Contents

List of Figures

List of Tables

Abstract

In a high energy physics experiment, *event building* is the process of collecting data fragments, spread among many buffers, in order to reassemble the original event. Each assembled event is assigned to a processor in a large farm for on-line analysis.

The goal of the present work was to demonstrate the feasibility of implementing event building protocol in the processors embedded in the new generation of Network Interface Cards (NIC), in particular for Gigabit Ethernet technology. This implied the software implementation of the event building algorithms, their adaptation for their operation in the embedded processors, the development of a model of the event building conditions and the measurement and interpretation of performance of this solution. A comparison with the standard implementation on host processors provides the yardstick to judge the suitability of the proposed method.

This development was carried out within the team developing the Data Acquisition (DAQ) system for LHCb, one of the four experiments that have been approved for the future high energy collider LHC (Large Hadron Collider) at CERN.

The work requirements were to provide event building for a data acquisition system, which must sustain an input rate of 40 kHz, with some 100 data sources providing event fragments with a size of $\sim$ 1 kByte. The aggregate data acquisition rate is 4 GByte/s.

It has been demonstrated that event building on a Gigabit Ethernet smart NIC (the Tigon 2 Network Interface Card, by Alteon *Web*Systems) at a frequency of almost 100 kHz is sustainable for fragments with a size up to 1465 Bytes.

Introduction

In the 20th century the need of gathering, processing and distributing information has become a dominant feature of our society.

"Among the other developments, we have seen the installation of worldwide telephone networks, the invention of radio and television, the birth and unprecedented growth of the computer industry, and the launching of communication satellites.

Because of the rapid technological progress, these areas are rapidly converging, and the differences between collecting, transporting, storing and processing information are quickly disappearing." [34]

Although the computer industry is young compared to the other industries, computers have made incredible progress in a very short time and they are now one of the leading system for handling information, besides telephones, televisions and radios.

The idea of the "computer centre" as a room with a large computer where users bring their work for processing has now totally disappeared. The old concept of a single computer serving all the needs of an organisation, like an university or a news agency, has also disappeared. It has been replaced by a model in which a large number of single-purpose computers, interconnected one to each other, works together and accomplishes jobs for many users.

These systems are called *computer networks* and they are one of the biggest challenges, in which the computer industry invests more. In few years we have migrated from small networks with a transfer power of few bits per second to wide area networks with gigabits transfer capabilities.

Due to this rapid and large development, networks have become quite common in many ordinary life aspects. Almost all business companies,

all universities, all researching activities have their network in order to share information inside the system and also outside, with the rest of the world. New buildings or transport systems (like trains) are built also with the network cabling prearrangement, beside the telephone cabling.

Nowadays, network capabilities can be used in many different fields but always with the same purposes: handling and moving around information.

Since few years, the possibility to take advantage of switching networks has been considered by the physics scientific community, in order to collect and store the data coming out from the physics experiments. New systems of data acquisition based on switching networks have been being studied. These are sort of "dedicated systems", because the initial requirements are usually very restricted and rigid, and the data rate is always high. Therefore, a data acquisition system must be perfectly studied, designed and developed, in order to be able to fulfil all the physics needs and to assure the experiments results.

The main topic of this work is to understand and explain what the event building really is, analysing its purposes, its general requirements and its typical solutions.

Then, a case of study will be proposed: the event building of the LHCb experiment. It will be analysed thoroughly in detail and the possibility to apply network capabilities to its requirements will be studied. Finally a data acquisition system, based on the new Gigabit Ethernet technology, will be proposed as a possible solution to the LHCb event building problem.

Working Environment

This work has been developed at CERN (European Laboratory for Particle Physics) [12], in Switzerland. The next future project of CERN is to build a new particle accelerator, the LHC (Large Hadron Collider) [13], which will start operating in 2005.

A collider is a machine which is designed to make bunches of particles circulate in opposite directions and to accelerate them until they achieve the desired energy. At this point, the trajectories of the two beams are

deflected by means of magnetic fields so that the particles collide. Surrounding the interaction points, experiments are built in order to detect the outcoming particles produced in the collision of the primary particles which the two beams consist of. The number of interesting physics interactions (*signal* events), which an experiment sees every second, is proportional to the so called *luminosity* of the accelerator. The luminosity L is defined by the following formula:

$$L = \frac{fkN^2}{4\pi\sigma_x\sigma_y}$$

where:

- f is the revolution frequency of the bunches
- k is the number of bunches circulating in the machine
- N is the number of particles in each bunch
- σ_x and σ_y are the transverse dimensions of the bunches

LHC is a circular accelerator which will take the place of LEP (Large Electron-Positron collider), inside the 27 Km underground tunnel in the Geneva area. It will bring protons into head-on collisions at the highest energies ever achieved; part of the LHC program will be dedicated to collide heavy nuclei. Inside LHC 2835 bunches of protons will circulate in each direction with an energy of 7 TeV, for a bunch crossing frequency of 40 MHz. Most of the 40 million interactions per second which will occur at LHC are not of interest from the physics point of view and they are referred to as background interactions. Therefore, each experiment, depending on its particular research field, will have to select among this big number the few thousands interesting events by means of a complex of hardware and software tools, called *trigger*. The data belonging to the events selected by the trigger are thus stored on tape for the physics analysis.

Four main experiments will exploit the LHC: the CMS experiment and the ATLAS experiment, mainly devoted to the research of the Higgs

boson, the ALICE experiment, which will study heavy nuclei interactions, and the LHCb experiment, whose physics motivation is shortly explained below.

Event Building and LHCb DAQ system

As previously mentioned, in high energy physics experiments, not all the collisions generate useful data for the analysis. Therefore, decision criteria must be immediately applied to the data at each collision time, in order to decide if either the produced event is an interesting one, from the physics point of view, and thus it has to be stored, or if it is an event which can be discarded, due to its lack of physics interest.

Because of the large amount of data and the high complexity of the decision criteria, the evaluation of an event is accomplished in several steps, or trigger levels. Depending on the experiment, one or two decision levels, based on simple topology and energy criteria, will be implemented in dedicated hardware logic, while one or more levels of software decisions are then applied to the remaining data for the final storage.

Only a small fraction of the total amount of data from each event is required for use in the initial decision. The rest of the data is scattered over many buffers and must be then collected in one place for the software analysis. The process of collecting the data fragments, spread among several buffers, into one destination, in order to reassemble the original physics event, is called *event building*.

LHCb [21] is the most recently approved of the four experiments which will run on the CERN's LHC accelerator. It is a special purpose experiment designed to study the CP violation [1] in the decay of b quarks. It has four levels of decision: Level-0 and Level-1, which are hardware driven, and Level-2 and Level-3, which are software decision level.

The role of the data acquisition (DAQ) system is to read data coming out from the Level-1 decision, to assemble complete events and to provide sufficient CPU power for the execution of the Level-2 and Level-3

[1] CP violation was first discovered in neutral kaon decays in 1964. Its origin is still one of the outstanding mysteries of elementary particle physics. More details regarding the LHCb physics motivation can be found in [21], Chapter 1.

algorithms. The flow of data through the DAQ system is being studied using simulation data. The input rates are determined by the average event size, which is 100 kByte, and the Level-1 accept rate, nominally 40 kHz. This result in a total data acquisition rate of 4 GByte/s.

The current design contains the following functional components. The Readout Units (RUs) receive data from one or more Level-1 decision links and assemble the fragments belonging to each event into sub-events. Full event building is achieved by having all RUs dispatch their data into a readout network such that the fragment belonging to the same event arrive at the same destination. Complete events are assemble at the destination by a unit called the Sub-Farm Controller (SFC). This unit also has the role of allocating each event to one of the free CPUs it manages, and the Level-2 and Level-3 algorithms are executed on this CPU. Accepted events are written to storage devices.

Our Contribution

This work aims at studying and possibly realising a solution for the LHCb DAQ system, in order to manage the flow of data and to ensure events are assembled correctly. It is principally focused on the readout network and on its interfaces with the RUs (sources) and the SFCs (destinations). It is based on a *full-readout protocol*, where data are immediately routed through the readout network to the destinations as soon as they appear at the RUs.

This work will concentrate on a particular network technology: the new Gigabit Ethernet standard. In particular, we will work with a special Gigabit Ethernet Network Interface Card, which has an embedded processor inside. This choice is made in order to verify the possibility to develop a new data acquisition system in which the event building functionalities are executed inside the network interface. The final aim of the work is to study and realize this new embedded event building architecture and to verify that it is a feasible solution for the performance requirements of the LHCb DAQ system.

Structure of the Work

Before going into the detailed treatment of the argument, we summarise the work structure.

The first two chapters introduce the general concepts and requirements of the event building problem. The different strategies adopted in the past in order to face it are then described and some considerations on the applicability of switching network capabilities to the event building system are discussed.

In the following chapter, Chapter 3, the possible network technology competitors for the event building system are quickly described: ATM, Ethernet and Myrinet. More emphasis is given to the description of the Gigabit Ethernet standard, because it will be the baseline technology of the embedded event building project.

Chapter 4 introduces in all details the case of study of this work: the LHCb event building. The DAQ architecture and requirements are fully analysed, with the description of the RUs, the readout network and the SFCs and their corresponding roles and tasks. At the end of the chapter the boundaries and the scope of the work are specified.

In Chapter 5 there is the description of three different event building protocols developed and their overhead measurements are shown.

Chapter 6 and 7 are dedicated to the description of embedded systems, especially the one used for our purposes. The motivations for the embedded event building project are specified and the performance of the studied Gigabit Ethernet "smart" NIC are shown.

Finally, in Chapter 8, results are presented which show that embedded event building on a Gigabit Ethernet technology is feasible and fulfil the LHCb DAQ system requirements.

Chapter 1

The Problem

The demands on data acquisition systems for high-energy physics have been increasing at a rapid rate due to the higher luminosities and interaction rates. From the early days of high energy physics to some of present-day's experiments, when readout of a physics event is initiated, triggering on subsequent events is disabled until the readout is complete. Other factors in an experiment contribute to the experiment "dead-time" but readout is the dominating factor. The "dead-time", measured as a percentage, is approximately equal to the product between event readout time and trigger rate. Whenever possible it is held to less than 10%.

Now, with very high interaction rates and consequently very high trigger rates, readout time is an even larger fraction of the time between triggers. New techniques for physics event readout are now essential if we are to minimise dead-time. Several events worth of data must be buffered on or near the detector during triggering, such that when the readout is triggered the buffered data for that event may be readout quickly, without disabling the trigger (*pipe-line*).

1.1 Event Building Overview

Only a small fraction of the total data from each event is required for use in the initial trigger decision. The remaining data is scattered over many front-end buffers and must be collected in one place for detailed analysis.

An event builder is the device in a data acquisition system which provides a connection between the individual data sources (detector front-end electronics) and the data destinations (high-level event processors or online data storage). The process of collecting the data fragments coming from all sources into one destination is called *event building* (independently of whether or not the complete set of fragments is assembled). Figure 1.1 summarises the main principles of event building.

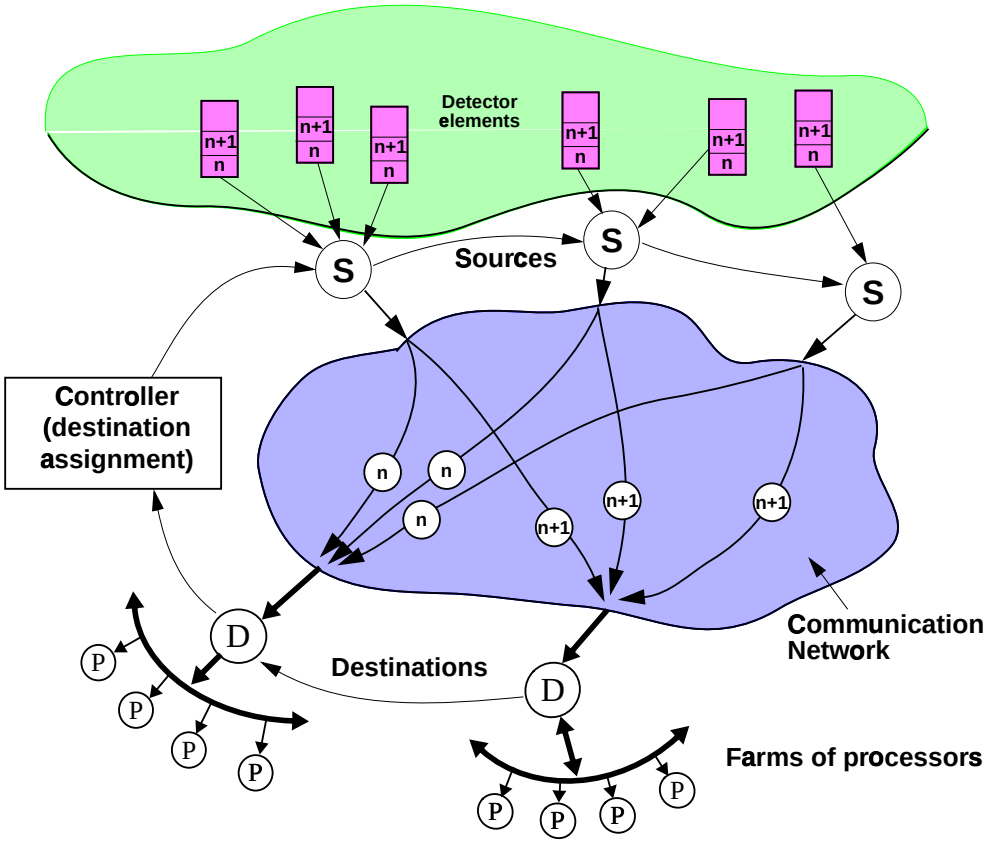

Figure 1.1: Principle of event building

Event builders have evolved in time from simple single channel 'funnels' through a minicomputer bus, to multiple parallel channels (each

with their own 'funnel' or characteristics) feeding arrays (farm) of processors. Regardless of the implementation, all event builders function as a data multiplexer. If data rates are low, this multiplexing operation can take place over a single time-shared bus using software controlled selection of source and destination. This is the technique used in the majority of data acquisition systems in the past. High-speed event builders were not necessary because the data rates which could be supported by the sources and destination were limited. This situation have changed in recent years. The ability to acquire, digitise and buffer data has increased and, similarly, the performance of high-level processors and the density of on-line data storage have been both improved. Unfortunately, the speeds of standard buses used for event building have not improved at the same rate and so the event builder has become the bottleneck in the data flow.

There are two possible solutions to this problem. Either the trigger efficiency can be increased, limiting data rates to the bandwidth of the event builder, or the event builder bandwidth can be increased. Techniques for improving trigger efficiency are dependent on the experiment. Techniques for improving event builder bandwidth can be considered independently.

1.2 Data Acquisition Strategies and Requirements

Nowadays the search for rare processes at the Large Hadron Collider [1] (LHC) experiments will require operation at very high luminosities. For example, for proton-proton collisions, the expected background interaction rate will be 10^9 Hz at the full luminosity at one collision point of

[1]The Large Hadron Collider (LHC) is the new CERN accelerator which will start working in the year 2005. It will bring protons and ions into head-on collisions at higher energies than ever achieved before. This will allow scientists to penetrate still further into the structure of matter and recreate the conditions prevailing in the early universe, just after the "Big Bang" (more information about LHC can be found in [13]). Four big experiments will run on the LHC accelerator and they are: the ATLAS experiment, the ALICE experiment, the CMS experiment and the LHCb experiment.

the LHC. Bunch crossing occurring at a frequency of 40 MHz approximately, 25 interactions will occur per crossing. The average volume of zero-suppressed data associated with each bunch crossing (or event) is expected to be 1 MByte. Under these conditions a rejection factor from the trigger and data acquisition systems better than 10^6 is required in order to limit to reasonable volumes the data recorded on mass storage. To achieve the required rejection factors the experiments are adopting multi-level trigger strategies [2].

Depending on the experiment, one or two levels of trigger decisions (Level-1 or Level-0 plus Level-1), based on simple topology and energy flow criteria, will use coarse granularity data and will be implemented in dedicated hardwired logic. The fine granularity data from those bunch crossing that are accepted will be locally readout, formatted into event fragments and stored in readout buffers. The background rejection ratio will be further improved on-line by applying one or more levels of software triggers using the full-granularity data held in these readout buffers.

The required CPU power will be provided by a scalable farm of processors. Each successive event candidate is to be assigned to a different processor, which will access the distributed fragments of the assigned event and execute the filtering algorithms. A switching network will interconnect the readout buffers with the members of processor farm in order to allow the assembly of event fragments into (partial or complete) event records accessible by the processors.

Figure 1.2 [21] shows the complete architecture of the trigger and data acquisition system of the LHCb experiment and gives an idea about the dimensions of these complex kinds of apparatus.

[2] The trigger is a function of:

$$T\left(\begin{array}{c} Eventdata\&Apparatus \\ Physiscschannels\&Parameters \end{array}\right) \Rightarrow \begin{array}{c} REJECTED \\ ACCEPTED \end{array}$$

Since the detector data are not all promptly available and the function is highly complex, T(...) is evaluated by successive approximations called: TRIGGER LEVELS^{0,1,2,3} (possibly with zero dead-time).

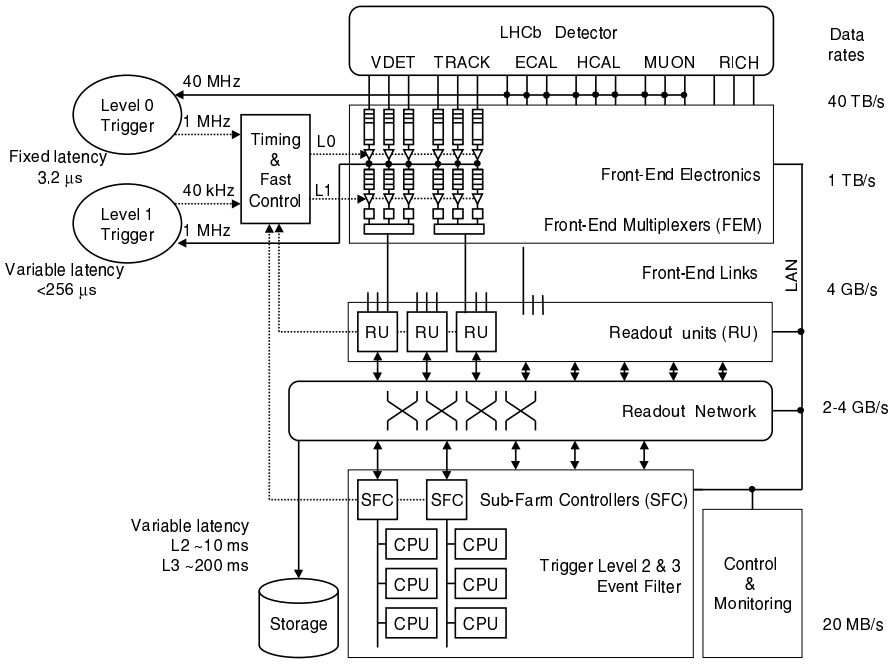

Figure 1.2: General architecture of the trigger and data acquisition system for the LHCb experiment

1.2.1 Partial Event Building versus Full Event Building

Very high (Level-0) Level-1 trigger rates and large event sizes can preclude the use of complete event data in the Level-2 triggers and force the choice of algorithms that operate on a subset of event data. This fact leads to what is called partial event building, which can be implemented in two different ways depending on the trigger strategy.

For example, the Level-1 trigger of CMS [16] is based on: a static subset of event data, the calorimeter and muon detectors, but does not use the tracker (which produces the largest part of event data). For the events accepted by the Level-2 trigger (at least an order of magnitude fewer than those that pass Level-1) the tracker data is then read

out and the more time-consuming and selective Level-3 algorithms are subsequently executed on the whole event data.

An other solution, implemented by the ATLAS experiment [22], consists in employing a dynamic partial event building strategy. In this architecture all detectors may contribute to the Level-2 trigger decision, but the algorithms are to be executed on subsets of event data guided by *Regions of Interest* (RoI) containing the physics *features* that caused the acceptance by the Level-1 trigger. The RoIs, which change for each event are located via pointers passed from the Level-1 trigger system. A sophisticated system is required to collect the data for the Level-2 trigger. Several solutions have been proposed regarding this topic and they can be found in [26] or in [22] (last review 31 March 2000, ch. 5 "DAQ/EF-1", pp. 27–56).

In the case of the LHCb experiment, a lower luminosity and the implementation of two levels of triggers (Level-0 and Level-1) on partial data allow to envisage to transfer all the data of accepted events to a processor executing the trigger of Level-2, taking advantage, in this way, of the standard full event building technique.

1.2.2 Data Flow Requirements

Table 1.1 gives the design requirements for the triggers and the data acquisition for the four LHC experiments.

As we can see, the required aggregate bandwidth spans a wide range depending on the experiment target and the strategy adopted for the trigger. An estimated minimum aggregate throughput of 16 Gbit/s is required for the LHCb architecture, while at the other end of the spectrum a bandwidth of 600 Gib/s is required by the CMS architecture.

1.2.3 General requirements for trigger/DAQ

General requirements for an event builder are:

- *Event building rate and latency*:
  In CMS experiment for example, the event building bandwidth is

PARAMETER	ALICE	ATLAS & CMS	LHCb
Event size	40 MByte	1 MByte	100 kByte
Level-0 output rate	not applicable	not applicable	1 MHz
Level-1 output rate	1 kHz	100 kHz	40 kHz
Level-2 output rate	50 Hz	1–10 kHz	5 kHz
Level-3 output rate	not applicable	100 Hz	200 Hz
Switching bandwidth	30 Gbit/s	20–600 Gbit/s	16–32 Gbit/s
Data storage	10 Gbit/s	1 Gbit/s	160 Mbit/s

Table 1.1: Trigger and data acquisition parameters of the four LHC experiments

planned to be handled by a 1000 × 1000 switch. Thus, each of the 1000 destinations must be capable of sustaining an average event building rate of ∼ 100 Hz for the Level-2 and at the same time ∼10 Hz for the Level-3.

- *Achievable load factor*:
  In order to minimise the required hardware investment, the system design should be balanced such that, at maximum system throughput, the event builder aggregate bandwidth is used in a efficient way (>50%).

- *Commercial products*:
  The use of commercially available components is preferred wherever possible in order to minimise development costs, to profit from lower prices due to high volume production and to ensure long-term maintenance.

- *Open Standards*:
  In order to offer a good choice of suppliers and to ensure interoperability between components from different sources, the network technology should be open international standard that is widely adopted by industry.

- *Technology life-cycle*:

If possible, the life cycle of the chosen technology should match the one of the experiments, being sufficiently mature at the start of the experiments to ensure low prices, adequate performance and high reliability, and being still supported by industry at the end of the experiments (10–15 years later).

- *Scaling*:
  The event builder architecture (hardware and software) must be scalable, so that it can be expanded (in dimensions and/or speed) without encountering bottlenecks that limit the growth in performance.

- *Partitioning*:
  In order to support parallel development and testing of the different detector sub-systems, it must be possible to partition the event builder into a number of independent, concurrently running data acquisition systems.

- *Operations and Management aspects*:
  Due to the large scale of the event builder it will be indispensable that the system incorporates features to ease its operation and management; for example, fault location, fault isolation, good monitoring and diagnostic tools, etc.

1.3 Size of the Problem

In order to estimate the magnitude of the problem that an event builder must cover and solve, it can be useful to compare event building requirements with some other parameters, more common and understandable. For example, a comparison between event building needs and telephone traffic requirements could be helpful to realize the size of the problem.

If we focus in particular on the LHCb event building requirements (which are the real aim of this work), we can see that the requested aggregate switching bandwidth is at maximum 4 GByte/s. Because data will go only one-way, from readout sources to processor destinations, the event building protocol will use the network bandwidth only in one

direction but the network itself would be able to support, in principal, two-way traffic of 8 GByte/s.

Now telephone speech, when digitised, creates 64 kbit/s in each direction, or a total two-way digital traffic of 128 kbit/s. Thus, with an aggregate switching bandwidth of 4 GByte/s and a data traffic of 64 kbit/s, a maximum of 500.000 people can communicate by telephone in one direction and at the same time:

$$\frac{4 \times 10^9 \times 8\ (bits)}{64 \times 10^3\ (bits)} = \ 5 \times 10^5 = \ 500,000$$

Of course, telephone communications involve people from both sites of the line and so we can conclude that a switching bandwidth of 4 GByte/s would cover a telephone communication for 1 million people.

If we assume that during the day 20% of the whole population of a country is occupied in telephone communications at the same time[3], our switching bandwidth of 4 GByte/s, that will be used for the LHCb data acquisition system, will cover the needs of 5 millions people, which are, for example, the needs of more or less the 10% of the whole Italian population.

[3]This assumption could be overestimated, but we believe that 20% of a country population involved in telephone calls at the same time is not so far from reality, at least in some times of the day.

Chapter 2

Event Building

Event building is faced with a lot of problems which can meet different solutions, according to the available technologies and the physics needs claimed by each specific experiment. Big difficulties are related to the event traffic management: the choice of which connection solution between data generators and data analysers is the most suitable, the choice of which kind of system architecture and communication protocol are the right ones to obtain the best performance, are really hard and, at the same time, decisive choices. Much effort have been spent to propose new and different solutions and to better exploit the new emerging technologies.

This chapter deals with these topics and tries to explain all the distinct solutions considered in the past and nowadays, paying special attention to still unresolved problems.

2.1 Event Building and Interconnection Networks

Figure 2.1 shows a generic Interconnection Network (IN) used in multiprocessor and telecommunications systems. In high energy physics, the data source (S) is typically a detector subsystem and the destination (D) is a single or a farm of programmable processors. The IN and its

associated control is referred to as an "event builder". Because the pattern of data flow is well defined (unidirectional and evenly distributed), a general-purpose IN can often be simplified for use as an event builder.

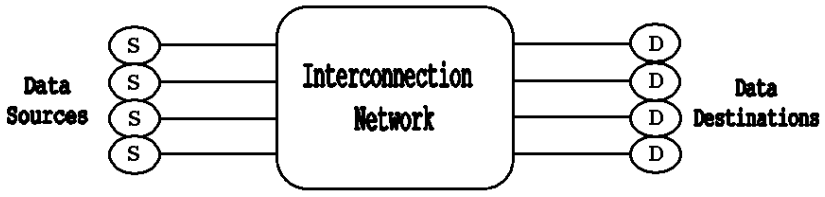

Figure 2.1: Generic Interconnection Network

Various kinds of techniques have been developed to improve the effects of using an IN on data acquisition systems and the following subsections give a brief overview about some of these historical architectures (further and in-depth information regarding this topic can be found in [9]).

2.1.1 Time-Shared Bus Architectures

The shared bus, shown in figure 2.2, is the most common method of interconnecting multiple sources and destinations. Bus bandwidths of several tens of MBytes can be supported during block transfers, but the average data rate is usually much less due to the overhead of processor setup and bus access protocols.

A single shared bus has the advantage of simple control and low cost. It also provides bidirectional transfer capability for down-load and initialisation. With repeaters it can scale indefinitely, although the total bandwidth does not increase and will usually decrease. The main cost element is the need for high-speed interface circuitry, which must be designed to support the full transfer rate of the bus even if each module is connected for only a small fraction of the total readout time. Failure of

the bus itself will disable the entire system, but failure of an individual module is usually not critical.

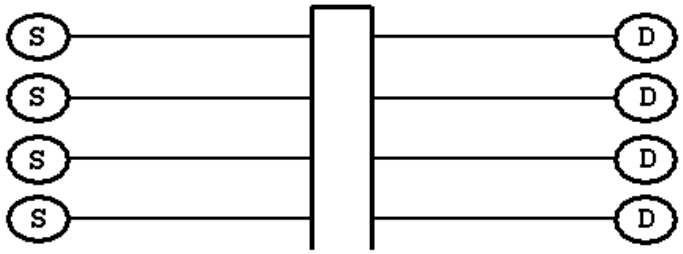

Figure 2.2: Shared Bus Interconnection Network

In most cases, data readout is controlled entirely by the processor. A processor will arbitrate for the bus and then read event data from each of the front-end buffers before releasing the bus to the next ready processor. In more complicated system, an intermediate event builder will read the front-end buffers and then write data directly into the memory of a selected processor. In some architectures several independent busses may operate in a tree-like structure to reduce dead-time at the front-end. However, without intermediate data compression, the net bandwidth in a tree-structured system is always equal to that of a single bus.

2.1.2 Multiple Bus Interconnection Network

Many standard bus specifications and multiprocessor implementations define a second or third bus (figure 2.3), which can operate in parallel with the main system bus. Additional bandwidth is gained only if processors do not contend for the same global resources. This approach is usually limited to one or two additional busses by the physical packaging constraints of standardised systems. A multiple bus architecture can be very reliable since failure of any single bus has no adverse effect other

than a reduction in total system bandwidth.

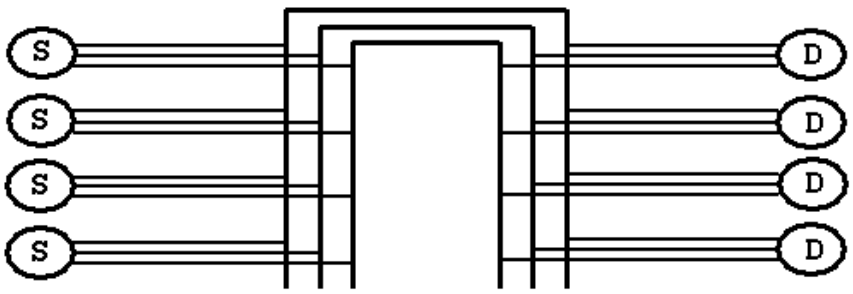

Figure 2.3: Multiple Bus Interconnection Network

With multiple busses, several events can be read out simultaneously. If events are assigned to specific buffers, then simple bus arbitration can be used to control readout sequences. Otherwise, a small amount of centralised control is necessary. As in any parallel system, the front-end buffers must be able to hold more than one complete event.

2.1.3 Multi-port Memory

Both the data sources and the destinations in the multiple bus architecture are multi-ported, but the same bandwidth can be obtained with multiple ports on only one side of the interconnect as shown in figure 2.4. Reliability is reduced because there is only one path from a particular source to a particular destination. Arbitration is handled by the multi-ported module rather than the bus.

This approach is still limited by the number of physical ports which can be supported by a module.

To allow greater expansion, multi-port memories can be further subdivided into an array of independent dual-ported buffers as shown in figure 2.5

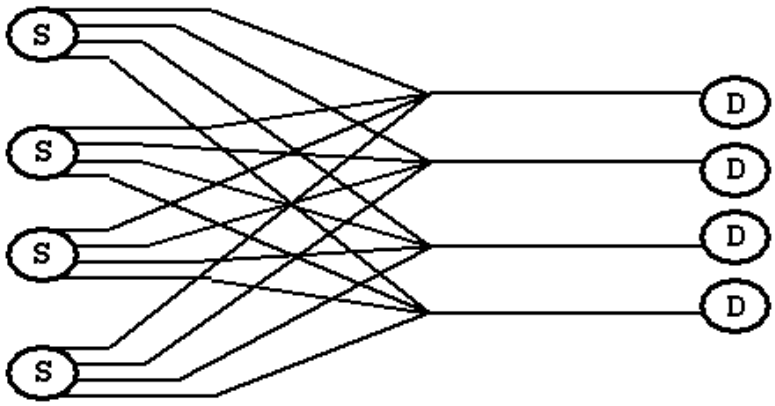

Figure 2.4: Multi-port Memory Interconnection Network

Dual-ported memory is easier to implement since it is available in the form of commercial integrated circuits (dual-ported static RAMs, FIFOs or video DRAMs).

With dual-ported memory, the limitation now becomes the total number of buffers required in a large system instead of the number of connections per buffer. The number of buffers can be reduced by using higher speed output busses, (allowing a rectangular instead of a square array) or possibly by implementing some kind of multistage memory architecture.

In the dual-ported memory architecture, the fragments of a given event are transmitted in parallel from the front-end subsystem to buffers in a selected row. These fragments are then readout sequentially by a processor while the next event is being transmitted to another row buffer.

2.1.4 Crossbar and Other Switch-Based Interconnection Networks

The dual-ported memory architecture in figure 2.5 is actually a form of buffered crossbar switch. A crossbar switch provides a complete,

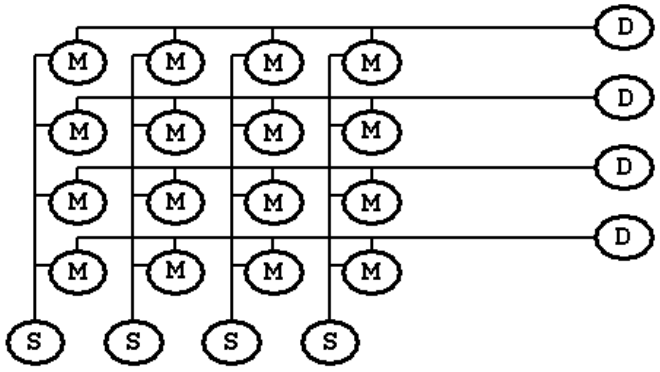

Figure 2.5: Dual-Port Memory Interconnection Network

non-blocking[1] interconnection between all inputs and outputs. It is an ideal interconnection network in terms of bandwidth efficiency. Crossbar switches, used in packet-switching networks, can be classified by the location of buffering (input, output, embedded) with respect to the switching matrix. If the buffers are moved to the inputs or outputs (figure 2.6) the switching matrix itself can be confined to a very small area. As an added advantage, only $2N$ large dual-ported buffers are required if the buffers are positionated at the inputs and outputs, whereas N^2 smaller buffers are required if they are embedded in the switching matrix. The total amount of memory required is the same regardless of where it is positioned, but as a practical matter it is easier and less expensive to implement a small number of large dual-ported buffers compared to a large number of small dual-ported buffers.

The full crossbar requires N^2 crosspoints, which may be impractical for larger systems. For systems with twenty or more data channels, a

[1]A network can be either *blocking* or *non-blocking*. Blocking occurs when information cannot be transmitted through the network due to the competition for the same internal or external data-path. When this feature can be avoided, the network is called non-blocking

multistage network can provide essentially the same non-blocking characteristics as the crossbar switch, using fewer crosspoints.

