

Control Software for Reconfigurable Coprocessors

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

vorgelegt von

Dipl.-Physiker Christian Hinkelbein
aus Mannheim

Mannheim, 2005

Dekan: Professor Dr. Matthias Krause, Universität Mannheim
Referent: Professor Dr. Reinhard Männer, Universität Mannheim
Korreferent: Professor Dr. Volker Lindenstruth, Universität Heidelberg

Tag der mündlichen Prüfung: 21. Oktober 2005

Control Software for Reconfigurable Coprocessors

Christian Hinkelbein
University of Mannheim
B6,23-29, D-68131 Mannheim, Germany
hinkelbein@ti.uni-mannheim.de

24th November 2005

Für meine Kinder.

Abstract

On-line data processing at the ATLAS general purpose particle detector, which is currently under construction at Geneva, generates demands on computing power that are difficult to satisfy with commodity CPU-based computers. One of the most demanding applications is the recognition of particle tracks that originate from B-quark decays. However, this and many others applications can benefit from parallel execution on field programmable gate arrays (FPGA). After the demonstration of accelerated track recognition with big FPGA-based custom computers, the development of FPGA based coprocessors started in the late 1990's. Applications of FPGA coprocessors are usually partitioned between the host and the tightly coupled coprocessor. The objective of the research that I present in this thesis was the development of software that mediates to applications the access to FPGA coprocessors. I used a software process based on iterative prototyping to cope with the expected changing requirements. Also, I used a strict bottom-up design to create classes that model devices on the coprocessors. Using these low-level classes, I developed tools which were used for bootstrapping, debugging, and firmware update of the coprocessors during their development and maintenance. Measurements show that the software overhead introduced by object-oriented programming and software layering is small. The software-support for six different coprocessors was partitioned into corresponding independent packages, which reuse a set of packages that provide common and basic functions. The steady evolution and use of the software during more than four years shows that the software is maintainable, adaptable, and usable.

Zusammenfassung

Die on-line Datenverarbeitung am ATLAS Teilchendetektor, der zurzeit bei Genf entsteht, hat einen Bedarf an Rechenkapazität, die mit herkömmlichen CPU - basierten Rechnern schwierig zu befriedigen ist. Eine der herausforderndsten Anwendungen ist hier die Erkennung von Teilchenspuren, die vom Zerfall von B-Quarks herrühren. Jedoch kann diese und viele andere Anwendungen von der parallelen Ausführung auf Field Programmable Gate Arrays (FPGA's) profitieren. Nach der Demonstration von beschleunigter Spurerkennung mit großen FPGA - basierten Spezialrechnern begann an der Universität Mannheim in den späten 1990ern die Entwicklung von FPGA basierten Koprozessoren. Anwendungen solcher FPGA Koprozessoren sind üblicherweise zwischen dem Gastrechner und dem Koprozessor aufgeteilt. Das Ziel der hier präsentierten Arbeit war die Entwicklung von Software, die Anwendungen den Zugriff auf FPGA Koprozessoren vermittelt. Um den erwarteten wechselnden Anforderungen an die Software gerecht zu werden, habe ich einen Software Prozess gewählt, der auf iterativen Prototypen basiert. Ich habe ein striktes Bottom-Up Design benutzt, um Klassen zu entwerfen, die Komponenten des Koprozessors modellieren. Unter Benutzung dieser Klassen habe ich Werkzeuge entwickelt, die verwendet wurden um die Koprozessoren während ihrer Entwicklung und Wartung zu initialisieren, ihre Fehler zu beheben und ihre Konfigurationen zu aktualisieren. Messungen zeigen, dass der Software Overhead, der durch die objektorientierte Programmierung und die geschichtete Software Architektur bedingt ist, klein ist. Die Unterstützung der Software für sechs verschiedene Koprozessoren wurde in entsprechende Pakete gegliedert, die einen Satz von Paketen wiederverwenden welche gemeinsame Basisfunktionalität bereitstellen. Die stetige Entwicklung und Benutzung der Software über einen Zeitraum von mehr als vier Jahren zeigt, dass die Software wartbar, anpassbar und nützlich ist.

Contents

Contents	11
List of Figures	13
List of Tables	15
1 Introduction	17
1.1 Problem Description	17
1.2 Literature Review	19
1.3 Materials	25
1.4 Principal Results	28
1.5 Main Conclusions Suggested by the Results	31
2 Materials	35
2.1 Introduction	35
2.2 LHC, Atlas, Trigger	36
2.2.1 Large Hadron Collider and Associated Detectors	36
2.2.2 Physics and Tracking at the ATLAS Detector	37
2.2.3 ATLAS Trigger	44
2.2.4 S-Link	47
2.2.5 High Level Trigger Software, Prototypes and Testbeds	48
2.3 Computing and Reconfigurable Systems	53
2.3.1 Conventional Computing	53
2.3.2 Custom Computing	54
2.3.3 Reconfigurable Devices and Systems	55
2.3.4 Programming of Reconfigurable Systems	57
2.3.5 JTAG	58
2.4 FPGA Processors and Coprocessors Built in Mannheim	59
2.4.1 Enable++	60
2.4.2 μ Enable and μ Enable2	60
2.4.3 Atlantis, ACB and AIB	62
2.4.4 MPRACE	64
2.4.5 ROBIN	64
2.5 Host and Software Environment	66

2.5.1	Introduction	66
2.5.2	Host Architecture	66
2.5.3	IA32 and IA64	67
2.5.4	PCI, PCI-Bridges and Data Transfer	68
2.5.5	Operating Systems, Device Drivers and Tools	70
2.5.6	Software Development	71
3	Results	73
3.1	Introduction	73
3.2	Description of the Software	73
3.2.1	Software Process	73
3.2.2	Software Architecture	75
3.2.3	Software Behaviour	84
3.2.4	Evolution and Coprocessor Packages	85
3.2.5	Performance	91
3.2.6	Build System, Operating Systems and Platforms	97
3.3	Applications	103
3.3.1	μ Enable S-Link	103
3.3.2	Enable++ TRT Scan	104
3.3.3	Atlantis Bootstrapping	105
3.3.4	Atlantis and the Atlas Second Level Trigger Testbeds	106
3.3.5	Atlantis TRT Scan with T2REF	108
3.3.6	Parallel TRT Scan on 2 ACB's in Atlantis	110
3.3.7	MPRACE TRT Scan with T2REF	112
3.4	Reconfigurable Software	112
3.4.1	Introduction	112
3.4.2	Bottom-Up Design and Activity Flow	113
3.4.3	Collaboration	116
3.4.4	Service Abstraction	117
3.4.5	Dynamically Connected Devices	119
3.4.6	Performance Evaluation	123
3.4.7	Dynamic Loading of Components	124
4	Discussion and Conclusions	127
	Abbreviations	133
	Acknowledgments	135
	Bibliography	137

List of Figures

1.1	Environment of the Software	18
1.2	Host and FPGA Coprocessor in a System	27
2.1	Simulations of Events in the Inner Detector	41
2.2	Hough Transformation	42
2.3	Hough Histogram	43
2.4	Hough Histogram, Surface View	43
2.5	Trigger Architecture A	45
2.6	Trigger Architecture B	46
2.7	Trigger Architecture C	46
2.8	S-Link in TDAQ	47
2.9	HLT and DAQ	52
2.10	μ Enable	62
2.11	Atlantis Computing Board (ACB)	63
2.12	Data Paths and JTAG on the ACB	64
2.13	Atlantis Input Output (AIB)	65
2.14	MPRACE	65
2.15	ROBIN	66
2.16	Host Architecture	67
3.1	Iterative Prototyping with Risk Addressing	75
3.2	Mapping from Hardware to Software	77
3.3	BitStream Interface	78
3.4	JTAG Controllers Class Diagram	79
3.5	JTAG on ACB	80
3.6	EEPROM Controllers Class Diagram	81
3.7	Adaptation of the Parallel Port Interface	81
3.8	EEPROM on the Parallel Port, Source Code	82
3.9	Bridge and Driver Work Together	83
3.10	The Control Class Encapsulates the Internals	84
3.11	Allocation, Configuration, and Freeing of a Coprocessor	86
3.12	Configuration with JTAG, Collaboration Diagram	87
3.13	Evolution of Code Size	88

3.14	Code Size of Packages in the Library	89
3.15	Two Packages Supporting Different Coprocessors	90
3.16	Hierarchy of Control Classes	90
3.17	Processor Clock Interface	93
3.18	Method Call Overhead, Object Chain	94
3.19	Latency of Method Calls	95
3.20	Code-Size of Method Calls	96
3.21	DMA Latency	97
3.22	PIO Performance	98
3.23	Memcpy Performance	99
3.24	Driver Base Class	102
3.25	Driver Classes Hierarchy	102
3.26	Setup for S-Link Loop-Back	104
3.27	Transfer Rates with S-Link Loop-Back	104
3.28	Transfer Latency with S-Link Loop-Back	105
3.29	Vertical-Slice Prototype using E++ as FEX Processor	105
3.30	Integration of Architecture-A and -C using E++ as Coprocessor	106
3.31	Physical Integration of Atlantis in the Testbeds	107
3.32	Logical Integration of Atlantis in the Testbeds	107
3.33	TRT Scan with Atlantis, Activity Diagram	111
3.34	Architecture of a Reconfigurable Coprocessor	113
3.35	Software Representation of Hardware Components	114
3.36	Clock Setting, Sequence Diagram	115
3.37	Abstraction of the Device Representations	117
3.38	Abstract Services, Collaboration Diagram	118
3.39	Multiple Services	118
3.40	Device Base-Class and its Attributes	119
3.41	Class Diagram of Concrete Devices and Base Class	122

List of Tables

2.1	Trigger Rates and Latencies	44
2.2	Comparison between PLD's and FPGA's	56
2.3	JTAG Signals	59
2.4	Evolution of FPGA Processors and Coprocessors	61
3.1	Code Size of Packages in the Library	89
3.2	Partitioning of Code in the Coprocessor Packages	91
3.3	Timing Functions	92
3.4	Size of fundamental Types on IA32 and IA64	101
3.5	Performance Comparison	124

Chapter 1

Introduction

1.1 Problem Description

The work on field-programmable-gate-array-(FPGA)-accelerated real-time pattern recognition for the planned ATLAS ¹ experiment led to the development of the Enable++ FPGA-based custom computing machine (FCCM). Architectural changes in the ATLAS high level trigger (HLT) and the size and the complexity of the Enable++ machine were creating the demand for the development of smaller FPGA-based processors with tighter host coupling in the late 1990's.

The successor of Enable++, the Atlantis system, was chosen to use FPGA-based computing devices in cooperation with a CPU-based host system. The peripheral component interconnect (PCI) bus is the most common computer bus in commodity of the shelf (COTS) computing, and Atlantis uses PCI for the coupling between FPGA's and host, like many other PCI devices with FPGA's nowadays.

In the nomenclature of this work such PCI-coupled and FPGA-based computing devices are called *FPGA coprocessors*. Atlantis was designed to use two different FPGA coprocessors, one for the acceleration of computations, and one for interfacing to data-paths in the ATLAS high level trigger (HLT).

Moore's law predicts the doubling of resources on an integrated circuit (IC) every 18 months. Since FPGA technology follows Moore's law just like CPU technology does, it was foreseeable that FPGA coprocessors would have to be redesigned on a regular basis to benefit from their higher computing power (at least for some applications) compared with the ever increasing computing power of CPU based computers.

Concluding, the situation before the millennium was as follows: The big Enable++ machine was dated, but still in use, the commercial μ Enable coprocessors were used in some applications (S-Link ², image processing, and others), Atlantis was in the process of development, and a number of future FPGA coprocessors

¹ATLAS is a big particle detector experiment at the large hadron collider (LHC) at the European organisation for nuclear research (CERN)

²S-Link is a CERN standard for an unidirectional high speed data link

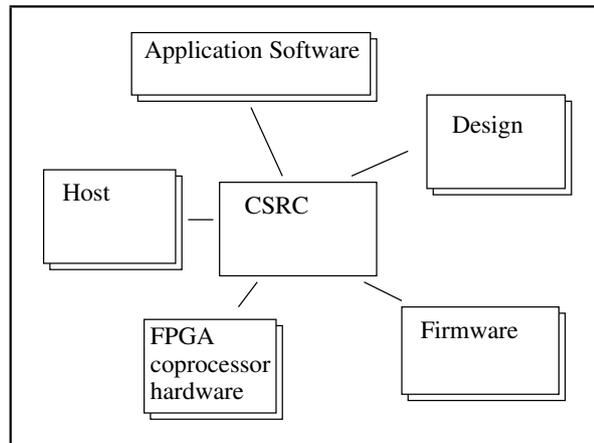


Figure 1.1: Environment of the Software
The stacked boxes indicate multiple instances.

were expected to come.

A system using FPGA coprocessors consists of several parts: Application software, host system (hardware and operating system software), FPGA coprocessor hardware, “slow” configuration data (firmware) for the coprocessor, “fast” configuration data (design) for the reconfigurable resources on the coprocessor, and software for interfacing and controlling the coprocessor. This work is dealing with the latter part, which I call *control software for reconfigurable coprocessors (CSRC)*. Figure 1.1 illustrates the components with which the CSRC interacts.

The ATLAS HLT is the main driving force behind the development of FPGA coprocessors and their applications at the Institute for Computer Science V at the University of Mannheim. These applications are the real-time pattern recognition in data from the ATLAS TRT detector, where the search for signatures of B-physics is creating high demands on computing power and throughput, and, more recently, the read-out buffer input (ROBIN). Enable++ was studied without a framework resembling the ATLAS HLT software or hardware until the late 1990’s. However, the approach of the LHC and ATLAS staging, at that time planned for the year 2005, lead to the creation of up two systems for carrying out studies on HLT design and performance. These were the software and testbed setup developed by the Saclay group, concentrating on the use of ATM networking technology, and the more generic reference software (T2REF) that was developed at CERN. The first testbeds for either setup were implemented in the late 1990’s, and it was time to integrate Enable++ and the upcoming Atlantis system into these testbeds. The ability to integrate with the HLT prototypes was generating requirements on the CSRC in terms of application interface, operating system compatibility, coding style, deployment, and others. The intended use of FPGA coprocessors for high-performance and soft real-time applications requires that the CSRC should not reduce system performance, and should not impair performance optimisation

efforts.

The purpose of this work is not only to create an interface between FPGA coprocessors and applications (e.g. in the ATLAS HLT) but also to facilitate development, bootstrapping, and debugging of the Atlantis FPGA coprocessors and their successors. This implies that the CSRC would have to provide low level access to controlled devices. Also, the software should be designed in a way to support the adaptation to future FPGA coprocessors.

Summary

This work addresses the problem of designing and implementing software for controlling FPGA based reconfigurable coprocessors. The software should provide an interface between FPGA coprocessors and different high-performance applications with an emphasis on applications for the ATLAS HLT. It should not restrict system performance and it should be open for optimisation efforts. The software should be self-contained and independent of operating systems and host architectures. The software should be designed with the evolution of hardware in mind, and it should support adaptation to future FPGA coprocessors. The software has to allow low level access to the controlled devices to support debugging and bootstrapping.

1.2 Literature Review

Reconfigurable Architectures

In 1998 S. Hauck reported on “The roles of FPGA’s in reprogrammable systems” [Hau98b]. He concludes that reprogrammable systems (RS) can provide significantly improved and even world-record like performance in generic reprogrammable architectures. RS’s are of a varied structure, using FPGA’s, memories, DSP’s, or even custom reprogrammable devices in systems ranging from one and two-chips to even hundreds or thousands of devices e.g. for logic simulation. The performance that is provided by such RS’s comes with the cost of some problems. Mapping software is slow and provides low quality compared to the requirements. Though FPGA’s represent economic solutions, they may create inefficiencies to system designers. Finally, Hauck states that much is left to be done. He proposes that reprogrammable functional units in standard processors and reprogrammable coprocessors could become the cornerstone of future computation systems.

In 2000 Schumacher et al [PSSL00] outweighs a software package that provides direct access to hardware components for testing, diagnostics or monitoring purposes. The software is implemented as a library of C++ classes. It provides low level access to hardware components and includes a graphical user interface. The software architecture is based on the separation of two class hierarchies; one for hardware objects such as FPGA’s, Memories, or Registers, and another hierarchy for hardware access, focusing on VME bus. The authors describe the software as “convenient to use, flexible and extensible” and report on successful use during

development of components for the pre-processor system of the ATLAS level-1 calorimeter trigger.

In 2001 H. Simmler [Sim01] described the realisation of preemptive multi-tasking for computing systems embedding FPGA based coprocessors. Besides a detailed problem analysis his work includes the implementation of operating system extensions supporting the reconfiguration of a commercial FPGA coprocessor upon OS-triggered process schedules. The analysis of the prototype performance is followed by design and implementation of a new FPGA based coprocessor that offers support for context schedules by hardware on the coprocessor itself.

Co-Synthesis

In 2002 M. Budiu, M. Mishra, A.R. Bharambe S.C. Goldstein published “Peer-to-peer hardware-software interfaces for reconfigurable fabrics” [BMBG02]. The authors propose a hardware-software interface between standard processors and reconfigurable devices (RD’s). They propose that the two devices act as equal peers, as opposed to the traditional view of the RD being a slave to the master processor. The authors state that based on that proposal, the RD is able to invoke software routines on the processor. The authors propose an interface like remote procedure calls between RD and processor and quantify the amount of computation that can be moved from the processor to the RD.

R. P. Dick and N. K. Jha [DJ98] reported 1998 on “CORDS: hardware-software co-synthesis of reconfigurable real-time distributed embedded systems”. The report emphasises the schedule of tasks executed on processing elements (PE), either processors or FPGA’s, in a real-time distributed system. The authors use an algorithm that bears similarities with simulated annealing and genetic algorithms for PE and communication resource allocation as well as task assignments. Experimental results are shown for estimated system costs for XC6000 and XC4000 based systems with a number of application examples. The results indicate that the greatly reduced reconfiguration delay of the XC6000 FPGA only seldom overweigh their much higher price.

OS-Support for Reconfigurable Systems

In 2003 H.Walder and M.Platzner [WP04] reported on a “Reconfigurable Hardware OS Prototype”. The authors approach the area of operating systems for reconfigurable hardware in a top-down manner. They started with a design concept followed by an implementation concept and finally case studies. The target systems for their work are composed from a CPU (PC) and an FPGA based board, both connected with a configuration and a data bus. Their approach partitions the reconfigurable resources of the FPGA into two parts, an OS frame that provides common services, and the user area, which accepts one or several *hardware tasks* (HWT). HWT’s communicate with the system through a standard task interface that is common to all HWT’s and the OS frame. HWT’s are designed to be relocat-

able, i.e. every HWT can fit into any of the *task slots* of the user area. The system provides a task scheduler, which executes on the CPU and reconfigures the user area according to the scheduler policy. Schedules can be triggered by components on the FPGA. In their prototype implementation scheduling relies on cooperative multitasking, i.e. the HWT's signal task completion as scheduling points to the system. Preemption is supported by the author's concept, however for this the HWT's need to fulfil further requirements to both their interface and internal operations. The prototype case study described by the authors executes some networking and multimedia tasks on an XESS XSV-800 board.

In 2003 C. Haubelt, D. Koch, and J. Teich reported on "Basic OS Support for Distributed Reconfigurable Hardware" [HKT03]. The target system of their work consists of reconfigurable nodes (ReCoNodes) and a network of connection links, which initially connects every node with every other. In the prototype implementation the nodes are made of Altera Excalibur development boards. The work aims at a distributed system which provides fault tolerance by hardware task migration and re-routing to compensate for resource errors. To achieve such the OS provides three basic features: *re-routing*, *repartitioning*, and *reconfiguration*. Re-routing is initiated by upon line error detection and done by the Altera NIOS soft-core processor implemented on all ReCoNodes, which also do the routing itself. Repartitioning is controlled by a task binding priority list which is common to all nodes. Task migration may be prepared by the copying of configuration data from one node to another. Reconfiguration is done by writing configuration data to a nodes configuration memory. Configuration data movement and writing is also done by the NIOS CPU's embedded in all nodes. The CPU issues a self reset on the node so that the node gets finally reconfigured.

JCOP and SCADA

"The LHC experiments' joint control project" [Mye99] describes the process of evaluation and selection of a distributed hardware/software system for acquisition of data relevant for the monitoring of an experiment, like gas pressure, power supply status, and temperature, and to control the experiment either automatically or manually upon operator intervention. It is assumed that the number of input data channels for an experiment like ATLAS is in the range of millions. The requirements for such control systems are similar in different experiments, and it is assumed that a commercial system would be the most cost-effective solution for CERN. [DS99b, DS99a] describe the architecture of a generic supervisory control and data acquisition system (SCADA) and describe measures for evaluating commercial SCADA systems.

Design Flow

S. Hauck published several reviews on reconfigurable systems in the late 90's. In 1996 he reported with A. Agarwal on "Software technologies for reconfigurable

systems” [HA96]. The report covers tools for mapping, placement, routing, partitioning and logic synthesis for single and multi-FPGA based reconfigurable systems. The authors argue that the requirements for the mapping software for multi-FPGA reconfigurable systems differ from those for standard hardware compilers because of constraints originating from the properties of FPGA interconnects. Additional constraints arise from the demand for fast compile times resulting from the need for short compile-test cycles. Scheduling concerns regarding runtime reconfiguration need to be taken into account. On the other hand, the fine-grain parallelism and low level programmability contrast to the fixed instruction set and sequential processing of standard processors. The authors conclude that there has been significant work on software tools optimised for reconfigurable systems but many issues remain in the development of a hardware and software reconfigurable system.

In 2001 S. Rühl described the programming of FPGA processors with active components [Rüh01]. An active component (AC) defines the interface of a functional unit along with methods to implement the unit. Different implementations may be provided for a single AC, e.g. one implementation coded in C++ that executes on the systems CPU, and one implementation defined as VHDL code to be executed on the FPGA. A programming framework developed on top of AC’s is used to compose an application using AC’s as building blocks. The framework ensures type compatibility between the interfaces of connected AC’s, possibly inserting AC’s as interface adapters. It implements the application using specific implementations from the different AC’s implementation libraries. The implementations are chosen so as to minimise implementation cost and maximise system performance.

In 2000 M. Eisenring and M. Platzner reported on “An Implementation Framework for Run-time Reconfigurable Systems” [EM00]. The framework provides a methodology and a design representation that allow to plug-in different design tools. Front-end tools cover design capture, temporal partitioning and scheduling; back-end tools provide reconfiguration control, communication channel generation, estimation, and the final code composition. The report discusses two of the framework’s main issues, the design representation and the hierarchical reconfiguration approach for multi-FPGA systems.

Reconfigurable Real-Time Software

In 1996 D. B. Stewart and P. D. Khosla published “ The Chimera methodology: designing dynamically reconfigurable and reusable real-time software using port-based objects” [SK96] (this publication was followed in 1997 by a similar one [SVK97]). The authors justify a system for rapid development of real-time applications through the use of reusable and dynamically configurable software. The target system is a distributed memory computing environment. The primary contribution of their research is the port-based object model of a real-time software component. The model is obtained by applying the port-automaton formal compu-

tational theory to object-based design. A finite state machine (FSM), detailed interface specifications, and a C-language template are used to define the port-based object. The FSM is used to cover the requirements for communication between the components that arise from the real-time nature of the system. In the Chimera Methodology links between objects are established by connecting input ports to output ports using port names as port identifiers. The ports are not implemented as message queues, because the model assumes that according to the real-time nature there are always most recent and actual data present at the inputs. The authors mention that their model is not usable for high volume data as would occur e.g. in imaging applications. They select C instead of C++ as specification language for port-based objects, since most programmers are assumed to be engineers instead of computer scientists. A detail of their work covers “reconfigurable device drivers” which the authors describe as a port-based object that communicate with hardware through the object’s resource ports (The authors differentiate between input and output ports, which are used to integrate the objects into a system, and resource ports which are used for communication with external systems). The resource ports are supposed to be configured at system start-up, and the objects are to be configured e.g. by reading configuration data from the external hardware that is to be controlled. The authors cover techniques for verifying correctness and analysing of the system performance and provide tools for integrating software using the port-based object model.

In 2002 S. Wang and K.G. Shin published the article “Constructing Reconfigurable Software for Machine Control Systems” [WS02]. The authors propose a software architecture based on object-oriented models (components) and an executable formal specification (control plan). The components are described as reusable software components; each modelled with a set of event based external interfaces, a control logic driver, and a set of service protocols. The component’s external interface, which describes the *functionality* of the component, can be customised after compile-time to react to changed requirements of the application or the execution platform. Also, the components service protocols are used to adapt the component to a specific execution environment. The control logic driver is implemented in every component and interprets part of the control plan, which is defined as a set of nested finite state machines (NFSM) and an operation sequence. The NFSM and operation sequence are specified in machine readable table form, and describe the system *behaviour*. The authors describe the software architecture as being runtime reconfigurable, but the reconfigurability is restricted to *non-structural* changes at runtime, whereas structural changes require code regeneration. On the other hand, the architecture provides for separation and independent configuration of structure and behaviour, thus allowing component specification and early testing at the design stage, which is important for some applications. The authors note that the proposed architecture is different from commonly used models such as CORBA, DCOM, or Jini, which are usually based on remote procedure calls and depend on some middle-ware infrastructure whose implementation makes the execution less predictable in timing and performance. The authors also

note that current control software development suffers from application partitioning and implementation with proprietary information as the result of a common top-down system partitioning. This deficiency would require that the components and their interaction are fully specified before their implementation.

In 1995 R.M. Kinmond [Kin95] published a “Survey into the acceptance of prototyping in software development”. The article reviews a survey investigating the commercial usage of prototyping in the UK industry. It concludes that prototyping has the advantage of better user communication, user involvement, and user satisfaction. Many organisations developed the (evolutionary) prototype further into the final system. The major problem with prototyping was the time required for user participation. Difficulties exist in the weakly defined process of documenting the prototype.

In 2002 D.B. Thomas and W. Luk wrote about a “Framework for development and distribution of hardware applications” [TL02]. The authors evaluate a framework called “imaging and graphics operator libraries” (IGOL). The target system is a host with FPGA-based coprocessors attached directly or via a network. The framework intends to discover available hardware resources (HW) at runtime, match requirements to the found HW, configure the HW, synchronise with the HW, and transfer application data between HW and host. To decouple different aspects of application development, IGOL provides a structure with four layers: application, operation, appliance, and configuration. Although the authors state that the framework is not bound to a specific HW or HW programming paradigm, IGOL supports only one FPGA based coprocessor board (RC1000-PP) via the Handel-C hardware programming language. The paper does not give performance numbers.

Interoperability

In 2002 P. Young, V. Berzins, Jun Ge, and Luqi published “Using an object oriented model for resolving representational differences between heterogeneous systems” [YBGL02]. An object-oriented model for interoperability between legacy systems is proposed. The model captures data and operations that ought to be shared between systems in a federation interoperability object model (FIOM). The FIOM also captures the translations required to bridge differences in data representations between the legacy systems.

C. Szyperski compares the component object model (COM), the common object request broker architecture (CORBA), and Java Beans [Szy98]. He concludes that although software component technologies have been proposed in the 1960’s, the component technology is only recently emerging. He states two reasons for the delay: strong industry standards are only recently available and object-oriented programming is only just replacing traditional tools or function-oriented approaches. Problems remaining to be addressed include the enormous complexity introduced by software evolution, versioning, and limited semantic precision of interface contracts.

In 2003 P. Young, N. Chaki, V. Berzins, and Luqi wrote about “Evaluation of

middle-ware architectures in achieving system interoperability” [YCBL03]. The work discusses modelling differences between independently developed systems. A set of criteria is defined to evaluate existing middle-ware architectures, among them CORBA, .NET, XML and others. The authors conclude that current interoperability approaches include several limitations. These include the need to modify existing systems because the requester system is required to utilise the provider system’s model of state and behaviour to access the provider. Another deficit is the limited or non-existing support for the development of translations required to resolve modelling differences among systems.

In 2003 J. Hugues, L. Pautet, and F. Kordon, too, wrote on middle-ware in “Contributions to middle-ware architectures to prototype distribution infrastructures“ [HPK03]. The paper discusses middle-ware under the aspect of *rapid system prototyping*. The authors identify deficiencies in middle-ware architectures being either configurable middle-ware or generic middle-ware. They propose a “schizophrenic” middle-ware concept and its implementation Poly-ORB. Their concept focuses on a neutral core middle-ware (NCM) and definitions of personalities, an application level personality and a protocol level personality. The work gives numbers for code reuse compared to an other architecture (Jonathan). The authors conclude that Poly-ORB has two main advantages important for rapid system prototyping: First, the amount of code that has to be written to support a specific distribution infrastructure is small. Second, any protocol personality can be associated with any application personality.

Conclusion

The above reviewed literature is the extract of a comprehensive scan of recent and current publications. The review shows that only little knowledge about the topic of this thesis is available. I conclude that it is worthwhile and necessary to conduct research on control software for reconfigurable coprocessors, and that the documentation of the findings of this research can provide significant insight.

1.3 Materials

The large hadron collider (LHC), an accelerator, storage ring, and collider for proton beams, is currently build in an international effort at CERN, Geneva, and will deliver proton-proton collisions at a centre of mass energy of 14 TeV. Several detector experiments are associated with the LHC and are under construction, with the ATLAS detector as the biggest of them in terms of size and cost. ATLAS is a general purpose detector designed to cover a broad range of physics research. One of the most popular research programmes is the search for the Higgs boson which is assumed to be the origin of mass, if it exists. The physics of particles with b-flavour (B-physics) will also be investigated at ATLAS, increasing statistics and precision of results also obtained at other experiments. B-physics is expected

to give insight into effects like the violation of combined charge conjugation and parity transformation (CP-violation), particle-antiparticle mixing, and others.

The enormous amount of data produced in the ATLAS detector, about 60 TB/s, has to be reduced to a rate of about 150 MB/s for storage and offline analysis. This reduction is done by a cascaded 3-level trigger that selects interesting events and discards the remaining ones.

One way to tag events involving b-flavour, as needed for B-physics, requires scanning of the whole transition radiation tracker (TRT) for tracks of particles with momenta down to the GeV range. Such scans could be triggered by a lepton with a momentum greater than e.g. 8 GeV, detected by the first level trigger. However, the scan of the TRT, with expected rates of several kHz and an allowed latency of 10 ms, puts high strains on the computing resources of the second level trigger (LVL2), where the scan has to be done.

The Hough transformation is a suitable algorithm for track-finding in dense detectors like the TRT, which has a resolution in the range of 100k evenly distributed space points for every of the four independent parts it consists of. The execution time of a Hough transformation based tracking algorithm on a COTS PC has been measured to be about 20 ms, which exceeds the allowed latency and would consume all of the computing resources in the LVL2.

