An Embedded Real-Time System on ATLAS ROBIN

Inauguraldissertation
zur Erlangung des akademischen Grades
eines Doktors der Naturwissenschaften
der Universität Mannheim

vorgelegt von

M.Eng. Informatiker Maoyuan Yu
aus Anhui, China

Mannheim, 2012

Dekan:	**Professor Dr. Heinz Jürgen Müller, Universität Mannheim**
Referent:	**Professor Dr. Reinhard Männer, Universität Heidelberg**
Korreferent:	**Professor Dr. Peter Fischer, Universität Heidelberg**

Tag der mündlichen Prüfung: **24.01.2013**

Zusammenfassung

ATLAS ist einer der größten Teilchendetektoren am Large Hadron Collider (LHC) für Hochenergieexperimente. Der ATLAS Detektor produziert Ereignisdaten von mehr als 40 Terabyte pro Sekunde mit einer Ereignisrate von 40 MHz. Dieses riesige Datenvolumen wird mit Hilfe der ATLAS Trigger und Data Acquisition Chain (TDAQ) vor der permanenten Speicherung reduziert. Das ATLAS Readout Buffer Input (ROBIN) Teilsystem ist ein wesentliches Bestandteil in der ATLAS TDAQ. Ereignisdaten erreichen ROBIN mit einer Rate von 100kHz mit einer GröSSe von 1 kByte für jedes Datenpaket. Es wird eine durchschnittliche Output Rate von 10 kHz erwartet.

Das ROBIN System wird von zwei Prozessoren gesteuert: einem Xilinx Virtex II 2000 FPGA und einem PowerPC 440 Mikro-Controller. Der FPGA Prozessor spielt die zentrale Rolle als der Datenfluss Kern für hohe Ereignis-Raten und -Bandbreiten, der die Ereignisdaten (messages) und Kontrollnachrichten on-the-fly überträgt. Der PowerPC stellt die Kontrollfunktionen wie Ordnen des Ereignis Puffers, Aufschlüsseln und Ausführen von eingehenden Anfragen vom ROS PC als auch das Auslösen von Antwort Nachrichten zur Verfügung

Diese Dissertation behandelt das Design eines eingebetteten Echtzeit Systems für einen IBM PowerPC 440GP Mikro-Controller als Management Kern für das ROBIN Teilsystem.

Für die Implementierung der Power PCs Anwendung wird eine Seiten basierte Lösung für die Verwaltung des Ereignis Puffers präsentiert und ein Hash Algorithmus wird für die Ereignis Suche verwendet. Für eine effiziente Suche im eingebetteten Software System wird eine ŞChained Free Hash-Node MethodeŤ verwendet, um die dynamische Struktur der Hash-Tabelle zu speichern. Dieses Vorgehen erzielt eine gute Performance und benötigt keinen extra Speicherplatz.

Als wesentlicher Bestandteil des ROBIN Systems muss die eingebettete Software für das ROBIN PowerPC System eine hohe Performance liefern. Es werden zwei Software Architekturen für den ROBIN PowerPC vorgestellt. Die erste ist als einfache Kontroll-Schleife verwirklicht. Im zweiten Design ist ein eingebettetes real time Linux Betriebssystem für den ROBIN PowerPC Prozessor konfiguriert, angepasst und optimiert worden. Die Performance beider Implementierungen wurde in aufwändigen Messungen in einer RROS/ROBIN Testumgebung gemessen. Die experimentellen Ergebnisse zeigen, dass das Nicht-OS basierte PowerPC System bereits die derzeitigen ATLAS DAQ Anforderungen übertrifft, während die Performance des Linux-basierten ROBIN PowerPC Systems die Basisanforderungen der

entsprechenden ROBIN Anforderungen nicht erfüllt.

Abstract

ATLAS is one largest particle detector at the Large Hadron Collider (LHC) for high energy physics experiments. The ATLAS detector produces over 40 terabytes of event data per second at an event rate of 40MHz. The huge volume of data are reduced through the ATLAS Trigger and Data Acquisition Chain (TDAQ) before permanent storage.The ATLAS Readout Buffer INput(ROBIN) subsystem is an essential device within the ATLAS TDAQ. Event data arrive at ROBIN with an input data rate of 100kHz with 1kBytes for each data packet, and an average output rate of 10kHz is expected.

The ROBIN system is controlled by two processors: a Xilinx Virtex II 2000 FPGA and a PowerPC 440 micro-controller. The FPGA plays the centric role as a high-rate and high-bandwidth data-flow core, which transmits event data (messages) and control messages on-the-fly across the board. The PowerPC provides the control functionalities, such as arranging the event data buffer, decoding and executing incoming request messages from ROS PC as well as initiating response messages backwards.

This dissertation addresses the software design of an embedded real-time system centering around an IBM PowerPC 440GP micro-controller, as the management core of the ROBIN.

For the implementation of the PowerPC's application, a page-based solution is presented to handle the event buffer management, and a hash searching scheme is applied to deal with the event lookup. For an efficient searching in the embedded software system, a Chained Free Hash-Node method is proposed to store the dynamic data structure of the hash table. This strategy achieves a good performance with no extra memory space.

As a main part of the ROBIN device, the embedded software for the ROBIN PowerPC system must provide high performance. Two software architectures for the ROBIN PowerPC are presented. The first is implemented as a simple control loop without any operating system. In the second design an embedded real-time Linux operating system is reconfigured, adapted and optimized for the ROBIN PowerPC processor. Performances of these two implementations are measured through elaborate experiments in a simulated ROS/ROBIN testing environment. The experiment results show that the standalone non-OS based PowerPC system is already above the current ATLAS DAQ requirements, while the performance of the RTLinux-based ROBIN PowerPC system does not meet the related ROBIN baseline requirement.

Contents

1 Introduction

1.1 Motivations

Since the 6th century BC human beings have already had the idea that all matter is composed of elementary particles. Ancient Greek created the philosophical doctrine of atomism. In the 19th century people believed that each element of nature was composed of a single, unique type of particle, the fundamental particles of nature, and named them atoms, after the Greek word *átomos*, meaning "indivisible". However, not until the end of the 20th century, physicists discovered that atoms were not actually the fundamental particles of nature.

Explorations of nuclear physics and quantum physics in the early 20th century not only led to the development of nuclear weapons, but also brought the discovery that one atom can be generated from another [33] [43].

Theory of Standard Model (SM) [55] in the particle physics is built in the 1970s. The theory explains the state-of-the-art classification of elementary particles. The SM contains 24 fundamental particles (i.e. 12 particle/anti-particle pairs). They are supposed to be the constituents of matter. However, many postulations of the Standard Model have not been proved. It predicts, for example, the existence of a type of boson known as the Higgs boson, which has yet to be discovered through high energy physics experiments.

To explore the theories beyond the Standard Model, high-energy physics experiments at an energy level of above 1 TeV have to be done. Currently Tevatron at Fermilab is the only high-energy particle accelerator, which reaches the required energy level. The accelerator is located near Chicago, USA. It has a centre-of-mass energy of 1.96 TeV.

In order to explore physics experiments at required energy regions, the European Particle Research Laboratory (CERN) sponsored the project of Large Hadron Collider (LHC) [14]. The LHC is a proton-proton collider. It is expected to become the

world's largest and highest energy particle accelerator and collider, if it completes the mission to accelerate two proton beams to an energy level of 7 TeV. Then a center-of-mass energy of 14 TeV can be reached by colliding the two proton beams. This may bring far more chances for physicists to explore the open issues in high energy physics [59].

Four main detectors are being constructed inside LHC for the measurement of particle interactions. ATLAS [7] is one of the largest LHC detectors. ATLAS is a general-purpose detector. When the proton beams produced by the LHC interact in the center of the detector, a variety of different particles with a broad range of energies will be produced. ATLAS is designed to measure the broadest possible range of signals, and to ensure that, whatever might take place from any new physical processes or particles, ATLAS will be able to detect them and measure their properties. Hence the preciseness and the large bandwidth are two essential measurement features for the ATLAS detector. To this end a number of sub-detectors are constructed inside ATLAS to observe a large variation of particles precisely.

Every second around 40 million proton beams cross the center of the ATLAS detector, which generate a new event every 25 nanosecond, i.e. at an event rate of 40MHz. The detector creates approximately 1MByte data for each event. That means, ATLAS must handle a data volume of 40TByte per second. Hence the ATLAS data acquisition system must support real time and huge data volume processing. Compared to other high energy physics experiments, the demand upon the ATLAS data acquisition system is substantially higher, as the data rate and the data volume are concerned.

The ATLAS data acquisition system comprises three levels of trigger systems, which use simple information to identify in real time the most interesting events out of 40 million events every second. The first level trigger is based on the electronics inside the ATLAS sub-detectors. The other two levels primarily run on a large cluster of computers near the detectors. The computers are equipped with similar technologies. The design aims to distribute the data selection tasks uniformly across the whole system, in order to reduce the efforts in system administration and various software implementation. After the first level trigger, about 100,000 events are selected every second; and after the third level trigger, only a few hundreds of events remain. The data reduction rate is altogether up to a factor of 10^5.

ReadOut Subsystem (ROS) is a core device in the ATLAS data acquisition chain, built between the level 1 and the level 2 triggers. The ROS layer receives detector data on 1600 point-to-point readout links (ROL). Each link has a data rate of up to 100kHz and a data volume of 100MByte/s. The ROS layer is designed to buffer all the event data temporarily and forward them on request to the level 2 trigger for

the trigger decision.

In order to sustain the huge volume of rapidly incoming data an extremely efficient buffering system is required. ROBIN (ReadOut-Buffer INput), as an essential device inside the ROS system, is expected to play the role. In the baseline architecture of the ATLAS level 2 trigger there are totally hundreds of ROS systems. Each ROS system comprises three or four active ROBIN devices; and each ROBIN device receives event data on three ROLs. For each ROL one 64MB SDRAM is available on ROBIN as the according event data buffer. The goal of the ROBIN device is to support an input rate of 100kHz and a bandwidth of up to 160MByte/s.

ROBIN comprises two processors, a XILINX XC2V2000 FPGA and an IBM PowerPC 440 micro controller, which coordinate with each other to realize the functionalities of ROBIN. Generally the FPGA plays the centric role as a high-rate and high-bandwidth data-flow core, which transmits event data and control messages on-the-fly across the system. The PowerPC provides the management and control functionalities. It provides solutions to the effective allocation of event data buffers and solutions to the efficient event data retrieval; it decodes and executes incoming request messages from ROS PC, and initiates response messages backwards.

1.2 Organization of the Dissertation

This dissertation deals with the involved algorithms, the software design and software optimization of the ROBIN PowerPC system. The structure of this dissertation is as follows.

Chapter 2 introduces the project background of ATLAS ROS/ROBIN. ROBIN is the centric device inside the ATLAS readout subsystem (ROS) and ROS is a core subsystem in the LHC/ATLAS data acquisition chain. This chapter gives an overview to the Large Hadron Collider (LHC) and one of its largest detectors, the ATLAS detector. Challenges in the ATLAS data acquisition chain (TDAQ) are pointed out. ROS is the essential buffering system inside the ATLAS level-2 trigger. Its system requirements and baseline architecture are particularly addressed.

Chapter 3 introduces the ATLAS final design of the ROBIN board and addresses its advantages over its previous solutions. In the final design of ROBIN an FPGA processor and a PowerPC micro-controller are integrated. It takes advantage of both processors.

Chapter 4 presents the strategy of ROBIN event buffer management and algorithms involved in the effective buffer allocation and efficient event retrieval.

Given the strategy for event buffer management, the software interface, software

requirements and software components for the ROBIN PowerPC system are analyzed in chapter 5 and its detailed tasks get also defined here.

Two architectures are proposed for the implementation of the ROBIN PowerPC system, depending on whether a real-time operating system is integrated into the system. Both architectures are realized in this work and presented in chapter 6 and chapter 7, respectively.

Finally, chapter 8 concludes the contribution of this work, points out possible improvements, and gives an outlook to the ATLAS ROBIN system in future experiments.

2 Background: LHC - ATLAS - ROS

This chapter gives an overview to the project background of ATLAS ROS/ROBIN. ROBIN (ReadOut-Buffer INput), the main topic of this work, is the centric device inside the ATLAS readout subsystem (ROS). ROS is one of the hundreds of subsystems inside the ATLAS detector, which is the largest detector inside Large Hadron Collider (LHC), the world's largest particle accelerator and collider. Meanwhile ROS is also one of the most essential data buffering systems in the LHC/ATLAS data acquisition chain.

The structure of this chapter is as follows. Section 2.1 introduces the Large Hadron Collider (LHC). Challenges in the LHC data processing are pointed out. Section 2.2 presents the ATLAS detector, one largest detector of LHC. The data acquisition chain in the ATLAS trigger system is particularly discussed. Section 2.3 deals with the ATLAS readout subsystem (ROS), the essential data buffering system inside the second ATLAS trigger level. The system requirements and the baseline architecture of ROS are addressed in detail.

2.1 LHC - Large Hadron Collider

2.1.1 Overview

Currently the most modern theory of the elementary particle physics is the Standard Model (SM) [24]. It explains the state-of-the-art classification of elementary particles. However, in spite of abundant experimental evidence supporting the SM theory, different arguments indicate that the SM is not the ultimate theory of elementary particle physics [57] [35], [40] [49]. Meanwhile, discussions over the SM and

Detector	Energy(TeV)	Event Rate(MHz)	Data Volume(TByte/s)
PEP II			
BaBar	0.01085	238	5.4
Tevatron(Run II)			
CDF	1.96	7.6	1.9
D0	1.96	7.6	1.9
HERA			
HERA B	0.134	10.4	10
LHC			
ATLAS	14	40	40
CMS	14	40	40
ALICE	14	40	80
LHCb	14	40	3

Table 2.1: Comparison of different high energy particle colliders [48][8][16][10][29][6] [11][9][13][7][66].

its extensions show that there are good reasons to expect more interesting physics at the TeV energy level.

In order to explore the TeV energy scale, the European Particle Research Laboratory (CERN) approved the project of the Large Hadron Collider (LHC) [34]. The LHC is a particle accelerator and collider located at CERN in Geneva, Switzerland. It is currently still under construction. The LHC is the world's largest and highest energy particle accelerator, when its commissioning at 7 TeV is completed. The LHC is being funded and built in collaboration with universities and laboratories from 34 countries.

Table 2.1 compares the performance of the LHC accelerator with that of other existing accelerators. The center-of-mass energy of LHC is substantially higher than that of the currently most powerful accelerator Tevatron at Fermilab. Accordingly the probability of rare new physical events are expected to increase in the LHC experiments. Regarding event rate the PEP II's BaBar experiment is the most demanding, but its data volume is by far lower than the detectors at LHC, excluding LHCb. As the huge data volume and the high data rate are concerned, the experiments at LHC are significantly more demanding than other high energy physics experiments.

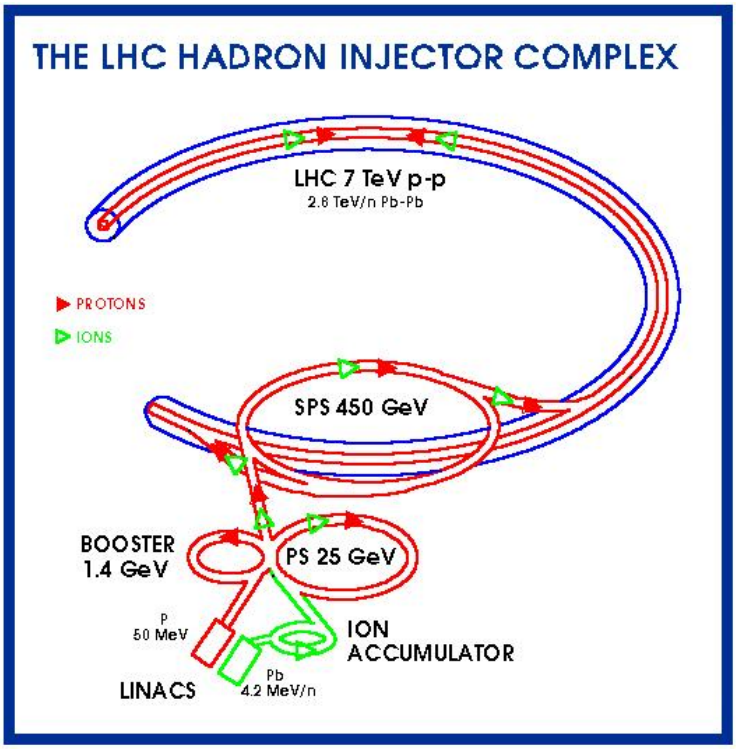

Figure 2.1: **Injection and acceleration scheme in the LHC collider [59].**

2.1.2 Construction

The LHC collider is contained in a 27 km circumference tunnel located underground at a depth ranging from 50 to 150 metres. The collider tunnel contains two pipes enclosed within superconducting magnets cooled by liquid helium. Each pipe contains a proton beam. The two beams travel in opposite directions around the ring. Additional magnets are used to direct the beams to four intersection points where proton-to-proton collisions of the two beams are expected to take place. The tunnel was formerly used to house the Large Electron/Positron Collider(LEP), which has stopped operating since 2000. LEP was working at an energy level of 200 Gev, which covers the mass region of the weak force carrying W^{+-} and Z^0 bosons. Due to the beam energy loss of synchrotron radiation, the lepton(e^+, e^-) collider can only accelerate the electron and positron to limited mass energy. Although LHC uses the same tunnel as LEP, all the infrastructures of LEP have been replaced by the superconducting magnets and high frequency cavity accelerators are used to accelerate and bend the beams. The beam bending magnets have a field strength of up to 8.4T.

Proton particles inside LHC have an energy level of around 7 TeV, which brings a total collision energy of 14 TeV. It takes around 90 microseconds for an individual proton to travel once around the collider. Instead of continuous beams, the protons will be "bunched" together into about 2,800 bunches, so that interactions between the two beams will take place at discrete intervals of over 25 nanoseconds. Besides, the LHC also offers an impressive luminosity ranging from the beginning $10^{33}cm^{-2}s^{-1}$ to the designed luminosity of more than $10^{34}cm^{-2}s^{-1}$. This enables the LHC experiments to generate even the rarest physics events.

Before being injected into the main accelerator, the proton particles are accelerated successively through a series of systems. Firstly a linear accelerator, Linac2, generates protons at an energy level of 50 MeV, and feeds the protons into a proton synchrotron. The Proton Synchrotron (PS) consists of two linear accelerators: the Proton Synchrotron Booster (PSB) and the Proton Synchrotron Ring (PSR). The PSB brings the particles up to an energy level of 1.4 GeV; and the PSR to 26 GeV. Moreover, the Low-Energy Injector Ring (LEIR) is used as an ion storage and cooler unit. The Antiproton Decelerator (AD) can produce a beam of anti-protons at 2 GeV, after cooling them down from 3.57 GeV. Finally the Super Proton Synchrotron (SPS) can be used to increase the energy of protons up to 450 GeV. An layout of the LHC injection and acceleration scheme is shown in figure 2.1.

2.1.3 Challenges in Data Processing

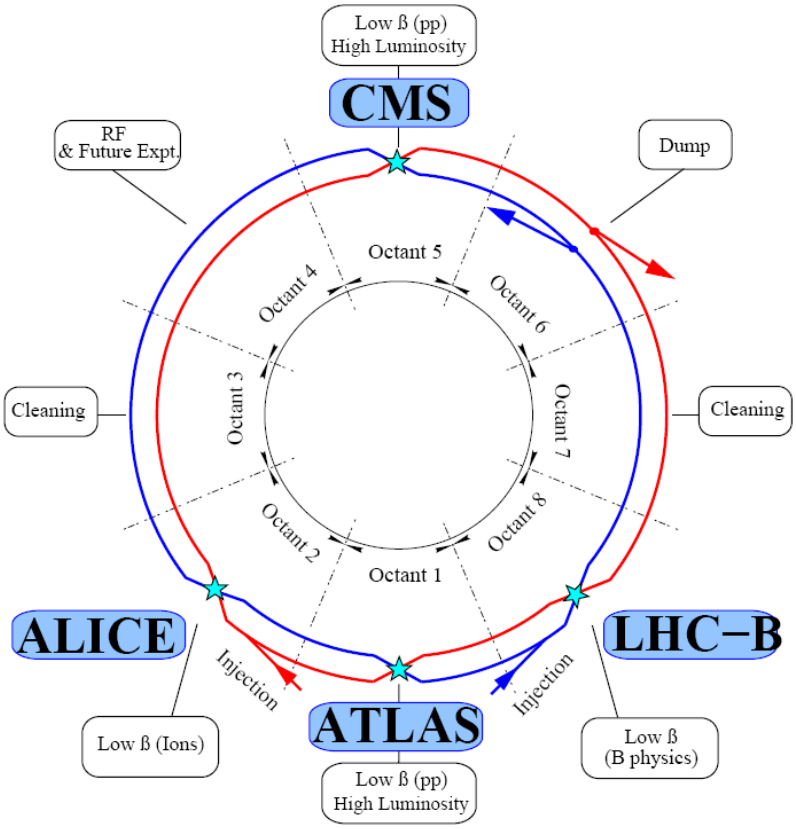

Figure 2.2: **Model layout of the LHC collider [59].**

The probability of physical events that are useful for the investigation of high energy particle physics is extremely low [7]. Therefore, the event detection rate has to be very high to observe sufficient target processes in a reasonable time. Moreover, to record rare physical events at the collider, highly precise detectors are necessary. Typically, one particle collider consists of a number of different detectors to cover the broad bandwidth of signals, and to ensure that a large variation of particles can be observed precisely. Each detector delivers the data in a large number of readout channels which transport either analog or digital data.

Four main detectors are being constructed inside LHC for the measurement of particle interactions. They are ATLAS (A Toroidal LHC ApparatuS), CMS (Compact Muon Solenoid), LHCb (LHC-beauty) and ALICE (A Large Ion Collider Ex-

periment). ATLAS and CMS are two relative larger and general-purpose particle detectors. LHCb and ALICE are smaller and more specialized. Figure 2.2 shows the layout of the LHC detectors.

All the four detectors are going to run at a rate of 40MHz that is more than five times the rate of Tevatron, which is currently the only accelerator reaching the TeV energy scale with a centre-of-mass energy of 1.96 TeV. Except LHCb the data volume at each cycle is an order of magnitude higher. Every second ATLAS and CMS will deliver nearly 40 terabytes, and ALICE even 80 terabytes. This raises a substantial demand upon the experiment's data acquisition process.

The huge amount of data will get never stored in any storage media. Therefore, a *pre-selection process* of the event data is necessary, so that a real-time storage of the final data amount will be possible. The pre-selection process analyzes the data, reduces the amount of data by a considerable factor (for example, a factor of 10^5 for ATLAS) and finally picks out the rare useful events from the background or irrelevant events. Besides a high factor of *data volume reduction*, a reasonable factor of *data-rate reduction* is necessary as well due to limited storage capability.

To achieve the high data-reduction factor in the pre-selection process, a complex event analysis algorithm is required. However, more complex algorithm implies more execution time. On the other side, the data acquisition process must sustain the data-detection rate of 40MHz, to ensure that no event data get lost. That means, the maximum allowable execution time for the data pre-selection in one cycle must not be over 25 nanoseconds.

One typical solution to the above contradiction is to use *staged triggers*. Firstly, the complex pre-selection algorithm is split into a series of independent steps. Each step reduces the data volume by a certain data-reduction factor, and hence the multiplicity of the reduction factors in all the steps is the overall data reduction factor of the whole pre-selection process. Each pre-selection step is realized at one trigger level, where decisions for data selection are made and data reduction is performed. Usually the data rate is also reduced at each trigger level. The staged triggers work on the event data successively and build a chain of data acquisition processes. On the other hand, the triggers are allowed to work in parallel independently and each trigger supports the input data rate from the previous trigger, so that the overall system can sustain the initial data-detection rate of 40MHz for the LHC collider.

Since in a staged trigger system the complexity of the data pre-selection algorithm is preserved, the process *latency* through the whole data acquisition chain is still inevitable. However, the parallel pipeline computing of all the triggers ensures that the initial data-detection rate is sustained. The latency time does not increase and no data get lost. Figure 2.3 shows the performance of an example staged trigger

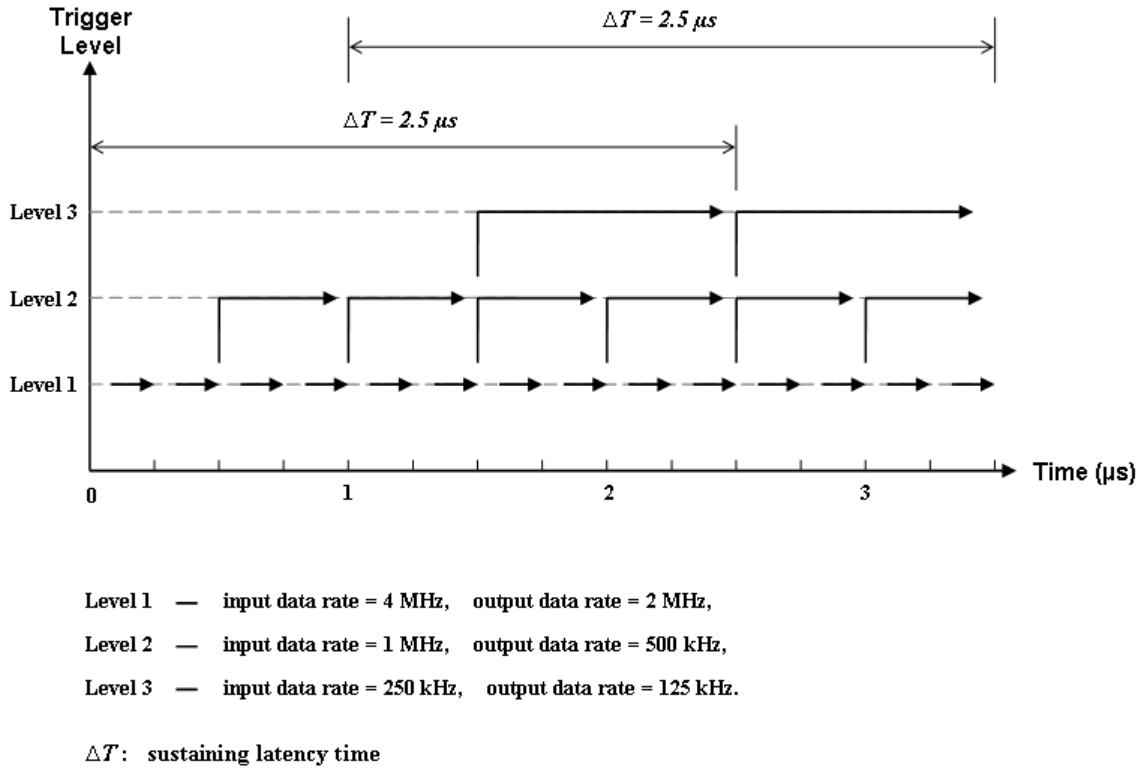

Figure 2.3: **Parallel pipeline computing of an example staged trigger system. Original data rate is sustained with fixed latency time.**

system. The reduction of data rate and the sustaining latency time are indicated in the figure.

2.2 ATLAS - A Toroidal LHC ApparatuS

2.2.1 The Detector

A Toroidal LHC ApparatuS (ATLAS) is one general purpose particle-interaction detectors constructed inside the Large Hadron Collider. The ATLAS detector has a shape of a cylinder, with a length of around 45 meters and a diameter of 25 meters. The detector weighs about 7,000 tons. Around 2,000 scientists and engineers from 151 institutions in 34 countries get involved in the development of the ATLAS detector. Physicists expect to use this detector to measure phenomena that involve highly massive particles which are not measurable in earlier lower-energy accelerators. The experiment might even shed light on new theories of particle physics beyond the Standard Model.

When the proton beams produced by the Large Hadron Collider interact in the center of the ATLAS detector, a variety of different particles with a broad range of energy levels may be generated. ATLAS is designed as a multipurpose detector. It is capable to detect and measure new physical phenomena predicted by currently available theories and to perform Standard Model measurements of high precision. At the same time, it is also open to unexpected signals from unpredicted physics scenarios and thus has to be sensitive to any kind of event topology. Therefore, rather than focusing on a particular physical process, the ATLAS detector is designed to measure the broadest possible range of signals. This means to ensure that, whatever form of new physics processes or particles take place, the ATLAS detector must be capable to detect them and measure their properties. Designs of detectors for earlier colliders, such as the Tevatron and LEP, were based on a similar philosophy. However, the new challenges of the LHC are its unprecedented energy scale and extremely high rate of collisions, which require the ATLAS to be larger and more complex than any detector ever built.

Since the ATLAS detector is expected to investigate a broadest range of physical signals, a number of special sub-detectors are constructed inside ATLAS. The sub-detectors are placed in several layers around the interaction point where the proton beams collide. Figure 2.4 shows the profile view of the ATLAS model.

The ATLAS detector can be divided into four major parts: the inner detector, the calorimeters, the muon spectrometer and the magnet systems. Each of these is further made up of multiple layers. The detectors are complementary: the inner

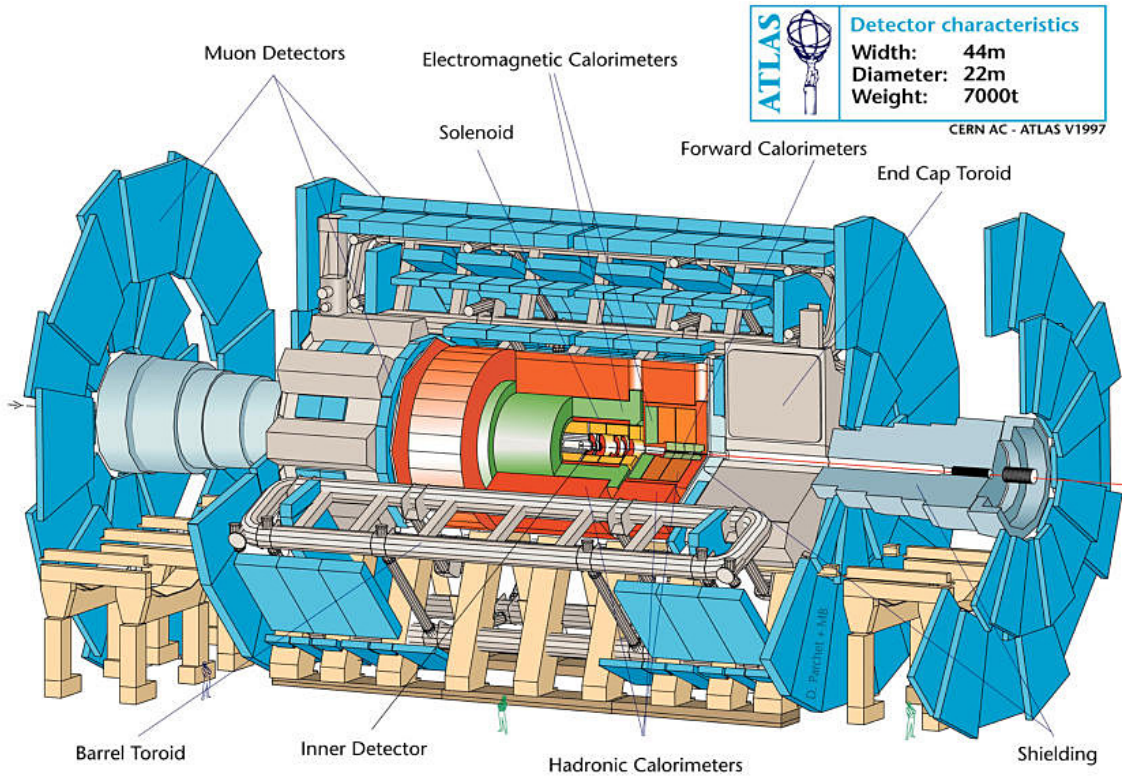

Figure 2.4: **ATLAS Model** [59]

detector tracks particles precisely, the calorimeters measure the energy of easily stopped particles, and the muon system makes additional measurement of highly penetrating muons. The magnet systems bend charged particles in the inner detector and the muon spectrometer, which allows their momenta to be measured.

The ATLAS detector produces an overall data volume of 40 terabytes every second. The data are delivered by the sub-detectors inside ATLAS, with an event rate of 40MHz and a data volume of 1 megabytes each cycle.

2.2.2 TDAQ - Trigger and Data Acquisition Chain

As with other high energy physics experiments, the ATLAS shares a same principal task in the data acquisition process: reducing the huge data volume and the high data rate. The staged trigger architecture has been commonly used in these experiments. The ATLAS data acquisition system follows also this principle.

In general the ATLAS trigger system is composed of three trigger levels. The trigger system attempts to use simple information to identify online the most interesting events that occur in the center of the detector through the beam intersections. The three trigger levels are level 1 trigger, level 2 trigger and the event filter (EF). The last two levels of the trigger system are also termed as High-Level Trigger (HLT), since both of them involve asynchronous intensive computing process, while the first stage is more synchronous hardware driven. Each trigger level refines the decisions made at the previous level(s) and applies additional selection criteria. Figure 2.5 shows the ATLAS trigger system and its data acquisition chain.

