------------------------------------------------------------------*/
void insert_event(double tempo, double lunghezza,short in, short fib, short l_onda,short tipo_ev) {

        struct evento *w1,*w2,*w3;

        w3=(struct evento *)malloc(sizeof(struct evento));
        w3->time=tempo;
        w3->lung=lunghezza;                    /* reserve space and create new event*/
        w3->input=in;
        w3->fiber=fib;
        w3->lung_onda=l_onda;
        w3->tipo=tipo_ev;
        if(lista_eventi->time>w3->time) {      /* control if the new event */
            w3->punt=lista_eventi;             /* will be at the beginning of the list*/
            lista_eventi=w3;
        }
        else {
            w1=lista_eventi;                   /* inizialize two pointers */
            w2=lista_eventi->punt;             /* in order to move the list */
            while((w2!=NULL)&&(w2->time<=w3->time)) {
                        w1=w2;                 /* move list */
                         w2=w2->punt;
            }
            w1->punt=w3;                       /* insert the new event sorted by time*/
            w3->punt=w2;
        }
}
/*---------------------------------------------------------------------------*/
void first_arrivals(){

/* generate first arrival for input 0,0,0 */
        generate_first_pack(&lpacch);
        uscita_mm1[0][0][0]=lpacch;
        insert_first_arrival(lpacch,0,0,0);
        pac_sim=1;
        lun_tot_offerta[0][0][0]=lpacch;

/* first arrivals for the rest of inputs */
        for (i=0;i<numout;i++)
            for (j=0;j<numfib;j++)
              for (k=0;k<numwdm;k++){
                        if ((i==0)&&(j==0)&&(k==0)){}
                        else{
                                generate_first_pack(&lpacch);
                                uscita_mm1[i][j][k]=lpacch;
                                insert_event(0.0, lpacch,i,j,k,ARRIVAL);
                                pac_sim++;
                                lun_tot_offerta[i][j][k]=lpacch;
                        }
              }
}
/*---------------------------------------------------------------------------*/

void eliminate_event() {

        struct evento * w3;

        if(lista_eventi->punt == NULL) {printf("Total time = %1.10f secs\n",lista_eventi->time);

        w3=lista_eventi->punt;         /* eliminate the first event from the list */
        free(lista_eventi);            /* updating the pointer */
        lista_eventi=w3;
}
/*---------------------------------------------------------------------------*/
void generator_mm1(int ingresso,int fiber, int l_onda,double *prox_arrivo,double *lun) {

        double tuscita;
        double tinter, lunghezza;
        int lung_in_byte;
        tinter=-log(1-ran2(&seed))/lambda;                   /* generate interarrival time */
        lunghezza=-log(1-ran2(&seed))*lungmedia;             /* and exponential length */
        lung_in_byte=(int)(lunghezza*GIGABITSEC/8+1);        /* trasform length into bytes */

        if((ting_mm1[ingresso][fiber][l_onda]+tinter)>=uscita_mm1[ingresso][fiber][l_onda])
            tuscita=ting_mm1[ingresso][fiber][l_onda]+tinter;
        else tuscita=uscita_mm1[ingresso][fiber][l_onda];

        ting_mm1[ingresso][fiber][l_onda]+=tinter
        uscita_mm1[ingresso][fiber][l_onda]=tuscita+lunghezza;


        *prox_arrivo=tuscita;                          /* fornisce i dati in uscita */

        /* +++++++++++++++++++++++++++++++++++++
        FORZATO A PROCESSO DI POISSON PURO
        +++++++++++++++++++++++++++++++++++++ */
        *prox_arrivo=ting_mm1[ingresso][fiber][l_onda];
        *lun=lunghezza;
}

/*-----------------------------------------------------------------------------*/

void minimax(int minormax, int *fib, int *wave, int *delay, int *numvalues, double
G[NMAXFIBPEROUT][NMAXWDM][MAXFDL], int numfib, int  numwdm,int numfdl ){