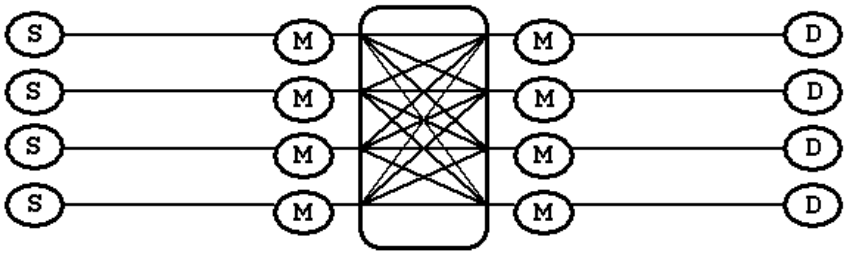

Figure 2.6: I/O Buffered Crossbar Interconnection Network

2.1.5 Integrated Processor Interconnection Network

The mesh interconnection network (figure 2.7) is popular in the construction of large multiprocessor systems. These INs are formed by overlaying an array of processors on the dual-ported memory array of the buffered crossbar switch. Some cost reduction may be possible with this approach. The mesh also allows direct processor to processor communication, not normally a requirement in event building but potentially useful in the analysis of overlapping events or methods of event building which divide the analysis software into stages, where each stage resides in a different processor. Reliability can be higher for a mesh interconnect since there are multiple paths for each packet transfer. In practice though, the control complexity and possibility of message deadlock allows only orthogonal routing. Otherwise a packet may inadvertently be routed into a circular path and lost or delayed. Intelligent buffered routers are necessary for event builders applications because there is nearly continuous traffic on all links in the network. If the processor managed the inter-node communication directly, there would be little time left for processing the data.

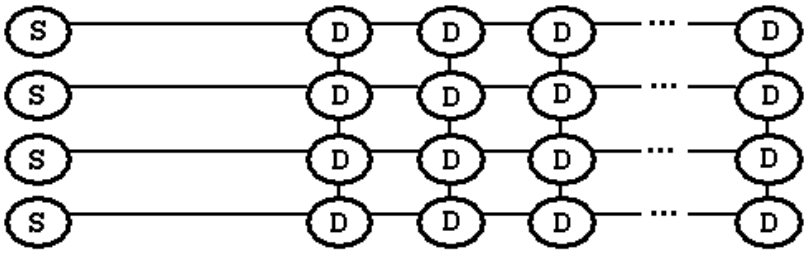

Figure 2.7: Mesh Interconnection Network

2.2 Event Building and Switching Technologies

Figure 2.8 shows the logical model of 3×3 event builder. By generalising the figure to a $N \times N$ event builder, it can be seen that each N sources contains N queues, one per destination, containing the fragments of those events that have been assigned to that destination. Each destination contains N receive buffers, one receive buffer per source. Each destination receives its event fragments over N logical connections between its N receive buffers and their corresponding queue in each of the N sources. Thus, a total of N^2 logical connections are required by the event builder, and at each source and destination module N different connections have to share a single interface and link to the network.

Each queue in the source can be considered as an independent "user" that transports data to another user (a destination receiver buffer) via a logical connection over a shared medium (the link) and switch. Three different transport modes can be applied to the links, namely *synchronous Time Division Multiplexing* (*TDM*), *asynchronous TDM* and *dedicated point-to-point connections* (i.e. no TDM). When synchronous or asynchronous TDM is used, the corresponding transport modes are known as Synchronous Transport Mode (STM) and Asynchronous Transport Mode

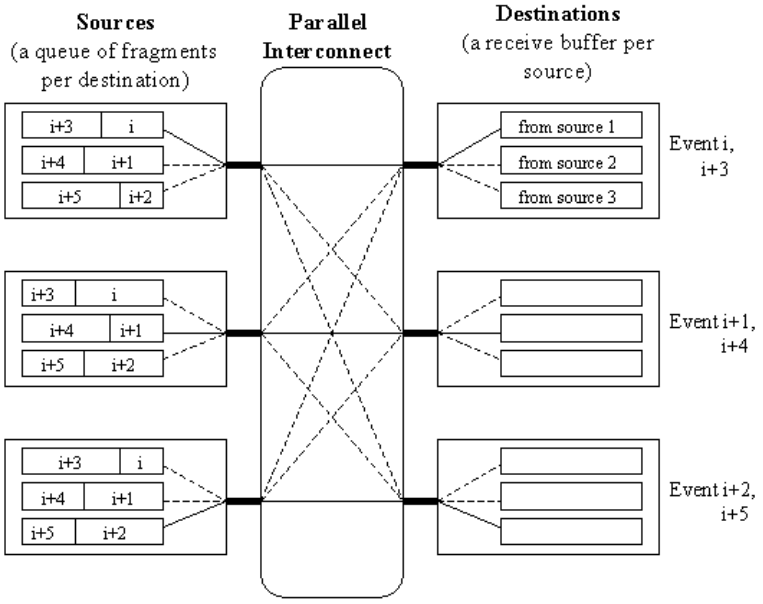

Figure 2.8: Logical model of an event builder

(ATM) [2] respectively. In the STM the data are transported on the link inside a fixed length *frame* which repeats indefinitely. The frame is divided into equal length time slots and each logical connection is allocated a fixed time slot in which it can transport data. Such technology was developed for telephony applications, where a constant bit-rate service is suitable to transport voice samples between subscribers at a fixed rate.

In the ATM the data are segmented into *packets*, which may be fixed or variable length, and the source sends packets over the link whenever it has data to transmit and can accesses the link. The identity of the logical connection to which the packet belongs has to be carried with it

[2]The use of ATM to describe a generic transport mode should not be confused with the specific implementation of this transport mode in a standard technology, which is also called ATM, using fixed-length 53-Byte cells.

in the packet header, either in the form of an explicit *virtual connection identifier* (VCI) or as a pair of source and destination addresses. The ATM is well suited to handling variable bit-rate services for applications where the required bandwidth fluctuates in time.

Dedicated point-to-point connections are used in applications where concurrent transport data between multiple different "users" is not required and the source and destination can sustain data IO at the full bandwidth of the link.

Whichever of the three transport modes is used the physical bandwidth of the shared-medium limits the maximum throughput and number of "users". Multiple shared-media segments or links can be interconnected by a switch to allow communication between larger number of users. To each of the three transport modes described above there corresponds a specific class of switching technology: respectively *Circuit Switching*, *Packet Switching* and *Connection Switching*. These classes of switching technologies have quite different characteristic and application areas. Table 2.1 summarise the transport modes and classes of switching.

Transport Mode	Switching Class	Characteristic
STM (Synchronous TDM)	Circuit Switching	"Telephone" switching technology Constant bit-rate services Equal bandwidth per circuit Concurrently active circuits
ATM (Asynchronous TDM)	Packet switching	Data and multi-media switching Burst traffic Bandwidth allocated per connections Concurrently active connections
Dedicated links	Connection Switching	Switching streams point-to-point Connection set up overheads Efficient for long block transfer Sequentially active connections

Table 2.1: Detector partitioning

2.2.1 Circuit-switched Event Builders

The first proposed parallel event builders were based on the same principles as circuit switching. Figure 2.9 shows how a global synchronous

control scheme is used to define time slots, in each of which circuits are set up between a different queue in each source and its corresponding receive buffer in a destination.

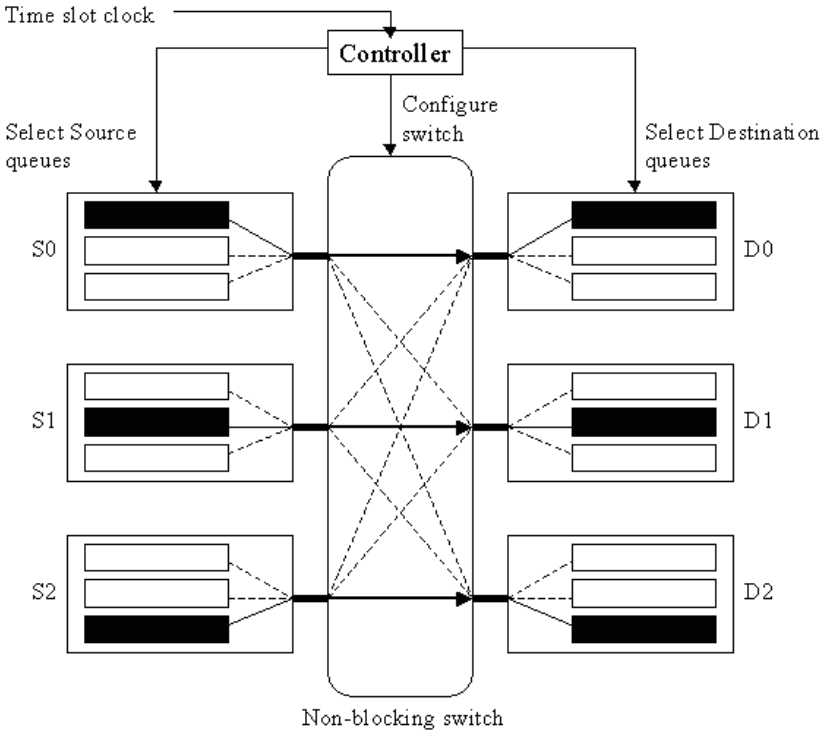

Figure 2.9: A circuit switched "barrel shifter" event builder

At each time slot the global controller enables the appropriate queues and receive buffers and configures the switch to route the event fragment data between them. The controller enables N circuits per slot, each one necessarily serving a different destination because only a one circuit can occupy a given slot on an output link. After N successive slots (equivalent to a frame length) all circuits will have been activated. If the slot length is chosen to be much more than the average event fragment transmission time, the scheme affectively supports the concurrent operation of N^2 circuits, each offering $1/N$ of the link bandwidth. This scheme is often

called "barrel shifter".

If the slot length is chosen to be greater than the transmit time of the largest possible event fragment, N complete events are built every N slots. However, the typical large dispersion in the length of event fragments means that aggregate bandwidth utilisation is low because the building of a new event cannot be started until the destination has completely received all the fragments. This inefficient use of available bandwidth can be eliminated by allowing all source queues to start sending their fragment belonging to the next event in the queue before the destination has completely received all the fragments of the current event. In this case, aggregate bandwidth efficiency is optimal, but each destination has to support the building of several events simultaneously.

A necessary condition for the circuit switched barrel shifter is that circuits can always be established through the switch without blocking each other. This can be guaranteed by using a non-blocking cross-bar switch. Unfortunately, large cross-bars of the dimensions needed for LHC event builders do not exist. However, there exist certain multi-stage network topologies that, although they are not non-blocking for all possible connection patterns, do not block under particular sequences of permutations of circuits that are sufficient to perform event building [27].

2.2.2 Packet-switched Event Builders

Packet switching is more suitable for LAN traffic or multimedia applications, where the average used bandwidth of user-to-user communications may vary widely depending on the application (for example, communication versus video communication) and, for a given communication, may also fluctuate rapidly in time (for example compressed video).

The packets are routed through the multi-stage network using a label or address in the packet header. Packet switching networks use a distributed routing control paradigm in which, at each stage, look up tables map packet header information into an output port number to which the packet must be forwarded. Once these network tables have been initialised with the information that maps routing label values or addresses into the desired destination port, we have effectively set up a *virtual con-*

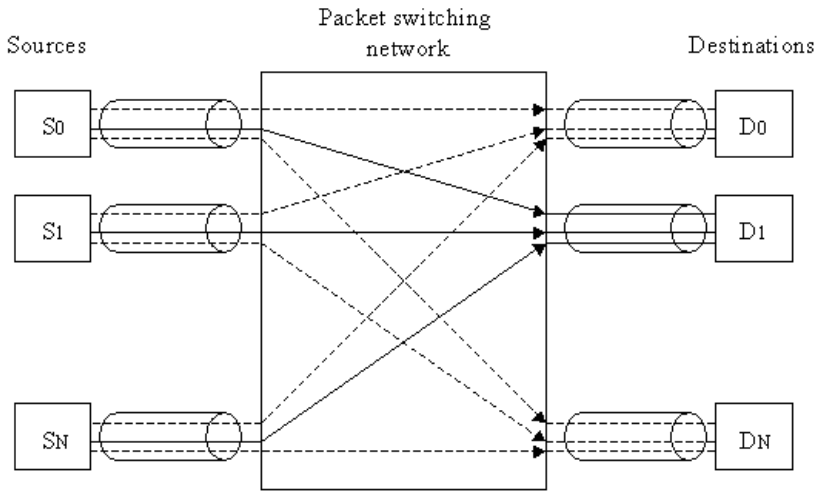

Figure 2.10: A packet switching event builder with permanent virtual connections fully interconnecting all sources and destinations

nection (VC). The connection is virtual because it does not reserve for itself all the resources along the path between source and destination. VCs can either be established for just as long as they are necessary to send a message (switched VCs) or they can be left in place indefinitely and only used on demand (permanent VCs). When there is no traffic flowing on a VC the network resources are available for use by traffic flowing on other VCs.

In principle the number of permanent VCs that can be set up through a network is only limited by the size of the mapping tables. Therefore a N $\times$ N event builder can be implemented with a packet switching network by opening all the N^2 VCs needed between the source queues and destination receive buffers as shown in figure 2.10.

Figure 2.11 shows how, when a connection switching technology is used such as fibre channel, each source establishes a connection to the

required destination receive buffer by reserving all resources along the path. It then transmits its entire event fragment from the appropriate queue and drops the connection. The source then moves on to servicing a different queue, which requires it to disconnect from the current destination and establish a new connection to a different destination. Each request for a new connection must pass through arbitration in order to resolve contention with requests from other sources. The connection set up and drop overhead can be quite significant in some technologies.

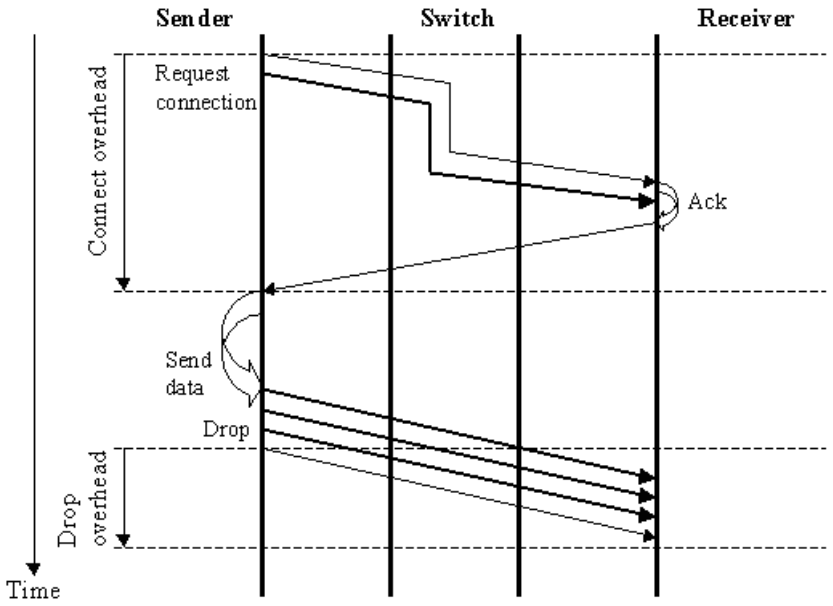

Figure 2.11: Connect overheads in connection switching

2.3 Data Flow Options

As already foreseen in the previous chapter, the choice of trigger algorithms and strategies defines the data flow of the event building process.

The data flow is controlled by a protocol operating between the readout buffers (*sources*) and the processors (*destinations*).

Two different data flow control models for event building can be distinguished by whether the responsibility for initiating the transfer of event fragments is given to the sources (*push architecture*) or to the assigned destination (*pull architecture*).

In the first case the event number and the destination identifier are broadcasted to the sources, which then transmit their event fragment data to the given destination.Event builders operated in push mode are characterised by many-to-one traffic patterns and by potential contention between the many independent traffic streams. Contention occurs either for access to the output port (*output blocking*) or for a common internal path within the switching network. If the bandwidth of a common internal path is insufficient to handle the traffic the switch becomes *congested*. Output blocking and congestion have an adverse effect on the event builder's throughput and the latency of event building.

Switching technologies developed for different application domains adopt different strategies to handle contention. In the event building application the consequences of the contention handling strategy are inefficient use of the switch's bandwidth, increased event building latency, loss of data and poor scaling of performance as a function of the size of the system. Event builder performance can be optimised by using a *traffic shaping* technique to coordinate the traffic generated by the sources in order to minimise output blocking and internal congestion. In push mode event fragments are received at the destination out of order. Because some sources may be empty or have broken down, a mechanism is required to decide when event building is completed. An event building protocol must provide functions such as completion detection, linking fragments, etc.; this introduces software overhead.

On the other hand, an event builder operated in pull mode can use the intelligence of the destination processor to select and readout subset of the sources. The traffic patterns characterising this strategy are few-to-one, or even one-to-one, hence the negative impact of contention is much reduced compared to the push case. This approach allows early rejection of an event based on a minimum of data, and only pulls in

more data for smaller number number of events that are not rejected. By applying a hierarchy of data pulls and event reject/accept decision, the overall required bandwidth of the event builder can be reduced to a minimum.

2.4 Readout Protocols

Two different protocols have been studied for managing the collection of event data [24].

The first one, called the *full-readout protocol*, requires that data, assembled in the readout buffers, are immediately transmitted and dispatched through the readout network, such that a complete event is assembled in a destination processor. In this protocol, the buffering capacities of the destinations must be designed such that in normal operation there is always sufficient space available to receive data from readout buffers. Obviously the system must also be designed to cope with bursts of data. For this a throttle signal is sent by the readout buffers anticipating the buffer overflow, and is used to reduce the trigger rate such that the buffer occupancy can be reduced and new fragments can be accepted. The latency of the throttle signal translates into buffering requirements at the different levels of readout. The destinations collects all event fragments, and once the event is complete passes it to a processor running the high level trigger algorithms. Since all data belonging to a trigger is immediately available to th high level trigger algorithms, there is no real need in this schema to distinguish between Level-2 and Level-3 . Thus the high level trigger algorithms can evolve with complete freedom.

In the second approach, called the *phased-readout protocol*, the transfer of data from the readout buffer to the high level trigger processors takes place in two or more phases [3]. In the first phase, the subset of the event data that is needed by Level-2 algorithm is transferred form the appropriate readout buffers to a destination processor. The data from

[3]More than two phases could be implemented, but this would not improve the system performance. A large number of phases would eventually result in a "data on demand" scheme

the remaining readout buffers must be buffered for the duration of Level-2 latency. The Level-2 decision must be transferred to all the readout buffers. On reception of a Level-2 "NO" decision the data are discarded, otherwise, on reception of a Level-2 "YES" decision the data must be sent to the processor that ran the Level-2 algorithm to execute further filtering algorithms (Level-3) on the complete event. As in the case of the full-readout protocol, the readout buffers will push the data to the destination processors in each phase, so again a throttle mechanism is required to prevent buffer overflows. The reduction factor of the bandwidth required of the readout network depends on the rejection power of the Level-2 algorithm and the fraction of the complete event needed to execute the algorithm. Moreover the phased readout protocol requires a new functional element called *event manager*. Its function is to collect the Level-2 decisions and distribute the to the appropriate readout buffers. Nevertheless the presence of an event manager would permit the dynamic load balancing across the processors of the high level triggers.

The full-readout protocol requires a larger scale readout network, but the complexity of the source modules is less than in the phased-readout case. From the point of view of simplicity of the protocols, and the flexibility it allows for the high level trigger algorithms, the full-readout protocol is preferable.

2.5 Problem of Effective Throughput

Data transfer rates for networks are measured in terms of throughput, which represents the amount of data transferred from one place to another or processed in a specific amount of time. Each typology of network has its specific, well-known throughput, which is defined by the standard or the network vendor and coincides with the full use of the network bandwidth. Unfortunately this "official" maximum throughput is never reached in real life and we usually speak of effective throughput, which is the effective use of network bandwidth.

The problem of limited throughput is an important and complex question, in which a lot of factors play different roles. First of all, the most

important point to be considered is the fact that there are two big distinct factors which determine the performance of a switching network and they are *the performance of point to point links* and *the performance of the switching network*. They are quite different problems that involve further considerations and so we are going to discuss them separately in the remainder of this section.

2.5.1 Effective Throughput of Point to Point Links

In the case of point to point links, our attention is concentrated only on the network access point (not really on the pure network) and throughput here is limited by several kind of overheads, which can be be grouped in two main sets:

- *Technology overheads* (figure 2.12):
  Delays, bottlenecks in network adapter...
  Contention for bus control or connection setup;
  Control data (headers, frame control);
  "Packetisation" (partial use of fixed length packets);
  Flow control...

- *Software overheads* (figure 2.13):
  Operating system (interrupts, context switching...);
  Use of additional communication protocol (like TCP/IP);
  User application...

Effective throughput results from the combination of both classes of overheads.

Some of these factors are unavoidable but some others can be reduced. An inevitable overhead, for example, is what we call the "packetisation" problem. This fact refers to the partial use of fixed length packets. If the network protocol requires a standard length for user data, like Ethernet, for instance, it can happen that not all packets sent through the network will be completely filled with useful data. In fact if user data size does not fit with the standard data length or its multiple, this packet field will be padded, when necessary, with useless data, with a consequent waste

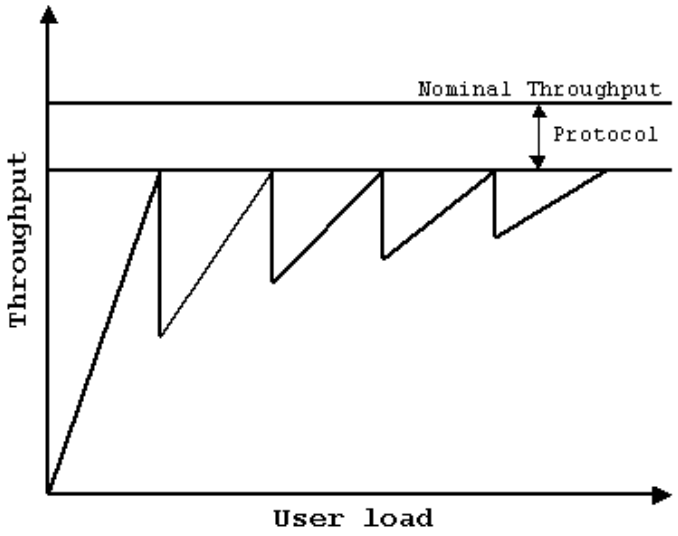

Figure 2.12: Technology overheads

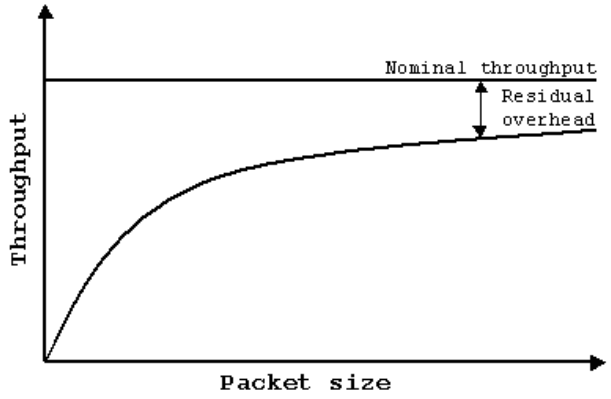

Figure 2.13: Software overheads

of bandwidth (figure 2.14). Of course, a variable and boundless data size would be better from the throughput point of view, but network protocols usually need a standard length and there is no possibility to avoid standard requirements.

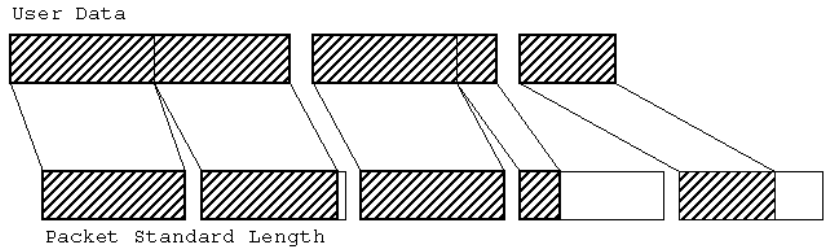

Figure 2.14: Example of "packetisation" overheads

On the other hand, in order to improve the performance of point to point links, some overheads could be spared. A big improvement, for example, could be the pipe-line of data submission and data preparation. To reduce (or eliminate) the dead-time between the transmission of two packets, data that have to be sent can be prepared in the right network protocol format while ready data are transmitted. So, instead of having the bad situation described in figure 2.15 where data preparation and submission are not pipe-lined, the solution depicted in figure 2.16 could be implemented, resulting in no loss of time.

In this way the effective throughput can be improved by a reasonable factor, as figure 2.17 shows.

2.5.2 Effective throughput of Network

In the case of switching networks, throughput is limited by several factors, like:

- Technology

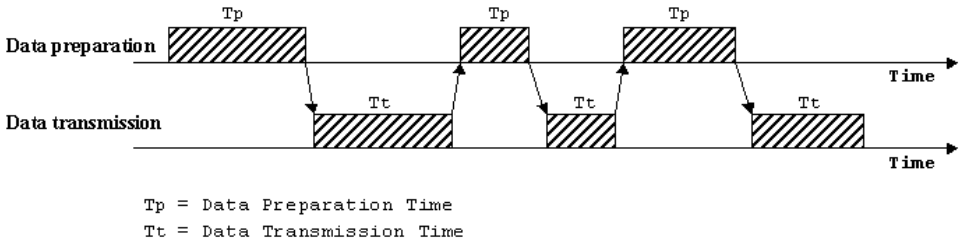

Figure 2.15: Bad solution between data submission and data preparation

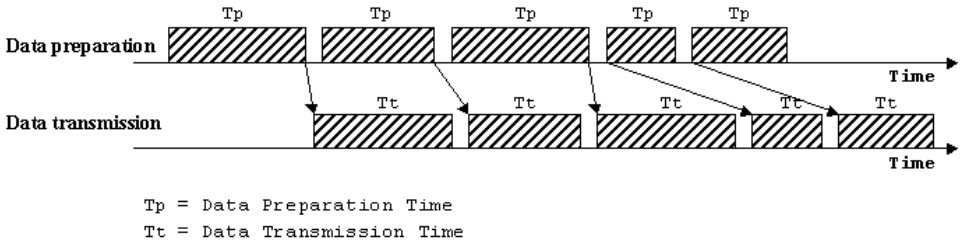

Figure 2.16: Pipeline of data submission and data preparation

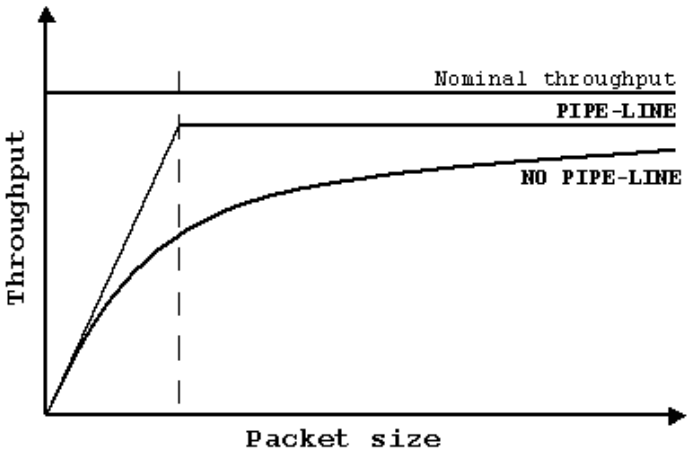

Figure 2.17: Different throughput between pipe-lined and no pipe-line data submission and data preparation

- Switch architecture
- Type of traffic: random versus coherent...

The network *congestion* problem and the different techniques to avoid it play the main role in this performance study. Congestion happens when two or more data transfers compete for the same network source and there is no any mechanism to resolve this contention. In this case the throughput can be degraded and, in the worst case, data can be lost.

Two classes of technologies have been applied to prevent or to avoid congestion, namely *flow control* and *traffic shaping*. In the flow control technique, the network provides the mechanism to avoid congestion. The source sends data packets to the various destinations as soon as they are submitted and performs the operations required by the data flow mechanism. This can be:

- *Back pressure*, implemented on networks with internal buffering (that can store and forward);

- *Collision detection*, used on shared media (Ethernet);
- *Connection based transfer*, which consists of initiating a protocol to open an exclusive connection between source and destination, before sending any data.

Traffic shaping techniques, on the other hand, aim at subjecting the data transfer to a global traffic control thus minimising contention and congestion. There are two strategies:

- *Barrel shifter*, where at any one moment all sources are connected to destinations in a non blocking fashion and can send data using the full bandwidth. At regulars intervals the connections are changed (so after a full cycle (T) all sources have been connected to all destinations during a period of time T/N, being N the number of destinations);
- *Rate division and randomisation*, where the sources emit data in the form of small packets successively to all destinations, at an average rate per connection equal to the nominal rate divided by the number of sources.

It should be noted that all these protocols, which prevent congestion, imply lower bandwidth utilisation. Traffic shaping seems to preferable because it allows to reach high loads, whereas flow control can have several limitations depending on the network topology and on the type of traffic. At the same time, it should also be noted that, even though techniques to avoid congestion are implemented, there can be data losses. In this case a retransmission of same data is required and this implies again a throughput decrease, possibly even worse than without congestion prevention.

Chapter 3

Technology

A difficult issue in a physics experiment environment of the size of LHCb (and the other LHC experiments), is the choice of the appropriate technology for the event building readout network. The range of possibilities nowadays is very large, because networking industries have put big efforts in developing and improving different kinds of technologies and solutions, especially during the last ten years. An important decision is the selection between an industry standard network technology or a dedicated network system. This choice can be quite different for several reasons: each single system in fact has its own advantages and disadvantages. In the next section we will try to explain and summarise them, highlighting the different aspects. Further in the chapter, we will shortly illustrate the three technology competitors for being the final LHCb readout network technology and we will introduce the choice of this work: the Ethernet technology.

3.1 Industry Standards versus Dedicated Systems

The first and biggest problem for starting the building of a readout network is the decision whether to apply an industry standard network or a dedicated one. The two solutions show a lot of differences, which make

the choice difficult.

An industry standard system has its greatest advantage in the fact that it is well known and certificated, and so it can usually ensure interoperability between different components, wide support from networking industries and scalability for future system expansions. Of course it has also a lot of boundaries due to the standard definition and it cannot be changed to meet specific requirements and needs of a particular system, like, for example, the unusual data acquisition architecture of a physics experiment.

On the other hand, a dedicated system can be modified to fit in a proper way the characteristics of a specific architecture but, of course, it does not offer any guarantee about future reliability. This kind of system in fact is strictly bound to its single vendor and thus it cannot be considered completely reliable in a long time scale. If, for some reason, its vendor crashes, all its supports will suddenly disappear.

Event building requirements can be defined quite specific and restrict and so, at a first quick view, a dedicated system, more adaptable than a standard one, could be considered the right solution. Unfortunately, it has to be taken into account that experiment lifetime is long (10–15 years) and it is necessary to guarantee adequate performance, availability and interoperability of components over a long time period. The evolution and upgrade path of the system has also to be taken into account. All these considerations make the industry standard choice more probable.

Regarding the LHCb event building switching technologies, three different solutions have been considered and investigated until today. Two of them are standards, and they are *ATM* and *Ethernet*, whereas the third one, *Myrinet*, is a dedicated system.

The remainder of the section summaries the main characteristics of these technologies and gives some suggestions for the final choice.

3.1.1 ATM

ATM [29] is a standard that has been developed for telecommunications applications by the ITU (International Union of Telecommunications, formerly known as the CCITT) to form the basis for the future B-ISDN

(Broadband Integrated Services Digital Network). It is also actively promoted within the ATM Forum by the computer industry as a future standard technology for very high bandwidth LANs (Local Area Networks) and WANs (Wide Area Networks), which will support real-time multimedia and distributed computing applications. It is expected to have a long life cycle because of its origin and expected large scale deployment in the telecommunication filed.

The ATM technology has been designed to support a massive, low latency, non-blocking switching capacity that is suitable for carrying, on a common infrastructure, the traffic generated by a wide range of services, each with its own specific performance requirements. In order to achieve this flexibility it is based on the principle of packet-switching, in which information is carried between "subscribers" in packets (called "cells" in ATM jargon), each of which carries routing information in its header. The network is composed of one or more switching fabrics, each of which uses this routing information to forward the cells through the network towards their destination.

The ATM standard specifies the connection between an end-station and the network (UNI = User Network Interface) and between sub-networks (NNI = Network to Network Interface). For simplification, the standard is sub-divided into three layers:

- The Physical Layer
- The ATM Layer
- The ATM Adaptation Layer (AALn)

The *ATM Layer* is the core of the standard and it is common to all services. It defines that the information (voice, image or data) is to be transported by means of small, fixed length, cells containing 53 Bytes (48 Bytes of payload and 5 Bytes of header). The cell header includes 3 Bytes that carry a label identifying a connection between a particular source and a particular destination. This label is used by the switching fabric hardware to route the cell to its destination (*self-routing*).

Connections need not to be assigned with a constant available bandwidth, but instead average and peak characteristics of the connection's traffic can be declared and bandwidth can be used on demand.

The *Physical Layer* specifies how ATM cells are physically transmitted over a link. It consists of two sub-layers:

- The Transmission Convergence Sub-Layer, which defines the bit rates and the framing patterns (grouping of cells into larger packets).
- The Physical Medium Sub-Layer, which defines the physical support for bit transfer and the timings.

Numerous standards have been defined at this level, for different applications, by the ITU on one hand and by the ATM Forum on the other hand, with the aim of allowing for early, cost-effective deployment of the technology for computer networking using existing components.

The *Adaptation Layer* defines how to adapt the ATM layer to the requirements of specific services. Several adaptation layer standards exist for different applications. For data transmission the so-called AAL5 protocol is used. It specifies that data can be transferred in variable length blocks of up to 64 kByte.

3.1.2 Ethernet

Ethernet [33] was first conceived in 1973 by a young engineer named Bob Metcalfe, working at the Xerox Palo Alto Research Center, and it become a standard later on, with the explosive increase of communications technology due to the large use of information sharing applications. At the beginning it was thought as a distributed office computing system, a big system that would completely eliminate the need for paper in the modern office. Thus the first Ethernet architecture was rather simple, it was a sort of bus topology, where all computers on the network hooked to the same wire. It has later on been developed into a more complete network system that has known an enormous and especially large diffusion all

over the world due to its low cost and its good performance. Nowadays, for example, most of the Internet traffic runs on Ethernet networks.