The final goal of the ATLAS system is to turn the pattern of signals from the detector into physics objects, such as jets, photons, and leptons. In the level 1 trigger physics objects are typically first identified and crudely reconstructed.[63] The high-level trigger progressively refines the reconstruction, rejects fake objects and improves the precision of the measurement. Inside ATLAS, the data-crossing rate is 40 MHz and on average about 23 proton-proton collisions will be produced at each proton bunch crossing at the machine's design luminosity of $10^{34} cm^{-2} s^{-1}$. The level 1 trigger makes the first level of event selection, reducing the initial event rate to about 100 kHz. After HLT the rate of selected events is reduced to hundreds of hertz for permanent storage. Altogether the TDAQ system in ATLAS reduces the data rate by a factor of 10^5. After the third level trigger only a few hundred events remain to be stored per second for further offline analysis. Even So, the remaining data still requires over 100 megabytes of disk space per second or over 2 petabytes (10^{15}) each year.

Offline event reconstruction (or physics object reconstruction) is performed on

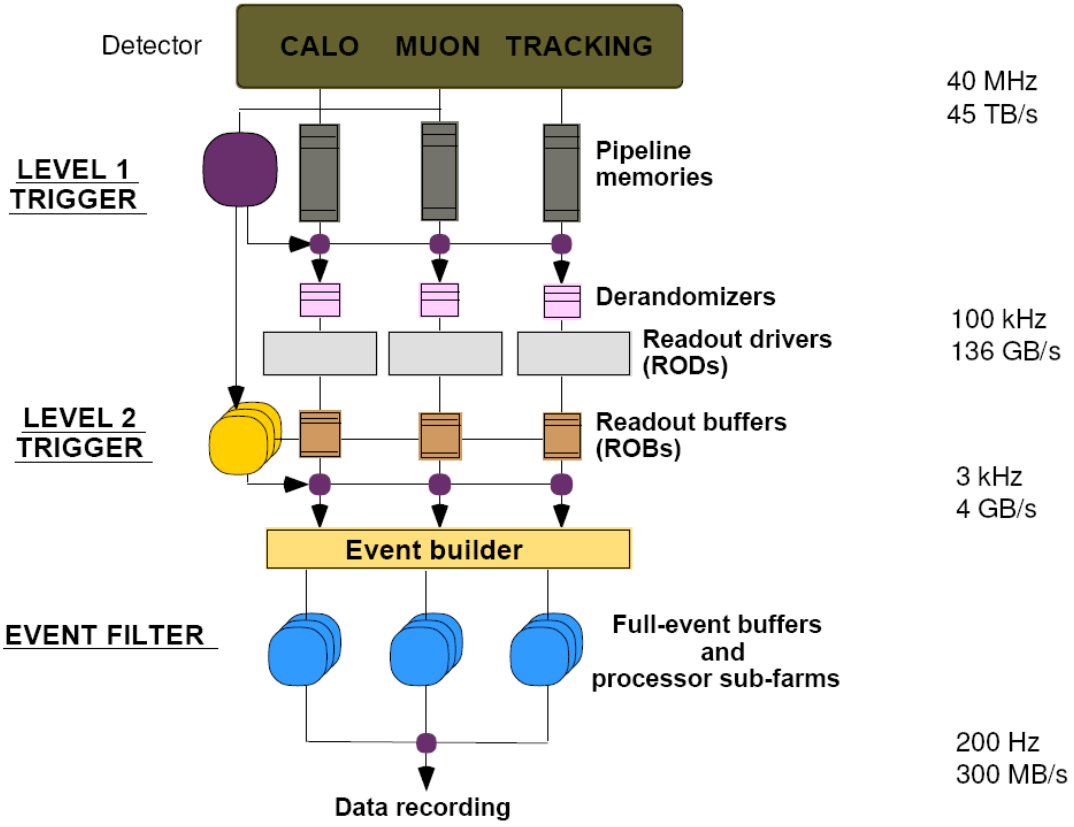

Figure 2.5: **The ATLAS trigger system and event data acquisition chain [59][53].**

all permanently stored events. Grid computing is extensively applied to event reconstruction, which allows the parallel use of the computer networks in different universities and laboratories throughout the world. This CPU-intensive task is to reduce the large quantities of raw data into a form that is suitable for physics analysis. The software for these tasks has been under development for many years, and will continue to be refined once the ATLAS experiment starts running.

The following discusses in detail the three-level triggers in the ATLAS data acquisition chain.

2.2.2.1 Level 1 Trigger

The *level 1 trigger* is based on electronics inside the detector. Special-purpose processors are used to act on a subset of the data from the detector. The level 1 trigger examines event data from the ATLAS calorimeter and muon sub-detector, and analyzes threshold information of energy and momentum to find possible particles and to make a trigger decision. It takes the level 1 trigger around 2 μs per cycle to collect data from detectors, to make trigger decision and to distribute the selected data. This time is called the level 1 latency. As this is longer than the proton bunch crossing time (i.e. 25 ns), pipeline memories are used firstly to store the events from multiple bunch crossings. Then the arithmetic logics for making the trigger decision are implemented with synchronous, pipelined, parallel processors (such as ASIC and FPGA processors) driven by the LHC 40 MHz clock. After the level 1 trigger, the data volume drops from 45 TByte/s to 136 GByte/s, and the data rate is reduced from 40MHz to 100kHz.[7]

On level-1 acceptance the level 1 trigger passes over the accepted event data to the readout drivers (ROD) along their according region-of-interest (RoI), i.e. the coordinates of the detector area where the events have been detected. The RoI restricts the area and thus the event data fraction. The level 2 trigger requires the information to make further trigger decision.

Altogether 1600 readout drivers are built in. Their tasks are to pre-format the event data, to provide a general interface from the detector to the DAQ system and to de-randomize the event data. Accordingly 1600 readout links (ROL) [4] transport the event data over a distance of up to a few hundred meters from the RODs to the readout buffers (ROB), which are temporary data buffers for the level 2 trigger to request the data and make further trigger decisions. The applied technology for the ROLs is SLink, a custom unidirectional point-to-point link standard developed at CERN [21].

2.2.2.2 Level 2 Trigger

The second stage of the ATLAS DAQ chain is the level 2 trigger. The level 2 trigger is a program driven trigger. It uses full-precision data from most of the detectors, but examines only the event data occurring inside a certain detector region, i.e. Region of Interests(RoI) as mentioned above.[15] The introduction of RoI alleviates the bandwidth and processing-power requirements of the level 2 trigger dramatically, because individual RoIs can be analyzed independently. At the same time the amount of data, transferred between the ATLAS readout buffers (ROB) and the level 2 trigger processors, is also reduced to a great extent.

One goal of the level 2 trigger is to reduce the event rate from 100 kHz further down to $1 - 5$ kHz, a rate that can be sustained by the following event building system. Different from the previous trigger, the level 2 trigger performs asynchronous operations on events with an average trigger decision latency of $1-10$ ms. To increase the throughput and meet the incoming data rate, parallel computing is exploited again. That is, the trigger runs on a large farm of dual-CPU PCs connected over a Gigabit Ethernet network. A number of supervisor PCs control the level 2 farm, and distribute the trigger tasks and the RoI information collected by the RoI builder at the previous trigger level.

The overall dataflow process is as follows. On event acceptance decision made by the level 1 trigger, the event data are moved out of the pipeline memories and stored in the ROBs, until they are cleared by a level 2 reject signal or moved on to the next stage of event builder (EB).

Due to the RoI concept only a small number of ROBs, which cover the desired area of the detector, are required. The amount of data to the level 2 processor is very limited. Modelling effort within the ATLAS community estimates, up to seven percent of all event data arriving at ROBs will be required by the level 2 trigger on average [12] and up to three percent will be accepted by the trigger decision [45].

2.2.2.3 Event Filter

The final stage of the ATLAS TDAQ chain is the Event Filter (EF). The EF analyzes the whole event data to make the final selection of events that are to be recorded for offline analysis.

The ATLAS event filter receives input data from the event builder (EB). The task of the EB is to gather data fragments that belong to a same event from buffers that are dispersed by the previous triggers, because the EF needs to consider all the event data from all sub-detectors for its analysis. Depending on the type of event, data volume reduction is achieved by a combination of event selection and possibly

event compression.

The data selection principles used by the EF resemble the offline algorithms that are used for the offline analysis of the already recorded data. The trigger decision making by the EF is also highly computing-intensive. Therefore, again a large cluster of general-purpose computers are employed. The farm of PCs build up a loosely coupled parallel system. Once an event with its full data assembled by the event builder is assigned to a PC, the PC starts working on it with no further communication with other PCs.

The EF is expected to achieve a further data-rate reduction by a factor of $10-20$. This will lead to a final event rate of up to 200 Hz. A data volume of up to 300 megabytes will be written to the permanent storage medium per second for future offline analysis by physicists.

2.3 ROS - ATLAS Readout Subsystem

As mentioned above the ATLAS readout buffer (ROB) is a temporary data buffer inside the second ATLAS trigger level. It receives the level-1-accepted event data from the readout drivers (ROD) through readout links (ROL) and buffers the data, until they are deleted or forwarded on request to the level 2 for trigger decision or to the event builder (EB) for event building.

For each ROD or ROL there is a dedicated readout buffer (ROB). That means, for 1600 ROLs 1600 ROBs are required for the buffering of the data accepted by the level 1 trigger. Besides ROBs and level 2 PC farm there exists another core device in the second trigger level of ATLAS. The device is called readout subsystem (ROS). The ROS device takes over the control and management functionalities of the ROBs and makes the data available to the level 2 trigger on demand. Due to the introduction of ROS the level 2 PC farm may concentrate only on the data analysis algorithm for trigger decision making.

2.3.1 Requirements

The ROS must fulfill a number of requirements in terms of performance and usability. Figure 2.6 shows the dependencies of the ATLAS ROS and external DAQ components. The readout drivers (ROD) are at the input side of ROS, while the level 2 PC farm and the event filter are located at the output side.

The readout driver sends event data, accepted by the level 1 trigger, to the readout buffer inside the ROS device. The data is transferred through 1600 readout links.

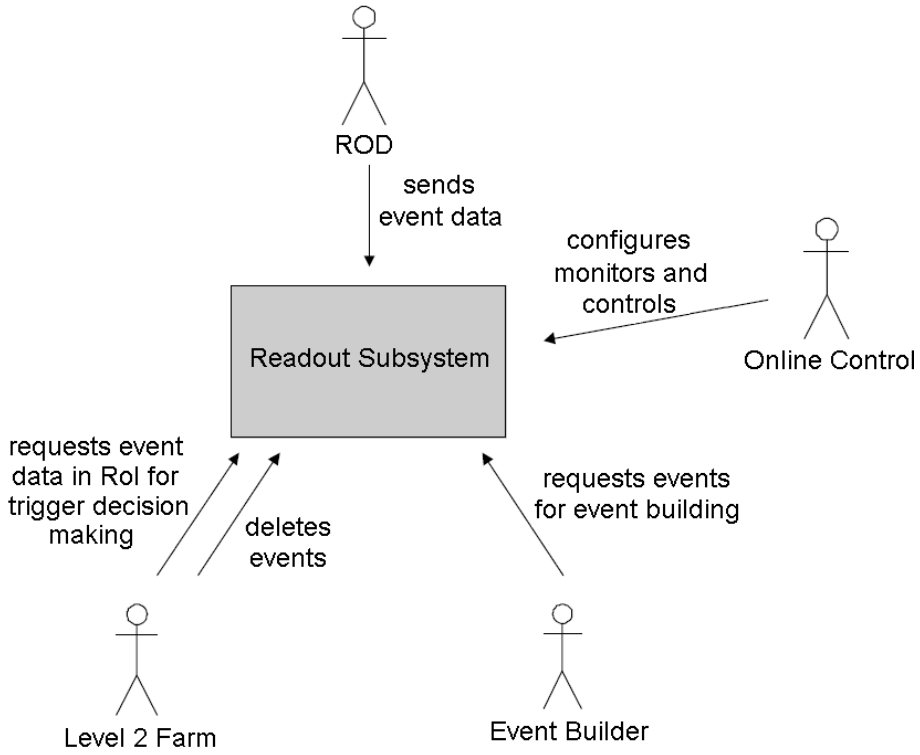

Figure 2.6: **A general use case diagram of ROS [37].**

Each ROL has a nominal bandwidth of 160 MByte/s and a data rate of up to 100 kHz.

Level 2 farm PCs collect required event data fragments from the ROS devices. It is estimated that up to 7% of all event data arriving at ROS will be requested by the level 2 farm PCs on average for trigger decision making [12]. The maximum latency time of the level 2 processing is expected to be up to 10 ms. Therefore, the readout buffers must be large enough to keep the event data at least for the latency time.

The event data, which are accepted by the level 2 trigger decision, are requested and transferred from the ROBs within the ROS to the event builder. Event fragments are delivered through a dedicated Gigabit switched network, interconnecting all the ROS and all the Event Builder PCs. On average about 3% of all events arriving at the readout buffers are accepted by the level 2 trigger and leave for event building. All event data rejected by level 2 will no longer be used within the ATLAS DAQ and are deleted at the ROB level.

At last the ATLAS online control system is responsible for ROS configuration and control. This requires the ROS to provide an implementation with a generalized software interface. With the interface the online control can pass configuration data and switch between various run levels. Errors are also reported through this interface [12].

2.3.2 Implementation

The readout buffer system aims to store the detector data before event building. It is also an indispensable component in other high energy physics experiments. One common feature of readout buffer systems is their high input rate and low output rate. Therefore, combining a number of ROL inputs to one network output is a basic implementation rule in all high energy physics experiments. Most readout buffer components use bus-based systems, e.g. VME bus crates with a number of custom readout modules. In some cases the bus system is changed from VME to PCI and the crate is replaced by a standard high performance PC. The readout modules are usually built on FPGAs.

The readout buffer component in the ATLAS trigger is more demanding compared with that of the other experiments. First, it supports “sequential selection”. That means, the buffer component does not forward all the incoming event data for trigger decision, but it supplies only these event data that are requested for trigger decision. The sequential selection brings extra complexity to the design of the readout buffer system. Only one already operating experiment has a similar readout buffer mechanism, which is the HERA B experiment. Its readout buffer system is based

on SHARC DSP processors placed on VME crate modules. The incoming event rate to this readout buffer system is 50kHz, and the SHARC links support 40MByte/s. This performance requirement is, however, much less demanding than that of the ATLAS readout buffer subsystem, which must support an incoming event rate of 100kHz and a bandwidth of up to 160MB/s per link.

Besides, the introduction of the RoI concept to the ATLAS DAQ also reduces the amount of necessary data transportation dramatically. This is also rarely found in other high energy physics experiments.

Many approaches have been discussed by the ATLAS community to implement the ATLAS readout subsystem, and several ROS prototypes based on different kinds of buses have been built and investigated, including VME-based ROS [23][25][3], PCI-based ROS [20][19][18] and CompactPCI-based ROS [56].

After intensive discussion and investigation the ATLAS community made a decision on the ATLAS ROS baseline architecture, which has been presented within the Trigger/DAQ Technical Design Report [12]. Considering the total estimated system cost, the satisfaction of performance requirements, as well as the influence on other parts of the ATLAS DAQ, a PCI bus-based solution with a standard, commercial off-the-shelf PC (i.e. ROS-PC) was chosen for the baseline implementation of the ATLAS ROS [52][1][5]. This ROS architecture is relatively cheap. Besides, it also provides the possibility to perform local, partial event building inside the ROS-PC, such that the size of the PC farm in the event builder can be reduced. The detailed decision making for the final ROS implementation strategy was discussed in [53].

Figure 2.7 shows a ROS device implementation of the baseline design. The centric control of a ROS device is based on a commercial off-the-shelf (COTS) high performance server PC with PCI buses of high I/O capability. Since this standard PC is unable to handle the event data coming from the readout drivers on a reasonable number of links, the PC has to be extended by four custom hardware - PCI boards, called ROBIN (ReadOut Buffer INput). Next chapter will address the hardware design of the ROBIN device.

The ROS PC host has three main tasks. It listens to requests from the level 2 PC farm and the event builder, and then distributes the request messages to the according individual ROBIN boards. Moreover, a local, partial event building is performed on the returning event data within the ROS-PC.

The tasks of each ROBIN board are to receive incoming event data directly from three ROLs that are connected to it and to execute and reply the level-2 trigger requests (e.g. RoI data requests) and the EB requests (e.g. EB data requests based on level 2 acceptance decision). Due to the high data rate of up to 100kHz and

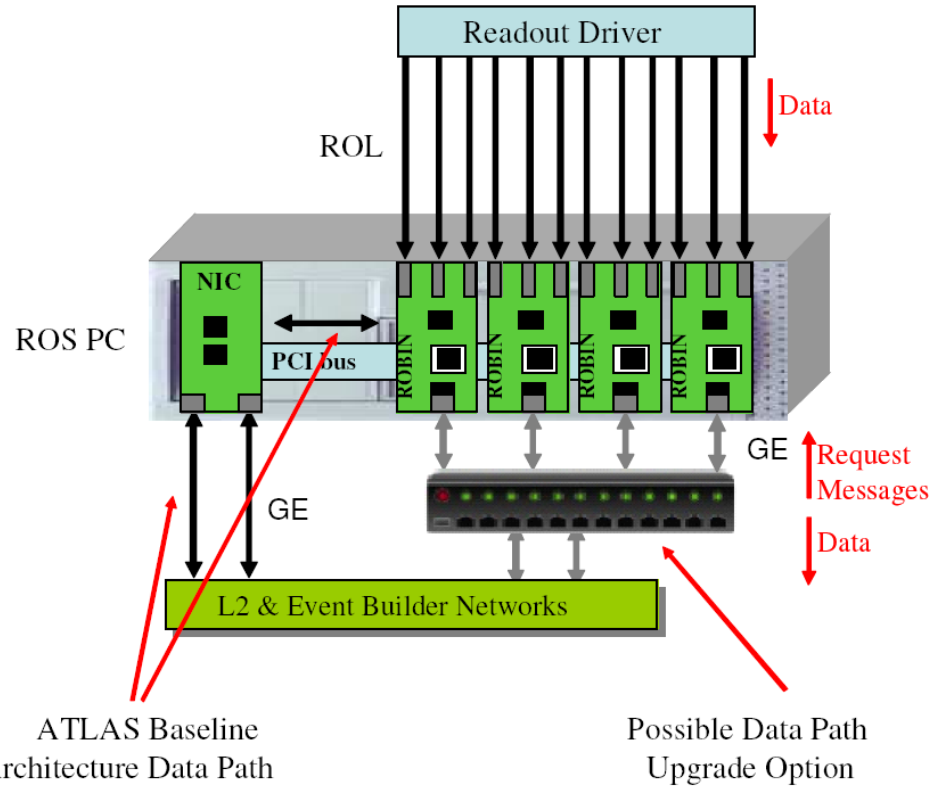

Figure 2.7: **A ROS baseline architecture [45]. Note, one ROS device may contain three or four ROBIN boards.**

the high bandwidth of up to 160MB/s per link, one important issue for the ROBIN board is to manage the buffering of the huge volume of event data arriving on the ROLs efficiently. Moreover, on the output side, the ROBIN must respond to the requests coming from ROS PC quickly.

In the ROS implementation all data requests from the level 2 PC farm or the event builder are sent to the ROS PC; and the ROS PC collects the relevant data fragments from some or all of their ROBINs and forwards them to the requesters. In this scenario each ROS PC will need to be connected to at least two network switches, through a multi-channel network interface installed on its PCI-E bus. The ROBINs have also their own on-board Gigabit Ethernet interfaces allowing an upgrade path where some data is passed into the network directly from the ROBIN.

Currently, each ROBIN device contains 3 ROBs connecting with 3 ROL channels, respectively. The output of the ROBIN is done via PCI to the host PC. Each ROS device has three or four active ROBIN devices.

2.4 Summary

The Large Hadron Collider is expected to become the world's largest and highest energy particle accelerator. The ATLAS detector, as one of the largest particle detectors of the LHC, has to confront unprecedented challenges in its trigger and data acquisition chain (TDAQ), due to its extremely huge data volume of extremely high data rate.

The ATLAS readout subsystem (ROS) is a core device in the ATLAS data acquisition chain. Its essential role in the DAQ is to play as a temporary data buffer for the level 2 trigger. It receives level-1 accepted event data through readout links at a data rate of up to 100 kHz, buffers the data temporarily for at least a latency time of up to 10 ms, delivers up to 7% of the received data (on level-2 request) to the level 2 PC farm at a rate of up to 7 kHz, and delivers up to 3% of the received data (on level-2 acceptance) to the event builder at a rate of up to 3 kHz.

The ROS device is composed of a COTS high performance server PC and three or four custom hardware PCI boards, i.e. ROBINs. The ROBIN boards provide the most essential buffering functionality inside ROS. Details about ROBIN are addressed in the following chapters.

3 Design of ATLAS ROBIN

The ReadOut-Buffer INput (ROBIN) is the core device inside the ATLAS ReadOut Subsystem (ROS). It receives and buffers incoming event data directly from readout links (ROL), and executes and replies request messages from the level 2 PC farm and the event builder. As mentioned in the previous chapter a PCI bus-based solution with a standard commercial off-the-shelf PC (ROS-PC) was eventually chosen for the implementation of the ATLAS ROS. This chapter discusses mainly the ROBIN designs for the PCI bus-based ROS system.

In the following of this chapter, different ROBIN designs for the PCI bus-based ROS are firstly reviewed in section 3.1. Then the choice for the final design of ROBIN is grounded in section 3.2. Details about the system architecture, board design and data flow of the final ROBIN device are addressed.

3.1 Previous Implementations of ROBIN

Different approaches for the implementation of the ATLAS readout subsystem (ROS) were investigated by the ATLAS community. For different ROS implementations, dedicated custom ROB input modules with the evaluated bus interface have been developed. The readout buffer input module is called ROBIN (ReadOut Buffer INput). Table 3.1 lists the different ROBIN modules implemented for different ROS designs.

Considering the entire system cost and system performance, the ATLAS community decided for a ROS baseline design based on a COTS PC with PCI bused of high I/O capability. This chapter discusses primarily the different ROBIN implementations which are specifically designed for the PCI bus and PC-based ROS system. To understand the different ROBIN designs, we firstly need to make clear the role and tasks of the ROBIN device inside the ROS system.

ROS Implementations	ROBIN Implementations
CERN VME ROS	MFCC-Based ROBIN
	UK ROBIN
Saclay CompactPCI-Based ROS	PMC-Format ROBIN
PC-Based ROS	SHARC DSP-Based ROBIN
	UK ROBIN
	FPGA-Based ROBIN
	Final ROBIN

Table 3.1: Different ROBIN modules implemented for different ROS designs.

The ReadOut-Buffer INput (ROBIN) is a core device inside the ATLAS ReadOut Subsystem (ROS). It receives and buffers incoming event data directly from readout links (ROL), and executes and replies request messages from the level 2 PC farm and the event builder. In more detail each readout link (ROL) delivers event data, which has been accepted by the previous ATLAS level 1 trigger, at a maximum rate of 100kHz and bandwidth of up to 160MB/s. ROBIN accepts the event data through the ROLs. It is expected to sustain the maximum data rate and bandwidth and store the complete incoming event data temporarily on the 64MB SDRAM buffers, and then forward the accepted data on request to the ROS PC through PCI buses. The ROS PC is on the other hand responsible to collect the relevant data fragments from the ROBIN boards and forward them to the requesters, i.e. the level 2 PC farm or the event builder [64].

According to Table 3.1, besides the final ROBIN design, there are another three previous ROBIN prototypes designed for the PCI bus and PC-based ROS system. They are the SHARC DSP-based ROBIN, the UK ROBIN, and the FPGA-based ROBIN. These different designs are reviewed in the following.

3.1.1 SHARC DSP-Based ROBIN

The SHARC DSP-based ROBIN has been developed by the NIKHEF institute [20]. It comprises an Altera 10k FPGA, a SHARC DSP, and 1MByte ZBT SRAM. The component diagram of the SHARC DSP-based ROBIN is shown in Figure 3.1.

The FPGA handles the input data stream from one SLink readout link (ROL) and accesses the event data buffer (i.e. the 1MByte SRAM) directly for event reading and writing. The SHARC DSP is responsible for the event buffer management and request messages handling. The event data buffer is organized as a ring buffer

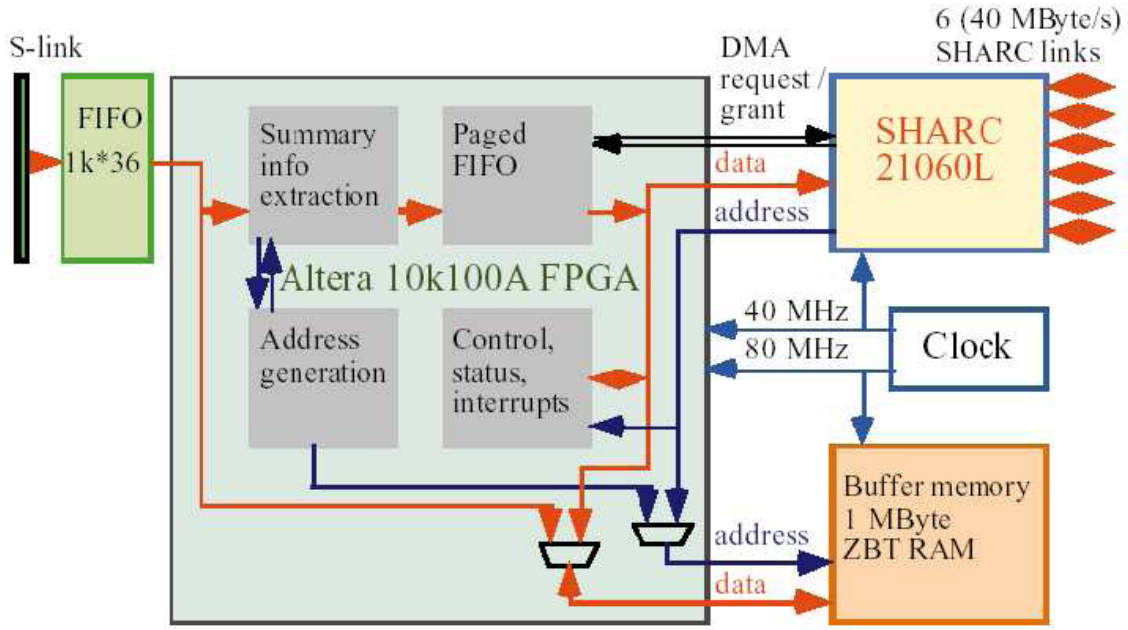

Figure 3.1: **PCI ROBIN based on a SHARC DSP** [20].

with two pointers, pointing at the beginning and the end of the empty buffer area, respectively. Up to four SHARC ROBIN boards can be combined to connect with one PCI interface.

3.1.2 UK ROBIN

The UK ROBIN was developed by the Royal Holloway University of London and the University College of London [19]. This ROBIN module is based on an i960 processor with PCI bus interfaces. For the buffering of the event data an SRAM of one megabytes is applied.

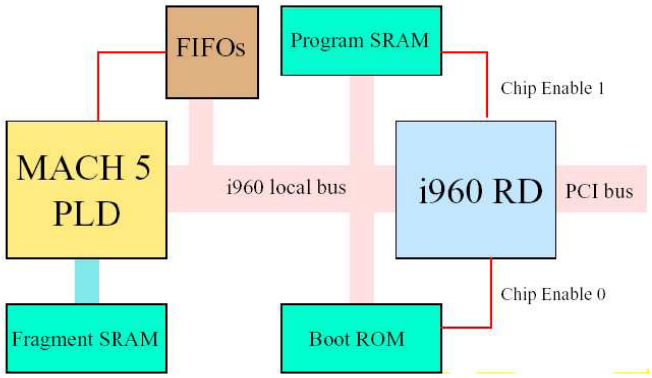

Figure 3.2: **UK ROBIN based on an Intel i960 processor [25].**

The SRAM event buffer is organized as 1024 pages and each page has 1024 bytes. The data stream from the connected readout link (ROL) is routed to the event buffer by a control logic, i.e. a MACH 5 PLD (Programmable Logic Device). The control logic maintains two FIFOs, to store the empty pages and the filled pages, respectively. Through the two FIFOs the control logic determines which page in the buffer to write. Figure 3.3 shows the mechanism of the event buffer management. On the hardware side, for each incoming event data fragment a free page is allocated from the free-page FIFO. With the free page the control logic directs the event data to the page address in the event buffer, and a new used-page record is generated inside the used-page FIFO. The software part on the Intel i960 processor is then notified of the arrival of new event data through the used-page FIFO. It is also the responsibility of the software part to manage the event requests and deletions, and to supply the free-page FIFO with new free pages after event deletions. Event data leave the hardware via the PCI interface which is integrated into the i960 processor.

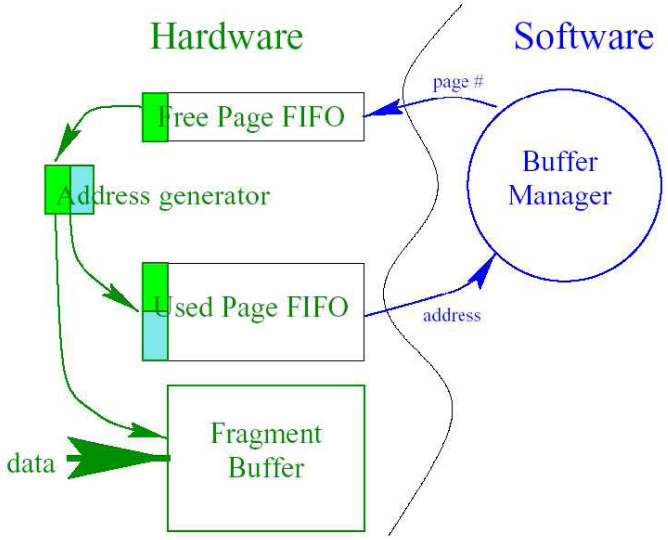

Figure 3.3: **Buffer management inside the UK ROBIN [19].**

3.1.3 FPGA-Based ROBIN

A most recent FPGA-based design of ROBIN device was presented in [53]. The ROBIN board is based on one MPRACE FPGA co-processor. MPRACE is developed as a multi-purpose PCI-based FPGA hardware. It has been used in various physics and computer science applications [46] [36] [22], and showed sufficient high performance in these systems.

Figure 3.4 shows the component diagram of the MPRACE ROBIN. Four ROLs are plugged on one of the MPRACE extension board connectors. Accordingly four independent ROL handlers are implemented inside the FPGA. They process incoming event data and store the data in four affiliated SRAMs inside MPRACE. Request messages from level 2 or event builder arrive at the ROBIN via the PCI and PLX9656 local-bus interface, and required event data is transmitted over the same interface backwards to level 2 or event builder.

3.1.4 Performance Comparison of the Previous ROBIN Designs

Performance of the previously-proposed PCI ROBIN prototypes is compared. For the evaluation a ROS-PC with equal software is used to test the three different ROBIN designs: SHARC ROBIN, UKROBIN, FPGA-based ROBIN.

For each ROBIN prototype, the maximum level 1 input rate (i.e. data rate at

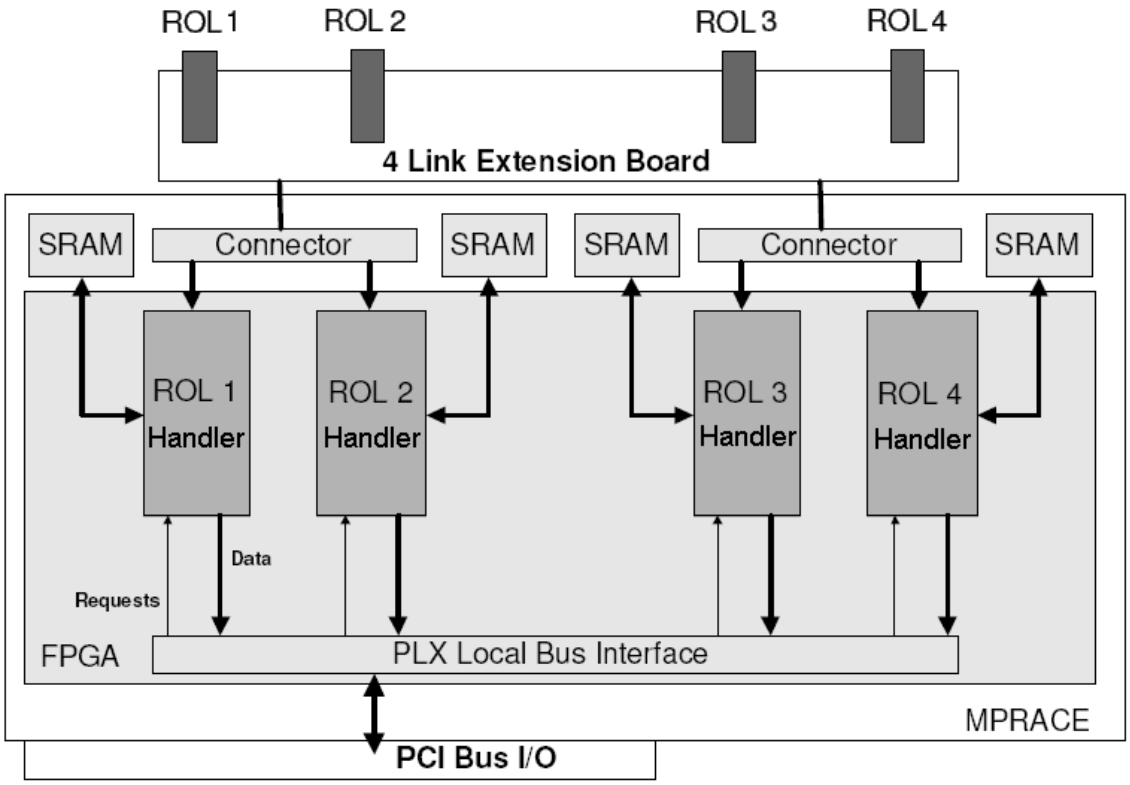

Figure 3.4: **PCI ROBIN based on a MPRACE FPGA Co-Processor [53].**

ROLs) is measured while varying request rate from level 2 and event builder. Figure 3.5 shows the results of the comparison. The FPGA-based ROBIN shows dominantly better performance over the other two ROBIN prototypes.