  /*Function providing coordinates of  mins (if first parameter is zero) or maxs (any other value) of a
given 3D array, as well as the number of minimums or maximums (numvalues). Pointers *fib, *wave*,
*delay will store coordinates of mins/maxs*/

        int i,j,k, tempnum;
        double value;
        tempnum=0;
        if (minormax==0){   /* find the minimum positive or null value */
        value=INFINITO;

        for (k=0;k<numfdl;k++)
           for (i=0;i<numfib;i++)

             for (j=0;j<numwdm;j++){
             if ((G[i][j][k]<value)&&(G[i][j][k]>=0)){
               value=G[i][j][k];
               tempnum=0;
               fib[tempnum]=i;
               wave[tempnum]=j;
               delay[tempnum]=k;
               tempnum++;
             }
             else if ((G[i][j][k]==value)&&(G[i][j][k]>=0)){
               fib[tempnum]=i;
               wave[tempnum]=j;
               delay[tempnum]=k;
               tempnum++;
             }
           }
         }

         else {            /*find the maximum */
           value=-1;
        for (i=0;i<numfib;i++)
         for (j=0;j<numwdm;j++)
        for (k=0;k<numfdl;k++){
          if ((G[i][j][k]>value)&&(G[i][j][k]>=0)){
            value=G[i][j][k];
            tempnum=0;
            fib[tempnum]=i;
            wave[tempnum]=j;
            delay[tempnum]=k;
            tempnum++;
          }
          else  if ((G[i][j][k]==value)&&(G[i][j][k]>=0)){
           fib[tempnum]=i;
           wave[tempnum]=j;
           delay[tempnum]=k;
           tempnum++;
          }
        }
      }
      *numvalues=tempnum;

}
/*---------------------------------------------------------------------------*/

void search_function(int tempout, int fib, int wl, int delay, double ta, double duration){

  if ((gp[tempout][fib][wl][delay]==NULL)||(gp[tempout][fib][wl][delay]>tstart>=ta+delay*T+duration)||
(gp[tempout][fib][wl][delay]->tend<=ta+delay*T)){                    /*Wrong delay gap...*/

      Heads[fib][wl][delay]=-INFINITO;
      Tails[fib][wl][delay]=-INFINITO;

  }else{

      Heads[fib][wl][delay]=ta+delay*T-gp[tempout][fib][wl][delay]->tstart;
      if ((delay==0)&&(Heads[fib][wl][delay]>=0)) Heads[fib][wl][delay]=0;
      Tails[fib][wl][delay]=gp[tempout][fib][wl][delay]->tend-ta+delay*T-duration;
  }

AUX[fib][wl][delay]=-INFINITO;

if ((Heads[fib][wl][delay]>=0)&&(Tails[fib][wl][delay]>=0)){
    switch(pol){

      case G_VF:
         AUX[fib][wl][delay]= Heads[fib][wl][delay]+Tails[fib][wl][delay];          /* creates matrix to
                                                         be minimized/maximized at the select function*/
       break;                                             /* it is not exactly H, as if T is negative H value is
                                                         not useful*/
      case G_NoVF:
         if (gp[tempout][fib][wl][delay]->tend==INFINITO) /*No packets scheduled after the gap...*/
                  AUX[fib][wl][delay]= Heads[fib][wl][delay];
         break;

      case D_VF:
         AUX[fib][wl][delay]=(numfdl-1)*T +max_lungmedia-delay*T-Heads[fib][wl][delay];
                                                                 /*AUX takes value of matrix D*/
      break;
```

```
        case D_NoVF:
            if (gp[tempout][fib][wl][delay]->tend==INFINITO){ /*No packets scheduled after the gap...*/
                        AUX[fib][wl][delay]= delay+Heads[fib][wl][delay];
            }
            break;
    }