The Ethernet system includes for building blocks that, when combined, make a working Ethernet:

- The frame, which is a standardised set of bits used to carry data over the system.
- The media access control protocol (MAC), which consists of a set of rules embedded in each Ethernet interface that allow multiple computers to access the shared Ethernet channel in a fair manner.
- The signalling components, which consists of standardised electronic devices that send and receive signals over an Ethernet channel.
- The physical medium, which consists of the cables and other hardware used to carry the digital Ethernet signals between computers attached to the network.

The building block of Ethernet data traffic is the *frame*. The network hardware—which is comprised of the Ethernet interfaces, media cables, etc.—exists simply to move Ethernet frames between computers, or stations. The size of an Ethernet frame is comprised between 64 and 1518 Bytes and the bits in it are formed up in specified fields. These fields are described in figure 3.1

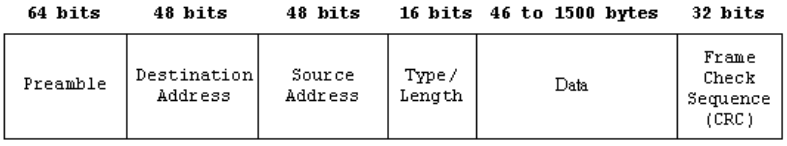

Figure 3.1: An Ethernet Frame

The half-duplex mode of operation described in the original Ethernet standard uses the *media access control protocol*, which is a set of rules used to arbitrate the access to the channel shared among a set of stations connected to that channel. The way the access protocol works is fairly simple: each Ethernet-equipped computer operates independently of all other stations on the network; there is no central controller.

Ethernet uses a broadcast delivery mechanism, in which each frame that is transmitted is heard by every station. It is up to the station to determine whether a message is aimed at itself, according to the frame headers information. In this way, before sending data, a station first listens to the channel, and if the channel is idle the station transmits its data in the form of an Ethernet frame. The message can be considered as successfully transmitted if the shared medium is not disturbed during transmission. If it is (collision of message) the frame must be re-transmitted.

This protocol, which is used by the Ethernet MAC is called *Carrier Sense Multiple Access with Collision detection* (CSMA/CD).

Finally, there are two basic groups of hardware components used in the system: the *signalling components*, used to send and receive signals over the physical medium, and the *media components*, used to build the physical medium that carries the Ethernet signals. These hardware components differ depending on the speed of the Ethernet system and the type of cabling used.

The original Ethernet standard of 1980 described a system that operated at 10 Mbps. This was quite fast at the time, but as computer technology continued to evolve, ordinary computers were fast enough to provide a major traffic load. Thus Ethernet was reinvented to increase its speed by a factor of ten. The new standard specified the 100 Mbps, *Fast Ethernet* system, which was formally adopted in 1995.

In 1998, Ethernet was reinvented yet again, this time to increase its speed by another factor of ten. The new *Gigabit Ethernet* standard describes a system that operates at the speed of 1 billion bits per second over fibre optic and twisted-pair media. The invention of Gigabit Ethernet makes it possible to provide very fast backbone networks as well as connections to high-performance servers.

In this work we will concentrate on this kind of technology to propose a new event building architecture based on a Gigabit Ethernet readout network. Therefore the next pages are dedicated to the description of this new standard.

Gigabit Ethernet

"Gigabit operation represents an *evolution* rather then an revolution, in Ethernet technology." [31] It is a Data Link and Physical Layer technology, as such it requires no changes to higher-layer protocols or applications. For these reasons we will briefly concentrate on the relevant aspects which characterise the Gigabit Ethernet technology.

Media Access Layer
The Gigabit Ethernet MAC is a proper superset of the 10/100 Mbit/s Ethernet, in the sense that it contains all of the capabilities that already exist and adds some additional functions and features specific to the gigabit operation.

The Gigabit Ethernet MAC can operate in either full-duplex or half duplex mode. Full-duplex operation is unchanged from other standards. Half-duplex operation at gigabit rates is problematic.

- Half-duplex mode

  The use of CSMA/CD as an access control mechanism implies a strict relationship between the minimum length of a frame transmission and the maximum round-trip propagation delay of the network. The minimum transmission time must be longer than the maximum round-trip propagation time of the LAN so that a station will still be transmitting a frame when it is informed of a collision. Since the length of time required to transmit a frame scales inversely with the data rate, increasing the speed by an order of magnitude (from Fast Ethernet's 100 Mbit/s) reduces the frame transmission time by that same factor 10. To accommodate this scaling, it is necessary either to reduce the size (extent) of the network, or to increase the minimum frame, or to change the MAC algorithm.

If there were no other modifications, the network extent would have to have been reduced to the order of $10-20$ meters to support half-duplex Gigabit Ethernet operation. This is clearly inadequate for most practical uses. So the approach was to increase the minimum frame transmission time and modify MAC algorithm itself.

$\Rightarrow$ A *carrier extension* was added to overcome this inherent limitation of the CSMA/CD algorithm and it appends a set of special symbols to the end of short frames so that the resulting transmission is at least 4096 bit-times in duration (4.096 microseconds) up to the minimum bit time imposed at 10 and 100 Mbps.

$\Rightarrow$ An optional *frame bursting* feature was defined to improve the throughput of gigabit CSMA/CD LANs. Frame bursting allows multiple short packets to be transmitted consecutively, up to a limit, without relinquishing control of the signalling channel and reverting to the CSMA/CD protocol between packets. Thus, it offers a way to avoid the overhead associated with the carrier extension technique for all but the first packet of a burst.

- Full-duplex mode
  In full-duplex mode, the CSMA/CD algorithm (including the carrier-extension and frame-bursting) is disabled. This means that data transmission and reception can occur simultaneously without interference, and with no contention for a shared medium. Easier to implement than the half-duplex (CSMA/CD) mode, the full duplex mode also provides higher bandwidth. Thus, all currently shipping Gigabit Ethernet equipment operate in full duplex.

The full-duplex Ethernet standard further supports dedicated channels and full-duplex operation by introducing link-level *flow control* which keeps switches from loosing frames due to buffer overflow. A *pause protocol* provides a mechanism where a congested receiver can ask the transmitter to inhibit (pause) its transmissions. The protocol is based on the transmission of a short packet known as a *pause frame*. The frame contains a timer value, expressed as a multiple of 512 bit-times, that specifies how long the transmitter should remain quiet. If the receiver becomes

uncongested before the transmitter's pause timer expires, the receiver may elect to send another pause frame to the transmitter with a timer value of zero, allowing the transmitter to resume immediately. Thus, the pause protocol is like an "XON/XOFF" flow control packet. The protocol operates only between the two endpoints of a point to point link. It is not an end to end protocol and cannot be forwarded through bridges, switches or routers.

Gigabit Ethernet builds on the pause protocol by introducing asymmetric flow control. It uses the Auto-Negotiation protocol (described later in the section "Physical Layer"). This protocol lets a device indicate that it intends to send pause frames to its link partner, but declines to respond to them. If the link partner is willing to cooperate, then pause frames will flow in only one direction on the link.

Physical Layer
The PHY accepts the data stream from the MAC and converts it into optical or electrical impulses for transmission across a given medium. Within the Gigabit Ethernet standard, the PHY functions are subdivided into a group of logic functions known as the Physical Coding Sublayer and a group of analog-digital mixed signal functions referred to as the Physical Medium Attachment sublayer. These groups are not described in this document, because they are not really relevant for the purpose of the work. Further information can be found in [31].

Gigabit PHY performs a link configuration protocol referred to as *Auto-Negotiation*. It is an automatic mechanism to ensure that certain link characteristics are properly configured when links are initialised. Auto-Negotiation was defined in Fast Ethernet in order to automatically select operation speeds between 10 and 100 Mbps. It was adapted to Gigabit Ethernet primarily to select between duplex mode and the use of link-level flow control.

Jumbo Frame
The Gigabit Ethernet standard uses the same variable-length (64- up to 1518-Byte packets) 802.3 frame format found in Ethernet and Fast Ethernet (see figure 3.1). Because the frame format and size are the same

for all Ethernet technologies, no other network changes are necessary.

Anyway, in order to overcome the performance bottlenecks of the frame size limitation, it has been proposed (and implemented) to increase the maximum valid frame size beyond 1518 Bytes. The reason for larger frames, called *Jumbo frames* is simple: they mean lower frame rates (especially for applications that transmit large chunks of data). They extend the Ethernet standard frame size to 9000 Bytes. The size of 9000 Bytes have been chosen first because Ethernet uses a 32 bit CRC that loses its effectiveness above about 12000 bytes, and secondly, 9000 was large enough to carry an 8 KByte application datagram plus packet header overhead. In the case of Gigabit Ethernet, wire-speed throughput at 1518-byte frames mean servers face a torrent of more than $8 \cdot 10^4$ packets—and $8 \cdot 10^4$ interrupts—per second. That is enough to bring many multiprocessor platforms to their knees. Jumbo frames, in contrast, reduce the rate by more than 80%. However, they are not yet part of the Gigabit Ethernet standard but strong requests to officially include them are coming up, especially from networking industries.

3.1.3 Myrinet

Myrinet, developed by Myricom [28], is a switched, Gigabit-per-second local-area network that uses variable-length packets. The packets are wormhole-routed through a network of highly reliable links and crossbar switches. Myrinet technology is widely used to interconnect *clusters* of workstations, PCs, or single-board computers. Clusters provide an economical way of achieving:

- *high performance*, by distributing demanding computations across an array of cost-effective hosts. For "tightly coupled" distributed computations, the interconnect must provide high data rate and low latency communication between host processes.

- *high availability*, by allowing a computation to proceed with a subset of the host. The interconnect should be capable of detecting and isolating faults and using alternative communication paths.

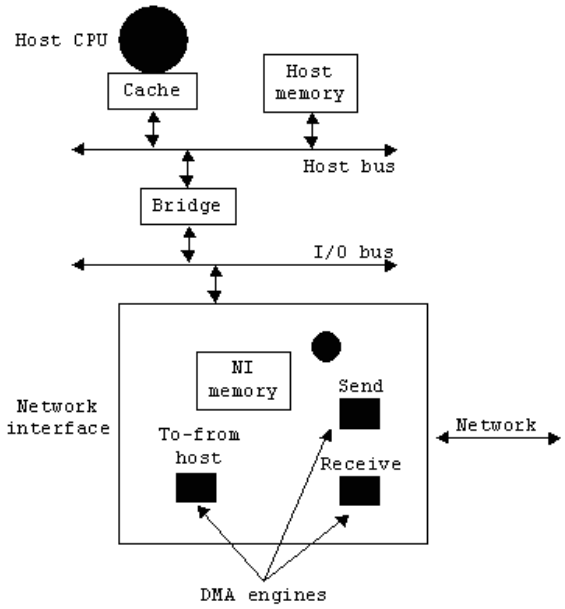

Figure 3.2: Host and network interface architecture of Myrinet

Figure 3.2 illustrates the architecture of a node in a Myrinet cluster. Each machine (host) has a network interface card that contains a processor and some memory, which is used to store the interface control program and data. The Network interface connects to the host I/O bus—a typical organisation for commodity hardware.

Myrinet requires that all packets be staged through network interface memory, both at the sending and the receiving side. It uses fast SRAM, hence the memory is relatively small. The packets may be of any length, and thus can encapsulate other types of packets, including IP packets, without an adaptation layer. Each packet is identified by a type, so that a Myrinet network , like an Ethernet network, can carry packets of many types or protocol concurrently. Both the host and the network interface can use DMA to access data in each other's memory, but DMA transfers

suffer from a startup overhead. The host can also access the network interface memory using programmed I/O, which has no startup costs, but results in high access times relative to host memory.

3.2 Summary

Table 3.1 summarises the main characteristics of the technologies discussed in the previous subsections. All the technologies considered provide bi-directional connections to the network (full-duplex). In the event building application, as we have already noticed, each port connected to the network uses only one transfer direction for high data transfer, thus the other direction can be devoted to control messages.

The three technologies which have been investigated present different advantages and disadvantages. Any of them perfectly fits some of the event building requirements while failing others. These considerations make it difficult to see clearly one's way out of this tough choice.

Myrinet [17] and ATM [10] [15] have been already analysed thoroughly in all details and they still do not seem to be the best solution for the LHCb event building, even if they have been proposed, as possible technologies, also for the other LHC experiments and, in particular, Myrinet has raised with a lot of interested in the CMS experiment as a solution to its specific event building requirements.

The most attractive technology for the LHCb event building problem is at the moment Ethernet, especially the new Gigabit Ethernet standard. It represents the last innovation in the networking field but it has an available tradition, due to the large use of Ethernet and its standards in the past. It also assures system operability and availability over a long time-scale. Before the advent of the new Gigabit standard, Ethernet technology was never taken into account for the event building purpose, because the previous network architectures (Ethernet and Fast Ethernet) were not able to support the high data rate requirements of the LHCb DAQ. Since Gigabit Ehternet has been implemented, also Ethernet could be considered as a technology competitor for the event building system. Anyway, it should be noted that, even if this network technology is largely

	ATM	Ethernet	Myrinet
Application	Telecom, WAN/LAN	LAN	Computer Clusters
Standardisation	ITU, ATM Forum	IEEE	Vendor specifications
Industry acceptance	Medium	High	Single Vendor
Technology	Connection oriented	Connectionless	Connectionless
Flow Control	Traffic shaping	Collision detection & retransmission	Back pressure
Physical layer	Twisted Pair, Fibre Optic	Twisted Pair, Fibre Optic	Float cables
Packet size	up to 64 k	up to 1518 Bytes	Variable (Max 1 MByte)
Overhead	11%	14 Byte header + 4 Byte CRC	Few Bytes per packet
Link Bandwidth	155, 622 Mbit/sec	1000 Mbit/sec	1.28 Gbit/sec

Table 3.1: Network Technologies

used and has demonstrated its reliability during its history and development, it has always been used together with either the UDP/IP protocol or better the TCP/IP protocol on the higher levels. These protocols does not really fit the LHCb event building requirements. Their redundancy, especially in the TCP/IP case, slows down the nominal network performance and most of their functions are non-essential for the event building needs. For these reasons, in the LHCB DAQ project those protocols could not be used and only raw Gigabit Ethernet is taken into account at present. Therefore there are no guarantees that this technology will give the same performance and the same reliability, Demonstrated in conjunction with either UDP/IP or TCP/IP protocol.

The aim of this work is to study in depth the LHCb event building

problem and propose a feasible solution, using the new Gigabit Ethernet standard as base technology. Our goal is investigating new event building possibilities with raw Gigabit Ethernet and implementing a proper hardware and software solution to the problem, which could be a real candidate for the final choice.

Chapter 4

Case Study: LHCb Event Building

The LHCb experiment [21] is the most recently approved of the four experiments under construction at CERN's LHC accelerator. It is a special purpose experiment designed to precisely study the CP violation parameters in B-meson decays by detecting many final states. The LHCb detector is a forward single dipole spectrometer, consisting of a micro-vertex detector, a tracking system, aerogel and gas RICH detectors, electromagnetic and hadron calorimeters, and a muon detector. The layout of the experiment is shown in figure 4.1

The expected b-quark production cross-section of 500 μbarn, at the LHCb working luminosity of $1.5 \cdot 10^{32} cm^{-2} s^{-1}$, leads to a rate of about 75 kHz of B-meson events. This is embedded in a total inelastic interaction rate of some 15 MHz. Typical branching ratios for the interesting final states of B-meson events lie between 10^{-5} and 10^{-4} leading to a rate of interesting events of $\sim$ 5 Hz. For rare decay modes the branching ratios are as low as 10^{-9}.

Thus triggering encounters special problems, since the B-meson events of interest are a small fraction of all the events involving B-mesons. Minimum bias events also offer a severe background.

The role of the DAQ system in the LHCb experiment is to collect the data, zero-suppressed in the Front-end electronics, and assemble complete

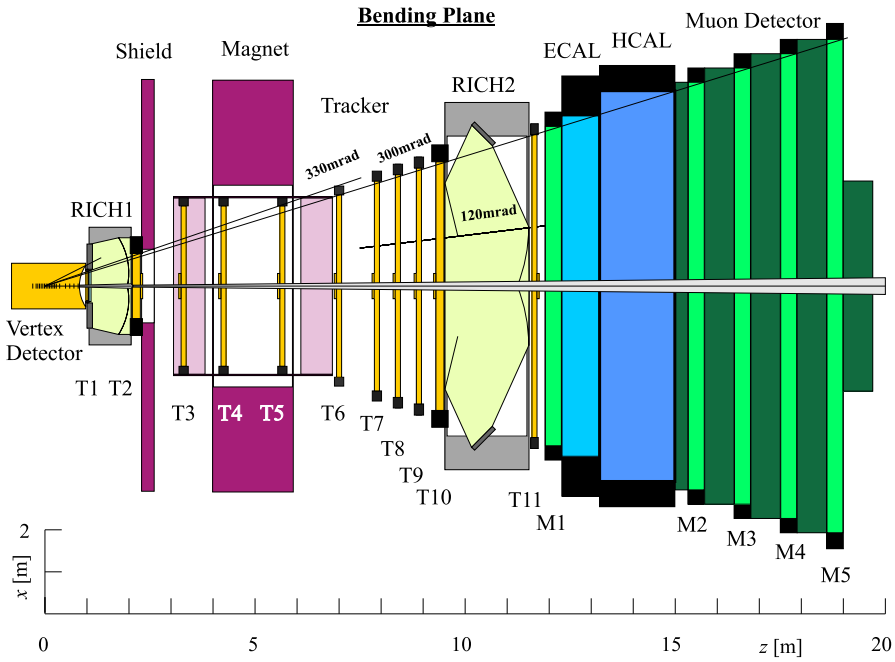

Figure 4.1: The LHCb detector

events in CPUs for further data-reduction by the Level-2 and Level-2 triggers.

4.1 LHCb DAQ Architecture and Requirements

Figure 1.2 shows systematically the overall architecture of the LHCb trigger and DAQ system. The main functional components are:

- Timing and Fast Control [23], to distribute a common clock synchronous to the accelerator and the the Level-0 and Level-1 decisions to all components needing this information, such as Front-end electronics, Trigger, etc.

- Two levels of "hardware" triggers: Level-0 and Level-1.
- The Front-end electronics where data are buffered during the latencies of the hardware triggers and subsequently processed (zero-suppression, formatting, etc.) and multiplexed before being passed to the DAQ system.

This section presents a generic DAQ Implementation Model based on the LHCb requirements specified in the LHCb Technical Proposal [21] and in the User Requirements Document [20]. The Timing and Fast Control Systems and the low-level triggers (Level-0 and Level-1) are not described in this document because they are not part of the work. We concentrate ourselves in this work on the DAQ system, which is composed of:

- The Readout Units (RUs), acting as a multiplexer of Front-end links and as an interface to the readout network.
- The readout network which provides support for event building, i.e. assembling all event fragments buffered in the Readout Units.
- Sub-Farm Controllers (SFC) which act as an interface between the readout network and the processor farm, that will run the higher-level triggers (Level-2 and Level-3).
- CPU farm to execute the higher level trigger algorithms (Level-2 and Level-3).

In the following subsections the whole model and all the components of the LHCb DAQ system are described in more detail.

4.1.1 DAQ Implementation and Functional Model

A *Readout Unit* (*RU*) performs the following functions:

- *Multiplexing of Front-end Links* (*Sub-Event Building*):
  A source collects data from one or more Front-end (FE) links (n_i

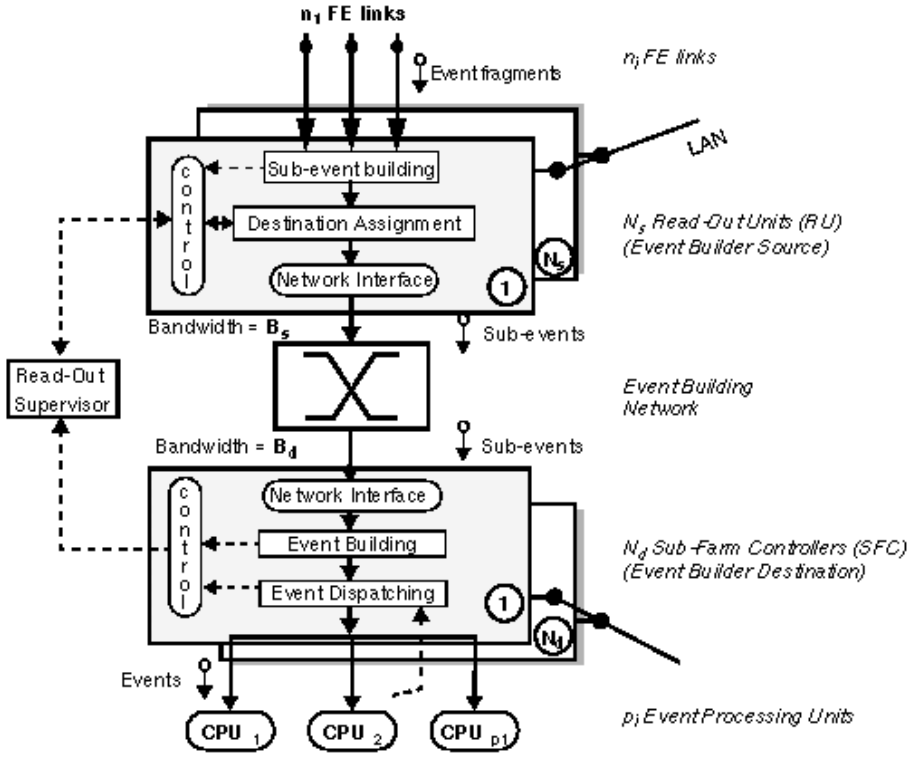

Figure 4.2: Functional Model of the event builder

for the source i) the sum over n_i is $\sim$ 160. The event fragments collected from each FE link are assembled into a *sub-event*. The multiplexing factor n_i depends on the FE link data rate and on the network link bandwidth (B_s) of the source, the aim being to achieve a load as close as possible to $\sim$ 50% for the working point conditions. Higher bandwidths lead to higher multiplexing factors n.

A constraint resulting from the partitioning requirement forbids links from different sub-detectors to be mixed.

If needed, the multiplexing function can also include the packing of

sub-events in order to reduce their frequency in the network.

- *Destination assignment*:
  Can be determined locally (look-up table) or by a dedicated controller that collects information on the state of the destination modules and the attached CPU's, in order to balance the load.

- *Sub-Event Buffering*:
  If phased event building is implemented, the source modules implement a buffering scheme where the Level-3 data are stored until a Level-2 decision is made. The additional buffer space required is modest (1–2 MBytes). This implies a more complicated management than the simple push protocol.

A *Sub-Farm Controller* (*SFC*) performs the following functions:

- *Event building*:
  Apart from the obvious task of linking sub-events to a full event, this function must implement the event building completion algorithm that takes into account possible sub-event losses.

- *Event dispatching*:
  In this model the destination module performs the dispatching function, the processors attached to a destination being "invisible" from the source. The advantage is that the dispatching function can include a local load balancing function and issue an overflow warning only when all the local resources become scarce.

- *Gateway between the Readout Network and the Sub-Farm of processors*:
  The number of processors depends on the network link bandwidth of the destination (B_d): the higher the bandwidth, the higher the number of processors. it may be necessary to implement the subfarm as a LAN, attached to the SFC which, in turn, acts as a gateway between the two networks.

The *Readout Network* performs the following functions:

- The main function of the Readout Network is to route all the parts of a given event to one destination. As such it participates in the event building process. The concentration of data towards one destination results in contention for the resources that are shared in the network.

- The network may also be used to route control message for one of the proposed control schemes.

4.1.2 Trigger/DAQ Performance Requirements

Selection criteria applied at Level-0 and Level-1 give a 40 kHz readout rate. The data acquisition system will transfer data into process where high level trigger algorithms reduce the event rate to storage to some 100–200 Hz.

LHCb group has decided to break down the high level algorithms into two broad classes:

- The *Level-2* algorithms, using the track vertex guidance from the trigger (Level-1), look to *reduce the rate to some 5 kHz* using partial event data. The strategies will include refining the vertex detector decision with full precision data and enhanced algorithms, and then combining Vertex Detector and data from the tracking system.

- The *Level-3* algorithms will involve partial or full event reconstruction to apply physics cuts appropriate to the CP violation channels.

The estimated *processing power* required to run the algorithm is:

- Level-2: 10 MI [1]

- Level-3: 200 MI

[1] MI = Million Instructions per event (from the analysis computation point of view)

4.1.3 Data Rates and Detector Partitioning

The implementation studies presented are based on a partitioning of the detector as shown in table 4.1 where the DAQ aspects are outlined. The data rates per sub-detector are given, as well as the distribution of read-out channels. Each sub detector is composed of segments, a segment being read out through one or more data links. An up-to-date layout of the detector partitioning and estimates of data produced can be found in [20].

Sub-detector	Event size [kByte]	Segments	Links per segment	# links	Data per link [kByte]	Throughput per link [MByte/s] (at 40 kHz)	Throughput [MByte/s]
Vertex	5.6	17	1	17	0.33	13.2	224
In-Tracker	16.7	11	4	44	0.38	15.2	668
Out-Tracker	37.5	10	4	40	0.94	37.5	1500
RICH 1	15.3	2	8	16	0.96	38.3	612
RICH 2	5.6	2	16	16	0.35	14.0	224
PRESHW	1.4	3	1	3	0.47	18.7	56
ECAL	8.3	3	4	12	0.69	27.7	332
HCAL	1.4	2	1	2	0.70	28.0	56
Muon	1.4	2	1	5	0.28	11.2	56
Trigger	6.9	1	10	10	0.69	27.6	276
Total	100	56		165			4004

Table 4.1: Detector partitioning

4.2 Network Implementation

If we assume an uniform distribution between the RUs ports, the number of ports required is given by

$$N = \frac{E}{\frac{B}{8} \cdot k \cdot u} \cdot F \cdot \left[\rho + \frac{(l - \rho)}{R} \right]$$

where
E = event size [Bytes]
B = network link bandwidth [bit/s]

k = link load factor (0 to 1)
u = fraction of link bandwidth available for user data
ρ = fraction of E needed by Level-2 algorithm
F = Level-1 trigger rate [kHz]
R = Level-2 rejection factor

Table 4.2 gives the estimated network size for the various technologies taken into account and for both readout protocols. The size is given by the total number of ports which, according to the assumption of equally loaded sources, is twice the number of RU ports. One assumes:

- An uniform distribution of data between the sources.
- 40 kHz Level-1 trigger rate and an occupation of 50% per link.
- For the phased event building solution: Level-2 algorithm using 40 kByte/event with a rejection factor of 8.

	total # of ports, with 50% loaded RU links	
Network technology	Full readout at L2	Phased L2/L3 readout
155 Mbits ATM	950	500
622 Mbits ATM	240	124
100 Mbits Ethernet	1600	850
1000 Mbits Ethernet	160	85
1280 Mbits Myrinet	112	60

Table 4.2: Estimate of network size

All the technologies considered here providing bi-directional connections (full-duplex), the network must provide twice the bandwidth required for DAQ data transfer. Table 4.2 gives the total number of ports required for sources and destinations. A first crude estimate is 50% of ports for RUs and for SFCs respectively.

The size indicated in table 4.2 is a lower limit: it assumes that it is possible to load the link network up to $\sim$ 100% to provide the required

safety factor. It must be also taken into account that flow control mechanisms reduce the effective bandwidth usage due to the unavailability of the link when the traffic is halted.

4.3 Implementation of Readout Units

A readout unit (RU) implements the following functions, some of them being compulsory while the others are only required as indicated:

- *Match the FE links to the network topology*:
  This is the main purpose of a source module: to match the DAQ requirements with the network topology. The Front-end (FE) links are determined essentially by the detector topology and partition. It is mandatory to keep a mapping of FE links to RUs that respects the detector topology in view of running the DAQ in partitioned mode. On the other hand, a RU has to collect enough data to efficiently load the network link.

- *Build sub-events*:
  A RU collects *event fragments* from one or several FE links and builds a *sub-event* by adding protocol data to distinguish each event fragment and to identify the sub-event.

The number of FE links is variable and must be configured for each RU.

- *Destination assignment*:
  Is another important function ensuring that, for a given event, all the RUs send their sub-event to the same destination: this is the *destination assignment* function that has to be implemented, possibly in connection with centralised control unit.

- *Buffering in case of phased event building*:
  If phased event building strategy is adopted, the RUs holding data not needed for Level-2 algorithm must implement the delivery of sub-events on request from a processor. The data of rejected events

must be discarded. It is desirable that this scheme is present in all RUs, even when they actually provide only Level-2 data.

4.3.1 Requirements

The data flow per FE link is given in table 4.1 where the event size has been normalised to 100 kByte. The data rates are given for a Level-1 rate of 40 kHz. The rate ranges between 11.2 MByte/s and 38.3 MByte/s per link.

The rate at which the event fragments delivered by the FE links to be assembled is n $\times$ F where n is the number of FE links read out by the RU and F is the Level-1 trigger rate. The working point is at F = 40 kHz. The event fragments are not too small: from 300 Byte to 1 kByte.

In order to fulfil the safety factor requirement, a RU should be able to sustain a data flow twice as large as the values given in table 4.1 or a trigger rate of F = 80 kHz.

The sub-events are submitted to the network at the rate F.

The RU must provide enough buffer space to cope with the normal fluctuations of the input data rate. When the rate exceeds the design values, the RU must be able to stop the Level-1 trigger until the buffer occupancy reverts to a safe level.

4.3.2 Determination of the Number of Readout Units

The number of RUs is determined:

1. by the network link bandwidth

2. by the association between FE links and RUs. There is not much flexibility in the definition of the FE links which must correspond to the partitioning of the detectors and to their data rates.

The network technology must provide a link bandwidth large enough to match the largest FE link demand. Possibly a RU can multiplex several FE links.

Link Bandwidth Limitations: Maximum Sub-Event Size

The maximum size of a sub-event S [kByte] depends on the available link throughput of user data B [MByte] and the trigger rate f [kHz]:

$$S = k \cdot \frac{B}{f}$$

k is the load factor of the link ($0 \leq k \leq 1$).

Figure 4.3 shows the sub-event size as a function of the link bandwidth:

1. at 40 kHz and for k = 0.5 and 1.0.
2. for k=1 (100% load) and 3 LHCb characteristic DAQ frequencies: 80 kHz (Level-1 maxi requirement), 40 kHz (Level-1 working point) and 5 kHz (Level-2).

Association of FE Links to Sources

Table 4.3 shows, for the full event building strategy and for the various technologies, a possible assignment of FE links.

Three values in each cell are:

• *1st value*: the multiplexing factor, i.e. the number of FE links that are combined in one source to form a sub-event (if a fraction: each Front-end link needs to be de-multiplexed in order to fit with the network bandwidth).

• *2nd value*: the resulting load on the network link for 40 kHz Level-1 rate, 100 kByte/event (the target is $\sim$ 50%).

• *3rd value*: the number of sources for the sub-detector.

The last row gives respectively: the maximum (de)multiplexing factor, the maximum load and the number of sources.
The maximum multiplexing is 4.

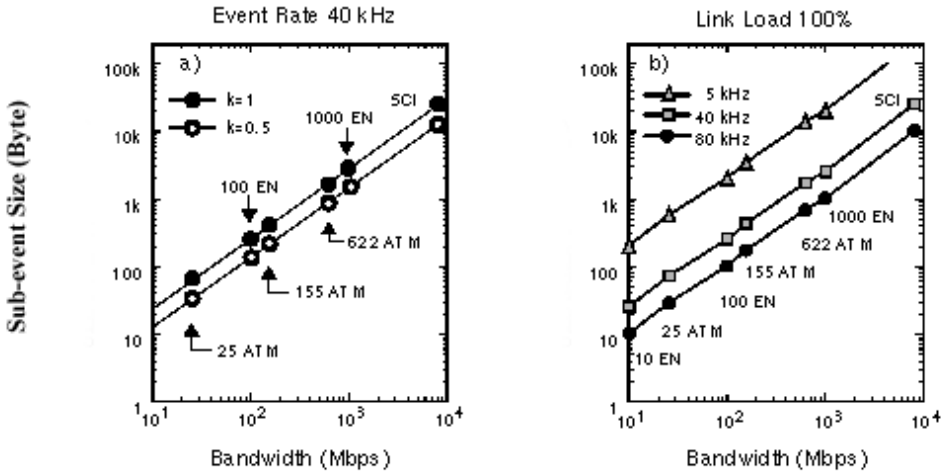

Figure 4.3: Sub-event size versus link bandwidth

4.3.3 Meeting the Performance Requirements

The real challenge in the implementation of sources is the high frequency of event fragment arrival: a Level-1 trigger rate of 40 kHz at working point and, possibly, 80 kHz as maximum sustainable rate. The multiplexing function (or sub-event building) that must be provided in a source multiplies also the rate of arrival of event fragments which, although arriving in parallel, must still be completed with protocol data and combined into a sub-event.

At 80 kHz a packet is emitted every 2.5 μsec. A multiplexing factor of 3 generates a stream of event fragments at 240 kHz. These very stringent conditions are by far beyond the possibilities of General Purpose processors.

This is also a difficult or even impossible task for networks with flow control requiring an acknowledgement procedure. In this case, it is necessary to group sub events from several consecutive events and build

Sub-Detector	ATM 155	ATM 622	Ethernet 1000	Myrinet 1280
Vertex	12 - 39 - 34	2 - 39 - 9	4 - 47 - 5	4 - 37 -5
Inner Tracker	12 - 45 - 88	2 - 45 - 22	2 - 27 - 22	4 - 42 - 11
Outer Tracker	14 - 56 - 160	1 - 55 - 40	1 - 33 - 40	2 - 52 - 20
RICH 1	14 - 57 - 64	1 - 57 - 16	1 - 34 - 16	2 - 53 - 8
RICH 2	12 - 42 - 32	2 - 41 - 8	3 - 37 - 6	4 - 39 - 4
PRESHW	12 - 55 - 6	1 - 28 - 3	3 - 50 - 1	3 - 39 - 1
ECAL	13 - 55 - 36	1 - 41 - 12	2 - 49 - 6	2 - 38 - 6
HCAL	14 - 42 - 8	1 - 41 - 2	2 - 48 - 1	2 - 39 - 1
Muon	12 - 33 - 10	2 - 33 - 3	2 - 20 - 3	3 - 23 - 2
Trigger	13 - 55 - 30	1 - 41 - 10	2 - 49 - 5	2 - 38 - 5
Total	14 - 60 - 468	1 - 57 - 125	4 - 50 - 105	3 - 53 - 63

Table 4.3: Allocation of FE links to sources for various technologies (full event building).
Cell contents: (De)-Mux Factor - Link load (%) - Number of links

"super-events" [2]. The frequency is reduced in proportion of the packing factor.