FPGA processors are well-known for their fast performance due to the parallel processing based on programmable gate array logics. They show significant advantages over general-purpose PCs in handling tasks of parallel computing. In the case of ROBIN the buffering of incoming event data from RODs and the response to the request messages from the ROS host PC can be executed in parallel independently. Besides, the MPRACE ROBIN is able to handle four ROLs at a maximum.

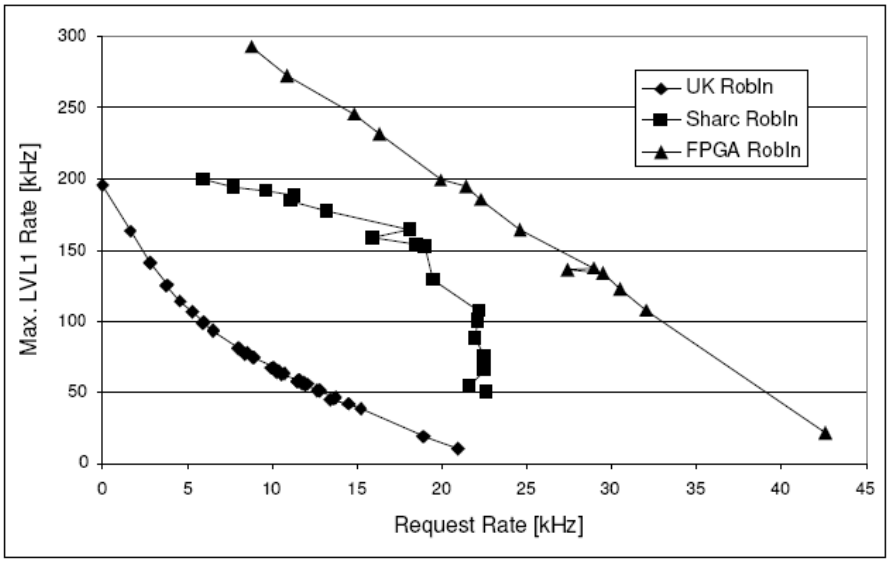

Figure 3.5: **Performance comparison of the three previous ROBIN prototypes: SHARC DSP-based ROBIN, UK ROBIN and FPGA-based ROBIN [28].**

However, to deal with complex tasks with complex data structures, it requires much FPGA resource, i.e. space of programmable gate array. Larger size of gate array means by far higher cost of a FPGA processor. The detailed ROBIN's tasks are discussed in the next chapter. It is shown that the ROBIN application employs data structures of stacks, linked lists and particularly hash tables to manage the event buffers. The implementation of all these tasks on an FPGA processor would be very resource-costly. Furthermore, the FPGA processor shows no better performance in dealing with serial tasks, since its frequency is generally lower than that of a CPU. Besides, the development cost of the FPGA code is dramatically higher than that

for a general-purpose CPU.

3.2 Final Design of ROBIN

Regarding the above considerations the ATLAS community made the decision for the final design of the ROBIN device [45] [44].

3.2.1 Hardware Deployment

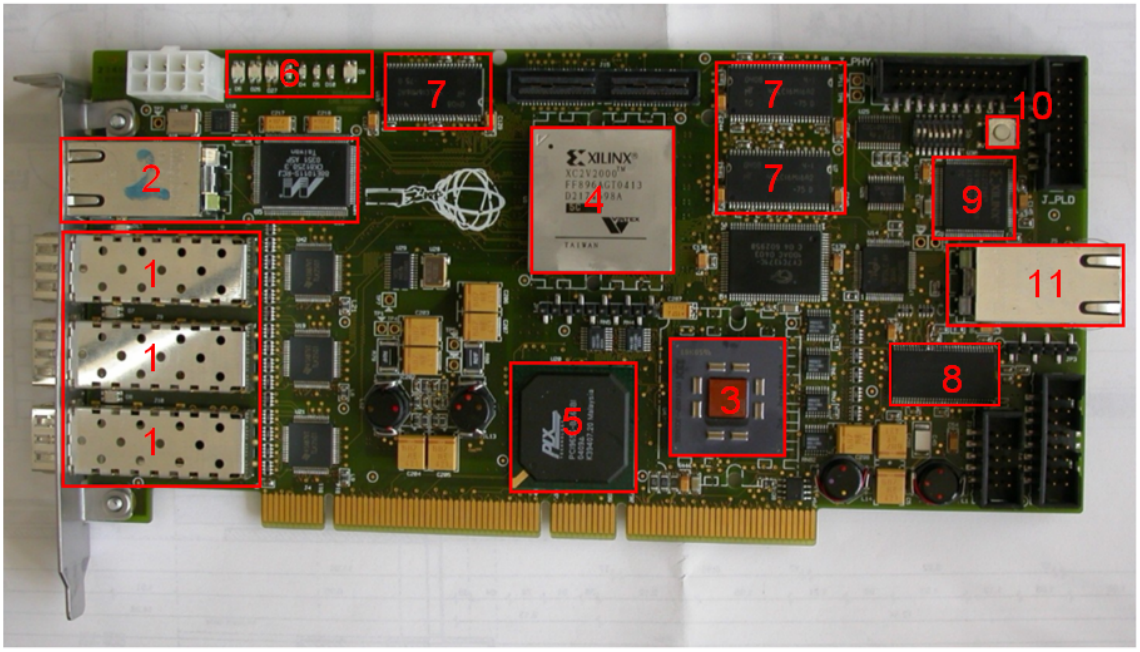

Figure 3.6: **A ROBIN prototype in the final design.**

Figure 3.6 shows a picture of a ROBIN prototype in the final design. Names of the labelled components in the figure are listed in Table 3.2. The according hardware deployment of the ROBIN board is illustrated in Figure 3.7. Two kernel processors, a Xilinx Virtex II 200 FPGA and a PowerPC 440 micro controller, are deployed on the board. The PowerPC microcontroller has an affiliated 128MByte DDR SDRAM used to store the PowerPC software. For the space consideration at most three ROLs can be connected with one ROBIN board in this design. Three 64MByte SDRAMs

1	three optical HOLA SLink input channels(160 MB/s per channel)
2	Gigabit Ethernet
3	IBM PowerPC 440GP micro-controler (466 MHz)
4	Xilinx XC2V2000 FPGA
5	PLX9656 PCI-X Bridge at 66MHz
6	LEDs
7	three 64MByte SDRAM buffers
8	PowerPC's affiliated 128MByte RAM
9	XILINX XC2C256 CPLD
10	reset button
11	PowerPC 10/100M Ethernet Interface

Table 3.2: ROBIN Components

are used for the storage of the incoming event data from three readout links (ROL), respectively.

The combination of an FPGA processor and a PowerPC micro-controller takes advantage of both kernel processors. According to the hardware deployment of ROBIN, the FPGA is mounted in the middle of the device. It acts as a bridge between three ROLs, three event buffers, the PowerPC and the ROS PC. The FPGA plays the centric role as a high-rate and high-bandwidth data-flow core. All the data transmitted between the components must go through the FPGA. It transmits event data and control messages on-the-fly across the system. On the other hand, the PowerPC micro-controller takes over more complex and flexible management and control jobs with relatively lower performance requirement. It arranges the event data buffers, instructs the FPGA where to store or transmit event data, decodes and executes incoming requests from the ROS PC, and initiates response messages with event data or status/debugging information backwards. This ROBIN design makes full use of the on-the-fly parallel processing capability of FPGA and the high clock-rate and high flexibility of PowerPC.

Moreover, compared with the previous FPGA-based ROBIN, the combination of an FPGA processor and a PowerPC micro-controller also makes the extension of the ROBIN device for a next generation development much more flexible and convenient. The first prototype stage of this final ROBIN design has already been approved [31].

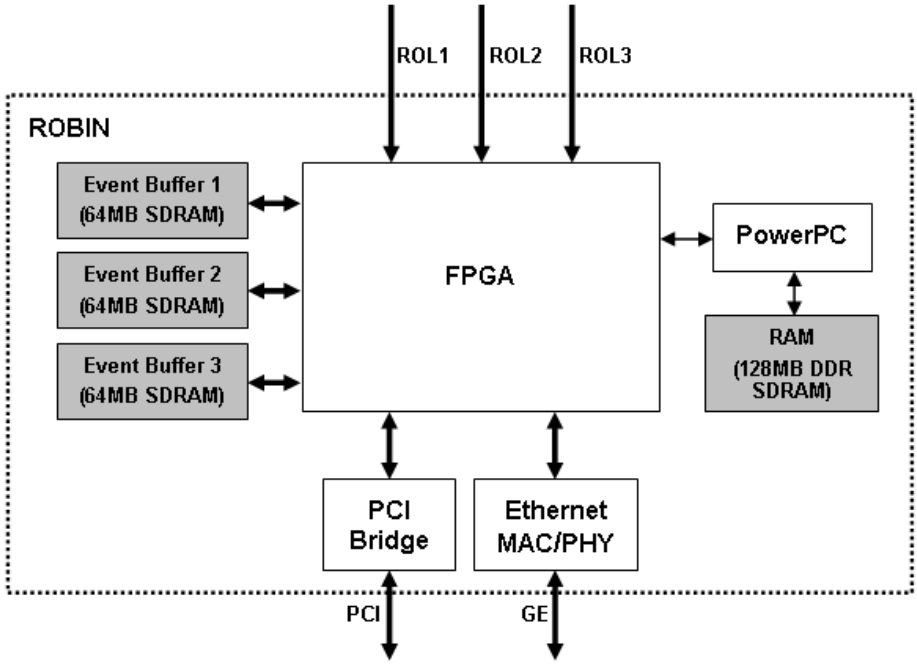

Figure 3.7: **Hardware Deployment of ROBIN. The highlighted three event buffers and PowerPC's affiliated RAM are to be organized by the PowerPC application.**

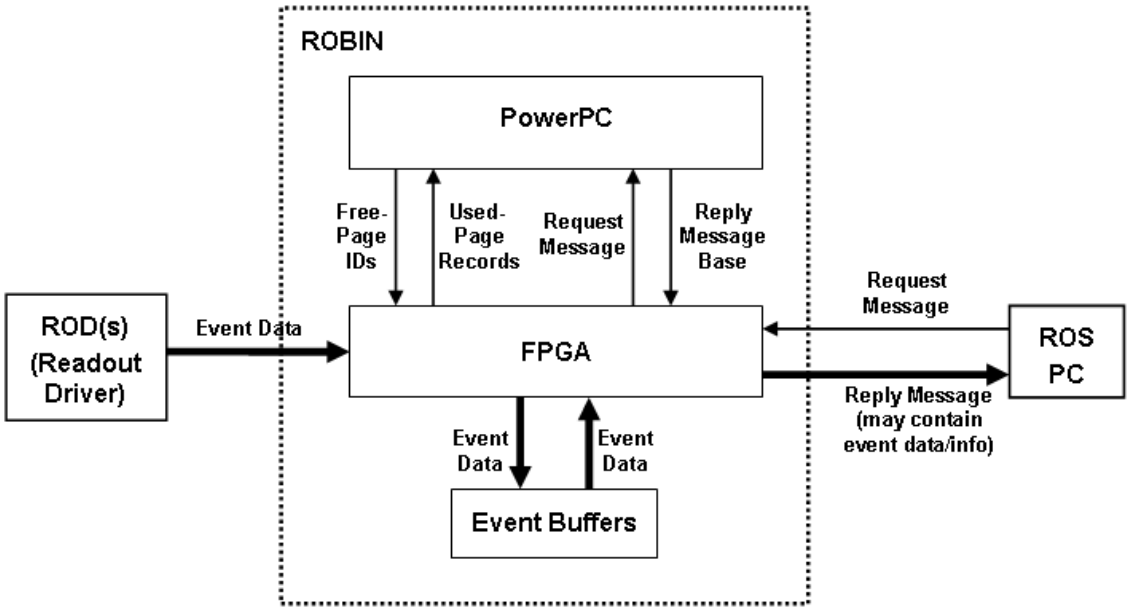

Figure 3.8: **Data Flow Diagram of ROBIN.**

3.2.2 Data Flow

Figure 3.8 shows the data flow through the ROBIN system. The bold arrows indicate the heavy traffic pathes for the transmission of event data or messages containing event data. The PowerPC processor gets never involved in the main event data path, and hence avoids being a bottleneck in the huge volume of data transmission. Only request messages and response messages excluding event data are transmitted between FPGA and PowerPC. Free page IDs and used page records are the specific control information for event buffer management. Details about event buffer management are found in the chapter 4.

3.3 Summary

This chapter reviews different ROBIN designs for the PCI bus and PC-based ROS and presents the final ROBIN design. Three previous ROBIN designs are particularly discussed. They are the SHARC DSP-based ROBIN, the UK ROBIN and the FPGA-based ROBIN. The performance comparison between the different ROBIN prototypes shows that the FPGA-based ROBIN is much efficient than the other two ROBIN designs.

However, due to the higher resource cost and the higher development cost of the FPGA-based ROBIN, the ATLAS community decided eventually to adopt the final ROBIN design based on two kernel processors: a Xilinx Virtex II 200 FPGA and a PowerPC 440GP micro-controller. This combination takes advantages of both kernel processors. The former controls the data flow with high performance requirements, and the latter is responsible for more complex and flexible management functions with relatively lower performance requirement. The final ROBIN design keeps not only the efficiency of the FPGA-based ROBIN, but also makes the extension of the ROBIN device for a next generation development much more flexible.

4 Event Buffer Management Algorithms

Before introducing the detailed software design of the ROBIN PowerPC System, this chapter discusses firstly the strategy of the ROBIN event buffer management and the algorithms involved in the effective buffer management and efficient event lookup.

Firstly, section 4.1 reviews the page-based strategy for event buffer organization, which was originally introduced by the UK group to the ROBIN project [19]. Section 4.2 presents a hash-table-based algorithm for fast event lookup inside event buffers. Section 4.3 proposes a storage strategy both for the hash-table storage management and for the event buffer storage management.

As mentioned previously three ROLs are connected with one ROBIN board. Event data from different ROLs are stored in different buffers and handled separately by the level 2 trigger. Accordingly, the management of different event buffers is also handled separately but in a similar way. The following text of this chapter addresses the management of one event buffer against one ROL.

4.1 Page-Based Event Buffer Organization

The event buffer management strategy presented in this work is based on a page-based buffer management scheme, which was originally introduced by the UK group [19] to the ROBIN project. Since then some improvements have been made and applied to the final design of ROBIN.

The page-based buffer management scheme is to segment each event buffer memory (i.e. a 64MB SDRAM) into pages of fixed size. Each page has a unique ID and is the smallest unit for buffer allocation. The size of each page is typically 1K

bytes, 2K bytes or 4K bytes. It is determined a priori depending on the type of the sub-detector that is connected to the corresponding ROL/ROD, since different signal detectors generate different sizes of event data. One event may occupy one or more pages, and one page can only be occupied by one event. To simplify the explanation, in the following it is assumed that each page is of 1K bytes. In this case there are totally 64K pages, and the page ID can be expressed by a 16-bit number. The absolute buffer address (up to 64M) can be easily computed through a bit-shift operation upon the 16-bit page ID by 10 bits to the left.

For a certain time period each incoming event from a single ROL has a unique 32-bit event ID. The event ID is assigned by the level 1 trigger for each newly detected event. The event ID is generated with a sequentially incremental number [32]. Hence for an event rate of 100kHz the 32-bit event ID restarts from zero around every 12 hours. Because the ROS PC sends periodically event-delete request messages to remove obsolete event data [54], it can make sure that an old event has already been processed and removed long before a new event with a same event ID is generated. That means, at any moment in one event buffer each event has a unique event ID.

As mentioned above, the data of one event is stored in one or more pages. Hence there exists an one-to-many mapping between 32-bit event IDs and 16-bit page IDs, which makes it possible to retrieve the according event data in the event buffer. The mapping between event IDs and page IDs is managed by the PowerPC application using a hash function. The hash function will be introduced in the next section.

Through a **free-page ID FIFO** the PowerPC application tells the FPGA which pages in the event buffer are free to store the data of newly-incoming events. When a new event arrives, the FPGA removes a free-page ID from the FIFO, and stores the event data to the corresponding page in the event buffer. After the storage the FPGA forwards accordingly a used-page record to the PowerPC application through **a used-page record FIFO**. A used-page record contains the event ID and the occupied page ID. If more than one pages are needed for the storage of the new event, the FPGA will take another free-page ID from the free-page ID FIFO and go through a same procedure as above. More details about the communication between the PowerPC application and the FPGA inside ROBIN are found in chapter 5.

4.2 Hash Table for Fast Event Lookup

Given an event ID, the storage location of the event data in the event buffer (i.e. the related page IDs) must be found out immediately for efficient event data transmission or deletion on request by the level 2 trigger. For this purpose a skillful management

of the one-to-many mapping between event IDs and related page IDs is necessary for the fast event lookup.

An event ID has 32 bits. It is unrealistic to set up a sparse array with a size of 2^{32} to contain a 26-bit event buffer address or a 16-bit page ID for each event. This method may provide the fastest performance for event lookup, but it requires an unavailable memory space of at least 8 TBytes.

This section presents a hash-table based algorithm for the fast event lookup. This algorithm is of not only low computational cost but also low memory space cost. The following of this section reviews firstly three standard searching algorithms and then addresses the choice of a hash table for the ROBIN event lookup. The creation of the hash table and the definition of the hash function are explained afterwards. The storage management of the hash buckets are also described. At last the complexity of the hash searching is discussed.

4.2.1 Choice of Event Lookup Algorithm

4.2.1.1 Standard Searching Algorithms

Book [62] presents in general three solutions to searching problems, including linear searching, binary search trees and hash tables. First, the *linear searching* refers to a search through a static array or a linked list where the mappings between keys and their values are stored. In the case of ROBIN event lookup the keys are event IDs and the values are the page IDs. It would be too computationally expensive to perform a linear searching through a list of the mappings between event IDs and page IDs, since the list could have a maximum length of 2^{16} for one event buffer.

Second, a *binary search tree* is a sorted two-way tree structure, where each node in the tree contains a key-value pair. "Sorted" means that, for each node in the tree, all the keys of the nodes in its left subtree are less than or greater than the keys in its right subtree. A bad search tree with n nodes may also have a complexity of $O(n)$ like the linear searching. Only a balanced binary search tree can provide a much faster performance. "Balanced" means that there are about the same number of elements on either side or subtree of each tree node. In this case a tree with n nodes has a height of $h = \log_2 n + 1$. To search for a node in the tree we need only do comparison checks for at most h times till finding the required key-node pair. The complexity of a balanced binary search tree is therefore $O(\log_2 n)$ in the worst case.

Third, a *hash table* is another data structure that associates keys with their values. In a narrow sense a hash table is merely an array of pointers. Each pointer points at a desired location, named as a hash "bucket", in which the related key-value pairs

are stored. Then given a key, a hash function is firstly required, to convert the key to a number. The number indexes into the hash table (i.e. a pointer array). The pointer in the indexed element leads to the desired hash bucket, where the key-value pair for the given key should be stored. A hash bucket can also be implemented using different data structures, such as a tree structure or a linked list of key-value pairs. As with the above binary search tree, a hash table can speed up a lookup process only if it is relatively balanced. "Balanced" here means that the key-value pairs are evenly distributed among the buckets.

4.2.1.2 Choice of Hash Searching for Event Lookup

As the memory space required for the storage of data structures is concerned, linear searching is the most economical among the above three searching algorithms. For linear searching with a linked list of key-value pairs, each node in the list contains one additional pointer pointing to the next node beside a key-value pair. The binary search tree needs to store for each tree node (i.e. for each key-value pair) two additional pointers pointing to the roots of its left and right subtrees. The hashing needs an extra hash table to store the pointers to each hash bucket. Besides, each hash bucket needs further a certain data structure for its storage. Although linear searching occupies the least memory space to store the data structure. However, the time complexity of linear searching, $O(n)$ in the worst case, is unacceptable for the real-time requirement of the ROBIN system.

A balanced binary search tree offers a time complexity of $O(\log_2 n)$, where n is at most 64K for ROBIN's event buffer management. This advantage of the balanced binary searching is very attractive. However, inside ROBIN the mappings between the event IDs and the page IDs for each event buffer are continuously modified. Many efforts have to be devoted to maintain a dynamically sorted and balanced binary search tree, both in terms of computational time and in terms of memory space. For example, besides two pointers to the left and right subtrees for each tree node, the number of the nodes inside either subtree must be stored as well for the purpose of balance evaluation; and in the case of node insertion or deletion a similar searching process has to be carried out, and so does the re-balance for each concerned subtrees. That means, operations of insertion or deletion are even more costly than the operation of node lookup.

As for hash searching, an additional buffer needs to be allocated for the storage of the hash table. Besides, if a unidirectional linked list is used to store a hash bucket, each node in the bucket contains not only a key-value pair but also a pointer leading to the next node, provided that each hash bucket is implemented as a uni-

directional linked node list. The hash searching supports a constant lookup time $O(1)$ on average, so long as a proper hash function can be found which balances the distribution of the key-value nodes throughout the hash buckets. The computational time for node insertion or node deletion is also constant, and the time complexity is $O(1)$. The hash searching method is finally chosen to solve for the ROBIN buffer management problem, because it is proven that there exists such a hash function that distributes the hash nodes uniformly into the hash buckets. The creation of the hash function is introduced in the next subsection.

4.2.2 Creation of the Hash Table

For the event buffer management problem in ROBIN the key-value pairs are the event-ID and page-ID pairs. Since the size of an event buffer is limited, there exists also an upper limitation to the number of hash nodes, i.e. key-value pairs. The limitation is the total number of pages in one event buffer, which is known as 64K (in the case that the page size is 1K bytes). Therefore, we might design a hash function such that the size of the hash table (i.e. an array of pointers to hash buckets) is equal to the total number of pages, and that each bucket contains only one hash node on average when the event buffer is full. Accordingly the index to the hash table can be expressed by a 16-bit number.

It has been mentioned in section 4.1 that each event has a unique event identifier. It is assigned by the level 1 trigger and has 32 bits. The level 1 trigger generates the event IDs increasingly according to the occurrence order of the events; and the events inside each event buffer are cleared up periodically. Therefore, one possibility is to use the lower bits of the event ID as the index to the hash table. This strategy guarantees that there is no collision of the hash indexes for a certain time span.

Since the index to the hash table is a 16-bit number, a best choice to compute the hash index is using the lower 16 bits of the 32-bit event ID. The according hash function is given as follows:

$$\text{Hash(key)} = \text{key} \ \& \ \text{0xFFFF} \tag{4.1}$$

where "&" is a bit-wise logical AND operation. Here the key is a 32-bit event ID. The result of the hash function is the index to the hash table.

With the hash index to the hash table, we can get the related hash bucket. Next subsection introduces the data structure and the implementation of hash buckets.

4.2.3 Storage Management of Hash Buckets

A hash bucket is a list of hash nodes. As described above the size of each hash bucket is less than one on average. Hence it is unnecessary to exploit a complex structure like trees to implement each hash bucket. Typically a unidirectional linked list is used as the data structure for a hash bucket, if the sizes of the hash buckets are small and uncertain. Each node in the linked list contains a key-value pair and a pointer to the next node.

A linked list is a dynamic data structure, whose size changes continuously over time. However, as is commonly known, dynamic memory allocation is discouraged inside an embedded system for security and stability's sake. Due to limited resource in an embedded system every application or software in it is assigned a priori a definite amount of memory for its code and data. When a dynamic data structure is necessary, often it has to be managed by the software itself within its assigned memory space. Mostly dynamic data structures are eventually implemented using some other static data structures in the background.

Inside ROBIN the management of the hash table is done by the ROBIN PowerPC micro-controller, which has an affiliated 128MByte DDR SDRAM for the storage of its software. The PowerPC software also avoids using dynamic memory allocation. Hence the following of this subsection presents an algorithm to manage the dynamic storage of the hash buckets using some static data structures.

4.2.3.1 Logical Structure

A hash bucket is implemented as a unidirectional linked list. The linked list is composed of a chain of hash nodes. For the hashing for ROBIN event lookup each hash node in a hash bucket contains not only a key-value pair (i.e. event-ID and page-ID pair) and a pointer to the next hash node, but also some brief information of the according event data. Excluding the pointer to the next hash node the other information are actually equivalent to that of a used page record. The used page record is generated by the ROBIN FPGA micro-controller for each occupied page inside the event buffer and forwarded to the PowerPC system through a used-page record FIFO as mentioned in section 4.1. A used page record has 16 bytes. Details about used page records are found in chapter 5.

See figure 4.1 for an example of the logical structure of the hash table applied to the fast event lookup in ROBIN.

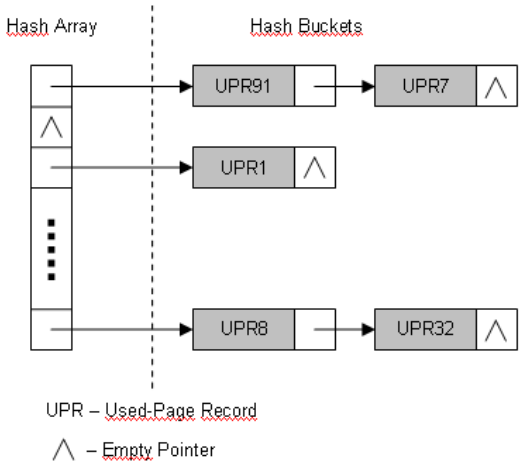

Figure 4.1: **Logical structure of an example hash table applied to the fast event lookup.**

4.2.3.2 Physical Storage

The data structure of a hash table is composed of two parts. One part is an array containing pointers to corresponding hash buckets, which is termed as hash array (or hash table in a narrow sense). The other part is the hash buckets. As mentioned in section 4.2.2, the size of the hash array is fixed, so the hash array can be defined as a static array structure. The essential problem for the storage of the hash table is the storage of its hash buckets with a changing number of hash nodes.

As the total number of the pages in one event buffer is 64K, the maximum number of hash nodes is also 64K accordingly. Then we may assign a static buffer with a size of 64K × sizeof(Hash Node) bytes for the storage of all the hash nodes. This buffer is named as the *hash node buffer* in this work. Figure 4.2 illustrates an example of the physical storage of a hash table inside the PowerPC software.

Moreover, since all the hash nodes are stored in an array, the pointer to a hash node can be expressed by the index of the node in the hash node array, instead of the physical address of the hash node in the memory. The former is of 2 bytes, and the latter is of 4 bytes. In this way the buffer sizes both for the hash array and for the hash node array can be reduced by 2*64K bytes.

The next problem is then how to manage the hash node buffer, including allocating a free space inside the hash node buffer for a new hash node, and releasing the space of a deleted node. Two solutions to the management of the hash node buffer are proposed in this work and presented in the coming section 4.3.3.

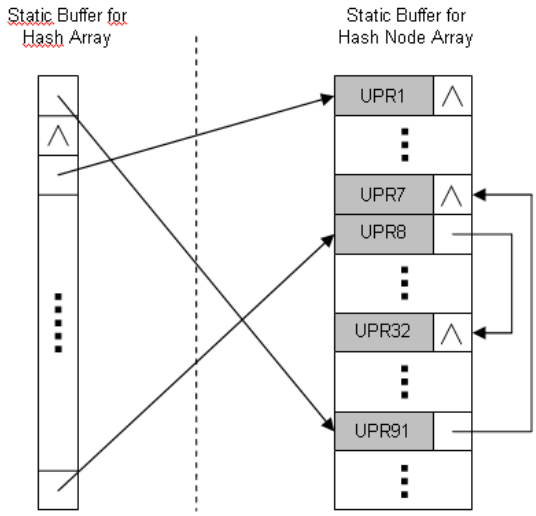

Figure 4.2: **Physical storage of an example hash table inside the PowerPC software. Note, the hash table is the same one as shown in figure 4.1).**

4.2.4 Discussion

A hash table is introduced above for the fast event lookup inside ROBIN. This subsection discusses the hash searching algorithm with respect to its hash collision probability, space complexity and time complexity.

4.2.4.1 Hash Collision

A good hash function is essential for good hash table performance. A poor choice of a hash function can lead to *clustering*, in which case the probability of keys mapping to a same hash bucket is significantly greater than that expected from a random function. This clustering atmosphere is termed as *hash collision*. Hash collision is an important factor to evaluate a hash function. A good-designed hash table attempts to avoid hash collision as much as possible. The following discusses the probability of hash collision for the hash table proposed in this section for the fast event lookup inside ROBIN.

Section 4.2.2 explains the choice of the hash function for the fast event lookup inside ROBIN. Firstly, the size of the hash table is designed to be the same as the total number of pages inside one event buffer. Therefore, on average each hash bucket contains one used page record or hash node, if the event buffer is full. Ideally there is no collision of hash nodes in any hash bucket. Moreover, equation 4.1 gives

the hash function for the hash table. It takes the lower 16 bits of a 32-bit event ID as the index to the according hash bucket. Because the event IDs are generated increasingly according to the occurrence order of the events, a hash collision with a current occupied hash node occurs once after 64K events have happened. Chapter 2 has introduced that ROBIN is essentially an intermediate buffer in the ATLAS data acquisition chain. It is designed to speed up the highly-selective event acquisition inside ATLAS. ROBIN is built between the level 1 trigger and the level 2 trigger. Statistically less than three percent of the total events get passed on from the level 1 trigger to the level 2 trigger; and event data get refreshed very often inside the ATLAS DAQ chain as well as in the ROBIN's event buffers. When historical event data get removed, the old event IDs get deleted and the chance for hash collision decreases at the same time.

4.2.4.2 Space Complexity

Table 4.1 shows the storage requirement of the hash table proposed in this section for various page sizes inside one 64MByte SDRAM event buffer. According to the table the PowerPC needs to reserve totally 1280K bytes for the storage of a whole hash table for the fast event lookup inside one event buffer, provided that the page size is 1K bytes.

Page Size	1K Bytes	2K Bytes	4K Bytes
Number of Pages	64K	32K	16K
Size of a Hash Bucket Pointer	2 Bytes	2 Bytes	2 Bytes
Size of a Used Page Record	16 Bytes	16 Bytes	16 Bytes
Size of a Hash Node	18 Bytes	18 Bytes	18 Bytes
Length of Hash Array	64K	32K	16K
Buffer Size for Hash Array	128K Bytes	64K Bytes	32K Bytes
Length of Hash Node Array	64K	32K	16K
Buffer Size for Hash Node Array	1152K Bytes	576K Bytes	288K Bytes
Buffer Size for the Whole Hash Table	1280K Bytes	640K Bytes	320K Bytes

Table 4.1: Memory requirement of the hash table for fast event lookup. Note that the event buffer is a 64MByte SDRAM.

4.2.4.3 Time Complexity

Table 4.2 lists the time complexity of different operations upon the hash table both in the average case and at the worst case. The running time for the initialization

of the hash table is in proportion to the total number of buffer pages, i.e. $O(M)$, where M is the total number of buffer pages. The time required for the insertion of one used page is fixed and independent of the number of pages, i.e. $O(1)$. The maximum response time of the two operations are predictable.

Operation	Time Complexity (average case)	Time Complexity (worst case)
Initialization	$O(M)$	$O(M)$
Used Page Insertion	$O(1)$	$O(1)$
Used Page Deletion	$O(1)$	$O(M)$
Event Lookup	$O(1)$	$O(M)$

Table 4.2: Time complexity of the hash table management for fast event lookup. Note that M is the total number of buffer pages.

Event lookup is a part of the process of event deletion. Once a requested hash node is found in a certain hash bucket, it takes no time to remove it from the bucket. Hence the time complexity of hash node deletion is similar to that of event lookup. In the average case the hash table provides constant running time, i.e. $O(1)$, both for event deletion and for event lookup. In the worst case when the event buffer is full of M event pages and all the M occupied hash nodes are clustering in one hash bucket, the worst-case time complexity is $O(M)$ for both operations. However, as indicated in the previous subsection 4.2.4.1, no hash collision occurs during a certain time span after buffer initialization or re-initialization, and periodical clear-up of obsolete event data diminishes the chance of collision. Both practically and theoretically the worst case of the hash table will never happen.

4.3 Hash Node Buffer Allocation and Event Buffer Allocation

This section presents two solutions to the hash node buffer allocation as well as to the event buffer allocation. Firstly it is shown that there exists an identical imaging between an event buffer and its related hash node buffer. Then a standard method based on a free-page ID stack is introduced for the allocation problem of both buffers. An improved algorithm is proposed afterwards in detail. The algorithm is based on a chained free hash-node list that is built inside the hash node buffer. Finally the two algorithms are evaluated and compared with respect to their space complexity and time complexity.