However, the look up table (LUT) based Hough transformation can benefit from parallelism in implementations on FPGA's. Modern FPGA's offer configurable logic blocks (CLB's) as reprogrammable logic resources and reprogrammable input-output blocks for communication with external systems. Additionally, FPGA's may embed dedicated multipliers, RAM cells, clock managers, and even processor cores. FPGA's are programmed with hardware description languages like VHDL or Verilog, languages derived from languages for sequential processors like HandelC [Cha01], or using libraries for traditional programming languages like JBits [GLS99], CHDL [citehdl], or SystemC [Sys]. High level description of FPGA-configuration is translated in multiple steps (synthesis, architecture mapping, and place and route) to configuration data that is uploaded to the FPGA in order to program its behaviour.

FPGA processors use one or several FPGA's as reconfigurable computing elements. Clocking resources, memories, non-volatile storage, processors, micro-controllers, connectors, and other devices may complete the FPGA processor. FPGA coprocessors are often smaller and may contain less additional devices with the exception of a bridge that connects to the hosting system bus. FPGA processors and coprocessors may use programmable logic devices (PLD's) for system integration. PLD's are programmable like FPGA's but have less computational resources. The configuration of PLD's is non-volatile whereas the configuration of FPGA's is volatile, and PLD's may offer faster and better predictable timing compared to FPGA's.

An FPGA processor is a stand-alone system with loose coupling to a controlling host. It may be implemented as a data-driven systolic processor which plugs into the data-path of an external system through dedicated inputs and outputs. The

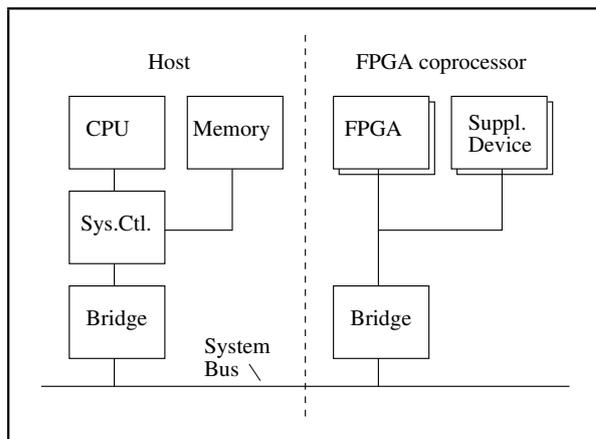


Figure 1.2: Host and FPGA Coprocessor in a System
The stacked boxes indicate multiple instances.

Enable++ FCCM is an FPGA processor that connects to the ATLAS HLT through an S-Link input and an S-Link output. The controlling host is not necessarily part of the data path and a slow serial connection is used to interface the host to the FCCM.

In contrast, an FPGA coprocessor may use the same interface for control and data exchange. For such a coprocessor the controlling host is also part of the data-path. The implementation and topology of the host system bus and the bridge that connects to the system bus on the FPGA coprocessor can influence the combined system performance in different ways. For example, maximum throughput and rate (the inverse of latency) are theoretically limited by the bus architecture, but can be further reduced by the concrete implementations of the corresponding devices. Figure 1.2 sketches a typical system consisting of a host and an FPGA coprocessor.

The hosts used in this work are standard COTS uniprocessor or symmetric multiprocessor PC's with IA32-compatible or IA64 CPU's. All hosts have one or several PCI (32 bit / 33 MHz) or PCI-X (64 bit / 66 MHz) buses, which are used to connect to the FPGA coprocessors. The hosts use GNU/Linux or Microsoft Windows NT operating systems (OS). The Linux OS kernel, whose development started in the early 1990's, is a UNIX-like OS kernel which is open source. Linux became, together with the GNU system utilities, a standard OS, widely used in scientific computing. The ATLAS HLT is build largely from COTS IA32 compatible PC's running GNU/Linux.

C is the most commonly used system programming language, developed in the early 1970's. C offers low-level programming constructs together with high level structured program control, data structures and modular program design. In the 1980's C++ added object orientation to C promoting encapsulation and code reuse, while retaining C's capability to program right above assembly level. Both languages can be used to develop near-optimal performant programs.

In the 1990's The unified modelling language (UML) became a standard tool for modelling software. UML provides a graphical representation of the model and helps in communicating software architecture between developers.

1.4 Principal Results

First problem analysis showed that the requirements for the software would likely change, that tools for e.g. bootstrapping had to be delivered quite early, and that the hardware family to support would grow in the future. I chose a spiral iterative prototyping software process since this process is known to cope well with changing requirements and to support the early delivery of prototypes [McC96].

I chose C++ as implementation language which enables strong encapsulation through the object-orientation paradigm, and I used UML diagrams for analysis and design. To support the development, bootstrapping, debugging, and maintenance of the Atlantis FPGA coprocessors, and the ones to follow, I chose a strict *bottom-up* design for the software. Independent physical devices on a typical FPGA coprocessor were modelled as classes. The resulting mapping between hardware and software led to class representations of components like FPGA, PCI-bridge, and system-integration PLD's. These low level classes can be used to grant access to low level components. I have included a device driver and a corresponding software interface for the parallel-port of PC's, which allows e.g. configuring the PLD's of the ACB within the software framework.

Configuration data (designs) for PLD's and FPGA's play a central role in the software. I implemented a Design class that offers services for parsing and verification of design files in different formats. This class is complemented by a flexible Bitstream class which implements operations on bit-serial representation of configuration data. Bitstream is optimised for fast set/get operations, resulting in negligible overhead for typical operations, like configuration of FPGA's with design files loaded from disk.

All reconfigurable logic on FPGA coprocessors is accessible through JTAG-ports for control, configuration, and inspection. A flexible Jtag class is implemented which supports low level JTAG command protocols. Because JTAG is a bit-serial protocol, Jtag uses Bitstreams for most of its data. A similar approach is applied through an Eeprom class for non-volatile storage (EEPROM) present on FPGA coprocessors and expansion modules. Both, Jtag and Eeprom, are to be adapted to the specific ports through which their respective devices are accessible. This can be done in a straightforward and compact manner. The JTAG class together with the Bitstream class were successfully used by M. Müller to verify most of the networks of the ACB's board layout by in-system boundary-scan.

Data transfer between host and coprocessor is routed through the host system bus, in all practical cases a PCI bus, and through the PCI bridge on the coprocessor. The according Bridge class has a complementary OS-dependent device driver class which grants access to the PCI address space occupied by the PCI-

bridge. The Driver class is part of a class hierarchy which is designed to support multiple drivers detected at runtime. All drivers allow mapping the PCI address space of the bridge into user memory space, providing means for memory mapped PIO data transfer. The bridges offer DMA engines for data transfer independent from the host CPU. DMA is handled by an interface of the bridge class, and a DMABuffer class that represents data chunks to be transferred. Such data chunks can be prepared in user- and kernel-space host memory. The bridge-, DMABuffer-, and driver-classes collaborate in DMA transfers, hiding issues specific to OS, bridge, and driver.

Every supported FPGA coprocessor, μ Enable, μ Enable2, ACB, AIB, MPRACE, and Robin, is assigned a respective Control class. All Control classes inherit common functionality and a common interface from the Control base class. The specialised Control classes act as builders, composing the runtime object hierarchies needed to control and access the specific coprocessors. Such an object hierarchy consists of instantiations of a specific bridge class, a Pld class, which encapsulates the services of the system integration PLD, and possibly other class instances. Since the control classes, as it is the case for all classes representing devices in the software, are implemented using the “instantiation is initialisation” paradigm, they provide type-safe allocation and identification for a specific coprocessor by their constructors.

The total code size of the software has grown from about 380 kB after initial implementation to currently about 1000 kB. More than half of the current code, 52%, belongs to core functions. The core is reused by the six support packages for the specific coprocessors which make up the remaining 48% of code. The size of the support packages ranges from 146 kB for the complex ACB (consisting of 4 FPGA's and 2 integration PLD's) to 50 kB for the Robin. The average size of code needed to support a specific coprocessor is 79 kB. On average, about 60 % of the code in a support package covers integration PLD services, while the remaining is partitioned between the Control class (30%) and miscellaneous code (10%).

The rating of performance is very important for most applications of FPGA coprocessors. Standard timing functions on Windows or GNU/Linux systems usually trade resolution against latency. I implemented a small package labelled PCC, which uses a special register present on all IA32 and IA64 CPU's for timing. Resolution and latency of the PCC timing functions scale with the clock frequency of the CPU. On a 2GHz CPU the resolution is about 2 ns, more than two orders of magnitude better than any standard timing functions. The latency of the PCC timing functions is about 50ns, 20 times better than standard high resolution timers.

Two kinds of deliverables are build from the software sources. These are a shared library that is linked to user applications, and a set of utilities that are also linked with the library. The utilities are e.g. used for bootstrapping and firmware update. The device driver for Linux systems is part of the sources, while commercial drivers are used on Windows systems. Only the device driver subclasses contain OS specific code which is hidden by the device driver base class. A set of makefiles is provided to build the library and the utilities on GNU/Linux. On Win-

dows the same is achieved by Microsoft's Visual-C projects. I ported the library and the Linux driver to Linux on IA64 based hosts, which was accompanied by porting the device driver to the Linux 2.6 series kernel. The software builds and executes correctly with GNU GCC compiler versions ranging from 2.95 to 3.4 on Linux 2.4 and 2.6, and with actual version of Visual-C (VC6, VC.net) on Windows NT 4 and Windows 2000.

The integration of the Enable++ FPGA processor into the ATM testbed as implemented by the Saclay group required connecting the S-Link input and output of the Enable++ with a modified destination (DST) processor of the testbed. Thus, data fragments can be routed through the FPGA processor. I made preparatory studies with a dummy S-Link loop-back to measure reliability and throughput of the S-Link data transfers. I measured the saturated throughput for big fragments (> 100 kB) being 27 MB/s for DMA transfers and 5 MB/s for PIO transfers. The overhead of 350 μ s, which is required for setting up DMA transfers, reduces the performance as compared to PIO transfers for small packets (< 2 kB). I inserted the Enable++ into the loop-back, and complemented the stubs that were provided by the Saclay group for interfacing to the testbed software framework. The resulting system was then successfully connected to the testbed network.

After successfully bootstrapping the ACB and building a first iteration of the software, we aimed at integrating the Atlantis system with the Pilot-Project testbed. I adapted the CPU-based TRT scan algorithm to occasionally redirect hit data to the reconfigurable computing facilities of the ACB and to receive results back from the ACB, completing the computation chain. Additionally, the control software sources together with the necessary adaptations of the algorithm were included in the build system of the T2Ref software used in the testbed. Integration of the resulting system with the testbed at CERN showed no problems, given that the algorithm executed on the ACB provided only a limited range of functions.

Subsequent studies with the ACB resulted in an implementation of the TRT algorithm that uses all four FPGA's of the ACB together with the large memory modules designed for holding the required LUT data. The application uses configuration and runtime reconfiguration of all four ACB FPGA's, calculation, preparation and transfer (using DMA) of the LUT fragments to the memory modules, and finally routing hit data to the ACB and reading calculated track data from the ACB. The application was then extended to use two ACB's together in a single Atlantis system, executing the algorithm parallel to each other on the two reconfigurable coprocessors. The adaptation layer used to adapt the CPU implementation of the algorithm to the control software, and the evolved control software itself, were later successfully used by A. Khomich in his implementation of the algorithm for the MPRACE reconfigurable coprocessor.

Based on the experience with the control software for reconfigurable coprocessors I worked on software that is runtime reconfigurable, resembling the runtime-reconfigurability of the hardware. I implemented a prototype that simulates a typical operation sequence involving three chained devices. The devices are, like in the above described software, represented as classes which encapsulate the soft-

ware view of the devices' behaviours. In contrast to the established architecture, the classes are completely independent from each other, which makes them component-like software entities. Connections between the components, which are required for component collaboration, are established by superordinate software at runtime. Such runtime-dynamic connections are enabled by a common interface to the involved components and the notion of typed transaction slots ("methods") between the components. A component may have two types of transactions, services which offer the component's functionality to clients, and requests, which need to be satisfied by servers. Services and requests are enumerated and connected pairwise by using the common interface. Components are instantiated, connected and disconnected at runtime.

A further step towards runtime reconfigurable software is the loading, registering and unloading of components from shared libraries at runtime. In an extended prototype, the libraries and the components contained in them are identified by name, in addition to the identification of transactions by name as it was established for the runtime-dynamic connections which have been described above.

1.5 Main Conclusions Suggested by the Results

I used an iterative prototyping process for the life-cycle of the software. The process supports early delivery, good visibility, and the adaptation to changing requirements, but tends to hide design deficiencies and thus may establish poor design. Accordingly, overall design can suffer from the process and maintainability may be reduced. The time-span for development and use of the software was not well defined during initial requirements collection and the software has been used in applications since an early stage in the development process. Accordingly, poor design often kept undetected and, if reasons for re-design appeared, it was often difficult to implement better solutions. Deficiencies that arose from this problem became visible in the case of e.g. the registry package, which breaks encapsulation and whose maintenance is cumbersome. I conclude that great care is needed in following the iterative prototyping software process to ensure that repeated small iterations do not veil the goal of good overall design.

Bottom-up design provides the base for tools, adaptation and evolution. The design supports encapsulation and maintainability. However, interfacing between collaborating classes can break encapsulation and introduce dependencies. Such dependencies exist in the software but they usually lie within the various coprocessor-support packages. Although bottom-up design tends to produce many small low-level classes, many of the classes that exist in the software are generalised by a few base classes, taking advantage of software reuse.

A major problem caused by the bottom-up design turned out to be the lack of a stable user-API (top level classes), that was present in the first iterations of the software. The mapping between hardware components and software models, which is a result of initial requirements analysis, supports reuse and encapsulation

of device-specific knowledge. Yet, the focus on devices and their software models seems to hide activities and processes like data transfer, FPGA configuration, and coprocessor identification and allocation. According problems still need to be properly addressed. I suggest a carefully biased combination of bottom-up with top-down design for future developments.

The software provides functions by services of low level software models, e.g. the read-out of the device identification of a JTAG-capable device is provided by the corresponding JTAG-class. This class processes according requests and forwards resulting low level request to classes that follow in a chain of collaborating classes, until the request leaves the software and reaches the coprocessor through the system bus. The chain of classes resembles the chain of devices on the coprocessor. This architecture is different to architectures like the one described by Schumacher et al [PSSL00], that uses two hierarchies of classes, one for modelling of devices, and one for accessing this devices. In my architecture, device models and device accessors are identical.

Performance of data transfer between host and coprocessor is critical to overall system performance. I made a number of measurements to quantify transfer rates and latencies, and the overhead caused by object-oriented software layering. The results of these measurements show that the transfer of data between host and coprocessor often dominates system performance and that the software-overhead is almost negligible. I conclude that the benefits of software layering, especially encapsulation, reuse, and maintainability, overweigh the small software overhead that has to be paid.

Several applications use the software which is presented in this thesis. These applications are executed on a variety of single-processor and multi-processors hosts running under different versions of the Microsoft Windows and Linux/GNU operating systems. Most of this applications use the μ Enable, the ACB, or the MPRACE reconfigurable coprocessors, and some applications use two or more coprocessors, even of different type, in parallel. The software can be compiled from a single code base by usual C/C++ compilers like Microsoft Visual C and GNU C/C++. This shows that the software received great acceptance and reached the goal of independency from operating system, compiler, and host architecture, and that the software supports a variety of different coprocessors. The software's source code has a size of about 1 MB, which is a hindrance for some applications. Yet, it would be easily possible to create simplified versions of the software by omitting the support for unneeded coprocessors or operating systems.

The run-time-reconfigurability of reconfigurable coprocessors is only marginally reflected by the software. I addressed this by the development of a compact and straightforward runtime-reconfigurable software architecture. Since my approach uses a simple naming scheme for the loading of components and the identification of transaction ports of the components, a software-overhead is introduced. Though this overhead is significant, it may be acceptable under specific circumstances. My approach does not depend on middle-ware like CORBA, which too introduces an overhead in code and performance. Further research is needed to

investigate the possible advantage and feasibility of the demonstrated architecture for real applications. Stewart and Khosla [SK96], and Wang and Shin [WS02] propose architectures for reconfigurable real-time software that is used for machine control. Both architectures separate functions from behaviour by using finite state machine interpreters. On the other hand, my approach provides reconfiguration of behaviour only by structural reconfiguration. Wang and Shin, too, avoid middleware because of performance requirements.

Chapter 2

Materials

2.1 Introduction

In this chapter I describe the motives for using reconfigurable processors and coprocessors, including some of their applications, the nature of reconfigurable computing and systems, and the architectures of reconfigurable coprocessors that were used, together with the architecture of hosting computers.

In the first part I present in some detail the main argument for using reconfigurable processors and coprocessors, i.e. real-time processing of ATLAS detector data, including some background information about ATLAS, physics research at ATLAS, and the LHC, to which ATLAS is associated. The ATLAS high-level-trigger (HLT) is the supposed environment of reconfigurable coprocessors and imposes requirements on the coprocessors and the control software for reconfigurable coprocessors (CSRC) that was developed as the result of this work. Thus, some room is given for a more detailed description of the ATLAS HLT architecture and software.

I will not mention further about other applications of reconfigurable coprocessors that were implemented using CSRC, like astrophysical simulations [KKM⁺99, Lie03, LKM02], data compression [SD02], and image processing [HGKM02].

In the second part I will present some background about computing in general, custom computing, and reconfigurable computing. I describe at a glance reconfigurable devices and the programming of reconfigurable systems.

In the third part I will list, and sketch the architectures of, reconfigurable processors and coprocessors that were developed by the FPGA group at the University of Mannheim during the last years, most of them now being supported by the CSRC. Finally, in the last part, I give a short overview on computers that were used to host and control reconfigurable coprocessors

2.2 LHC, Atlas, Trigger

2.2.1 Large Hadron Collider and Associated Detectors

The Large Hadron Collider (LHC) is the next-generation particle accelerator and collider located at CERN, Geneva, Switzerland [Eva95, LHC95]. LHC is currently in the phase of construction and is commissioned to start operation in 2007, replacing the Large Electron/Positron Collider (LEP). LEP stopped operation in 2000.

LEP was working with centre of mass energies of up to ~ 200 GeV, which covers the mass region of the weak-force-carrying W^{+-} and Z^0 bosons. The nature of lepton (e^+, e^-) colliders prevented that the centre of mass energy of LEP was boosted further. This barrier is caused by the beam energy loss due to synchrotron radiation.

Using hadrons (protons in the beginning with the possibility to upgrade to heavy ions, e.g. lead, in later operation modes), the relative energy loss of the beam, that is bent to a circle with a circumference of about 27 km, is reduced by more than 13 orders of magnitude. This is a result of the mass ratio $m_p/m_e \sim 2000$ between protons and electrons.

Although the LHC uses the same tunnel as the former LEP, all the infrastructure of the LEP had to be removed to give room for the new superconducting magnets and high frequency cavities used to shape, bend and accelerate the beams. The beam bending dipole magnets will have a field strength of 8.4 T.

As a consequence, the centre of mass energy can be increased to 14 TeV for proton-proton collisions. Thus the LHC opens a window to a new energy range, that even exceeds the energy of the Tevatron at Fermilab, Illinois, USA, that is designed for 2 TeV.

Additional to the energy boost, the LHC offers a impressive luminosity ranging from about $10^{33} \text{ cm}^{-2} \text{ s}^{-1}$ at machine start-up up to the design luminosity of more than $10^{34} \text{ cm}^{-2} \text{ s}^{-1}$. This enables the experiments at the LHC to search for even the rarest physics signatures. The beam lifetime is 10 hours.

The LHC beams will consist of bunches separated by 25 ns corresponding to a bunch crossing rate of 40 MHz. At design luminosity every bunch crossing will on average produce 23 inelastic proton-proton collisions.

Detectors at the LHC

LHC will use two beams of particles travelling in opposite direction in the same tunnel. The beams are directed to cross at eight interaction points, and four of them will be equipped with dedicated particle detectors. Additional to the four experiments (ALICE, LHCb, CMS and ATLAS) at the interaction points there will be a fifth one, TOTEM [TOT99], that is partly integrated into the CMS design and partly on the beam line. TOTEM is mostly used for calibrating beam properties like luminosity.

ALICE (A Large Ion Collider Experiment) is a general-purpose heavy-ion detector designed to study the physics of strongly interacting matter and the quark-gluon plasma in nucleus-nucleus reactions [ALI95]. The detector is designed to cope with the highest particle multiplicities anticipated for Pb-Pb collisions. ALICE will be operational at the start-up of the LHC.

LHCb [LHC98] is designed to study parity-violation and rare events in the decays of heavy-flavoured hadrons, in particular B mesons. [DFH⁺99] gives a short survey of the LHCb trigger and data acquisition.

CMS, the Compact Muon Solenoid, is one of two general-purpose proton-proton detectors [CMS94]. The main focus in the CMS design is its strong 4 T superconducting solenoid with muon detectors in its outside flux return yoke made of iron. The solenoid will be the largest magnet ever built. The magnet coil is big enough to give room for almost all calorimeter detectors together with the inner tracking detectors.

ATLAS is the second general-purpose detector for recording proton-proton collisions [ATL94]. (The feasibility of ATLAS for recording collisions of heavy ions is under study [Nev03, Tak03].) Like CMS, ATLAS has a central superconducting solenoid. In contrast to CMS the solenoid is smaller and weaker and the return yoke consists of the iron in the hadronic calorimeter tiles located outside the magnet coil. Due to the reduced size of the solenoid, the calorimeter is placed outside the solenoid, and the inner detector consists entirely of tracking detectors. In addition to the inner solenoid, ATLAS has three other superconducting magnet systems, a barrel toroid surrounding the inner solenoid, and two endcap toroids. The toroids are instrumented with drift tubes and cathode strip chambers as muon detectors. The overall size of the cylindrical ATLAS detector is 46 meters along its axis and 22 meters in diameter. ATLAS has about eight times the volume of CMS and a total mass of 7000 tons with 4000 tons according to the hadronic calorimeter.

2.2.2 Physics and Tracking at the ATLAS Detector

The boost in centre of mass energy provided by LHC is mostly motivated by the search for new physics, possibly answering the questions that the overwhelming successful *standard model* (SM) leaves unanswered. The most popular of these questions is the search for the origin of mass. Others are the reduction of the 18 parameters that need to be given as an input to the SM as results of measurements, the exact nature of symmetry breaking (e.g. the asymmetry in matter - antimatter densities in the known universe), or the mechanisms that cause parity violation in some rare particle decays.

Another area of physics research at the LHC is the investigation of the quark-gluon-plasma that is assumed to establish at high-energy collisions of heavy ions.

This is especially important because of the strong relation to big-bang-cosmology, assuming the universe went through a stage of quark-gluon-plasma in its very early lifetime. The detector design that is needed to detect and analyse such plasma differs significantly from the design of a “general purpose” or dedicated B-physics detector, so consequently ALICE is designed for exactly this purpose.

Finally, though the SM is understood well, some precision measurements can provide means for refining it, or even could give evidence for extending or modifying it. The most popular sector in this research programme is the B-Physics. A dedicated detector, LHCb, is installed at the LHC to fulfil the requirements for precise and high statistics measurements in the area of B-Physics. However, since B-Physics is characterised by the need for high accuracy measurements, knowledge on B-Physics can benefit from the redundancy of information given by different experiments, increasing statistics, accuracy, and evidence.

Surprisingly it is the signatures of the decay of the relatively light B mesons that impose heavy requirements in terms of computing power in the ATLAS trigger.

ATLAS, like other detectors, is build from several sub-detectors and magnet coils arranged in onion-like layers surrounding the interaction point (IP). The IP is located in the beam pipe that crosses the cylindrical detector on its axis. The sub-detectors are grouped in the inner detector (ID) that surrounds the beam pipe, followed by the outer detector (OD). OD and ID are separated by the central solenoid in the barrel. The OD encloses the solenoid and houses the toroid coils.

The magnet field produced by the superconducting coils bend the particle trajectories recorded in the different detectors, giving means for reconstructing the particles momentums. The particles energies are measured by the calorimetry system that is located in the innermost layers of the OD. The calorimetry consists of the electromagnetic calorimeter followed by the hadronic calorimeter. Surrounding the calorimeter in the OD is the muon spectrometer, that houses the toroid.

Besides neutrinos, that give no signal in the ATLAS detector at all, only muons can escape the calorimeter, that absorbs all remaining particles produced in the proton-proton collisions at the IP like photons, electrons, and hadrons.

The ID consists of 3 layers. These are: the pixel -, the semiconductor tracker (SCT) -, and the transition radiation tracker (TRT) - detectors. The pixel detector is a 3 layer (4 discs in the endcaps on each side) silicon semiconductor tracking detector with a resolution of $12 \mu\text{m}$ in $R\phi$ and $66 \mu\text{m}$ in z . The SCT uses 4 layers of silicon microstrip detectors in the barrel and 9 wheels in the endcaps. The resolution of the SCT is $16 \mu\text{m}$ in $R\phi$ and $580 \mu\text{m}$ in z .

TRT The TRT uses straws of of about 0.5 - 1.5 meters length and 0.004 meters diameter. The straws are each equipped with a central readout wire and and filled with Xe-rich gas and operate as drift tubes. The distance from a particle trajectory crossing the straw to the central readout wire can be determined with a resolution better than $150 \mu\text{m}$ using drift time information. The space between the straws is filled with a *radiator*. The radiator is made of a material with an alternating

index of refraction. Some fast charged particles will produce soft X-rays (transition radiation) while crossing the radiator. This radiation will be recorded in the straws as an additional *high threshold* signal, and is used for particle identification, i.e. telling electrons from muons and pions.

The TRT consists of left and right endcaps and the barrel, which is divided in left and right barrel. The straws in the endcaps are pointing to the beam axis and lie in planes perpendicular to the beam. The endcaps are organised as wheels containing planes of straws. Left and right endcap consist of 160,000 straws each.

The barrel straws are parallel to the beam, left and right barrel contain about 50,000 straws each. The barrel straws are organised in sectors with a 32-fold rotational symmetry around the beam axis. The TRT is designed such that every particle with $|\eta| < 2.5$ ¹ range will cross at least 36 straws. Combining the space-point information from all the straws hit, the tracking resolution exceeds 50 μm , which resembles the resolution of the two other silicon detectors in the ID.

The near-uniform magnetic field inside the central solenoid has a strength of approx. 2 Tesla. The magnet field bends the trajectories of charged particles to circles or circle segments, that are recorded in the ID. The radius of these circles is proportional to the particles momentum and equals 1.6 meters for a single charged particles with a momentum of 1 GeV.

The cylindrical ID has a radius of 1.15 meters and encloses the beam pipe over a length of 7 meters.

Most of the interesting particles origin at or near the IP. The vertex resolution of the ID reaches about 100 μm in z and 10 μm in the x-y plane.

The occupancy describes the fraction of active detector elements in a given event. It is a function of the luminosity, the physics of the interaction of the particle collisions, and the (sub)detectors response to particles. All these factors are only known to a limited certainty before the initial data taking.

The occupancy of the TRT exceeds 25% at high luminosity ($10^{34} \text{cm}^{-2} \text{s}^{-1}$) in ATLAS detector simulations.

B-Physics The standard model, together with the theory of general relativity, describes all known forces and matter, except the gravitation, which is still not fully understood. The electromagnetic, weak, and strong (nuclear) forces are mediated by integral spin *bosons*, whereas matter consists of *fermions* which have spin 1/2.

Fermions are divided into *leptons* and *hadrons*, and are grouped into three families. All known stable matter is made of fermions from the first family, that consists of the electron (e^-) and the anti-electron neutrino ($\bar{\nu}_e$) as leptons, and the up and down quarks (u, d) which constitute the hadrons, together with their anti-particles ($e^+, \nu_e, \bar{u}, \bar{d}$). The other two families have the same structure, but their constituents decay fast by electroweak interactions into members of the first family or energy, therefore their constituents only can be observed in high energy physics

¹ η denotes *pseudo-rapidity*, which is essentially a measure for the angle at which a particle is leaving the IP.

experiments and cosmic radiation. The quarks which make up the second and third family are the strange (s) and the charm (c) quarks, and the bottom (b) and the top (t) quarks, respectively.

Quarks can not exist on their own, they are always bound into particles containing two (*mesons*) or three (*baryons*) of them.

Of special interest here are the mesons containing the lighter quarks of the second and third family, that are the strange and the bottom quarks. Among lightest of these two meson families are the K_0 and the B_0 mesons. These two quark systems show the effects of particle- antiparticle mixing and parity violation, and the study of these effects can give measures for refining or even modifying the SM. The mechanism of mixing is illustrated as Feynman diagrams in equations 2.2 and 2.3. The most popular decay mode of the B_0 meson is illustrated in eq. 2.1. The physics of particles containing b quarks is called *b-physics*.

$$B^0(d\bar{b}) = \left\{ \begin{array}{l} \text{---} d \text{---} \\ \text{---} \bar{s} \text{---} \\ \text{---} \bar{b} \text{---} \\ \text{---} \bar{c} \text{---} \end{array} \right\} = \begin{array}{l} K^0(d\bar{s}) \\ J/\psi(c\bar{c}) \end{array} \quad (2.1)$$

$$B^0 = \left\{ \begin{array}{l} d \text{---} t, c, u \text{---} b \\ \bar{b} \text{---} \bar{t}, \bar{c}, \bar{u} \text{---} \bar{d} \end{array} \right\} = \bar{B}^0 \quad (2.2)$$

$$K^0 = \left\{ \begin{array}{l} d \text{---} t, c, u \text{---} s \\ \bar{s} \text{---} \bar{t}, \bar{c}, \bar{u} \text{---} \bar{d} \end{array} \right\} = \bar{K}^0 \quad (2.3)$$

Why it is so difficult Most of the particles created in the proton-proton collisions at the interaction point (IP) decay fast into secondary particles and finally into “stable” particles like electrons, muons, and pions. Here, stable means that the particles’ lifetimes are long enough to survive until they are stopped in the calorimeter (pions), or to leave the detector volume (muons).