If phased event building is implemented, the RUs collecting data not needed for Level-2 trigger have to implement a higher multiplexing factor: being solicited at a lower frequency (by a factor 8, namely $\sim$ 5 kHz), the should multiplex more data ($\sim$ 8 times more) in order to achieve the desired load on the network. This is true only if the technology provides a single bandwidth value. If lower bandwidth links can be used, the multiplexing factor can be reduced. As an example, if the Level-2 RUs are implemented with 622 Mbps ATM links, the Level-3 RUs can be implemented with 155 Mbps links and the multiplexing factor needs to be increased by a factor of 2 only.

Phased event building causes a shift of buffer occupation from the destinations (SFCs) to the RUs, as compared to the full event building scenario. One can estimate the buffer required to store sub-events that are only used at level-3.

[2] With "super event" we mean several events packed all together to solve the difficult problem of a high event trigger rate.

4.4 Implementation of Sub-Farm Controllers

A Sub-Farm controller implements the following functions:

- *Event building*:
  Assembly of sub-events with the possibility to have several events built concurrently. The determination of the completion for an event can be based on simple enumeration or on an explicit list attached to each event or on time out. The latter method must be implemented in all cases to detect accidentally missing sub-events.

- *Dispatching of events*:
  Completely assembled events have to be transferred to one of the processors of the sub-farm for executing the high-level trigger algorithms. This can be combined with resource management, the SFC keeping track of the individual CPU occupancy.

- *Permanent storage*:
  The SFCs should also take care of the transfer of finally accepted events to the storage sub-system.

4.4.1 Requirements

There is no bandwidth requirements on individual SFC network ports. However the aggregate bandwidth must match with the bandwidth effectively delivered by the RUs. In the case of equal link bandwidth for RU and SFC links, the number of SFC ports may be lower than the number of RU ports: due to the inefficient mapping of FE links, the RU links are usually under loaded.

Each SFC must accommodate a sub-farm offering a processing power sufficient to process the flux of events. The sub-farm network must provide a bandwidth large enough to accommodate the data rate from the DAQ network.

It is desirable that a SFC performs itself the resource control of the sub-farm and has the means to request a reduction of flux to avoid overflow.

It is reasonable to expect that the sub-farms will not all be equivalent from the point of view of processing power. The DAQ event building control should be able to cope with this situation, while making the best possible use of the available processing resources.

4.4.2 Number of Sub-Farm Controllers and CPUs per SFC

The number of CPUs required to process the data arriving at destination is determined by:

1. the number of events arriving at the SFC per unit of time;
2. the processing time per event which, in turn, depends on the processing power of the individual processors.

For the calculation of the average processing time per event in a given CPU, we define the following variables:

- MIPS = processing power of 1 CPU [Million Instructions/s]
- η = effective use of processing power (0 to 1) — takes into account the overheads
- MI(L2) = algorithm requirements for Level-2 algorithm [Million of Instructions]
- MI(L3) = algorithm requirements for Level-3 algorithm [Million of Instructions]
- R = rejection factor of Level-2 algorithm
- T_{avr} = average processing time per event [s]

The average processing time per event is given by the formula:

$$t_{avr} = \frac{1}{\eta \cdot MIPS} \cdot \left[MI(L2) + \frac{MI(L3)}{R} \right]$$

Using values from the requirement document and assuming processors of 1000 MIPS with an affective CPU usage of 70%, we can compute an average processing time as follows:

- MIPS = 1000, $\eta = 0.7$
- MI(L2) = 10, MI(L3) = 200
- R = 8
- $\Rightarrow t_{avr} = 50$ msec

For the calculation of the number of events arriving in a destination, in the case of *full event building*, we need the following parameters:

- B = the link bandwidth [bits]
- k= the average load of the link (0 to 1, usually 0.5, corresponding to the safety factor)
- E = the full event size [Bytes]
- u = fraction of the bandwidth available for user data

The transfer time for 1 full event on the network link is given by:

$$T_{full} = \frac{8 \cdot E}{B \cdot u}$$

If we choose the working point of a link to a destination with a load factor k, then the average number of events arriving at a destination per second is given by

$$\frac{k}{T_{full}}$$

The minimum number of processors required to process the events at a destination is given by:

$$N_{proc} = k \cdot \frac{t_{avr}}{T_{full}}$$

In the case of phased event building:

- The average processing time per event is the same
- The number of events arriving at an SFC is larger since only a fraction ρ of an event data is transferred while the remaining (1-ρ) is transferred every R events (R = rejection factor of Level-2)

The average event size is given by:

$$E_{phased} = E \cdot \left[\rho + \frac{1-\rho}{R}\right]$$

This value can be substituted to E (full event size) in the previous formula to calculate the number of processors.

Table 4.4 gives the number of processors that must be provided at a destination port for each technology and for both protocols. These number have been calculated under the assumptions given previously. It should be noted that, although a safety factor of 2 is applied for the link load, the processors are assumed to be fully busy.

	Full readout		Phased readout	
Network technology	# of CPUs per destination	# of destinations	# of CPUs per destination	# of destinations
155 Mbits ATM	5	400	8	250
622 Mbits ATM	17	120	34	60
100 Mbits Ethernet	2 - 3	800	5	400
1000 Mbits Ethernet	25	80	57	35
1280 Mbits Myrinet	36	56	73	27

Table 4.4: Destination module design considerations

4.4.3 Meeting the performance requirements

The aggregate bandwidth of the sub-farm network needs to be at least as large as the network link bandwidth. There should be no problem of congestion on this sub-network since this is a “one to many” traffic controlled by the SFC. Thus networks like Ethernet, that usually have a low efficiency, should deliver high throughput in this particular application.

As expected, low bandwidth connections need to accommodate a low number of processors, whereas high bandwidth links require a large farm of processors. The choice of the destination link bandwidth, if possible, will be trade-off between the cost of an SFC and the cost of the sub-farm network. The frequency of sub-event arrival in a SFC depends on the ratio between the RU link bandwidth B_s and the SFC link bandwidth B_d.

4.5 Boundaries of This Work

This chapter has minutely presented and explained the whole LHCb DAQ system and its feasibility, according to the assumptions and the model results discussed in the DAQ Implementation Studies document [19].

As already stated, the region of interest of this work is the LHCb DAQ system, and in particular our attention is focused on the readout network and its interface with both the source models (RUs) and the destination modules (SFCs). We are not going to study in detail and then implement a complete RU module (this is the project of another group of study with which we were closely, of course) and, at the same time, we are not involved in building the software for the higher trigger algorithms. The aim of this work is to investigate the problem of the event building through a readout network and propose new feasible solutions. We will specifically focus on the RU and SFC interfaces with the network, studying a new idea of embedded event building on a specific Gigabit Ethernet network interface.

Thus a Gigabit Ethernet readout network implementation with special interfaces at source and destination access points will be main topic of the following chapters of this document.

Chapter 5

Event Building Protocol

This chapter focuses on the definition of the software for source and destination interfaces. Calling this event building software a protocol is not completely correct. According to the standard literature, in fact, a protocol is basically an agreement between the communicating parties on how communication is to proceed. This means that all parties must take part in both sides of the communication or, in other words, this means that each party should send and receive data. That is not completely true in the case of event building architecture: here sources just send event fragments through the network, and destinations simply receive data to assembly. There is not a real communication, according to the standard definition, because some parties only "speak" and the others only "listen".

However, we think we can speak of protocol about event building software because even in this case there are some restricted rules that govern the communication, like the destination assignment, and the choice of no retransmission of lost fragments is one of these and characterises our protocol.

5.1 General Concepts

Event building software develops the source and destination interfaces and their event fragment management. There are some basic features

that the software should implement to characterise the event building protocol and these characteristics are specific of each interface, as will be described in the following subsections.

Anyway, before going on with the analysis of the general properties of the event building protocol, it should be remembered that from this point of view the source model has the only aim of sending events fragments to a defined destination and it has nothing to do with the sub-event building already done by the previous part of the readout unit. Similarly, the destination model performs the task of assembling received fragments into complete events and it does not do any kind of analysis on the assembled data, which is done by the Level-2 and Level-3 algorithms in the CPUs of the sub-farm.

5.1.1 Sending an Event Fragment

When data from a new event fragment arrives, it is formatted in an event fragment PDU (Protocol Data Unit), which contains a payload and a PCU (Protocol Control Unit) with information about destination address, event number and optional event building control information (for example the number of the source sending the fragment).

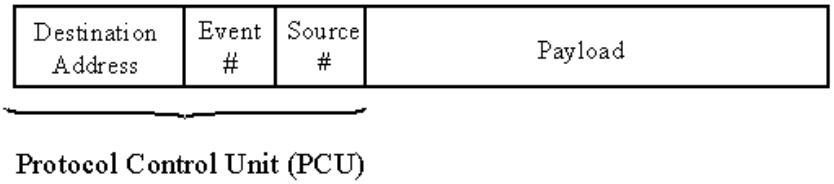

Figure 5.1: Event Fragment PDU

An important issue of the source model is the destination assignment. It can be either static or dynamic.

In the *static destination assignment*, each source determines the destination as a function of the event number. The resulting system is quite simple and does not requires a separate control network linking RUs and

SFCs to the Trigger Manager, as figure 5.2 shows. The static destination assignment could be implemented as a round robin scheme, but in this case the system performance would be determined by the least performing sub-farm.

A better solution could make use of destination assignment tables which roughly balance the load according to the computing powers of each SFC.

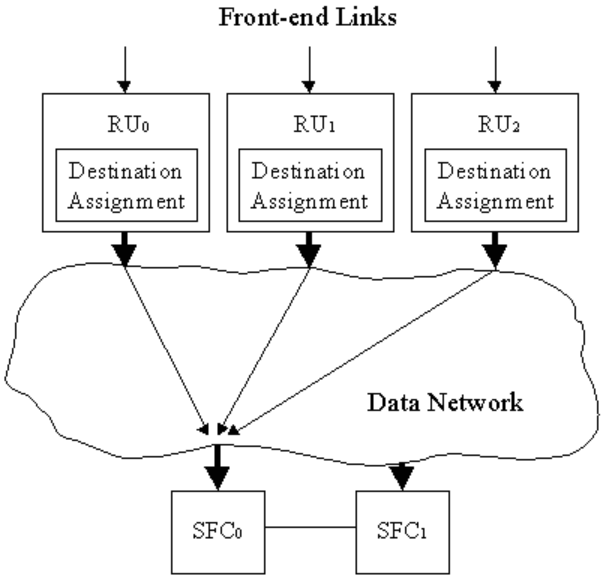

Figure 5.2: Static destination assignment

This second solution has the following consequences:

- Requires benchmarks of the relative sub-farm performance.
- A change in the destination assignment function can only occur when the system is completely stopped in order to ensure identical information in all the RUs.

In both solutions (round robin and table driven assignment), the failure of a SFC leads to a loss of all data sent to this destination, until the situation is detected, signalled and corrected.

As pointed out before, it is not defendable that a SFC running out of resources causes a reduction of trigger rate for the whole system, when resources might be available elsewhere. The problem of scarce resources in sub-farms is less severe with SFCs implementing large sub-farm and performing local load balancing amongst the processors of the farm as the destination assignment is not directly to the end processor but to the SFC.

In the *dynamic destination assignment*, shown in figure 5.3, the basic idea is to implement a control system that integrates data transfer and protocol signalling on a single network in order to realize a dynamic load balancing on the destinations.

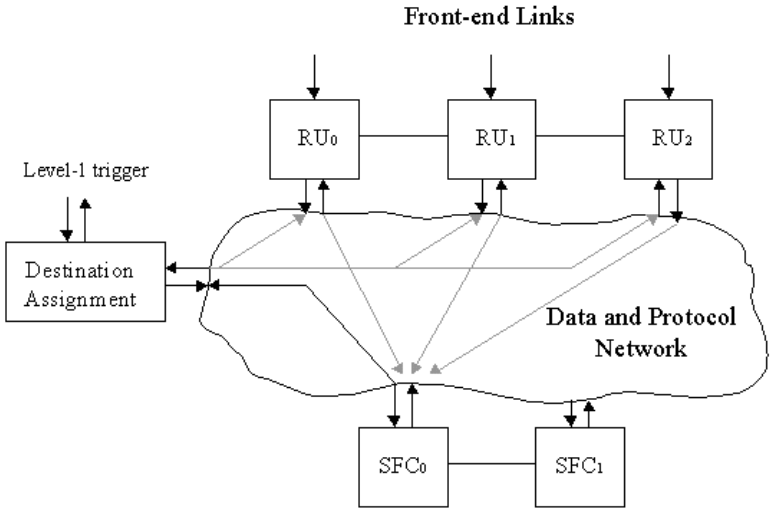

Figure 5.3: Dynamic load balancing

The protocol is based on credits managed by a supervisor connected

to the DAQ network. An SFC allocates credits whenever an event (or a group of events) has been processed in the sub-farm. The credits are collected by a supervisor which, on their basis, multicasts destination allocations to the RUs. In order to keep the message traffic at a low frequency, destination allocations for consecutive events may be grouped.

This solution has the following characteristics:

- Allows dynamic load balancing on the destination: a destination receives an event only if the resource to handle is available.
- Requires a supervisor module (possibly more than one if many partitions are active simultaneously).
- Data loss is limited to the amount of unused credit issue by a SFC just before its failure.
- The message rate is very high (Level-1 trigger rate), but by grouping messages it can be reduced to an arbitrarily low rate.

The main drawback of this solution is to rely on a single component that would stop the system in case of failure, unless a backup system is implemented. In addition, the source modules have to implement an input stage from the RN to receive the messages from the supervisor.

5.1.2 Receiving an Event Fragment

A destination receives multiple event fragment PDUs for every event, one fragment per event from each source. Furthermore, even if packets from the same source arrive in sequential order, packets from different sources are received in random order. This means that, at the same time, a destination has to manage not only several fragments, belonging to the same event, but also more than one event.

For each event the destination manages an *event descriptor* which contains status information and a table of pointers, one for each source. Each pointer either points to a fragment or has a special value that specifies that no fragment has been sent for this event from this source.

When the software gets a fragment, it checks the event and source numbers. Then it searches for the corresponding event descriptor and writes the pointer to the appropriate field of the event descriptor. If the fragment belongs to a new event a new event descriptor is allocated to that fragment and the building of a new event is started.

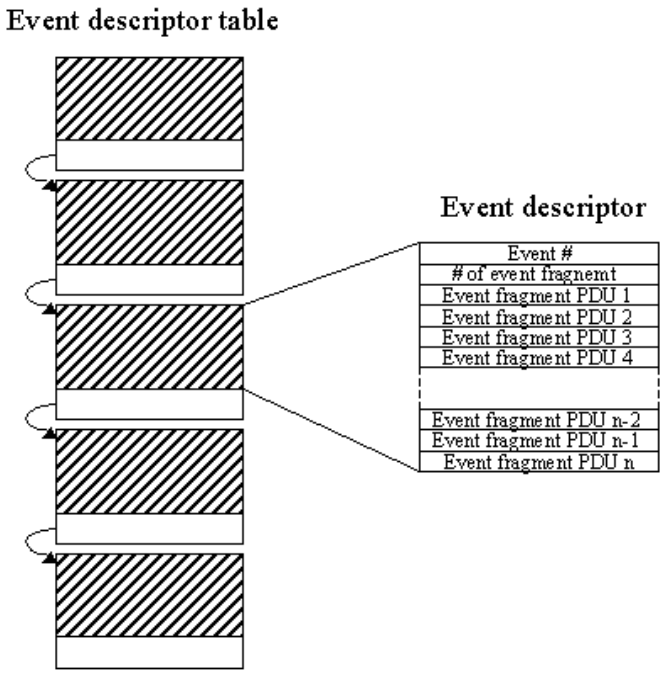

Figure 5.4: Event descriptors

5.1.3 Event Building Completion

The final choice of an algorithm for deciding when all fragments have arrived and so the event is completed has to be made according to the specific requirements of the event builder. This is an important issue

that can have many different solutions, as we can see from the following subsections.

Implicit Algorithm

This algorithm is the simplest implementation for detecting if an event has been completed. The basic idea is that all the destinations know the number of sources which are supposed to send a fragment for each event. In this way, when an event fragment arrives, it is assigned to the correct event descriptor and the counter of sources that are expected to send fragments is updated. When all sources have sent their fragment, the event is considered complete. Of course, if a fragment is lost the corresponding event will never be completed and analysed. It should be better to implement solutions which take into account the problems of data losses or dead sources and avoid data deadlocks in the destinations, like the following algorithms.

This algorithm in fact is not usable in a real system, but it will be useful to evaluate the cost, in terms of performance, of various type of safety measure implementations.

Implicit + Time-out Algorithm

When a new event descriptor is activated it is put in an *active event descriptor list* or *table* and an expiration timestamp is written in the descriptor. Event descriptors contain a counter of the number of sources from which an event fragment should be received. When a new event fragment arrives, the software checks if all fragments belonging to that event have been received. Furthermore the software checks if the other events in the table have a timestamp older than the maximum allowed.

In other words, an event is considered as complete a) when it has received fragments from all sources or b) when its timestamp is expired. The expiration time is usually determined from the tail distribution of the total event builder latency in order to guarantee a low probability of data loss in the rare case where missing fragments may arrive after event time expiration. Thus the time-out is a static time assignment which is computed according to the system dimensions.

By this method, a source needs not to send data when it has nothing to send. However, this case cannot be distinguished from the case where the source is faulty. A drawback of this solution is that in the case of sparse data most events have to wait until the timestamp expiration.

Notification Algorithm

For each event, the sources send either a fragment or a notification PDU, namely a special packet which informs the destination that a source has nothing to send for the current event. Once a fragment or a notification from a new event arrives, a new event descriptor is activated in the *active event descriptor queue*. Event descriptors contain a counter of the number of sources form which either a notification or an event fragment is going to be received. When all sources have sent something than the event is completed.

A time-out mechanism is still maintained for detecting data losses or dead sources.

Event Sequence Number algorithm

If we are sure that the sequence order of fragments sent by a source to a destination is preserved, it is easy to implement a scheme where empty fragments need not to be notified: any missing fragments in a sequence can be interpreted as empty fragments by a destination. In order to keep the time between two non-empty fragments within an acceptable value and to trace dead sources, notification of empty fragments can still be sent, whenever adequate.

The event sequence number tells how many events have been sent to a given destination. As an event is assigned to a destination there is a correspondence between an event and a sequence number. Each source sends together with a fragment, the sequence number.

In the receive side there is an *active event descriptor table* where each sequence number is associated to an event descriptor pointer. There is also a *last event list* where to each source is associated the last sequence number received. When a fragment arrives, its sequence number is compared to the maximum. If it is greater, it means that a fragment from a

new event has been received and an event descriptor is assign to it and put in the active descriptor table. Then the sequence number is compared to the last sequence number received from the source that has sent the fragment. If the the difference is greater than one there are events for which there was no data from that source and the corresponding event descriptors are updated. If an event has been completed, than it is put in an *event descriptor complete queue* for the analysis.

In the source a time-out mechanism guarantees that a destination does not wait too long for a fragment from a source which has nothing to send. When the source has not sent anything to a certain destination for a fixed amount of events assigned to that destination, it sends a notification fragment to the destination.

In the destination there is a second time-out in case of errors or sources out of work. The active event descriptor list is implemented as a circular queue. When it becomes full the oldest event descriptor is considered to be completed and it is put in the event descriptor complete queue.

5.2 Implemented Algorithms

According to all the problems and relative solutions discussed above, we have created our own event building protocol, making reasonable assumptions concerning the requirements.

The aim of this implementation is to measure the software overhead due to the protocol in the overall event building system. This software will also be used to implement a small scale model of the event builder, as will be described later.

All the software we have developed is written using the C standard language and its libraries. C was chosen because, at the moment, it seems to be still the best and well performing solution for programming embedded systems (which is the final goal of our work).

First of all we have concentrated our attention on the *source module* and its functions. As pointed out before, a source has two main tasks to accomplish, which are the formatting of an event fragment, sent by the RUs, in an event building PDU and the destination assignment.

In our software the PCU, that represents the protocol header of an event fragment, is made of four fields, which are:

- Destination address;
- Event number;
- Source number;
- Length (that includes header and payload).

We have decided to limit ourselves and thus implement a static destination assignment. It is based on the "modulo" function, as we can see from the following formula:

$$\textit{Destination Assignment} = (\textit{Event Number}) \text{ modulo } (\textit{Source Number})$$

Even if this function is quite simple, it should be a good solution for the problem of the destination assignment and it is computed in few instructions, so it is not a big issue in terms of execution time, thanks to its simplicity. With a number of SFCs equals to 100 (which is the estimate of the LHCb DAQ system), a destination will manage 1 event every 100 events generated. This will give to the destination a safety margin of time to do the event building. In fact with an aggregate rate of 40 kHz means an average time of 25 microseconds per fragment.

The most complex part in the study of our demonstrator code was the simulation of several sources and the simulation of the random traffic generated by them. Of course, at the moment, the full DAQ system with 100 sources, 100 destinations and the readout switching network is not available. Thus, in order to estimate the behaviour and the adequacy of the destination module, we have to emulate the presence in a single source of n sources which send to a destination one fragment per event. Furthermore, it is also necessary to simulate somehow a random event fragment traffic, because, in reality, the fragments of an event will not arrive at one destination all at the same time, due to the event building protocol overheads and the readout network. For these reasons, other

than formatting received fragments in PDUs and assigning them the destination address, our source module presents a consistent part of code which performs the simulation functions.

In the performance evaluation, the overhead due to this emulation of n sources needs to be subtracted. In fact we know that the source is not so much loaded, because its tasks are limited and rather simple, and so we are not really interested in its performance measurements. The real aim is studying the destination performance and developing a destination model which could be as fast as possible while meeting all the event building requirements.

In the next subsections we will explain in detail the fragment generation for the random traffic simulation and the three different destination algorithms that we have studied and implemented.

5.2.1 Event Fragment Generation

The random fragment generation is based on the following assumptions:

- For each event all sources must send a fragment to the assigned destination
- All fragments belonging to the same source must arrive in order (according to the event enumeration order).

The number of sources is a parameter decided at the beginning and it can be changed at any time the program is stopped. It is not a run time variable, of course. It must be decided before the run and the whole model is dimensioned according to this value.

Once the number of sources is given, the generation function starts creating the event fragments following the next steps: it first generates an event (one fragment per source) and assigns to it a random value (t_{evt}) chosen in a limited range of an inverse exponential distribution. Then it gives to each event fragment a random value (t_{frg}) chosen in limited range of a poissonian distribution (which can be simplified as figure 5.5 shows). At the end all the fragments will have their own value ($t_{fragArrival}$) which is given by the sum of $t_{evt} + t_{frg}$. At this point the generation function

orders in a stack all the fragments generated according to their $t_{fragArrival}$ value and starts the generation of another event following the same procedure. Once another event has been created, all the fragments (the ones of the last generated event and the others of the previous event) are reordered again in the stack according to their $t_{fragArrival}$ value. Figure 5.5 shows all the steps needed for generating an event.

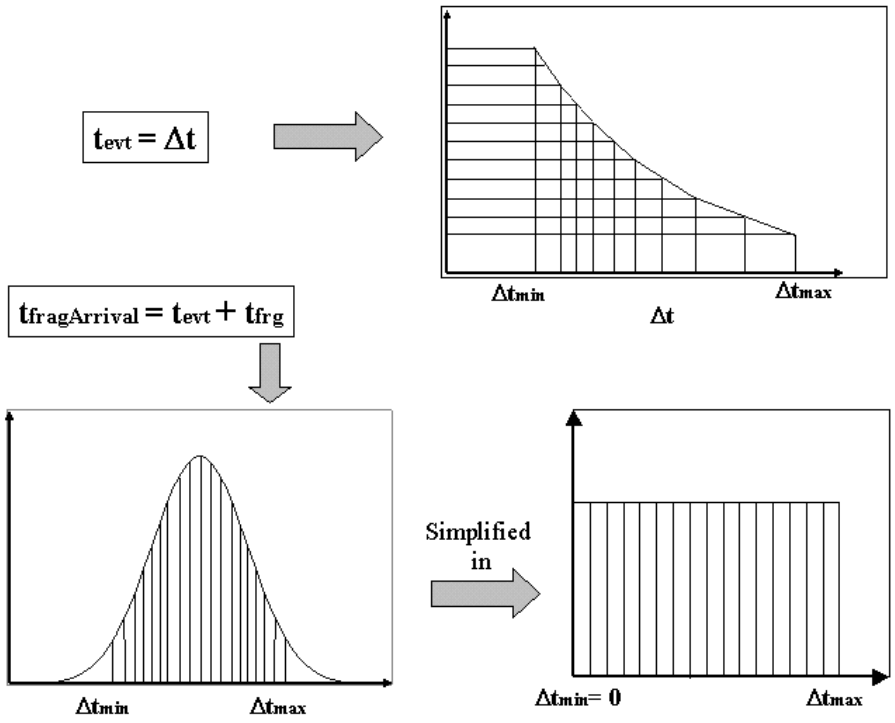

Figure 5.5: Fragments generation

Of course, the generation function takes care also of the fact that different event fragments belonging to the same source must be ordered. Thus if a fragment, with an event number bigger than the event number of a fragment belonging to the same source, has a $t_{fragArrival}$ value smaller than the $t_{fragArrival}$ of the other fragment, it is anyway put in the stack

after the second fragment (i.e. after the one with smaller event number). In this way the fragment order is kept correctly.

The generation function goes on with the event generation until it creates an event with all fragments having a $t_{fragArrival}$ value bigger than the previous event fragments. At this point all the events in the stack are randomly mixed in a correct order and they can be consumed by the event building protocol for the formatting and the forwarding.

The generation function restarts when all the fragments in the stack have been consumed, except the ones of the last generated event, and it follows always the same procedure (figure 5.6 illustrates this procedure).

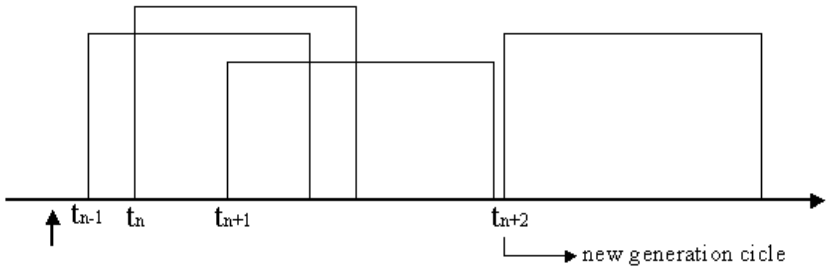

Figure 5.6: Event generation

This function has been developed starting from the assumption that all the events generated are addressed only to one destination, because all the simulations we have done can count on the presence of only one destination. Anyway it can be immediately adapted and used also with a simulation that presents more than one destination.

5.2.2 No time-out

The *no time-out* algorithm is the simplest version of the destination model. It is based on the implicit event building completion: all sources are supposed to send a fragment per event, and so the destination considers an event complete only when it has received for that event a fragment

from each source.

The destination keeps a descriptor table where all the events under construction in a certain moment are stored. Each event descriptor contains the event number, a counter of the sources which have not sent their fragment yet, and a list of as many pointers as the number of sources (figure 5.7).

Event #	Event #	Event #	Event #	Event #	Event #
Remaining sources	Remaining sources	Remaining sources	Remaining sources	Remaining sources	Remaining sources
ptr 1	ptr 1	ptr 1	ptr 1	ptr 1	ptr 1
ptr 2	ptr 2	ptr 2	ptr 2	ptr 2	ptr 2
ptr 3	ptr 3	ptr 3	ptr 3	ptr 3	ptr 3
ptr 4	ptr 4	ptr 4	ptr 4	ptr 4	ptr 4
.	.	.	.	.	.
.	.	.	.	.	.
.	.	.	.	.	.
ptr n-1	ptr n-1	ptr n-1	ptr n-1	ptr n-1	ptr n-1
ptr n	ptr n	ptr n	ptr n	ptr n	ptr n

Figure 5.7: Structure of the descriptor table

When a new fragment arrives, the algorithm checks immediately if this fragment belongs to an event already in the table. If not, the algorithm simply assigns to the fragment a free descriptor in the table and decrements the counter of the missing sources.

Otherwise, if the arrived fragment belongs to an old event already in the table, the algorithm looks at the source number the fragment comes from and addresses the right pointer in the descriptor list to the fragment. Then it decrements the counter of the sources which have not sent anything yet and checks if the event has been completed (which means that the missing source counter must be zero). When an event is

completely built, it is given to the processor for the physics analysis (in our simulations it is simply discarded) and the corresponding space in the descriptor table is freed. Figure 5.8 summarise the main steps of the algorithm.

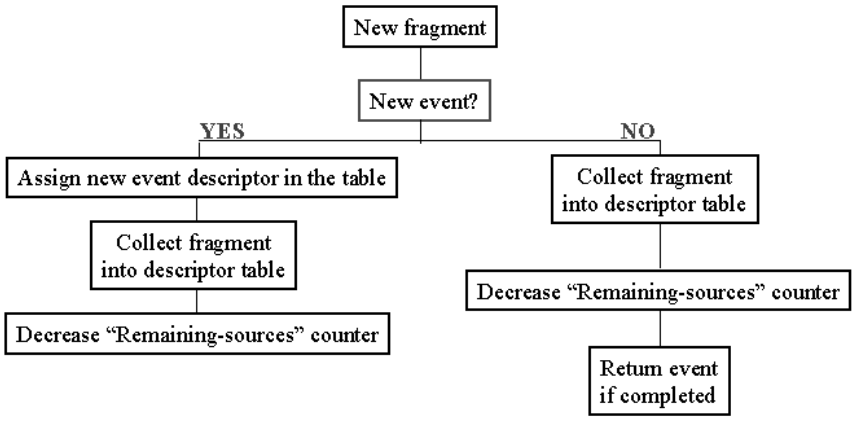

Figure 5.8: No time out algorithm

As already stated, this implicit completion algorithm is fast in its computation but is not reliable, because it trusts to a perfect system with no data losses and no dead sources. With such destination module, in fact, it can happen that uncompleted events lie forever in the descriptor table, because some fragments are missing, lost in the network or never sent, due to a source crash. Therefore, a sort of time-out should be implemented to avoid event deadlocks and to make the whole system more secure. This is what we have done with the second destination algorithm, the simple time-out, whose description in coming in the next subsection.

5.2.3 Simple Time-out

The *simple time-out* algorithm is a sort of implicit + time-out completion. Like the no time-out, the simple time-out algorithm stores the events

under construction in a descriptor table which has the same fields of the previous one (number of the event, number of sources that have not sent the fragment and the list of pointers, one for each source) but keeps an additional label with the timestamp value for the time-out (figure 5.9). The destination maintains also a global variable where is written and updated the value of the next event number which is expected.

Event #	Event #	Event #	Event #	Event #	Event #
Remaining sources	Remaining sources	Remaining sources	Remaining sources	Remaining sources	Remaining sources
Timestamp	Timestamp	Timestamp	Timestamp	Timestamp	Timestamp
ptr 1	ptr 1	ptr 1	ptr 1	ptr 1	ptr 1
ptr 2	ptr 2	ptr 2	ptr 2	ptr 2	ptr 2
ptr 3	ptr 3	ptr 3	ptr 3	ptr 3	ptr 3
ptr 4	ptr 4	ptr 4	ptr 4	ptr 4	ptr 4
. . .	. . .	. . .	. . .	. . .	. . .
ptr n-1	ptr n-1	ptr n-1	ptr n-1	ptr n-1	ptr n-1
ptr n	ptr n	ptr n	ptr n	ptr n	ptr n

Figure 5.9: Structure of the descriptor table

When a new fragment arrives, it is compared with the value of the next expected event and, according to the comparison result, the algorithm behaves in three different ways:

1. *The event number of the new fragment is less than the expected event number*

   In this case the fragment belongs to an old event which could be:

   - still situated in the descriptor table for the the reconstruction;

- already returned to the processor for the physics analysis, even if incomplete, because of the time-out.

Thus the algorithm checks first if the event is still in the table. If so it assigns to the fragment the right pointer in the descriptor list, according to the source number, and decreases the counter of the sources which have not sent the fragment yet. If the event becomes complete after this source update, the algorithm marks it as "complete event", gives it to the processor for the final analysis and frees the space in the descriptor table.

Otherwise, the algorithm marks the fragment as "out of time" because, for some reason, it has arrived after the standard time given for a single event building. The rejected fragment is then passed to the processor that will treat it as an exception.

In both cases, after the new fragment analysis, the algorithm checks if there are event descriptors in the the descriptor table with the event number less than the one of the just arrived fragment, that have not received anything form the source the current fragment belongs to. If this is the case, it means that those missing fragments will probably never arrive (this can be argued from the emitted fragment order of a single source) and so the algorithm marks them as "missing fragment" and decreases the source counters of the associate descriptors. At the same time the algorithm checks if, after this updating, any event has become complete, in the way that it has not to wait for any other source. If so, the algorithm gives the event to the processor, marking it as an "incomplete event", and free the space in the descriptor table.

The main steps are summarise in figure 5.10

2. *The event number of the new fragment is equal to the expected event number*

   In this case the fragment arrived belongs to a new event and so the algorithm assigns it a new event descriptor in the table, addresses the right pointer to the fragment and decreases the corresponding missing source counter.