It has been described in the beginning of this chapter that an event buffer is a 64MB SDRAM memory and the buffer is segmented into pages of fixed size. The size of each page is pre-defined. It could be 1K bytes, 2K bytes or 4K bytes typically. To simplify the explanation, in the following of this section it is assumed that each page is of 1K bytes. In this case there are 64K pages in one event buffer.

4.3.1 An Identical Imaging between Event Buffer and Hash Node Buffer

Compare an event buffer with its related hash node buffer. The former is divided into 64K pages with a fixed size of 1K bytes per page. The latter is also divided into 64K hash nodes of fixed size of 18 bytes, in which 16 bytes is for a used page record and 2 bytes is for a node ID pointing to the next hash node. Besides, each occupied hash node in the hash node buffer corresponds to a used page in the event buffer. Naturally we may consider arranging the two buffers in a same way.

As mentioned in the previous section, the ROBIN PowerPC software needs to offer the FPGA continuously free-page IDs inside the event buffer. After copying the data of a newly-incoming event to a free page in the event buffer, the FPGA returns a used-page record to the PowerPC. The used page record is then wrapped in a hash node and inserted to the hash table. A used page record contains both an event ID and a page ID; therefore in the hash node buffer (or array) if we take the hash node also at the index identical to its page ID to contain the used page record, we can guarantee an identical imaging between the arrangement of the event buffer and the arrangement of the hash node buffer. Each used page in the event buffer has a corresponding node in the hash node buffer at a same position, and vice versa.

4.3.2 Standard Buffer Allocation Algorithm Using a Free-Page ID Stack

A standard method to manage the storage of such a buffer divided into segments/pages of fixed size is to use a free-page ID stack. The free-page ID stack is used to store the IDs of the free pages inside the event buffer. The size of a page ID is 2 bytes and there are at most 64K free pages in one event buffer. Therefore, the size of the free-page ID stack is determined to be 64K $\times$ 2 bytes.

For the initialization the event buffer is empty and all the pages in the buffer are free, then all the page IDs are pushed into the free-page ID stack. In the case of page allocation for new events page IDs are popped out of the stack, and in the case of page release for deleted events page IDs are pushed into the stack.

As stated in the preceding subsection the allocation of the event buffer can be easily extended to manage the allocation of the related hash node buffer.

4.3.3 Chained Free Hash-Node List for Hash Node Buffer Allocation

4.3.3.1 Notations

The solution proposed in this subsection for the hash-table storage management employs same data structures as introduced in section 4.2 for the hash table. The hash table is notated by a pair of arrays $< H, G >$. In the notation $H = h_1, h_2, \cdots, h_M$ denotes the hash array, an array of pointers pointing at hash buckets; $G = g_1, g_2, \cdots, g_M$ is the hash node buffer composed of an array of hash nodes, in which hash buckets are stored.

An entry of H, h_i, points to the i-th hash bucket, i.e. the head hash node in the linked hash node list of the i-th hash bucket. h_i is the index to the head hash node in the hash node array G. An entry of G, g_i, is a hash node. It contains a used page record $g_i.data$ and a pointer $g_i.next$ pointing to g_i's next hash node in a chained/linked hash node list. $g_i.next$ is also an index of an hash node in the hash node buffer G.

M is equal to the total number of pages in one event buffer. Because there exists an identical imaging between the event buffer and its hash node buffer, the length of the hash node buffer G are also M. Besides, according to the design of the hash table, when the event buffer is full, the hash node buffer is also full and each hash bucket contains on average one hash node. Therefore, the length of H is M as well.

4.3.3.2 Main Idea of the Buffer Allocation Algorithm Based on a Chained Free Hash-Node List

This subsection introduces a chained free hash-node list to solve the allocation problem both for the hash node buffer and the event buffer. The chained list of free hash nodes is created inside the hash node array G. *Free hash nodes* are the hash nodes inside the array G that are not inserted to any hash buckets in the hash table. All the free hash nodes inside G are linked together and compose a chained free hash-node list. The main idea of the buffer allocation algorithm is to maintain a chained list of all the free hash nodes. Each time when a new used page record is reported from the FPGA and a free hash node is required to store the used page record, a node is removed from the chained free hash-node list and inserted into the corresponding hash bucket; and each time when a used page record is required to be deleted, the

corresponding hash node is removed from the related hash bucket and inserted back into the chained free hash-node list.

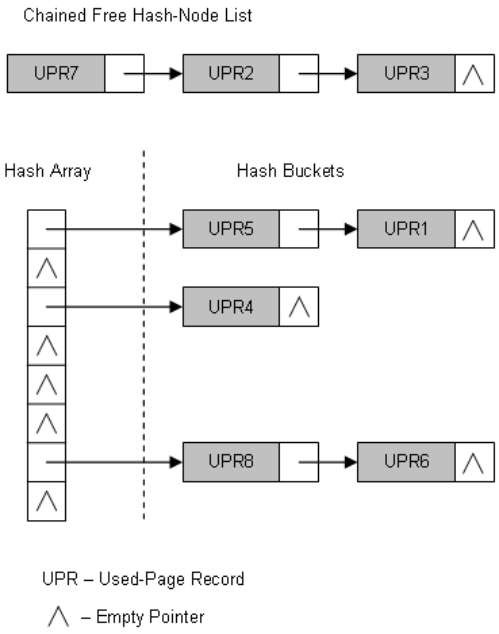

Figure 4.3: **Logical structure of an example hash table and an according chained free hash-node list.**

Figure 4.3 shows the logical structure of an example hash table and an according chained free hash-node list. In the example the maximum number of the hash nodes is assumed to be eight, and accordingly the length of the hash array and the length of the hash node array are also eight. Figure 4.3 illustrates the corresponding physical storage of the example hash table and the built-in chained free hash-node list is indicated.

With the introduction of a chained free-node list the hash table is denoted by a ternary form $< H, G, f >$, where f points to the chained free hash-node List. It is the index to the head hash node in the chained free hash-node list.

4.3.3.3 Extension to Event Buffer Management

The chained list of free hash nodes proposed above can not only deal with the dynamic buffer allocation and release for the hash table which is applied to the fast event lookup, but also handle the event buffer management inside ROBIN.

As mentioned previously the event buffer can be arranged in a same way that

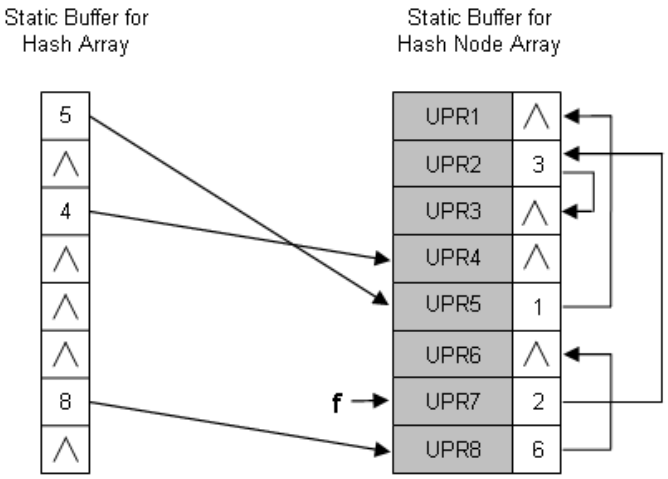

Figure 4.4: **Physical storage of the example hash table shown in figure 4.3. Note, f is the header of the built-in chained free hash-node list.**

the hash node buffer is organized, and vice versa. Therefore, when the PowerPC is required to update the free page ID FIFO for the FPGA, it may forward the IDs of the free hash nodes in the chained free hash-node list as free page IDs. When a used page record is returned from the FPGA, which contains an event ID and a page ID, the PowerPC may take the page ID as an index to allocate a hash node in the hash node buffer, fill the hash node with the used page record and insert it then into the relevant hash bucket.

Algorithms 4.3.3.3-4.3.3.3 give the detailed operations for the ROBIN event buffer management and event lookup based on the above proposed algorithm of a chained free hash-node list. In the operations only one event buffer is concerned and the notations introduced in section 4.3.3.1 are used. Besides, the hash table is denoted in the ternary form of $< H, G, f >$ introduced in section 4.3.3.2, where H is the hash array, G is the hash node array and f points to the header of the chained free hash-node List.

4.3.4 Discussion

This section introduces a chained free hash-node list for the allocation of an event buffer and its hash node buffer. The space complexity and the time complexity of the method are analyzed in the following.

Table 4.3 lists the memory requirements for the main data structures in the

Algorithm 4.1 Initialization

Problem Statement

When the ROBIN system starts to work, or when a request from the level 2 trigger commands to clear all the event data, every buffer in the ROBIN is cleared up and all hash nodes in the hash node buffer are set free. The free hash nodes are linked together and build up the initial chained free hash-node list, and on the other side all hash buckets are empty.

Algorithm

1. Initialize the hash array H. For all $i \in \{1, 2, \cdots, M\}$ set h_i =NULL. Here all hash buckets are set empty.
2. Initialize the hash node buffer G. Link all hash nodes in the hash node buffer into one linked list. For all $i \in \{1, 2, \cdots, M-1\}$ set $g_i.next = i + 1$ and $g_M.next$ =NULL.
3. Initialize the chained free hash-node list f. Let f point to the first hash node of the above linked list g_1, i.e. $f = 1$.

ROBIN PowerPC application for the management of one event buffer, when the size of each page inside the event buffer (64MByte SDRAM) varies from 1K bytes, 2K bytes to 4K bytes.

Page Size	1K Bytes	2K Bytes	4K Bytes
Hash Array (Pointers to Hash Buckets)	128K Bytes	64K Bytes	32K Bytes
Hash Node Array	1152K Bytes	576K Bytes	288K Bytes
Chained Free Hash-Node List	2 Bytes	2 Bytes	2 Bytes
Total	1280K Bytes	640K Bytes	320K Bytes

Table 4.3: Memory space requirements for the main data structures in the ROBIN PowerPC application for one event buffer (64MByte SDRAM).

An identical mapping between the page arrangement in the event buffer and the hash node arrangement in the hash node buffer is maintained, which simplifies the storage management inside the PowerPC application significantly. Table 4.4 shows the time complexity of the primary operations in the ROBIN PowerPC application both in the average case and in the worst case. As stated in section 4.2, both practically and theoretically the worst case for the hash lookup are far from happen.

Algorithm 4.2 Free Page FIFO Update

Problem Statement

The PowerPC provides the FPGA free page IDs through a free page FIFO. The free page FIFO is implemented as a static cycled queue and denoted in a ternary form by P, s, t, where $P = p_1, p_2, \cdots, p_N$ is an array and $N > 0$ is the length of the free page FIFO. Each entry in P is a page ID. s and t are two indices of the array P. They point to the start element and the end element of the static cycled queue with valid free page IDs. At initialization s =NULL and t =NULL. To update the free page FIFO the PowerPC application takes the IDs of the first hash nodes in the chained free hash node list as free page IDs, inserts them into the free page FIFO and removes these hash nodes from the chained list, till the FIFO is full or the chained list is empty.

Algorithm

1. Define $t' = (t + 1) \bmod N$.
2. If t' is equal to s, which means that the free page FIFO is full, no further operations are then required and exit. Otherwise, continue.
3. If f is NULL, which means that the chained free hash node list is empty, no more hash node or free page is available and exit. Otherwise, continue.
4. If s is NULL, which means that the free page FIFO is currently empty, then set $s = 1$ and $t = 1$. Otherwise, set $t = t'$.
5. Assign the last element p_t in the valid free page ID queue with the first node in the chained free hash node list: $p_t = f$.
6. Assign f with the next hash node in the chained list: $f = g_f.next$.
7. Go to step 1.

Algorithm 4.3 Insertion of a Used Page Record

Problem Statement

When a page in the event buffer is filled, a used page record is then created in the FPGA and forwarded to the PowerPC through a used page record FIFO. On receiving a used page record the PowerPC allocates a free hash node in the hash node buffer and fills it with the used page record. Then insert the hash node into the hash table, more exactly, the relevant hash bucket. Let r denote a new used page record to be inserted to the hash table, with $r.pid$ for its page ID and $r.eip$ for its event ID.

Algorithm

1. According to the identical imaging between the event buffer and the hash node buffer, the page ID $r.pid$ is also the ID of the hash node allocated in the hash node buffer to contain the used page record r. Define $p = r.pid$. and the hash node g_p is then the required free node.

2. Assign the data of g_p with the used page record r: $g_p.data = r$.

3. Put the event ID $r.eid$ into the hash function, which yields the hash key: $k = \text{Hash}(r.eid) = r.eid$ & 0xFFFF.

4. Insert the hash node g_p at the head of k-th hash bucket h_k:

$$g_p.next = h_k \quad \text{and} \quad h_k = p.$$

Algorithm 4.4 Event Lookup

Problem Statement

On event data request from the level 2 trigger the PowerPC software needs to find the relevant used page record(s) from the hash table. Then initiate a reply message involving the related page ID(s) and forward it to the FPGA. The FPGA is responsible to get the data from the event buffer and assembles the final reply message for the level 2 trigger. Let e denote the ID of the event to be deleted.

Algorithm

1. Put the event ID e into equation 4.1, which yields the hash key:

$$k = \text{Hash}(e) = e \ \& \ \text{0xFFFF}$$

   . The related hash node(s) are in the k-th hash bucket.

2. Define $p = h_k$, i.e. the head node in the k-th hash bucket.

3. If p is null, which means that p is the end of the hash bucket, stop searching and exit then.

4. If $g_p.data.eid$ is equal to e, which means that g_p is the required hash node and its data $g_p.data$ is one wanted used page record for the event e.

5. Let $p = g_p.next$.

6. Go to step **??**, to check whether any other hash node in the bucket is also wanted.

Algorithm 4.5 Hash Node Deletion

Problem Statement

On event deletion request from the level 2 trigger the operations in algorithm 4.3.3.3 for event lookup in the hash table are also performed in the PowerPC software, and the related hash node(s) with the used page record(s) are then removed from the hash table. No further operations by the FPGA or direct upon the event buffer are necessary. Here shows only the operations required for the deletion of a hash node from the hash table. To maintain the chained free hash-node list, the removed hash node needs to be inserted back into the chained free hash-node list. Let g_i denote a hash node that is to be removed from the hash table. Assume that the hash node g_i is currently in the k-th hash bucket h_k, and that g_j is the preceding node linked to g_i in the bucket (i.e. $g_j.next = i$) if g_i is not the first node in the bucket (i.e. $h_k != i$).

Algorithm

1. If h_k is equal to i, which means g_i is the first node in the k-th hash bucket, then go to step 3.

2. Since g_j is assumed to be the currently preceding node linked to g_i in the k-th hash bucket, let g_j then point at the next node after e_i: $g_j.next = g_i.next$. Go to step 4.

3. Let the head of the k-th hash bucket point at the next node after g_i: $h_k = g_i.next$.

4. Insert g_i at the head of the chained free hash node list:

$$g_i.next = f \quad \text{and} \quad f = g_i.$$

Operations	Average Case	Worst Case
Initialization	$O(M)$	$O(M)$
Free Page Allocation	$O(1)$	$O(1)$
Free Page FIFO Filling	$O(N)$	$O(N)$
Hash Node Insertion	$O(1)$	$O(1)$
Hash Node Deletion	$O(1)$	$O(1)$
Insertion of Used Page Record	$O(1)$	$O(1)$
Event deletion	$O(1)$	$O(M)$
Event Lookup	$O(1)$	$O(M)$

Table 4.4: Time complexity of the primary operations in the ROBIN PowerPC application both in the average case and in the worst case. Note that M is the total number of buffer pages; and N is the number of page IDs to be filled into the free page FIFO. N should be smaller than the size of the free page FIFO inside the FPGA.

Compared with the standard buffer allocation algorithm using a free-page ID stack, the second proposed algorithm based on a chained free hash node list provides a comparable computational efficiency but with no requirement for extra memory space.

4.4 Summary

Due to limited resources in the embedded ROBIN system, the ROBIN event buffer management strategy must exert all efforts to be economic both in the computational cost and in the memory space cost. Besides, for the sake of system security and stability it must also take into consideration to avoid dynamic memory allocation. This chapter presents a complete strategy for the event buffer management in the final ROBIN system. The problem of ROBIN's buffer management centers essentially around three tasks: event buffer arrangement and assignment, fast event lookup and the management of related dynamic data structures.

In this chapter firstly a page-based strategy for ROBIN event buffer organization is briefly reviewed. The 64MB SDRAM event buffer is segmented into fixed-sized pages. To deal with the mappings between the event IDs and the page IDs, a hash table is then introduced. An appropriate hash function is designed for the hash table, which guarantees a balanced distribution of hash nodes over the hash buckets. Moreover, a hash table is basically a dynamic data structure, since various number of hash nodes are dynamically inserted and deleted into the hash table over

time. Therefore, a buffer of fixed size is defined specific for the allocation of hash nodes. The buffer is named as hash node buffer.

As there exists an one-to-one mapping between the used pages in an event buffer and the occupied hash nodes in the related hash node buffer, a same mechanism can be adopted to arrange both the allocation of the event buffer and the allocation of the hash node buffer. In such a way the computational effort of the ROBIN PowerPC software is skillfully reduced. To deal with the buffer allocation both for the event buffer and the hash node buffer, a chained free hash-node list is introduced. The chained free hash-node list is built within the hash node buffer and hence no extra memory space is required.

The proposed buffer allocation strategy based on a chained free-node list can be easily extended to handle the buffer management for other embedded systems. The solution contributes even to solve a generic memory management problem, if the memory has to be divided into partitions with fixed size and each partition is a minimum unit for memory allocation and release. In such a case, the proposed buffer management method is an optimal solution both in respect of space complexity and in respect of time complexity.

5 ROBIN PowerPC System Analysis

As stated in Chapter 3, the hardware of ROBIN consists of two main parts centered on two processors, a Xilinx Virtex II 2000 FPGA and a PowerPC 440 micro-controller. The two processors cooperate to realize the functionalities of ROBIN. The FPGA plays the centric role as a high-rate and high-bandwidth data-flow core, and the PowerPC provides the management and control functionalities. This chapter and the following two chapters concentrate on the work in the ROBIN PowerPC system. This chapter discusses mainly the system requirements and principal tasks of the ROBIN PowerPC application.

The chapter is organized as follows. Firstly section 5.1 introduces briefly the hardware setup of PowerPC 440GP micro-controller. Section 5.2 discusses the communication interface (i.e. eight FIFOs) between the PowerPC application and the FPGA inside ROBIN. Through analyzing the data rate of eight FIFOs section 5.3 addresses the real-time requirements upon the ROBIN PowerPC application. Regarding the eight FIFOs section 5.4 discusses further four main cyclic tasks in the PowerPC application. Finally two possible architectures are presented in section 5.5 for the design of the PowerPC system, depending on whether an embedded real-time operating system is integrated.

5.1 PowerPC 440GP Microcontroller

The IBM PowerPC 440GP micro-controller offers exceptional performance, high design flexibility, and robust features geared to given networking and storage. The PowerPC 440GP is designed specifically for high-performance embedded applications. Figure 5.1 shows the block diagram of the PowerPC 440GP micro-controller.

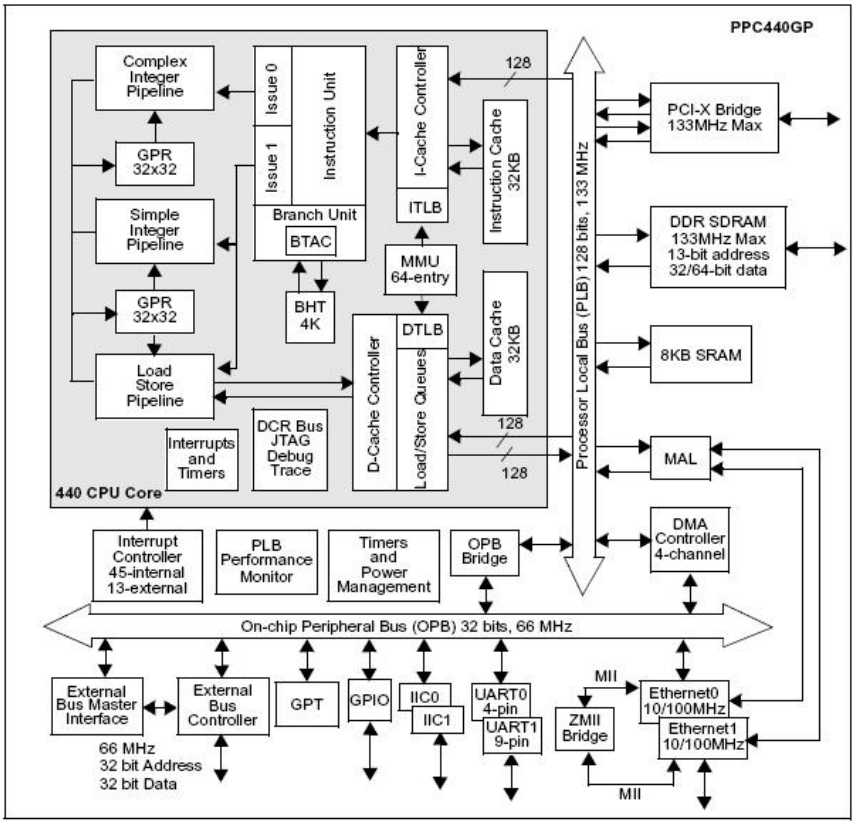

Figure 5.1: **PowerPC 440GP block diagram.**[38]

The IBM PowerPC 440GP micro-controller has an IBM PowerPC 440 core. The IBM PowerPC 440 core is a 32-bit RISC (Reduced Instruction Set Computer) core. It is implemented with IBM's advanced 90-nm copper CMOS technology, and provides up to 667 MHz and 1334 DMIPS (worst-case) performance. The 440 core is designed for high performance and high scalability. The PowerPC 440 core incorporates a super-scalar seven-stage pipeline and executes up to two instructions per cycle, which enhances dramatically the overall system throughput. Separate instruction and data caches, a JTAG port, trace FIFO, multiple timers and a memory management unit (MMU) are also supported by the core. The core's large data cache (32K) and instruction cache (32K) are 64-way set associative, with versatile configurations to enhance performance tuning. The integrated memory management unit (MMU), with 2.0 DMIPS/MHz performance, allows software developers to configure cache regions in three different modes to optimize their applications. For instance, locked regions can be used for low-latency code or interrupt service routines; transient regions handle use-once data without disturbing the whole cache; and normal regions is managed using typical least-recently-used (LRU) algorithms. Moreover, the 440 core employs the scalable and flexible Book E enhanced Power Architecture, which is optimized for embedded applications. The core can be integrated with various peripherals and application-specific macro cores using the CoreConnect^{TM} bus architecture to develop custom System-on-a-Chip (SoC) solutions. Peripheral options include memory controllers, DMA controllers, PCI interface bridges and interrupt controllers.

The PowerPC 440GP incorporates a width range of features, including on-chip Double Data Rate (DDR) SDRAM controller, PCI-X interface, External Bus Controller (EBC) with 8/16/32-bit external data bus width, DMA controller, on-chip Ethernet, 8K on-chip SRAM, debug support and other on-chip peripherals such as two serial ports, two I2C controllers, up to 32 GPIO, up to 13 external interrupts, and general purpose timers. The versatile features complement the RISC PowrPC 440 core, to provide powerful solutions to diverse embedded applications.

5.2 Communication with FPGA

To discuss the requirements upon the PowerPC system, we first need to know the PowerPC's system interface to the outside. The interface indicates the input or output data pipes of the PowerPC system. Through analyzing the requirements of each data pipe (e.g. its data rate or its data bandwidth), the requirements upon the PowerPC system are accordingly explained.

As mentioned before, the FPGA plays a centric role in ROBIN. It acts as a high-rate and high-bandwidth data-flow core. The communication of the PowerPC system with outside are actually all through the FPGA, or more exactly through a number of FIFOs inside the FPGA.

There are totally eight FIFOs used for the information exchange between the PowerPC system and the FPGA. Three free-page FIFOs and three used-page FIFOs are defined for the storage management of event data; and two message descriptor FIFOs are defined for the message exchange between ROBIN and the ROS host PC. The three free-page FIFOs and one response message descriptor FIFO are written by the PowerPC system and read by the FPGA; the three used-page FIFOs and one request message descriptor FIFO are written by the FPGA and read by the PowerPC.

The following of this section discusses how the eight FIFOs work in the ROBIN system and how the PowerPC and the FPGA communicate with each other through the FIFOs.

5.2.1 Free-Page FIFOs and Used-Page FIFOs

The three free-page FIFOs and the three used-page FIFOs are used to deal with the buffering of the incoming event data from three level-1 readout drivers (RODs), respectively. Figure 5.2 shows the activities inside FPGA in handling a fragment of incoming event data from one ROL. The roles of the two kinds of FIFOs are also indicated in the figure.

As described in the previous chapter a page-based strategy is used for the event buffer management in ROBIN. According to the strategy one event buffer is divided into pages of a fixed pre-defined size and the pages are the minimum unit for the storage of event data. The PowerPC is responsible to tell the FPGA in which pages to store the data of an incoming event through a free-page FIFO. Periodically the PowerPC writes the free-page IDs into the free-page FIFO. One free-page FIFO is dedicated to one event buffer.

When an event data fragment arrives at a readout link (ROL) connected to at the FPGA, it is firstly stored in a data FIFO of 256 words. An input handler inside the FPGA reads the data from the FIFO and extracts the event ID and the event status information (e.g. the last fragment flag, transmission flag, link errors, consistence errors, etc). At the same time, the input handler reads a free-page ID from the according free-page FIFO and removes it from the FIFO and then writes the event data fragment to the according free page in the event buffer. The address of the free page in the event buffer is meanwhile computed by the buffer arbiter through a

bit-shift operation on the free-page ID.

After the storage a used-page record is generated by the input handler. A used page record has 16 bytes and is composed of the event ID, the occupied page ID, as well as the status information, debug information, and the exact number of bytes filled in the page. The input handler inserts the used-page record into the used-page FIFO. The PowerPC is notified about the arrival of new events, and reads the new used-page records from the used-page record FIFO. Hence, the PowerPC application is informed of the occupation of another page in the event buffer. As described in the previous chapter a hash table is applied in the PowerPC application to manage the used-page records for later fast event retrieval.

The size of the free page ID FIFO is 1K words and each page ID takes 2 bytes. Hence, the free page ID FIFO contains at most 1k free-page IDs. The size of used-page record FIFO in the DMA is 128 bytes. Each used-page record has 16 bytes. Hence, the used-page record FIFO contains at most 8 used-page records.

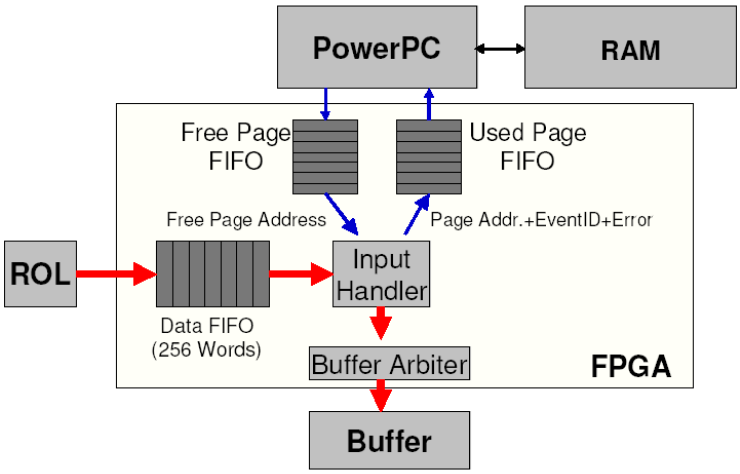

Figure 5.2: **Handling of incoming event data from one ROL. The thick red arrows indicate the flow of event data, and the thin blue arrows indicate the flow of control data [45].**

5.2.2 Message Descriptor FIFOs

One request message descriptor FIFO and one response message descriptor FIFO are applied for the message exchange between the PowerPC application and the

ROS host PC. Figure 5.3 shows how the ROBIN system handles request messages from the ROS host PC and assembles response messages to the ROS PC by using the two message descriptor FIFOs.

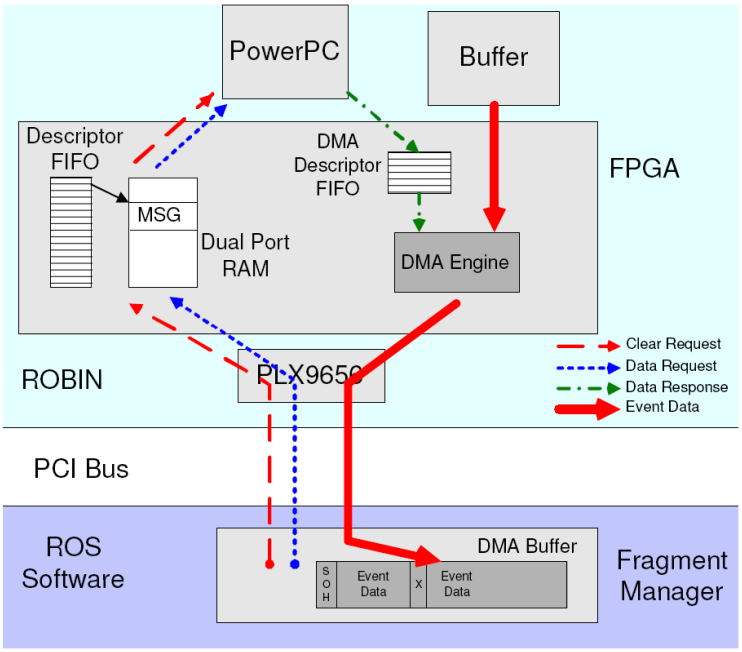

Figure 5.3: **Handling of request and response messages exchanged between the ROS host PC and the PowerPC application [45].**

When a request message from the ROS host PC arrives at the FPGA, it is firstly written into a 2048-word dual-port RAM. The address within this dual-port RAM is then written to a 32-word FIFO, which is the request message descriptor FIFO. With the address in the descriptor FIFO the PowerPC application can read the message from the dual-port RAM, and then decode and execute it. The address in the dual-port RAM is of two bytes, i.e. one word. Hence the request message descriptor FIFO may contain up to 32 request message addresses.

Response messages from the ROBIN system to the ROS PC are initiated by the PowerPC application. The response messages may contain event data or other information (status or debug information). Event data are stored in the event buffer. But the other information can be found in the related use-page records, which are provided directly by the PowerPC application without accessing the event buffer. In either case the PowerPC writes a 3-word command to the response message descriptor FIFO. The 3-word command is executed by a DMA engine inside the FPGA, which assembles and transmits the final response messages.

5.3 Real-Time Performance Requirements

This section addresses mainly the real-time performance requirements of the ROBIN PowerPC system. As described above there are eight FIFOs used for the communication between the PowerPC application and the FPGA. The performance of the PowerPC application must be high enough to support the required input or output data rate of each FIFO.

5.3.1 Event Data Rate from One Readout Link

In the ATLAS TDAQ after event selection by the level 1 trigger the data volume drops from 45 TBytes to 136 GBytes per second, and the data rate is reduced from 40 MHz to 100 kHz. In the level 2 trigger there are altogether 1600 readout drivers, i.e. 1600 readout links. Let v_d denote the data volume arriving at the level 2 trigger, r_d denote the data rate and n_{ROL} denote the number of ROLs. Then $v_d = 136$ GBytes/s, $r_d = 100$ kHz and $n_{ROL} = 1600$. It can be calculated, every cycle $v_d/r_d = 1360$ bytes arrives at the level 2 trigger and every cycle on average $v_d/r_d/n_{ROL} = 850$ bytes at each ROL.

Data rate is actually event rate. In each cycle one event arrives at one ROL. One event occupies at least one page and at most, let's say, two pages in an event buffer. Hence with an event rate of 100 kHz every 10 μs at most two free pages are filled. Accordingly every 10 μs up to two free page IDs are removed from the free-page ID FIFO and up to two used-page records will be filled into the used-page record FIFO. As mentioned in the previous subsection, the free-page ID FIFO contains 1k free-page IDs and the used page record FIFO contains 8 used-page records. Then it can be calculated, every 5 milliseconds the free-page ID FIFO must be updated at least once by the PowerPC application and every 40μs the used-page record FIFO must be read and cleared once by the PowerPC application. Note that the maximum cycle time of 5 ms for free-page ID update is based on the assumption that every time the free-page FIFO is fully filled after updating. But this is mostly not the case in the running. Hence the cycle time for free-page ID update must be less than 5 ms.

5.3.2 Request Message Rate from the ROS PC and the Event Builder

On the other hand, the real-time performance with respect to the two message descriptor FIFOs is much less required. Firstly, the ROS host PC has a local re-

quest message queue. If the FPGA's request message FIFO is full, the ROS PC will wait till there is space inside the FPGA's FIFO. Secondly, regarding the response message FIFO there is generally no direct performance requirement upon the PowerPC application, but upon the polling mechanism of the ROS host PC application. The ROS PC must read and clear up the FIFO in time. However, there is always an one-to-one correspondence between a request message and a response message. Therefore, regarding the both message descriptor FIFOs a same real-time performance requirement is preferably applied.