Another final state resulting from hadrons (quarks) leaving the IP is called jet, i.e. a bundle of mostly hadronic particles that form a narrow cone. Final states with electrons and pions are illustrated in figure 2.1(a) and with jets in figure 2.1(b)

The lightest particles with bottom-flavour, i.e. particles containing a b -quark, are the B^\pm and B^0 mesons, having a mass of about 5.3GeV . The cleanest signatures of B meson decays are such with final states of two lepton-antilepton pairs, e.g. e^+, e^-, μ^+, μ^- or with a lepton and a pion pair, like e^+, e^-, π^+, π^- . Consequently

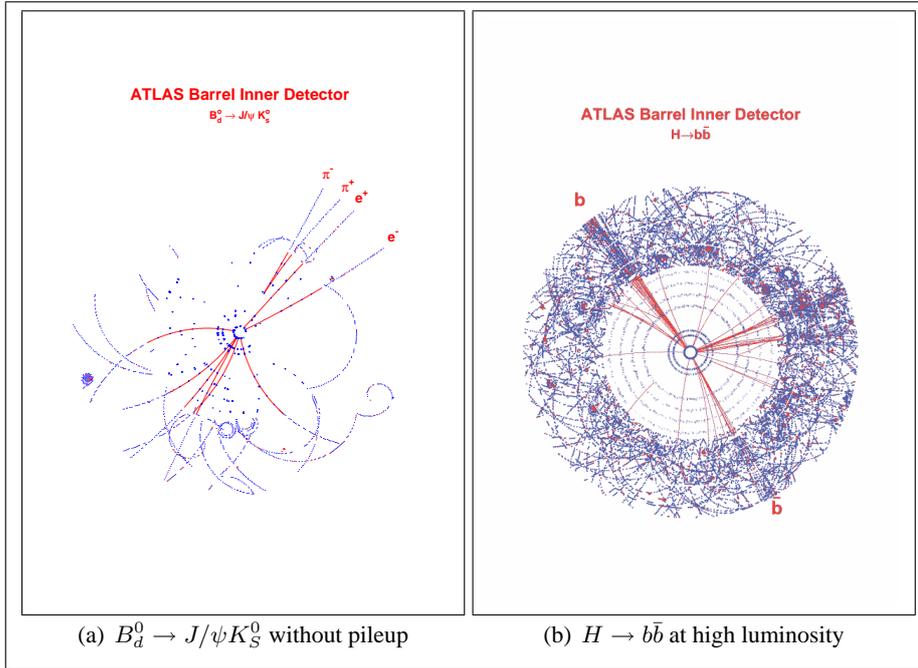


Figure 2.1: Simulations of Events in the Inner Detector

the tagging of a B meson requires identifying 4 particles in the detector with momenta down to 0.5GeV . This can in general only be done by scanning the whole inner detector, or at least the TRT, for tracks with $p_t > 0.5\text{GeV}$. As can be seen from figure 2.1(a) this is reasonable in the case of low luminosity ($10^{33}\text{cm}^{-2}\text{s}^{-1}$) but becomes very difficult when luminosity is enhanced in later LHC runs (figure 2.1(b)).

Hough Transformation

The trajectory of a charged particles in a magnetic field is bend to a circle whose radius is proportional to the momentum of the particle. Identifying particles and measuring their momentum relies on finding circles or circle segments in the event recording of the tracking detectors.

A circle in a plane can be described by a point in a three-dimensional parameter space, two dimensions for the position of the circles centre and one for the radius r . Since the trajectories of the particles in question are originating at or near the interaction point (IP), the parameter space is reduced to two dimensions. For the following, a suitable choice for the first dimensions is the curvature $c = 1/r$. The starting angle ϕ_0 of the trajectory at the IP is chosen for the the second dimension.

As a consequence, a track (a circle, circle segment, or straight line crossing the IP) in the (r, ϕ) plane can be described as a *point* in the (ϕ_0, c) plane. I refer to the former (r, ϕ) plane as *tracking space*, to the latter (ϕ_0, c) plane as *Hough space*.

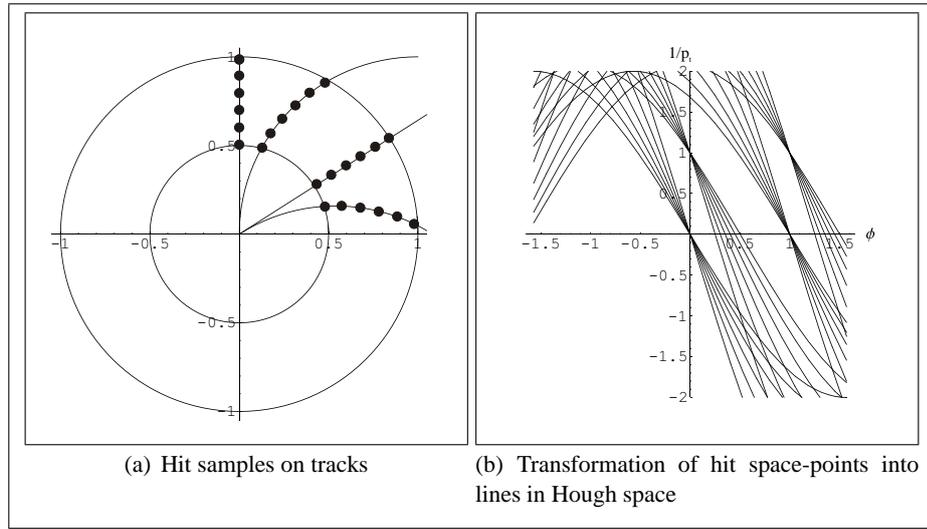


Figure 2.2: Hough Transformation

The transformation from tracking space to Hough space is the *Hough transformation* [Hou58]. The collectivity of circles touching the IP and a given point (r, ϕ) in the tracking space is represented by a line in the Hough space (equation 2.4).

$$(r, \phi) \xrightarrow{\text{Hough}} c(\phi_0)_{r, \phi} = 2/r \sin(\phi - \phi_0) \quad (2.4)$$

Tracking detectors have limited resolution and give a quantised image of the recorded particle tracks. The image consist of a list of *hits*, i.e. detector elements that were triggered by passing particles. Figure 2.2(a) gives a simplified example of simulated tracks and corresponding hits in a cylindrical tracking detector, and figure 2.2(b) shows an excerpt of the lines in Hough space after Hough transformation.

LUT-Hough-Algorithm

The TRT- Look Up Table (LUT) - Hough algorithm [Ses00] uses an initial discrete Hough transformation plus maximum finding to find track candidates in the event recording of the TRT detector. After track-finding the algorithm applies the steps of track-splitting, track-merging, and track-fitting to refine the track parameters and occasionally reject track candidates.

The discrete Hough transformation requires that for every active hit in an event an amount of histogram counters in Hough space is incremented. The histogram counters are identified through a LUT which stores lists of histogram counter identifiers corresponding to every possible hit in the TRT.

Figure 2.3 shows a density plot of the discrete Hough space after summing up and figure 2.4 shows the same histogram in a bar plot. The peaks in this plot clearly indicate the parameters of the four tracks from figure 2.2(a).

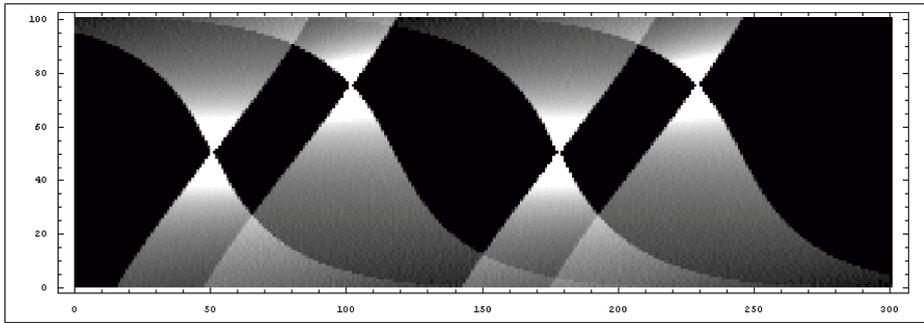


Figure 2.3: Hough Histogram

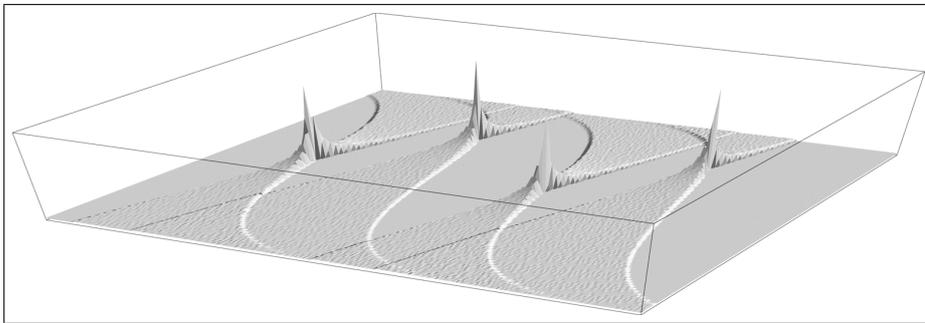


Figure 2.4: Hough Histogram, Surface View

LUT based histogramming in Hough space is also used in TRTxK algorithm [GS03] and in an algorithm for pattern recognition in the precision tracker [BDS99], and has evolved to a standard technique for initial track finding in ATLAS second level trigger feature extraction (FEX) algorithms. [S⁺03] gives a survey of FEX algorithms in the ATLAS high level trigger.

The initial histogramming and maximum finding step takes a significant fraction of processing time for all considered algorithms, reaching 80% in the case of the TRT-LUT-Hough algorithm ([HKM⁺99]). The processing time is proportional to the size of the LUT and the number of hits processed, and is dominated by memory load and store operations, i.e. retrieving the histogram counter ID's from the LUT and counter load, increment, and store operations. The traditional CPU implementation requires sequential update of approx. 100 histogram counters for every processed hit. Both, histogramming and maximum finding, can gain from systolic or parallel implementations on custom computing machines, which is additionally relieved by the absence of floating point operations. A number of FPGA based processors and coprocessors have been built at the University of Mannheim that serve as platforms for implementations of the histogramming and maximum finding steps of the TRT-LUT-Hough algorithm.

Table 2.1: Trigger Rates and Latencies

Stage	Output Rate	Latency
Detector	40 MHz	1 μ s ^a
LVL1	75 kHz	2.5 μ s
LVL2	3 kHz	10 ms
EF	100 Hz	1 s

^aRough estimation of the average latency which depends on the definition of the interface between detector and front-end-electronics. It is dominated by about 75 meters cable delay and digitisation latency.

These numbers are estimations depending on luminosity, (possibly unknown) particle physics, detector performance, and trigger menus. However, the 40 MHz detector rate is determined by the LHC bunch-crossing rate.

2.2.3 ATLAS Trigger

ATLAS operates at a bunch crossing rate of 40 MHz. Every event recorded in the detector has a data size of approx. 1.5 MB, depending on the actual occupancy. The enormous amount of data produced in the detector has to be reduced to a event rate of 100 Hz, suitable for storage and offline analysis. This is done in a three level trigger [ATL03].

The first level trigger (LVL1) is build with dedicated electronics on the detector. LVL1 uses coarse-grained calorimeter data and a subset of the muon spectrometer. This trigger stage is designed to have a trigger rate of 75 kHz. Additional to the trigger decision the LVL1 delivers region of interest (ROI) pointers to the second level trigger (LVL2), which are used as guidance for the second level trigger algorithms.

The second level trigger (LVL2) has access to all event data stored in the read-out buffers (ROB's) in addition to the LVL1 trigger decision and ROI pointers. The LVL2 uses commodity off the shelf (COTS) computing and networking technology to reduce the trigger rate to approx. 3 kHz.

The last trigger level is called event filter (EF). The EF is responsible to finally reduce the event rate to approx. 100 Hz which is suitable for mass storage. The total data rate leaving the EF will be of the order of 150 MB/s. The EF has, like LVL2, access to all event data stored in the ROB's and uses additional calibration data to be more precise on its decisions. The algorithms used in the EF are derived from offline algorithms.

Table 2.1 surveys the expected rates at the output of the concerned trigger stage while figure 2.9 sketches the main components of the current high level trigger and data acquisition architecture.

Previous studies for the second level trigger (LVL2) have been seeded by two possible architectures presented in [ATL94], Architecture-A and Architecture-B.

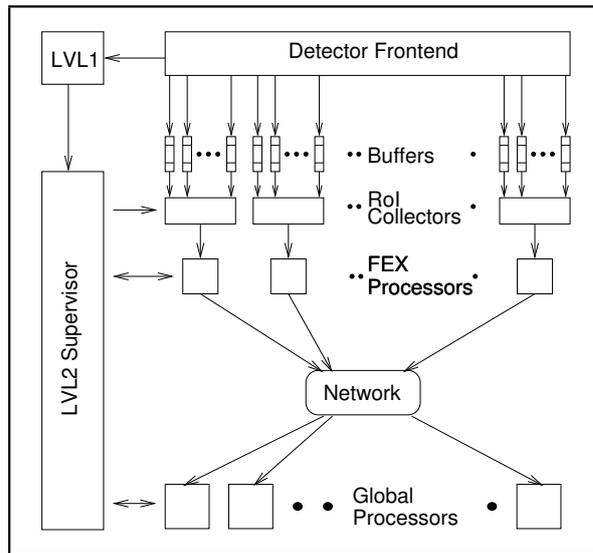


Figure 2.5: Trigger Architecture A

A third proposal for the LVL2, developed on top of the understandings of Arch-A and -B, Architecture-C ([ATL98]), is currently assumed to represent the final choice for the trigger staging in 2007. The different architectures are described in brief in the following.

- Architecture-A

Sub-detector specific feature extraction (FEX) is performed in custom dedicated data-driven processors with full level-1 trigger rate. After feature extraction the event data is transferred by a global network to a global processing unit (GP) that was allocated by the supervisor to process the event and make the final LVL2 decision. Refer to figure 2.5.

- Architecture-B

The data-driven FEX processors from Arch-A are replaced by small farms of general purpose local processors (LP) dedicated to specific sub-detectors. LP's are connected by local networks to the readout buffers (ROB's) and by a global network to the GP's. Refer to figure 2.6.

- Architecture-C

Unified symmetric architecture with single network and global farm, refer to figure 2.7. A single GP is allocated by the supervisor to do the entire high-level-trigger processing. Event data is transferred through a single network from the ROB's to the GP.

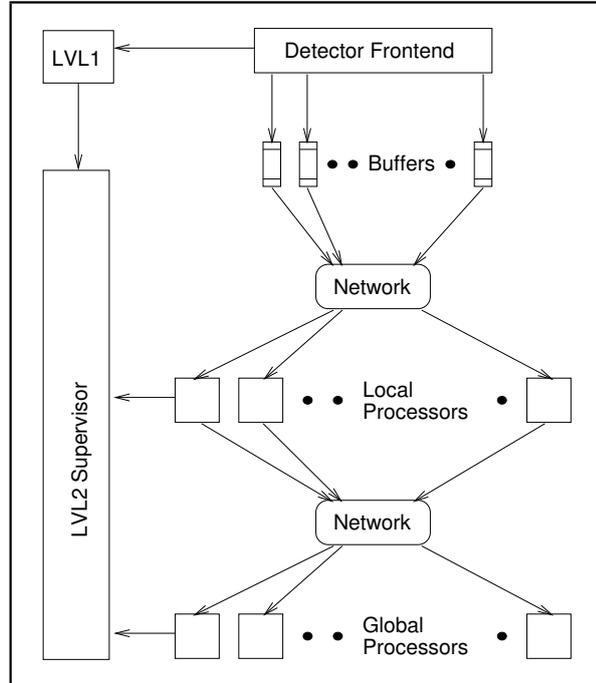


Figure 2.6: Trigger Architecture B

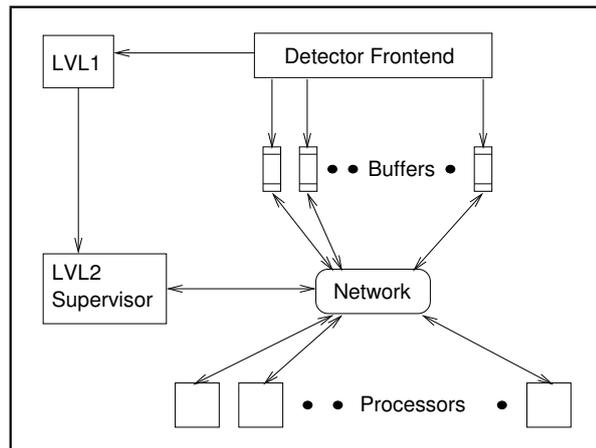


Figure 2.7: Trigger Architecture C

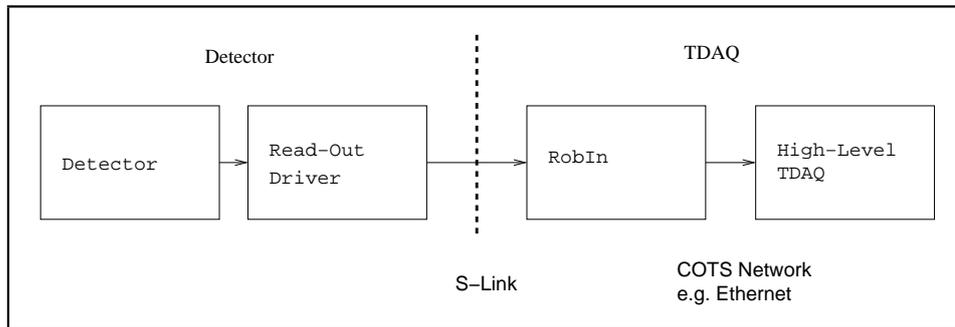


Figure 2.8: S-Link in TDAQ

S-Link is used in ATLAS to move event data across the boundary between the detector and TDAQ. The number of S-Link channels required is about 1600.

2.2.4 S-Link

S-Link defines a simple link interface that is used to move data and control messages between different layers of detector front-end-electronics and read-out. The S-Link definition is a CERN standard and is used in CERN experiments like CMS, LHCb and ATLAS, and in projects outside of CERN.

S-Link provides unidirectional data transport between the link source interface (LSI) and the link destination interface (LDI), error detection, and self-test. The duplex version of S-Link has a small return channel for flow control and return line signals.

S-Link does not define the physical link and different implementations like optical fibres and parallel electrical cables are used.

Common to all S-Link implementations is the conformance to the definition of the signals on the LSI and LDI. These include clock, data, and control signals like reset input and error output. The definition allows for 8, 16, and 32 bit wide transfers, but 32 bit wide transfers are commonly implemented. Data is transferred with at least 40 MHz clock, resulting in a rate of 160 MB/s.

The use of S-Link to move event data from the ATLAS detector to the high level trigger and data acquisition (TDAQ) is illustrated in figure 2.8. Here S-Link connects between the readout driver (ROD), which is part of the detector front-end electronics, and the readout buffer input (ROBIN), part of the TDAQ. The total bandwidth required here is in the order of 150 GB/s and about 1600 S-Link channels will be used. The distance between the detector and the TDAQ is of some 100 meters. Additionally, the detector environment where the S-Link connections start is not very “friendly” because of the strong radiation and electromagnetic field during LHC operation. Therefore, the physical S-Link medium to be used here is supposed to be optical fibres, because they can carry long distances with low attenuation and are more immune to electromagnetic fields.

FPGA processors have been used in recent architecture studies for the ATLAS trigger. These processors were supposed to perform fast data-driven feature extrac-

tions (FEX) algorithms on local event data from selected sub-detectors. The results of the FEX were meant to build the basis on which another stage of processors or processor farms would make the final level 2 trigger decisions. The FPGA based FEX processors under study were designed as data driven systolic processors and the properties of the S-Link definition makes S-Link the natural choice for data input and results output of these. Accordingly, the vertical slice tests described later have been using S-Link for the data path.

[Sof, K⁺98, vdBH, Iwa, BMvdB97, Ver00]

2.2.5 High Level Trigger Software, Prototypes and Testbeds

Many FPGA based processors and coprocessors have been build by the Mannheim FPGA group. All these systems have been integrated into adequate prototype frameworks of the ATLAS high level trigger (HLT). These integrations have been done in Mannheim, at CEA Saclay, and at CERN, Geneva.

In contrast to the ATLAS detector design, the ATLAS HLT, as part of the overall ATLAS computing project, is a dynamic and changing project. This is the result of the aim being as cost-effective as possible. The detector itself is a huge hardware installation, and has been planned since the 1980's. The building of the ATLAS detector includes a lot of work that is done by external commissioners, and follows an elaborate time schedule. Nearly all parts of the detector are designed and built for this single purpose and are therefore unique.

However, the ATLAS computing has to follow a different approach. The fast evolution of computing hard- and software mandates a constant effort to be as close to the "mainstream" as possible since not following the development of computing technology would be a waste of resources.

This can easily be seen by looking at the development of mainstream processor prices and performance: The ratio between performance and cost of processing power rises as quick as delaying the purchase of some fixed processing power by a year would drop the cost by at least one third.

That is why the Mannheim FPGA group was challenged by three major HLT frameworks in the past four years. These are the "ATLAS High Level Trigger ATM Testbed", the "Second Level Trigger Reference Software", and finally the current "High Level Trigger and Data Acquisition" framework.

ATLAS High Level Trigger Demonstrator and ATM Testbed

Until 2001 a group from CEA/Saclay continued contributing to the ATLAS high level trigger (HLT) [DCG⁺99]. Denis Calvet, Irakli Mandjavidce, Patrik Le Dû and others collaborated with my group in studies regarding the integration of FPGA based processors and coprocessors built by the Mannheim FPGA group into the ATLAS HLT frameworks. Unfortunately, the Saclay group left the ATLAS HLT.

The main inputs from the Saclay group to the ATLAS HLT were the investigation of asynchronous transfer mode (ATM) technology for the central HLT network

and the early construction of a prototype including software and hardware which was used for performance and feasibility tests. In the following I will refer to this software as the “Saclay software”.

The complete hardware and software system is called “ATLAS HLT ATM Testbed” or, because it is a consolidation of two earlier architectures, “Demonstrators A + C”. Its physical architecture has a lot in common with the HLT and data acquisition (DAQ) architecture that is illustrated in figure 2.9. In difference to the current TDAQ architecture the ATM testbed does not consider the event filter as a distinguishable component. Additionally, the physical setup consists of a central ATM switch instead of Ethernet switching technology for the central data-flow carrying network. Another Ethernet network is used for configuration of the testbed.

ATM is a connection oriented networking technology that is commonly used in the telecommunication and telephone industry. ATM is designed for providing quality-of-service (QoS) networking properties, i.e. a given connection is guaranteed to deliver data at a minimum throughput rate and within a minimum latency. In contrast, Ethernet is a switched and packet oriented technology that does not guarantee QoS. Ethernet is also much cheaper due to its widespread use, and it is easier to find software and hardware experts supporting Ethernet.

The use of ATM networking in the testbed includes ATM network interface cards (NIC) in the data sources and destinations, i.e. the read out buffer to switch interfaces, the supervisors, and the processing nodes.

The Saclay software and the second level trigger reference software (T2REF) differ in many aspects. The Saclay software is coded in “C” and has a leaner structure, however omitting some features that T2REF offers. Thus, the software is more portable and can execute on more exotic hardware like VME processors. In contrast, T2REF has a full object-oriented architecture implemented in C++, is very feature-rich, and restricted to PC and workstation hosts. The Saclay group reported impressive performance measurements for latency, rate and throughput of data transfer in distributed HLT testbeds.

Second Level Trigger Reference Software

The second level trigger reference software (T2REF) is a framework that abstracts most of the concepts in the HLT [Hau00]. T2REF was used in the ATLAS level 2 pilot project to demonstrate a trigger prototype with most of the interfaces to external systems [B⁺02].

The objectives in T2REF’s design are:

- Unification of the approaches of different groups in the HLT development leading to a common code base and enabling code reuse.
- Operating system independence and ability to run on the two major off-the-shelf operating systems, Linux and MS Windows NT.
- Run on commodity hardware (Personal Computers, Workstations).

- Support for the study of different networking technologies.
- Support the the development and benchmarking of physics algorithms for the LVL2.
- Self containment; no dependency on external hard- or software.
- Modular architecture. Give the different groups the possibility for easy adaption to their specific needs.
- Provide software emulators for hardware components not available, e.g. the supervisor or the ROB's.

T2REF is build using a modular structure to achieve these objectives. The modules are collected in packages, and each package is managed by a responsible person. The modules are designed in a way eliminating circular dependencies and thus enabling a layered software architecture.

Different stable interfaces are provided by T2REF to support the application developer. Among these are interfaces for error reporting and logging, managing threads, configuration, timing, message passing (networking), and application framework.

On top of T2REF's application framework T2REF provides skeleton applications for the supervisor, the ROB, and the steering, which also makes use of the physics package. The physics package provides an interface for building physics algorithms which can be used for testing physics feature extraction in the T2REF framework.

The T2REF framework supports several system-architectures:

- Single node feature extraction
- Single node trigger
- Multi node with skeleton applications
- Multi node with algorithms

ATLAS Level-2 Pilot Project Testbeds

Besides different setups in small networks and a lot of single node installations, both also in Mannheim, the two HLT approaches, CEA Saclay ATM Demonstrator and commodity Ethernet with T2REF, have been installed at CERN in a common effort during a combined test. In this common installation the same computing nodes, some 25 commodity PC's, have been used for both networking technologies and software frameworks. The ATM testbed additionally incorporated ten Power-PC single-board computers running LynxOS and, additionally to the Ethernet network, a 48-port ATM switch. Both systems have been using software emulators for the ROB's and the supervisors. At that times Gigabit Ethernet was starting to

become available and thus the four Ethernet switches were partly equipped with Gigabit (1000 MBit) Ethernet and partly with the more common Fast (100 MBit) Ethernet.

ATLAS High Level Trigger and Data Acquisition

The current view of the architecture of the ATLAS high level trigger and data acquisition is illustrated in figure 2.9. Solid line arrows illustrate the event data flow. Event data is transported from the detector at the top to the offline analysis at the bottom. The data-flow from LVL1 to the supervisor, and finally the HLT network contains LVL1 decisions and region of interest pointers, indicated by dotted arrows. The stacked boxes indicate a, possibly large, number of identical subsystems. The numbers on the right indicate the event rate at the respective subsystem boundaries.

Second Level Trigger Processing Node

The second level trigger processing node, or PU for Processing Unit, is the workhorse for the second level trigger algorithms. The HLT design is flexible regarding the number of PU's, however it is assumed that the number of PU's is in the order of 1000, however . The PU's will be commodity PC's, possibly with SMP architecture, running the Linux OS. Every PU gets assigned a single event from one of the level 2 supervisors (LVL2SV) in a round-robin manner. The application on the PU will be multi-threaded, with every thread working on a single event.

Upon receiving the event ID and occasional pointers to Region Of Interest (ROI) from the LVL2SV the PU will request event fragments from the readout buffers on which the physics algorithms work in a sequential manner to verify the LVL1 trigger decision. Sequential selection is chosen to ensure that only such fragments are requested that are needed to make a final LVL2 trigger decision thus limiting the load on the HLT network and the readout buffers. The final LVL2 trigger decision will be send back to the LVL2SV which forwards positive decisions to the EF. Negative decisions will cause the LVL2SV to request the readout buffers to discard the stored event fragments.

The workload on the PU's will mainly be caused by the physics algorithms. It is assumed that some of the algorithms can benefit from acceleration by reconfigurable coprocessors, giving the possibility to perform more advanced algorithms, to reduce the number of PU's and to reduce the latency in the HLT.

It is assumed that the PU's are connected with cascading switches to the central HLT switch. The reason is to reduce the number of ports on the central Gigabit Ethernet switch. It is assumed that the data rate between a single PU and the HLT network is in the order of 100 Mbits/s such that grouping about 10 PU's at a cascading switch would match the bandwidth of a port on the central switch.

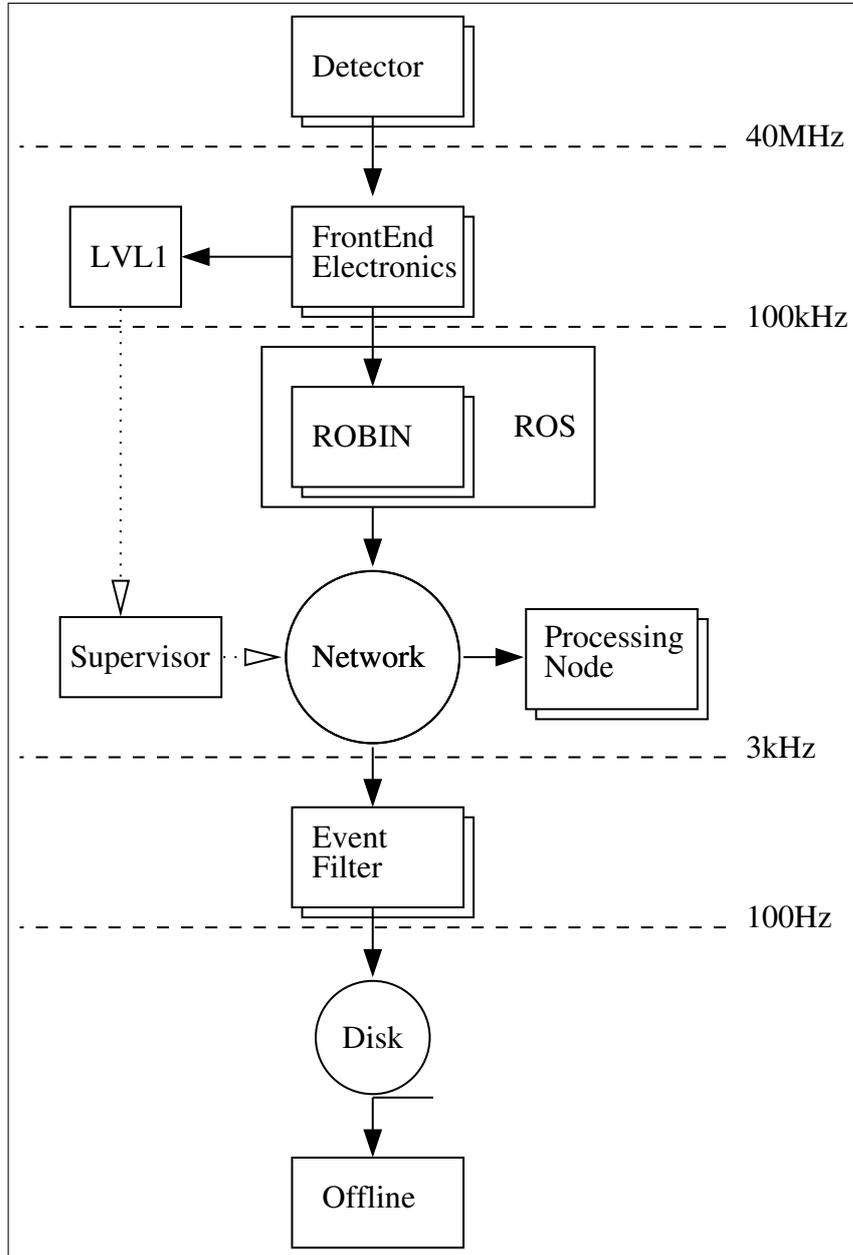


Figure 2.9: HLT and DAQ

2.3 Computing and Reconfigurable Systems

2.3.1 Conventional Computing

Conventional computing is based on architectures like the sequential von-Neumann-architecture. These architectures separate memory units that store data and algorithms, input / output units for communicating with the outside, and control and arithmetic- units, which actually execute the algorithms on the data. Control and arithmetic units are usually integrated in a central processing unit (CPU). The CPU executes algorithms (programs) by sequentially applying statements fetched from memory on data that is also loaded from memory and stored back to memory upon completion of computation. Generally, the algorithm (program) can be altered dynamically, e.g. for using the same machine for different problems or to use improved algorithms. Such an architecture, though being very flexible, shows two major drawbacks. First, data and statements have to be moved across the interface between memory and CPU, which is of limited capacity. Accordingly, the performance of data-intensive algorithms is limited by the available memory bandwidth. Second, the sequential processing of CPU's limits the performance of complex algorithms and inefficiencies caused by the general purpose architecture of CPU's, which render parts of the CPU useless that are not needed by the actually executed algorithm and also limit the efficiency of the active (used) parts of the CPU.