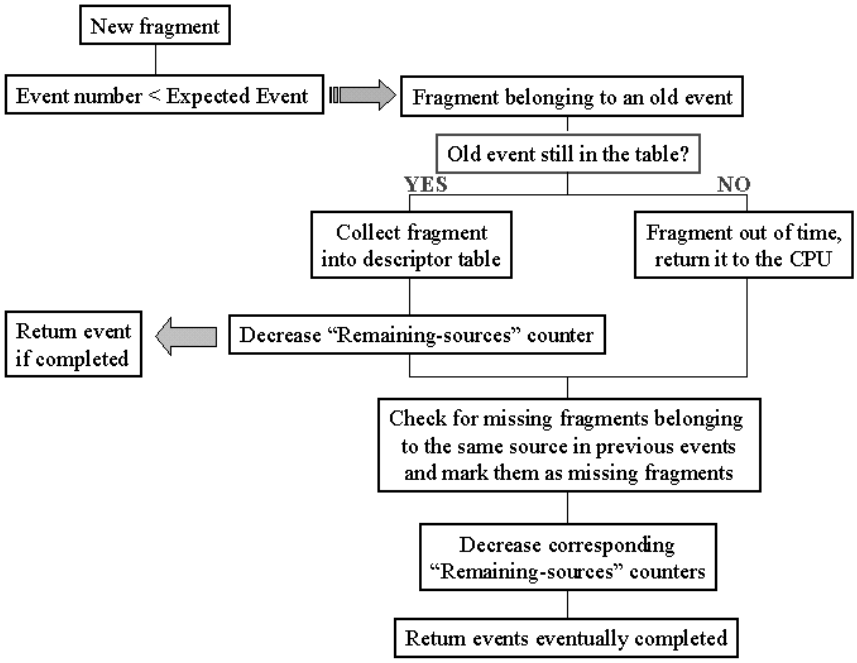

Figure 5.10: Simple time-out: Case 1

After the new event fragment management, the algorithm checks again if there is any event in the descriptor table with a number smaller than the one of the event just allocated, that has not received any data from the same source the last fragment comes from. If so, the algorithm acts as described in the previous point, marking those fragments of such events as "missing fragments" and decreasing the source counter of the associate event descriptors. It also checks if there are events which have been completed after the source counter updating and, eventually, it returns them to the processor, freeing the corresponding descriptors in the table. Furthermore, the algorithm decreases the timestamp of all the events in the table (except the one of the last event inserted), because a new event has arrived. At the same time, while it does the times-

tamp decrease, the algorithm checks if any event in the descriptor table is in time-out. If this is the case, the algorithm marks those events as "event in time-out", gives them to the processor for the physics analysis and frees the corresponding space in the descriptor table. At the end the algorithm updates the value of the next expected event value to the next event number that should come.

Basic ideas are are depicted in figure 5.11

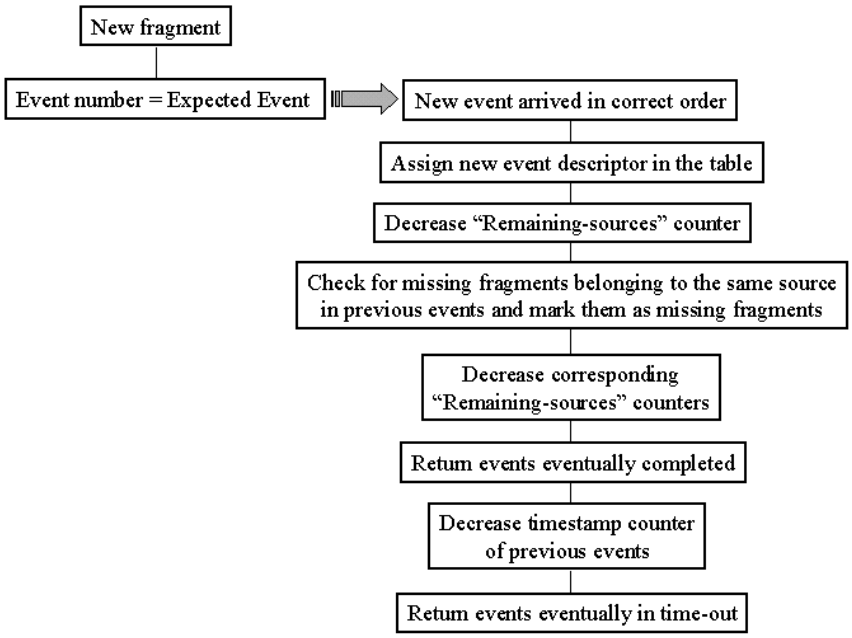

Figure 5.11: Simple time-out: Case 2

3. *The event number of the new fragment is greater than the expected event number*

   In this case the fragment arrived belongs again to a new event and the algorithm does the same steps already described before. The only difference is that the new event allocated in the descriptor

table is not the the expected one, but it is bigger. This means that there are some missing events whose fragments will arrive later, that will not certainly receive any fragment from the source which has sent the last fragment.

The algorithm of course takes note of this and allocates not only the descriptor for the event arrived with the new fragment but it prepares also the descriptors for the missing events which will come. In these additional descriptors it decreases immediately the source counter and marks as "lost fragment" the pointer associated with the source which has sent the last fragment.

All the other steps, then, made by the algorithm are equal to those made in the previous case. Figure 5.12 shows the complete scheme of this case.

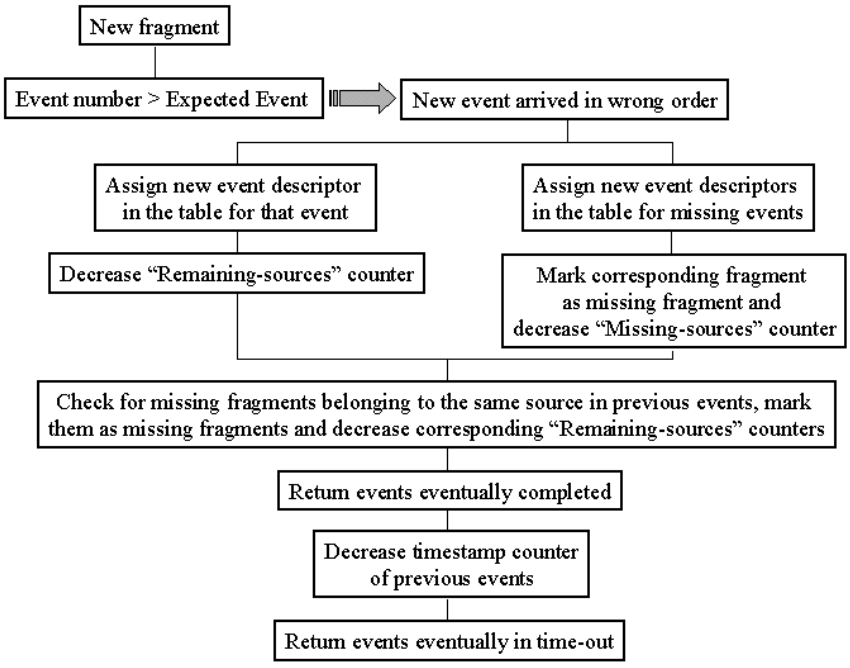

Figure 5.12: Simple time-out: Case 3

From the description just given of our simple time-out algorithm is evident that the time-out implemented in this work is not really based on time, but is calculated on the number of events received by the destination. This means that, if the time-out is set to j, when the first fragment of the event $n + j$ arrives, the event number n (if incomplete) is considered to be in time-out. With this solution the descriptor table will never exceed its own maximum size which corresponds to a maximum number of events equals to the time-out value.

5.2.4 Automatic Adjustment

The *automatic adjustment* algorithm performs the same functionalities as the implicit + time-out algorithm, but it also pays attention, at the run time, on the source state and adapts its behaviour accordingly. The number of sources remains a parameter, which has to be specified at the beginning, and the whole system is dimensioned to this parameter value. In this algorithm, the destination keeps also a variable with the number of sources, which are considered being alive and active at any time of the run. At the start up, usually, all sources are supposed to be up and running, and so generally this variable is set equal to the maximum number sources (this is not compulsory and the variable can be set equal to any value smaller than the real number of sources. The destination algorithm will then adjust it to the right value at the run time). The destination also keeps an array with all the source identifiers and their respective states: 1 = active, 0 = crashed. At the beginning, all source states are usually set to 1 (see figure 5.13).

When a fragment arrives, the number of the source sending the fragment is checked and it is compared with the corresponding source state in the array. If the source was an active source even before the fragment arrival, nothing changes and the algorithm does the same steps we have described in the previous subsection.

Otherwise, if from the source state array it turns out that the source sending the fragment was a non-active source before the last fragment, the source state is immediately turned from crashed into active and the global variable of current active sources is updated. The source is then

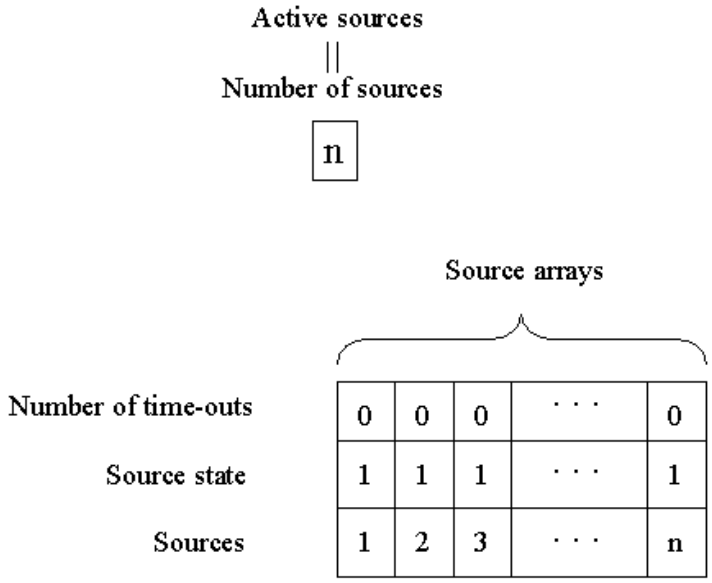

Figure 5.13: Start up

considered active from that event number value and all the other events, with an identification number bigger than the event number of the last fragment, are supposed to receive a fragment from that source, even if they are already in the descriptor table and their building is already started (see figure 5.14).

Then, when a new event fragment arrives, a new event descriptor is allocated in the table. Before updating the remaining source field in the descriptor header, the algorithm lists the state array and checks which sources are active and which ones are crashed (see figure 5.15). Each time it finds a dead source, it immediately marks the associate fragment pointer in the descriptor list as "lost fragment" and does not wait any more for a fragment from that source for that event. Then the algorithm updates the remaining sources field in the descriptor header with the real current active source value.

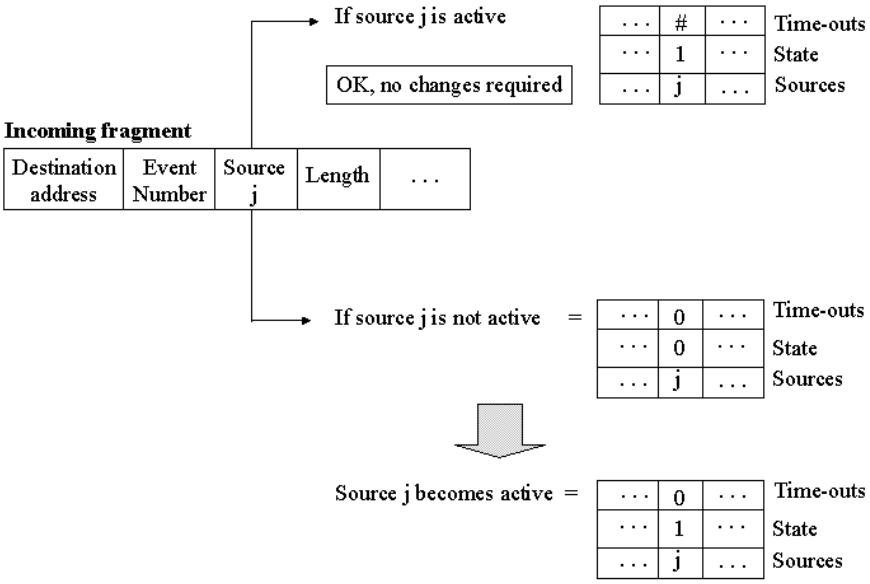

Figure 5.14: New fragment

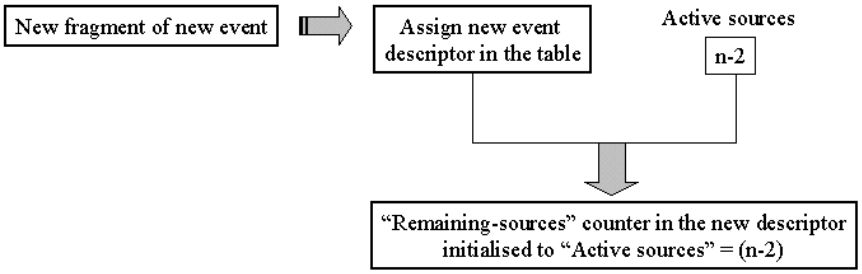

Figure 5.15: Remaining sources counter updating when a new event fragment arrives

Finally, when an event is completed or is returned because it is in time-out, the algorithm checks whether or not some fragments are missing. If so, it controls which sources have not sent the fragment and increases the counters relative to those sources. These counters (one for each source) are kept in the source state array and are initialised at zero. They are increased by one when an event with one or more missing fragments belonging to the associate source is found. Whenever a counter reaches a certain numeric value (stated at the beginning, of course), the corresponding source is considered crashed and its state is modified from active, 1, into non-active, 0, and the global variable with the current active source number is updated. The dead sources are then supposed to be down until they send another event fragment (see figure 5.16). As soon as a fragment is received form a dead source, the destination algorithm changes the source state into active and sets the corresponding counter to zero.

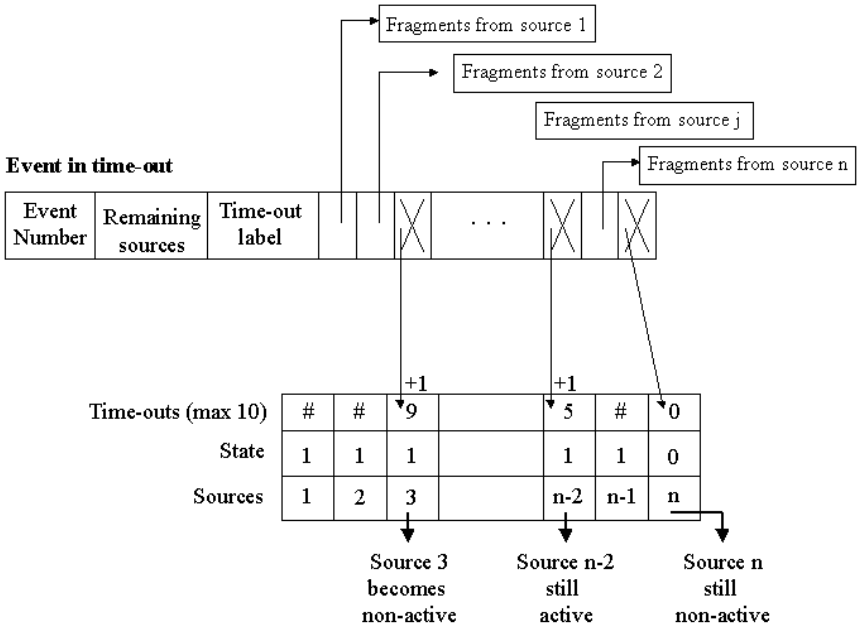

Figure 5.16: Effects of events in time-out on source arrays

We can recapitulate the way of working of the *automatic adjustment* algorithm remembering that it accomplishes the same functionalities as the *implicit + time-out* algorithm and it also adapts its actions on the source states, according to the following assumptions:

- After the completion of n events which present a missing fragment from the same source, the algorithm considers that source dead and does not wait any more fragment from that source for the next events;
- As soon as a fragment belonging to a dead source arrives, the algorithm considers that source active again and starts waiting for fragments coming from that source for next events.

5.3 Performance on a PC

In order to have a general idea about the performance of our event building code, we have done some measurements on a common desktop computer. The machine used for these measurements work was a Pentium II with a 400 MHz Intel processor inside. The operating system running on the computer was Windows NT and the programming environment used was Visual C 5.0.

The steps we have done in our performance measurement work are the following:

1. Generation of a fixed number of fragments;
2. Time measurements on the whole program running with the fixed number of fragments;
3. Switching off the event building functions;
4. Time measurements on the program without the event building code but running with same number of fragments as before;
5. Computation of the event building time overhead by subtracting the second time measurement from the first one.

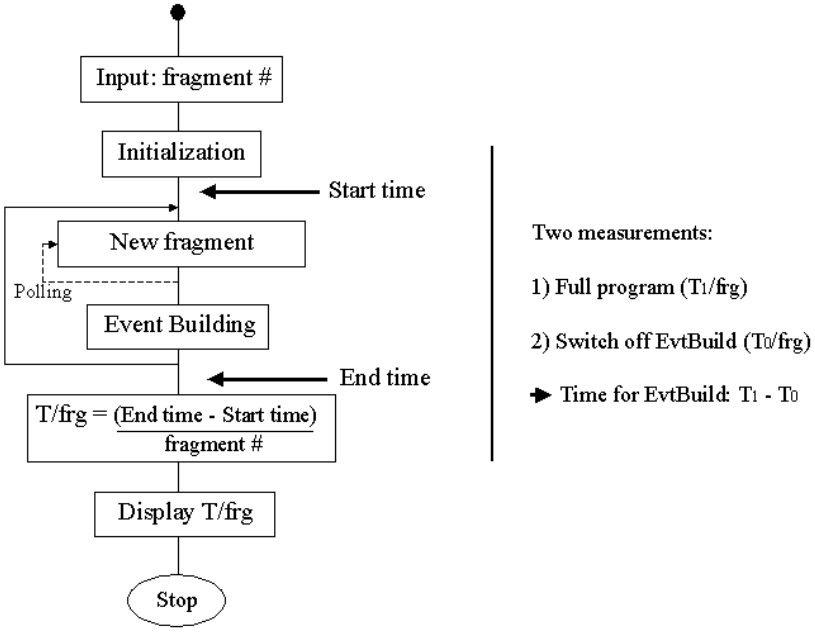

Figure 5.17: Measurement of event building overhead

Figure 5.17 shows this procedure in detail.

We have done the same tests for all the three different types of algorithms, running each time the three protocols with the same (increasing) amount of fragments, so that we could draw conclusions for the different protocols, analysed under equal conditions. Furthermore, in order to obtain valid results, we have had to generate big numbers of fragments each time we run the tests. For small numbers of fragments, in fact, the processor computation was too fast, and the time function was not so precise to be able to really measure very short intervals of time.

Finally, we did not send real data, but we sent fragments with empty payloads because at the moment with are interested in measuring only the event building software overhead.

Of course, this type of measurements we have realized are not completely realistic but show pure software protocol overheads. In real sys-

tems, in fact, we usually register bigger wastes of time due to interrupt handling, data copies, operating system, and so on. However, these results are quite important because they measure the real software overhead. Considering the fact that we have limited time allocated for the management of each single fragment, we can estimate from these measurements how much time the software needs for its computation and to what extent we can make it more elaborate.

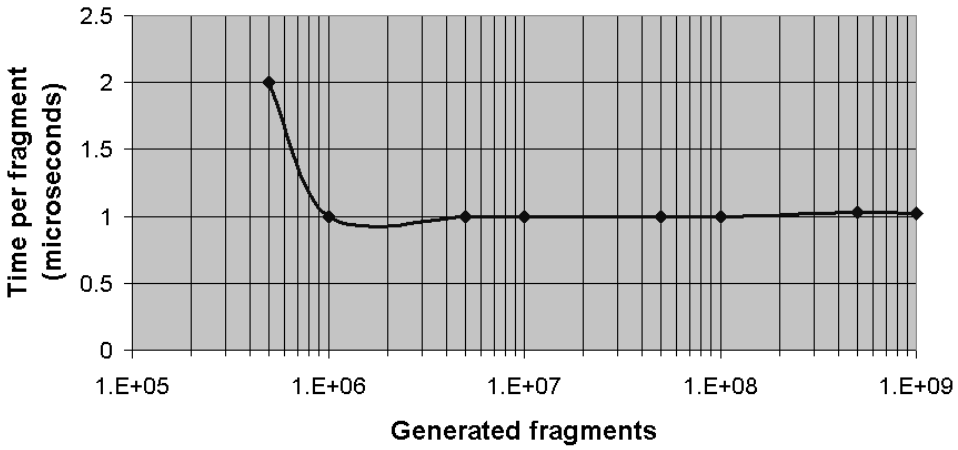

Figure 5.18: Performance measurements of no time-out algorithm (in logarithmic scale)

The first results reported here (shown in figure 5.18) are the performance results of the *no time-out* protocol. The algorithm seems to be quite good and fast (an average overhead of 1 microsecond per fragment) and this is what we would expect. The algorithm is really simple and it does few operations per cycle. It expects that all fragments of all events arrive for sure and in the simulation we have done there are not exceptions, of course.

As we can see from the graph, there is an initial value which is quite bigger (by a factor of 2) than the others. Unfortunately, this is due to the time function that is not accurate enough for these kind of measurements.

By increasing the number of fragments, the algorithm becomes more stable and the measurement results are definitely good. With an overhead of 1 microsecond per fragment, the safety margin of this protocol is quite big and so it can be reasonably made more elaborate in order to fulfil a long set of functionalities and thus make the event building protocol more reliable and complete.

The next results, shown in figures 5.19 and, 5.20 are the performance results of the *simple time-out* protocol—which is, by the way, the most interesting and the preferred candidate (from the functionality point of view) for the final choice.

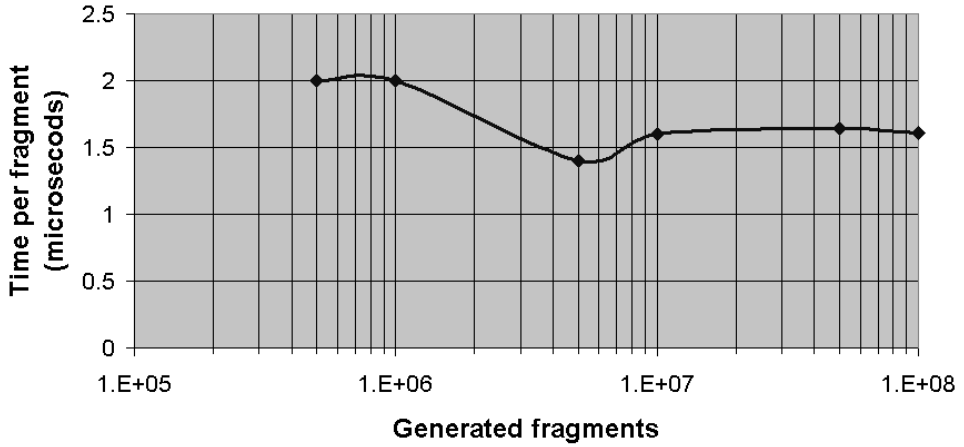

Figure 5.19: Performance measurements of simple time-out algorithm (in logarithmic scale)

This algorithm implements a time-out procedure and handles possible losses of fragments. For this reason we have done two different types of measurements: in the first test (whose results are shown by graph 5.19) all the fragments generated reach the destination, in the second one, instead, (whose results, compared with the previous ones, are shown by graph 5.20) there is a small percentage of fragments randomly lost.

In both cases, the first results obtained are not completely reliable,

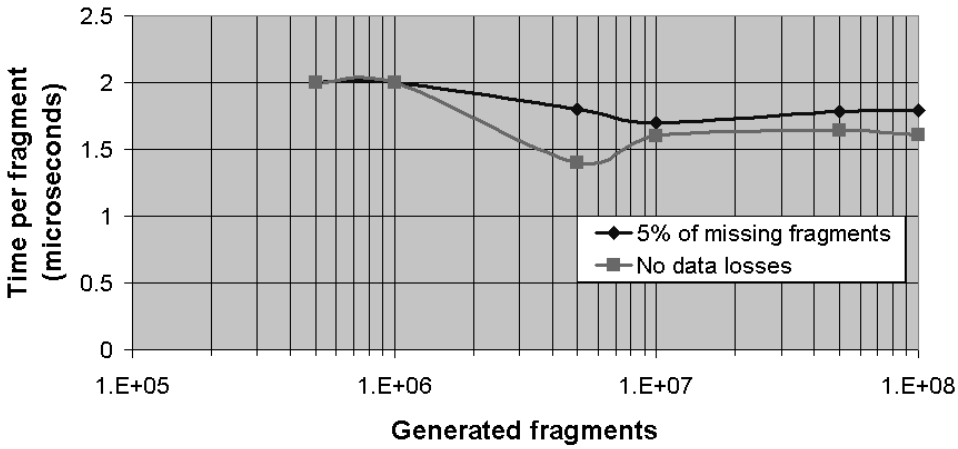

Figure 5.20: Performance measurements of simple time-out algorithm (in logarithmic scale) with data losses

because of the low accuracy of the time function. We can notice, anyway, that, after the initial values, the algorithm behaviour becomes stable and reliable.

Furthermore, as we can see from figure 5.20, in the second test, the protocol is a little bit slower than in the first one, but this is expected. In fact, with real missing fragments, the receiving algorithm must do more controls and checks and these make it slower, of course.

By the way, in both cases, the results are quite good and comfortable. An average, included between 1.5 and 2 microseconds, of overhead per fragment gives a good performance to the protocol implemented.

Finally we relate the last performance results, the ones of the *automatic adjustment* protocol. This protocol implements a time-out procedure, like the previous one, but also adapts its steps according to the source states. Because of the time-out and the source state adjustment, we have done two types of tests: in the first one, there are no missing data losses and no dead source, in the second test, instead, there are 5% of lost fragments and, consequently, some sources could crash and then

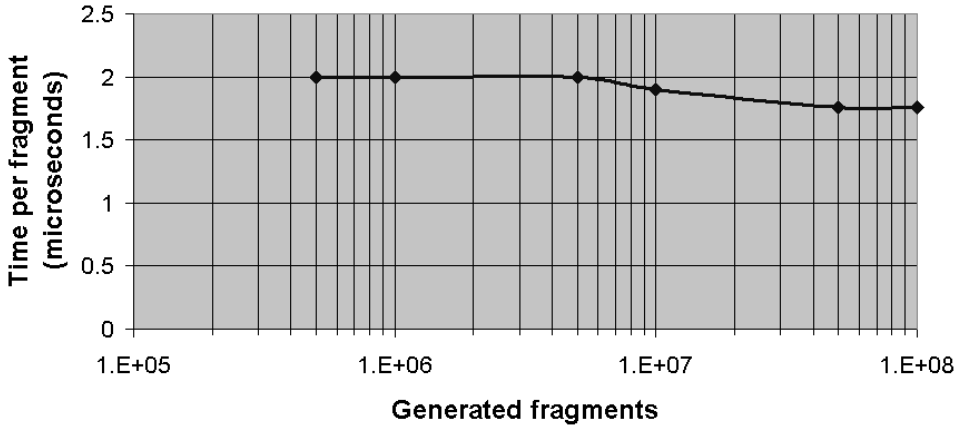

Figure 5.21: Performance measurements of automatic adjustment algorithm (in logarithmic scale)

become active again.

Figure 5.21 shows the results of the first type of measurements and figure 5.22 the results of the second measurements (compared with the first ones).

The receiving algorithm of this protocol is a little bit slower than the simple time-out one, because it has some more controls to do and some more functions to perform.

Nevertheless, here we can draw the same conclusions we gave in that case. The second test, in fact, has a bigger overhead time than the first one (as figure 5.22 nicely confirms), because with real missing fragments the receiving algorithm is forced to check more conditions and so wastes more time in its computation. The results of both tests are quite good (an average of almost 2 microseconds per fragment), except few of them due to the inaccurate time function.

We can deduce that all three protocols implemented gave reasonable performance. Of course, the other standard overheads (due to the operating system, interrupt handling, an so on) must be taken into account, but we are allowed to assume that our safety margin of about 25 microseconds

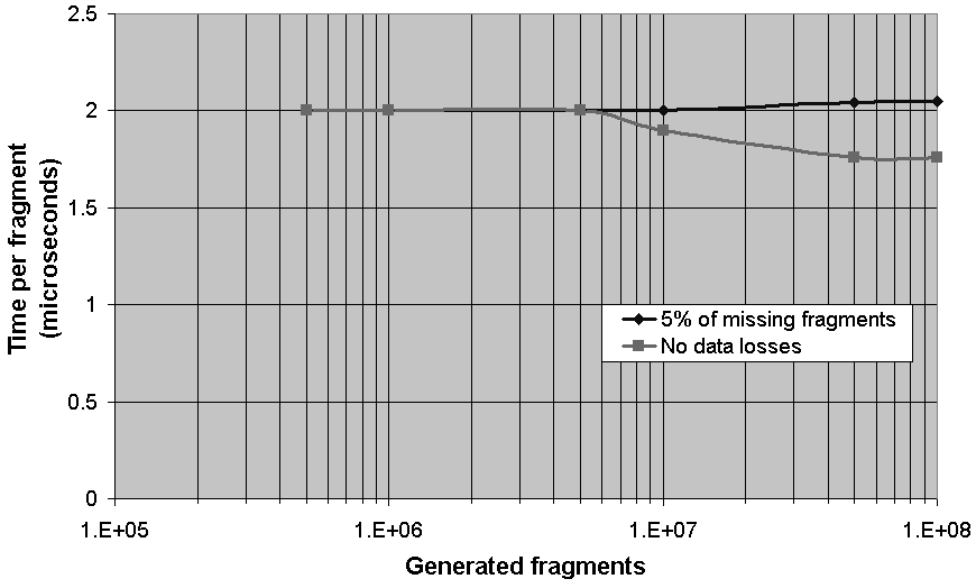

Figure 5.22: Performance measurements of automatic adjustment algorithm (in logarithmic scale), with data losses

per fragment is well respected, even with the addition of those overheads.

Further, in next chapters, we will discuss the improvements which should be expected by implementing event building in a embedded system.

Chapter 6

Embedded Event Building

One of the more surprising developments of the last few decades has been the ascendancy of computers to a position of prevalence in human affairs. Today there are more computers in our houses and offices than there are people who live and work in them. Yet many of these of these computers are not recognised and known as such by their users: this is specially the case of embedded systems.

In this chapter, we will describe first what embedded systems are, where they are found and why they are useful. Later on, we will introduce the justification to the main and new idea of our work: the *embedded event building*. After the description of the event building protocol and its implementations, seen in the previous chapter, we will explain here our platform choice, the related motivations which support it and the results we are going to expect.

6.1 What is an Embedded System?

An embedded system is a combination of computer hardware and software, and additional parts electronics or mechanics, designed to perform a specific function. A good example is the microwave oven. Almost every household has one, and tens of millions of them are used everyday, but very few people realize that a processor and software are involved in the preparation of their lunch or dinner.

This is in direct contrast to the personal computer in the family room. It is also comprised of computer hardware and software and mechanical components (disk, drives, and so on). However, a personal computer is not designed to perform a specific function. Rather it is able to do many different kind of things. A lot of people use the term *general-purpose computer* to make this distinction clear. As shipped, a general-purpose computer is a black box, the manufacturer does not know what the costumer will do with it. One costumer may use it as a network server, another may use it exclusively for writing documents and a third may use it to for playing games.

Frequently an embedded system is a component within some larger system. For example, modern cars and trucks contain many embedded systems. One embedded system controls the anti-lock brakes, another monitors and controls the vehicle's emissions and a third displays information on the dashboard. In some cases, these embedded systems are connected via some some sort of communications network, but this not a requirement, of course.

It is important to point out that a general purpose computer is itself made up of numerous embedded systems. For example, a computer consists of a keyboard, a mouse, a video card, a modem, a hard disk, a floppy disk and a sound card—each of which is an embedded system. Each of these devices, in fact, contains a processor and software, which is hardwierd, and is designed to perform a specific function.

If an embedded system is designed well, the existence of the processor and software could be completely unnoticed by the user of a device. Such is the case for a microwave oven, a VCR or an alarm clock. In some cases, it would be possible to build an equivalent device that does not contain the processor and the software. This could be done by replacing the combination with a custom integrated circuit that performs the same functions in hardware. However, a lot of flexibility is lost when a design is hard-coded in this way. It is much easier, and cheaper, to change a few lines of software than to redesign a piece of custom hardware.

6.1.1 History and future

According to the definition of embedded systems given earlier in this chapter, the first such systems could not possibly have appeared before 1971. That was the year Intel introduced the first microprocessor of the world. This chip, the 4004, was designed for being used in a line of business calculators produced by the Japanese company Busicom. In 1969, Busicom asked Intel to design a set of custom integrated circuits—one for each of their new calculator models. The 4004 was Intel's response. Rather than designing custom hardware for for each calculator, Intel proposed a general-purpose circuit that could be used throughout the entire line of calculator models. This general-purpose processor was designed to read and execute a set of instructions—software—stored in an external memory chip. Intel's idea was that the software would give each calculator its unique set of features. The microprocessor was an overnight success and its use increased steadily over the next decade. Early embedded applications included unmanned space probes, computerised traffic lights and aircraft flight control systems. In the 1980s, embedded systems quietly rode the waves of the microcomputer age and brought microprocessor into every part of our personal and professional lives. Many of the electronic devices in our kitchens (bread machines, food processors and microwave ovens), living rooms (televisions, stereos and remote controls) and workplaces (fax machine, pagers, laser printers, cash registers and credit card readers) are embedded systems.

It seems inevitable that the number of embedded systems will continue to increase rapidly. Already there are producing and selling new embedded devices that have enormous market potential: light switches and thermostats that can be controlled by a central computer, intelligent air-bag systems that do not inflate when children or small adults are present, palm-size electronic organisers and personal digital assistance (PDAs), digital cameras and dashboard navigation systems.

6.1.2 Real-Time Systems

A well known and studied subclass of embedded systems is the set of the real-time systems. As commonly defined, a *real-time system* is a

computer system that has timing constraints. In other words, a real time system is partly specified in terms of its ability to make certain calculations or decisions in a timer manner. These important calculations are said to have deadlines for completion. And, for all practical purposes, a missed deadline is just as bad as a wrong answer.

The issue of what happens if a deadline is missed is a crucial one. For example, if the real-time system is part of an airplane flight control system, it is possible that the lives of the passengers and crew are being endangered by a single missed deadline. However, if the system is involved instead in satellite communication, the damage could be limited to a single corrupt packet. The more severe the consequences, the more likely it will be said that the deadline is "hard" and, thus, the system a hard real-time system. Real-time systems at the other end of this continuum are said to have "soft" deadlines.

All the topics relative to embedded systems are applicable to the designers of real-time systems. However, the designer of a real time system must be more diligent in his work. He must guarantee reliable operation of the software and hardware under all possible conditions.

6.2 Role of Embedded Processors

Unlike software designed for general-purpose computers, embedded software cannot usually be run on other embedded systems without significant modification. This is mainly because of the incredible variety in the underlying hardware. The hardware in each embedded system is tailored specifically to the application, in order to keep system costs low. As a result, unnecessary circuitry is eliminated and hardware resources are shared wherever possible. In this section we will summarise what hardware features are common across all embedded systems and we will give some examples of their usual, commercial application.