However, an average data rate at the request message descriptor FIFO must be supported by the PowerPC application. The most frequent request messages are the event data request messages. Up to 7% of all data coming on one ROL are requested by the level-2 PC farm on average [12]. Up to 3% will be accepted by the level-2 trigger and requested by the event builder [12]. With a data rate of 100 kHz at each ROL, the event data request rate for a ROBIN board with totally three ROLs is then 100*3*(7%+3%) kHz, i.e. 30 kHz. That means, on average every 33 μs one event data request message arrives at the ROBIN. Moreover, the descriptor FIFO contains the addresses of 32 request messages at most. Therefore, every 32*33 μs (i.e. 1.07 ms) the request message descriptor FIFO is completely filled. Taking other request messages such as event deletion message into account, every one millisecond an empty request message descriptor FIFO is fully filled and must be read and cleared once by the PowerPC application.

Table 5.1 lists the required processing rate from the PowerPC application with respect to different communication FIFOs between the PowerPC and the FPGA. According to the list the used-page record FIFO requires by far higher processing rate compared with other FIFOs.

5.4 Cyclic Tasks in the PowerPC Application

Section 5.2 discusses four types of FIFOs that are defined for the communication between the PowerPC application and the FPGA inside the ROBIN system. Regarding the four types of FIFOs four principal cyclic tasks of the PowerPC application are defined: 1) free-page update, 2) used-page record handling, 3) request message decoding and execution and 4) response message initiation.

5.4.1 Free Page Update

The direct access to event buffers is done by the FPGA, but the organization and management of the buffers is handled by the PowerPC application. Free page update

FIFOs	Tasks of the PowerPC application	Minimum processing rate	Maximum Cycle time
Free-page ID FIFO	Free-page update	> 200 Hz	< 5 ms
Used-page record FIFO	Used-page record handling	25 kHz	40 μs
Request message descriptor FIFO	Request message decoding and execution	> 1 kHz	< 1 ms
Response message descriptor FIFO	Response message initiation	> 1 kHz	< 1 ms

Table 5.1: Real-time requirement upon the PowerPC application with respect to different communication FIFOs (between the PowerPC and the FPGA) and the according tasks of the PowerPC application.

is for the PowerPC application to fill the three free page FIFOs in the FPGA with the IDs of the unoccupied pages in the respective event buffers. With the free-page IDs the FPGA can then store newly incoming event data from RODs into the according pages in the according event buffers. Details about the strategy of free page management is found in chapter 4.

As mentioned in the previous section, according to the rate of the event data arriving at the FPGA and the size of the FIFOs the minimum update rate of the free-page ID FIFOs must be over 200 Hz. In other words the task of free page update must be performed at least once every five microseconds by the PowerPC application.

5.4.2 Used-Page Record Handling

For an incoming event fragment from ROD to ROBIN, the FPGA stores the event data into a free page in the event buffer, creates a used-page record for the page and appends the used-page record into the according used-page record FIFO. Used-page record handling is for the PowerPC application to read out the used-page records in the used-page record FIFOs, and insert the records into the according hash tables in the PowerPC application. The hash tables are applied to manage the storage of the event buffers and to provide efficient strategy for later event lookup. Details about the design of the hash tables have been given in chapter 4.

According to the incoming event data rate from RODs and the size of the used-page record FIFOs, the FIFOs must be cleared by the PowerPC application every 40 μs. Detailed explanations are found in the previous section. That means, the

task of used-page record handling must be performed at least once by the PowerPC application every 40 μs.

5.4.3 Request Message Decoding and Execution.

Request messages from the ROS PC or the event builder are forwarded to the PowerPC application through the request message descriptor FIFO in the FPGA. Then they are decoded and executed in the PowerPC application.

Types of the request messages from the level 2 trigger are described in [32], including a series of messages for accessing the system configuration parameters. Three most complicated-to-handle request messages are related to the management of the event buffers. They are "get fragment", "clear fragment", and "clear all fragment". The message of "get fragment" is a request for event data regarding a given event ID. Each event in the ATLAS has a unique event identifier, which is assigned by the level 1 trigger, and each request from the level 2 trigger must be attached with the according event ID(s). "Clear fragment" is to delete event data, given one or multiple event IDs. "Clear all fragments" commands ROBIN to remove all the data inside the event buffers. This request is used to re-initialize event buffers. In order to reply the first two types of messages in real time, the PowerPC application is required to retrieve immediately the event storage address inside event buffers for a given event ID. The event lookup strategy for ROBIN is presented in chapter 4.

According to the real-time requirement analysis regarding the message descriptor FIFOs in the previous section, the task of request message decoding and execution must be called at least once every one millisecond.

5.4.4 Response Message Initiation.

Besides request message handling, the response messages backwards for the ROS PC and the event builder must also be initiated by the PowerPC application. For every response message the PowerPC application will write a 3-word command into the response message descriptor FIFO in the FPGA. Except replies to "get fragment" requests, which require event data in the event buffers, information for other response messages can be provided by the PowerPC application directly. As with the above task of request message decoding and execution, the task of response message initiation must also be performed at least once every one millisecond.

The correspondences between the above tasks and the different communication FIFOs are listed in table 5.1. The respective real-time requirements are also given

in the table.

5.5 System Architectures

In this work two system architectures are investigated for the implementation of the ROBIN PowerPC system. One is a standalone software implemented as a single loop of subroutines (or tasks) running directly upon the PowerPC hardware layer. No operating system is needed in this first implementation. The other is based on an embedded real-time operating system (RTOS). In this implementation the multiple tasks of the PowerPC application are managed by the real-time scheduler from the operating system .

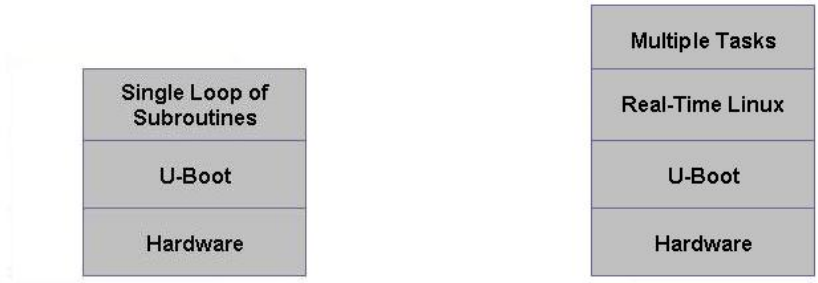

Figure 5.4: **Two architectures for the ROBIN PowerPC system.**

Figure 5.4 shows the two system architectures. In both designs the U-Boot is chosen as the bootstrap loader for the system. The bootstrap loads the image of the final target software into the system memory and executes it on the machine. Details about the U-Boot and its extensions for the PowerPC system are found in the appendix B.

5.6 Summary

This chapter addresses mainly the ROBIN PowerPC system requirements. Specially the real-time performance requirement upon the PowerPC application is analyzed according to its communication interface with the FPGA, i.e. eight FIFOs in the FPGA. Regarding the eight FIFOs four major cyclic tasks in the PowerPC application are defined and addressed.

Moreover, two system architectures are proposed in this chapter for the implementation of the PowerPC application inside the ROBIN. The difference between

the two architectures lies on whether an embedded real-time operating system is integrated. In the following two chapters the software design and software optimization based on the two architectures are discussed, respectively. The real-time performance requirement addressed in this chapter is applied to check the feasibility of the two system architectures and optimization measures are proposed against the performance requirement.

6 Standalone PowerPC Application

In the previous chapter two architectures are presented for the implementation of the PowerPC system inside ROBIN. This chapter concentrates on the design and optimization of the PowerPC software based on the first non-OS architecture, i.e. the standalone PowerPC application without an operating system.

The organization of this chapter is as follows. Section 6.1 addresses the detailed software design of the standalone ROBIN PowerPC application. Section 6.2 proposes several measures to optimize the performance and reliability of the software. Finally, the proposed standalone PowerPC system is tested together with an entire ROS/ROBIN system in a simulated ATLAS testing environment and the experimental results are given in section 6.3.

6.1 Software Design

Since there is no operating system, the standalone PowerPC application is implemented as a single-thread program. The cyclic tasks discussed in the pervious chapter 5 are implemented in a single loop in the program.

This section presents the software design of the standalone PowerPC application in three diagrams: the component diagram, the use case diagram and the activity diagram. The component diagram explains the major construction of the software. Through expanding the component diagram the use case diagram is drawn. Finally, the activity diagram shows the work flow of the standalone PowerPC application.

6.1.1 Component Diagram

Chapter 4 introduces a page-based strategy for the event buffer management in the final ROBIN design and a hash table for the fast event lookup in the event buffers. Implementation of the two baseline functionalities constitutes two major components of the PowerPC application. A third major component of the PowerPC application is the message handling module.

Figure 6.1 shows the component diagram of the ROBIN PowerPC application. Three main modules in the application are: event buffer management module, event lookup module and message processing module. The major data flow through the modules and the FPGA is indicated in the figure. The module of event buffer management maintains a free page ID list for each event buffer to record the current occupation status of the related buffer. The event lookup module manages a hash table for each event buffer for the fast lookup of event data.

Section 4.3 proposed a data structure of a chained free hash-node list for the storage both of the hash table and of the free page ID list. In the component diagram as well as the use case diagram in the next subsection, the hash table and the free page ID list are denoted as two data resources for the easier understanding by the readers.

6.1.2 Use Case Diagram

Figure 6.2 shows the use case diagram of the ROBIN PowerPC application. The use case diagram is in a sense an expanding diagram of the component diagram.

Four actors are indicated in the diagram: the message controller, the message executor, the event buffer organizer and the used-page manager (hash-table manager). The first two actors are from the message processing module. The message controller talks directly to the FPGA through the message descriptor FIFOs inside the FPGA. The message handling is primarily done by the message executor. To execute the messages that are related to event buffers, the message executor needs to ask the used-page manager from the event lookup module as well as the event buffer organizer from the buffer management module to fulfill the task together. The used-page manager maintains a hash table for each event buffer for event lookup. The event buffer organizer manages a free page ID list for each event buffer to record the current occupation status of the buffer.

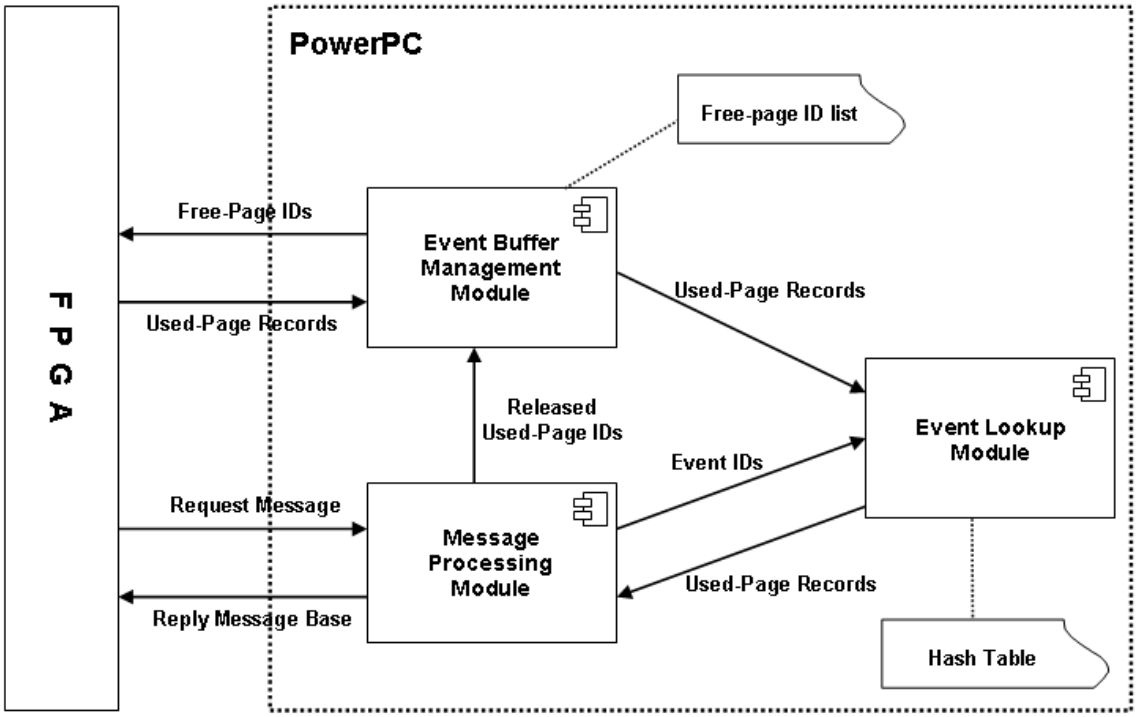

Figure 6.1: **Component diagram of the ROBIN PowerPC application.**

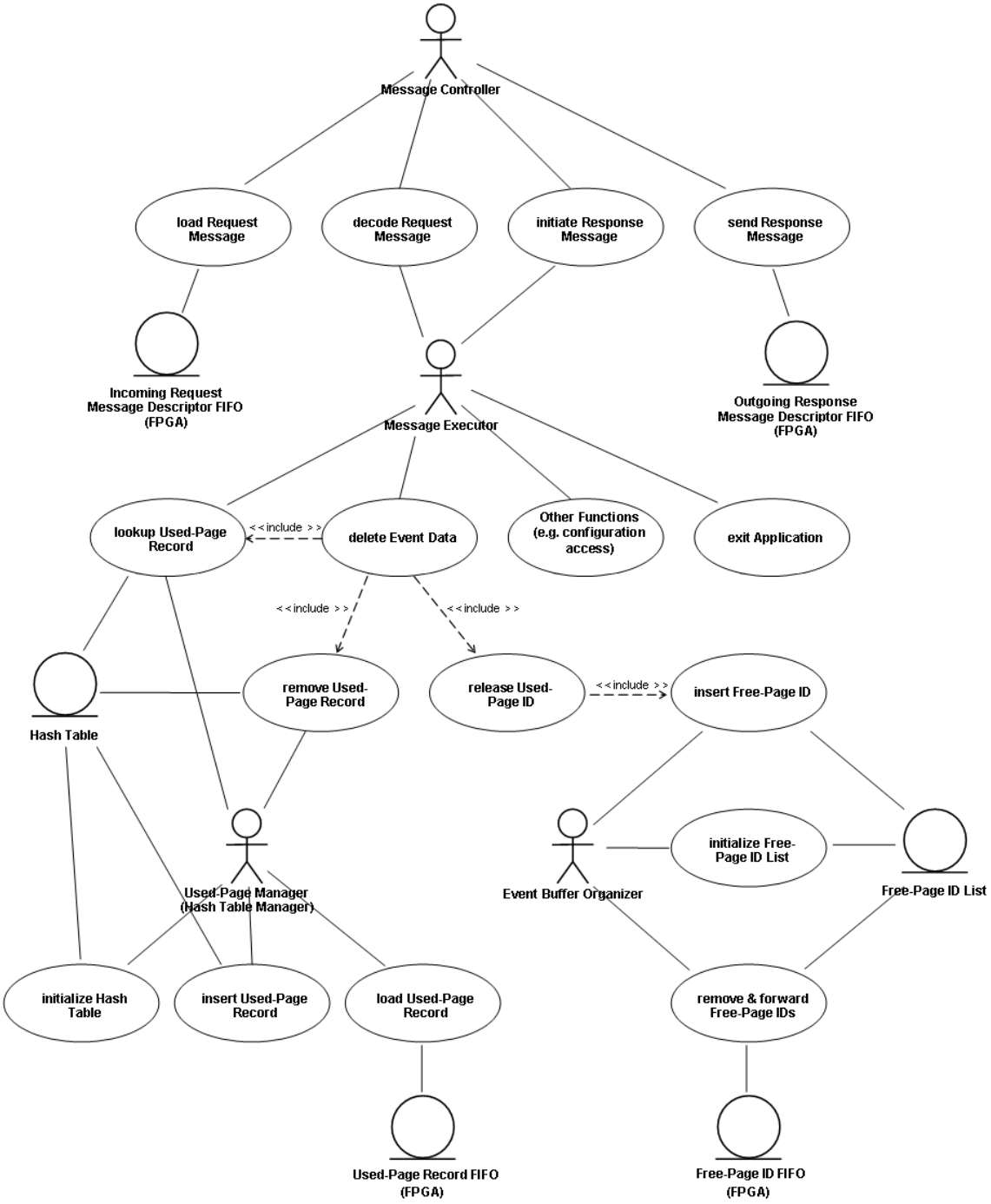

Figure 6.2: **Use case diagram of the ROBIN PowerPC application.**

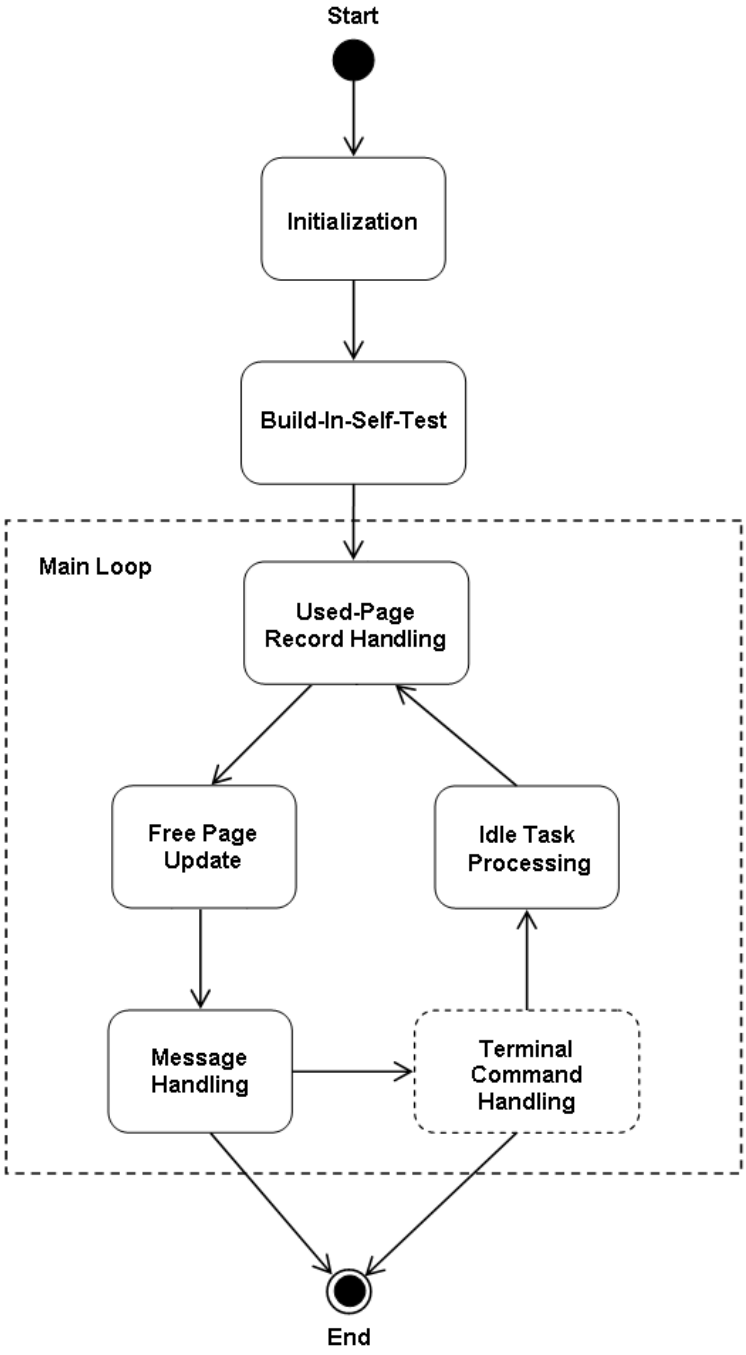

Figure 6.3: **Activity diagram of the standalone ROBIN PowerPC application based on the non-OS architecture.**

6.1.3 Activity Diagram

The previous chapter 5 addresses four major cyclic tasks of the PowerPC application regarding the four types of communication FIFOs between the PowerPC application and the FPGA. In the standalone ROBIN PowerPC application the four cyclic tasks are implemented in a single loop. Figure 6.3 shows the activity diagram of the standalone PowerPC application based on the non-OS architecture. In the diagram the task of request message decoding and execution and the task of response message initiation are indicated as one task, since the two tasks with a same cyclic time have an one-to-one identical mapping.

The application starts with an initialization procedure, including loading configuration parameters, initializing PowerPC registers and FPGA registers, as well as creating a common address map for the PowerPC's affiliated memory, the event buffers and the shared buffers with the FPGA. Global data structures in the PowerPC application, e.g. the hash tables and the chained free-node lists, are also initialized here. Then a Build-In-Self-Test (BIST) routine is performed, to check the status of the FPGA, the ROLs, the accessibility of the shared buffers and the temperature of the PowerPC itself.

If the system self-test is passed, the program goes into the main loop of the application. Five operations get involved in the main loop. The first three tasks are the cyclic tasks addressed in the previous chapter 5. The other two tasks are for idle-task processing and terminal command handling. Idle tasks include monitoring the device temperature measured by the temperature sensor and making some statistic analysis over the buffer occupancy. Terminal command handling is enabled only in the development phase for system testing and debugging. In the final target board it will be disabled.

The exit of the program results from a user terminal command or from a request message command from the ROS PC.

6.2 Performance Optimization

This section deals with the performance optimization of the standalone PowerPC application. Before the introduction of optimization measures the goal of the performance optimization is firstly addressed.

6.2.1 Goal of the Optimization

The activity diagram of the standalone PowerPC application in figure 6.3 shows five cyclic tasks involved in the main loop. Among the five tasks the idle task processing and the terminal command handling have by far lower priority compared with the other three tasks. These three tasks have been discussed in chapter 5 and their minimum processing rates in the PowerPC application are analyzed and listed in table 5.1. The PowerPC application must guarantee to reach the required processing rates of these tasks.

In the standalone PowerPC application although five tasks get involved in the main loop, it does not mean that all the five tasks must be executed in each loop cycle. The execution rates of the tasks can be adjusted according to their relative priority. In this context the priority of a task is defined according to its processing rate; task requiring higher processing rate has higher priority and vice versa. Task of a higher processing rate may be executed more often, e.g. once in every loop cycle, while a task of a relatively lower processing rate may be executed once in several cycles. In the case of the PowerPC application, the task of used-page record handling requires the highest processing rate, i.e. 25 kHz, and the task of idle task processing requires the processing rate of 250 Hz. According to the relative proportion of their processing rates, if the former task is executed once in every loop cycle, the latter is hence executed once in 100 cycles.

When the relative processing rates of the tasks are guaranteed, another performance criterion is then how fast the entire main loop can run. The cycle rate of the entire main loop must reach the required processing rate of the highest-priority task. Therefore, the standalone PowerPC application must also minimize the processing time of one main loop cycle.

Because the main loop runs continuously without a timer, the running time of each main loop cycle varies due to the varying amount of workload in the cycle, for example, varying numbers of used page records to be handled in the cycle or varying numbers of messages to be processed. The cycle rate of the main loop is a varying factor.

Concerning the ROBIN PowerPC application, the used-page record handling task is the most time-critical task. To ensure the safe and security of the ROBIN PowerPC system, the processing rate of used-page record handling task (25 kHz) must be guaranteed at any moment. It means that the running time of every main loop cycle, including the slowest cycle, must not exceed a limited time period, i.e. 40μs.

For the simplification of explanation, the running time of the slowest main loop cycle is defined as the *cycle time of the main loop*, and the corresponding cycle rate

is defined as the *cycle rate of the main loop*. So long as the defined cycle rate of the main loop is above the required processing rate of the most time-critical task, the safe and security is guaranteed.

At this point we can tell, the goal of the performance optimization for the standalone PowerPC application is to maximize the cycle rate of the main loop while the required relative processing rate of each task is guaranteed. In other words, it is to minimize the running time of each main loop cycle while the relative processing rate of each task is guaranteed.

6.2.2 Application-Specific Optimization

According to the previous statement the optimization for the standalone PowerPC application is to minimize the running time of each main loop cycle. To achieve this goal, it is actually required to distribute the workload in the main loop uniformly into each cycle. This is the principle of the optimization measures proposed in the following of this section.

Without a real-time preemptive scheduler in the non-OS based PowerPC system, a number of application-specific measures have to be taken to arrange the operations in the main loop of the PowerPC application in order to improve the overall processing rate of the entire system. It will be seen that, with the help of these measures, the running time of the main loop cycle in the worst case is even expectable and so is the overall processing rate of the PowerPC system.

6.2.2.1 Used-Page Records Handled in Every Main Loop Cycle

According to table 5.1, the task of used-page record handling needs to be executed the most frequently in the PowerPC application and has accordingly the highest priority. The only reasonable setup for the standalone PowerPC application is to execute the task once in every main loop cycle.

The varying cycle rate of the main loop represents then the varying cycle rate of this task. Therefore, at any moment the cycle rate of the main loop must not be lower than the required minimum processing rate of the used-page record handling task, i.e. 25 kHz. The running time of the slowest main loop cycle must not exceed 40μs.

6.2.2.2 At Most Two Messages Handled in One Cycle

The task with the second highest priority is message handling. According to table 5.1 the minimum cycle rate of the task is 1 kHz. This number is given under the

assumption that all messages in the request message descriptor FIFO are processed at one cycle. That is to process up to 32 messages within one cycle. In this case the running time of the cycles in the main loop with message handling could be much extended. This is, however, against the above optimization principle of uniform workload distribution.

In order to reduce the running time of every main loop cycle, it is preferable that at most one message would be handled in one main loop cycle. However, as stated in section 5.3 the average rate of request messages arriving at the ROBIN board is around 30 kHz, which is faster than the expected cycle rate of the main loop, 20 kHz. Therefore, it is suggested for the standalone PowerPC application to process at most two request messages in one main loop cycle. If more than two request messages are available, only the first two of them are handled in the current cycle and the other messages must be handled in a next or later cycle. In this way it guarantees, in one main loop cycle at most two messages are handled and at the same time the required processing rate of request messages is also sustained.

6.2.2.3 At Most Three Tasks Executed in One Cycle

According to the descending priority order, the tasks are arranged as 1) used-page record handling, 2) message handling, 3) free-page update, 4) idle-task processing 5) and terminal command handling.

As stated above, the used-page record handling task and the message handling task are executed in each cycle. The next question is then when to execute the other three tasks. To keep the running time of one main-loop cycle short, a straightforward measure is not to execute the other three tasks in a same cycle. That also means, at most three tasks are executed in one main loop cycle. They are task 1, task 2 and one of the other three lower-priority tasks. Since the real-time performance requirements for the three lower-priority tasks are not critical, this measure is easy to realize.

6.2.2.4 Event Deletion Messages not Handled Together with Other Lower-Priority Tasks in a Same Cycle

Most event deletion request messages are massive event deletion messages. Handling of these messages is also a costly operation in the PowerPC application. In the operation a number of events have to be firstly looked up in the hash table, in which the used-page records of these events are stored, and then removed from the hash table. Hence another measure to reduce the maximum running time of one main

loop cycle is to handle event deletion messages not together with any of the other three lower-priority tasks in a same cycle.

Although event deletion request messages are not so often received, an additional measure for the system security must be taken in case event deletion request messages continue arriving in every cycle and the three lower-priority tasks are blocked. Therefore, the minimum processing rates of the three low-priority tasks must be defined.

According to table 5.1, the free page ID FIFO in the FPGA needs to be updated every 5 ms in the extreme case when the FIFO is full after updating, i.e. 1k valid free page IDs in the FIFO. Surely this is not always the case in the running. Actually one condition needs to be satisfied for the free page ID FIFO update. That is, the number of valid free page IDs in the FIFO keeps being equal to or greater than eight, i.e. the size of the used-page record FIFO. It is easy for the PowerPC application keep recording the current number of free-page IDs in the free-page ID FIFO. When the number is less than 8, the task of free-page update is forced to be executed.

Minimum calling rates for the other two low-priority tasks can be fixedly configured in the application. In this work the two tasks are called at least once every 100 main loop cycles.

Note that the last three measures stated above are, however, not so critical for the PowerPC application based on the OS-based architecture, because the scheduler in a real-time operating system guarantees the preemption for higher-priority tasks even when a lower-priority task is in the running. On the contrary lower-priority tasks are not encouraged to run more often than enough, in order to reduce the overhead of task switching costs.

6.2.3 Worst Case after Optimization

Following the optimization measures proposed in the previous subsection, the maximum workload within one main loop cycle in the worst case is expectable. The worst case occurs when all the three used-page record FIFOs are full and two event-deletion request messages arrives. The maximum workload within one main loop cycle is then to handle 24 used-page records and two event-deletion request messages.

6.3 Experiments

In this section the proposed standalone PowerPC application is tested together with a ROS/ROBIN simulation system. In the following the experiment environment is

firstly introduced. Then the performance of the standalone PowerPC application is measured. In particular the processing rate of its main loop is tested. Finally, the performance of the entire ROS/ROBIN system, with the proposed standalone PowerPC application integrated, is also measured against the baseline requirement of the ATLAS DAQ chain.

6.3.1 Setup of Testing Environment

The PowerPC application is developed on a desktop with a 2.4GHz Pentium 4 CPU. The GNU toolchain is used to compile the code for the target PowerPC platform, i.e. an IBM 440GP micro-controller. In order to load the final executable binary code of the application to the PowerPC platform, an Abatron BDI2000 JTAG Debugger is firstly used to write the U-Boot code to the PowerPC's flash memory via an Ethernet Hub. When the PowerPC starts to run, the U-Boot loads the executable code of the PowerPC application from the desktop to the PowerPC's memory (i.e. the on-chip DDR SDRAM) via a serial connection, and then triggers the application to run. Details about the setup of this cross development environment are found in appendix B.

Since the ATLAS detector is still under construction, a simulation testing environment is built up to test the designed ROS/ROBIN system. Figure 6.4 illustrates the setup of the ROS/ROBIN testing environment.

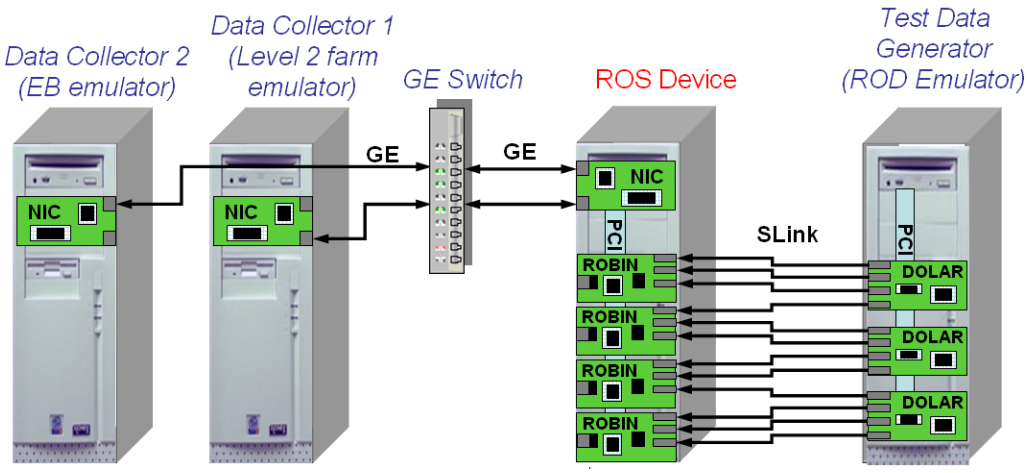

Figure 6.4: **Setup for the ROS/ROBIN testing environment**

It is described in chapter 3 that a commercial "off-the-shelf" high performance PC is chosen as the ATLAS ROS host PC. The PC has a 3.4GHz Pentium 4 CPU and four PCI buses. The PC is equipped with four ROBIN boards and two Gigabit Ethernet network adapters. Setup of the ROBIN boards are the same as the ROBIN prototype in the final design described in section 3.2.

As the event data input and output of the ROS system is concerned, three additional PCs are used to set up the testing environment. They are one test data generation PC and two data collection PCs.

At the event input side a PC with an 866MHz Pentium III CPU is the test data generator, also named as the ROD emulator. It emulates the readout drivers (ROD), which generates "event data" and forwards the data to the ROS system through the SLinks on its affiliated ROBIN boards. Three SLink source cards called DOLAR [39] are plugged into three PCI slots on the PC. Each DOLAR card contains four HOLA SLinks. The event data fragments generated by the PC are sent to the ROS PC via the 3×4 HOLA SLinks with configurable size and frequency.

Two data collection PCs are deployed at the event data output side of the ROS system. One of them has a 2.66GHz Xeon processor and is used to emulate the level-2 farm. The PC generates and forwards two types of request messages (data requests and event deletion requests) to the ROS PC and collects reply event-data messages that are sent back from the ROS PC. Note that each event deletion request message contains a list of up to 100 events. The other data collection PC has a 2.4GHz Pentium 4 CPU and emulates the event builder. It forwards only data requests to ROS and collects data messages from ROS. The two PCs are connected through a Gigabit Ethernet switch to the network adaptors of the ROS PC. The Gigabit Ethernet switch that connects the two data collection PCs with the ROS host PC has an Allied Telesyn AT-9410GB with 10 ports.

In the following of this section, the standalone PowerPC application proposed in this chapter is firstly tested. The performance of its single-loop-of-subroutines design is measured. Then experiments are conducted to measure the performance of the entire ROS/ROBIN system with the proposed PowerPC system integrated.