The aforementioned performance limitations caused by the memory bottleneck and sequential processing are mainly addressed by modification or extensions to the simple van Neumann architecture. The memory bottleneck is relaxed by intermediate stages of fast memory units *caching* data on its way between main memory and CPU. Such memory caches are usually located within or nearby the CPU, reducing the load on the external interface to main memory. *Pipelining* is addressing the problem of inefficient utilisation of functional units of the CPU by automatic reordering and decomposing of complex statement sequences in a way that independent statements can be executed in parallel in different functional units. Specialised CPU's, like vector processors, digital signal processors (DSP), or network and I / O processors, are optimised to address specific computational needs.

Other approaches exploit possible parallelisations of computational problems. The simplest approach being the use of multiple computers in a *farm* working in parallel. This approach requires that the problem can be divided into totally independent tasks. In general the computational throughput of a farm is proportional to the number of farm computers (nodes). However, additional problems may arise with the assignment of tasks to the individual nodes, the transfer of data to and from the nodes, and the management of the nodes. The ATLAS high level trigger uses a farm with some hundred nodes to perform physics algorithms on detector data.

Clusters use tightly coupled computers to work on problems that can only partly be decomposed into subtasks. A typical problem from this category is computational fluid dynamics, where significant amount of computing can be done

independently on the cluster nodes before part of the results have to be communicated between neighbouring nodes. Clusters impose high demands on inter node communication latency and bandwidth which are usually satisfied with specialised high performance networking technologies like the commercial scalable coherent interface (SCI) or Myrinet, or the ATOLL [DSB03] research project.

Multiprocessor (MP) computers use several processors in a single system. Symmetric multiprocessor (SMP) systems and non uniform memory architecture (NUMA) systems represent the main architectures for MP computers. SMP systems have equal latency and bandwidth between all CPU's and every memory location whereas NUMA systems have lower latency and higher bandwidth between associated CPU's and memory units. SMP systems are usually restricted to some ten CPU's, whereas NUMA systems contain up to hundreds of CPU's. MP systems run a single instance of an operating system that manages the whole system. On the contrary, every node of a farm or a cluster runs its own OS instance.

2.3.2 Custom Computing

Custom computing machines (CCM) are used to satisfy specific computational needs that are difficult to meet with conventional computing systems. For such CCM's the performance overhead caused by the generality and universality of computers based on the von Neumann architecture is traded against more powerful or cost-effective but less versatile specialised solutions.

CCM's can be more effective because their architecture is tailored to a specific problem. This includes I / O paths which are designed to match the data sources and drains in terms of throughput and data format, internal resources like memories, power supplies and other supporting components, and the design of the computational core(s). The tailoring requires a specific system layout and specialised electronic devices. Such devices are often implemented as application specific integrated circuits (ASIC).

The implementation of an ASIC is derived from a structured high level description using an hardware description language (HDL), often incorporating and reusing macro cells provided by intellectual property (IP) vendors. IP cells define independent building blocks like processor cores, memories, communication components, and others. After verification this description is stepwise transformed into the descriptions of several masks that are used to manufacture the ASIC in a standard semiconductor process. The topology of an ASIC, i.e. its structural composition from transistors, transmission lines, capacities, and resistors, that determines its behaviour, is defined by the HDL description and the manufacturing process and can't be altered after production. The development of ASIC's require an high effort, often dominated by verification, and high initial production cost caused by mask manufacturing. These costs only drop with mass production of the ASIC. However, CCM's are often low volume systems, which makes the use of ASIC's a cost intensive solution. Another problem with ASIC's is long time-to-market causing a long timescale for the development of an ASIC based CCM.

2.3.3 Reconfigurable Devices and Systems

The use of ASIC's in low volume systems is often prevented by two aspects : high effort needed for development, especially for verification, and high initial production cost. *Field programmable gate arrays* (FPGA) provide means for *emulation* of arbitrary logic circuits at a fine grained level. The emulated logic can be fairly complex and can be changed after FPGA production by *configuring* the FPGA. The hardware emulation realized in FPGA's is significantly speeding up ASIC design verification as compared to *simulation* of logic with conventional computers. Since logic emulation shows in principal the same behaviour, yet with reduced performance, like the actual execution (e.g. in an ASIC), FPGA's can sometimes replace ASIC's. This comes at the benefit of reduced system costs because FPGA's don't require mask production. Consequently, FPGA's are used in low volume systems and prototype systems replacing ASIC's, and in logic emulation systems for emulating ASIC's.

The ability for logic emulation of FPGA's is realized by an array of configurable logic blocks (CLB) (naming may differ depending on FPGA vendor). The CLB array is accompanied by an array of input / output blocks (IOB) which connect the FPGA to other devices in a system. The architecture of CLB's is dependent on FPGA vendor and family, but mostly based on a look up tables (LUT), calculating an output value from some inputs. The LUT's are complemented by registers and other logic, e.g. for implementing fast carry bit chains. A configurable routing network connects CLB's and IOB's. On most FPGA's the routing network is hierarchically composed of short overlapping networks connecting neighbouring CLB's and IOB's, and long routes for connecting distant parts of the FPGA.

Modern FPGA's may complement the logic resources of their CLB's with dedicated components like RAM blocks, multipliers, and even processor cores. The distribution network for clock signals and components for clock conditioning and synthesis are usually separated from general purpose logic and routing, but also configurable.

The configuration of an FPGA defines the contents of the LUT's of the CLB's, multiplexer selects, configuration of IOB's and clocking components, the topology of the routing and clocking networks, and other components, e.g. the contents of embedded RAM blocks. The configuration of an FPGA is volatile, i.e. the configuration is lost if the supply voltage is removed from the device. Generally, FPGA's allow for the upload of configuration data (programming) through their JTAG interface (see below) which is simple but slow. Most FPGA's provide also a fast, e.g. 8-bit parallel, interface, reducing configuration latency down to some 10 milliseconds. The size of configuration data depends on the complexity and size of the FPGA reaching some 10 MBit for modern and big FPGA's.

Programmable logic devices (PLD) represent another class of configurable logic devices. PLD's are similar to FPGA's in the sense that they can be configured to show arbitrary behaviour based on low level digital logic. However, their intended use and environment differs from that of FPGA's, although the use cases

Table 2.2: Comparison between PLD's and FPGA's

	PLD	FPGA
User IO pins	30-300	60-1000
Gates	1k-10k	5k-10.000k
Configuration speed	approx. 5min	approx. 1sec
Configuration storage	persistent	volatile
Reconfiguration	limited	unlimited
Timing predictability	accurate	poor
Price	low	medium-high

for FPGA's and PLD's may overlap. PLD's are mostly of much reduced complexity as compared to FPGA's, generally omitting high level specialised components like memories and others. The desired logic of an configured PLD is e.g. realized by a set of wired OR's whose inputs and outputs are connected with a regular mesh of traces. The crossovers of the mesh can be configured with techniques like that used for programming EEPROM's. The resulting topology, and therefore the implemented logic, is permanent and not lost if the supply voltage is removed. However, according to the technique used, configuration latency is long, reaching some 5 minutes. The nonvolatile configuration is accompanied by the ability to exactly predict the timing behaviour from the configuration, and the generally better (faster) timing as compared to FPGA's. Also, PLD's can show very low power consumption, making them suitable for power critical environments.

Accordingly, PLD's are mainly used for system integration purposes, where the required computational complexity is low and requirements for timing are tight.

Table 2.2 compares key properties of current PLD's and FPGA's offered by Xilinx².

Both, PLD and FPGA, provide configurable logic and configurable routing. The routing connects between internal logic and IO connections of the device. Some reconfigurable devices, Field Programmable Interconnect Components (FPIC), do not have significant logic resources; their routing capabilities are supposed for providing a configurable connection matrix between their IO pins. In most applications FPIC's can be replaced by PLD's or FPGA's, however, FPIC's potentially provide better performance.

Reconfigurable systems can be classified according to their coupling to a host [CH02, GG95]. The weakest coupling is present in *reconfigurable processors*, standalone reconfigurable systems, that are only occasionally connected to a controlling host e.g. for system configuration or monitoring. Here, the reconfigurable processor is often implemented as a data driven processor, connecting to external systems with dedicated inputs and outputs for data transfer. The controlling host

²Xilinx, <http://www.xilinx.com>

is possibly not part of the data path of the computational system. The Enable++ FCCM (see 2.4.1) is an example for a reconfigurable processor used in implementation studies of the ATLAS HLT.

A great variety of systems exists for the class of *reconfigurable coprocessors* (RC) that are attached to a controlling host by the hosts system bus, in many practical cases a PCI bus. Most RC's use the the same interface for data transfers and control, some offer additional interfaces for data transfer to other systems. The strong coupling between RC and controlling host renders systems possible that partition a computational task between host and RC, therefore transferring significant amount of data between both participants. The performance of such a distributed system is often influenced by latency and throughput of the interface between RC and host. RC's are usually smaller as compared to reconfigurable processors. Examples for RC's are the μ Enable (see 2.4.2) and the MPRACE (see 2.4.4).

The Pilchard reconfigurable unit is connected to a host by the host memory bus [L⁺01]. Even stronger coupling can be achieved by directly attaching reconfigurable logic to a processor in the traditional "coprocessor" manner, or by embedding a reconfigurable functional unit into the hosts processor, e.g. for dynamically extending its instruction set.

In general, tighter coupling implies less communication overhead and higher bandwidth between host processor and reconfigurable logic, while looser coupling gives greater flexibility and higher execution parallelism between reconfigurable logic and host CPU [Hau98a],[Hau98b], .

2.3.4 Programming of Reconfigurable Systems

Creating applications for reconfigurable systems requires several steps. For systems that are attached to a controlling host that also takes part in computation, the application has to be *partitioned* between host and reconfigurable units. Such partitioning is guided by the feasibility of subtasks for the different components, CPU's and FPGA's, in terms of performance and implementation cost. Restrictions for the partitioning result from limited available resources of reconfigurable logic and from tradeoffs generated by data transfers between the host and the reconfigurable system. A further partitioning step is required for multi-FPGA systems, where the desired logic has to be distributed across the different FPGA's according to the architecture of the reconfigurable system.

Reconfigurable logic can be described at a very low level through hand mapping of the basic functional blocks (e.g. CLB's) to compose the desired logic. However, such a description is costly and only reasonable if the circuitry is small and has high performance requirements.

Hardware definition languages (HDL) like VHDL and Verilog facilitate a structural description that may resemble a hand mapping approach like above. However, the description given in an HDL is usually more abstract. Tools (compilers) are used to translate an HDL description into a format specific to Vendor and FPGA. Additional to FPGA-specific functional blocks an HDL description may use more

elaborate units like multipliers and adders which are provided as library elements by FPGA and tool Vendors. Commercial available intellectual property (IP) macros may provide complex units like communication protocol converters or CPU's.

HDL's offer behavioural description besides structural description of logic. Behavioural description resembles traditional software in that it focuses on the definition of states, events, actions and control flows. Behavioural description tends to be more general and abstract and is closer to natural language problem description. Such description, while being more convenient for human developers, shows the drawback of potentially worse performance in terms of circuit timing and resource consumption as compared to pure structural description.

HDL's have been developed for simulation and compilation of hardware descriptions. Their grammars and features are often seen as an hindrance for software developers to efficiently develop reconfigurable logic [Com99]. Many programming systems have been proposed, some of them being actively used, that use traditional software languages like C++ and Java for reconfigurable hardware description. Most of them offer libraries for efficient creation of structural logic descriptions in the context of their language. CHDL [K⁺98] JBits [GLS99] are examples of library based programming systems. Another class of systems adapts or restricts the grammar of traditional programming languages (mostly C), e.g. by annotations, to be used for behavioural logic description. SystemC [Sys] and HandelC³ [Cha01] represent such languages.

Synthesis tools translate behavioural or structural descriptions to Register Transfer Level (RTL) descriptions which hold a low level definition of the desired logic. Vendor specific tools are used to convert RTL descriptions into configuration files which can be uploaded to according configurable devices.

2.3.5 JTAG

The Joint Testing Architecture Group (JTAG) is an association of hardware vendors, that standardise a common testing interface and infrastructure for electronic devices. JTAG infrastructure is present mostly on devices with many IO pads and that are used as components for complex systems, e.g. RAM's, CPU's, PLD's, and FPGA's. JTAG is an IEEE standard [IEE]. Using JTAG the internal functionality of a device can be disconnected from the devices IO pads and the logic levels of the pads can be set and retrieved in an arbitrary manner. This technique, called "boundary scan", can be used to test complete systems (boards assembled from one or more JTAG-capable devices) for the correctness of the on board traces. M. Müller⁴ used boundary scan to test the ACB (the ACB is described in 2.4.3).

JTAG defines a standard set of JTAG-commands that is supported by all JTAG-capable devices. The instruction set can be extended with special commands. Most PLD and FPGA vendors use special JTAG commands to support the programming of their devices. Some FPGA's support configuration and status read-back with

³Celoxica

⁴M. Müller, mmueller@ti.uni-mannheim.de, University of Mannheim

Table 2.3: JTAG Signals

short name	in/out	long name	description
TCK	in	test clock	TMS and TDI are sampled at rising edge, TDO is valid at falling edge.
TMS	in	test mode select	Steps through the JTAG state machine.
TDI	in	test data in	Commands, (configuration-) data.
TDO	out	test data out	Register values, identification, read-back.

JTAG, a technique that can be used for in-system debugging of the devices functionality or e.g. for scheduling of FPGA configurations [Sim01].

JTAG defines a JTAG-port consisting of four signals as summarised in table 2.3. JTAG uses bit-serial protocols on the TMS, TDI and TDO signals to step through the JTAG-controller state machine and to read and write commands and data. The standard allows for the connection JTAG ports in different topologies. An exemplary configuration would be to arrange an arbitrary number of JTAG ports in a chain by connecting their TDI and TDO signals pairwise, making all devices accessible through a single, four-signal JTAG port on the system.

2.4 FPGA Processors and Coprocessors Built in Mannheim

Some ten vendors worldwide (e.g. Aptix, Celoxica, Nallatech, SiliconSoftware, and many others) sell FPGA processors and coprocessors with a variety of architectures, the most of these being PCI coupled FPGA coprocessors. All these commercial FPGA coprocessors, except SiliconSoftware's μ Enable, are not covered in this thesis because they come with their own closed and proprietary control software.

During the last decade the FPGA group at the University of Mannheim has developed a variety of reconfigurable processors, starting from big standalone FPGA processors, Enable and its successor Enable++, up to recent smaller FPGA coprocessors like the versatile MPRACE and the Robin, that combines FPGA, CPU, and fast network I/O's. Table 2.4 gives an survey of these systems. The table shows in chronological order the different systems, along with small pictures. The table is divided in an upper part, containing the older FPGA processors that have been developed in the mid and late 1990's, and a lower part, listing the newer and smaller FPGA coprocessors that supercede their predecessors. The technology used for coupling to a controlling host, the number of FPGA's, and an estimation of the number of networks on the board is given for every system. The table shows the

general trend to lower FPGA count and tighter host coupling. This trend reflects the increasing density and computational power of modern FPGA's and the resulting demand for higher I/O bandwidth. As indicated by the numbers of networks the step from processors to coprocessors involves a significant reduction in complexity of the boards, although a bridge for interfacing to the host bus is added to all coprocessors.

2.4.1 Enable++

Enable++ and its predecessor Enable are FPGA-based processors designed to provide the computing power for real-time pattern recognition in the ATLAS second-level trigger [R⁺93, HKL⁺95, NZK⁺95, Ses96, Nof96, Lud98]. Enable was the only system that met a benchmark that was set up in 1991 by the "Architectures for Second-Level Triggering" (EAST RD/11) collaboration.

The modular Enable++ system consists of IO-boards connecting to the trigger with e.g. S-Link, the very high bandwidth active backplane, and the Matrix (MX) boards, which provide the computing power for the real-time pattern recognition. The system is configured and controlled from a workstation by a serial link connection, that is routed through local controller modules attached to all system boards. The Enable++ MX board uses 16 Xilinx 4013 FPGA's that provide the computational resources. The FPGA's are statically connected to each other in a ring topology and additionally to a programmable crossbar switch, realized by an array of FPIC's. The crossbar switch also connects the computational FPGA matrix to the backplane for communication with the IO board(s).

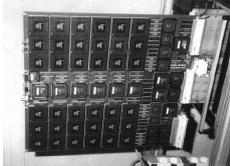
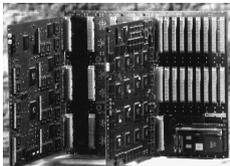
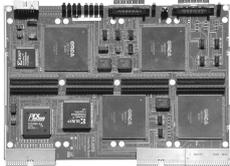
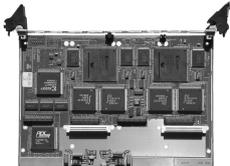
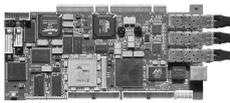
The experience gained from Enable and Enable++ was the basis for the development of the Atlantis system [KKL⁺98c, KKL⁺98a, KKL⁺98b].

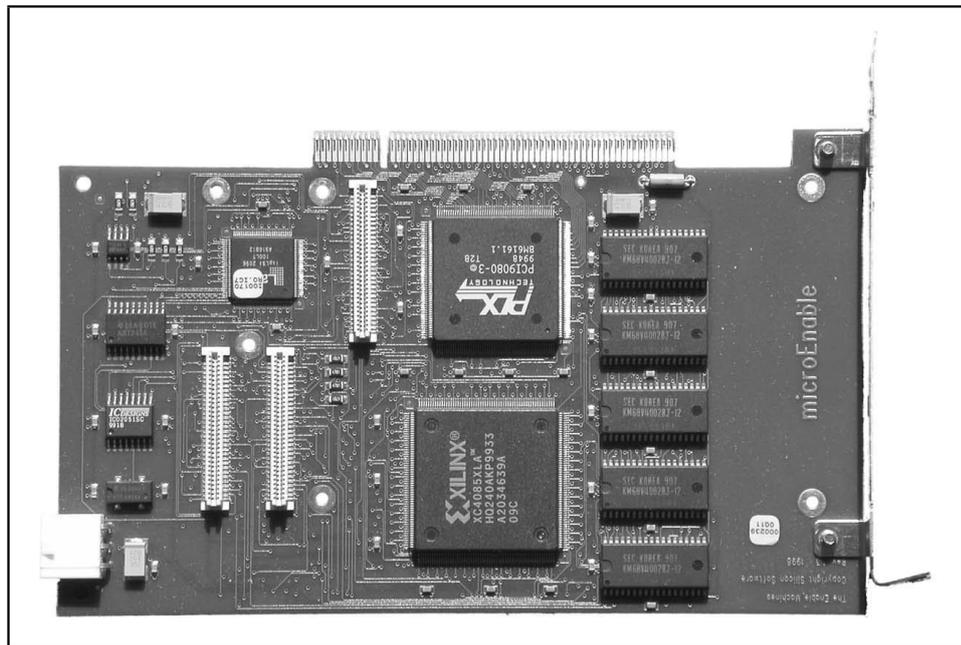
2.4.2 μ Enable and μ Enable2

The μ Enable (figure 2.10) was the the first FPGA coprocessor we have been working with [BDK⁺98]. μ Enable was developed at the University of Mannheim and is now a commercial product sold by Silicon Software (SiSo) [Sof], which is a spin-off company of the University. μ Enable uses a PLX9080 PCI bridge for connecting to the controlling host. The computational resources are provided by a Xilinx 4000 series FPGA to which an array of static RAM modules is attached. It also provides a set of connectors that can be used to interface to mezzanine cards, e.g. realizing S-Link connections. The μ Enable has been used for a variety of applications like frame grabbers, image recognition, encryption and number crunching. The μ Enable2 is an enhanced version of the μ Enable and is equipped with a Xilinx Virtex FPGA.

2.4. FPGA PROCESSORS AND COPROCESSORS BUILT IN MANNHEIM 61

Table 2.4: Evolution of FPGA Processors and Coprocessors

year	name	host conn.	FPGA	networks (approx.)	picture
1995	Enable	VME	36	3500	
1997	Enable++ MX	serial	12	4000	
1997	μ Enable	PCI 32/33	1	400	
1999	ACB	PCI 32/33	4	1800	
2000	AIB	PCI 32/33	2	800	
2001	MPRACE	PCI 64/66	1	970	
2003	ROBIN	PCI 64/66	1	1150	

Figure 2.10: μ Enable

2.4.3 Atlantis, ACB and AIB

The Atlantis system [BHH⁺00, KHM⁺00] is based upon a 19-Inch crate with a compact-PCI (cPCI) backplane and a cPCI computer board. The crate accepts up to 6 cPCI boards. Two reconfigurable coprocessors have been developed by H.Singpiel that fit into the cPCI backplane, completing the Atlantis system: The Atlantis Computing Board (ACB) (fig. 2.11) and the Atlantis Input Output Board (AIB) (figure 2.13). The backplane can be extended by the configurable high bandwidth active backplane (ATB), connecting ACB's and AIB's, that was also developed by H. Singpiel. The Atlantis system was designed to be a substitute and enhancement for the outdated Enable++ FPGA processor. Thus, the main anticipated application for the Atlantis system was the real-time pattern recognition in the ATLAS trigger using the TRT-LUT-Hough algorithm (see section 2.2.2).

The ACB is designed to provide high computational power with the possibility of extension through a set of very high bandwidth connectors. The ACB is rather complex and consists of:

- 4 FPGA's
- 2 PLD's
- 2 clock generators
- 3 programmable clock buffers

2.4. FPGA PROCESSORS AND COPROCESSORS BUILT IN MANNHEIM 63

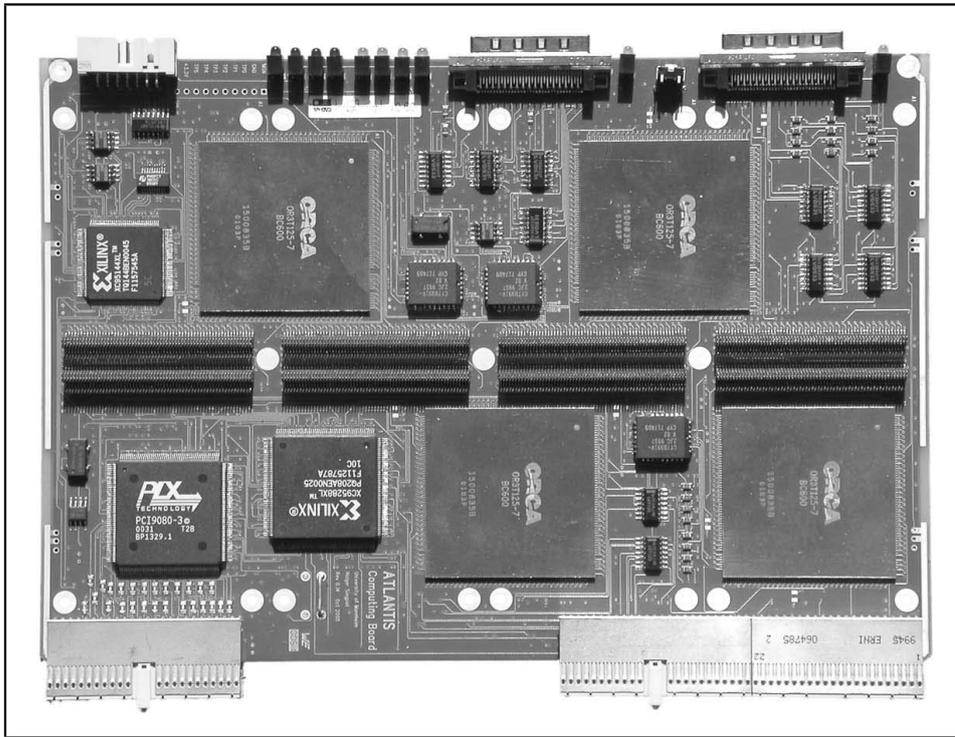


Figure 2.11: Atlantis Computing Board (ACB)

- PCI bridge and configuration EEPROM
- 4 module ports and an external connector

This board is based on Compact-PCI and has four ORCA ORT125 FPGA and an array of four connectors with a total pin-count of about 800. These connectors are supposed to interface each FPGA to a mezzanine expansion module. One application for the modules is an array of four memory modules with 11 MB static RAM each. Each of the Modules has a word length of 180 bits and an address range of 19 bits or 512K words. The word width and address range of all four modules together is 720 bits x 512K. The memory modules are designed to hold the look-up-table (LUT) for the TRT-LUT-Hough algorithm. Another application that has been realized for the expansion module is the Volume Graphics Engine (VGE) [CVHM03]. The VGE was developed to provide real-time rendering of volume data. It makes use of another FPGA device together with dynamic RAM (DRAM) for holding the volume data.

The ACB was developed together with the Atlantis Input Output Board (AIB) (figure 2.13) and the active backplane (ATB). The AIB is based on Compact-PCI too and is equipped with two Xilinx Virtex XCV600 FPGA. The main use of the AIB is as an Input/Output component, occasionally in a system with several other AIB's or ACB's. Communication between these boards can be done through the

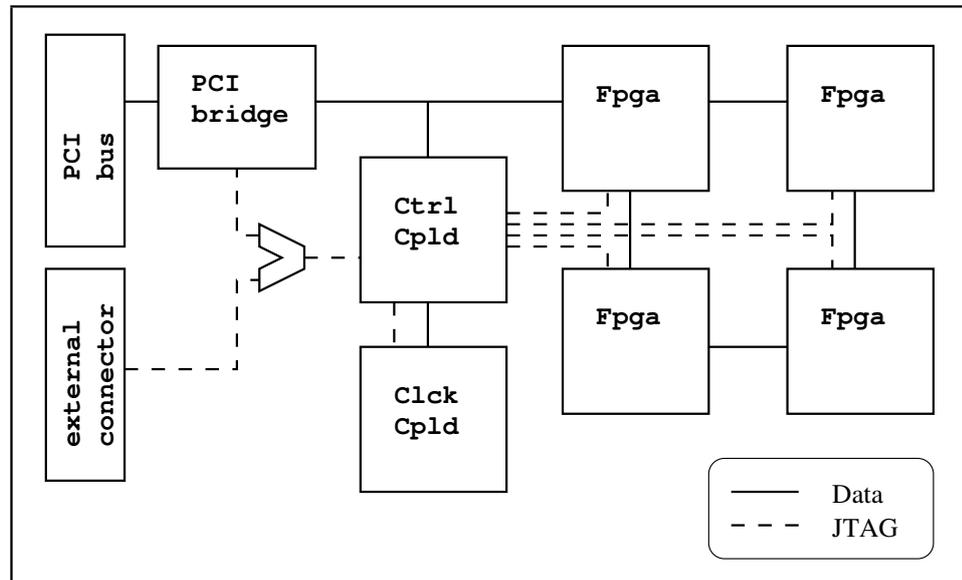


Figure 2.12: Data Paths and JTAG on the ACB

ATB programmable cPCI backplane or the system PCI bus. Four mezzanine sockets are provided on the AIB to plug in e.g. S-Link or Ethernet adapters.

2.4.4 MPRACE

MPRACE is the actual general-purpose PCI-based reconfigurable coprocessor (figure 2.14) [Kug02]. It is build around a Xilinx XC2V3000 or XC2V6000 FPGA and connects to the host with a 64 bit / 66 MHz PCI bridge. A XC95288 PLD is used for integration and system control. The architecture of the MPRACE resembles an update to the architecture of the μ Enable. MPRACE provides local memory through 6 SRAM modules and has an additional socket accepting an SDRAM memory module.

2.4.5 ROBIN

The ROBIN (figure 2.15) is a special development for use in the ATLAS experiment. Build around a Xilinx XC2V1500 FPGA and an IBM Power-PC microprocessor, the ROBIN (Read-Out Buffer INput) is used as a fast configurable buffer for detector data between the detector read-out electronics and the ATLAS high level trigger and data acquisition [GJP⁺03]. It interfaces to the read-out with S-Link connections and provides an additional Gigabit Ethernet network interface. The ROBIN uses the same 64 bit / 66 MHz PCI bridge as the MPRACE for interfacing to the host.