By definition all embedded systems contain a processor and software, but what other feature do they have in common? Certainly, in order to have software, there must be a place to store the executable code and temporary storage for runtime data manipulation. These take the form

of ROM and RAM, respectively; any embedded system will have some of each. If only a small amount of memory is required, it might be contained within the same chip as the processor. Otherwise, one or both types of memory will reside in external memory chips.

All embedded systems also contain some type of inputs and outputs. For example, in a microwave oven the inputs are the buttons on the front panel and a temperature probe, and the outputs are the human-readable display and the microwave radiations. It is almost always the case that the outputs of an embedded system are a function of its input and several other factors (elapsed time, current temperature, and so on). The inputs to the system usually take the form of sensors and probes, communication signals, or control knobs and buttons. The outputs are typically displays, communication signals or changes to the physical world. Figure 6.1 shows a general example of an embedded system.

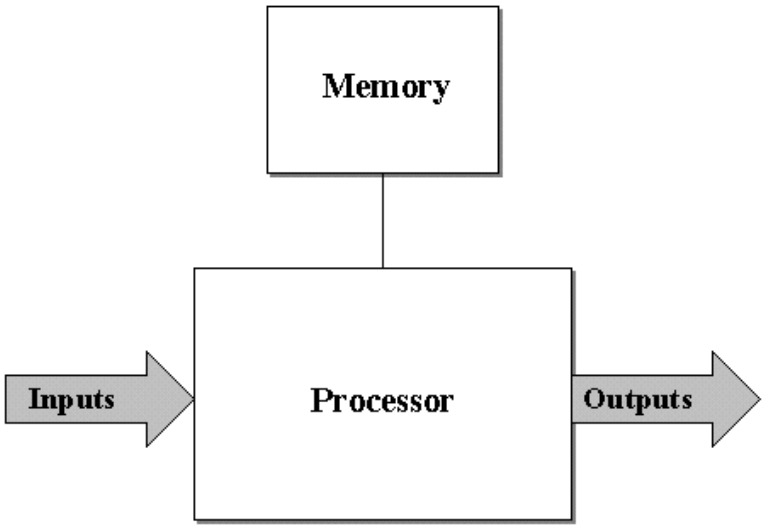

Figure 6.1: Generic embedded system

With the exception of these few common features, the rest of the embedded hardware is usually unique. This variation is the result of many competing design criteria, Each system must meet a completely different set of requirements, any all of which can affect the compromises and tradeoffs made during the development of the product. For example, if the system must have a production cost of less than $10, then other things—like processing power—might need to be sacrified in order to meet that goal.

Of course, production cost is only one of the possible constraints under which embedded hardware designers work. Other common design requirements include the following aspects:

- *Processing power*

  The amount of processing power necessary to get the job done. A common way to compare processing power is the MIPS (millon of istructions per second) rating. if two processors have rating of 25 MIPS and 40 MIPS, the latter is said to be the more powerful of the two, of course. However, other important features of the processor need to be considered. One of these is the register width, which typically ranges from 8 to 64 bits. Today, general-purpose computers use 32- and 64-bit processor exclusively, but embedded system are still built sometimes with older and less costly 8- and 16-bit processors.

- *Memory*

  The amount of memory (ROM and RAM) required to hold the executable software and the data it manipulates. Here the hardware designer must usually make his best estimate up front and be prepared to increase or decrease the actual amount as the software is being developed. The amount of memory required can also affect the processor selection. In general, the register width required establishes the upper limit of the amount of memory it can access (for example, an 8-bit address register can select one of only 256 unique memory locations [1]).

[1] Of course, the smaller the register width, the more likely it is that the processor

- *Development cost*

  The cost of the hardware and software design processes. This is a fixed, one-time cost, so it might be that money is no object (usually for high-volume products) or that is the only accurate measure of system cost (in the case of a small number of units produced).

- *Number of units*

  The tradeoff between production cost and development cost is affected most by the number of units expected to be produced and sold.

- *Expected lifetime*

  The time the system will continue to function. This affects all sorts of design decisions, from the selection of hardware components to how much the system may cost to develop and produce.

- *Reliability*

  The reliability the final product must have. If it is a toy for children, it does not have to work always correctly, but if it is part of a space shuttle or a car, it has to do correctly its job each time it is requested.

In addition to these general requirements, there are some detailed functional requirements of the system itself. Those are the things which give the embedded system its unique identify as a microwave oven, a pacemaker or a printer.

In order to demonstrate the variation from one embedded system to another and their different commercial applications, we will give in next subsection a brief description of three embedded systems.

6.2.1 Video Game Player

When one pulls the Nintendo-64 or Sony Playstation out from its entertainment center, he or she is preparing to use an embedded system.

employs tricks like multiple address spaces to support more memory.

In some cases, these machine are more powerful than the comparable generation of personal computers. Yet video games players for the home market are relative inexpensive compared to personal computers.

The companies that produce video game players do not usually care how much it costs to develop the system, so long as the production costs of the resulting product are low—typically around a hundred dollars. They might even encourage their engineers to design custom processors at a development cost of hundreds of thousand of dollars each. So, although there might be a 64-bit processor inside a video game player, it is not necessary the same type of processor that would be found in a 64-bit personal computer. In all likelihood, the processor is highly specialized for the demands of the video games is intended to play.

Because production cost is so crucial in the video game market, the designers also use tricks to shift the costs around. For example, one common tactic is to move as much of the memory and other peripheral electronics as possible off of the main circuit board and onto the game cartridges. This helps to reduce the cost of the game player, but increases the price of each and every game. So while the system might have a powerful 64-bit processor, it might have only a few megabytes of memory on the main circuit board. This is just enough memory to bootstrap the machine to a state from which it can access additional memory on the game cartridge.

6.2.2 Digital Watch

At the end of the evolutionary path that began with sundials, water clock and hourglasses is the digital watch. Among its many features are the presentation of the date and time (usually to the nearest second), the measurements of the length of an event to the nearest hundredth of a second. As it turns out, these are very simple tasks that do not require very much processing power of memory. In fact, the only reason to employ a processor at all is to support a range of models and features from a single hardware design.

The typical digital watch contains a simple, inexpensive 8-bit processor. Because such small processors cannot address very much memory,

this type of processor usually contains its own on-chip ROM. And, if there are sufficient registers available, this application may not require any RAM. In fact, all of the electronics—processor, memory, counters and real-time clocks—are likely to be implemented on a single chip. The only other hardware elements of the watch are the inputs (buttons) and outputs (LCD and speaker).

The watch designer's goal is to create a reasonably reliable product that has an extraordinary low production cost. If, after production, some watches are found to keep more reliable time than most others, they can be sold under a brand name with a higher markup. Otherwise, a profit can still be made by selling the watch through a discount sales channel.

6.2.3 Mars Explorer

Recently, NASA launched the Pathfinder mission. Its pimary goal was to demonstrate the feasibility of getting to Mars on a budget. It was actually two embedded systems: a landing craft and a rover. The landing craft had 32-bit processor and 128 MByte of RAM; the rover, on the other hand, had only an 8-bit processor and 512 kByte. These choices reflect the different functional requirements of the two systems.

6.2.4 Conclusions

Embedded systems are becoming very common: they are present everywhere, in our daily life as well as in big experiments of high technology resolution, and the previous subsections state quite well these various aspects. Naturally, embedded systems find their working and developing environment in the computer science world, where the research field is moving further and further in this direction and new ideas and solutions are coming at any time. Nowadays we can count on the presence of embedded systems in many places for improving our computer performance. Embedded systems are in drivers, printers, network cards and so on. The last generation products are the credit card PCs: a complete and working computer with the dimensions of a credit card.

In consequence of this great development which seems set to continue

and grow, it has been decided in our working environment to undertake a new guide line which explores and utilises this type of technology, completely unfamiliar to the event building strategies. Our aim is to investigate and analyse the scope of this technology for suggesting at the end a new solution which possibly improves the performance of the event building implementation, according to the considerations described in the folllowing section.

6.3 Embedded Event Building Justification

The idea of exploiting the capabilities of embedded systems for the event building needs has come out of several considerations, that we will try to explain in detail in this section. These considerations have convinced us to concentrate on the embedded system solution but, of course, we cannot be sure that our work will be the best answer to the event building requirements and will be really adopted in 2005, when the LHCb experiment will start running. By the way, the following considerations remain valid and need solutions, regardless of the results presented in this work.

First of all, it should be noticed that with the LHCb expected rate of 40 kHz and an aggregate bandwidth of 4 GByte there is no guarantee that a farm of general-purpose CPUs can be sufficient and fast enough, because of the time overheads due to the interrupt handling. Interrupts, in fact, have a great impact on the CPU performance, specially when computation time is very limited and restricted, like in our case. If we have a look at the technology improvements during last few years, we can immediately notice that, while the CPU power has increased by a significant factor, the interrupt latency has gone down in a less evident way. Even if the time to handle an interrupt has decreased by some factor, interrupt latency is still high and it is not comparable with the recent processor power improvements, as shown in table 6.1 [4].

Moreover, we should notice that general-purpose CPUs usually rely on cache for their computation. Interrupts tend to be "cache-breakings", i.e. tend to end-up in flushing cache locations, which , in turn, means that these locations have to be refilled from memory. This is a slow process,

Interrupt Latency	Scheduling Latency	Processor
7.54 μs	12.57 μs	33 MHz 486
1.84 μs	4.73 μs	100 MHz Pentium
1.38 μs	2.93 μs	200 MHz Pentium

Table 6.1: Interrupt latency and scheduling latency versus CPU power

during which the processor is basically idle [11]. A typical risk is then to run out of cache. When this happens, all the system performances goes down significantly. We are not in a position to stand this risk and so it is essential for us to find out a solution which avoids this problem.

We could invest a lot in CPU power but this would be a big and expensive investment.

However, also all the TCP/IP functionalities use a lot of the CPU power. Thus, there is a strong possibility that computer industry will try to down-load in the future the TCP/IP stuff inside *network interface cards* (NICs), in order to make the CPU load lighter [6]. "Smart" NICs, with processor power inside, are the new technology challenge.

Therefore, if future NICs are smart enough, in the sense that they have enough processor power, we will be able to do event building inside them, exploiting their primitive environment, free of all the latencies which are an unavoidable feature of standard operating systems.

With embedded event building, only complete events will be given to the the CPU and so the number of interrupts will be reduced by a significant factor (one interrupt per event, instead of one interrupt per fragment).

Of course we perfectly know that there are some risks which must be taken into account. First of all, smart NIC commerce might not be the real investment of the industry in the future. For example, at the moment, it is becoming popular among the networking industries the idea of implementing IP functionalities directly in hardware, without using a processor inside NICs. If this solution spreads, it will not be useful for us. Our need, in fact, is the presence of a general-purpose CPU inside the NIC, for doing event building there.

Anyway, even if smart NICs do not succeed, we will need them, at least from the source point of view. Destinations can be realized in software way, by using general-purpose PCs, but RUs are completely hardware. There we need smart NICs because a standard NIC cannot be easily driven by an FPGA, which is immediately before network interface in the RU setup (as shown in figure 6.2).

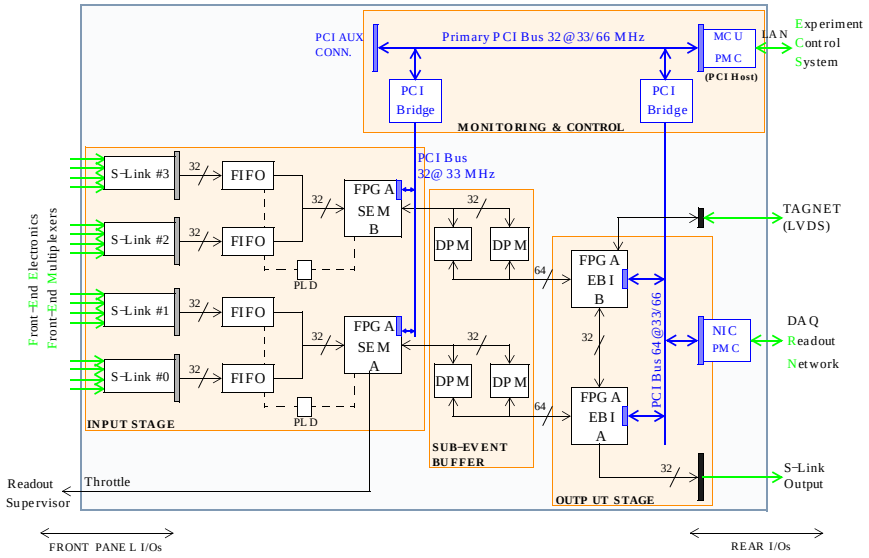

Figure 6.2: Architecture of the RU model

If smart NICs die, the only possibility will be to build the RU network interface with a standard NIC, driven by a credit card PC. By the way, the smart NICs actually in commerce are powerful enough to be used in the RU unit model. The next chapter will describe the hardware/software definition of the smart NIC we have studied: the Tigon 2 by Alteon *Web*Systems.

Chapter 7

A Gigabit Ethernet smart NIC: Tigon 2

Tigon 2 is a "smart" NIC produced by Alteon *Web*Systems. It supports 10/100/1000 Mbps transmissions and fully complies with the IEEE 802.3 specifications and updates. It seems to be a good candidate for the embedded event building project. It has been considered and studied also by the ATLAS DAQ group, which does not have the embedded event building aim, but has appreciated its nice performance as network programmable interface.

In this chapter we will introduce the Tigon 2 smart NIC and its features. First of all, we will give an overview of the hardware architecture, placing emphasis in particular on the interface between the NIC and the network. We will carefully go through the sending/receiving structures and routines. They are important features for our work and we will need to know them quite well. At some point, in fact, we should modify those routines to performs our project functionalities.

After the hardware overview, the NIC software description will follow and the way it interfaces and communicates with the host [1].

Finally we will report some evaluations on the Tigon performances.

[1]Further and in-depth information regarding the Tigon PCI/Gigabit Ethernet can be found in the Alteon web site [7] and in the two papers which describe the hardware/software characteristics of the smart NIC, [5] and [8].

7.1 Tigon 2: Architecture

The Tigon PCI/Gigabit Ethernet Network Interface Card is designed to be a high integration and high performance Gigabit Ethernet ASIC used in adapters for workstations and computers. This ASIC is compliant with the PCI Local Bus Specification Revision 2.1. The Tigon contains the DMA function necessary to off load either the host or the adapter from having to move data between the different memory spaces. A block diagram of the Tigon chip is shown by figure 7.1.

As data is transferred between the two interfaces it is buffered in a FIFO. This allows for buffering and proper synchronisation of the data to the output interface. Internally there are two FIFOs, one dedicated to each direction.

The design fully supports automatic handling of misaligned transfers. This means that the data to be moved can be transferred between any two buffers regardless of the alignment of either buffer. For implementations which support the ability to DMA the data directly into the application's memory spaces, this feature might prevent the host from having to make unnecessary data copies. Internally, data is aligned to the FIFO boundaries.

A TCP/IP style checksum is calculated on all data that is transferred through the Tigon FIFOs. Any implementation in which the host supports hardware checksum assist can benefit from this hardware calculation. The checksum is calculated and automatically stored for each DMA buffer descriptor.

The PCI specification is inherently little-endian. Not all hosts want data presented in the little-endian format. The Tigon contains the ability to do little-endian to big-endian swaps on either 8 Byte or 4 Byte boundaries. It is also possible that the Tigon can be used as a basis for non-PCI interfaces which expect the data to be in big-endian format. This feature allows for the maximum flexibility.

The Tigon also contains a processor to manage the reading and chaining between buffer descriptors. Buffers can be chained to support the "scatter/gather" model of host memory. The code for this processor can be modified in order to optimise the format of the buffer descriptors to

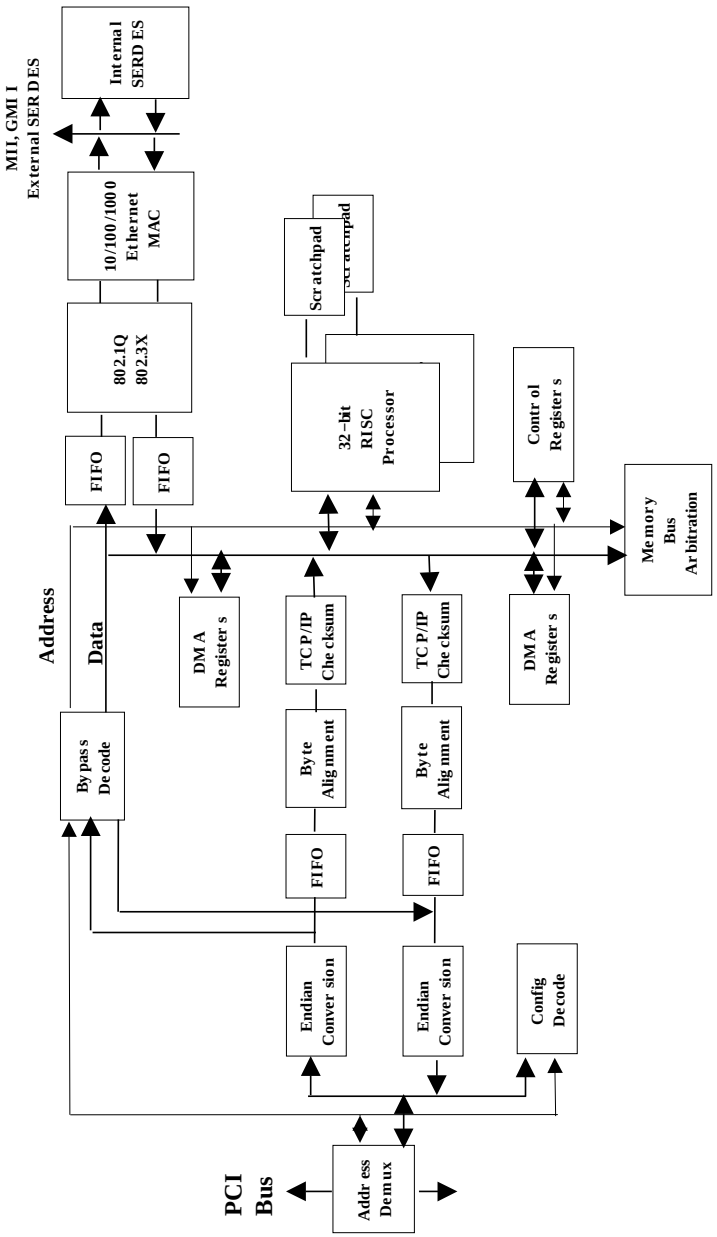

Figure 7.1: Tigon 2 architecture

that which is most efficient for the host.

7.1.1 PCI Supported Features

The PCI specification provides a mechanism for adapters to be uniquely configured depending on the resource available in the host in which they are inserted. In the personal computer market this feature is called "Plug and Play". The Tigon fully supports this configuration process.

At the heart of the configuration process is a maximum 256 Byte region in which the adapter's PCI interface can be controlled. The adapter uniquely identifies itself and indicates to the host what it requires in terms of memory space, I/O space, bus latency, interrupts, and so on. The host in return provides the adapter with the necessary information.

In order to simplify the controller, only configuration registers are controlled by the PCI clock while all other registers are controlled by the Memory Bus clock.

All supported configuration registers are preloaded after a reset from the serial EEPROM. This preserves the adapter configuration information even if the adapter is moved between machines. When necessary, software can update the contents of the EEPROM if the configuration needs to be altered.

Since the Tigon is capable of being a PCI bus master, it participates in the bus arbitration process, whose mechanism is outlined in the PCI specification [32].

7.1.2 Local Memory

The Tigon PCI/Gigabit Ethernet Controller provides a 64-bit high-speed memory bus to connect to local memory. The internal data structure of the Tigon is built around this 64-bit bus width.

The maximum frequency of the memory bus is 50 MHz. This allows for a maximum bandwidth of 800 MBytes/second with synchronous memory and 400 MBytes/second with asynchronous memory. Each access into the SRAM located on the memory bus can be to any address. Therefore no bursting is required by this memory design. The same clock

which controls the local memory is used to control the majority of the Tigon chip; it is considered asynchronous from the PCI bus clock.

Any combination of Byte, half-word or word writes are allowed by the hardware. Reads always supply 64-bits of data from the external memory. The memory sub-system has no knowledge as to whether the useful portion of data is a Byte, a word, and so on.

7.1.3 Internal Processors

The Tigon contains two embedded 32-bit processors, which can be used for any function, including parsing buffer descriptors and controlling the DMA hardware registers. The processors can be set up to accommodate different buffer descriptor format, allowing the Tigon chip to adapt to the communication method which best suits the host. This approach is consistent with the goal of providing a flexible PCI Controller solution.

Fully contained inside the Tigon, the processors are capable of generating operations to the PCI bus as well as to local memory. This allows control information to be located in either memory space. All registers internal to the Tigon can also be accessed by either the embedded processors or by the host through the Tigon's target interface.

The operation of the processors is governed by firmware. External to the Tigon will be a small non-volatile serial EEPROM which will contain all the power on diagnostics and PCI initialisation tasks. Any additional software can be down-loaded by the host driver into the local SRAM memory. Since each local memory operation fetches 8 Bytes, up to two processor instructions can be loaded into the Tigon on each instruction fetch cycle.

An instruction cache as well as internal SRAM is provided to enable the processor to not use any of the valuable local memory bandwidth for most firmware execution. The instruction decoding is modelled after the R4000 RISC processor with some instructions removed and several new ones added for embedded optimisation.

7.1.4 Events

The model with which the embedded processor executes is an event model rather than an interrupt model. This means that the processor is not interrupted by the task at hand, but each task should be kept to a small thread of code. There is a significant hardware assist for event processing within the Tigon to facilitate a robust execution environment.

There is a single 32-bit register in the Tigon which includes bits for each of the events that the processor must handle. The processor main loop is to wait for events and then deal with them in priority order. Two special instructions were created to allow the processor to index into an event jump table based on the most significant (highest priority) bit set in the event register.

Not all events are created by the hardware. There are several bits within the event register which are completely controlled by software. This allows software to queue events to be handled within the priority encoded main loop.

7.1.5 Flash

The Tigon PCI/Gigabit Ethernet Controller has been designed to interface to an external flash memory for storage of important configuration and manufacturing information. All PCI adapters must provide configuration information to the host after power-up. The processor in the Tigon chip initially executes from this flash and loads all internal PCI registers from predetermined locations inside that flash after a reset. This enables the adapter to respond properly throughout the configuration process.

The flash technology allows vital adapter information to be accessed and updated whenever it is necessary. The host or local processor may read or write these locations via the Control Register. The two processors should not attempt to write the flash at the same time.

The flash has additional memory locations which are available for storage of non-PCI information. These locations can be used to store addresses, manufacturing data, diagnostic results, and so on. Software is responsible for managing these locations.

7.1.6 Mailboxes

A common technique to facilitate communication between host processors and adapters, are mailboxes. Typically these are locations which are written by one processor that causes an interrupt to the other processor. The value written may or may not have any significance. The number of mailboxes in each direction is usually fixed. Each processor is allowed to read the mailbox only once it is cleared by the hardware.

The Tigon has generalised the concept of mailboxes in order to provide greater flexibility in the interaction between the host and the adapter.

In order to implement mailboxes in which the value of the mailboxes is not significant, the processor need only to define the software interrupts such that each interrupt has a predetermined meaning.

In order to implement mailboxes in which the value of the mailbox is significant, the adapter can map some of its local memory onto the PCI bus which functions as a mailbox window. This window is accessible by both processors and is predefined to be 1 kByte in size. The second 256 Bytes of this window are divided into 32 8-Byte locations and have a special meaning. A write to the upper half of any of these 32 locations will cause an event to the internal processor with a notification as to which location was written. Actually, there is a separate event bit for each of these 32 locations. The lower 256 Bytes or the upper 512 Bytes (that is, not mailbox area) of this window have no predetermined meaning and can be software defined.

Software should be designed so that it is clear which processor is allowed to write to any given portion of the memory. This technique can be used to create many logical mailboxes of various size without being strictly limited by the hardware. An additional benefit to this technique is that the mailbox may be read as many times as necessary without the loss of the mailbox contents.

The first mailbox has special meaning since the PCI interrupt pin is immediately deasserted when it is written. This feature allows software to define a mechanism for a host processor to clear the interrupt at the same time as passing a message to the Tigon's processor.

7.1.7 Ethernet Transmit Interface

The Tigon's transmit Ethernet interface fully complies with the IEEE 802.3 specifications and is kept up to date with the 802.3z Gigabit Ethernet proposal.

The Tigon uses Ethernet descriptors to keep track of packets being sent to the serial Ethernet interface. The format of these descriptors is fixed in order to allow the hardware to directly reference the fields within the descriptors.

The Ethernet transmit interface is responsible for sending packets to the external network interface by reading the associated transmit descriptor and the packet from the local memory buffer. Error conditions are monitored during the packet transmission and reported to the software through an event at the moment in which they occur. For the transmit interface, errors are considered unusual and are therefore treated as rare events.

Transmit Descriptors

The Ethernet transmit descriptors are set up by software in order to indicate to the transmit hardware where the packets to transmit are located. The descriptors are organised into a ring with a producer and a consumer index. After software initialises the fields within the descriptor and the first doubleword located at the Starting-Address, it updates the producer index indicating to the hardware that it can read the descriptor and establish any connection required. Whenever the producer and the consumer indexes are equal there are no packets to transmit.

Each of the transmit descriptor are eight Bytes in length, as illustrated by figure 7.2. The first 32 bits contain the Byte address at which the packet starts within the external SRAM associated with the Tigon's buffer memory. Since all Ethernet packets must start on a doubleword boundary, the least significant three bits of the address should always be zero. All reserved bit position are not referenced by the Tigon and can be used by firmware to store state information if so desired.

If while sequencing through the buffer reading the packet for transmission, the last word fetched was from the top of the transmitting buffer

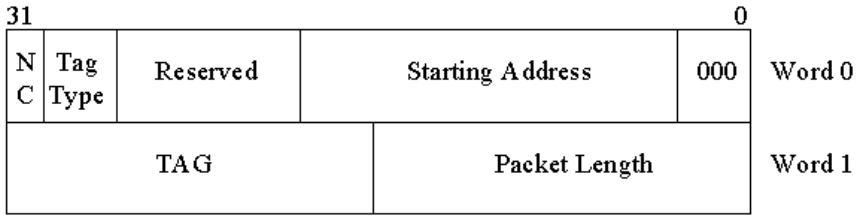

Figure 7.2: Ethernet transmit descriptor

space, the buffer consumer pointer will automatically be set to the transmit buffer address.

The lower 32 bits contain the length of the portion of this packet in Bytes which is described by this descriptor.

Transmit Descriptor Indexes

The Ethernet transmit descriptors are organised in a ring of descriptors located in a consecutive block of memory. The offsets into this block of memory which point to specific descriptors are called "Indexes". The two indexes which are used to maintain transmit descriptors are the producer index and the consumer index. The producer index is updated by the firmware after the respective transmit descriptor fields have been initialised. The consumer index is updated by the hardware after it transmits the packet corresponding to that descriptor. Whenever the two indexes are equal, there are no additional packets to transmit.

Both the transmit producer and consumer indexes are 8 bits in length; therefore there are a total of 256 transmit descriptors respectively. One transmit descriptor must remain unused so that the producer does not overtake the consumer. This means that there are 255 usable transmit descriptors at any time. Separate address decodes are provided which allow software to read or write either of these two indexes. For normal operations however, software would write the producer index and read the consumer index.

To facilitate the generation of internal events when the hardware has completed the transmission of a packet, a reference index register was created inside the hardware. This reference index is the same size as the producer and consumer index. If at any point the reference index is not between the producer and the consumer nor equal to the producer, then an internal event is generated. Software can now set up the reference index to the current packet, knowing that an event will be generated when the hardware advances beyond that packet.

Transmit Buffer Indexes

The Ethernet transmit buffer is organised as a circular buffer ring occupying a consecutive block of memory. The offset into this block of memory which point to specific addresses are called "Indexes". The two indexes that are used to maintain transmit buffer are the transmit buffer producer index and the transmit buffer consumer index. The producer index is updated by firmware to point to the next doubleword after the valid data to be sent. The consumer index is updated by the hardware as it transmits the packet to indicate where to fetch the next transmit data. Whenever the two indexes are equal, the transmit buffer is empty.

Transmit Flow Control

The Ethernet transmit interface supports the 802.3x flow control mechanism in hardware. This feature must first enabled in order to allow the hardware to send such flow control packets. Transmission of a valid 802.3x packet is done based on high and low water marks for number of receive descriptors unused and receive buffer space which is unused.

Once high water mark has been exceeded, an "XOFF" packet will be sent as the next packet. If the "XOFF" length is about to expire another flow control packet will be sent to refresh it up until the low resource drops below the water mark at which point an "XON" message will be sent.

Ethernet CRC Calculation

The Tigon uses standard 32-bit CRC required by the Ethernet specification in all packets. It also uses the same CRC during receive as the hash function for multicast filtering, and to verify the integrity of areas of external non-volatile memory.

7.1.8 Ethernet Receive Interface

The Tigon's Ethernet receive interface fully complies with the IEEE 802.3 specification and is kept up to date with the 802.3z Gigabit Ethernet proposal.

The Tigon uses receive descriptors to keep track of packets being received from the serial Ethernet interface. The format of these descriptors is fixed in order to allow the hardware to directly reference the fields within the descriptors.

The Ethernet receive interface is responsible for accepting packets from the external network interface and placing them in the local memory buffer along with an associate descriptor. All data being received from the Ethernet interface goes through a small synchronising FIFO which synchronises the data to the internal clock frequency.

Error conditions are monitored during the packet reception and reported to the software through the status word located in the Ethernet descriptor. Serialiser/deserialiser integrity errors are always reported directly to the internal processor. They may also be indicated in the final status word if a receive packet was in progress. This allows the processor to monitor the quality of the channel and perform any necessary error reporting to the host.

Packet Structure

The packet structure for received packets in the local buffer memory is shown in figure 7.3. This structure will always start on a doubleword boundary.

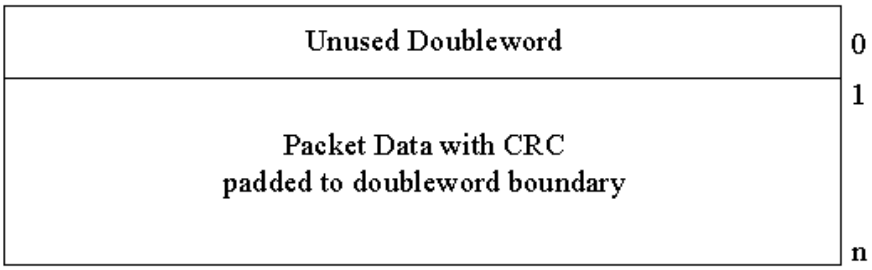

Figure 7.3: Ethernet receive memory layout

Errors

The Ethernet receive interface monitors several indicators during the reception of a packet. Not all of these indicators can be considered fatal errors. In fact in some cases, such as diagnostic, they maybe created intentionally. The hardware makes no attempt to do anything but report the information along with each packet. The only cases in which the packet is aborted happens when the serial Ethernet interface indicates a fatal condition. Errors which occur during a packet reception are reported in the status word filed of the receive descriptor.

Receive Descriptors

The Ethernet receive descriptors are set by hardware in order to indicate to the software where the received packets are located. In a similar way to that used to communicate with the sending part, the descriptors are organised into a ring with a producer and a consumer index. The hardware writes the fields within the descriptor only at the end of the packet. The hardware also updates the producer index indicating that software that it can now start processing the received packet. Whenever the producer and consumer indexes are equal there are no packets in the receive buffer.

Each of the receive descriptors are 8 Bytes in length as illustrated by figure 7.4. All 8 Bytes of the descriptors are written at the end of the

packet being received. The first 32 bits contain the Byte address within the external SRAM associated with the Tigon's receive buffer at which the packet starts. All received packets will be placed in the receive buffer starting doubleword boundaries. All reserved bit positions are not used by the Tigon and will therefore be set to zero by the hardware.

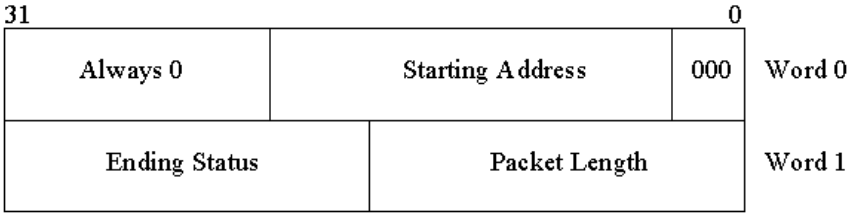

Figure 7.4: Ethernet receive descriptor

Receive Descriptor Indexes

The Ethernet receive descriptors are organised in a ring of descriptors located in a consecutive block of memory. The offsets into this block of memory are called "Indexes". The two indexes which are used to maintain receive descriptors are the producer index and the consumer index. The producer index is updated by hardware after the respective receive descriptor fields have been initialised at the end of a packet. The consumer index is updated by the firmware after it completes the processing on the packet corresponding to that descriptor. Whenever the two indexes are equal, there are no packets in the receive buffer.

Both the receive producer and consumer fields are 9 bits in length; therefore there are a total of 512 receive descriptors. The receive indexes points to either word in the receive descriptor, as follows: the producer index advances to point to the second half of the descriptor when a new packet reception has started. The producer index advances to point to the beginning of the next descriptor when the packet has been completely received. This feature is utilised as a mechanism by the hardware to

indicate the difference between a packet reception in progress and one in which the entire packet has been received.

To facilitate the generation of internal events when the hardware has completed the reception of a packet, a reference index register was created inside the hardware. This reference index has the same size as the producer and consumer indexes. If at any point the reference index is not between the producer and the consumer nor equal to the producer, then an internal event is generated. Software can now set up the reference index to the current packet, knowing that an event will be generated when the hardware advances beyond that packet.

Receive Buffer Indexes

The Ethernet receive buffer is organised as a circular buffer occupying a contiguous block of memory. The offsets into this block of memory which point to specific addresses are called "Indexes". The two indexes which are used to maintain the receive buffer and the receive buffer producer index and the receive buffer consumer index. The producer index is updated by the hardware as it receives the packet to indicate where it will place the next data received. the consumer index is updated by firmware to point to the first doubleword of valid data it has yet to process. Whenever the two indexes are equal, the receive buffer is empty.

Receive Flow Control

The Ethernet receive interface supports the 802.3 flow-control mechanism in hardware. Reception of a valid 802.3x packet will update a counter which indicates how long the transmit interface should stop sending packets. Any packets which has already started transmission will not be affected. Both "XOFF" and "XON" packets are allowed.