6.3.2 Performance of Standalone ROBIN PowerPC Application

As mentioned in section 6.2.1, the goal of the performance optimization for the standalone PowerPC application is to minimize the running time of every main loop cycle. The cycle rate corresponding to the running time of the slowest main loop cycle is defined as the main-loop cycle rate that the standalone PowerPC application

is capable to sustain. The experiment in this section is to measure the main loop cycle rate of the PowerPC application, while varying event data request rate. The event data request rate here refers to the event data rate that are requested by the level-2 PC farm and the event builder.

Each event request from the level-2 PC farm or the event builder are firstly sent to the ROS PC and then forwarded by the ROS PC to the ROBIN. For each requested event a corresponding data request message is sent from the ROS PC to the ROBIN and finally to the PowerPC application. All the event data, either requested or non-requested, are always deleted in the end through event deletion request messages. As mentioned previously, each event deletion request message contains a list of up to 100 events.

An additional prerequisite for this test is, that at each loop cycle the used-page record FIFO in the FPGA is always previously fully filled. That means, the test data generation PC fills the ROBIN board with as much event data as the ROBIN PowerPC application is capable to sustain, to guarantee a maximum data throughput during the measurement.

Figure 6.5 shows the curve of the main loop cycle rate of the PowerPC application while the event data request rate increases from zero to twenty percent. Higher event request rates are not tested in this experiment. Because at most two messages are handled in one main loop cycle, the according message handling rate will be lower than the request message incoming rate if a higher event request rate above twenty percent is provided.

When the event request rate increases, more event data request messages arrive at the ROBIN board and more messages need to be handled in the main loop of the PowerPC application. However, according to the curve the main loop processing rate does not drop significantly with the increased event request rate. Due to the optimization measures introduced in the previous section, the increased workload of the application is distributed uniformly into the main loop cycles. Under a certain limitation of workload, the most computational costly cycles are always those involving two event deletion requests. This is because the handling of an event deletion request is the most costly operation. Besides, in this test the number of events to be deleted in one event deletion request message is also fixed; therefore, the variation in the processing time of these messages is very limited.

The figure also shows, when the event request rate is 10 percent, the cycle rate of the main loop is around 29 kHz, i.e. a cycle time of about 34 μs. That means, the used-page record FIFO in the FPGA can be updated once every 34 μs. This is obviously above the real-time performance requirement (40 μs) as given in table 5.1.

Figure 6.6 shows the maximum level-1 event data incoming rate that the stan-

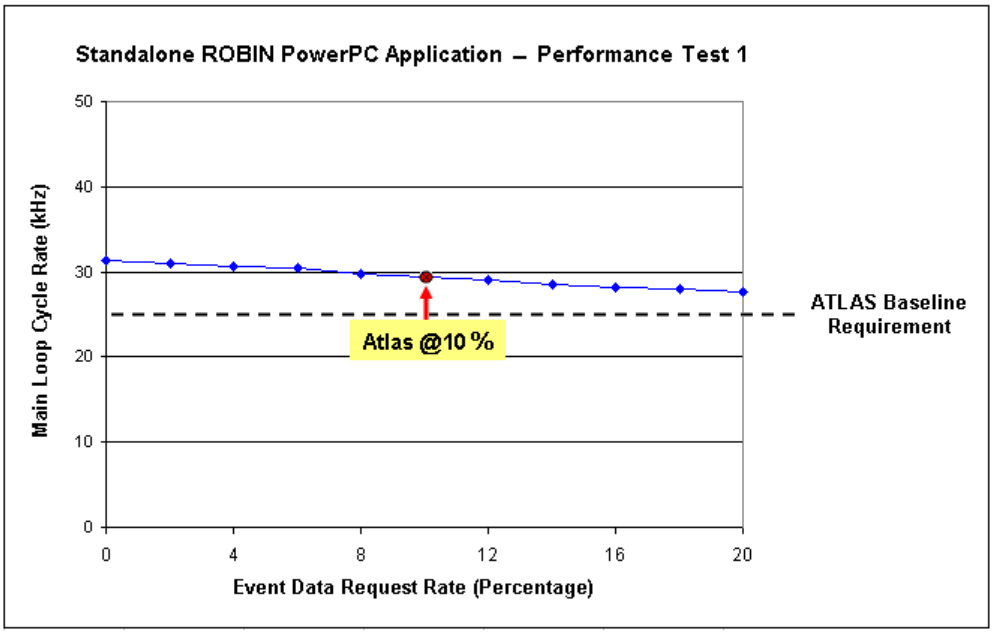

Figure 6.5: **Main loop cycle rate (supported by the standalone PowerPC system) vs. event data request rate**

dalone ROBIN PowerPC system can sustain while the event data request rate increases from zero to twenty percent. Two curves are shown in the figure. The blue curve is for the case that each event data fragment occupies one page in the event buffer; the yellow curve is for the case that each event data fragment occupies two pages in the event buffer. It is self-explanatory that the first curve is about one time higher than the second curve. According to the ATLAS baseline requirement the event request rate is about 10 percent and the level-1 event data incoming rate is around 100 kHz. In this test when the event request rate is 10 percent, the supported level data incoming rates indicated in both the curves are above 100 kHz, i.e. above the ATLAS requirement.

6.3.3 Performance of Integrated ROS/ROBIN System

The experiment in the previous subsection shows the satisfying performance of the standalone PowerPC application itself. The experiments in this section are to measure the performance of the entire ROS/ROBIN system, in which the proposed standalone PowerPC application is integrated.

For the ROS/ROBIN system the most concerned factor is the maximum level-1

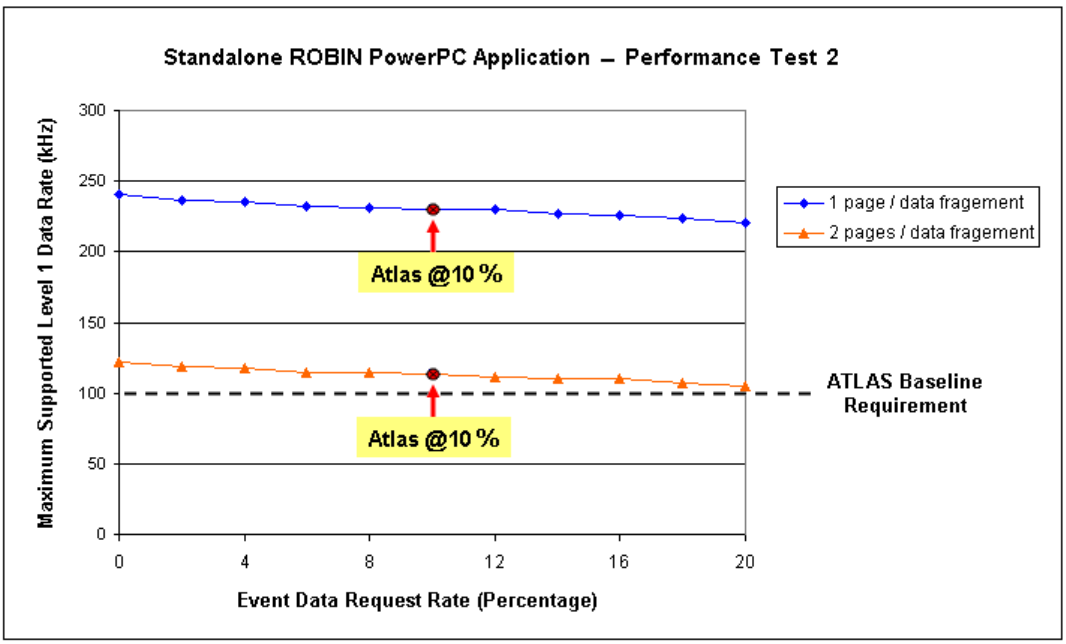

Figure 6.6: **Maximum level-1 data rate (supported by the standalone PowerPC system) vs. event data request rate.**

incoming data rate that the ROS/ROBIN system is capable to support. In the following this factor is measured, while varying different input or output factors of the ROS/ROBIN system, including the data acceptance rate of the level-2 farm, the data acceptance rate of the event builder and the size of event data fragments.

6.3.3.1 Maximum Supported Level-1 Data Rate versus Level-2 PC Farm Data Request Rate

The first test measures the maximum allowed level-1 data rate that the ROS/ROBIN system supports, while the data request rate from the level-2 PC farm varies. The ROD emulator simulates the level-1 side and inputs event data continuously through SLinks to the ROBIN boards. The level-2 farm emulator sends data request messages to the ROS system, requires event data from the ROS, analyzes the data and makes the level-2 data acceptance decision. The size of the pages inside the event buffers is fixed to be 1K bytes and the sizes of event data fragments are up to 1K bytes.

The curve in figure 6.7 shows the results of this test. Obviously the higher the level-2 data request rate is, the more data the ROS system must output. The more the output volume is, the more workload the ROBIN devices have to take and

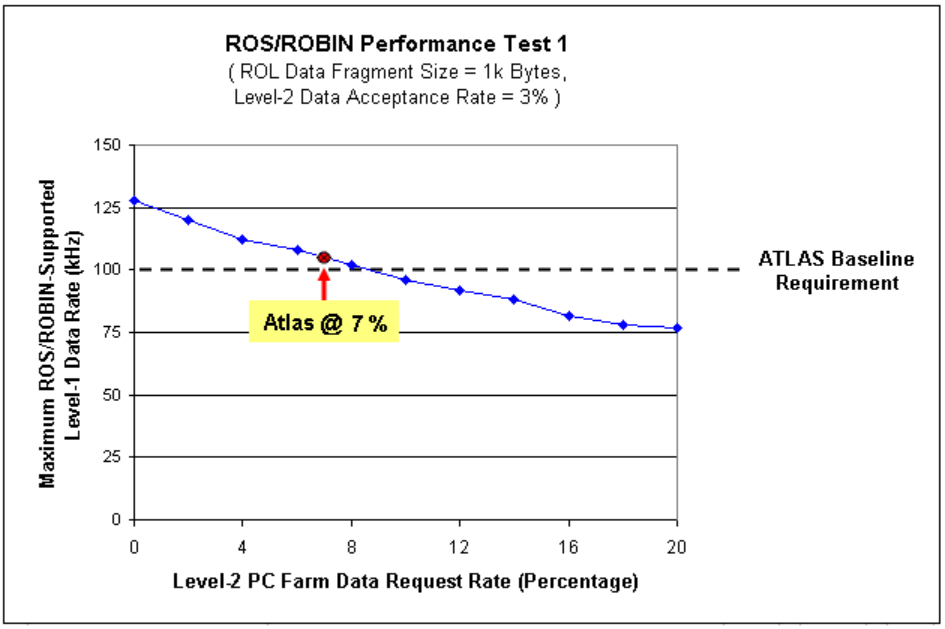

Figure 6.7: **Maximum ROS/ROBIN-supported level-1 data rate vs. level-2 PC farm data request rate**

accordingly the lower level-1 data rate the ROBIN devices can support.

According to the baseline requirements of the ATLAS data acquisition chain the output data rate of the level-1 trigger is at a maximum 100kHz. As shown in figure 6.7, to support this level-1 data rate, the data request rate by the level-2 PC farm can be up to 8.3%. This is above the expectation by the ATLAS community. The ATLAS community estimated, statistically around 7% of the level-1 event data are requested by the level-2 PC farm for data analysis and data selection.

6.3.3.2 Maximum Supported Level-1 Data Rate versus Event Builder Data Acceptance Rate

In this test the data request rate by the level-2 farm emulator is fixed to be 7%. The balance between the data input from the ROD emulator and the data request from the EB emulator is measured. That is, the maximum level-1 data rate that the ROS/ROBIN system supports is measured, while varying the data rate of the event builder. In this test the size of the pages inside the event buffers is also 1K bytes.

The diagram in figure 6.8 shows the results of this experiment. As with the above experiment the higher the EB data request rate is, the lower level-1 data rate the

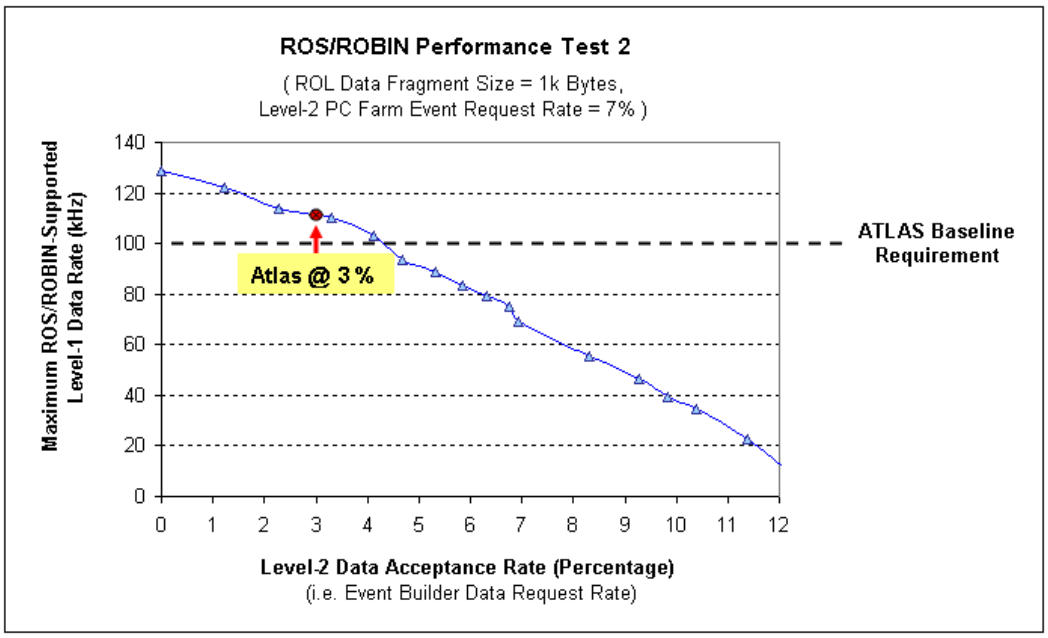

Figure 6.8: **Maximum ROS/ROBIN-supported level-1 data rate vs. event builder data acceptance rate**

ROBIN devices can support.

In the ATLAS DAQ the level-2 accepted events are always requested by the event builder. The ATLAS community estimated that statistically around 3% event data are accepted by the level-2 trigger. Therefore, the expected data request rate of the EB must be above 3%. According to the result of this simulation test as shown in figure 6.8, to support the ATLAS baseline requirement (i.e. a level-1 data rate of 100 kHz), in the current ROS/ROBIN system the event data acceptance rate by the event builder can reach 4.3%, which is above the ATLAS DAQ requirement.

6.3.3.3 Maximum Supported Level-1 Data Rate versus Event Data Fragment Size

This test aims to measure the maximum supported level-1 data rate in case of different event data fragment sizes. The diagram in figure 6.9 shows the test results. Two curves are drawn in the diagram. They are for the cases when the level-2 acceptance rate is 3% and 5%, respectively.

The greater the size of ROL event data fragments is, the more data volume has to be transmitted across the ROS/ROBIN system, and accordingly the lower level-1

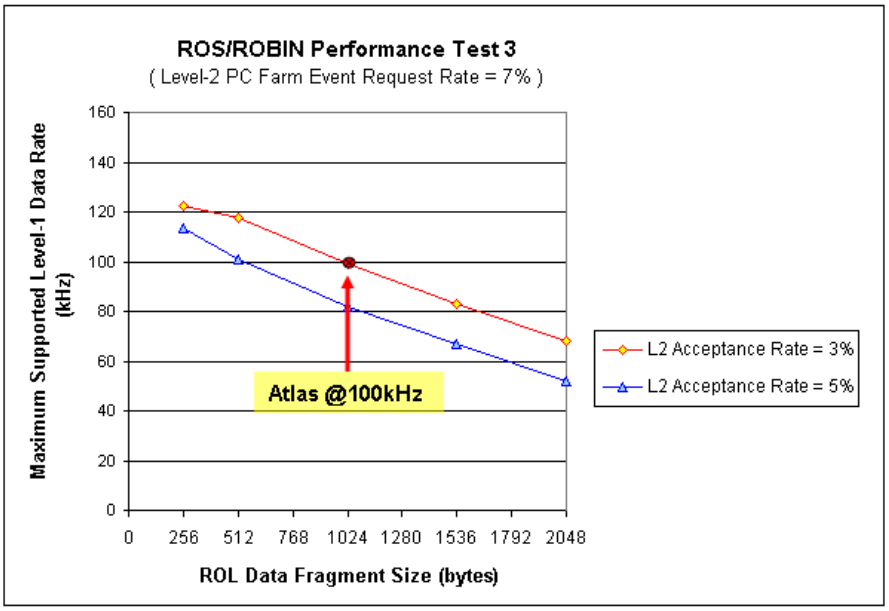

Figure 6.9: **Maximum ROS/ROBIN-supported level-1 data rate vs. event data fragment size**

data rate the ROBIN boards can support. To support a level-1 data rate of 100 kHz, the size of the data fragment could be 1024 bytes. This is generally above the ATLAS baseline requirement.

According to the experiments above, both the integrated ROB/ROBIN and the standalone ROBIN PowerPC application have reached the ATLAS baseline requirement. Compare the graph in figure 6.6 with the graph in figure 6.9. The standalone ROBIN PowerPC application itself has a relatively higher performance compared with the integrated ROB/ROBIN.

6.4 Summary

This chapter presents a ROBIN PowerPC system based on the non-OS architecture introduced in the previous chapter. Without an operating system the PowerPC application is implemented as a single-thread program. Its cyclic tasks of the PowerPC system, including free-page updating, used-page record handling, message handling and idle-task processing are performed in a single loop.

Since no real-time preemptive scheduler is available inside the standalone non-

OS based PowerPC system, a number of application-specific measures have to be figured out to organize operations in the standalone PowerPC application, in order to improve the overall processing rate of the system. Several measures are proposed in this chapter to optimize the the standalone PowerPC application. The objective of the optimization is to distribute the workload in the main loop uniformly into each cycle. With these optimization measures the maximum processing time of one main loop cycle is both minimized and expectable. Accordingly the overall processing rate of the PowerPC application is improved to meet the real-time performance requirement in the ROBIN system.

The proposed standalone PowerPC system is tested together with an entire ROS/ROBIN system in a simulated testing environment. Both the performance of the proposed PowerPC system itself and the performance of the entire ROS/ROBIN system with the PowerPC system integrated are tested. The experimental results tell the satisfying performances both of the PowerPC system and of the integrated ROS/ROBIN system, with respect to the baseline requirement of the ATLAS DAQ chain.

7 Real-Time Linux Based PowerPC Application

This chapter deals with the ROBIN PowerPC system built upon the OS-based system architecture. MontaVista real-time Linux is chosen as the operating system for the OS-based ROBIN PowerPC system. The real-time capability of the real-time Linux OS and the software design and optimization of the according ROBIN PowerPC application are addressed in the chapter.

The chapter is organized as follows. Section 7.1 reviews the features of a real-time Linux OS and discusses in particular its supports for real-time systems. Section 7.2 addresses the software design of the real-time Linux based ROBIN PowerPC application. Section 7.3 introduces strategies to optimize the performance and to improve the reliability of the real-time system. Finally, the performance of the RT-Linux kernel and the performance of the proposed OS-based ROBIN PowerPC system are measured and results are presented in section 7.4.

7.1 Real-Time Linux

Many commercial embedded real-time operating systems have emerged in the last decades, including VxWorks, Windows CE, QNX, and some versions of real-time Linux [65]. Real-time Linux is chosen for the implementation of the ROBIN PowerPC system, due to its open source, its flexibility and its ability to support a wide range of hardware platforms. Linux is inherently modular. It can be easily scaled into compact configurations and customized according to system-specific requirements.

This section reviews firstly several related concepts in real-time systems and then discusses in detail the real-time mechanism provided by the real-time Linux OS.

7.1.1 Concepts in Real-Time Systems

Stankovic and Ramamritham gave a formal definition to a real-time system [61]:

A real-time system is a system in which the correctness of the system depends not only on the logical results that the system produces, but also on the time in which the results are produced.

"Response time" is an importance concept in a real-time system. Dankwardt gave its definition [26]: the *response time* of an application is the time interval from when the application receives a stimulus, e.g. a hardware interrupt, to when the application has produced a result based on that stimulus. Along with the response time, "deadline" is another important term. The *deadline* of a given task is the longest acceptable response time of the task.

According to the strictness of the real-time requirements, real-time systems can be classified into two groups: hard real-time systems and soft real-time systems. A *hard real-time* system must guarantee that all the deadlines of the tasks must be met at any time. The system designer must make sure that the deadlines can be met, and that the system must not be overloaded. A *soft real-time* system, on the other hand, is a system in which the deadlines are generally met. In such a system it may be acceptable if a small number of deadlines are occasionally missed [26]. For example, an air-traffic controller is an example of a hard real-time system, where it is critical that every deadline is met. An audio sampling application could be an example of a soft real-time system, where it is still acceptable if some samples are lost from time to time, as long as it does not happen too often.

As the ROBIN PowerPC system is concerned, the system is a hard real-time system, since the deadlines of its tasks must be strictly met. Otherwise, the incoming event data from RODs are not processed in time, which may lead to data overwriting and data missing.

7.1.2 Traditional Linux Kernel and its Limited Real-Time Capability

Real-time capability of an operating system depends principally on its task scheduling. It needs to determine, when to call the scheduler under a given circumstance and which task to choose for the next to run. These two points are termed as the scheduling time and the scheduling strategy, respectively. When a real-time task is requested to run, a real-time system must guarantee the task being scheduled within a certain time delay and also guarantee the task is chosen as the next task to be called by the scheduler [58].

7.1.2.1 Linux Scheduling Time

A task may terminate itself at any time or suspend itself by calling the system functions, e.g. pause(), sleep() and etc. In such cases the task gives up the CPU actively and causes the scheduler function to be called. Obviously, scheduling caused by the volunteered task termination or suspension cannot meet the real-time requirement. For example, when a task is in the running, a real-time task request occurs; but the running task itself does not give up running; at last, the real-time task fails to start in time.

Therefore, a preemptive scheduling is required. When a real-time task request occurs, the system must have the ability to terminate or suspend a currently running task and to schedule the real-time task to run. In Linux the preemptive scheduling occurs each time when the system returns from the kernel space to the user space, i.e. when a system call or an interrupt or error handling call is finished.

The Linux operating system provides a scheduling mechanism based on cyclic preemption points [50] [60]. At each preemption point the OS scheduling function is forced to be called. The preemption points give the operating system the possibility to suspend or terminate a running task even when it refuses to give up running. For easy understanding of the scheduling mechanism, figure 7.1 gives a simplified illustration of the preemption-point based scheduling. When a real-time task request arrives, it must wait till the next preemption point for the task to be called. Actually a preemption point is a time span, instead of a time point, since the scheduling call at a preemption point costs also CPU time. Figure 7.2 shows the actual task switching processing for the example given in figure 7.1.

Linux applies a cyclic timer interrupt to realize the preemption points. The cycle time varies for different versions of Linux. It is, however, always above microseconds for the Linux systems. The cycle time decides the granularity of the preemption points as well as the timing preciseness of the system. The response time for a real-time task is also much related to this factor. For example, a cyclic task with a cycle time of 100 μs cannot be realized on a Linux OS.

Besides, the Linux kernel the preemption-point scheduling is disabled. That means, when the kernel is running, a real-time task cannot be called even at preemption point. The calling time of a real-time task is non-deterministic. This is a significant limitation for Linux to be used as a real-time system.

7.1.2.2 Linux Scheduling Strategy

Linux supports three scheduling schemes: SCHED-FIFO, SCHED-RR and SCHED-OTHER. SCHED-FIFO and SCHED-RR are for real-time tasks, while SCHED-

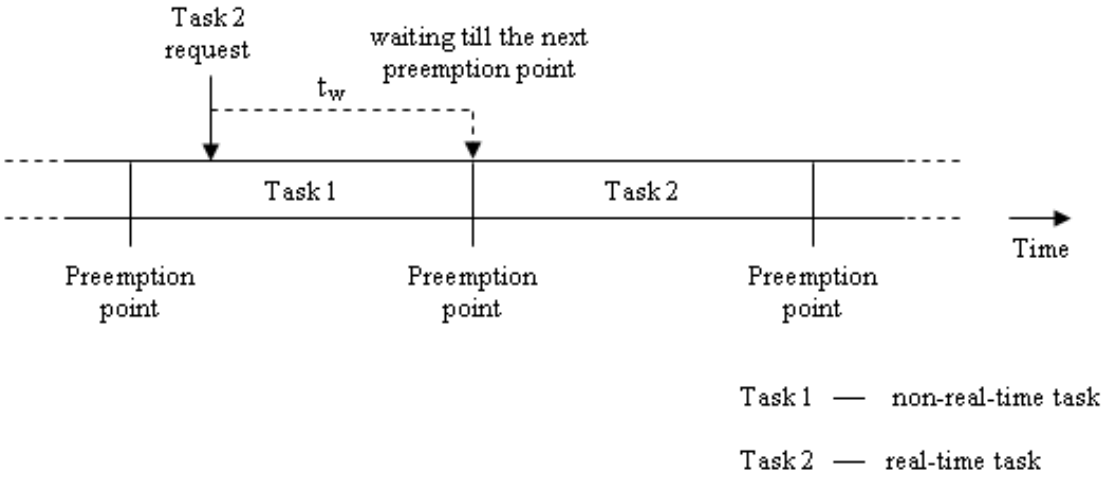

Figure 7.1: **Simplified illustration to the mechanism of the preemption-point based scheduling.**

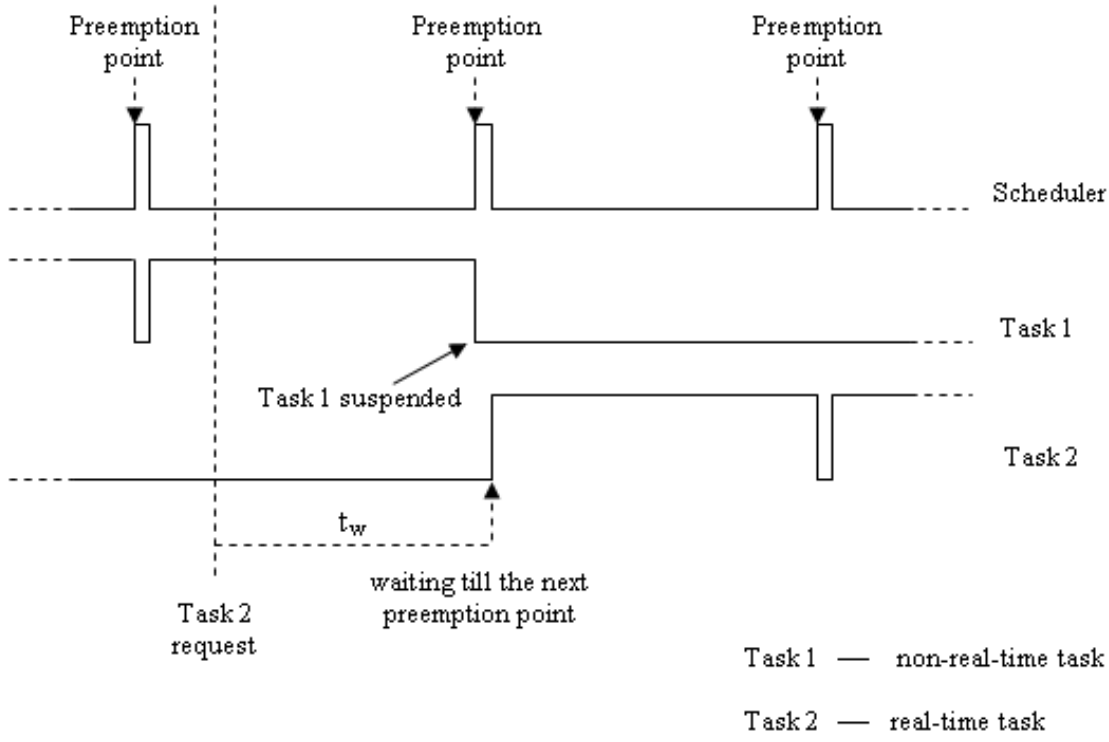

Figure 7.2: **Actual task switching process for the example given in figure 7.1.**

OTHER is for non-real-time tasks. SCHED-FIFO is usually used for the real-time tasks with relatively shorter running time. When these tasks are running, they are not allowed to be preempted. SCHED-RR is used for the real-time tasks with relatively longer running time. Among these real-time tasks with SCHED-RR the round-Robin scheduling is applied.

Although each task has a certain scheduling scheme, the scheduling order of the tasks depends on a positive weight value of each task. The scheduling scheme of a task is surely taken into account when computing its weight value. Besides, for a real-time task its weight value is always multiplied with a significant factor (e.g. 1000). Therefore, when a real-time task is ready, a non-real-time task has no chance to be scheduled.

The above three scheduling schemes of Linux guarantee a real-time task to be scheduled. In other words, the Linux OS is preemptible. However, the scheduling time of a real-time task is still non-deterministic [17]. If a real-time task has a deadline, a Linux OS cannot guarantee to meet the deadline. Therefore, the Linux OS cannot be used in a "hard" real-time system, but only in a soft real-time system, where the deadlines of real-time tasks are relative longer or must not be met strictly.

7.1.3 Improvements in Real-Time Linux Kernel

7.1.3.1 An Additional Real-Time Kernel Layer

Many researches have been done to improve the real-time performance of Linux. One widely-used approach is to add a hardware abstraction layer between system hardware and Linux. Also a new separate real-time scheduler is used which runs Linux as its lowest priority thread. The hardware abstraction layer takes control over the system interrupts and passes them on to Linux only if no real-time task is running. When Linux tries to disable interrupts, it only sets a flag in the hardware abstraction layer and cannot really turn off the interrupts or prevent itself from being preempted. Thus, the real-time scheduler has full control over the system and Linux runs virtually unmodified. Actually the hardware abstraction layer and the separate real-time scheduler construct another OS kernel, i.e. a small "hard" real-time kernel.

Above such a real-time Linux OS a user application is usually written in two parts: time-critical part and non time-critical part.

- Tasks in the time-critical part have strict timing requirements. They are written as kernel modules and executed as real-time tasks within the *kernel space*,

which prevents the tasks to be swapped out and also the number of TLB (Translation Lookaside Buffer) misses is reduced. These tasks have full access to the kernel real-time API and the underlying hardware.

- Tasks in the non time-critical part, such as user interface, are written and executed in the user space outside of the kernel.

The two different parts communicate with each other through, for example, FIFO queues or shared memory. The software split requires obviously a new design of the application. Besides, the real-time tasks are written as kernel modules using the kernel real-time API, instead of the standard Linux API. Different programming skills are required from the application developers to write Linux kernel modules instead of a Linux application process. More cautions must be taken, since an error in a kernel module may crash the whole system. This is a price that has to be paid for the additional deterministic environment, which is required by every hard real-time system.

RTLinux [30] and RTAI [27] are two extensions of Linux that provide the above technique about a real-time kernel layer. RT-Linux is a shared space system. That means, both the OS kernel and real-time tasks are running in the kernel space and the user space is only for the non-real-time tasks to run. Due to its compact small-sized real-time kernel it is chosen for the ROBIN PowerPC system.

Figure 7.3 shows the kernel structure of RT-Linux. Through combining the RT-Linux kernel with the Linux kernel, RT-Linux not only meet the requirement of real-time systems but also has the possibility to access numerous powerful functions in the modern Linux operating system.

7.1.3.2 Increasing the OS Timing Preciseness

As mentioned above, the timing preciseness of a traditional Linux OS is at the millisecond level. This preciseness is much below the real-time requirement of many real-time systems. Hence an intuitive idea to improve the real-time performance of the Linux OS is to increase the granularity of its preemption points. In this case the kernel checks more often if a higher priority process is ready to run, to reduce the response time to the real-time tasks.

Moreover, to further increase the preciseness of the OS timing system, the RT-Linux scheduling uses a one-shot timer instead of a cyclic timer, which was used previously to realize cyclic preemptive points, and adjusts the one-shot timer for the scheduling point of the next real-time task. In such a way the CPU resources can be much spared, since there is no need any more to call the OS scheduling cyclically at

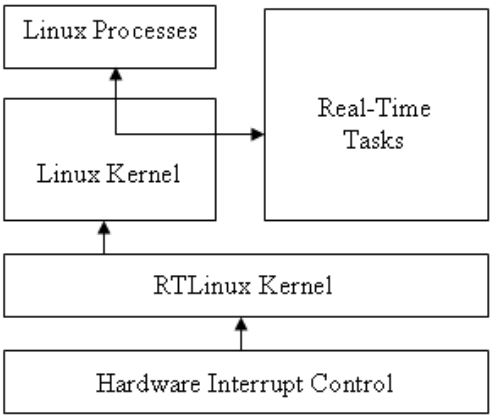

Figure 7.3: **RTLinux kernel structure.**

each preemptive point even when no high-priority task occurs. Besides, the waiting time t_w for a high-priority task request can also be spared. For the notation t_w refer to figure 7.1 and figure 7.2.