}}
/*-----------------------------------------------------------------------------*/

int delay_case_tie(int numvalues,int *c ){
/* function selecting the minimum with a smaller delay, or a random value if many minimums have the
same delay*/
        int numofminc=0;
        int mindel[numvalues];
        int minc=INFINIT;
        int i;

        for (i=0;i<numvalues;i++){
            if (c[i]<minc){                 /*for to generate a vector with all minimum values*/
                        minc=c[i];
                        mindel[0]=i;
                        numofminc=1;
            } else if (c[i]==minc){
                        mindel[numofminc]=i;
                        numofminc++;
            }
        }   //end for

  return mindel[randinteger(numofminc)];
}

/*-----------------------------------------------------------------------------*/

int search_conv(int inwdm, int outwdm) {

        int c=0;      // It remains = 0 if no conversion is needed

         if(inwdm!=outwdm) {
            if(num_conv[inwdm][outwdm]>0) { // N in->out case 1
                        c=1;
            } else if(num_conv[inwdm][numwdm]>0) { // N in->F case 2
                         c=2;
            } else if(num_conv[numwdm][outwdm]>0) { // N F->out case 3
                         c=3;
            } else if(num_conv[numwdm][numwdm]>0) { // N F->F case 4
                        c=4;
            } else { // No converters available
                        c=-1;
            }
        }
         return c;
}

/*-----------------------------------------------------------------------------*/

void sel_conv(int conv_t, int inwdm, int outwdm) {

        switch(conv_t) {
            case 1:
                        num_conv[inwdm][outwdm]--;
                        tempo_conv1[conv1_occupati]=tempo_conv1[conv1_occupati] + (lista_eventi->time -
oldtime1);
                        conv1_occupati++;
                        oldtime1=lista_eventi->time;
                        break;
            case 2:
                        num_conv[inwdm][numwdm]--;
                        break;

            case 3:
                        num_conv[numwdm][outwdm]--;
                        break;

            case 4:
                         num_conv[numwdm][numwdm]--;
                         tempo_conv4[conv4_occupati]=tempo_conv4[conv4_occupati] + (lista_eventi->time -
oldtime4);
                        conv4_occupati++;
                        oldtime4=lista_eventi->time;
            break;
  }
}

/*-----------------------------------------------------------------------------*/

void set_free_conv(int conv_t, int outwdm){

            if(conv_t==1){ // N in->out case 1
                        num_conv[lista_eventi->input][outwdm]++;
                        tempo_conv1[conv1_occupati]=tempo_conv1[conv1_occupati] + (lista_eventi->time -
oldtime1);
                        conv1_occupati--;
                        oldtime1=lista_eventi->time;
            }else if(conv_t==2){ // N in->F case 2
                        num_conv[lista_eventi->input][numwdm]++;
            }else if(conv_t==3){ // N F->out case 3
                        num_conv[numwdm][outwdm]++;
            }else if(conv_t==4){ // N F->F case 4
                        num_conv[numwdm][numwdm]++;
                        tempo_conv4[conv4_occupati]=tempo_conv4[conv4_occupati] + (lista_eventi->time -
oldtime4);
                        conv4_occupati--;
                        oldtime4=lista_eventi->time;
            }
}

/*-----------------------------------------------------------------------------*/

int sel_fib_wdm(int tempout,double t,double duration,int *del, double *tau, int *outfib, int *outwdm){

        int numvalues=0;
        int tempvalue;
        int found;

        int a[numfib*numwdm*numfdl],b[numfib*numwdm*numfdl],c[numfib*numwdm*numfdl];

            for (i=0;i<numfib;i++){
```

```
        for(j=0;j<numwdm;j++){

                conv_type[i][j] = search_conv((int)lista_eventi->lung_onda,j);

                if (conv_type[i][j] == -1) { // there isn't any converter free, all the values of AUX for
this fiber and out wavelength to -infinito
                        for (k=0;k<numfdl;k++) {
                                AUX[i][j][k]=-INFINITO;
                        }