7.1.9 Gigabit Ethernet MAC

Built into the Tigon is an Ethernet MAC capable of running at either 10/100/1000 Mbit/second. Support for 10/100 modes is achieved

through an MII interface. Support for 1000 mode is currently done via external Fibre Channel components operating at Gigabit Ethernet speeds.

The transmit and receive Gigabit Ethernet interfaces function independently of each other when configured for full-duplex operation. The MAC is also capable of operating in half-duplex mode.

7.2 Host/NIC Software Interface

The main mechanism to effectively communicate between the host and the Tigon PCI/Gigabit Ethernet Controller is via buffer descriptors located in host memory. This is not the only technique: mailboxes are also supported, as already stated. Host descriptors are efficient since they allow for larger amounts of data to be passed without the need for host access over the PCI bus. In most environments each time the host processor directly accesses a location on the PCI bus, the host slows down and loses some of its processing time.

The following description gives the standard way of working of the Tigon and its way of interacting with the host. This is the point from where we started: we have studied the usual functionalities of the NIC, in order to be able to modify them when necessary and useful for our aims. For example, we will change a little bit the data DMA between the NIC and the host. Usually, as soon as the NIC receives frames from the network, it does some checks and then transfers the data contained onto the host. With event building code working on the NIC, we will need to keep the data on the card memory until an event is completed, when all the fragments belonging to it will move to the host at once.

This and other features will be modified on the NIC, in order to accomplish the project requirements, but, first of all, it is necessary to give an overview on the standard behaviour of the NIC, to know what must be kept and what must be changed.

The NIC manages data structures in host memory using DMA and the following control block:

- The *Shared Configuration Block*

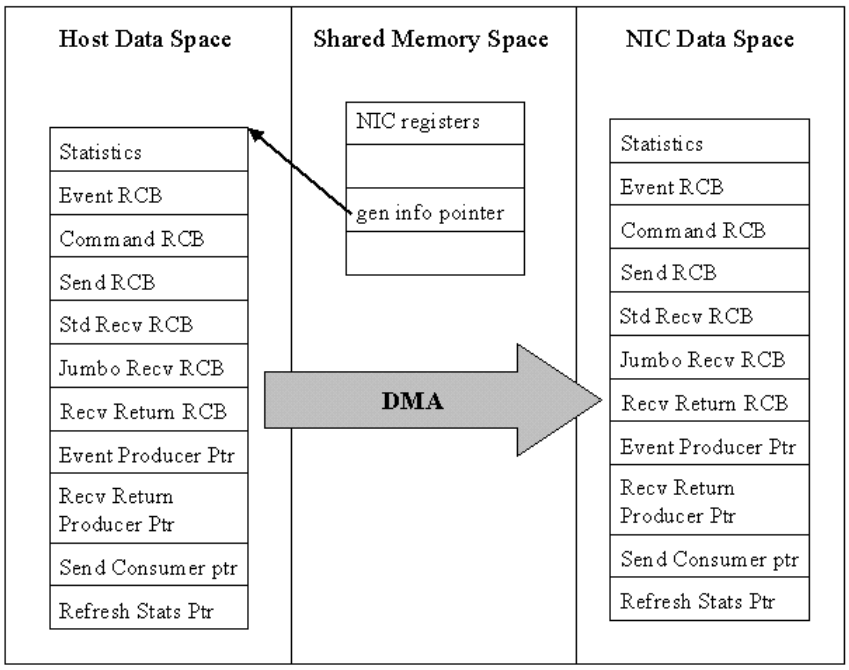

Figure 7.5: Memory model

This is a 16 kByte shared memory region of host mapped memory, as shown in figure 7.5. All registers in this block can be shared between the host and the NIC, however some are used operationally by the host while others are used operationally by the NIC.

The Tigon accesses the shared memory through a 2 kByte window. The base of this window is set by the Window Base Address register (in the PCI Configuration Region), which defines a local memory address (from the point of view of the NIC's internal processor). That register may contain any valid local memory address, but the usage of the least significant 11 bits varies depending on how the local memory is addressed. If the 2 kByte window is used, then the least significant 11 bits are ignored and are substituted with zeros.

Shared Configuration Block	Address
PCI Configuration Region	0x0000 - 0x003F
Tigon Configuration Region	0x0040 - 0x04FF
Mailboxes	0x0500 - 0x05FF
General Communications Region	0x0600 - 0x07FF
Local Memory Window	0x0800 - 0x0FFF
Reserved	0x1000 - 0x3FFF

Table 7.1: Shared Configuration Block

Otherwise the entire Window Address register is used to indicate the local memory address of the operation.

The host maps any Tigon local memory region into the shared memory, using the Local Memory Window, which is a 2 kByte memory region. This window is accessed by setting the window Base Address Register (the host uses this window to download the firmware into the NIC and update the Send Ring).

- The *General Information Block*

  This is in the host memory and contains the statistics area and the control blocks for the shared ring structures. The NIC DMAs this block from the host during firmware initialisation.

7.2.1 Shared Rings

As said before, the host and the NIC use a series of shared rings to communicate with each other. While each ring itself is shared, there are two indices that control the operation of each ring which are not shared. These are the producer index and the consumer index. The producer adds elements to the ring and increments the producer index while the consumer removes elements from the ring and increments the consumer index. When the producer and the consumer indices are equal, the ring is empty. When the producer is one "behind" the consumer, the ring is full. Figure 7.6 shows the producer-consumer model.

General Information Block	Offset
Statistics	0x000 - 0x3FF
Event Ring Control Block	0x400 - 0x40F
Command Ring Control Block	0x410 - 0x41F
Send Ring Control Block	0x420 - 0x42F
Receive Standard Ring Control Block	0x430 - 0x43F
Receive Jumbo Ring Control Block	0x440 - 0x44F
Receive Mini Ring Control Block	0x450 - 0x45F
Receive Return Ring Control Block	0x460 - 0x46F
Event Producer Pointer	0x470 - 0x477
Receive Return Ring Producer Pointer	0x478 - 0x47F
Send Consumer Pointer	0x480 - 0x487
Refresh Stats Pointer	0x488 - 0x48F

Table 7.2: General Information Block

7.2.2 Data rings

Data rings contain some information about the data in the ring element and a pointer to actual data buffer. There are two sets of data rings, a single send ring and a set of receive rings. The ring element for all data rings is the buffer descriptor, which describes an area within host memory where data is waiting to be transmitted or received. Figure 7.7 shows the producer-consumer model with data.

Send Ring

The send ring is used to transfer data from the host to the NIC for transmission on the network. The host is the send ring producer and the NIC is the send ring consumer.

This ring is in the NIC memory and the host writes to it using the Local Memory Window.

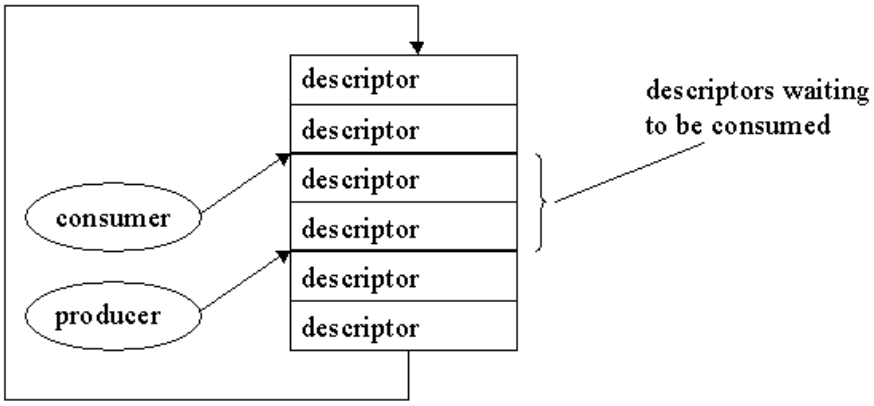

Figure 7.6: Producer-consumer model

Receive Rings

A set of receive rings are used to transfer data that has been received from the network from the NIC to the host. There are three rings into which the host places buffer descriptors that point to empty buffers, and one ring that the NIC uses to place buffer descriptors that point to filled buffers. Three rings are used to allow for different sized data buffer. One ring, the Standard Ring, uses standard (for Ethernet) 1514 Byte buffers. Another ring, the Jumbo Ring, uses extended 9014 Byte buffers. The last ring, the Mini Ring, uses small buffers whose maximum length can be specified during run time.

The host writes the receive buffer descriptors, pointing to empty buffers, into the Standard Ring and the Jumbo Ring and hands them out to the NIC. When the NIC fills the buffers associated with these buffer descriptors, it returns them back to the host in the Return Ring. The host determines which buffer descriptors was used by looking a the flag bit (to indicate the Standard or the Jumbo Ring) and the index.

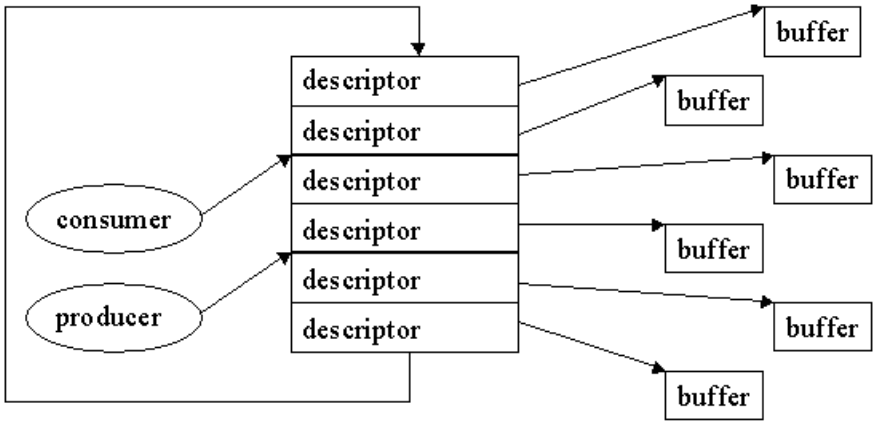

Figure 7.7: Producer-consumer model

7.2.3 Transmit Flow Diagram

The following diagram (figure 7.8) shows how a single frame is sent from the host to the NIC and onto the network. For this example, the frame is described in a single buffer descriptor.

1. The host creates a frame in the host memory.

2. The host creates a buffer descriptor or series of buffer descriptors that describe the frame and places it or them in the send ring on the NIC, using the Local Memory Window.

3. The host updates the send producer index and writes it into mailbox 2 in the shared memory region.

4. The NIC receives an internal mailbox event informing that the send producer index has been modified.

5. The NIC starts to DMA the frame from the host to the transmit buffer in the NIC and enqueues the frame for the transmission.

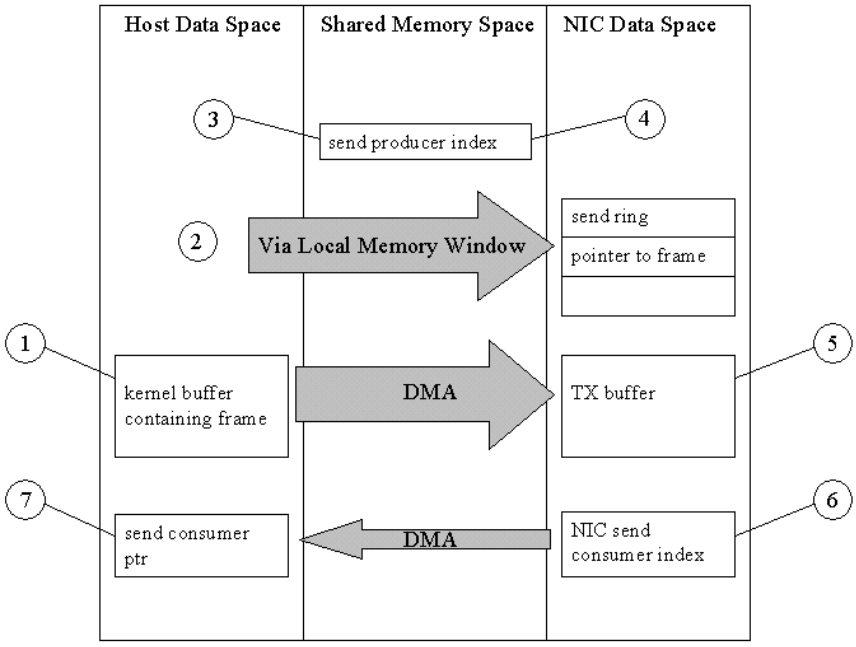

Figure 7.8: Transmit Flow Diagram

Once the DMA of the frame is completed, the frame is transmitted onto the network.

6. The NIC starts a DMA of the send consumer index to notify the host that the frame pointed to by the buffer descriptor(s) has been consumed. The DMA goes to the address specified in the send consumer pointer. The NIC may interrupt the host at this time.

7. If the host was interrupted or is already processing events or sending and receiving updates, it gets the new value of the send consumer and can now free the frame, as no further references are made it.

7.2.4 Receive Flow Diagram

The allowing diagram (figure 7.9) shows how a single frame is received from the network into the NIC and into the host. For this example, the frame fits into a single buffer descriptor.

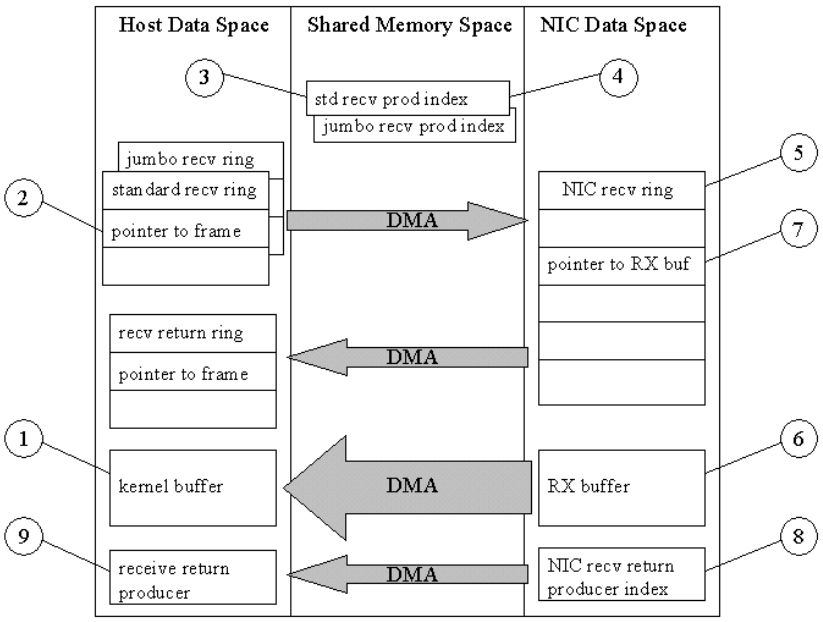

Figure 7.9: Receive Flow Diagram

1. The host allocates a buffer in host memory.
2. The host creates a buffer descriptor that describes the buffer (standard or jumbo) and places it in the correct host resident receive ring.
3. The host updates the corresponding ring producer index.
4. The NIC receives an internal mailbox event informing that a receive ring producer index has been modified and starts the DMA of the

new buffer descriptor from the appropriate receive ring to the NIC copy of that receive ring.

5. The DMA of the new buffer descriptor completes and the NIC now waits for a frame to arrive from the network.

6. A frame arrives from the network. The NIC immediately begins to DMA the frame into the host buffer described by the buffer descriptor. If the frame is larger than the value specified in the max_len field of the Standard Ring, the NIC will use a buffer from the Jumbo Ring, if one is available. If none Jumbo buffer is free or the packet is less than max_len size, the NIC will use one or more buffers described in the Standard Ring.

7. When the reception of the frame completes, the NIC updates the length and the flags fields in the buffer descriptor in the NIC copy of the receive ring and starts the DMA of the buffer descriptor(s) back into the host receive Return Ring.

8. The DMA of the filled receive buffer descriptor(s) completes. The NIC start a DMA of the receive Return Ring producer index to notify the host that the frame pointed to by the buffer descriptor has been filled. The DMA goes to the address specified in the receive Return Ring producer pointer. The NIC may interrupt the host at this time.

9. If the host was interrupted or was already processing events or sending and receiving updates, it gets the new value of the receive Return Ring producer index and can now use the filled buffer, as no further references are made to it by the NIC.

7.3 NIC Performance Evaluation

Once installed the two Tigon NICs we bought from Alteon *Web*Systems, we decided to work under a Linux environment. There are several Tigon's drivers built for different operating systems, like Windows NT, Solaris

and Linux; but the NIC standard firmware is compiled into the machine language opcode by using GNU R4000 tools and the usage of GNU tools is easier in a Unix-like environment. Thus, we decided to install the Linux driver, developed at CERN by Jes Sorensen in 1998–1999. Like all Linux device drivers, this is made up of object code (not linked to be a complete executable) that can be dynamically linked to the running kernel by the *insmod* program and can be unlinked by the *rmmod* program (further information regarding Linux device drivers can be found in the Linux web page [3] and in [30]). We have then built the GCC cross compiler, version 2.95.2 (for references see the GNU web site [2]), and used it with a i.686 Linux machine on a MIPS R4000 target, with some patches supplied by Alteon *Web*Systems.

We have also built a debugger: the GDB debugger, version 4.18, which works as a remote debugger. For this, we had to developed a dedicated driver.

The test system is shown in figure 7.10. Two PCs are running Linux with two Alteon Gigabit Ethernet cards connected by optical fibre. Each PC is also equipped with a standard NIC for general purpose communication.

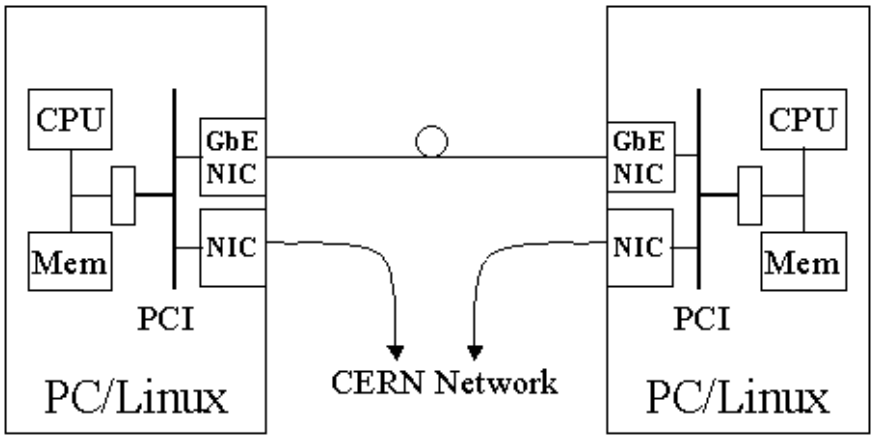

Figure 7.10: Test setup

With this setup, we have done tests to evaluate the Gigabit Ethernet NIC performances. First of all, we measured the "host to host" communication performance, using UDP and TCP protocols. We have measured the time to transfer a block of 32 MBytes chopped in fixed size packets and varied the size of the packet. We used powers of 2 for varying the frame size and we used also Jumbo frames, to overcome the standard packet size limit of Ethernet (1.5 kByte). For these tests, we run the standard driver, without any kind of performance tuning. We simply switched off the Nagle's algorithm, as usually suggested to reach better performances [35].

The performance results are shown by figures 7.11 and 7.12.

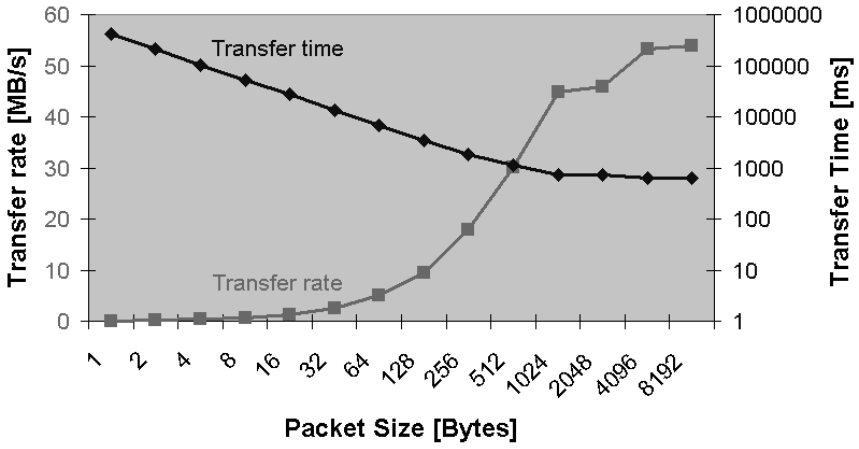

Figure 7.11: UDP performance measurements

The maximum throughput is obtained, as expected, with large data packets It reaches 55 MByte/s for UDP and 14 MByte/s for TCP.

As we can see, in both cases, we are quite far from the Gigabit nominal bandwidth of 125 MByte per second. That is a consequence of several factors: all the tests done are protocol dependent and there are no tuning features applied. All these things affect the throughput.

Of course, UDP performances are much better than TCP ones. UDP is a unreliable, connectionless protocol, so it does not have to wait for

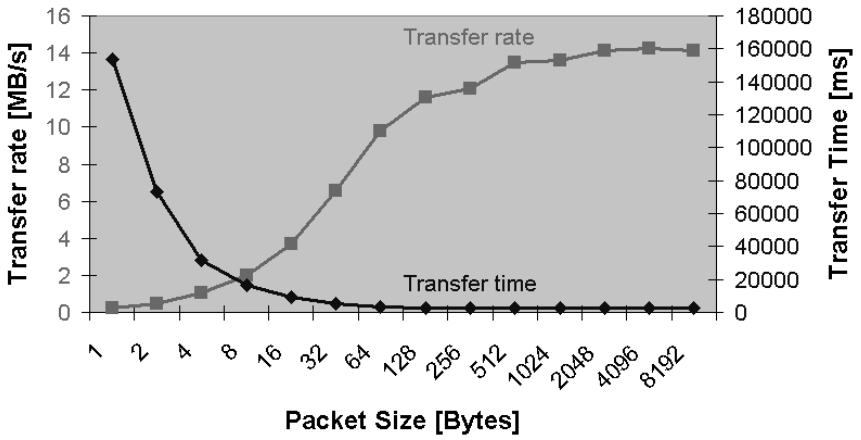

Figure 7.12: TCP performance measurements

connection establishment and data acknowledgements, it does not manage flow control and it does not have to keep data copies for possible retransmissions. These features make UDP faster than TCP, but does not give any guarantee of delivery. Our setup, of course, is quite reliable: the sender and the receiver are directly connected by optical fibre and thus no data losses are expected and we verified that was it was the case. Thus the results obtained are consistent.

In the TCP case, performances measurements could be improved by properly setting some variables, like the send and receiver buffer size, the socket buffer size and so on, as suggested in [1]. However, we can already notice from these results that increasing the packet size, the throughput improves by a significant factor. Also Jumbo frames are enabled and they seem to be a great performance improvement. Of course, we cannot avoid the protocol delays, which are quite high but make TCP a reliable protocol.

The second type of tests we have done is the "NIC to NIC" performance measurements. In these tests we sent Ethernet frames directly from one NIC to the other and measured the time the packets took to reach the destination. We did not utilise any kind of protocol but we

used raw Ethernet, to avoid higher layer protocol overheads. We expect a higher link bandwidth utilisation.

We had to modify a little bit the standard firmware, of course. From the receiver point of view, we changed the firmware receiving routine, in order to check and count the frames arrived and then discard them, without DMAing any data to the host. We simply receive frames, check them and immediately free buffer space, by updating the descriptors. In this way we are sure that all frames sent have correctly arrived and we also avoid the delays due to the DMA.

From the sender point of view, on the other hand, we had to write our own sending routine. This routine is triggered by the mailbox mechanism (described before) and forces the NIC to send Ethernet frames, directly built inside the card, without waiting for any data DMA from the host. This trick is done by allocating an Ethernet frame in the first free buffer descriptor and making the send producer descriptor pointing to it. The frame is then automatically completed by the hardware with the checksum and sent through the network. The hardware updates also the send consumer descriptor and so the card and the software are ready to sent another packet.

We assigned a "LHCb magic number" to the frames sent by our sending routine, in order to distinguish raw Ethernet frames from the others and to be sure to send and count them in the proper way. We placed this number in the *Length* or *Type* field of every frame, because it was an empty and free field, due to the lack of any kind of protocol, whose type should be specified here.

The tests realized are the same as in the "host to host" performance measurements: the same amount of data, 32 MByte, is sent using different packet sizes (always powers of 2) and the time the data takes to go from the source to the destination is measured. In this case we could use the internal clock of the NIC to measure the delay, instead of a standard time routine. The NIC internal clock works with a precision of about one microsecond and so the measurement results are pretty precise. They are shown in figure 7.13.

The formula reported in figure 7.13 has the following meaning:

- a = firmware overhead/frame [μsec]

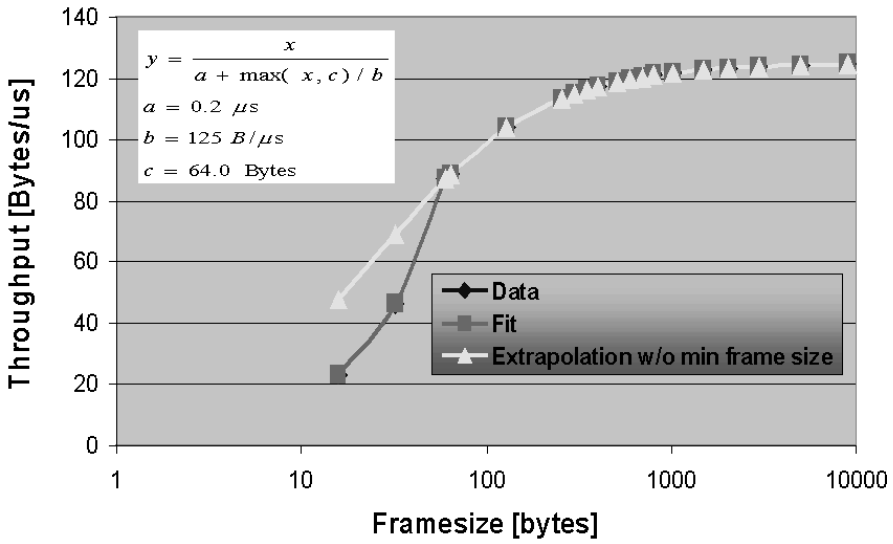

Figure 7.13: Raw Ethernet performance measurements

- b = link banwidth [Byte/μsec]
- x = frame size (minimum = 64 = c) [Byte]

$\Rightarrow$ Time to sent a fragment is:

$$a + b^{-1} \cdot x$$

$\Rightarrow$ Throughput is:

$$\frac{x}{a + b^{-1} \cdot x}$$

The fit to the measured points shows that the firmware overhead per packet is 0.2 microseconds. The influence of the overhead on the throughput decreases asymptotically when the packet size increased.

As we can see, the throughput is significantly higher than in the case of UDP and TCP tests. With raw Ethernet a maximum throughput of 124

MByte/s (which is almost the nominal Gigabit Ethernet bandwidth—equals to 125 MByte/s) is reached, while with UDP/IP and TCP/IP we get a maximum throughput of 54 MByte/s and 14 MByte/s respectively.

With raw Ethernet frames sent directly from one NIC to the other, we can fill the wire at any given packet size (from 64 to 9000 Bytes).

Figure 7.13 shows a drop for frames smaller than 64 Bytes, which does not fit with the extrapolation curve. This is correct, because 64 Bytes is the minimum frame size, according to the Ethernet standard. All packets smaller in size than this threshold, are padded by the hardware, and hence take the same time to be transmitted. The Bytes real transmitted in fact are 64 all the time but the throughput computation is done with the frame payload size, which is less then 64 Bytes.

However, increasing the frame sizes until the minimum size is reached, the performances improve significantly: it is possible to send out frame at a frequency up to 1.4 MHz. For frames bigger than 512 Bytes more than 95% of the Gigabit nominal bandwidth is used.

Jumbo frames give the best results: sending them with raw Ethernet we can almost use 100% of the nominal bandwidth available for data.

Chapter 8

Event building on the NIC

This chapter is devoted to the embedded event building implementation and results. Embedded event building is the final goal of the study presented in this document. All the previous chapters aim to analyse the motivation which supports it and to justify the choices and steps necessary to build a correct working environment and to realise embedded event building.

Before going into detail about the necessary code changes (in order to adapt event building functionalities to the smart NIC environment) and the final results, some considerations are requested to make clear what are the assumptions and the purposes of the work presented in this last chapter. Our first intention, in fact, is to demonstrate that embedded event building is feasible, but we want also to analyse the capabilities and the limits of the project. The first section gives some considerations which are the foundations of the embedded event building evaluation.

Further in the chapter the implementation on the NIC will follow and the corresponding results will be estimated and analysed.

8.1 Frequency of Fragments

The maximum frequency at which a destination can handle fragments is an important feature in order to evaluate the performance of an event building system.

The first aspect which has to be considered is the fact that in a SFC the data readout occurs concurrently with the fragment processing. The time to read a fragment depends on the fragment size and on the link bandwidth. In fact:

$$t_{read} = \frac{d}{b}$$

where

- t_{read} = time to read a fragment [μs]
- d = fragment size [Bytes]
- b = link bandwidth [MByte/s] or [Byte/μs]

For example, on a Gigabit Ethernet technology, the link bandwidth is equal to 125 Byte/μs and so the time to read fragments of 1000 Bytes is the following:

$$t_{read} = \frac{d}{b} = \frac{1000Bytes}{125Byte/\mu s} = 8\mu s$$

At the same time, the maximum frequency at which a destination can manage fragments depends on the fragment size and on the software overhead to handle each fragment (t_{ov}). According to this and to the assumption that data readout occurs concurrently with fragment processing, we can distinguish two cases:

1. $t_{read} < t_{ov}$

   In this case the time to read a fragment is shorter than the time necessary to handle it, as figure 8.1 depicts. The maximum frequency of fragments that can be sustained is given by the overhead time:

$$MaxFreq = \frac{1}{t_{ov}}$$

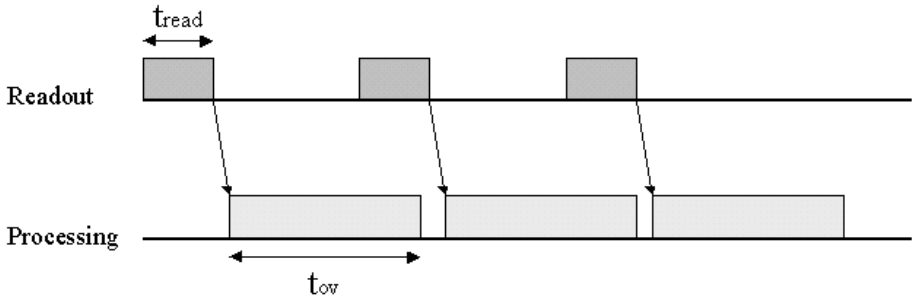

Figure 8.1: Case 1: $t_{read} < t_{ov}$

Although the network link is capable of delivering fragments at a higher frequency, this value cannot exceed $1/t_{ov}$, otherwise buffer overflow will occur.

2. $t_{read} > t_{ov}$

   In this case the time to read a fragments is longer than the time necessary to handle it, as shown in figure 8.2. The maximum fre-

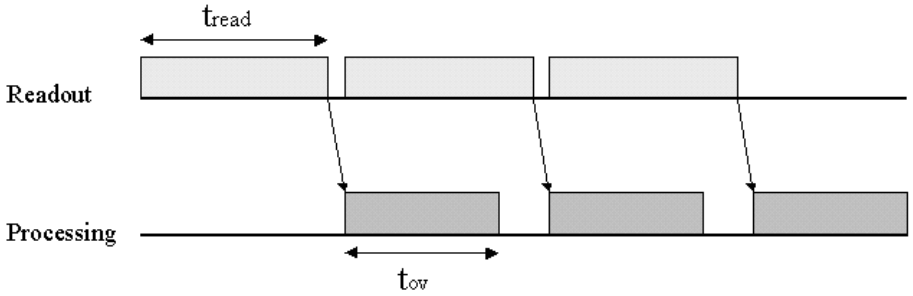

Figure 8.2: Case 2: $t_{read} > t_{ov}$

quency of fragments is then limited by the link bandwidth and is given by the following relation:

$$MaxFreq = \frac{1}{t_{read}} = \frac{b}{d}$$

Figure 8.3 shows the maximum achievable frequency of packets as a function of the packet size. The plateau ($1/t_{ov}$) is the interval where the packet handling overhead is limiting. The asymptotic part is the limit imposed by the link bandwidth.

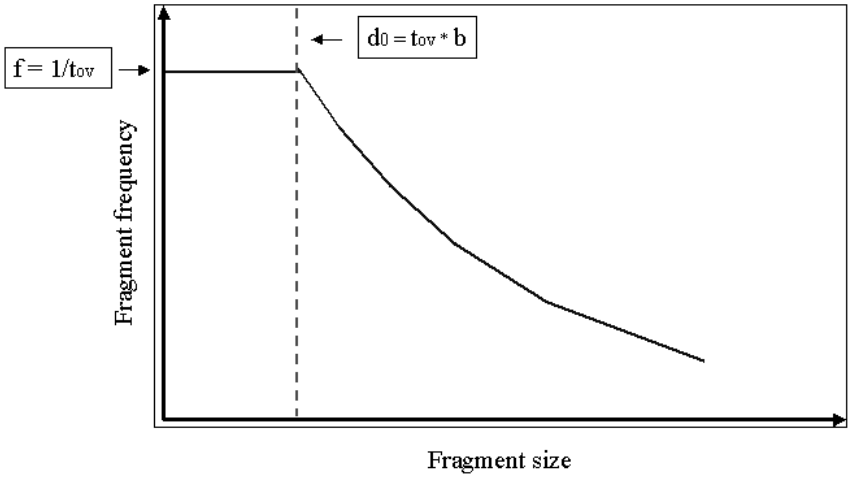

Figure 8.3: Diagram of fragment frequency as function of fragment size

8.2 Embedded Event Building Implementation

In order to perform and evaluate embedded event building, we had to port the event building code, described in Chapter 5, to the NIC environment.

From the destination point of view, we have decided to concentrate on the simple time-out algorithm, because it will be the main model on which the LHCb DAQ group will focus for the time being. Also the original fragment generation function in the source algorithm had to be modified, due to the lack in the primitive NIC environment of all the C standard libraries and functions (such as the *rand*() function), we have largely used in the first release. Most of them, actually, could be ported, but they would generate large object files and this is doubtful how useful they would be.