2.4. FPGA PROCESSORS AND COPROCESSORS BUILT IN MANNHEIM 65

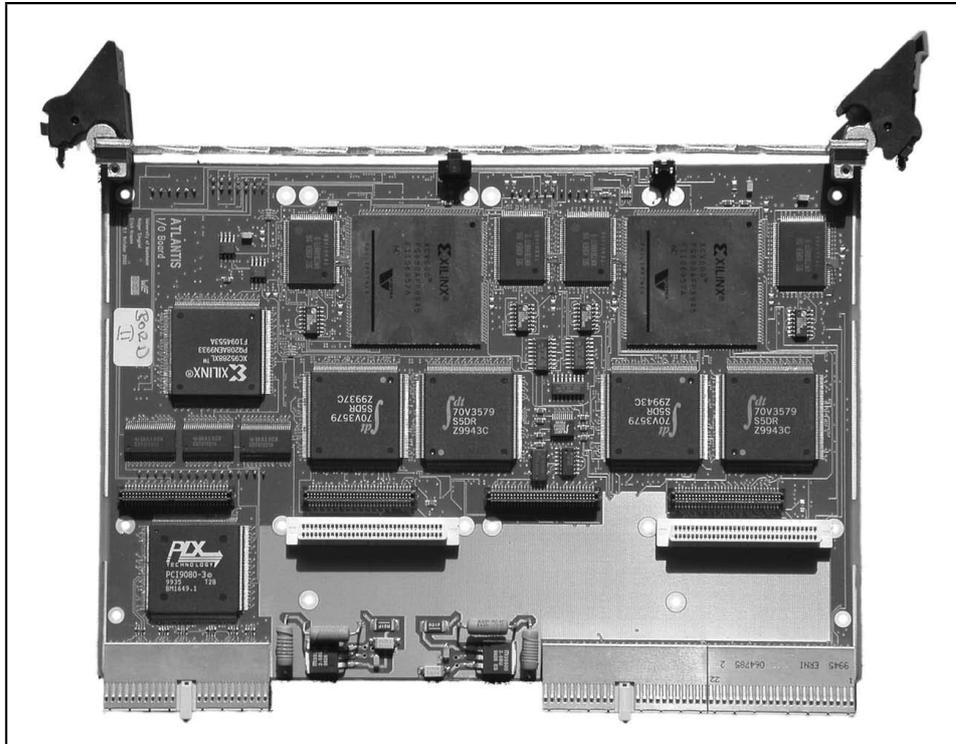


Figure 2.13: Atlantis Input Output (AIB)

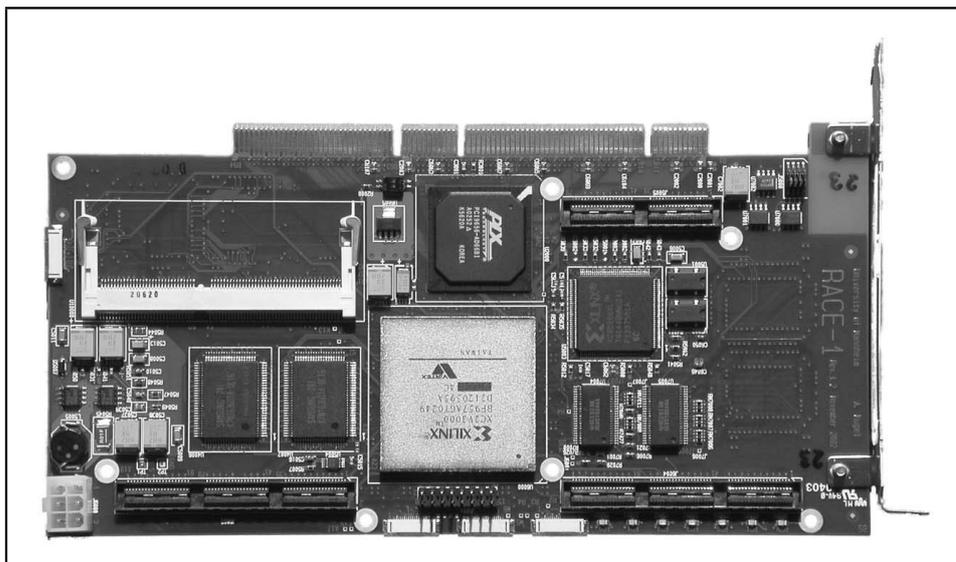


Figure 2.14: MPRACE

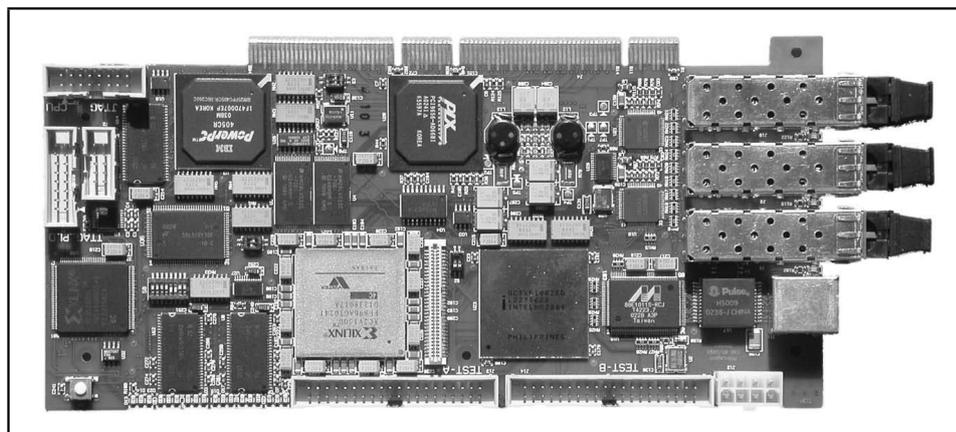


Figure 2.15: ROBIN

2.5 Host and Software Environment

2.5.1 Introduction

In the context of this work reconfigurable hardware (RH) is used in conjunction and under control of hosting computer systems. The general structure of such hosts is described in the following sections. Details are given in areas where the coupling with RH makes these desirable. The description of the host platform covers hardware and software.

2.5.2 Host Architecture

The control software developed in this work supports a variety of FPGA-based coprocessors, all of them attached to a host by Peripheral Component Interconnect (PCI) buses [PCI, SA99].

Although the host architecture would not limit the use of the coprocessors we have been solely concentrating on IA32 and IA64 compatible desktop and server systems. Most of this systems are uni-processor (UP) IA32 [Inta] systems with Intel Pentium like CPU's, recently enhanced to dual or quad Symmetric Multiprocessor (SMP) systems equipped with Intel Xeon processors. Tests have been done with Intel IA32 compatible AMD processors. Additionally we have been recently using a dual Itanium [Intb] workstation.

The SMP systems have a Uniform Memory Access (UMA) architecture, i.e. all CPU's have access to all system memory with equal latency and priority. The memory architecture of all systems (UP and SMP) has two or three levels of caches, speeding up memory access, which still often is a bottleneck for High Performance Throughput (HPT) applications. The memory caches are located on the CPU die or are attached to the CPU on CPUcards. Memory is attached to the system by a system controller, which also mediates access to the system bus, often using an-

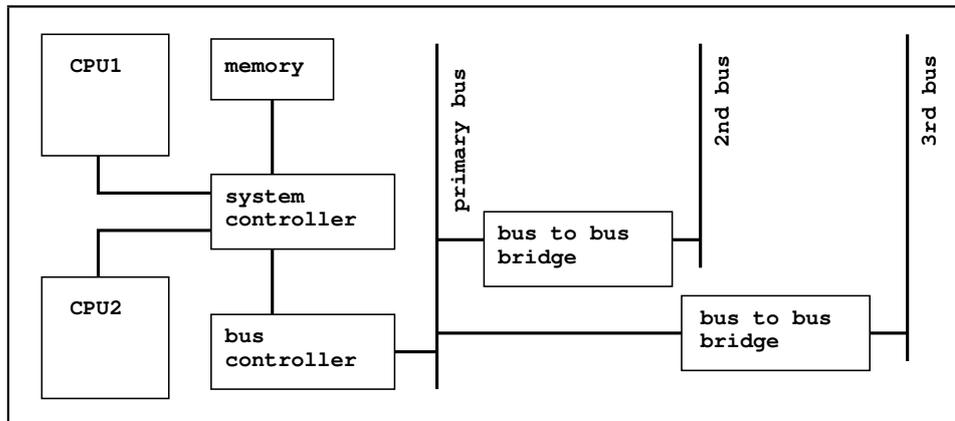


Figure 2.16: Host Architecture

other intermediate bus controller/bridge, refer to figure 2.16. Some of the systems have more than one PCI system bus. Secondary buses are usually connected in a tree-like structure to a primary bus. Earlier measurements show, that the design of such a bus hierarchy can have a great influence on data transfer throughput between CPU and the coprocessors [Sin00].

2.5.3 IA32 and IA64

The Intel architecture 32 (IA32) is widely accepted as the Personal Computer (PC) and workstation standard, enabling low prices, good price/performance ratio, and is evolving fast in terms of computing power. This wide acceptance and deep market penetration makes IA32 the only architecture that is currently actively evaluated for the second level trigger in ATLAS. IA32 is a 32 bit complex instruction set computing (CISC) architecture implying an address space of 2^{32} byte = 4 GiB . Although modern IA32 compatible CPU's (like Intel Xeon) can in principle address more (e.g. 64 GiB) memory, this addressing can't be included seamless into operating systems and applications and has to use segmentation-like techniques. Consequently the address space that is available to a single process is limited to 4 GiB, and this address space has to be shared between memory and IO buses, usually limiting the available memory space per process to 1 or 2 GiB.

Many 64bit processor architectures exist, that natively overcome the 32bit address space limitations, the Intel architecture 64 (IA64) being just one of them, although a modern one. The main features of IA64 are:

- native 64 bit addressing
- fast, extended precision floating point arithmetic
- Explicit Parallel Instruction set Computation (EPIC) paradigm
- IA32 compatible subsystem

Of these features native 64 bit addressing and the EPIC paradigm are important in the scope of this work. 64bit addressing concerns since it demands reviewing and refining of the control software and Linux driver sources and also implies the use of a different operating system, Linux on IA64, together with its GNU tool-chain and utilities. IA64 and the porting of Linux to IA64 is described in great detail in [ME02].

EPIC is important since its demands on the tool-chain, especially the compiler. Linux on IA64 requires GNU GCC-3.x compilers, older compilers do not support IA64.

The basic ingredients of EPIC are:

- Bundling of 3 Reduced Instruction Set Computing (RISC) like instructions into a 128 bit instruction bundle.
- Use of a CPU-version dependent number of memory, integer, floating point, and branch units, that can execute in parallel.
- Explicit grouping of instructions in consecutive bundles and scheduling them for parallel execution on the different execution units. The grouping is done by the compiler/assembler.

As a result an Itanium processor, as the first member of the IA64 family, can execute up to 6 instructions in parallel.

2.5.4 PCI, PCI-Bridges and Data Transfer

Peripheral component interconnect (PCI) is a bus with several devices operating on the same physical bus lines, therefore requiring bus arbitration cycles as part of data transfers. It has a packet-based load/store architecture, i.e. burst write and read transactions are possible with a single address cycle at the beginning, indicating the target of the transfer as part of the PCI address space. Usually the PCI address space (32/64 bits respectively) is mapped into the physical address space that the host CPU accesses via ordinary memory load/stores. IO (`inp`, `outp`) operations are mostly not used on IA32/IA64 based systems, since their IO space is very limited.

PCI uses reflective wave switching, i.e. the signalling device drives only half of the signalling voltage on the different PCI traces, the remaining half resulting from the addition of the signal reflected at the traces end. This signalling technology puts high demands on the bus layout, the connectors, and the connected devices in terms of trace lengths, capacitances, inductances, and resistances. It also fixes the maximum operating frequency of a given bus implementation and the performance of the bus can not be enhanced by connecting more capable devices, as in contrast to e.g. Ethernet.

PCI-Bridges mediate transactions between some local bus and the system PCI-bus. We use two PCI-bridges from PLX, the PCI9080 and the PCI9656 that interface to 33 MHz / 32 bit PCI and 66 MHz / 64 Bit PCI respectively [PLXa, PLXb].

The local interface, i.e. the interface that not connects to PCI but to the on-board devices, is essentially the same for both bridges. The local bus can be used as address/data multiplexed or de-multiplexed, we use the multiplexed version, i.e. addresses and data are transferred successively on the same bus. The 32 bit local bus of the PCI9080 can run with a clock rate up to 40 MHz, however, the 33 MHz / 32 Bit PCI bus restricts transfer rates to a theoretical maximum of 132MB/s. Rates exceeding 125 MB/s have been measured with this device. The PCI9656 accepts 66 MHz / 64 Bit PCI according a PCI bandwidth of 528 MB/s. Its local bus has the same 32 bits width but can handle a clock rate of 66 MHz, resulting in a theoretical bandwidth of 264 MB/s. Our measurements have shown a maximum bandwidth of 251 MB/s.

The two ATLANTIS coprocessors, ACB and AIB, use 32 bit / 33 MHz CompactPCI (cPCI), μ Enable uses 32 bit / 33 MHz PCI, and the MPRACE and the ROBIN use 64 bit / 66 MHz PCI. All the coprocessors fit into standard system cases, 19 inch CompactPCI crates in the case of the ATLANTIS coprocessors.

The bridge maps three address spaces between the local bus and the PCI bus: one for its own configuration register space, and two local bus address spaces. The bridge can act as master or slave to the PCI bus, i.e. both buses (PCI and local) can trigger transactions on their counterpart. The buses are connected by FIFO's, which decouples the bus-clocks and aid in bursting data (both buses, PCI and local, can transfer data in bursts, omitting intervening address cycles). The bridge has two independent DMA engines, which can transfer data without the aid of the host CPU or some local controller.

Data Transfers between host and coprocessor can be initiated by the host, the PCI-bridge, and the coprocessor logic. Transfers initiated by the coprocessor logic, i.e. the FPGA design, are not covered in this work. Host initiated transfers are called programmed I/O (PIO). PIO transfers are done by the host CPU that reads and writes data to the PCI address space. Regions of the PCI address space can be mapped into kernel or user address space, making it available to ordinary memory operations, e.g. `memcpy`. Usually the CPU is blocked during an PIO transfer, and often successive PIO transfers require repeated arbitration of the PCI bus.

DMA (Direct Memory Access) transfers are driven by the DMA engines on the PCI-bridges. The bridges are configured for DMA transfer by the host, which also provides DMA-capable memory, i.e. memory that can be reached from PCI devices. Mostly, DMA provides higher throughput than PIO because DMA transfers can burst on the PCI bus, eventually saturating the theoretically PCI bandwidth. During DMA transfers the host-CPU is free executing other instructions, since the transfer is driven by the PCI-bridge. However, DMA must be initiated and prepared by the host, which may introduce an additional overhead compared to PIO. The PLX PCI-bridges, that we use, can be programmed for DMA transfers from the local bus by the coprocessor, too, but we did not use this feature.

2.5.5 Operating Systems, Device Drivers and Tools

The operating systems (OS) used are the two most common: Linux and WindowsNT. As of Windows it is Windows NT 4, Windows 2000, and Windows XP. The Linux systems are Linux 2.2, 2.4, and 2.6; partly build from the kernel sources [TC04], and partly from Linux distributions like SuSE, RedHat, and the CERN RedHat distributions. The so called Linux OS would not be complete without all the necessary system utilities, the most important being the *tool-chain* of compiler, assembler and linker. Tool-chain and other OS utilities come from the GNU software collection [Com].

The compilers that were used are Microsoft Visual C (MSVC) in the versions 5, 6, .NET, and .NET2003, and GNU GCC in versions 2.95 to 3.3. The build system were the MSVC IDE, and GNU make. CVS was used for the code repository, and a common file system was established across the platforms using NFS and Samba on a file server. For the build system to work correctly it was necessary to synchronise date and time on the different workstations, this was done using the network time protocol (NTP). After investigating some other sources of timing information all timing was done using the time stamp register or equivalent, that is present on all IA32 and compatible and IA64 CPU's

The initial bootstrapping of most of the reconfigurable coprocessors was aided by using logic analysers, pci-analysers and oscilloscopes.

For preparation of (simulated) detector data and for making algorithm performance tests in the context of the ATLAS HLT the reference software [Hau00] was used.

Device Drivers mediate between the coprocessor device, the host OS, and the user application that uses the coprocessor. Device drivers are OS specific, i.e. every OS needs its own driver. It is the drivers responsibility to register the used resources of the coprocessor, like the occupied PCI address space or the interrupt resources within the OS, to prevent other applications from simultaneous access to the coprocessor. The device driver may map parts of the coprocessors PCI address space into kernel- or user address space, to make it accessible by other system- or user- software. The driver also provides kernel memory, that is suitable for DMA. DMA can be done in principle from any host memory that is accessible from the PCI bus. However, user mode memory (malloc'ed memory) must be locked, e.g. to prevent it from being swapped to disk, before it can be used for DMA. The driver may accomplish this.

The PCI bridges used on all the supported coprocessors were manufactured by PLX Technology and the first coprocessor I have been working with was the μ Enable from Silicon Software [Sof]. Consequently, the PLX SDK [PLXc] containing Windows device drivers and an API library, and a similar package from Silicon Software, also supporting the Windows OS, were used in the beginning of development. Additional, WinDriver from Jungo [Jun] was considered and tested.

However, the main effort went into a Linux driver that was initially developed

at NIKHEF⁵ for the S-Link.

2.5.6 Software Development

All software that was developed during the work for this thesis was written in the C++ programming language [Str00]. Exceptions to this are some tiny functions used for timing measurements that were implemented as assembler macros. The standard template library (STL) was used for some containers and for most input and output [Aus98]. Intentionally or by chance I used some concepts of object oriented analysis and design [Boo94], and design patterns [GHJV95]. I tried to avoid the pitfalls stated in [Mey96] and [Mey97]. The Unified Modelling Language (UML) [Boo99, SvG00, Dou99, Qua99] was used during analysis, design, and later for reverse engineering.

⁵Jan Evert van Grootheest, National Institute for Nuclear Physics and High Energy Physics, Amsterdam, Netherlands

Chapter 3

Results

3.1 Introduction

In the previous chapter, I described the motive, i.e. online analysis of ATLAS detector data, the environment, i.e. high level trigger of the ATLAS experiment, and the materials, i.e. FPGA coprocessors and host computers, that relate to the work for this thesis.

In this chapter, I present the results that I obtained from my research conducted for this thesis. The first part of the presentation starts with the definition of the software process that I used during the development of the software. In the following, I depict use-cases of the software, and software concepts deduced from these. I describe the static software architecture realising the software concepts, and the behaviour of the software, implementing the use-cases. The evolution of the software and the structure of software packages are shown in the following. Tools are described, that I developed and used for precise timing measurements, and the results of some timing measurements are shown that quantify performance of data transfers and the overhead introduced by object-oriented programming. At last, I shortly describe operating system related issues.

In the second part, I describe results from integrating FPGA coprocessors and the control software into high level trigger software frameworks and trigger prototypes. Finally, in the third part, I describe results from investigations carried out to reflect the reconfigurability of the hardware by a component-based runtime-reconfigurable software framework.

3.2 Description of the Software

3.2.1 Software Process

Even before detailed requirements analysis the software process, or life-cycle, has to be defined. The process describes the general strategy that is followed in order to deliver the software according to the user's requirements. The finding of the right

process needs to take into account user requirements, resources (e.g. manpower), constraints (e.g. schedules), and possibly other external inputs, that influence the process of software engineering.

Steve McConnell [McC96] describes that the careful selection of a software process according to environment and requirements has a great influence on software quality, cost-effectiveness, compliance with agreed schedules, and other measures.

The environment and requirements of the software that was developed during the research work for this thesis can be characterised as follows:

- The first aim for the software was the support for the bootstrapping of the ACB. For achieving this, some low level tools would have to be developed very early.
- The functions of the ACB were not strictly defined during development, but rather subject to change and refinement due to configuration of the system integration PLD's. Thus, the above mentioned tools were to be used to define the functions of the ACB.
- The functions required from the software were assumed to change according to changed requirements from client applications. Also the software frameworks in which the software was expected to be integrated were assumed to be very different from each other.
- Performance is a critical issue for the software and redesign and optimisations according to findings from the use of the software were expected.
- The FPGA coprocessor family that the software should support was known to grow steadily.
- The life-cycle of the software was expected to exceed the time-span for this thesis' work.

Among the software processes described in the literature, e.g. pure waterfall, design-to-schedule, code-and-fix, and others, only the *iterative prototyping*, or *spiral*, process explicitly addresses the risks of changing requirements. Also, iterative prototyping is well suited for the early delivery of tools and other prototypes, that later can be adapted or refined, or just be delivered as is, if they meet the requirements.

Figure 3.1 illustrates the process as a spiral. This process is based on the repetition of requirements analysis, definition of use cases, risk addressing, design, implementation, test and delivery. A working prototype that meets the before defined requirements is delivered after every iteration. I followed the spiral process throughout all of this research work.

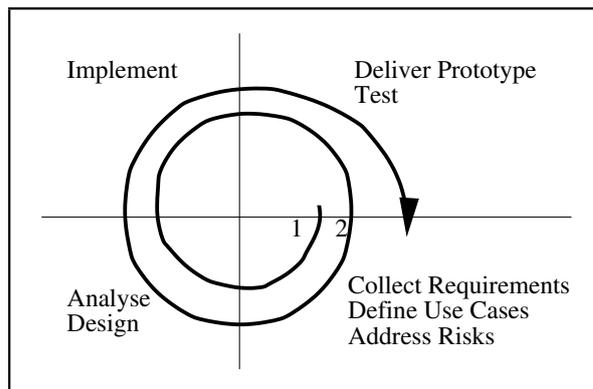


Figure 3.1: Iterative Prototyping with Risk Addressing

3.2.2 Software Architecture

The software that I created during this work is not abstract or theoretical but was supposed to do real work right from the start. The software is now in use for more than 4 years although the time-span for its life-cycle was not well defined when I started the requirements analysis, the first phase of the software life-cycle. Also, it was foreseeable from the beginning, that the environment of the software would be rather difficult and changing in terms of requirements, users, and developers.

This work started during the development of the Atlantis Computing Board (ACB) (see section 2.4.3), and bootstrapping of the ACB was the first application of the software. The requirements analysis resulted in the following use-cases, ordered by the sequence in which they were applied during the bootstrapping of the ACB:

- configure JTAG devices
 - using external connector
 - using PCI-bridge
- read/write configuration EEPROM
 - using external connector
 - using PCI-bridge
- setup clock resources
- make the address space of the FPC-FPGA ¹ available to the user
- transfer data with DMA from/to the FPC-FPGA

¹The FPC-FPGA of the ACB is the only FPGA that is directly accessible for data transfers by the PCI-bridge. Transfers to the other three FPGA's must be routed through the FPC-FPGA.

I chose a strict *bottom-up* design approach for several reasons. Among these reasons were:

- It was foreseeable that both the software and the hardware would need a lot of debugging aid.
- The extension to other hardware platforms, like the AIB, was already planned.
- The process of bootstrapping the hardware required low-level access to various devices.

I tried to make the software as modular as possible to prepare the software for the extension of the supported hardware platforms and to support reuse. I deduced the following software concepts according to the modular bottom-up approach:

- Configuration data and bit-stream
- JTAG-capable device
- EEPROM
- Clock generator and programmable clock buffer devices
- PCI bridge
- PLD functions

Most of these concepts refer to real devices on the hardware. Figure 3.2 illustrates the mapping from physical devices to software entities (classes) for the case of the μ Enable coprocessor. (The μ Enable is chosen for the illustration because it is much less complex than the ACB, albeit showing the general principle.)

The following paragraphs and figures describe the classes and class-relations which I implemented according to the above sketched software concepts which in turn resulted from the requirements analysis.

Bitstream The configuration data for the programmable logic devices has to be loaded from external storage and stored in host memory for preparing it to be uploaded to the device. The bit-serial JTAG protocol requires access to the bit-stream data at bit-level, whereas the parallel configuration usually works with 8-bit words.

A bit-serial protocol is also used by the configuration EEPROM's present on all coprocessors and on most of the expansion modules. Accordingly, bit-stream data plays a central role in the software. I created a `Bit` package containing the `BitStream` and `Bit` classes; the package has no external dependencies. The `BitStream` initially was using an STL container of chars, each char containing one bit, but now it uses raw binary data for storage. The `BitStream` and `Bit` interface did not change during the whole life-cycle of the software besides this change in internal representation. The `BitStream` interface is sketched in figure 3.3.

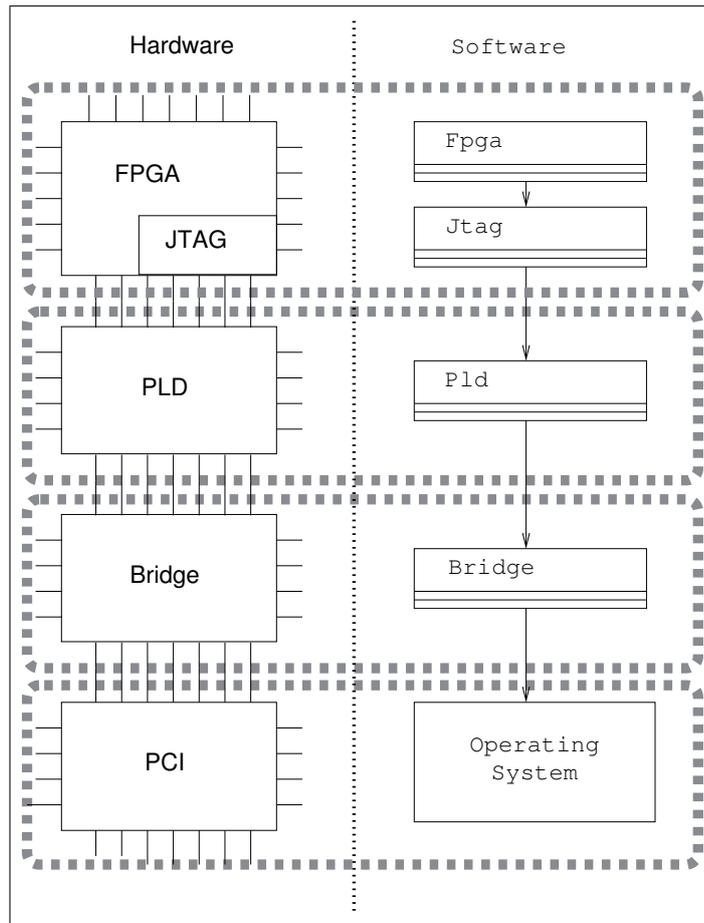


Figure 3.2: Mapping from Hardware to Software
Shows the close relationship between hardware devices and classes created during first analysis.

```

class BitStream {
public:
    typedef enum {LOWEND, HIGHEM} Endian;
    typedef size_t Size;

    BitStream();
    ...
    void          Set(const char *b);
    ...
    Size          GetLength() const;
    unsigned char GetUInt8(Size i, Endian e) const;
    ...
    const Bit     GetAt(Size i) const;
    void          SetAt(Size i, const Bit b);
    ...
    void          Reverse();

    BitStream& operator = (const BitStream & bs);
    bool operator == (const BitStream & bs) const;
    bool operator != (const BitStream & bs) const;

private:
    AIntVar<unsigned int> bits;
};

```

Figure 3.3: BitStream Interface

Design Different storage formats exist for configuration data, often called “design file”, like ASCII encoded `.bit` and raw binary `.rbit` files. The `Design` class provides methods for reading and writing both formats, parsing them into internal `BitStream` attributes and providing access to additional information extracted from the design files like device type and creation date, among others.

JTAG is used for uploading PLD configuration data, and can be used for uploading configuration data to the FPGA’s. Also JTAG provides means for FPGA operation control, configuration reset, configuration and status read-back, and boundary scan (see 2.3.5).

I designed the `JtagController` class that captures the details of the JTAG protocol, especially the stepping through the devices’ JTAG state machine with the TMS signal. The class provides methods for initialising and resetting the state machine together with read/write access to the instruction and data register. Four pure virtual methods, `SetTck()`, `SetTms()`, `SetTdi()`, and `GetTdo()`, are defined that correspond to the four JTAG port signals as summarised in table 2.3.

Access to physical JTAG port signals is provided by intermediate devices. Two examples taken from the ACB are the external connector, usually connected to a host’s parallel port, and the control-PLD, that provides access to the four FPGA’s’ JTAG ports. Usually these intermediate devices are captured as classes in the software, here as `ParPort` and `AcbCtlPld`. The abstract `JtagController` class is *adapted* to the intermediate devices interfaces by sub-classing, comparable to the

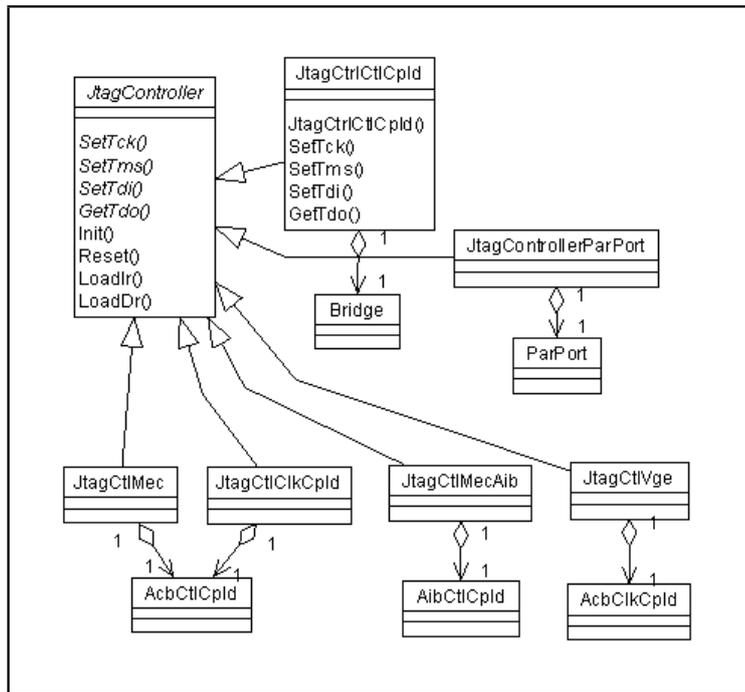


Figure 3.4: JTAG Controllers Class Diagram

adapter design pattern as described in [GHJV95]. Figure 3.4 surveys the resulting concrete JtagController classes.

Sub-classing is done by adding an attribute to the used device, e.g. ParPort or Bridge, that is initialised in the subclasses' constructor, and by overwriting the abstract JTAG port set/get methods. The attributes of the adapted JtagController classes are represented as class objects in figure 3.4. The overwritten set/get functions are hidden in the figure for clarity except in the case of the JtagCtrlCtkCpld, the JtagController used for configuring the control-PLD on the ACB. Figure 3.5 illustrates the relations of classes that implement JTAG functions for the ACB. The arrows with white arrow heads indicate inheritance; arrows with black heads represent ownership. The shown Fpga class acts as a client.

EEPROM's are present on all coprocessors. They provide configuration data for the PCI-bridges and information for identification of the specific coprocessor like type and serial number. Some expansion modules also contain EEPROM's that provide module identification data.

I designed an EepromAdapter class hierarchy similar to the JtagController class hierarchy, that provides a uniform abstract interface to the different EEPROM's. The physical interface to the serial EEPROM is captured in pure virtual set/get methods that are overwritten in concrete subclasses like PlxEepromAdapter for the configuration EEPROM in the running coprocessor.

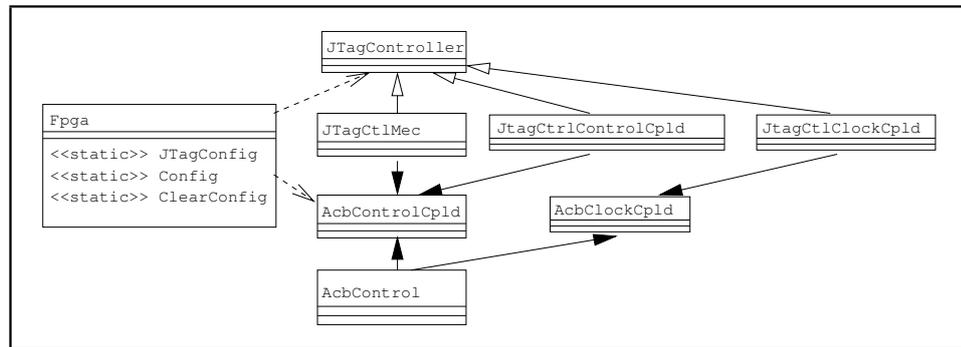


Figure 3.5: JTAG on ACB

The class diagram shows the relation of classes that are used for configuration of devices on the ACB using JTAG.

sor, and like `PEepromAdapter` for a blank EEPROM connected to an parallel port via the external connector for uploading initial configuration data. Figure 3.6 illustrates the class hierarchy.

I implemented a `ParPort` class, that provides an OS independent interface to the legacy parallel port present on most PC. The parallel port was used during bootstrapping the ACB for initial configuration of the PLD's and configuration EEPROM through the external connector of the ACB. The class diagram in 3.7 illustrates the adaptation of the parallel port interface to the JTAG and EEPROM interfaces. The code in figure 3.2.2 illustrate the compactness of the adaptation as an example for all the interface-adapters in the software.