                } else {
                        for (k=0;k<numfdl;k++){
                                search_function(tempout,i,j,k,t,duration);
                        }
                }
        }
    }

    switch(pol){
        case G_VF:
          minimax(0,a,b,c,&numvalues,AUX,numfib,numwdm,numfdl); /*find minimum values in
matrix Heads*/
          if (numvalues==0) {return 1;}
          if (numvalues>1){                          /*more than one minimum value, choose the one*/
            tempvalue=delay_case_tie(numvalues,c);   /*with smaller delay. If there is more than one */
            *outfib=a[tempvalue];                     /*select randomly among them */
            *outwdm=b[tempvalue];
            *del=c[tempvalue];
          }else if (numvalues==1){
            *outfib=a[0];
            *outwdm=b[0];
            *del=c[0];
          }
          break;

        case G_NoVF:
          minimax(0,a,b,c,&numvalues,AUX,numfib,numwdm,numfdl); /*find minimum values in
matrix Heads*/
          if (numvalues==0) {return 1;}
          if (numvalues>1){                          /*more than one minimum value, choose the one*/
            tempvalue=delay_case_tie(numvalues,c);   /*with smaller delay. If there is more than one */
            *outfib=a[tempvalue];                    /*select randomly among them */
            *outwdm=b[tempvalue];
            *del=c[tempvalue];
          }else if (numvalues==1){
            *outfib=a[0];
            *outwdm=b[0];
            *del=c[0];
          }
        break;

        case D_VF:
          minimax(1,a,b,c,&numvalues,AUX,numfib,numwdm,numfdl);  //finds max values in matrix D
        if (numvalues==0) return 1;
        else{
            *outfib=a[0];
            *outwdm=b[0];
            *del=c[0];
          }
```

```
               break;

             case D_NoVF:
               minimax(0,a,b,c,&numvalues,AUX,numfib,numwdm,numfdl);  /* finds min values in matrix
AUX containing delays */
                  if (numvalues==0) {return 1;}
               if (numvalues==1){
                 *outfib=a[0];
                 *outwdm=b[0];
                 *del=c[0];
               }else if (numvalues>1){
                 tempvalue=0;

 for (i=0;i<2;i++){

                      if ((Heads[a[i]][b[i]][c[i]]<tempgap)&&(Heads[a[i]][b[i]][c[i]]>0)) {
                              tempgap=Heads[a[i]][b[i]][c[i]];
                              tempvalue=i;
                      }
                 }

                 *outfib=a[tempvalue];                  /*select the one with smaller gap*/
                 *outwdm=b[tempvalue];
                 *del=c[tempvalue];

               }

               if(lista_eventi->lung_onda==*outwdm) pacch_senzaconv++;
               break;

            return 0;

}
/*--------------------------------------------------------------------------*/

void update_gp(double tempo){
        /*after current time has been updated, pointer to gaps in every channel and delay
        must be updated--- */

  struct gap *g;
  g=(struct gap *)malloc(sizeof(struct gap));
  int t;

  for(i=0;i<numout;i++) {
        for(j=0;j<numfib;j++) {
            for(k=0;k<numwdm;k++) {
                      for(l=0;l<numfdl;l++) {
                              g=gp[i][j][k][l];
                              t=l;
                              while(g==NULL) {g=gp[i][j][k][t+1];
                              t++;
                      }
                      while(((g->tstart<tempo+l*T)&&(g->tend<tempo+l*T))){
                              g=g->next;
                      }

                        if(g->tstart<tempo+l*T)              gp[i][j][k][l]=g;
                        else if(g->tstart<tempo+(l+1)*T) gp[i][j][k][l]=g;
                        else if(l==(numfdl-1))               gp[i][j][k][l]=g;
                        else gp[i][j][k][l]=NULL;

                        if(l+1<numfdl){
                                if((gp[i][j][k][l+1]!=NULL)&&(gp[i][j][k][l]!=NULL)){
                                        if(gp[i][j][k][l+1]->tstart<gp[i][j][k][l]->tstart){
                                                gp[i][j][k][l+1]=gp[i][j][k][l];
                                        }
                                }
                        }

                      }
              }
          }
  }
}

/*---------------------------------------------------------------------------*/

int update_gaplist(double tempo, double duration, int out, int fib, int wl, int delay){
        /*procedure to update the list of gaps in every channel after a packet is scheduled*/
        struct gap *g1,*g2,*g3;

        while (lista_gap[out][fib][wl]->tend <= tempo) {
            g1=lista_gap[out][fib][wl];
            lista_gap[out][fib][wl]=lista_gap[out][fib][wl]->next;
            free(g1);
         }
        lista_gap[out][fib][wl]->prev=NULL;
        g2=lista_gap[out][fib][wl];
        while (g2->tend < tempo+delay*T){
            if (g2->next!=NULL) g2=g2->next;
            else {
                    return 0;
            }
        }