The resulting code is a little bit simplified, compared to the previous one, but the main steps and ideas, which characterised it, are kept and function in the same way. Thus we can speak about it as the same algorithm, re-adapted to a new working environment.

The following subsections describe in detail which changes have been done to the source and destination models to make them working the embedded system of the Alteon NIC.

8.2.1 Source Model

The source model performs the function of sending event fragments directly from the NIC. An event fragment has the following lay-out, which is graphically displayed in figure 8.4:

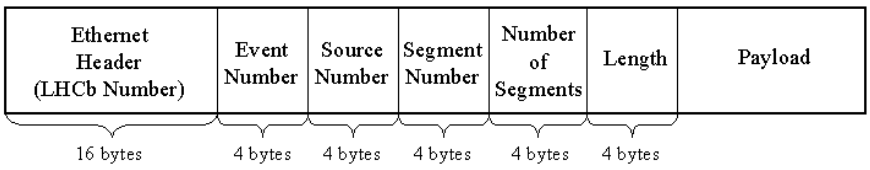

Figure 8.4: Event fragment structure

- *Ethernet Header*

  It is composed of the *destination address* (6 Bytes), the *source address* (6 Bytes), the *Length* or *Type* field (4 Bytes), which is

used to store the "LHCb magic number", that characterises our protocol packets. The Ethernet header is defined to be 14 Bytes long, but we have padded it with 2 more Bytes in the Type field, in order to make the following fragment header 32-bit word aligned. The architecture of the NIC is a 32-bit architecture and so, with the 4-Byte alignment, the data writing/reading functions are more efficient.

- *Event Number*

  It is the event number the fragment belongs to (4 Bytes) [1].

- *Source Number*

  It is the identifier of the source which has sent the fragment (4 Bytes).

- *Segment Number*

  In case a physics fragment only fits in more than one Ethernet frame, this value keeps the information of the frame order in the fragment structure. This value is necessary because, without an higher-level protocol, the reconstruction of several frames into the fragment they belong, must be performed by the destination algorithm, which needs some basics information to accomplish the task.

- *Number of segments*

  It stores the number of total frames which belongs to the same fragment, in case the fragment spans over more than one frame. This information is then used by the destination to know how many frames must be expected to rebuild the complete fragment.

- *Length*

[1] For the time being, event numbers are considered increasing all the time and no care is taken regarding the problem of the possible wrap around. Anyway, it should be noted that with a 32 bits field as event counter and 40 kHz of Level-1 rate it will take 10^5 seconds, i.e. more than a day, to wrap around.

It is the length of pure data field, not including the Ethernet header and the fragment header.

The fields concerning the fragment segmentation are not used in the algorithm, because, for the moment, we assume that all fragments fit in one Ethernet frame. Our aim, in fact, is to verify if embedded event building can be implemented as a basic idea. We are mostly interested in the performance of the algorithm running on the NIC. Segmented fragments will simply perform like the corresponding number of one-frame fragments. Moreover, if Jumbo frames are really implemented by a Gigabit Ethernet architecture (especially by Gigabit Ethernet switches), all fragments will probably fit in one frame.

The source model writes the fragments directly in the NIC sending buffer. The sending routine implemented is quite similar to that one realized for the raw Ethernet tests. It is triggered by the mailbox mechanism and allocates event fragments, with a predefined length, inside the sending buffer and sets up the producer descriptor. In this way the hardware is notified that there are data to be sent and thus it automatically completes the frames with the checksum and sends them through the network. The hardware then updates the producer descriptor and repeats the sending procedure until there are no more data to send.

The amount of fragments to be sent is determined at the startup, when the number of sources in the system to test and the number of events to process are defined. Of course, the number of fragments is given by the total amount of events times the number of sources.

The generation function is a simplified version of the previous one, described in Chapter 5. As already stated, in the NIC primitive environment the use of the standard C libraries would be hard and inefficient, and so we cannot use most of the predefined functions utilised in the previous code. In this case the generation function simply generates all the requested events, one after the other, and then mixes up the order of the fragments belonging to one event.

8.2.2 Destination Model

The destination model performs the event building functions. As outlined before, we have fixed our attention on the simple time-out algorithm. This choice is due to the following considerations:

1. The no time-out algorithm, even if fast, is not reliable because it does not handle data losses and thus it will not be implemented in a real system. It was developed in order to have a baseline of performance for comparison with more complex algorithms. We have tested that the software overhead of simple time-out or automatic adjustment algorithm is not much higher than the no time-out algorithm overhead (see figures 5.18, 5.19 and 5.21 in Chapter 5) and it can be fulfil in the event building requirements.

2. The automatic adjustment, on the other hand, even if feasible, is exceeding the event building basic needs. It was developed in order to verify the possibility of increasing software overhead for gaining more flexibility at system exceptions or crashes. We have tested that the event building protocol can be completed with this facility, if other requirements must be satisfied. However, now we are interested in experimenting the possibility to implement event building in the NIC. Whenever this works with secure results and performances, we will be able to really develop more complex algorithm, if necessary for system reliability.

Therefore, the receiving model performs the simple time-out algorithm functionalities, described in Chapter 5, section 5.2.3.

All fragments arriving at destination are stored in the receive buffer. As soon as a fragment arrives, the receiving routine copies its buffer address into a new variable and frees the associated receive descriptor. In this way the descriptors are immediately released and the NIC is able to receive new coming fragments and does not have to wait that stored data are DMAed to the host.

After freeing the receive descriptor, the routine starts the event building steps. Due to the limit of the buffer ring and NIC memory, there is

only pointer management, to avoid time-consuming copying around of data, which would be unacceptably slow. Thus the fragments received reside in the receive buffer until the event is completed and the event building functions and the descriptor table handle the pointers to those buffer locations.

As soon as an event is completed or is in time-out, all the fragments belonging to it are DMAed to the host and the related buffer occupancy can be freed. The memory management is quite complex and shows some problematic aspects. The receive buffer is a portion of contiguous memory, which is handled like a ring structure, as described in the previous chapter. The producer and the consumer indexes are the two managers of the buffer. When a new fragment arrives, it is stored into the first free location of the receive buffer and the producer index is automatically updated by the hardware to point to the next free location where new incoming data will be placed. On the other side, the consumer index is updated by the receiving routine to point to the first doubleword of valid data which have to be processed. This means that the consumer index points to the first arrived fragment of an event which has not been completed yet. When that event is completed, all the associate fragments are DMAed to the host and the corresponding buffer locations should be freed. The problem is that the receive buffer is organised as a circular buffer occupying a contiguous block of memory and the fragments are not coming in order but fragments belonging to different events can be mixed up (according to their arrival order) into the buffer. Therefore, when an event is completed or is in time-out, the receiving routine gives all the associate fragment pointer values to the DMA engine and marks the relative buffer positions as space that can be freed, because they are no more used by the event building functions. Then it updates the consumer index to point to the first doubleword of valid data which have yet been processed. This implies that if one or more fragments of an incomplete event are mixed among the fragments which should be returned, only those before the first non-free-able will be really released, while the others will stay there until all the previous locations will be freed. Thus the memory at some points could be fragmented, as figure 8.5 shows. This is an objectionable and non-optimised feature but it is unavoidable

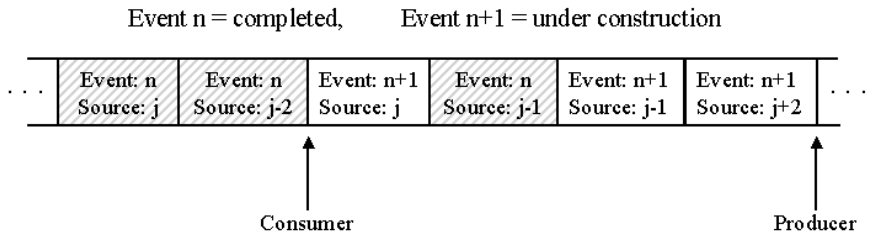

Figure 8.5: Example of fragmented memory

for memory managements which do not support memory relocations [25].

Fortunately the sending routine does not need to take care of the risk of running out of descriptors or buffer space in the destination. The flow control mechanism handles this exceptions forcing the hardware to send an "XOFF" packet whenever there are no free descriptors left or the buffer space is exceeded. The flow control packet, received by the sender, will disable the transmitting part of the MAC for an adjustable amount of time.

The flow control mechanism is based on high and low water marks for number of receive descriptors unused and receive buffer space which is unused. When the destination exceeds the high water mark an "XOFF" packet is sent . The "XOFF" length can be programmed: the "XOFF" is set on a 16 bits field (which correspond 0.5 microseconds) and can have either the value of 0x1ff or 0xffff.

If the "XOFF" length is about to expire and the destination is still above the high water mark, another flow control packet is sent to refresh it up until the low resource drops below the low water mark. At this point an "XON" message is sent and the source can restart sending.

In order to avoid triggering the flow control mechanism, which slows down the performances of the whole system, we have implemented most used functions and the descriptor table of the event building code in the NIC *scratchpad*.

A scratchpad RAM is similar to a L1 cache [2] but it is mapped into the processor address space. The scratchpad is visible only by the CPU and can be used by it for any purpose, such as storing instructions or intermediate values. The NIC has two scratchpads, one for each processor. The use of the scratchpad for the event building code makes the NIC computation faster because the scratchpad is reserved only to the processor and there are no competitors to access it, as it happens in the case of accessing the memory bus. In this case, the DMA engine, the CPU and the MAC interface compete and the MAC interface usually has the priority. This feature can be configured in order to assign the priority to the processor and make the computation faster, but, in this way, the MAC receiver might slow down and frame losses or corruption would occur.

In the case of DMA transfers, instead, there are no speed or performance problems because the PCI bus is four times faster than the MAC interface and thus the DMA transfers are not performance bottlenecks. Moreover, the NIC has the necessary features required to efficiently support the scatter/gather process [3] during the DMA transfers. This ability of off-loading this task from the processor increases the host efficiency and the overall system performance.

8.3 Performance Results

In order to evaluate the embedded event building performances and verify its applicability to our system requirements, we have done measurements on the same setup as the one used for the raw Ethernet tests (see figure 7.10 in Chapter 7).

One of the two PCs generates the event fragments, emulating the presence of several sources. The other PC is performing the event building ("destination"). The number of sources which the system consists of, the

[2] The Level 1 cache is a memory cache built into the microprocessor

[3] The idea of scatter/gather is that a portion of data to be moved may be placed in several fragments throughout the memory and can be collected and transfered without any data copying

fragment size and the number of events to send are interactively defined from the keyboard. As soon as the source module gets these values, it first sends a start-time packet to the destination, then all the fragments required by the total amount of events and the number of sources, and finally it sends the end-time packet.

On the other side the destination waits for data coming and, as soon as it receives the first packet, it starts the timing function and then performs the event building, until it gets the end-time packet. At this point the destination stops the time counting and the event building tasks and performs the appropriate statistics evaluations, calculating the average time per fragment and per event, the number of fragments possibly lost and events uncompleted, and so on.

Like in the case of raw Ethernet performance measurements, we have used the internal clock of the NIC to precisely evaluate the time delays with a precision of about one microsecond.

Moreover, in order to measure the overhead per fragment we have limited ourselves to fragment size smaller than $t_{ov} \cdot b$ (see section 8.1) and we have estimate the time to manage the event building functions for each single fragment incoming. We observe two distinct peaks, as shown in figure 8.6:

- The first peak, of 10 microseconds, was measured for the majority of the incoming fragments;

- The second peak, whose duration depends on the number of sources in the system, correspond to the last fragment of each event and includes the event termination operations (scatter/gather functions to collect all the fragments belonging to the completed event, necessary tasks to free the ring buffer, and so on).

If we plot the time overhead values for those last event fragments, we can note that it increases linearly with the number of sources in the system (see figure 8.7). This is what it is expected. More steps are required for collecting all the event fragments and freeing the ring buffer locations. Moreover, following the straight line depicted in figure 8.7, it

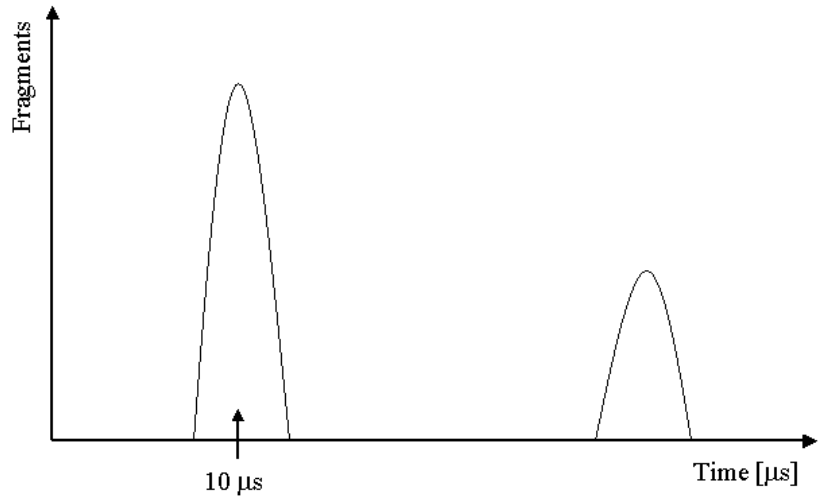

Figure 8.6: Measurements of overhead per fragment

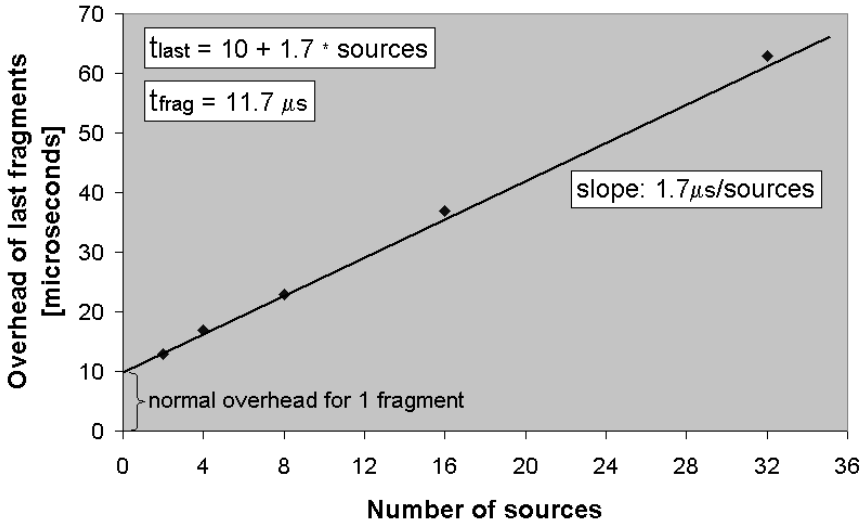

Figure 8.7: Tested values for last event fragments

comes out that for zero sources we should have 10 microseconds of overhead for each last event fragment and this is exactly the time overhead per fragment we tested on average. The slope of the straight line is equal to 1.7 [μs/sources]. Thus the average time for the last event fragment is given by the following relation:

$$t_{last} = 10\mu s + 1.7 \cdot Sources$$

hence, the average overhead per fragment is:

$$t_{ov} = 11.7\mu s$$

We get the same value per fragment when averaging the measurements of a large number of events.

We can conclude that the results obtained from these measurements are really encouraging. We have not experienced data losses and we can perform embedded event building with an average time per fragment of 11.7 microseconds, decomposed in 10 microseconds for normal fragment operation and 1.7 microsecond for cleaning up. The overhead for one event is then:

$$t_0 + Sources \cdot t_{frag} \cong Sources \cdot t_{frag}$$

where t_0, the time overhead to set up a new event descriptor, is negligible.

This means that event building at a frequency of incoming fragments of almost 100 kHz is possible with the network interface card that we have tested. This frequency is sustainable for fragments with a size up to 11.7μs $*$ 125 Byte/μs = 1465 Bytes.

Figure 8.8 shows the estimated event frequency in one destination (SFC) as a function of number of sources. Naturally, increasing the number of sources, the event frequency decreases.

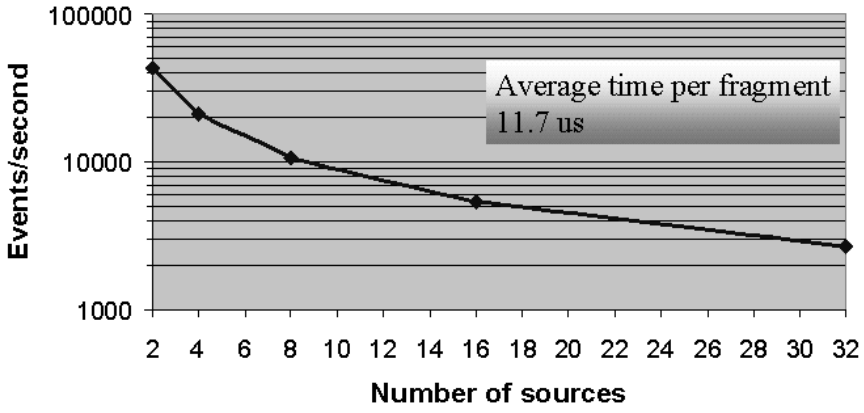

Figure 8.8: Embedded event building performances, as function of number of sources (in logarithmic scale)

8.3.1 Comparison with Host to Host Event Building

In order to validate the performance of the embedded event building, we have done event building measurements from host to host, using the TCP/IP protocol. The test setup is again the same as the one used for the previous measurements: two PCs (233 MHz) running Linux, with an Alteon smart NIC each, as interface to the Gigabit Ethernet link.

The event building software runs on the two host CPUs, performing the same functionalities as the one running on the NIC: one of the two PCs generates the event fragments, emulating the presence of several sources, the other performs the event building.

The destination opens a socket [14] connection and waits for data coming. The source also open a socket and connect it to the destination's one. Once the communication channel is established, the source sends first a packet with the total number of fragments which are coming to the destination and then the amount of fragments claimed. The first packet is sent with the purpose of facilitating the timing on the destination (a similar procedure was applied in the case of embedded event building measurements). On the other side, the destination waits for data, and

as soon as it receives the first packet, it starts the timing function and performs the event building, until all fragments have arrived and have been handled.

The number of sources which the system consists of, and the number of fragments to send are interactively defined from the keyboard.

We have done the following measurements:

- With S being the number of sources and N the number of events, the source sends $N * S$ fragments, with empty payload (this means minimum Ethernet frames of 64 Bytes);
- The total time (T) to build N events is first measured;
- Then we repeat this procedure, but with the event building being bypassed, to measure the time T_0).

In this way, the quantity $T - T_0$ gives the event building time necessary to handle N events, i.e. $N * S$ fragments.

The total overhead per fragment, given by the value T, is:

$$t_{frag} = 9.3\mu s$$

The measurement of T_0 shows that this overhead is made of two components:

1. The overhead due to the event building protocol:

$$t_{ov} = 3.1\mu s$$

2. The overhead due to TCP/IP protocol and operating system:

$$t_{prot} = 6.2\mu s$$

A comparison with the NIC to NIC event building leads to the following considerations:

- The event building overhead is roughly in the inverse ratio of the two processor clock speeds: 3.1 microseconds on a 233 MHz processor for host to host, compared to 11.7 microseconds on the 88 MHz processor of the NIC.
- The NIC to NIC event building has no additional protocol overhead. This may not be true in a real system if SAR (Segmentation And Reassembly) of data packets and/or flow control need to be implemented.

The value of the event building overhead on the NIC is expected to decrease when future versions of the NIC card will be available which will certainly include include faster processors. However one can expect that the hosts will in turn implement faster processors (anyway, it must be said that the code on the NIC can still be optimised in order to make all the event building procedure faster—ongoing optimisations have already achieved an average overhead time per fragment of 9 microseconds).

The main advantage in using the NIC based event building is to release the host from a task that would occupy its CPU at $\sim 100\%$, whereas it has other tasks to fulfil, like distributing events to processors in the attached farm and monitoring the analysis process.

8.4 Performance of Event Building

As a summary of the study and the work done to implement and test the feasibility of embedded event building, some considerations regarding its frequency are discussed.

In a data acquisition system with S sources, which send (for simplicity reasons) all fragments with the same size d, the time to build one event in a destination is equal to:

$$S \cdot max(t_{ov}, t_{read})$$

(see section 8.1 for the terminology).
Therefore, the maximum frequency of event building in one destination is given by the following relation:

$$f = \frac{1}{S \cdot max(t_{ov}, t_{read})} = \frac{1}{S \cdot max(t_{ov}, \frac{d}{b})}$$

and the absolute maximum frequency in one destination is given by the next formula:

$$f_{MAX} = \frac{1}{S \cdot t_{ov}}$$

In a system with D destinations working in parallel the total frequency of event building is equal to:

$$F_{TOT} = \frac{N}{S} \cdot f$$

where f is the frequency for 1 fragment.

Thus the event building frequency can be increased by adding more destinations (sub-farm controllers). However, that process is limited to the maximum frequency at which the source can emit fragments and this is a function of the overhead in the source and of the fragment size.

In order to illustrate these considerations, we can focus on the embedded event building implemented by using raw Gigabit Ethernet and Alteon smart NICs. We already know that the nominal network bandwidth is 125 MByte/s, t_{ov_dest} is equal to 11.7 microseconds and, therefore, d_0 is equal to 1460 Bytes and the maximum frequency in one destination is 85 kHz (see section 8.1 for the word terminology).

If we assume a time overhead in the source for emitting fragments of 1 microsecond, d_0 becomes equal to 125 Bytes.

Consequently, the absolute maximum event building frequency reached for fragments with a size smaller than or equal to 125 Bytes would be 1 MHz. Then the number of destinations required in order to sustain this

rate would be 12 destinations per source. Such an event building system would solve the LHCb Level-1 trigger problem.

These considerations are based on a point to point connection and do not take into account the communication network that links the sources and the destinations in a real system. The question then arises whether the network can effectively provide the necessary bandwidth and whether congestion occur.

In the particular case of Gigabit Ethernet, we do not know yet whether the XON/XOFF mechanism is effective across a switch. On the other hand, when more than one source send data to the same destination, refraining all of them but one from transmitting reduces the usable bandwidth. It would be desiderable that the switch implements buffer at output ports ("output queueing").

Investigating the characteristics of a Gigabit Ethernet switching network is presently the main goal of the ongoing event building project for the LHCb data acquisition system.

Conclusions

The goal of the present work was to demonstrate the feasibility of embedding the event building protocol in the processors recently provided by the new generation of Network Interface Cards (NIC), in particular for Gigabit Ethernet technology. This implied the software implementation of the event building algorithms, their adaptation for their operation in the embedded processors, the development of a model of the event building conditions and the measurement and interpretation of performance of this solution. A comparison with the standard implementation on NICs' host processors would provide the yardstick to judge the suitability of the proposed method.

This development was carried out within the team developing the Data Acquisition (DAQ) system for LHCb, one of the four experiments that have been approved for the future high energy collider LHC (Large Hadron Collider) at CERN.

Characterisation of event building

At the location of the experiment LHCb, the crossing of particle "bunches" will occur every 25 nsec, i.e. at a frequency of 40 MHz. Practically all those crossings generate proton-proton (p-p) collisions with production of secondary particles that will generate electric signals in a variable number of the $95 \cdot 10^4$ sensitive "channels" of the detectors. The goal of the LHCb experiment is the detail study of the "Beauty" particles (B mesons). Production of B mesons is very rare compared to all the possible outcomes from p-p collisions.

It is impossible to record the full amount of data generated and per-

form an analysis later on. This data can be estimated to be of the order of 40 MHz $*100$ kBytes = 4000 GByte/s. The role of the "trigger" is to apply selection criteria on-line in such a way as to enrich the sample of the events eventually retained in interesting B meson pair candidates. It is possible to apply the selection criteria as a sequence of tests with increasing complexity, rejecting a certain fraction of the events at each level. At LHCb, this selection sequence consists of four levels, named Level-0 up to Level-3.

Level-0 processes data locally on the detectors whereas level-1 requires that data from several detectors be collected in a processor and levels 2 and 3 need all the available data (some 100 kBytes) in order to elaborate the decision.

For levels 2 and 3, it is necessary to collect the data from the whole detector in one processor that will take the decision. Obviously, spending some 200 msec per event at an input rate of 40 kHz requires several hundreds of processors working in parallel, each one analysing the data from one event.

The task of collecting the data from the whole detector (via intermediate concentrators called Readout Units(RU)) into a designated processor is called "event building". The strategy followed by all experiments at the LHC for event building is to use a switching network that interconnects the RUs with the SFVs (Sub Farm Controllers). The network is routing the "fragments" issued by the RUs while a dedicated process in the SFC is collecting these fragments and reconstructs the full event.

At LHCb, typically a hundred of RUs will deliver fragments to an equivalent number of SFCs, each SFC controlling several analysis processors. RUs and SFCs emit and receive fragments of some 1000 bytes at a frequency of 40 kHz, i.e. every 25 usec.

Justification for an implementation on an embedded processor

The high rate of interruptions, the inefficiencies of the TCP/IP protocol usually adopted for data interchange on host processors and the multiple data copies implied by the operating system suggest that the task of event building could better be executed on a dedicated processor, without

operating system, that would poll for the arrival of fragments rather than rely on interrupts. Furthermore the processor on an SFC must carry out several important tasks like distributing full events to the analysis processors and monitoring their work.

The advent of embedded processors, implemented on Network Interface Cards (NIC) of the new generation of fast data links, gave us the opportunity to test this concept.

The work developed

First of all, the event building requirements have been studied and analysed and a model of the DAQ system has been defined.

Then the attention has been focused on the event building protocol. The strategy adopted is a "push protocol", where the RUs send an event fragment to a pre-defined SFC, as a function of the event numbers, without any dialogue between those two units. Three different algorithms have been developed in order to study and compare different solutions. The first algorithm performs the tasks of a simple fragment collector and event builder, which relies on a reliable system with no data loss. The second one implements time-out functions in order to face with possible occurrences of missing fragments and avoid deadlocks. The third algorithm, then, adapts its steps to the system configuration, detecting and handling possible source crashes. The software overheads of the protocols have been measured to evaluate their impact on the event building performance. The results show that low software overheads are achievable in the processors available now and that this fulfils the requirements.

We have then concentrated on the Gigabit Ethernet network technology and on the Alteon smart NIC. In order to implement the software on an embedded system we had to become familiar with the embedded environment and the new Gigabit standard. Many difficulties were found in understanding the smart NIC hardware and software architecture and its way of working, but they have been successfully solved. At the beginning we tested the standard features and performance of the card. Then we started changing the firmware in order to learn how to communicate with the NIC and its embedded processor. Once the card architecture

was clear, we adapted it to our needs and requirements to implement the event building software, and we tested the new system. We measured then the implementation performance, whose results are presented in the next section.

The results

It has been demonstrated that event building on a Gigabit Ethernet smart NIC (the Tigon 2 Network Interface Card, by Alteon *Web*Systems) at a frequency of almost 100 kHz is sustainable for fragments with a size up to 1465 Byte. The LHCb Level-1 output rate is 40 kHz and the event fragment size is envisaged to be $\sim$ 1 kByte. Therefore, the results produced by this work perfectly fit the requested fragment size and give a safety margin threshold of a factor of 2, from the event building frequency point of view, for possible future scalability of the data acquisition system. Moreover, ongoing optimisations on the work done have already improved the performance results obtained.

In principle, event building for the Level-1 trigger (1 MHz, $\sim$ 100 Byte/source) is feasible, from the point of view of software overhead. However a faster embedded processor would be an advantage in order to reduce the number of sub-farm controllers. Furthermore, the performance of the readout network still needs to be confirmed.

Finally, this work was presented at the DAQ2000 Workshop, held during the IEEE Nuclear Science Symposium and Medical Imaging Conference in Lyon, on 20th October 2000 [18]. It has subsequently been published in the conference proceedings and it has been submitted for a peer-review to IEEE Transactions on Nuclear Science.

Possible future developments

Some other future goals and developments have been envisaged for the ongoing project:

- First of all it would be interesting to evaluate the impact of *interrupt coalescence* on the host performance. This idea consists on reducing the number of interactions between the host and the NIC.

Instead of interrupting the host as soon as new data come, the NIC could pass to it information in chunks. The advantage of provide large amount of information is that it reduces the number of time the host and the NIC must interact and thus the host CPU could perform event building functions. The disadvantage is that it can create latencies. Setting the coalescing threshold too high can cause longer latencies.

- As already stated in the last chapter, the next future main goal of the project will be *to investigate the characteristics of a Gigabit Ethernet switching network*. "Real world" tests on a system with a Gigabit Ethernet switch, which interconnects 16 sources and 16 destinations, have been already started.

- Finally, it would be useful to use measured parameters in a *detailed simulation* of the readout network. The simulation project has been initiated on summer 2000. At the moment a simple "two by two" event building system has been simulated and studied, using the Ptolemy tools [36]. Much effort needs to be spent on this task, in order to investigate the characteristics of the whole system.

Acknowledgements

At the end of this work, I want to thank my supervisor, Prof. Francesco Dalla Libera, for having trusted me, my ideas and my work, and for his support and precious advice during the development of the project and the writing of the thesis.

I want to thank my supervisors at CERN, Dr. Jean-Pierre Dufey and Dr. Beat Jost, for their essential contribution towards my technical education, for having given me the great opportunity to be involved in this project, and, especially, for the many hours they spent to read and correct this document.

I want to thank in particular Dr. Niko Neufeld for his priceless help in understanding that terrible smart NIC architecture and for his patience in teaching me all he knows about hardware. But, above all, I want to thank him for his nice friendship.

Many thanks also to my colleagues at CERN, especially Sebastien and Stefan; their positive attitude have encouraged me day by day.

I cannot forget to thank my parents and my sister Giovanna. They always trusted me and my skills and supported my choices.

A special thank to my two great friends, Chiara and Giovanna. Even if far away, I could feel their affection and their support all the time.

Many thanks to all my friends, in particular to Paola and Roberto for their friendship and hospitality during my stay at CERN, and to Tito for his help, his kindness and for his understanding of my sometime nervous attitude.

Finally, I want to thank Paolo, without whom I would have never been working at CERN and I would have never lived this great experience.

Bibliography

[1] *Enabling High Performance Data Transfer on Host.* http://www.psc.edu/networking/perf_tune.html.

[2] *GCC home page.* http://www.gnu.org/software/gcc/gcc.html.

[3] *Linux HeadQuarters.* http://www.linuxhq.com.

[4] *QNX Literature, Neutrino System Architecture.* http://www.qnx.com/literature/nto_sysarch/kernel3.html.

[5] AlteonNetworks, *Tigon/PCI Ethernet Controller.* Revision 1.04.

[6] AlteonWebSystem. Private communication.

[7] AlteonWebSystems, *AlteonWebSystem, Weeb Speed for e-Buisiness.* http://www.alteonwebsystem.com.

[8] ——, *Gigabit Ethernet/PCI Network Interface Card, Host/NIC Software Interface Definition.* Revision 12.4.11.

[9] E. Barsotti, A. Booth, and M. Bowden, *Effects of various event building techniques on data acquisition system architectures*, in Computing for high luminosity and high intensity facilities, 1990.

[10] C. Bizeau et al., *On the Feasibility og High Performance ATM-based Event Builders*, in DAQ96 Conference Proceedings, 1996.

[11] D. Bucar, *Reducing Interrupt Latency.* http://www.e.kth.se/ e96_dbu/ex/literature_study.html.

[12] CERN, *CERN home page*. http://www.cern.ch.

[13] ——, *LHC, The Large Hadron Collider Project*. http://lhc.web.cern.ch/lhc.

[14] D. E. Comer and D. L. Stevens, *Internetworking with TCP/IP*, PRENTICE HALL, 1994.

[15] M. Costa et al., *Results from an ATM-based Event Builder Demonstrator*, in IEEE Transactions on Nuclear Science, 1996.

[16] M. Della Negra et al., *Technical Proposal (CMS collaboration)*, technical report, CERN/LHCC/95-71, 1995.

[17] J.-P. Dufey et al., *The LHCb Trigger and Data Acquisition System*, in 11th IEEE NPSS Real Time Conference, 1999.

[18] J.-P. Dufey, B. Jost, N. Neufeld, and M. Zuin, *Event Building in an Intelligent Network Interface Card for the LHCb Readout Network*, in DAQ2000 Conference Proceedings, 2000.

[19] J.-P. Dufey and I. Mandjavidze, *DAQ Implementation Studies*. Internal note LHCb/98-029 (unpublished), 1998.

[20] F. Harris and M. Frank, *LHCb Data Flow Requirements*. Internal note LHCb/98-027 (unpublished), 1998.

[21] H. J. Hilke et al., *Technical Proposal (LHCb collaboration)*, technical report, CERN/LHCC/98-4, 1998.

[22] P. Jenni et al., *Technical Proposal (ATLAS collaboration)*, technical report, CERN/LHCC/94-43, 1994.

[23] B. Jost, *Timing and Fast Control*. Internal note LHCb/99-001 (unpublished), 1998.

[24] B. Jost et al., *DAQ Architecture and Readout Protocols*. Internal note LHCb/98-031 (unpublished), 1998.

[25] D. E. Knuth, *FUNDAMENTAL ALGORITHMS, The Art of Computer Programming, second edition*, Addison Wesley, 1973.

[26] M. F. Letheren, *Architectures and technologies for data acquisition at the LHC experiments*, in Second workshop on electronics for LHC experiments, 1996.

[27] C. Miron et al., *Fibre Channel Performance with IBM Equipment*. Internal note RD11/95-05, 1995.

[28] Myricom, *Myrinet Products*. http://www.myricom.com.

[29] R. O. Onvural, *Asynchronous Transfer Mode Networks, second edition*, Artech House, 1995.

[30] A. Rubini, *LINUX Device Drivers*, O'REILLY, 1998.

[31] R. Seifert, *Gigabit Ethernet*, Addison Wesley, 1998.

[32] T. Shanley and D. Anderson, *PCI System Architecture*, Addison Wesley, 1995.

[33] C. E. Spurgeon, *Ethernet, The Definitive Guide*, O'REILLY, 2000.

[34] A. S. Tanenbaum, *Computer Networks, third edition*, PRENTICE HALL, 1996. Page 1.

[35] ——, *Computer Networks, third edition*, PRENTICE HALL, 1996. Pages 534–535.

[36] University of California at Berkeley, *The Ptolemy Project*. http://ptolemy.berkeley.edu/.