Clock Resources are captured in some classes written by A. Kugel². These classes are implemented as utility classes as opposed to the adapter-like pattern as in e.g. the `jtag-` and `eeprom-` classes. Nevertheless they provide encapsulation for the calculation of the data needed to program both the programmable clock generators and the programmable clock buffers that are present on the ACB and the other boards supported by the software.

The clock system on the ACB is complex and manipulating the clock settings is a rather fragile process incorporating both PLD's and the different generators and buffers. The reason for this is that some of the clock signals themselves are routed through the ACB clock-PLD, which also connects to the clock buffers' programming interfaces whereas the ACB control-PLD controls the clock generators. As a result, the code providing access to the clocking resources is spread in the ACB control- and clock-PLD- classes, and is called from methods of the `ACBControl` class.

²Andreas Kugel, University of Mannheim, kugel@uni-mannheim.de

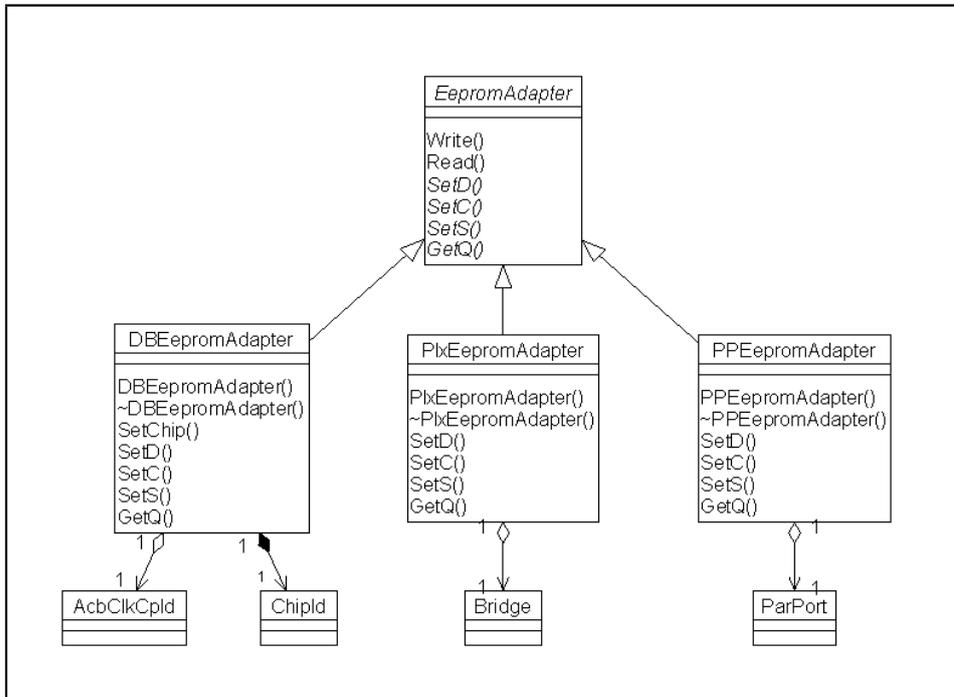


Figure 3.6: EEPROM Controllers Class Diagram

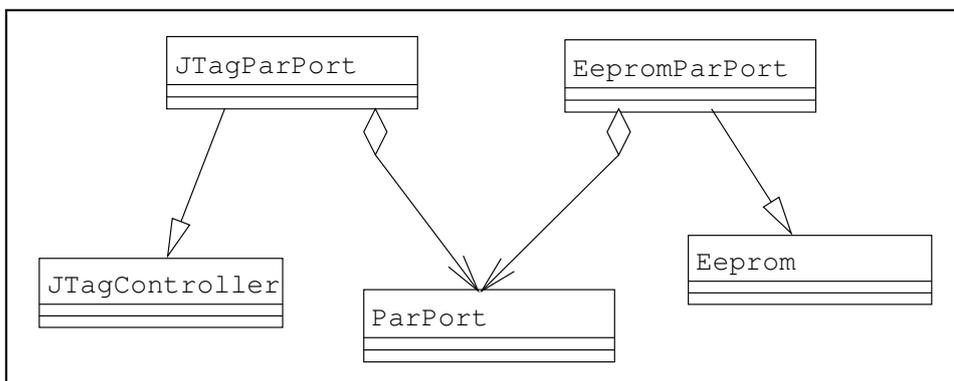


Figure 3.7: Adaptation of the Parallel Port Interface

```

class PPEepromAdapter : public EepromAdapter {

    ParPort & pp;

    void SetD( bool b) { pp.SetPin( 2, b ); }
    void SetC( bool b) { pp.SetPin( 3, b ); }
    void SetS( bool b) { pp.SetPin( 5, b ); }
    bool GetQ( )      { return pp.GetPin( 15 ); }

public :
    PPEepromAdapter(ParPort & parPort) :
        pp(parPort)
    {
        pp.SetPin( 4, 1);
        pp.SetPin( 6, 0);
        pp.SetPin( 7, 0);
        pp.SetPin( 8, 0);
        EepromAdapter::Init();
    }
};

```

Figure 3.8: EEPROM on the Parallel Port, Source Code

The PCI-Bridges connect the coprocessor to the host system via the PCI bus. We use two different types of these, one for 33 MHz / 32 Bit PCI and one for 66 MHz / 64 Bit PCI. However, the differences between the two turned out to be mostly transparent to the software. I captured the remaining incompatibilities in two specialisations of the common `Bridge` base class.

The bridge class provide methods for identification, allocation and freeing of bridges on the system buses through its constructor and destructor. It uses the driver class (see 3.2.6) for different purposes, e.g. for the mapping of the PCI address spaces covered by the bridge into the host's memory address space. Also, DMA transfers are provided by the bridge class but belonging memory locking issues are left to the driver class. However, DMA is a complicated issue since some drivers do not allow for separate memory locking and hide the DMA control in their respective libraries. In these cases, the bridge class transparently hands over DMA-related issues to the driver class. Figure 3.9 illustrates the relation between `Bridge` and `Driver`.

PLD's for system integration are present on all coprocessors that are supported by the software. They provide functions like local bus arbitration and interrupt signal routing from the FPGA's to the bridge, mediate access to the JTAG ports and the low level control signals (INIT, PROG,...) of the FPGA's, and provide access to clocking resources and some control signals on expansion and module connectors, if present.

All coprocessors have a central control-PLD that is a participant on the local bus, together with the PCI-bridge and the FPGA. The ACB has a second PLD, the

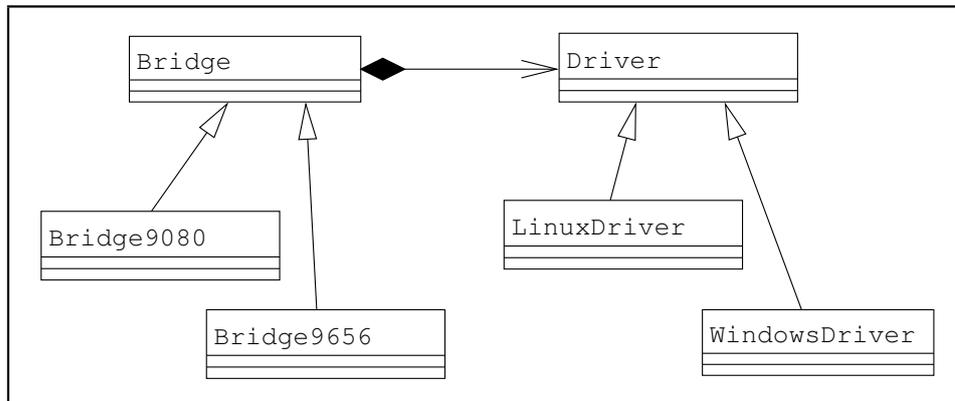


Figure 3.9: Bridge and Driver Work Together

clock-PLD, merely for controlling the powerful clocking resources present on the ACB. Figure 2.12 illustrates the arrangement of the two PLD's on the ACB.

The PLD's functions are accessed by read and write transfers to registers. These registers are offered to their respective address space on the local bus. This address space is in turn mapped to the address space of the PCI-bus by the coprocessor's bridge.

The PLD's functions are, besides some similarities, specific to every coprocessor, and so is the register-set layout. Non-volatile configuration data ("design") determines the actual PLD's behaviour; the configuration was uploaded to the PLD during board manufacturing with JTAG (see above), and may be changed later to refine, extend, or debug the PLD's behaviour. Every PLD instance needs its own software to capture its behaviour. Since the PLD's are board-specific and are subject to change during development it is desirable to encapsulate as much of the behaviours internals from the rest of the software. Consequently every PLD has a unique corresponding class in the software. Examples are the classes `AcbCtlPLD` and `RobinPLD`.

Register offsets and bit-masks have to match between the design description, usually VHDL or Verilog sources, and the PLD class definition, a C++ source file. A. Kugel implemented a framework that extracts such information from the VHDL design sources, and imports them into the PLD class definition, which aids in the implementation of the C++ class.

Control class specialisations are implemented for all supported coprocessors. All control classes derive from the `Control` base class, which provides common services. The specialised control classes act as "builders" that build the coprocessor specific runtime object hierarchy that is needed to control the coprocessors, and provide interfaces which cover the underlying internals. Figure 3.16 shows the control classes hierarchy. Figure 3.10 shows the relations between the basic classes that are composed by the control class for the case of the MPRACE coprocessor.

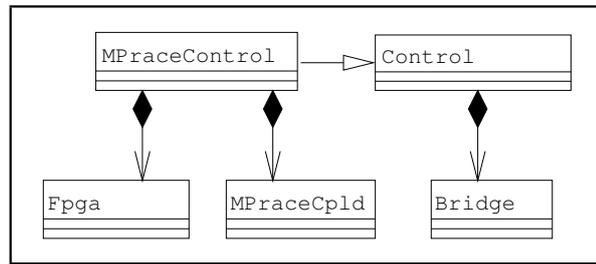


Figure 3.10: The Control Class Encapsulates the Internals

`MPraceControl` is a specialisation of its base class `Control`, from which it inherits a `Bridge`. `MPraceControl` additionally holds an `Fpga` and a `MPracePld`.

Identification of coprocessors is done in two steps. First, the bridges in the system are enumerated according to their types by the hardware driver. Second, the configuration EEPROM present on all coprocessors is inspected through the bridges and the coprocessors are identified and enumerated according to information read from the EEPROM's. Some application, like complex image recognition, feature extraction through Hough transformation, read-out-buffers, and others, use several coprocessors in parallel. Accordingly, hosts may contain several different coprocessors and a resource allocation schema and identification services are provided by the software to manage the presence of multiple coprocessors.

Error Handling is traditionally done often by checking the return values of function calls. This technique is error prone; leaving errors undetected, and implies a lot of coding overhead. I chose to use C++ exceptions for all error handling. All exceptions in the software are of type `MsgException` or a derivate of it, which in turn is derived from the standard C++ `exception` class. Since every exception carries a unique textual error description it is almost trivial to deduce the error cause even if the exception is caught at levels higher then the level where the exception was thrown. In general, exceptions are critical and unrecoverable errors, causing the software to abort with a dump of the textual description of the exception. Thus, the return values of functions and methods are reserved for their intend role, the transfer of data from the called function to the caller of the function. The choice of the strong typed C++ programming language helps in the detection (semantic) errors at compile-time rather than at run-time.

3.2.3 Software Behaviour

In the preceding section I described the static architecture of the software, as it resulted from requirements analysis. In this section, I describe the dynamics of the software, i.e. the construction, destruction, interactions and collaborations of objects.

I use in most practical cases the paradigm of “instantiation is initialisation” i.e. creating an object initialises it into a defined state, ready for use. Failing to initialise is a critical error causing an exception to be thrown and the application to abort. Vice versa, destroying an object implies cleanup of all the resources it used, both on the host and on the coprocessor, again leading to a save state.

The first diagram (fig. 3.11) visualises the responsibility of a `Control` object for creating and destroying the set of objects that is needed to interact with a given coprocessor. It is carried out as a sequence diagram, stressing the sequence of allocation, use, and de-allocation of a coprocessor. When the client creates a control object, the object checks the `Registry` if the requested coprocessor is available before it registers itself with the registry. After that, the objects owned by the control object are created and initialised, that are the `Bridge`, `Xpld`, and `Fpga`. Finally, the client releases the acquired coprocessor by destroying the controller object, which in turn unregisters itself with the registry, after destroying the objects that were previously allocated by it.

The second diagram (3.12) illustrates the process of configuration of an FPGA on a coprocessor. It gives an expanded view of the “configure” sequence from diagram 3.11. In that diagram I used a `<<jtag>>` stereotyped message from the `Fpga` to the `Pld` to indicate the use of JTAG for the configuration.

The diagram is carried out as a collaboration diagram to emphasise the relationships between the involved objects. `BitStream` and `Design` objects which hold the configuration data are omitted for clearness. Some of the supported FPGA’s can be programmed to disable their JTAG ports upon configuration. Therefore, a save reconfiguration or configuration clearing algorithm requires access to the `PROG`, `INIT`, and `DONE` signals of the FPGA since these signals cannot be disabled and provide means for configuration clearing without using JTAG. This causes the loop in the relations between the `Fpga`, `Jtag`, and `Pld` objects, since `PROG`, `INIT`, and `DONE` are accessed through the `PLD`. Besides this loop, the relations are unidirectional from the initiator, the `Control`, to the target, the `Bridge`. The message signatures indicate the interfaces used, the arrows indicate that all messages flow from the control object to the bridge object. None of the targets reference any objects upstream, e.g. the `Pld` uses only the `Bridge` and does not know about `Jtag` or `Fpga`.

3.2.4 Evolution and Coprocessor Packages

The lifetime of the software, from initial requirements collection until the present, now exceeds 50 months. The software’s architecture was designed so that it supports the adaptation to forthcoming coprocessor implementations.

In this section, I present software measures that were taken from the software’s history that show the evolution of the size of the software together with diagrams illustrating the support for new coprocessor implementations. Also, the partitioning of the software between different packages and domains is described.

Steady extension, maintenance, refinement, and adding of documentation and

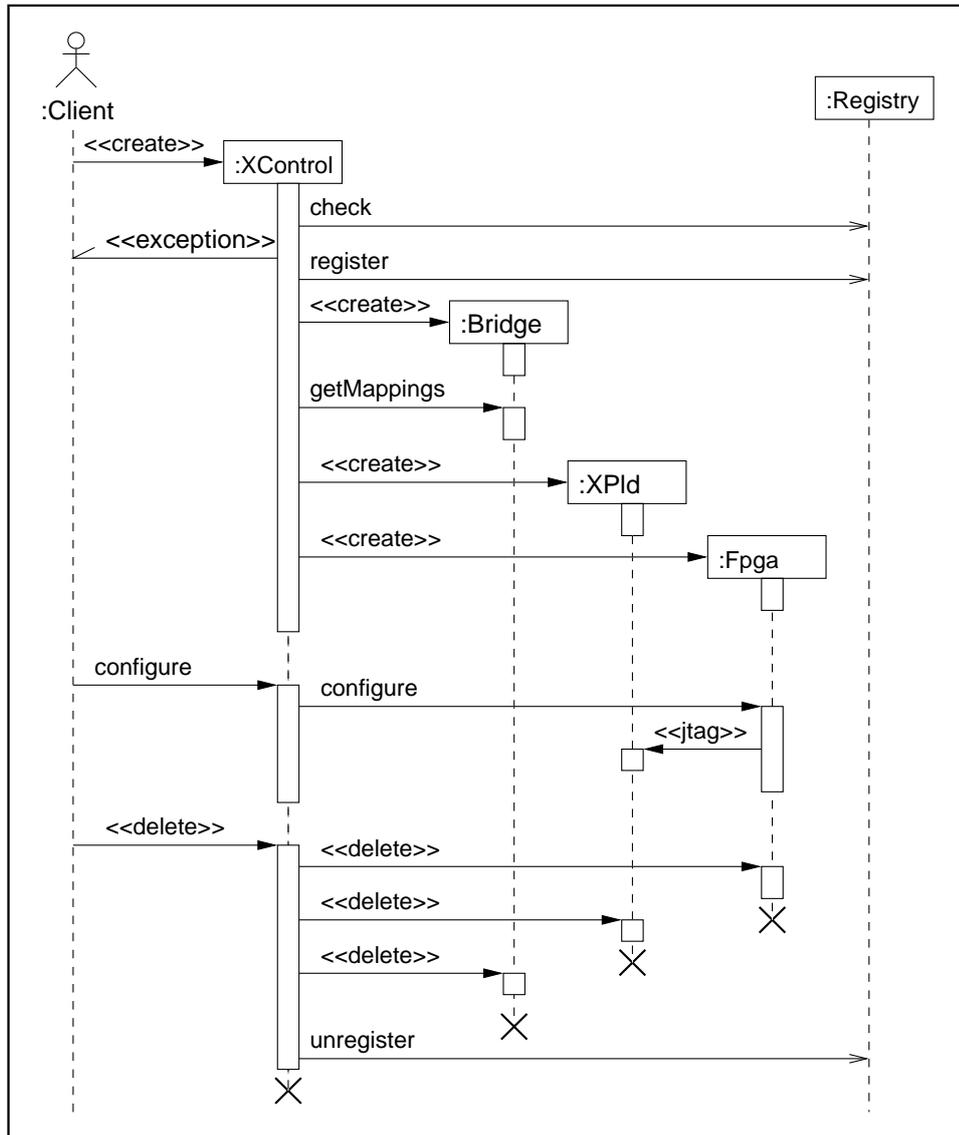


Figure 3.11: Allocation, Configuration, and Freeing of a Coprocessor

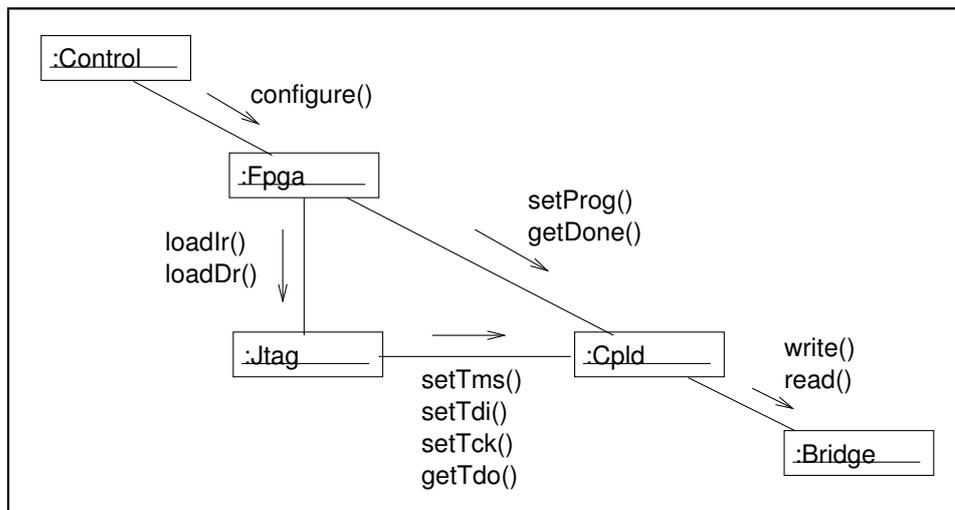


Figure 3.12: Configuration with JTAG, Collaboration Diagram

comments increased the size of the software's source code as shown in figure 3.13. The graph was created by extracting the software from the repository at a one month cycle. All additional files, like makefiles, readme, and not-essential utilities, were removed, leaving just the sources of the library. The starting point of the graph, "0 months", is well after initial design and implementation, and the software was already supporting the ACB and μ Enable coprocessors at that time. Support has been added for the AIB, the microEnable2, the MPRACE, and the ROBIN coprocessors during the covered time-span.

The software is partitioned into packages that reflect the layered software structure. The bar-chart 3.14 illustrates the size of the packages, table 3.1 gives the exact numbers. The `core` package is a collection of basic services used by all coprocessor support packages and by most tools that are build on top of the software. This includes e.g. infrastructure for OS- and compiler-independency, timing functions, some base classes that are specialised in coprocessor support packages, logging, and memory handling. The `bitstream` package provides efficient manipulation, extraction of bitstreams together with their storing and parsing in different formats. `clock` collects services for programming different clock generators. The `driver` package manages the relations to OS dependent device drivers, while the `bridge` provides classes to control the two supported PCI bridges. The `fpga` package provides services for the configuration of the various supported FPGA's. Finally, the `registry` package is responsible for the detection, allocation, and de-allocation of the supported coprocessors that are found in a system. All these base-packages have sizes in the range of approx. 50 - 80 kB, except the small `clock` package which has a size of approx. 20 kB, and the `core` package, which is the biggest package of all base-packages, having a size of approx. 154 kB.

Besides the base-packages, a number of packages (`acb` `aib`, `ue`, `ue2`,

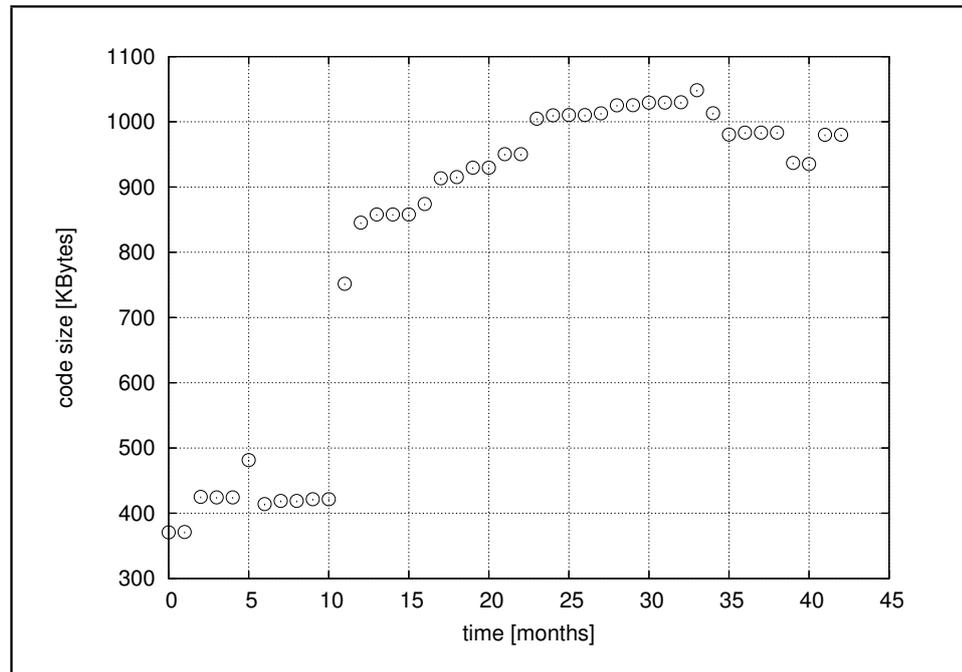


Figure 3.13: Evolution of Code Size

`mprace`, and `robin`) comprise support for the different coprocessors. All these coprocessor-support-packages are dependent on the base packages. The sizes of these packages range between approx. 50 - 70 kB for the `ue`, `ue2`, `mprace`, and `robin` packages, approx. 100 kB for the `aib` package, and approx. 150 kB for the `acb` package.

The layered and modular architecture of the software enables the adding of support for new coprocessor implementations in clearly separated and independent packages. However, some coprocessors have been designed in such a way, that they share some common properties in a so called “ μ Enable compatible” fashion. E.g. the way in which the access to the JTAG port of the FPGA’s is routed through the control-PLD to the coprocessor’s address space is more or less compatible on the μ Enable, the μ Enable2, the MPRACE, the ACB, and the AIB. Most of this “compatibility” is captured in the `core` package. Figure 3.15 illustrates the general structure of support packages for the cases of the μ Enable and the ACB coprocessors in the `ue` package and the `acb` package respectively.

The class diagram 3.16 surveys the top-level control classes for the different coprocessors. These classes represent, among other functions, the user interface to the coprocessors and reside in their respective packages. The `Control` base class is located in the `core` package.

The PLD devices on the various coprocessors play central roles in their respective coprocessor designs. Although they are of limited complexity and IO capability compared to the FPGA devices, they control and implement sensible

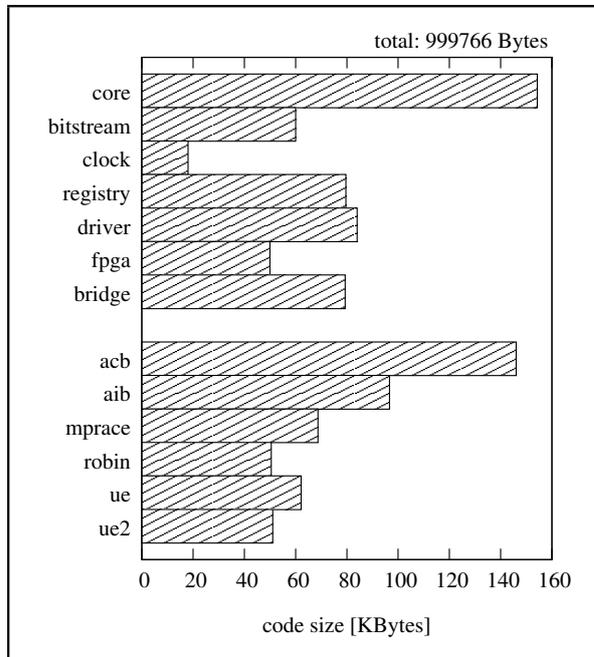


Figure 3.14: Code Size of Packages in the Library

Table 3.1: Code Size of Packages in the Library

package	size[bytes]	package	size[bytes]
core	154230	acb	145973
bitstream	59987	aib	96566
clock	18014	race1	68653
registry	79593	robin	50448
driver	83955	ue	62106
fpga	49940	ue2	51031
bridge	79270		
subtot.	524989	subtot.	474777
total			999766

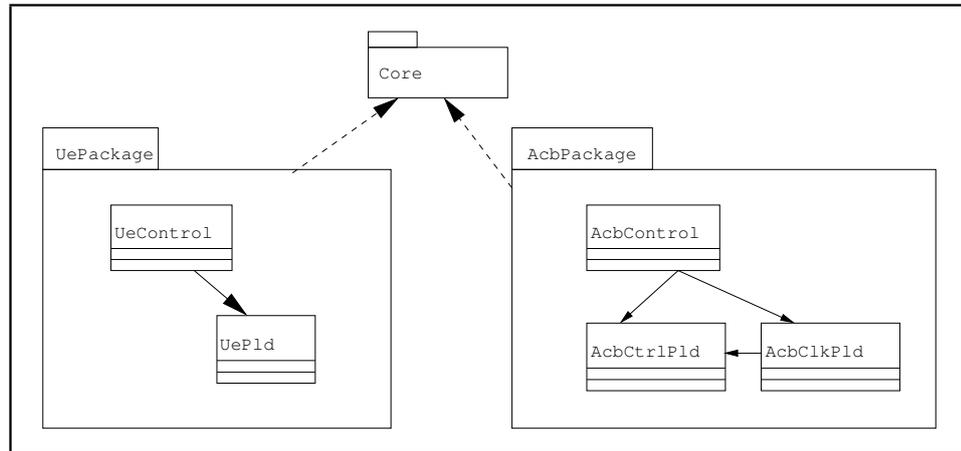


Figure 3.15: Two Packages Supporting Different Coprocessors

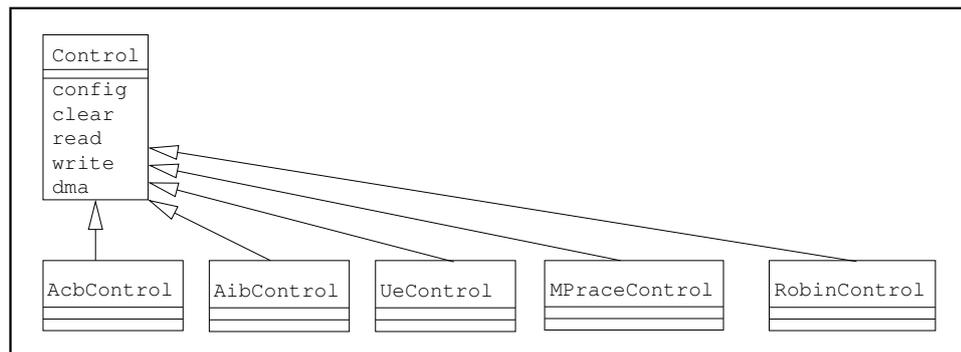


Figure 3.16: Hierarchy of Control Classes

Table 3.2: Partitioning of Code in the Coprocessor Packages

domain	size[bytes]	percentage[%]
PLD	286835	60.4
Control	133822	28.2
misc.	54120	11.4
total	474777	100

and fundamental system functions.

Some of their functions, e.g. the arbitration of the coprocessors local bus, are not visible to the software at all. On the other hand, access to, and control of clocking resources or access to the JTAG ports of the FPGA(s), is fundamental to the software. The PLD support takes in total approx. 60 % of the size of the coprocessor-support-packages. Refer to table 3.2 for exact numbers.

3.2.5 Performance

Reconfigurable coprocessors are used for many different applications, e.g. interfacing to external systems, acceleration of computation, or prototyping. Most of these applications have high demands on system performance. Performance is often rated in terms of throughput and latency, which both need time information to be calculated. Time is not only necessary for determining performance; it is also needed to control time-critical operations on the coprocessor. My thesis covers host software that is needed to control and access coprocessors, and it is the viewpoint of the host or the host's CPU from which I measure time.

Computer workstations obtain absolute time information from their real-time-clock (RTC) hardware device. The battery buffered RTC derives its time from a quartz crystal. Usually, the host system receives regular timer interrupts from the RTC, often with a frequency of about 1 kHz. The host OS synchronises the system time with the RTC time. Another source of time information can be the network time protocol (NTP). NTP is used to synchronise the host's system time with a server host's time with accuracy down to some microseconds. However, despite the limited accuracy of the system time being not better than a microsecond, timing is further rendered difficult by the task of *obtaining* the system time.

I identified the following timing functions in the respective c-runtime-libraries:

- `clock` gives the elapsed time since the start of the process in units of `CLOCKS_PER_SEC`. `clock` is POSIX³ and thus available on Windows and GNU/Linux.
- `time` gives the number of *seconds* since some fixed date in the past, often

³POSIX is a rudimentary standard ensuring application compatibility between operating systems

Table 3.3: Timing Functions

function	OS ^a	resolution [10 ⁻⁹ s]		latency [10 ⁻⁹ s]
		claimed	measured	
time	w/gl	1 · 10 ⁹	1 · 10 ⁹	500
clock	w	1 · 10 ⁶	15 · 10 ⁶	160
clock	gl	1,000	10 · 10 ⁶	1,100
_ftime	w	1,000	1,000	20,000
PerfCounter ^b	w	280	280	2,000
gettimeofday	gl	1,000	1,000	1,000
clock_gettime	gl	1	1,000	1,000
pcc ^c	w/gl	0.5	2	50

^aw:Windows, gl:GNU/Linux

^bAbbrev. for QueryPerformanceCounter

^c2GHz IA32

1970/01/01. Since `time` is a POSIX standard it is available on Windows and GNU/Linux.