      /*if the gap where the packet will be put isn't the last, keep the others in g3*/
         if (g2->next!=NULL)  g3=g2->next;
        else g3=NULL;

         if ((tempo+delay*T > g2->tstart) && (tempo+delay*T+duration < g2->tend)){
                                                        /* packet in the middle of a gap */
            g1=(struct gap *)malloc(sizeof(struct gap));
            g1->tend=g2->tend;
            g1->tstart=tempo+delay*T+duration;
            g2->tend=tempo+delay*T;
            g2->next=g1;
            g1->prev=g2;
            if (g3!=NULL) {
                    g3->prev=g1;
            }
            g1->next=g3;                        /*link the rest of gaps (if there are more) */
        }

        if ((tempo+delay*T == g2->tstart) && (tempo+delay*T+duration < g2->tend)){
                                                /*packet at the beginning of gap*/
            g2->tstart=tempo+delay*T+duration;
        }
        if ((tempo+delay*T > g2->tstart) && (tempo+delay*T+duration == g2->tend)){
                                        /*packet at the end of the gap*/
            g2->tend=tempo+delay*T;
        }

         if ((tempo+delay*T == g2->tstart) && (tempo+delay*T+duration == g2->tend)){
                                        /*packet filling a whole gap*/
            g1=g2;
            if (g2->prev!=NULL){
                        g2=g2->prev;
                        g2->next=g3;
            }else{
                        if (g3 != NULL)
                        g3->prev=NULL;
                        lista_gap[out][fib][wl]=g3;
                        }
                        free(g1);
        }
return 0;
}

/*---------------------------------------------------------------------------*/

void manage_events(){

        int testvar=0;
        //manage events while list isn't NULL
        printf("START MANAGING EVENTS\n");
        while(lista_eventi!=NULL){

        switch(lista_eventi->tipo) {
            case ARRIVAL:               /* starts the management of event ARRIVO */
                        tempo=lista_eventi->time; /* aggiornamento della variabile tempo */
                         update_gp(tempo);
                        /* generates next arrival and insert it in the event */
                        if(pac_sim<pacch) {
                          generator_mm1(lista_eventi->input, lista_eventi->fiber,lista_eventi-
>lung_onda,&tprossimo,&lpacch);
                          insert_event(tprossimo,lpacch,lista_eventi->input,lista_eventi->fiber,lista_eventi-
>lung_onda,ARRIVAL);
                          pac_sim++;

                                    /* generazione dell'uscita e selezione della lunghezza d'onda */
                        out=randinteger(numout);
                                    /*Update gap pointers to new time*/
                        if (sel_fib_wdm(out,tempo,lista_eventi->lung,&rit_ass,&tau,&fib,&lout)==0){  /
                                    /*selects delay, fiber & wavelength */
                                    sel_conv(conv_type[fib][lout],(int)lista_eventi->lung_onda,lout);
                                    double conv_time;
                                    conv_time= lista_eventi->lung; //conversion time in seconds

                        /*gettimeofday(&monitortime,NULL);
                        double time1 = (double)monitortime.tv_sec+(monitortime.tv_usec)/1000000.0;*/

                                    insert_event(tempo+conv_time,lista_eventi->lung,lista_eventi-
>lung_onda,conv_type[fib][lout],lout,FREE_CONV);

                                    scelta_wdm[out][fib][lout]++;
                                    lun_tot_eff_smalt[out][fib][lout]+=lista_eventi->lung;
                                    update_gaplist(tempo,lista_eventi->lung, out, fib, lout, rit_ass);

                        /*gettimeofday(&monitortime,NULL);
                        double time2 = (double)monitortime.tv_sec+(monitortime.tv_usec)/1000000.0;
                        if((pac_sim%(pacch/100))==0) {
                        printf("Time: %.6f\n",time2-time1);
                        }*/