7.1.4 Choice of MontaVista Linux

The Linux community is very active in adding features to support new hardware, fixing bugs in the kernel, as well as making general improvements in a timely manner. This results in having a new release of a stable Linux tree roughly every 6 months or less. Different kernel trees and patches for specific architectures are maintained by different maintainers. When choosing a kernel for a project, one needs to evaluate how stable the release is, whether it caters to the project requirements and the hardware platform, whether it comforts the programming from the point of view of the system developers, and so on. It is also very important to find out all of the patches that need to be applied to the base kernel to tune it for the specific architecture.

Several alternatives of real-time Linux operating systems have been developed in the last decades of years. MontaVista Linux was chosen and built in the ROBIN PowerPC system, because it is one of leading real-time Linux solutions and is reported to support the PowerPC very well [51]. The MontaVista Linux offers a wide range of benefits in terms of reducing time requirements and minimizing risk.

Before compiling MontaVista, the real-time Linux kernel must be configured firstly with proper parameters for the ROBIN board. The MontaVista Embedded Linux

3.1 is applied to the ROBIN PowerPC system, which is running with Linux kernel 2.4.26.

MontaVista was designed to run virtually for any embedded system. Now that the OS kernel is available, the ROBIN Board Support Packages(BSP) needs to be added to make the kernel fit into the ROBIN board.

7.2 Software Design

Due to the same functionalities the software design of the RTLinux-based PowerPC application shares a same component diagram and a same use case diagram as in the standalone PowerPC application. Refer to section 6.1.1 and section 6.1.2 for details of the two diagrams, respectively. Difference between the RTLinux-based PowerPC application and the standalone PowerPC application lies mainly on its scheduling of its predefined tasks.

7.2.1 Multi-Task Scheduling

The standalone PowerPC application implements its tasks in a single loop and adjusts a cycle counter to control the calling rate of each task. A disadvantage of this single-loop mechanism is the possibility that multiple tasks must be finished in one loop. This leads to extended cycle time every now and then, such that the system is not capable to sustain an expected data processing rate stably, especially a stable handling rate of incoming event data or used-page records. Therefore, several application-specific approaches have to be proposed in section 6.2 in order to optimize the performance of the standalone ROBIN PowerPC application.

For the OS-based ROBIN PowerPC application, the multi-task scheduling of the real-time Linux OS provides a generic solution to the scheduling of its multiple tasks. Tasks are prioritized. A higher-priority task can preempt a running lower-priority task. Each task can define its own individual cycle time. It is the job of the operating system to manage the scheduling of the tasks automatically. It guarantees firstly the highest-priority task to be performed within its required cycle time, then the second highest-priority task and so on. In a logic real-time application the expected cycle time of each of its tasks are met strictly.

Figure 6.3 shows five tasks in the main loop of the standalone ROBIN PowerPC application. In the OS-based ROBIN PowerPC application each of the five tasks can run in a single thread. The priorities of the threads are set according to their real-time requirement. According to the real-time requirement analysis of these tasks as shown in table 5.1, the task of used-page record handling requires the highest

processing rate and accordingly the highest priority, and then the message handling and the free-page update, sequentially. The idle task handling and the terminal command handling have the lowest priority.

7.2.2 Cautions in Multi-Task Scheduling

Two things must be taken care of in the development of an RTOS-based application. One is about shared resources among the multiple tasks; the other is about the overall performance of the application, i.e. whether the required cycle time of each task is met strictly. The second point tells also the performance feasibility of an application.

7.2.2.1 Shared Resources

For single-loop scheduling, tasks are called one after another and at one moment shared resources are always occupied by a single task. No conflict takes place in that case. In an RTOS-based application, an unfinished operation on a shared resource inside a lower-priority task may be broken up by a higher-priority task and the higher-priority task may modify the shared resource further based on the unfinished operation. For example, the hash table of used-page records in the ROBIN PowerPC application is a shared resource for the task of message handling and the task of used-page record handling. While the former task is deleting a used-page record from the hash table according to the command of an event deletion message, the latter task may preempt the former and insert newly-incoming used-page records into the hash table. This may cause fatal data inconsistence in the ROBIN system.

Therefore, cautions must be taken to protect shared resources among the tasks, either by defining a shared-resource operation as a critical section that allows no preemption, or by using a mutex lock to protect a same share resource.

7.2.2.2 Performance Feasibility

Although the real-time requirements of higher-priority tasks are firstly satisfied, the performance of the overall application must also be considered. Design of the whole application must guarantee the required cycle time of each task is met strictly. In single-loop scheduling, overloaded tasks cause a performance dropdown of the overall application. But for real-time multi-task scheduling, accumulated overloaded job may crash the system.

Therefore, a reasonable determination of the cycle time of each task is particularly important for a real-time system with multi-task scheduling. It decides the feasibility

of the application. A too long cycle time may not meet the real-time performance requirement of the system, while a too short cycle time may cause system crash by overburdened job.

7.3 Performance Optimization

The most principal performance cost for automatic priority-based real-time multi-task scheduling is the overhead of task switching. Therefore, in order to improve the performance of a real-time system with multi-task scheduling, it is not only necessary to improve the efficiency of each task by reducing the processing time of the tasks themselves, but also necessary to reduce the switching rate among the tasks.

Apparently the task switching rate is in inverse proportion to the cycle time of the tasks. When the tasks are called more often, switching between the tasks occurs also more frequently. Besides, it will be shown in the following that the task switching rate is in direct proportion to the number of tasks, when the computational time of the tasks are relatively short. Based on the two principles, this section aims to minimize the overhead of task switching in the OS-based ROBIN PowerPC application by reducing the number of scheduled tasks reasonably and determining a moderate cycle time for each of the tasks.

To distinguish the cyclic tasks in the ROBIN PowerPC application from the tasks scheduled by the OS scheduler, the latter "tasks" are termed as "OS tasks" in this section.

7.3.1 Reducing the Number of Tasks

As stated in section 7.2.1 there are five cyclic tasks in the ROBIN PowerPC application. (Refer to figure 6.3 in section 6.1.3 for the details of the five cyclic tasks.) The simplest design for the OS-based PowerPC application is to let each of the five tasks run in one single thread and use the OS multi-task scheduler to schedule the five tasks.

Except the highest-priority task of used-page record handling, which has a strict limitation to its minimum processing rate of 25 kHz (see table 5.1), the other four tasks only need to reach an average processing rate. In other words, the used-page record handling task is a "hard" real-time task and the other four tasks are "soft" real-time tasks. Therefore, there is a great flexibility to join the four lower-priority tasks into a smaller number of OS tasks or to adjust their cycle time to reach an optimal performance of the PowerPC application.

Consider the design of the standalone non-OS based ROBIN PowerPC application, which implements all its five cyclic tasks in one loop. Following the idea of one-loop-of-subroutines, an extreme design is to put all the four lower-priority tasks into one OS task and use a cycle counter to adjust the relative processing rates of the four tasks. Since the four tasks are soft real-time tasks, the extreme design is theoretically acceptable, so long as the average processing rates of the tasks are reached. In this case there are only two OS tasks scheduled by the operating system. However, this does not guarantee a reduced rate of task switching. If the running time of the lower-priority task at one cycle is too long, e.g. multiple times of the cycle time of the higher-priority task.

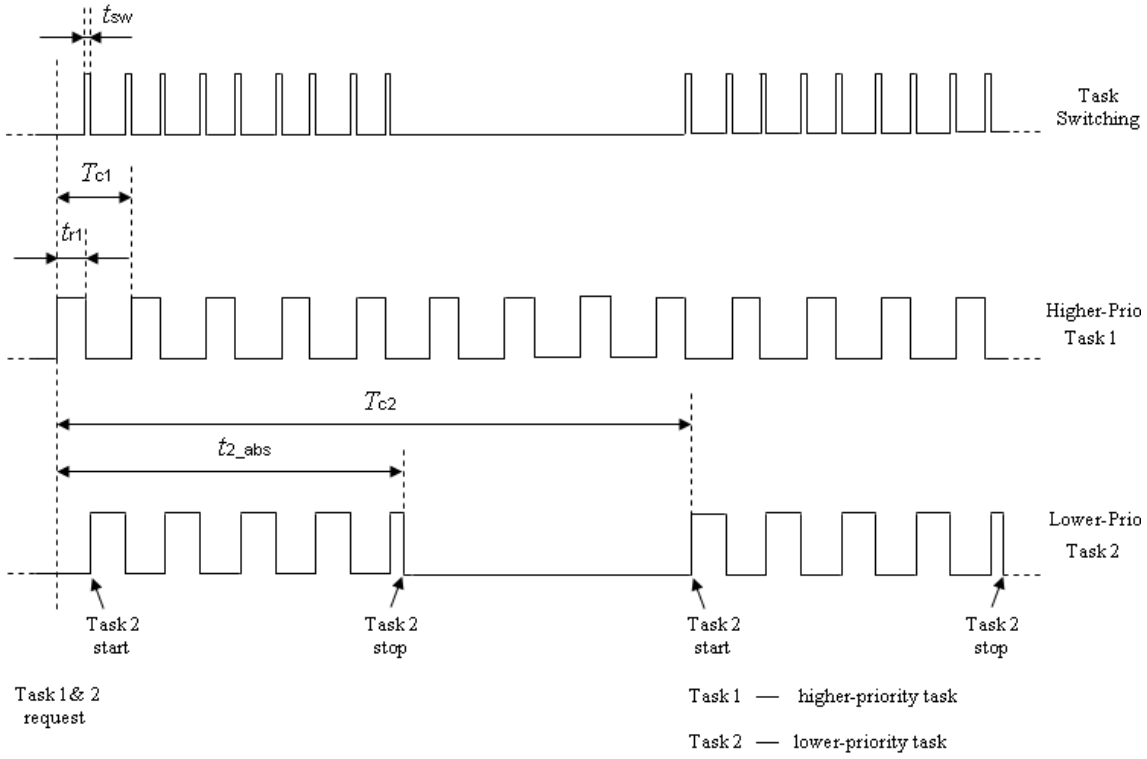

Figure 7.4: **Task switching.**

Figure 7.4 shows an example for task switching. Two cyclic tasks are sharing the CPU in the example. Task 1 is a higher-priority task. Task 2 is a lower-priority task. Task switching occurs, when task 1 preempts task 2 or when task 1 finishes its operation at one cycle and task 2 resumes its operation. To simplify the denotation,

it assumes that the computational time for task switching at any situation is constant and the computational time for either task at each cycle is also constant. Besides, the starting requests of both tasks are assumed to arrive at a same time. In this case the number of task switching within one task-2 cycle is always an odd number. Let N denote the number of task switching and let $N = 2 * n + 1$. Then we can derive the following equation.

$$t_{2\_abs} = 2 * n * t_{sw} + n * t_{r1} + t_{r2} = n * T_{c1} + t_{2\_extra}$$

where

- t_{sw} is the computational time for task switching,
- T_{c1} is the cycle time of task 1,
- T_{c2} is the cycle time of task 2,
- t_{r1} is the computational time of task 1 at one cycle,
- t_{r2} is the computational time of task 2 at one cycle,
- t_{2_abs} is the absolute running time of task 2 at one cycle, (i.e. from task-2 start to task-2 stop)
- and t_{2_extra} is a small fraction of t_{r2} value, equal to $t_{r2} - n * (T_{c1} - (2 * t_{sw} + t_{r1}))$ and smaller than $T_{c1} - t_{r1} - 2 * t_{sw}$.

See figure 7.4 for the illustration of the above denotation. When t_{sw}, T_{c1} , T_{c2} , t_{r1} and t_{r2} are known, a unique solution to the integer n can be derived from the above equation. A similar equation can be derived when more than two tasks get involved in the scheduling.

The example in figure 7.4 tells, the task switching rate does not always drop significantly, when the number of tasks is reduced. This is the case of the extreme design as mentioned above, i.e. merging all the four lower-priority tasks into one OS task. The computational time of the four merged tasks is a number of times of the cycle time of the single highest-priority task of used-page record handling.

However, when two or more tasks with relatively short computational time, (shorter than $T_{c1} - (2 * t_{sw} + t_{r1})$ for the example in figure 7.4,) merging of these tasks will obviously decrease the task switching rate. This is the case of the three lowest-priority tasks in the ROBIN PowerPC application: free-page ID FIFO updating, idle task handling and terminal command handling. Each of the three tasks has very short

computational time and their processing rates are also by far lower than the other two highest-priority tasks. Therefore, the first optimization measure is to merge the three lowest-priority tasks into one OS task. The number of OS tasks to be scheduled by the real-time operating system is reduced to the following three in the descending priority order:

- OS Task 1 : used-page record handling,
- OS Task 2 : message handling,
- OS Task 3 : free-page ID update, idle task processing and terminal command handling.

7.3.2 Determination of Task Cycle Time

For a real-time system it is necessary to set a reasonable cycle time for each of its tasks, so that it is not too long to miss the real-time performance requirement of the system, and that it is not too short to cause too much overhead of task switching and lead to system overload.

A overloaded real-time system results in task overrun. It means that not all its real-time tasks are executed within their respective expected cycle time. For instance, for the example system with two real-time tasks as shown in figure 7.4, the system gets overloaded when the following condition is satisfied:

$$t_{2\_abs} \geq T_{c2}.$$

In this case the actual running time of the lower-priority task is longer than the expected cycle time of the task. If the coming task-2 requests are not ignored, the waiting task queue grows, which will eventually leads to system crash.

As mentioned in the previous section the five tasks in the ROBIN PowerPC application run in three cyclic OS tasks, i.e. three real-time tasks scheduled by the operating system. The following of this subsection discusses the strategies to determine the cycle time of the three tasks.

7.3.2.1 OS Task 1

Used-page records are handled in the first cyclic task. This most time-critical task has the highest priority in the system. The maximum allowed cycle time for this task is 40 μs as given in table 5.1. For security the actual cycle time must be shorter than 40 μs. In this work the shortest system-allowed cycle time of this task is not

determined a priori, but regarded as a criterion to evaluate the real-time capability of the ROBIN PowerPC system. Experiments presented in section 7.4 are done to measure the shortest cycle time of this task that the OS-based ROBIN PowerPC application is capable to reach.

7.3.2.2 OS Task 2

Message handling task is performed in the second OS task. A limitation to the average processing rate of this task is 1 kHz, which is also given in table 5.1. But the determination of the cycle time of this task is more flexible, since shorter cycle time means in one cycle fewer messages arriving at the message description FIFO and fewer messages need to be handled. It is not absolutely necessary to process all 32 messages in the message description FIFO in one cycle. According to practices a balanced choice of the cycle time for message handling task is set to be 200 μs. It is five times as the cycle time of the used-page record handling task and one fifth of the maximum-allowed average cycle time of the task as given in table 5.1. It is easy to count, there are 6.4 (i.e. 32/5) messages on average to be handled in one cycle.

7.3.2.3 OS Task 3

Free-page ID FIFO update is the only job in the third OS task that has certain real-time requirement. In order to reduce the burden of the system this thread is called so often as necessary. For this purpose the cycle time of this OS task is adjusted dynamically during the running according to the following strategy.

At each cycle of OS task 3, the cycle time for its next round T_{c3} is re-determined by the current number of free-page IDs in the free-page ID FIFO n_f and the a-priori known event data rate (or free page occupation rate r_f), i.e. $T_{c3} = n_f/r_f$. At the system initialization the event buffer are empty and all pages are free. Hence, the initial number of free pages in the free-page ID FIFO n_{f_0} is the minimum of the size of free-page ID FIFO S_f and the total number of pages N_p, i.e. $n_{f_0} = \min(S_f, N_p)$. Therefore, the initial cycle time of thread 3 is $T_{c3_0} = n_{f_0}/r_f$.

7.4 Experiments

Like the standalone ROBIN PowerPC application, the OS-based PowerPC application is also developed with a cross development environment as described in appendix B. In this section the real-time performance of the MontaVista RT-Linux OS is firstly measured. Then the ROBIN PowerPC application based on the MontaVista Linux

is tested. In order to reach the best performance of the OS-based ROBIN PowerPC system, the system performance is tested against difference cycle rates of its tasks.

7.4.1 Performance of MontaVista RT-Linux Scheduling

This subsection aims to measure the performance of MontaVista real-time Linux operating system. The most concerned OS performance for the ROBIN PowerPC application is the performance of the OS scheduling and in particular task switching delays caused by the scheduling.

For the ROBIN PowerPC application there are two scenarios for task switching. In the first scenario a higher-priority task preempts a running lower-priority task; a delay is caused when the running lower-priority task is suspended and the higher-priority task is activated. In the second scenario a suspended lower-priority task is resumed when all higher-priority tasks finish their operations; a delay occurs when the last higher-priority task is terminated or suspended to wait for the next cycle and the lower-priority task is resumed.

Experiments in this section are to measure the two kinds of task switching delays caused by the OS scheduling. Two tasks are defined in the test: *task 1* and *task 2*. *Task 1* is a cyclic task and its cycle time is one millisecond. At each cycle *task 1* switches a LED on and off once. *Task 2* is continuously running task, which toggles another LED constantly. *Task 1* has a higher priority compared with *task 2*. Since the priority of *task 1* is higher, it preempts the continuously-running *task 2* cyclically.

The following is the code of *task 1*:

```
while (true)
{
        SetLedOn(1);    /* set LED 1 on */
        SetLedOff(1);   /* set LED 1 off */
        Sleep(1);       /* suspend itself for one millisecond */
}
```

The code of *task 2* is given as follows:

```
while (true)
{
        SetLedOn(2);    /* set LED 2 on */
        SetLedOff(2);   /* set LED 2 off */
}
```

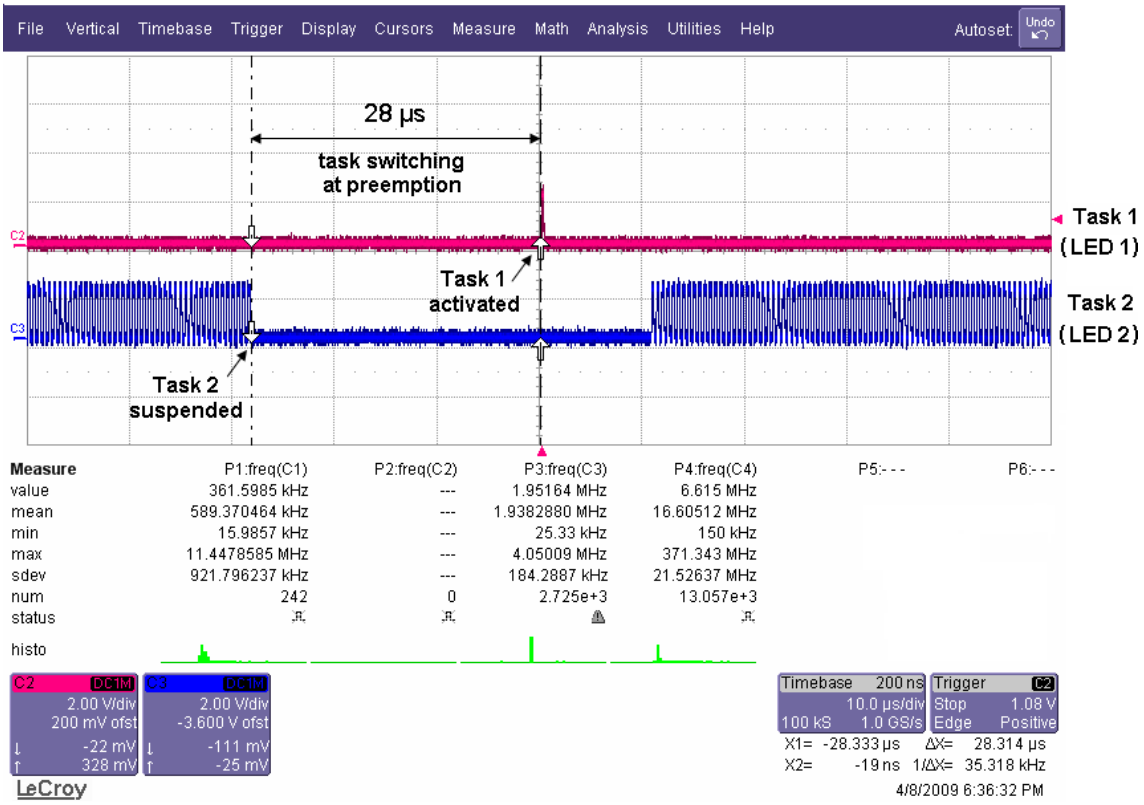

Figure 7.5: **Task switching delay caused when a higher-priority task preempts a running lower-priority task.**

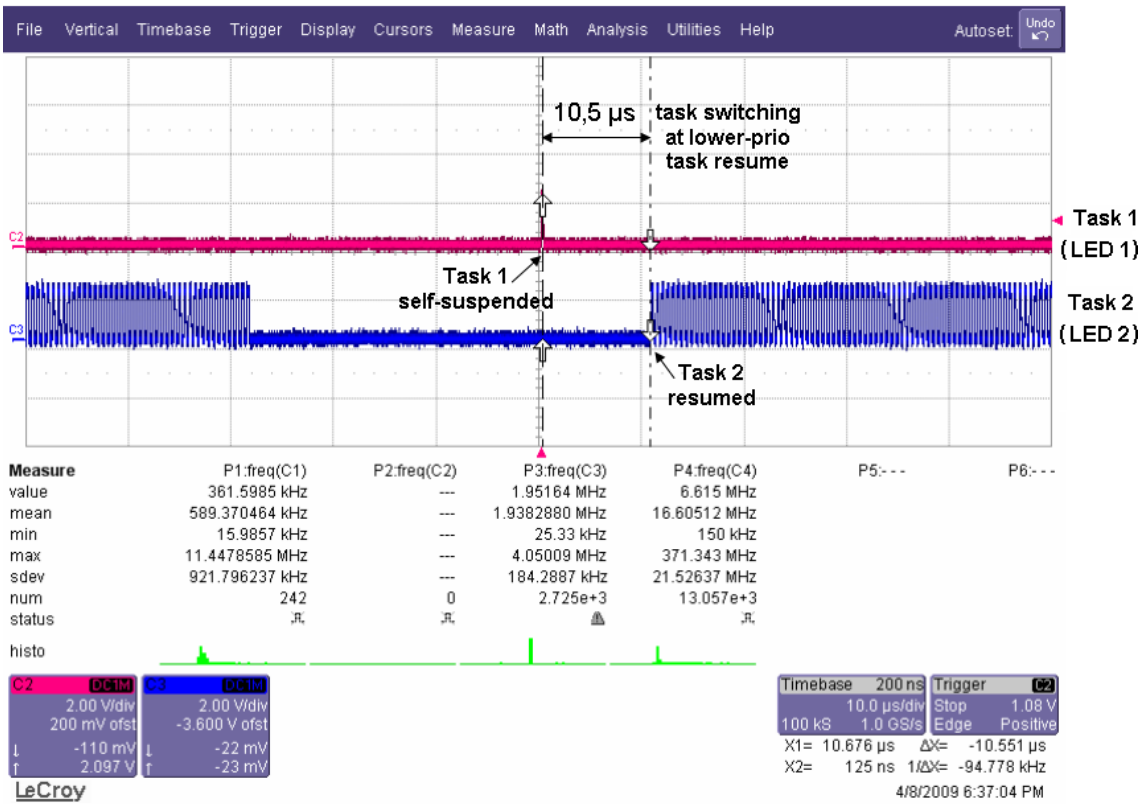

Figure 7.6: **Task switching delay caused when a higher-priority task finishes its operation in one cycle and a waiting lower-priority task is resumed.**

In the first task switching scenario *task 1* preempts the running *task 2*. The LED controlled by *task 2* stops toggling and a pulse is generated at the LED controlled by *task 1*. An oscilloscope is used to detect the signals at the two LEDs. Figure 7.5 shows the result acquired by the oscilloscope. According to the figure the delay caused by the task switching at task preemption is around 28 microseconds.

In the second task switching scenario for this test, *task 1* finishes its operation in the current cycle and suspends itself to wait for its next cycle and the waiting *task 2* is resumed. That is, a pulse has been generated at the LED controlled by *task 1* and the LED controlled by *task 2* resumes toggling. Figure 7.6 shows the measurement result of the task switching delay at lower-priority task resume. According to the figure the task switching delay in the second task switching scenario is about 10,5 microseconds.

7.4.2 Performance of RTLinux-Based ROBIN PowerPC Application

As mentioned in section 7.3, three final cyclic OS tasks are defined and scheduled by the RT-Linux operating system. Refer to page 103 for the definitions of the three tasks. The maximum cycle rate that the first OS task (i.e. used-page record handling) can reach is the most important concern for the real-time performance of the OS-based ROBIN PowerPC application. As mentioned before, this factor is regarded as a criterion to evaluate the real-time capability of the ROBIN PowerPC system. The experiment in this subsection is to find the optimal setup of the OS-based ROBIN PowerPC application, to reach the best real-time performance of the PowerPC system, i.e. the maximum cycle rate that the first OS task of used-page record handling.

The experiment is carried out under the ROS/ROBIN testing environment as described in section 6.3.1. On one side a ROD emulator generates event data and feeds the data to the ROS PC which is mounted with ROBIN boards. On the other side two data collection PCs emulate the level-2 farm and the event builder, respectively. They generate and forward data request messages and event deletion messages to the ROS/ROBIN system and collect requested event data backwards.

The cycle time of the second and the third OS tasks is determined according to the strategies introduced in section 7.3.2. The cycle time of the second OS task is fixed to be 200 μs, while the cycle time of the third OS task is adjusted dynamically according to the actual system workload. The experiment in this section is to measure the maximum cycle rate of the first OS task, while varying event data request rate. As with the experiment in section 6.3.2 an additional prerequisite for this experiment

is a full load of incoming event data, which is as much as the system can process. In other words the used-page record FIFO is always full at each cycle of the first OS task for used page handling.

Figure 7.7 shows the curve of the maximum cycle rate of the used-page-handling task while the event data request rate increases. Figure 7.8 shows the corresponding maximum incoming rate of the event data that the OS-based ROBIN PowerPC application is capable to support.

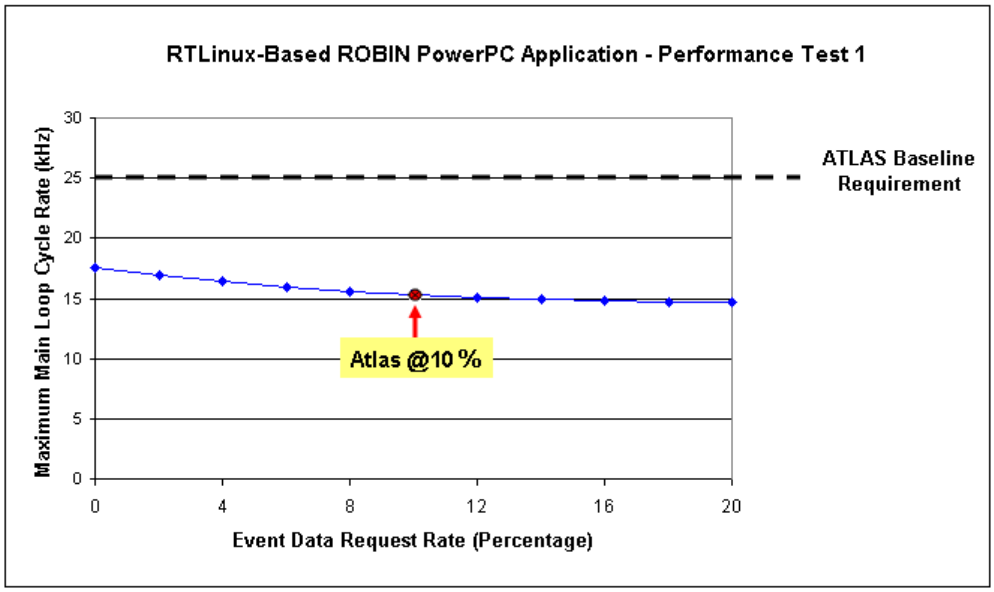

Figure 7.7: **Maximum cycle rate of the used-page handling task (supported by the OS-based ROBIN PowerPC system) vs. event data request rate**

According to the two diagrams the performance of the proposed RTLinux-based ROBIN PowerPC application does not meet the ATLAS baseline requirement. Only when every event data fragment is guaranteed to take no more than one page in event buffers, the performance of the RTLinux-based PowerPC application is capable to meet the real-time requirement. However, this is not always the case for the ROBIN system.

Analyzing the consumption of the CPU time, it is easy to tell that most of the CPU time is devoted to the OS scheduling. According to the performance measurement of the RTLinux scheduling, 28 μs is needed for task preemption and 10 μs for suspended task getting resumed. This means, for this experiment the OS scheduling for task switching consumes over two thirds of the total processing time, while less than one

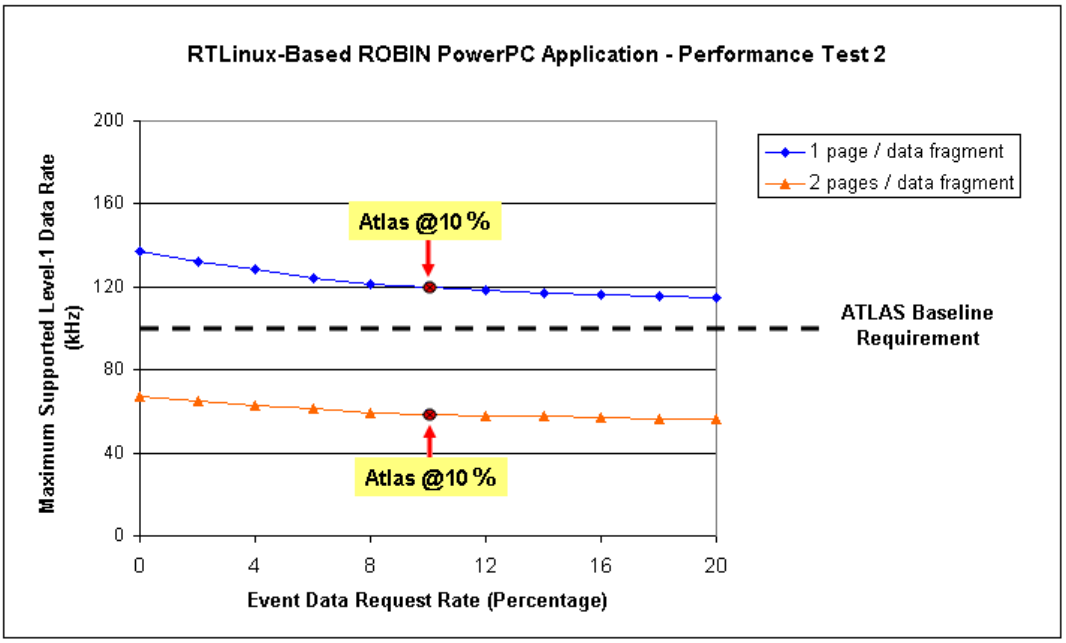

Figure 7.8: **Maximum level-1 data incoming rate (supported by the OS-based ROBIN PowerPC system) vs. event data request rate.**

third of the CPU time is devoted to task execution.

7.5 Summary

This chapter presents an OS-based ROBIN PowerPC system. MontaVista RT-Linux is chosen as the real-time operating system for the PowerPC system. With the existence of a real-time operating system, user tasks are prioritized and scheduled automatically by a real-time OS scheduler, to achieve a hard real-time performance of the tasks.

However, one tradeoff for this convenience is an extra computational cost for the multi-task scheduling or task switching. In order to reduce this cost several measures are proposed in this chapter to improve the performance of the system by reducing the number of user tasks and making an optimal choice for the cycle time of the tasks.

Finally, the performance of the RT-Linux kernel as well as the performance of the proposed OS-based ROBIN PowerPC system are measured. Results show that the performance of the RTLinux-based ROBIN PowerPC system does not meet the related ATLAS DAQ baseline requirement. However, according to the analysis of

the CPU time consumption, most of the computational time is devoted to the OS scheduling for task switching. The attempt in this chapter tells that there would be a chance for an OS-based ROBIN PowerPC system to meet the ATLAS DAQ baseline requirement, if an upgraded real-time operating system would emerge in the future with a real-time scheduler of higher performance, especially for task switching.

8 Conclusions

The goal of this dissertation is to realize an embedded real-time system for the ATLAS Readout Buffer INput (ROBIN). ROBIN is the centric device inside the ATLAS readout subsystem (ROS), which is one of the most essential buffering systems in the LHC/ATLAS data acquisition chain (TDAQ).

For the final design of ROBIN the ATLAS community decided to adopt the one based on two kernel processors: a Xilinx Virtex II 200 FPGA and a PowerPC 440GP micro-controller. The combination of an FPGA processor with a PowerPC micro-controller takes advantages of both kernel processors. The former controls the data flow with high performance requirements, and the latter is responsible for more complex and flexible management functions with relatively lower performance requirement. This work focuses on the design and optimization of the ROBIN PowerPC system.

Due to limited resources in the embedded ROBIN PowerPC system and strict real-time performance requirement, effective strategies have been studied in this work for the ROBIN event buffer management, which are economic both in the memory space and in computational cost. Three algorithms are introduced in this work for the event buffer arrangement and assignment, the fast event lookup and the storage of the related data structures.