- `_ftime` is only available on Windows. It is an enhanced version of `time` in that it additionally retrieves the fraction of seconds in *milliseconds*.
- `QueryPerformanceCounter` is available on Windows. It retrieves the value of the “high-performance counter” implemented in the Windows OS. The counter frequency is determined by a call to `QueryPerformanceFrequency`.
- `gettimeofday` is only available on GNU/Linux. It is the pendant to the Windows `_ftime` but provides extended accuracy down to *microseconds*
- `clock_gettime` is available on GNU/Linux. It is an enhanced version of `gettimeofday`, but gives the fraction of seconds in *nanoseconds*.

I made measurements on latency and accuracy for all these library calls on the OS where they apply. The results are summarised in table 3.3 and show that none of the functions gives a resolution better than microseconds, and all functions have latencies worse than 100 nanoseconds.

Processor Clock Counter

The Intel Titanium Architecture Software Developers Manual [Intb] states in volume 1: “The Interval Time Counter (ITC) is a 64-bit register which counts up at

```

class Pcc
{
public:
    class Now{};

    Pcc(const Now& );
    Pcc();

    void set();
    double seconds(void) const;

    Pcc operator - (Pcc const & ) const;

    Pcc.t m.CPUTicks64;
};

```

Figure 3.17: Processor Clock Interface

a fixed relationship to the processor clock frequency.” The ITC is equivalent to the IA32 Time Stamp Counter (TSC). I made comprehensive tests on a number of IA32-, IA32 compatible (AMD)-, and an IA64-host, to discover the relationship between the ITC frequency and the processor clock frequency. I calibrated these tests using the native `QueryPerformanceCounter` or `gettimeofday` with an accuracy demand of better than 1%. The tests showed that on all hosts, regardless of the OS and architecture, the ITC- and the processor clock-frequency match within an accuracy of better than 1%, therefore I assume that the ITC-frequency and the processor-clock-frequency do match exactly.

The Linux OS kernel synchronises the ITC of the CPU’s on multi-processor (MP) hosts on boot-up, and I found no evidence for a discrepancy of the ITC values on MP hosts. Reading the ITC value from the CPU register must be done with assembler macros in C++ sources, whose syntax relies on the CPU architecture and the C++ compiler used. I implemented such macros for IA32 with MS-Visual-C and GNU GCC and for IA64 with GNU GCC, and encapsulated the resulting functions with a C++ interface called `PCC` as an abbreviation for Processor Clock Counter (see figure 3.17 for an excerpt). The last row in table 3.3 labelled `pcc` shows the superiority of the `pcc` timing functions over the native functions in terms of accuracy and latency. I used `PCC` for all timing in this work.

Overhead caused by Object-Oriented Programming and Software-Layering

I chose C++ as implementation language for the software, and it is not possible to measure the influence of this choice on the software’s performance, since no alternative implementation exists.

A central property of the software’s design is software layering, which results from the encapsulation of the representation of hardware devices’ functions into classes, and the bottom-up design approach.

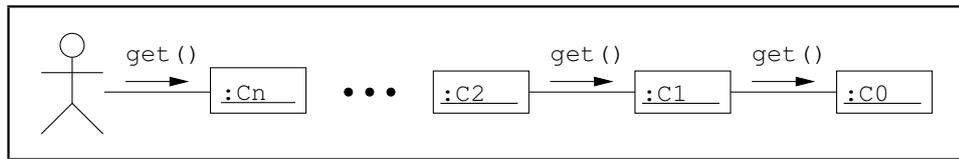


Figure 3.18: Method Call Overhead, Object Chain
Visualise the a *has-a* object chain.

The design requires that a message that triggers a desired action on the coprocessor has to travel a directed chain of objects, as it is illustrated in figure 3.12.

Figure 3.18 shows a simplified object chain, that I used to measure the influence of software layering on performance and the size of the executable machine code. In this setup the member function `get()` is used to read an integral value from the top-level object, `:Cn`. `:Cn` redirects the read to its attribute `:C(n-1)`, which redirects the read to its attribute `:C(n-2)` and so on, until `:C0` is reached. Finally `:C0` returns a value. To prevent the compiler from aggressive optimisation an arbitrary constant is added to the return value in every stage in the return path.

Figure 3.19 and figure 3.20 show the influence of the length of the object chain on latency and size of the machine code respectively. Figure 3.19 also shows the impact of compiler technology on code quality in terms of latency. An actual GNU GCC-3.3.2 introduces about 8.5 additional CPU clock cycles, corresponding to about 4 nsec, for every indirection in the object chain, which is more than 4 times faster than the fully optimised code produced by an older GNU GCC-2.95.

Figure 3.20 shows that the compiler generates a maximum of 66 bytes code for every additional indirection stage.

Throughput

All applications for reconfigurable coprocessors require data to be transferred between the host and the coprocessor. Transfers can be done in two modes: PIO and DMA. I made measurements on a couple of host systems equipped with a MPRACE (see 2.4.4) coprocessor to evaluate the PIO and DMA performance.

Figure 3.21 shows results from DMA on a typical SMP host with 64 Bit / 66 MHz PCI. The data is shown as latency against packet size. Clearly visible is the overhead of about 6 microseconds to initiate the DMA. Included in the graph are fits to the data that indicate a saturated throughput of 255 MB/s for read and 75MB/s for write transfers. The saturated read transfer rate reaches the maximum theoretical throughput of the local bus, which is $32bit * 64MHz = 256MB/s$. (On the MPRACE the local bus is clocked with 64 MHz instead of the maximum allowed 66 MHz.) The poor write transfer rate is caused by the host architecture; other hosts show symmetrical read and write throughput of both about 255 MB/s.

Figure 3.22 shows PIO throughput against packet size on four different hosts, labelled A, B, C, and D, equipped with MPRACE coprocessors. The PIO transfers are done with the `memcpy` c-runtime-library function. Included in the graph for

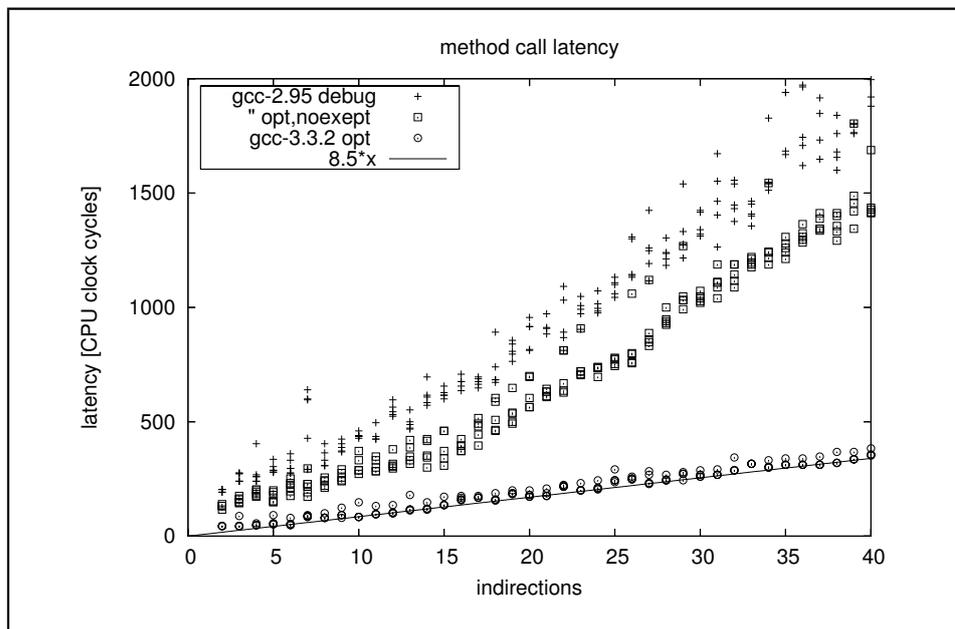


Figure 3.19: Latency of Method Calls

The overhead introduced by routing a method call through a *has-a* object chain is measured. Measurements were done on a 2GHz IA32 CPU i.e. 2000 CPU clock cycles correspond to $1\mu\text{sec}$. The difference in latency between two builds using GCC-3.3.2, one with, and one without exceptions enabled, is smaller than 1%, and thus not visible in this graph.

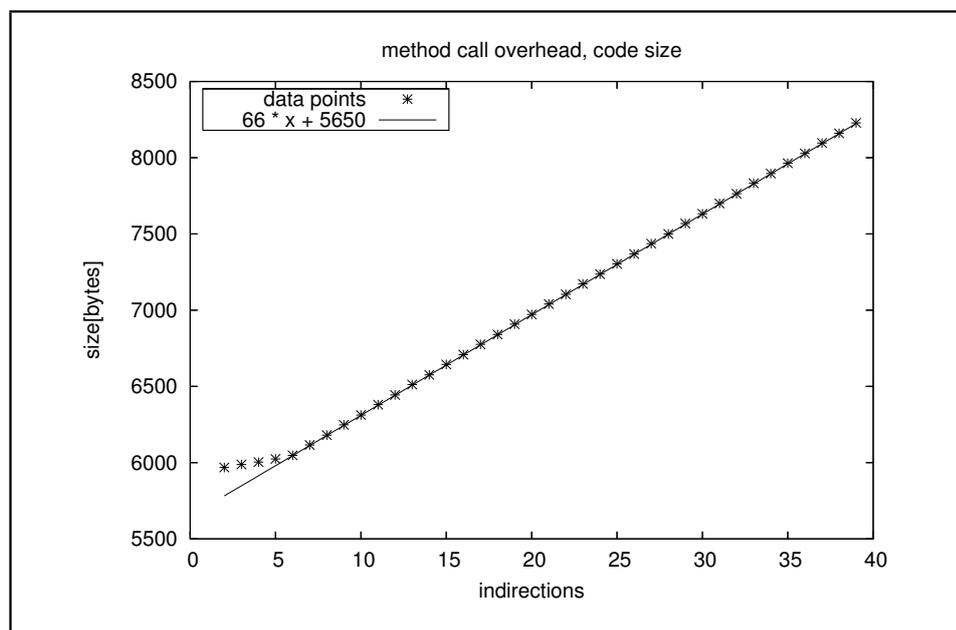


Figure 3.20: Code-Size of Method Calls

The code overhead introduced by routing a method call through a *has-a* object chain is measured.

comparison are the DMA fit functions from the DMA measurements from figure 3.21. Saturation is reached for packet sizes of about 1000 bytes. The saturated write throughput ranges from about 10 MB/s for host A to about 40 MB/s for hosts B and D. The saturated read throughput ranges from about 4 MB/s for host B to about 6.5 MB/s for host D. All write throughput data shows increased throughput for packet sizes of about 10 to 100 bytes. These rates reach a maximum of more than 500 MB/s for packets of about 50 bytes size on host B. This is an effect of the host architecture which is capable of collecting successive PIO writes and delaying the transaction until after the CPU is released. This implies that the CPU is continuing in its instruction thread *before* the PIO write data reaches the coprocessor.

Additional to data transfer between host memory and the coprocessor it is sometimes necessary to move data from one location in host memory to another. This may be caused by restrictions in DMA-capable memory on some host systems, or because data has to be rearranged before or after being transferred between host and coprocessor. I made measurements on a couple of hosts to evaluate the memory bandwidth, and to compare it with the bandwidth between the host and the coprocessor. In figure 3.23 the results of such measurements on two hosts are shown, together with the 255 MB/s DMA-fit from 3.21 and a hypothetical 500 MB/s DMA transfer function for comparison. A DMA transfer bandwidth of 500 MB/s would result from a local bus width of 64 bits instead of the current 32 bits.

The memory bandwidth results are depending on the memory access patterns

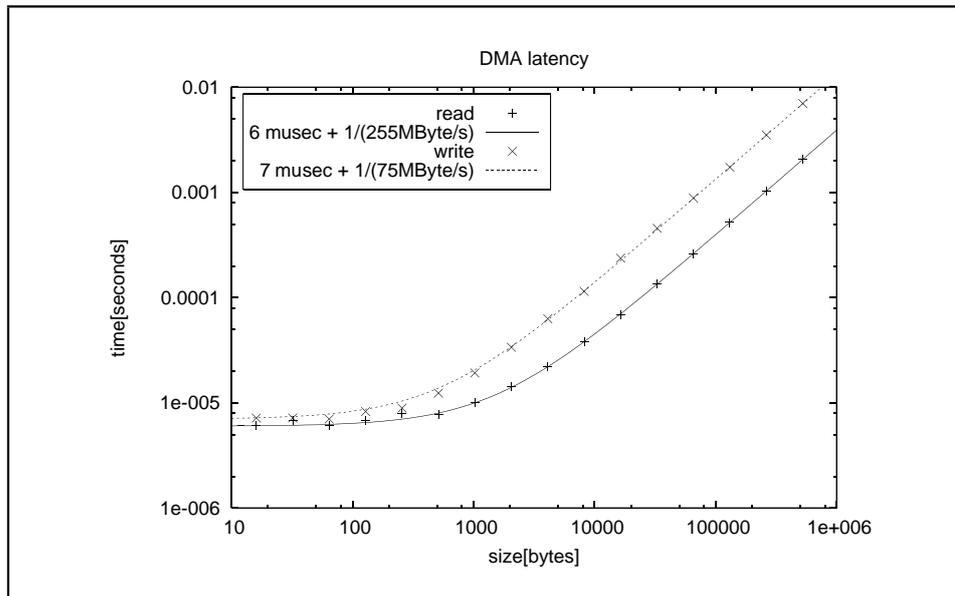


Figure 3.21: DMA Latency

This graph shows the results of accurate measurements of DMA transfer between host and coprocessor.

used in the measurements, which is an effect of the memory caches. Therefore I chose to give minimum and maximum values extracted from a variety of measurements with different memory access patterns rather than average numbers. In some cases, the obtained minimum and maximum values differ by a factor of 20, in others the difference is in the order of 20%. The two hosts from which the results were extracted are of quite different architecture, although both being IA32 based systems. The first one is a UP system with a 2 GHz CPU and RAMBUS DRAM (RDRAM). Its results in figure 3.23 are labelled “rdram”. The second is a SMP system with two 800 MHz CPU’s and ordinary DRAM. Its results are labelled “sdram”. Two manually fitted curves that match the maximum values for packets of sizes between some 100 bytes and about 10,000 bytes are included in the graph. These very high bandwidths seem to result from moving data within memory caches. Remarkable is the superior bandwidth of the UP RAMBUS architecture, which is more than three times greater compared to the bandwidth on the SMP architecture host for big packets.

3.2.6 Build System, Operating Systems and Platforms

The software distribution consists of two packages: the Linux PCI-driver (pciDriver) package and the library package. Both packages are accompanied by some simple test applications that can be used to verify the correctness of the build. The pciDriver packages is replaced by the device drivers and library from the PLX-

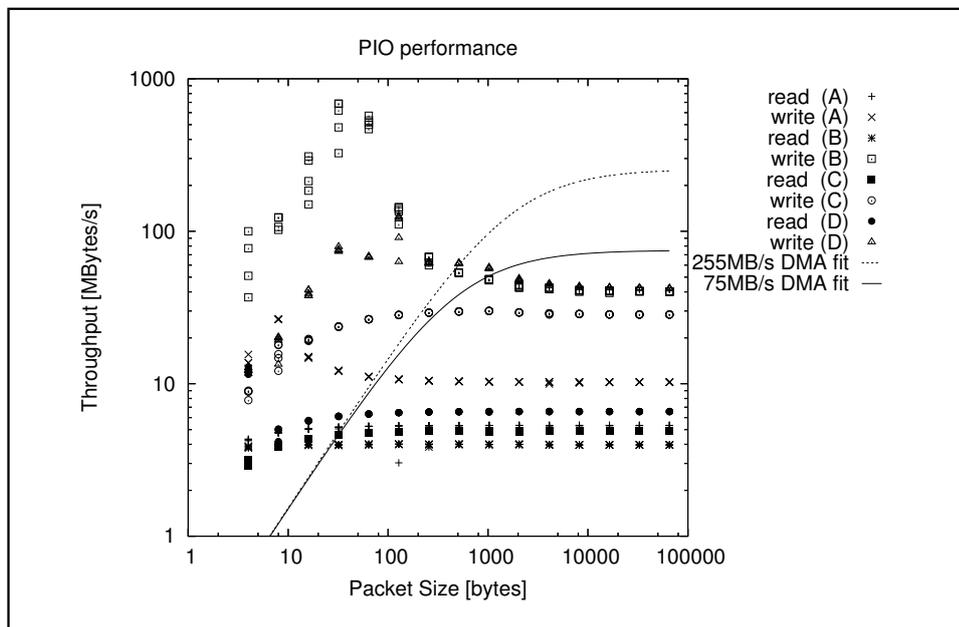


Figure 3.22: PIO Performance

This graph shows the results of accurate measurements of Programmed I/O data transfer between host and coprocessor. Remarkable are the exceedingly high throughput rates for packet sizes of about 100 bytes.

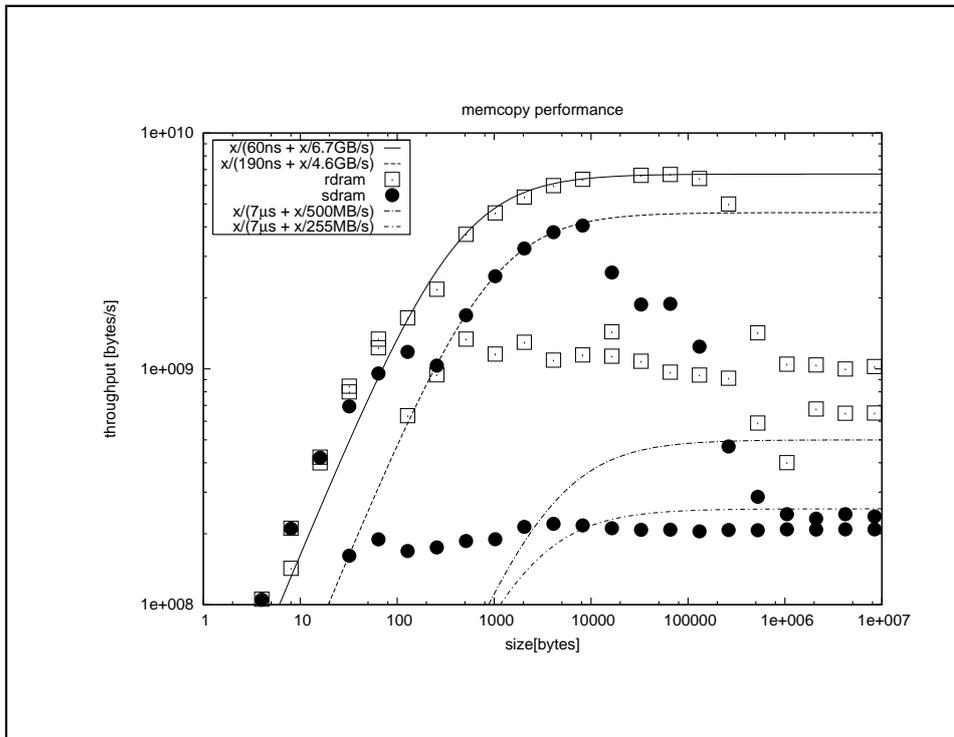


Figure 3.23: Memcopy Performance

The throughput of the memcpy function is measured on two hosts. Minimum and maximum results are given together with fitting curves.

SDK when using Windows as OS. Both packages are stored in a central repository managed by CVS, assuring all development is based on the same up to date sources. CVS also synchronises concurrent development. The CVS repository was continuously managing the source from the beginning of the software's lifetime in 1999 until the present.

Driver

The pciDriver kernel module is specific to the actual Linux kernel "flavour" e.g. being a 2.4.9 RedHat Linux, or a 2.4.18 vanilla kernel from kernel.org, and has to be build and installed separately on every host. This is usually done by exporting the actual sources from CVS into a system directory like `/usr/src` and executing `make; make install`. The differences between the different 2.4 kernel flavours are captured in `#define` directives in the drivers source.

Since the kernel interface changed significantly between 2.4 and 2.6 kernels I have branched a different revision tree for the driver's sources for 2.6 kernels. About one third of the driver had to be changed, however, the function remained the same. Additionally, the makefile for the 2.6 build is different, but the according changes are transparent to the user. Consequently, the only difference to the software and its build is a branch tag in the CVS export statement for the driver package.

Windows is not supported on the IA64 platform. The Linux pciDriver had to be patched against subtle differences in the virtual memory (VM) system. These differences are also captured in `#define` directives in the driver source. However, the main difference between the IA32 and IA64 Linux kernels is the size of memory addresses. IA64 uses 64 bits respectively 8 bytes addresses instead of 32 bits respectively 4 bytes as on IA32. Linux uses integer types for representing memory addresses in the kernel as opposed to pointers in user space. Enabling the pciDriver for IA64 Linux essentially means taking care of using `long` for addresses in the data structures and interface, together with fixing the bugs that were triggered by this change. Refer to table 3.4 for a survey of the difference of the sizes of the fundamental types on IA32 and IA64.

Another issue on IA64 is the size of memory pages. Pages are the fundamental unit of memory and address space organisations. On IA32 a page always spans 4 KiB (4096 bytes). Due to the extended capabilities of the IA64 processors and the support of these by the Linux kernel one can choose between different page sizes when building the kernel. Supported sizes are: 4, 8, 16, and 64 kiB (1 kiB = 1024 bytes) and 4 MiB ($4MiB = 42^{20}bytes = 4194304bytes$). Like most IA64 Linux installations, I have chosen 16 kiB pages on kernel build. The changed page size triggered some additional bugs and made it necessary to export the page size information to the user space, since some optimisations regarding DMA transfers are currently done in userspace.

The interface to the different drivers, being pciDriver or PLX-SDK, is abstracted by the `class Driver` interface (figure 3.24). All the code working

Table 3.4: Size of fundamental Types on IA32 and IA64

type	bytes	
	IA32	IA64
char	1	1
wchar_t	4	4
short	2	2
int	4	4
long	4	8
long long	8	8
void *	4	8
void (*)()	4	8
enum	4	4
float	4	4
double	8	8
long double	12	16

Types with differing size are shown with bold face.

on this interface is essentially OS- (Linux, Windows) and platform- (IA32, IA64) independent. The `instance()` fabric in the Driver class will select the appropriate implementation for the current OS. Figure 3.25 shows the hierarchy of the Driver classes.

Library

In contrast to the `pciDriver` the library executes entirely in user mode. Hence it can be build an installed at any suitable place including the user's home directory.

Usually the file-system on which the library is exported from CVS is mounted on all host systems on which it is to be used. This is done by means of NFS for the Linux hosts and by Samba for the Windows hosts. The library is build with MSVC workspaces on Windows hosts. For the Linux OS I have created a makefile hierarchy that is capable of building different versions of the library dependent on the c++ compiler version and the platform architecture from the same sources and into the same installation directory. Different versions are separated into different subdirectories with their names indicating the platform and the compiler version like `i386-linux-2.95` and `ia64-linux-3.3.2`. The makefiles supports simultaneous builds on different hosts, speeding up the verification of the build. Some simple test applications are included in the library package. The makefile hierarchy allows for simple adding of applications to the package and build.

```

class Driver
{
public:
    virtual ~Driver() {};

    static Driver * Instance();
    virtual Open(...) = 0;
    virtual Close() = 0;
    virtual IsOpen() = 0;

    virtual unsigned * AllocKernelMem() = 0;
    virtual FreeKernelMem(unsigned *mem) = 0;

    virtual LockUserMem(...) = 0;
    virtual UnLockUserMem(...) = 0;

protected:
    virtual OpenDma(...) = 0;
    virtual StartDma(...) = 0;
    virtual CloseDma(...) = 0;
    virtual CancelDma(...) = 0;
    virtual DmaIsDone(...) = 0;

    friend class BridgePlxBASE;
};

```

Figure 3.24: Driver Base Class

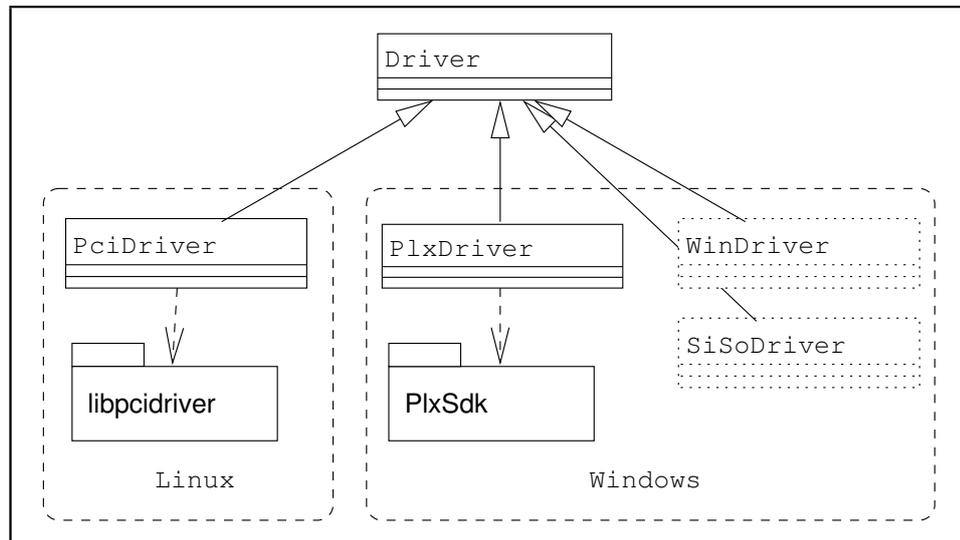


Figure 3.25: Driver Classes Hierarchy
The shaded classes are no more supported.

Linux on IA64

The Itanium workstation that I have been using was shipped with installation media for 64 bit Windows XP and HP-UX. Both OS were not suitable for software development. In the case of Windows this is due the lack of a native compiler and device driver and for the HP-UX it was the lack of device drivers. Also both operating systems are not an option for the ATLAS LVL2 and for most other applications of the software. Consequently I chose to use Linux on this workstation. Initially this was done with a commercial SuSE Linux distribution, but it came clear quite fast that this system was not very suitable. Hence I built a so-called “linux from scratch”⁴. This system is entirely based on free available sources, mostly from kernel.org (Linux) and gnu.org (utilities, tool-chain). Albeit the big effort (the pure compilation time was several days), the resulting system made it possible to stay close to the development, which can be important in the cases of the kernel and the tool-chain. E.g. following recent development of the GNU C compiler dropped the time needed to build the library package on IA64 from about 460 seconds to 160 seconds, which is still a lot compared to 40 seconds on some fast IA32 PC’s.

3.3 Applications

The results presented in this section are, except for the first two subsections, obtained by applying the control software described in the previous sections to FPGA coprocessors and integrating them into external trigger software frameworks and trigger prototypes. Such, the control software demonstrates that it meets the requirements raised in the problem description (section 1.1) and in section 2.2. The results in the first two following subsections have been obtained before the development of the control software, using legacy and commercial software.

3.3.1 μ Enable S-Link

S-Link is the standard interface for moving event data from the detector to the trigger, and the Enable++ feature extraction (FEX) system used S-Link for event-data input and feature-data output (see section 2.2.4).

Our group uses μ Enable reconfigurable coprocessors to interface PCI based hosts to the S-Link LSC and LDC mezzanine cards. I made several tests on the reliability and bandwidth of the μ Enable LSC and LDC tandem in an “S-Link loop-back” setup as sketched in figure 3.26. The FPGA configurations for the μ Enable acting as S-Link sender and receiver have been originally developed by K.Kornmesser using CHDL, though I had to implement a small enhancement to the original, providing reliable transfer of S-Link packets with sizes exceeding 64 kB. These tests ran for days, moving thousands of GB, and no single error was detected when comparing sent and received data. The tests were done in two modes, DMA

⁴linux from scratch, <http://www.linuxfromscratch.org>

and PIO. Figures 3.27 and 3.28 show that the overhead of $350 \mu\text{s}$ introduced by preparing the DMA transfers leads to superior performance of PIO transfers when using packet sizes of less than 2 kB. The saturated bandwidths are about 27 MB/sec for DMA and 5 MB/s for PIO transfers respectively. The sent and received data has to be transferred on the same 32 bit / 33 MHz PCI bus, limiting the available PCI bandwidth to 66 MB/s. [APP98] reports about saturated S-Link bandwidth of approx. 50 MB/s when using a single S-Link per host.

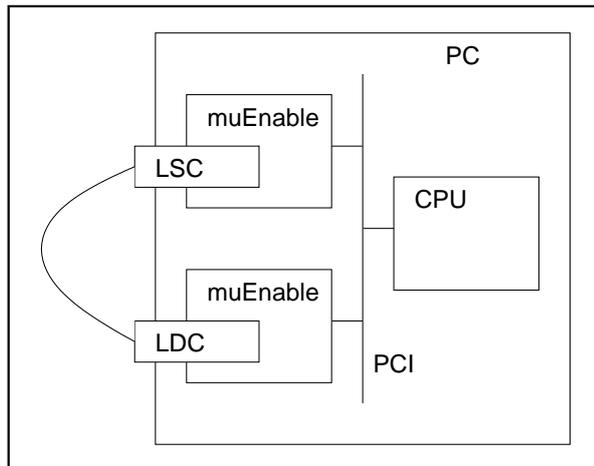


Figure 3.26: Setup for S-Link Loop-Back

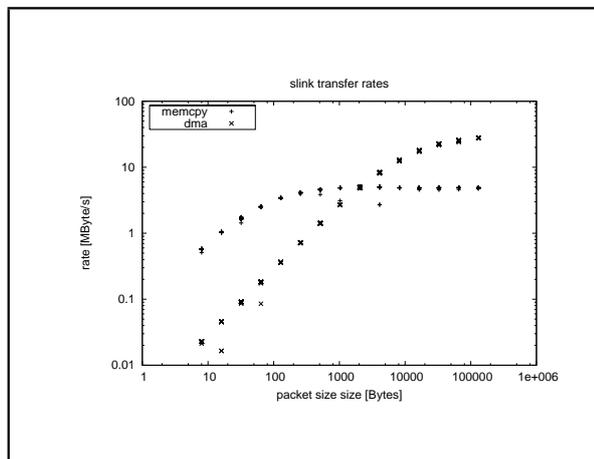


Figure 3.27: Transfer Rates with S-Link Loop-Back

3.3.2 Enable++ TRT Scan

Figure 3.29 shows the data-path in the vertical-slice setup used for tests done with the Enable++ FEX system in the context of “Architecture-A” studies (see section

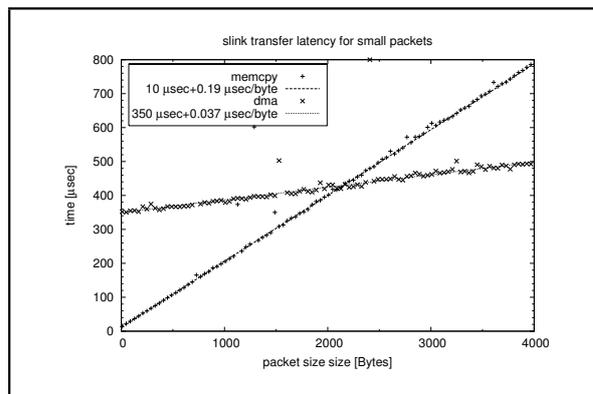


Figure 3.28: Transfer Latency with S-Link Loop-Back

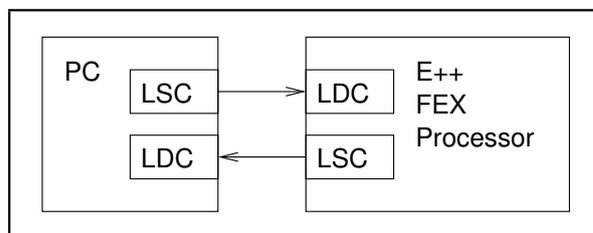


Figure 3.29: Vertical-Slice Prototype using E++ as FEX Processor

2.2.3). I used the vertical-slice setup as a prototype for the “coprocessor” component sketched in figure 3.30.