                                    tfreeout[out][fib][lout]=tempo+(double)rit_ass*T+lista_eventi->lung;
                        }else {   pacch_persi++;
                        }

                        eliminate_event();

                        /*stampa su stderr lo stato della simulazione */
                        if((pac_sim%(pacch/100))==0) {
                          fprintf(stderr,"%d\r",pac_sim/(pacch/100));
                          fflush(stderr);
                        }
                        break;


            case FREE_CONV:

                        set_free_conv((int)lista_eventi->fiber, (int)lista_eventi->lung_onda);
                        eliminate_event();
                        break;

                 }
        } /* end switch, end while */
}


/*--------MAIN------------------MAIN-----------------MAIN--------------------*/

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

        clock_t t_ini, t_fin;
        double secs;
        t_ini = clock();
        int c;


        pol = 4; numout=2;numfib=4;numwdm=8;numfdl=2;Tbyte=1000;pacch=5000000; lungbyte=1000;
carico=0.8; seed=2; nconv1=0; nconv2=0; nconv3=0; nconv4=70;

        //getoptions
        while ((c = getopt (argc, argv, "a:p:f:w:d:g:n:l:c:s:h:i:j:k")) != -1)   // : if value is required afer optio
            switch (c){
                        case 'a':  //algorithm
                                    pol = atoi(optarg);             //add option for constant random seed!!!!
                                    break;
                        case 's':  //constant seed
                                    seed = time (NULL);
                                     break;
                         case 'p': // input/output ports
                                    numout = atoi(optarg);
                                    break;
                        case 'l': // average length in bytes
```

```
                        lungbyte = atoi(optarg);
                        break;
                case 'f': //number of fibers per in/output
                        numfib = atoi(optarg);
                        break;
                case 'w': //number of wavelengths per fiber
                        numwdm = atoi(optarg);
                        break;
                case 'd': //number of fiber delay lines
                        numfdl = atoi(optarg);
                        break;
                case 'g': //granularity in bytes
                        Tbyte = atoi(optarg);
                        break;
                case 'n': //number of packets to be simulated
                        pacch = atoi(optarg);
                        break;
                case 'c': //traffic load
                        carico = atoi(optarg);
                        break;
                case 'h':
                        nconv1 = atoi (optarg);
                        break;
                case 'i':
                        nconv2 = atoi (optarg);
                        break;
                case 'j':
                        nconv3 = atoi (optarg);
                        break;
                case 'k':
                        nconv4 = atoi (optarg);
                        break;
        }
    T=Tbyte*8/GIGABITSEC;
    lungmedia=lungbyte*8/GIGABITSEC;    /* trasforms average length from bytes to seconds */
    lambda=carico/lungmedia;            /* calculates arrivals frequence for MM1 generator */
    MAXRIT=numfdl-1;

    init_vars();
    first_arrivals();

    manage_events();

    double pe=(double)pacch_persi/pacch;
    printf("  packet av. length: %f // granularity: %f // \n ",lungbyte, Tbyte);
    printf("       loss probability: %1.10g\n",pe);

    t_fin = clock();

    secs = (double)(t_fin - t_ini) / CLOCKS_PER_SEC;
    double tt1=0;
    double tt4=0;

    if(nconv1 != 0){
        for(i=0;i<=(nconv1*(numwdm-1)*(numwdm));i++) {
                printf("%d convertitori tipo1 occupati durante %1.10f secondi\n",i, tempo_conv1[i]);
                tt1=tt1+tempo_conv1[i];
        }
        printf(" tt1= %1.10f secondi\n",tt1);
        }

        if(nconv4 != 0){
           for(i=0;i<=nconv4;i++) {
                    printf("%d convertitori tipo4 occupati durante %1.10f secondi\n",i, tempo_conv4[i]);
                    tt4=tt4+tempo_conv4[i];
           }
           printf(" tt4= %1.10f secondi\n",tt4);
        }

        return 0;

  }
```