Firstly, a page-based scheme is adopted for the organization of event buffers, i.e. segmenting a 64MB SDRAM event buffer into fixed-sized pages. A hash table is introduced to deal with the mappings between event IDs and page IDs, which guarantees a balanced distribution of hash nodes over hash buckets. The dynamic organization of the hash table is managed with static data structures. Dynamic memory allocation is avoided in order to keep system security and stability. Secondly, since there exists an one-to-one mapping between occupied pages in the event buffer and hash nodes in the hash table, a same mechanism is proposed both for the arrangement of the event buffer and for the arrangement of the static hash node

buffer. In such a way the computational effort of the ROBIN PowerPC application is skillfully reduced. Thirdly, a chained free hash-node list is introduced as the buffer allocation mechanism for the two buffers above. The chained free hash-node list is built within the hash node buffer, with no extra memory space. This proposed buffer allocation strategy based on a chained free-node list can be easily extended to handle buffer management problems for other embedded systems. The solution even contributes to solve a generic memory management problem, if the memory has to be divided into partitions with fixed size and each partition is a minimum unit for memory allocation and release. In such a case, the proposed algorithm is an optimal solution to the memory management both in respect of space complexity and in respect of time complexity.

With given strategy and algorithms for the ROBIN event buffer management, the primary software components and functionalities for the ROBIN PowerPC system are defined. Moreover, following the baseline requirement of the ATLAS data acquisition chain, the real-time performance requirements of the PowerPC software are also determined.

For the implementation of the ROBIN PowerPC system, two architectures are presented in this work, depending on whether a real-time operating system is integrated into the system.

In the standalone PowerPC system based on the non-OS architecture the ROBIN PowerPC application is implemented as a single-thread program. Without a real-time preemptive scheduler, all the cyclic tasks of the ROBIN PowerPC application are performed in a single loop, although these tasks have different priorities and different real-time performance requirements. In order to improve the overall processing rate of the single main loop, a number of application-specific measures have been proposed to organize operations in the loop optimally. The goal of the optimization is to distribute the operations uniformly into each cycle of the main loop, to minimize the maximum processing time of one main-loop cycle. This is because the processing rate of the main loop reflects the cycle rate of the most time-critical task in the ROBIN PowerPC system.

MontaVista RT-Linux is chosen as the real-time operating system (RTOS) for the OS-based ROBIN PowerPC system. With the existence of an RTOS, user tasks are prioritized and scheduled automatically by a preemptive real-time multi-tasking scheduler. One tradeoff for this convenience is an extra computational cost for the scheduling or task switching. In order to reduce this cost several measures are proposed to improve the performance of the system by reducing the number of user tasks and making an optimal choice for the cycle time of the tasks.

Performances of the two implementations above of the ROBIN PowerPC system

are measured through elaborate experiments in a simulated ROS/ROBIN testing environment. The performance of the standalone non-OS based PowerPC system is slightly above the baseline requirement of the ATLAS DAQ chain, while the performance of the RTLinux-based ROBIN PowerPC system does not meet the related ATLAS DAQ baseline requirement. According to the analysis of the CPU time consumption, over two thirds of the computational time for the RTLinux-based system is devoted to the task scheduling or switching. Apparently, to make the OS-based ROBIN PowerPC system work, a higher-performance real-time scheduler is required.

However, the attempt with the OS-based PowerPC system tells that there would be a chance for an OS-based ROBIN PowerPC system to meet the ATLAS DAQ baseline requirement, if an upgraded real-time operating system with a real-time scheduler of higher performance, would emerge in the near future. For a complex multi-tasking application an OS-based system architecture is always the tendency. Despite an extra cost of memory space for the OS kernel and an extra computational cost for the scheduling, the introduction of a real-time operating system into a multi-tasking application saves by far more efforts for the software development.

A Glossary

ALICE	A Large Ion Collider Experiment (Detector)
ASIC	Application-Specific Integrated Circuit
ATLAS	A Toroidal LHC ApparatuS (Detector)
AUX	auxiliary
BIOS	Basic Input Output System
BIST	Build-In-Self-Test
BSP	Board Support Packages
CMS	Compact Muon Solenoid (Detector)
CERN	European Particle Research Laboratory
COTS	Commercial "Off-The-Shelf"
CPU	Central Processing Unit
DAQ	ATLAS Data Acquisition Chain
DDR RAM	Double Data Rate RAM
DDR SDRAM	Double Data Rate Synchronous Dynamic RAM
DF	Data Flow
DMA	Direct Memory Access
DSP	Digital Signal Processor
EB	Event Builder
EBC	External Bus Controller
EEPROM	Electrically Erasable Programmable ROM
EF	Event Filter
FIFO	First-In-First-Out
FSM	Finite State Machine
FPGA	Field Programmable Logic Array
FPL	Free-Page List
GDB	GNU Debugger
GE	Gigabit Ethernet

GNU	GNU Not Unix
GPIO	General Purpose IO
GPL	General Public License
IC	Integrated Circuit
ICE	In-Circuit Emulator
IO	Input/Output
ISR	Interrupt Service Routine
JTAG	Joint Test Action Group
LED	Light-Emitting Diode
LEIR	Low-Energy Injector Ring
LDC	Link Destination Card (SLink)
LEP	Large Electron/Positron Collider
LHC	Large Hadron Collider
LHCb	LHC-beauty (Detector)
LIFO	Last-In-First-Out
LILO	LInux LOader
LRU	Least-Recently-Used
LSC	Link Source Card (SLink)
MAC	Media Access Control
MBR	Master Boot Record
MMU	Memory Management Unit
OS	Operating System
PCI	Peripheral Component Interconnect (Local Bus)
PCI-E	PCI-Express
PLD	Programmable Logic Device
PSB	Proton Synchrotron Booster
PSR	Proton Synchrotron Ring
RISC	Reduced Instruction Set Computer
RTAI	Real-Time Application Interface
RTHAL	Real-Time Hardware Abstraction Layer
RT-Linux	Real-Time Linux
RTOS	Real-Time Operating System
ROB	ATLAS ReadOut Buffer
ROBIN	ATLAS ReadOut-Buffer INput
ROD	ATLAS ReadOut Driver
RoI	Region of Interest
ROS	ATLAS ReadOut-Subsystem
ROL	ATLAS ReadOut-Link

RR	Round Robin
SDRAM	Synchronous Dynamic Random Access Memory
SM	Standard Model
SoC	System on a Chip
SPD	Serial Presence Detect (Memory)
SPS	Super Proton Synchrotron
TDAQ	ATLAS Trigger and Data Acquisition Chain
TeV	Tera Electron Volt
TLB	Translation Lookaside Buffer
UART	Universal Asynchronous Receiver/Transmitter
USB	Universal Serial Bus
VME	Versa Module Europa

B Software Development Platform for the PowerPC System

B.1 Cross-Development Environment

An embedded system runs usually on a special target platform, for example, a microcontroller that runs with a minimal amount of memory for its own purpose. On such a platform it is inconvenient or impossible for its software developers to develop their applications or compile their code directly above. Therefore, it is common that the embedded software is developed on another platform, such as an x86 desktop PC. Accordingly, a special toolchain is required for the cross development. The toolchain must be capable of creating executable code for the target platform other than the one on which the toolchain runs. This process is referred as cross compiling, and the special toolchain is termed as cross compiler.

The software development for the ROBIN PowerPC system requires also a cross-development environment and a cross compiler or toolchain. The toolchain executes on an X86 platform, but generates binary code for a PowerPC platform. The U-Boot program, the real-time Linux OS as well as the application for the ROBIN PowerPC system are all integrated with the cross-platform toolchain.

The GNU toolchain is chosen for the software development for the target ROBIN PowerPC system. The GNU toolchain is composed of a GNU C/C++ compiler, a GDB debugger, an assembler, a linker and other binutils. Figure B.1 shows the cross development environment for the ROBIN PowerPC system.

Furthermore, nowadays the increasing complexity of software and hardware design leads to new approaches for debugging. Silicon manufacturers offer also increasing

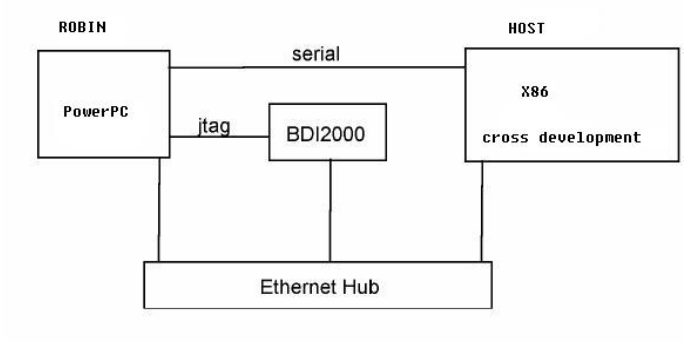

Figure B.1: **Hardware deployment of the cross development environment of the PowerPC system.**

on-chip debugging features to assist embedded software developers to debug their code.

Joint Test Action Group (JTAG), implemented for various processors, follows the IEEE 1149.1 standard which entitles standard test access ports for testing. With JTAG debug port, one can control and monitor the microcontroller solely through the stable on-chip debugging services. This debugging mode keeps running even when the target system crashes, which enables developers to continue investigating the cause of the crash. This is a significant advantage over generic software debuggers. Moreover, JTAG debugger is also relatively cheaper and more general purpose than e.g. an in-circuit emulator(ICE).

For the software debugging on the ROBIN PowerPC system, Abatron BDI2000 JTAG Debugger is used. The BDI2000 can be used for many types of processors, including CPU12/16/32, PowerPC, ColdFire, M-CORE, MIPS, XScale, ARM, etc. The BDI2000 sets up a communication path between the development PC and the target ROBIN board via RS232 or 10 BASE-T Ethernet. The BDI2000 converts the debug commands automatically into appropriate JTAG sequences, which are transferred to the target ROBIN board via a JTAG port. The use of a JTAG interface occupies no system resource on the target system, i.e. in this case the PowerPC system.

B.2 U-BOOT for the ROBIN PowerPC System

U-Boot is the abbreviation for das U-Boot (Universal Boot Loader). It is a prevailing boot loader implementation specially for embedded systems. It supports a number of different computer architectures, including PPC, ARM, MIPS, x86, m68k, Nios,

PowerPC, etc. It is released under the GNU General Public License (GPL) and takes advantage of an open development process. U-Boot is usually built on an x86 PC for any supported architecture using a cross development environment. Because the U-Boot provides strong supports for PowerPC architectures, it is chosen as the boot loader for the ROBIN PowerPC system.

In this section some basic concepts about bootstrap loader are firstly introduced. Then the specific features of the U-Boot is given. At last the U-Boot adaptions and extensions specially for the ROBIN PowerPC system are presented.

B.2.1 Bootstrap Loader

Any computer system, both for a personal computer and for an embedded chip mounted on a car, an aircraft, a robot, or a toy, can only execute codes that already exist in the memory, such as Read-Only Memory (ROM) or Random Access Memory (RAM). However, an operating system or a single application-specific program are often stored on non-volatile storage devices, e.g. hard disks, USB disks, or CD drivers. Therefore, a bridge solution needs to be developed to load the target operating system or application-specific program into the memory and then to trigger their start. A bootstrap loader is commonly applied to accomplish this task.

This section reviews the basic concepts of bootstrap loaders, their principal tasks in embedded systems and the usual implementation hierarchy of a bootstrap loader. reviewed in this section.

B.2.1.1 Definition

A bootstrap loader is also referred to as a boot loader or boot monitor. Its goal is to load the image of the final target software into the memory and run it on the machine. The target software can be an embedded operating system or a single application-specific program.

On a desktop PC, with Linux for example, LILO is commonly used as the OS boot loader, which resides on the master boot record (MBR) of the hard drive. When the PC is powered on, the BIOS performs firstly various system initializations and then executes the boot loader located in the MBR. The boot loader then passes system information to the kernel and then executes the kernel. For instance, the boot loader tells the kernel which hard drive partition to mount as root.

However, in an embedded system the role of the boot loader is more complicated since these systems do not have a BIOS to perform the initial system configuration. The low-level initializations of microprocessors, memory controllers and other board-specific hardware vary from board to board and CPU to CPU. All these

initializations must also be conducted by the boot loader, besides the loading of softwares from disk drives to the memory.

B.2.1.2 Principal Tasks

Generally the boot loader for an embedded system must provide at a minimum the following functions, including

- initializing the hardware, especially the CPU, the memory controller, and the flash memory,
- providing boot parameters for the target software or the operating system kernel if there is one,
- and starting the target software or the OS kernel.

Additionally, most boot loaders provide also more convenient features to simplify the programs developed above them. The features are listed as follows:

- reading and writing arbitrary memory locations,
- uploading new binary images to the board's RAM via a serial line or Ethernet,
- and flash functions, like copying the binary images from RAM to flash memory.

B.2.1.3 Implementation Hierarchy

A boot loader may be implemented in multiple stages. Several small programs summon one another sequentially, until the last of them loads the entire target software. The first stage is usually designed in a most convenient and simplest way; and only on the final stage the boot loader eventually transfers control to the target software. The name of bootstrap loader comes just from the one-by-one steps of program-loading process.

Because the implementation of a boot loader depends tightly upon the individual hardware platform, it is impossible to build a universal boot loader for the immerse embedded world. But generally most boot loaders consist of two major stages in general. The code for hardware-related initializations is put in stage 1, which is usually implemented with assembler language and is compressed. Stage 2 is usually implemented with C language, which supports more complex functionalities and has better readability and portability.

Basically stage 1 includes the following steps:

- initialize the hardware devices,

- assign a RAM space for the codes of stage 2,

- copy the codes of stage 2 to the RAM space,

- initialize memory stack,

- and go to the code entry of stage 2.

Stage 2 includes the following steps:

- initialize the related hardware devices used in this stage,

- check the system memory map,

- load the kernel map and root file system map from flash to the RAM space,

- initialize the kernel parameters,

- and call the kernel.

B.2.2 Features of U-Boot

It has been introduced in the previous section that a boot loader is a small piece of software that executes soon after powering up a computer. Its goal is to load the image of the final target software into the memory and run it on the machine. Das U-Boot, commonly used in embedded systems as the boot loader, is intended to provide a common, flexible and easily extensible boot program for embedded devices.

In the embedded world it is very important to provide a flexible way to configure the system environment. Accordingly, one primary goal of U-Boot is to achieve the flexibility. Developers must be able to decide which components are really needed within the actual target system. Besides, automatic detection of hardware components at runtime is also an important feature. For example, automatic detection of the CPU type, size of SPD memory or size of flash memory allows the extension of hardware without changing the application code.

B.2.3 U-Boot Adaption

In order to port the U-Boot source code to the ROBIN board, it is necessary to add some specific code to the U-Boot source code. This process is like the Board Support Packages(BSP) development in the Linux kernel. For the ROBIN board the following modifications and adaption are made.

- Read values of the I2C bus, set the CPU speed, and enable instruction/data caches;
- set up the stack pointer;
- initialize the interrupt controller;
- initialize the DDR memory controller;
- initialize UART and set the baudrate;
- initialize External Bus and build one common address system for PowerPC's RAM and FPGA's FIFOs;
- initialize the FLASH layout and programming;
- initialize other devices, such as Ethernet;
- set up boot parameter area and construct parameter structures. (Note, boot parameters are used by the OS kernel to identify the root device, page size, memory size, etc.)

B.2.4 U-Boot Extensions

The PowerPC part acts as the auxiliary core in the ROBIN system. It is also responsible for self-testing the whole board, as well as monitoring the configuration. Hence the following four add-ons are attached extra to the standard U-Boot program:

- memory test utility,
- more APIs, like the flash reading/writing/clearing,
- second UART support for ROBIN,
- and FPGA configuration through the GPIO(General Purpose IO).

C MontaVista RT-Linux Configurations for the ROBIN PowerPC System

This chapter presents details about how the MontaVista real-time Linux OS is configured and adapted to the ROBIN PowerPC system.

C.1 Boot Sequence

The embedded Linux boot sequence is more complicated than a proprietary embedded operating system, and there are many more options to configure it. In general, the boot sequence goes as follows:[47]

- After Power-On or reset, processor branches to the U-Boot startup code.
- The U-Boot initializes the CPU and memory, Flash, performs only minimal initialization of on-chip devices, such as the console serial port to provide boot diagnostic messages.
- U-Boot also sets up the memory map for the kernel to use in a format that is consistent across platforms.
- The U-boot decompresses the Linux kernel from flash into RAM, and jumps to it.
- The Linux kernel sets up the caches, initializes each of the hardware devices via the init function in each driver, decompress the Initial RAM disk (initrd) into ram, mounts the root file system (including busybox) and executes the

init process, which is the ultimate parent of all user mode processes, typically /sbin/initd.

- Executing the first program linked against the shared C runtime library (often the init function) causes the shared runtime library to be loaded.
- In a typical Linux system, the init function reads /etc/inittab to execute the appropriate run control script from /etc/rc.d, which executes the start scripts to initialize networking and other system services. In the ROBIN PowerPC system the init function is replaced with a C program to start the PowerPC application programs. [41]

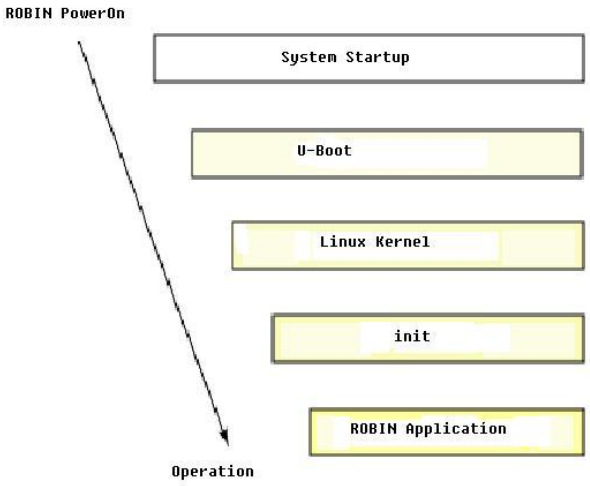

Figure C.1: **The ROBIN PowerPC Startup Process**

C.2 Linux Kernel Adaptions

In order to run the MontaVista Linux on the ROBIN PowerPC system, a development host PC is connected to the target ROBIN board. The host PC must have a normal desktop Linux. A GNU cross development environment is set up on the host PC, and the embedded Linux sourcecode of MontaVista for PowerPC is installed.

The Linux kernel sourcecode is divided into two parts: the architecture-specific part and the architecture-independent part. The architecture-specific part executes firstly. It sets up hardware registers, configures the memory map, performs

architecture-specific initialization, and then transfers control to the architecture-independent part of the kernel. During the second phase the rest of the system is initialized.

The directory arch/ under the kernel tree consists of different subdirectories, each for a different architecture (ARM, i386, PPC, and so on). Each of these subdirectories includes kernel/ and mm/ subdirectories, which contain architecture-specific code to do things like initializing memory, setting up IRQs, enabling cache, setting up kernel page tables, and so on. These functions are called, once the kernel is loaded and given control; then the rest of the system is initialized.

The MontaVista Linux sourcecode has already provided good support for PowerPC 440GP that is used in ROBIN. When porting MontaVista to the ROBIN PowerPC system, only a few ROBIN-specific adaptions need to be made for the target system, including memory alignment, serial port and baudrate, network driver, and Flash Memory.

The kernel is then compiled as a vmlinux image file for the ROBIN board. After the compiling, the U-Boot communicates with the host using the serial or ethernet port to transfer the kernel into the PowerPC's flash, together with the Initial RAM disk (initrd). In the boot process, the kernel is loaded into the PowerPC's memory by the U-Boot. After the kernel is fully loaded, the U-Boot passes control to the address where the kernel was loaded. The kernel then decompresses the Initial RAM disk (initrd) into ram, mounts the root file system and executes the init process. Section ?? gives more details.

C.3 Ramdisk

The purpose of the Initial RAM disk (initrd) image is to provide a root file system for the Linux kernel when it boots.[42] In a normal Linux, the ramdisk is only a temporary root file system that is mounted during system boot to support the two-state boot process. The initrd contains various executables and drivers that permit the real root file system to be mounted; afterwards the initrd RAM disk is unmounted and its memory freed. But in many embedded Linux systems, the initrd is just the final root file system. This implies that the file system contains a number of things, including file system structure(/bin, /dev, /etc, /lib, /proc ...), binaries (such as busybox), configuration files (such as rc.sysinit), device entries (/dev/kmem, etc.), proprietary applications and the frequently used runtime libraries.

To create an initrd for the PowerPC system, begin by creating an empty file, using /dev/zero (a stream of zeroes) as input writing to the ramdisk.img file. The file size

is normally several megabytes. Then use the mke2fs command to create an ext2 (second extended) file system using this empty file. Now that this file is an ext2 file system, mount the file to a directory as a loop device. [42]

The next step is to create the necessary subdirectories that make up the root file system: /bin, /sys, /dev, and /proc. To make this root file system useful, BusyBox is used in ROBIN PowerPC. Busybox is a single image that contains many individual utilities commonly found in Linux systems. Refer to section **??** for more details.

Then it is the creation of a small number of special device files. Copy these directly from current desktop /dev subdirectory, using the -a option (archive) to preserve their attributes.

The penultimate step is to generate the linuxrc file. After the kernel mounts the RAM disk, it searches for an init file to execute. If an init file is not found, the kernel invokes the linuxrc file as its startup script. The basic setup of the environment is in this file, such as mounting the /proc, /sys file system. Then ash(a Bourne Shell clone) is invoked, so that an interact console is ready. The linuxrc file is thus made executable using chmod.

Finally, the root file system is complete. It is unmounted and then compressed using gzip. The resulting file (ramdisk.img.gz) is copied to the flash so that it can be loaded by linux kernel.

C.4 Busybox

BusyBox is a software application which provides many standard Unix tools, much like the larger (but more capable) GNU Core Utilities. BusyBox is designed to be a small executable program for the use with Linux, which makes it ideal for special purpose Linux distributions and embedded devices. It has been called “The Swiss Army Knife of Embedded Linux”. [2]

BusyBox is a single image that contains many individual utilities commonly found in Linux systems (such as ash, awk, sed, insmod, and so on). The advantage of BusyBox is that it packs many utilities into one package while sharing their common elements, which results in a much smaller image. This is ideal for the ROBIN PowerPC embedded system. Copy the BusyBox image from its source directory into the /bin directory. A number of symbolic links are then created, all of which point to the BusyBox utilities. BusyBox figures out which utility was invoked and performs the according functionality.

C.5 C Library

In the embedded system, if a customized application binary needs the standard C library, there is an option beyond the massive glibc. It is the uClibc, which is a minimized version of the standard C library for space-constrained systems. If the uClibc is used, the binaries are needed to be recompiled with these libraries, hence some additional work is required. However, in the ROBIN PowerPC system, the BusyBox image is statically linked so that no libraries are required. uClibc was tested but not used.

List of Figures

List of Tables

Bibliography

[1] *Atlas Modelling Web Page [online].* http://www.nikhef.nl/pub/experiments/atlas/daq/modelling.html.

[2] *BusyBox: The Swiss Army Knife of Embedded Linux.* http://www.busybox.net/about.html.

[3] *CES - Creative Electronic Systems [online].* http://www.ces.ch.

[4] *HOLA S-Link documentation.* http://hsi.web.cern.ch/HSI/s-link/devices/hola.

[5] *RS Components, Electronic Components [online].* http://www.rs-components.com.

[6] *Technical Design Report.* The BaBar Collaboration, SLAC, 1995.

[7] *ATLAS Detector and Physics Performance, Technical Design Report.* The Atlas Collaboration, CERN/LHCC 99-14, CERN, May 1999.

[8] *The CDF IIb Detector Ű Technical Design Report.* The CDF Run IIb Collaboration, Fermilab, 2002.

[9] *CMS - The TriDAS Project Technical Design Report, Volume 2: Data Acquisition and High-Level Trigger.* The CMS Collaboration, CERN, 2002.

[10] *D0 Run IIb Upgrade Technical Design Report.* The D0 Collaboration, FERMILAB-PUB-02/327-E, Fermilab, 2002.

[11] *Data Acquisition and Experiment Control - Technical Design Report.* LHCb Collaboration, CERN, 2002.

[12] *ATLAS High-Level Trigger, Data Acquisition and Controls Technical Design Report.* ATLAS HLT/DAQ/DCS Group, 2003.

[13] *ALICE - Technical Design Report of the Trigger, Data Acquisition, High-Level Trigger, and Control System.* The ALICE Collaboration, CERN-LHCC-2003-062, CERN, 2004.

[14] *LHC - The Large Hadron Collider.* http://lhc.web.cern.ch/lhc/, 2007.

[15] Abolins, M. and et al.: *Specification of the LVL1/LVL2 trigger interface.* CERN ATL-DAQ-99-015, year = 1999.

[16] Baranovski, A., C. Brock, D. Bonham, Lauri Loebel-Carpenter, Lee Lueking, Wyatt Merritt, Carmenita Moore, Igor Terekhov, J. Trumbo, Sinisa Veseli, J. Weigand, Steve White and K. Yip: *D0 Data Handling Operational Experience.* Computing Research Repository, cs.DC/0306114, 2003.

[17] Bird, T.: *Comparing two approaches to real-time Linux. Guest column at Linuxdevices.com.* http://www.linuxdevices.com/articles/AT7005360270.html, 12 2002.

[18] Bock, R., J. A. Bogaerts, P. Werner, A. Kugel, R. Maenner and M. Mueller: *Active Rob Complex: An SMP-PC and FPGA based solution for the Atlas ReadoutSystem.* In *IEEE Realtime Conference*, pages 199–203, Valencia, 2001.

[19] Boorman, G., P. Clarke, R. Cranfield, G. Crone, B. Green and J. Strong: *The UK ROB-in a Prototype ATLAS Readout Buffer Input Module.* CERN ATLAS Note, CERN, May 2000.

[20] Boterenbrood, H., P. Jansweijer, G.Kieft, R. Scholte, R. Slopsema and J. Vermeulen: *A SHARC based ROB Complex : design and measurement results.* CERN ATLAS Note ATL-DAQ-2000-021, CERN, May 2000.

[21] Boyle, O., R. McLaren and E. van der Bij: *The S-LINK Interface Specification.* Technical report, CERN, 1997.

[22] Brosch, O.: *A Kaon Trigger for FOPI.* PhD thesis, Ruprecht-Karls University of Heidelberg, Heidelberg, Germany, May 2004.

[23] Calvet, D., O. Gachelin, M. Huet and I. Mandjavidze: *A Scheme of Read-Out Organization for the ATLAS High-Level Triggers and DAQ based on ROB Complexes.* CERN ATLAS Note ATL-DAQ-2000-014, CERN, 2000.

[24] Cheng, T.P. and L.F. Li: *Gauge theory of elementary particle physics.* Oxford University Press, 1995.

[25] Crone, G., D. Francis, M. Joos, J.Petersen and S. Veneziano: *Read-Out Buffer in DAQ/EF prototype -1.* CERN ATLAS Note ATL-DAQ-2000-053, CERN, 2000.

[26] Dankwardt, K.: *Real-Time and Linux, Part 1.* Embedded Linux Journal, 1 2002.

[27] *The DIAPM RTAI project Homepage.* http://www.rtai.org.

[28] Francis, D., M. Mueller, L. Tremblet and J. Vermeulen: *Summary of ROS system tests.* http://agenda.cern.ch/askArchive.php?base=agenda&categ=a02164&id=a02164s5t2/transparencies.

[29] Fruehwirth, R., M. Regler, R.K. Bock, H. Grote and D. Notz: *Data Analysis Techniques for High-Energy Physics.* Cambridge Monographs on Particle Physics, Nuclear Physics and Cosmology, 2000.

[30] FSMLabs, Inc. Homepage. http://www.fsmlabs.com.

[31] Gorini, B., M. Joos, J. Petersen, A. Kugel, R. Maenner, M. Mueller, M. Yu, B. Green and G. Kieft: *A RobIn Prototype for a PCI-Bus Based Atlas Readout-System.* In *The 9th Workshop on Electronics for LHC Experiments*, pages 152–156, Amsterdam, Netherland, 2003.

[32] Green, B., G. Kieft and A. Kugel: *ATLAS TDAQ/DCS ROS Prototype-ROBIN Software Interface.* CERN EDMA Note ATL-DQ-EN-003, CERN, 9 2002.

[33] Griffiths, D.J.: *Introduction to Elementary Particles.* Wiley, John & Sons, Inc., 1987.

[34] Group, LHC Study: *The Large Hadron Collider, Conceptual Design Report.* CERN/AC 95-05, 1995.

[35] Group, Particle Data: *Review of Particle Physics.* The European Physical Journal C3, 1998.

[36] Hezel, S.: *FPGA-basiertes Template-Matching mit Distanztransformierten Bildern*. PhD thesis, University of Mannheim, Germany, 2004.

[37] Huet, M.: *An UML description of the ATLAS ROB viewed from the Level-2 Trigger*. CERN ATLAS Note.

[38] IBM: *PPC440GP Embedded Processor User Manual*.

[39] Iwanski, W. and E. van der Bij: *32-bit S-LINK to 64-bit PCI interface - Users Guide, CERN*. https://edms.cern.ch/file/249657/1/userguide.PDF, February 2002.

[40] Jarlskog, G. and D. Rein (editors): *Proceedings of the Large Hadron Colider Workshop, CERN 90-10 / ECFA 90-133*, Aachen, Germany, 1990.

[41] Jones, M. Tim: *Inside the Linux boot process*. http://www.ibm.com/developerworks/linux/library/l-linuxboot/, 2006.

[42] Jones, M. Tim: *Linux initial RAM disk (initrd) overview*. http://www.ibm.com/developerworks/linux/library/l-initrd.html, 2006.

[43] Kane, G.L.: *Modern Elementary Particle Physics*. Perseus Books, 1987.

[44] Kugel, A.: *The ATLAS ROBIN - A High-Performance Data-Acquisition Module*. PhD thesis, University of Mannheim, Germany, 2009.

[45] Kugel, A., R. Maenner, M. Mueller, M. Yu, E. Krause, B. Gorini, M. Joos, J. Petersen, S. Stancu, B. Green, A. Misiejuk, G. Kieft and J. van Wasen: *The Final Design of the ATLAS Trigger/DAQ Readout-Buffer Input (ROBIN) Device*. 2005.

[46] Lienhart, G.: *Beschleunigung Hydrodynamischer Astrophysikalischer Simulationen mit FPGA-Basierten Rekonfigurierbaren Koprozessoren*. PhD thesis, Ruprecht-Karls University of Heidelberg, Heidelberg, Germany, 2004.

[47] LinuxLink: *The Linux Startup Process*. https://linuxlink.timesys.com/docs/startup_overview, 2006.

[48] Litvintsev, D. O.: *The CDF Data Handling System*. In *Conference for Computing in High Energy and Nuclear Physics*, La Jolla, California, 2003.

[49] Llewellyn-Smith, C.H.: *Particle Physics in the Future*. In *The Perkins Conference*, Oxford, England, 1993.

[50] Mao, D. and X. Hu: *LINUX Kernel Source Code Scenario Analysis (in Chinese).* Zhejing University Publisher, 2001.

[51] MontaVista: *Platform Support for MontaVista Linux.* http://www.montavista.co.jp/products/boards.html.

[52] Mornacchi, G.: *Architecture, deferrals and costing. ATLAS Week Presentation.* http://agenda.cern.ch/askArchive.php?base=agenda&categ=a03192&id=a03192s0t4/transparencies, 2003.

[53] Mueller, M.: *Evaluation of an FPGA and PCI Bus based Readout Buffer for the Atlas Experiment.* PhD thesis, University of Mannheim, Germany, 2004.

[54] Mueller, M.: *ATLAS ROBIN PCI Communication Interface Description.* Technical Report, CERN, 2005.

[55] Novaes, S.F.: *Standard model: An introduction.* In *10th Jorge Andre Swieca Summer School: Particle and Fields*, pages hep–ph/0001283, Sao Paulo, Brazil, 1999.

[56] PCIMG, PCI Industrial Computers: *CompactPCI Specification.* Technical report, 1997.

[57] Perkins, D.H.: *Introduction to High Energy Physics.* Addisen-Wesley, 1987.

[58] Qu, Zhenxin, Qingwei Zeng and Bo Han: *Design and Implementation of Embedded Real-Time Operating System Kernel (in Chinese).* Computer Engineering and Application, 2001.

[59] Sessler, M.: *Algorithms on CPUs and FPGAs for the ATLAS LVL2 Trigger.* PhD thesis, Ruprecht-Karls University of Heidelberg, Germany, 2000.

[60] Stallings, William: *Operating System - Kernel and Design Theory ??? (Version IV).* ???, 2001.

[61] Stankovic, J.A. and K. Ramamritham: *What is Predictability for Real-Time Systems?* Journal of Real-Time Systems, (2):247–254, 1990.

[62] Storer, J.A.: *An Introduction to Data Structures and Algorithms.* Springer, 2002.

[63] Veneziano, S.: *The Read-Out Crate in ATLAS DAQ/EF prototype.* In *CHEP*, 2000.

[64] Vermeulen, J.C. and et al: *The Baseline DataFlow Systems of the ATLAS Trigger and DAQ*. In *9th Workshop on Electronics for LHC Experiments in Amsterdam*, pages 147–151, 2003.

[65] Wang, C.L., B. Yang, Y. Yang and Z. Zhu: *A Survey of Embedded Operating System*. http://www.cs.ucsd.edu/classes/fa01/cse221/projects/group2.pdf, 2001.

[66] Wolf, T.: *Die Systemsoftware fuer den First Level Trigger des HERA-B Experiments*. PhD thesis, University of Mannheim, Germany, 1998.