Figure 3.30 shows the data-paths in the ATM testbed, that was implemented by D.Calvet and I.Mandjavidce from CEA⁵, Saclay, (see section 2.2.5). The ATM testbed was essentially an implementation of Architecture-C, however, the setup in 3.30 uses the Enable++ FEX component from Architecture-A as an enhanced implementation of the DST (destination) component, labelled COP in the figure. Using components from Architecture-A in an Architecture-C setup lead to the naming “Demonstrator A+C” for the shown system.

I integrated the Enable++ system into Saclay’s ATM testbed in winter 1998/1999. The integration was successful, however, the TRT-LUT-Hough algorithm done in the Enable++ system was not executed correctly. The Enable++ project ended with effort, since our group was then concentrating on the next generation FPGA processor, the Atlantis system.

3.3.3 Atlantis Bootstrapping

Bootstrapping the ACB and AIB boards was done parallel with the implementations of the first iterations of the software, which evolved to the software that is described in sections 3.2.2 and 3.2.3.

⁵Commissariat à l’Energie Atomique, Saclay, France

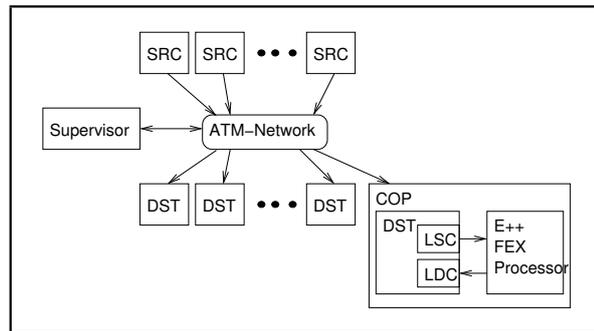


Figure 3.30: Integration of Architecture-A and -C using E++ as Coprocessor

For both boards the bootstrapping requires the initialisation of the PCI bridge's configuration EEPROM that is used to initialise the bridge upon system boot. This is done by connecting the ACB's or AIB's external connector to a PC's parallel port and uploading the configuration data to the EEPROM using a stand-alone application implemented on top of the software.

A similar process is used for configuration of the control-PLD on the ACB and AIB, and the clock-PLD on the ACB.

3.3.4 Atlantis and the Atlas Second Level Trigger Testbeds

The primary application of the Atlantis system is the acceleration of tracking algorithms for the ATLAS second level (LVL2) trigger. Atlantis provides this acceleration as a replacement for the Enable++ FEX system, or as a replacement of standard processing nodes in the current "Architecture-C" LVL2 trigger.

Two approaches for realizing the LVL2 trigger have been followed by different groups in the past: The group from Saclay was investigating ATM networking with a compact software written C language and another group at CERN was working on COTS Ethernet technology with the object-oriented second level trigger reference software (see section 2.2.5). Both systems have been implemented in installations at several locations until 1999, when they have been installed at a common testbed site at CERN during the Pilot Project effort. Since the systems were sharing part of their resources, e.g. most computing nodes, the Pilot Project was giving measures to compare each other and guide future development of the LVL2 trigger.

The Atlantis system was integrated into the common testbed during the Pilot Project. Figure 3.31 sketches the physical integration setup. Atlantis takes the role of a standard processing node. The Ethernet network is used for the testbed system control e.g. for remotely starting and stopping the tasks on the nodes. The ATM network is used to transfer event data from the data sources, in this case readout-buffer emulators, to the processing nodes, and transfer trigger decisions from the processing nodes to the supervisors.

Figure 3.32 shows the software structure on the integrated Atlantis system. The components that are added to a standard processing node by the Atlantis system are

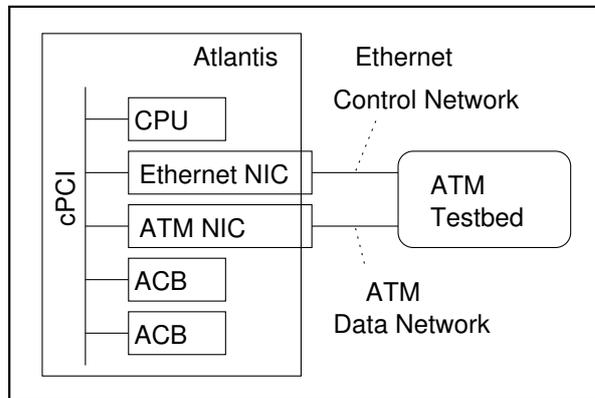


Figure 3.31: Physical Integration of Atlantis in the Testbeds

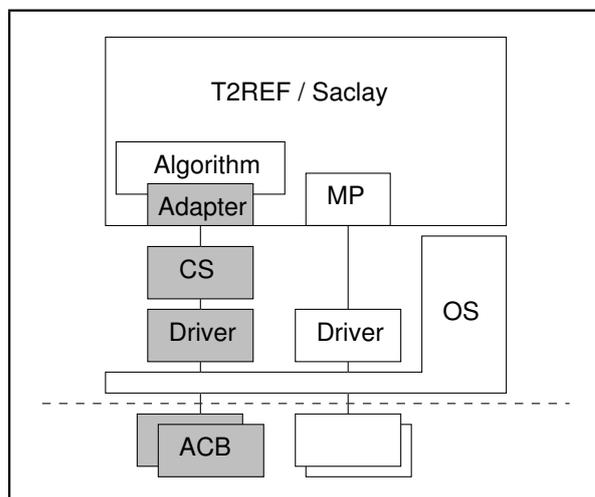


Figure 3.32: Logical Integration of Atlantis in the Testbeds

shown shaded:

- Algorithm Adapter. Modifies the CPU only TRT scan algorithm to redirect hit data to be processed by the ACB.
- CS. The control software used by the algorithm adapter.
- Driver. The device driver that is needed to access the ACB.

On top of the system is the second level trigger reference software (T2REF) implementing the LVL2 trigger. Also shown as a subsystem of T2REF is the message passing (MP) component, which serves as the interface to networking technologies for the LVL2 trigger. The MP is shown separately because some of its implementations require special drivers, like the RAW Ethernet and the ATM MP, to interface to the network adapter card (NIC).

The lower part of figure 3.32 shows the hardware components through which data enters and leaves the processing node.

A simple FPGA configuration was developed by H. Singpiel ⁶, which served as an algorithm stub for the ACB board. The stub allowed for event data to be transferred to the ACB and dummy track feature results to be read back from the ACB.

The integration showed that Atlantis can be made an integral part of the testbed without any modification to the remaining testbed components. Atlantis is seen from T2REF as a standard processing node. Access to the ACB reconfigurable hardware did not interfere with other hardware like the ATM NIC used for event data transfer and the Ethernet NIC used for testbed configuration. The additional device driver used to access the reconfigurable coprocessors (ACB's) and the control software itself did not interfere with T2REF or the device drivers used for the NIC's. An algorithm adapter which redirects event data to be processed by the algorithm stub on the ACB was added to the CPU only TRT scan algorithm in T2REF. Consequently, event data was processed by the ACB with minimum changes to the T2REF.

3.3.5 Atlantis TRT Scan with T2REF

T2REF is not only used in multi node testbeds, but also allows for single node operation. I have prepared the integration into the Pilot Project testbed at CERN using the T2REF in single-node mode in Mannheim. This setup was later used to develop and benchmark a complete implementation of the TRT barrel scan algorithm on the Atlantis system.

To achieve the most seamless integration of Atlantis into the T2REF framework I inserted the CS sources into the T2REF source tree and build system. Resulting, a single build step in the T2REF builds the CS library together with all other T2REF libraries and applications.

⁶Holger Singpiel, singpiel@uni-mannheim.de

In T2REF physics algorithms are captured in `Algorithm` classes. Consequently, the CPU-only TRT-LUT-Hough algorithm is implemented as class `TrtLutHough`. I duplicated and modified this algorithm class to make use of the FPGA implementation of the histogramming, thresholding, and maximum finding steps. The remaining split, merge, and fit steps remained duplicates of the CPU-only algorithm implementation. The resulting class was named `TrtLutHoughAtlantis`.

The ACB is the reconfigurable coprocessor used by the Atlantis system. The implementation of the Hough transformation on the ACB consists of three parts:

- FPGA configuration implementing the Hough transformation histogramming.
- FPGA configuration for upload of the look up table (LUT).
- The LUT itself.

The FPGA configuration for executing the histogramming together with thresholding and maximum finding was developed by H. Simmler using VHDL.

The configuration for the four ACB FPGA's for loading the LUT was developed by K. Kornmesser using CHDL [K⁺98]. The configuration allows for read and write access from the host to all memory locations on all four memory modules that are plugged into the ACB.

Additional to the interface to event data, T2REF provides an interface to runtime configuration data. I used this interface to retrieve the location and file names of the eight FPGA configuration files, four files for the histogramming configuration and four files for the LUT upload. Consequently, it is possible to change the FPGA configuration data without recompiling the software.

The CPU implementation of the histogramming uses a LUT consisting of an array of 96.000 lists of 130 16-bit histogram-counter-id's. However, the FPGA implementation requires a different LUT, where the histogram counters are selected by the addresses of bits set to 1 in the very long ACB memory module data words ($4 \cdot 178 = 712$ bits). Also the resolution of the FPGA LUT is somewhat reduced to 76.000 roads as compared to the CPU LUT. I implemented some classes to simplify and encapsulate the transition from the CPU LUT to the FPGA LUT. On top of these classes is a `TRTMemoryHostRep` class, which allows for bitwise access to a representation of the FPGA LUT in host memory for setting up the LUT entries. This host representation of the LUT can then be transferred to the memory modules on the ACB by DMA, using the FPGA configuration mentioned above. I also replaced the use of file-stored road bundles as present in the CPU based TRT-LUT-Hough algorithm with on-the-fly calculation of the roads.

T2REF provided a single node application for benchmarking the CPU version of the LUT-Hough algorithm using event data from disk storage. I modified this application to alternatively make use of the FPGA implementation of the algorithm instead.

The principle steps for executing the LUT-Hough algorithm on Atlantis with the single node application is as follows:

1. Calculate LUT and prepare host representation of the LUT.
2. Configure the four ACB FPGA's for upload of LUT.
3. Upload the LUT to the memory modules on the ACB's.
4. Reconfigure the FPGA's for the histogramming algorithm.
5. Process data
 - (a) Receive event data from the T2REF framework.
 - (b) Reformat event data and transfer it to the ACB by DMA.
 - (c) Receive track candidates from the ACB using DMA.
 - (d) Perform the remaining split, merge, and fit algorithm steps on the host CPU.
 - (e) Report final track data.
 - (f) Repeat until no more event data available.

The above scenario not only exploits the speedup in the parallel FPGA implementation as compared to the sequential CPU implementation, but also uses the runtime-reconfigurable nature of the ACB. For simplicity and effectivity the functionality of LUT-upload and histogramming was separated into two different FPGA configurations, which are uploaded on the FPGA's one after another. The LUT data stored in the memory modules in step 3 keeps present during reconfiguration (step 4) of the FPGA's and the following event data processing (step 5).

3.3.6 Parallel TRT Scan on 2 ACB's in Atlantis

The cPCI crate that houses the Atlantis system has 6 cPCI slots. However, only 2 ACB coprocessors were built in Mannheim. We made tests on executing the TRT-LUT-Hough scan algorithm on an Atlantis system equipped with 2 ACB coprocessors. In this tests the same event data was simultaneously transferred to 2 ACB boards by DMA, taking advantage of the cooperative scheduling of DMA chunk transfers by the PCI arbiter. The PCI transfer rate on that system was measured to be approx. 40 MB/s, which is poor compared to the theoretical 132 MB/s that could be expected from a 32 bit / 33 MHz PCI bus. Accordingly, the use of interleaved DMA transfers to 2 boards did only marginally enhance the throughput of the system. The activity diagram in figure 3.33 illustrates the parallel TRT-LUT-Hough algorithm. More details and exact numbers can be found in [HKM⁺00].

Concludingly, the performance of the FPGA-accelerated TRT-LUT-Hough scan was limited by the poor DMA performance of the different cPCI hosts which were investigated, and even on modern architectures like 64 bit / 66 MHz PCI buses with MPRACE coprocessors the performance of the combined system is often limited by the data transfer between host and coprocessor.

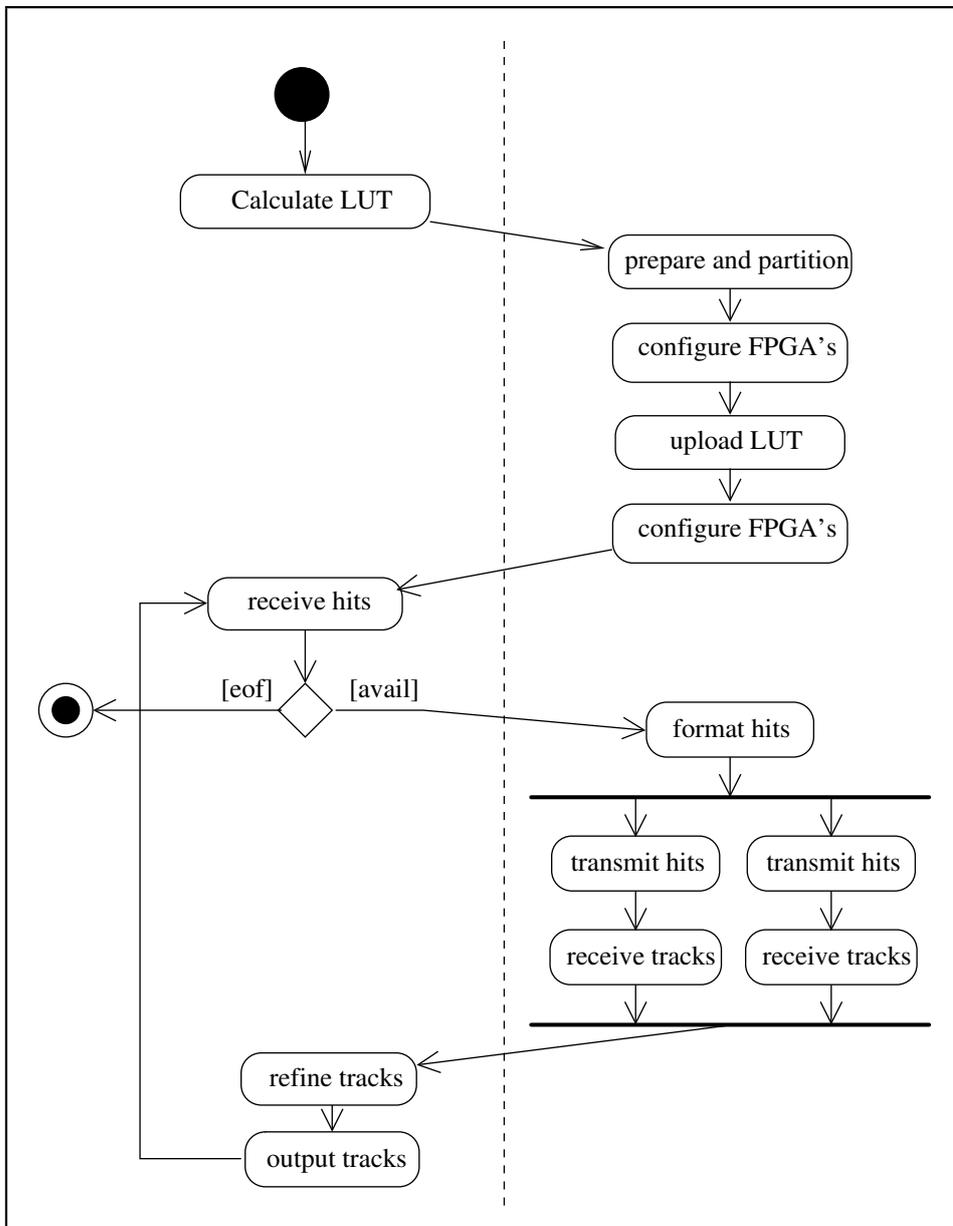


Figure 3.33: TRT Scan with Atlantis, Activity Diagram

The diagram shows on the left side the activities from the original CPU based implementation of the TRT-LUT-Hough algorithm that remained unchanged. The activities on the right side are added or changed in order to transfer the LUT to the two ACB's and to redirect hit-processing to the ACB's. Since the ACB's can simultaneously transfer data by DMA and can compute the histograms independent of each other, the histogramming step is implemented by two simultaneous threads. (However, the host PCI bus is scheduled by the host's PCI arbiter between the two requesting DMA engines.)

3.3.7 MPRACE TRT Scan with T2REF

MPRACE is the latest reconfigurable coprocessor built in Mannheim. MPRACE is supported by the control software just like the ACB. Because of the steady evolution of FPGA technology the LUT-Hough-histogramming now suits on the single-FPGA MPRACE board. My colleague A. Khomich ⁷ has developed the FPGA configuration for the algorithm, histogramming, thresholding, and maximum finding, and reused the infrastructure that was provided by the integration of the Atlantis system. To my knowledge, his implementation is the first one to reliably execute the FPGA based TRT-LUT-Hough-transformation on big sets of event data, providing exactly matching results when compared to the CPU only implementation [BHK⁺03]. This is a remarkable step, since former implementations (Enable++ and Atlantis) have been suffering from more or less frequent failures in executing the algorithm. Khomich reports speedup factors greater than 2 when comparing the whole algorithm which is distributed between host and coprocessor [HKK⁺04]. The speedup of the histogramming step is much greater, however, the overhead introduced by data transfers between host and coprocessor limits the achievable performance gain.

3.4 Reconfigurable Software

3.4.1 Introduction

The control software for reconfigurable coprocessors (CSRC) described in the previous sections supports a variety of hardware platforms. The diversity of coprocessors is reflected in the software by sub-classing of common base classes and coprocessor specific relations between the objects that model the hardware in software. The reconfigurable nature of e.g. the central control PLD present on all supported coprocessors is considered by implementation of classes that capture functionality of the configuration of the PLD. Therefore, reconfigurability of the hardware is only supported at compile-time. Reconfiguration of e.g. the coprocessor's FPGA at runtime is possible but not reflected in the software. Another shortcoming of the current software is the strong coupling of some of its components, hindering encapsulation and maintainability.

The following describes an approach that reflects runtime reconfiguration of hardware by runtime-reconfigurable software. To achieve this, an indirection layer is introduced into the software architecture. This layer aims at abstracting away the interface of the class that encapsulates the software-representation of a hardware device. The use of this abstract interface enhances encapsulation, facilitates reuse, and gives better possibilities for debugging and monitoring. As the major outcome, the modified architecture will enable run-time dynamic connection of the components of which the software is built from, as opposed to the compile-time fixed connection scheme that has been followed in recent past in software.

⁷Andrei Khomich, khomich@ti.uni-mannheim.de

3.4.2 Bottom-Up Design and Activity Flow

Resembling object oriented analysis and to promote software modularity, independent hardware components are described as classes in the software. All information about the device that are relevant for the software are captured by these class definitions. Examples for such information may be line protocols, signal levels, or internal register offsets.

An exemplary reconfigurable coprocessor (RC) architecture is sketched in Figure 3.34. In the following, part of the clocking system of this RC is described at a glance, highlighting its architectural relevance for the software.

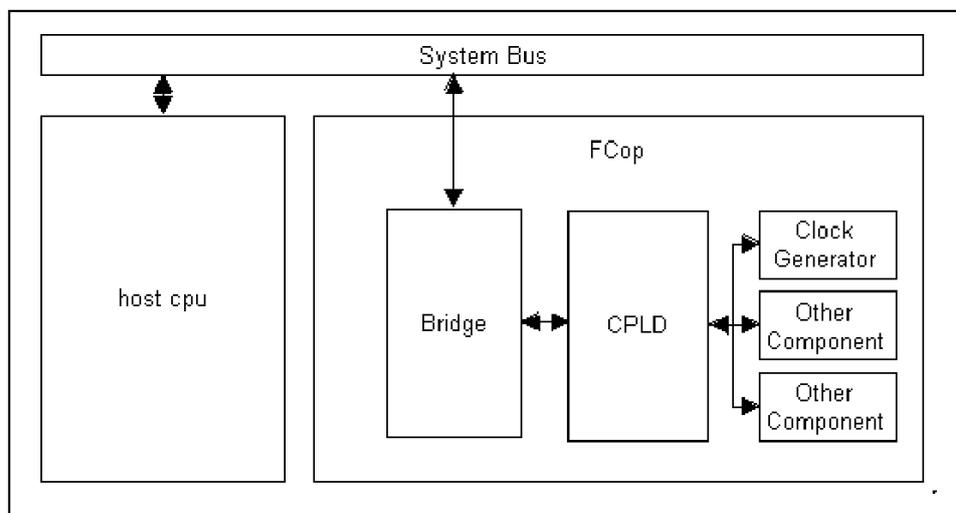


Figure 3.34: Architecture of a Reconfigurable Coprocessor

A programmable clock generator (CG) is used on the RC to generate an adjustable clock signal from a fixed-frequency crystal oscillator. The CG is programmed by applying commands to a serial input signal. The signalling uses a serial line protocol specific to that CG chip. The programming input of the clock generator is connected to an output of the system integration PLD of the RC. The PLD in turn is connected to a bridge using a local address/data multiplexed bus. The bridge connects to a host bus.

The description given above starts at the clock generator and ends at the host bus. Following an alternative view is given, describing the physical activity flow through the system: The bridge of an RC is used to map transactions from the system bus, e.g. PCI, to the RC local bus. Connected to the local bus is a PLD that captures some of the transactions on bus and maps them to reads and writes of registers internal to the PLD. A bit in one of these registers is implemented to represent the output level appearing on the PLD's clock generator output. The output is connected to the command input of the clock generator.

Two contrary viewpoints of the RC architecture are given above. These are referred to as *software* view and *hardware* view.

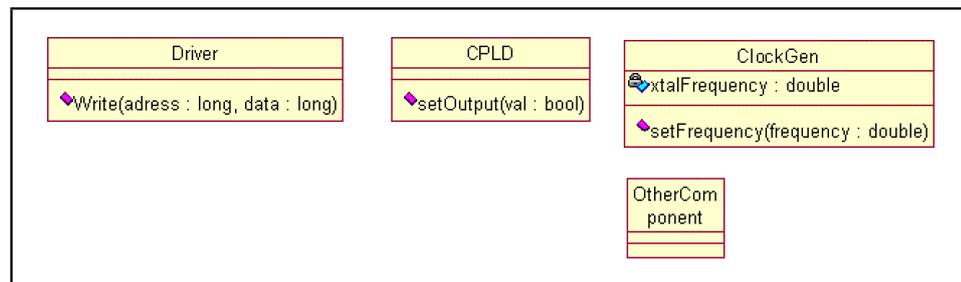


Figure 3.35: Software Representation of Hardware Components

The hardware components as described above are represented as classes in the software view of the system. The following code shows the class declarations:

```

class ClockGen
{
public:
    void setFrequency(double frequency);
};

class PLD
{
public:
    void setOutput(bool value);
};

class Driver
{
public:
    void write(long address, long data);
};
  
```

Figure 3.35 illustrates the classes in an UML class diagram. The interfaces given above are independent of each other. However, the class methods have to be tied together in an collaboration to achieve the desired activity, e.g. setting the clock frequency.

```

ClockGen clockGen;
PLD    pld;
Driver  driver;

void ClockGen::setFrequency(double f)
{
    Bitstream bs = calculateBS(f);
    for(i=0;i<bs.length;i++)
        pld.setOutput(bs.at(i));
}

void PLD::setOutput(bool v)
{
    driver.writeLocalBus(PLD::outputreg, v);
}

void Driver::writeLocalBus(long address, long data)
{
    Transaction t(WRITELOCAL,address,data);
    ioctl(fp,&t);
}

main()
{
    clockGen.setFrequency(50e6);
}
  
```

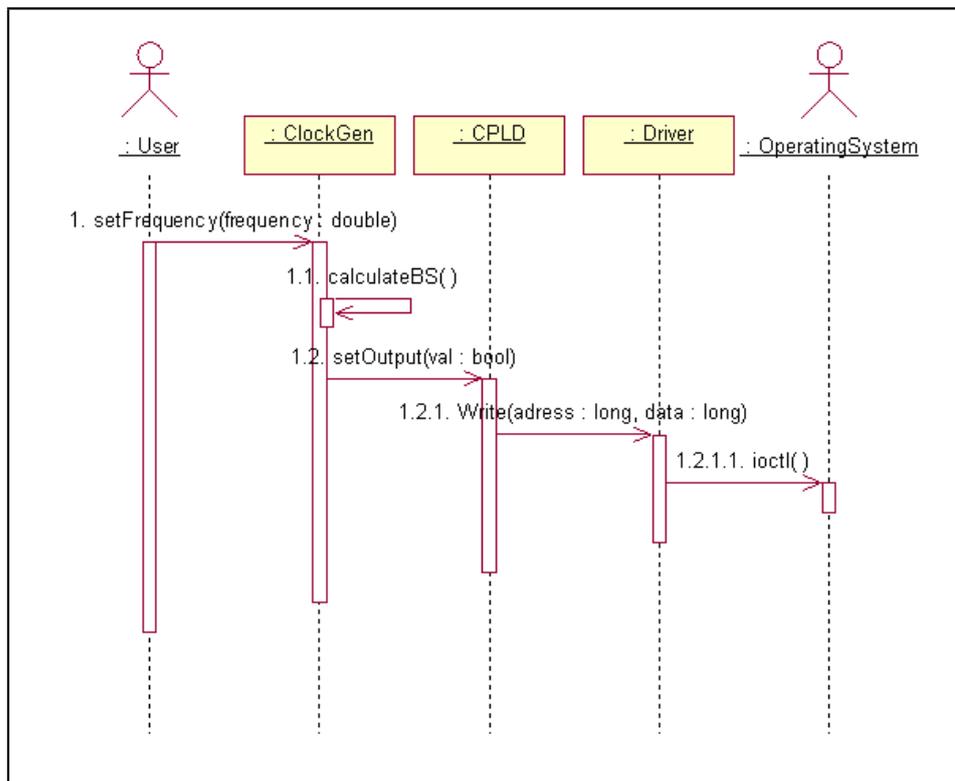


Figure 3.36: Clock Setting, Sequence Diagram

Here, the activity is triggered by invoking the `setFrequency` method of a `ClockGen` object. For this simple example, there would be no need to create class instances, since the classes defined so far do not contain any data. However, most real implementations would require local data, e.g. for storing state and for referencing collaborators.

To request action on the *target* device, e.g. the physical clock generator, the user employs the *software-representation* of the clock generator. An object of class `ClockGen` is used to trigger action on the physical clock generator. The software activity flow starts at the software-representation of the target as opposed to the hardware signalling on the RC, which starts at the host bus interface.

Figure 3.36 shows the sequence diagram resulting from the above scenario. The activity flow is initiated by the user, promotes to the software-representation of the target device and is routed through intermediate device representations (the PLD and the Driver representations) until it leaves the scope of the software. The direction of activity flow is opposite to the *hardware* action flow: hardware driver, hardware PLD, hardware clock generator, and finally clock signal on the hardware.

This architecture has proven to be highly adaptable and maintainable, whilst imposing too many internals of the RC structure to the end-user. Therefore, although the component-wise decomposition well suits for the representation used

internally by the software, additional abstraction levels are implemented to hide internals from the user.

```
// An API hiding much of the internals.
class RCApi
{
private:
    ClockGen gen;
public:
    void setClockFrequency(double frequency)
    {
        gen.setFrequency(frequency);
    }
};
```

RCApi is a *fat* interface. The corresponding problems, concerning e.g. maintainability and complexity, will not be discussed here.

3.4.3 Collaboration

The code fragments shown in the last section sketch a scheme for collaboration of the class representations of devices. Although being lean and effective, this scheme imposes problems regarding encapsulation, reuse, and the support for runtime re-configuration of the hardware.

The collaboration framework of the objects making up the software-representation is an important part of design. It affects many important runtime architecture and performance issues in the software, of which the following gives a survey:

- *Collaboration Dynamics* The components are hard-wired at compile-time, the topology can't be changed at run-time.
- *Runtime Performance* Code following the above hard-wired scheme should yield near-optimal performance. Improvements would require a flat (i.e. not layered) architecture.
- *Operational Debugging and Monitoring* can only be introduced by instrumenting the code at compile-time (“printf-debugging”).
- *Encapsulation* is weak, since every class knows about all other classes involved in the collaborations, and the global name-space is spoiled by global class instances `ClockGen gc; PLD pld; Driver driver;`. Method implementations refer to the specific interfaces of collaboration partners.
- *Reuse* is rendered difficult and requires introduction of abstract base classes.
- *Maintenance* suffers from weak encapsulation.
- *Tangibility* The code is straightforward and easy to understand.

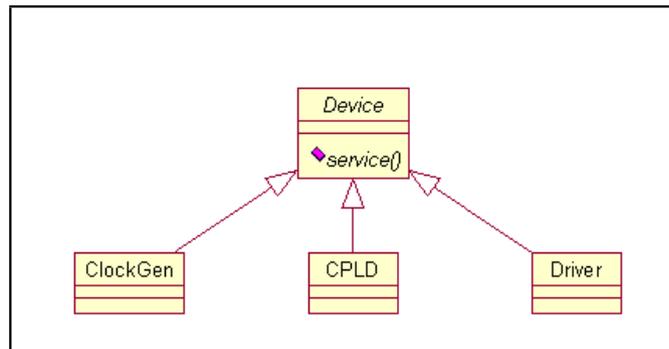


Figure 3.37: Abstraction of the Device Representations

3.4.4 Service Abstraction

To overcome the drawbacks in the present framework, as outlined in the last section, a new architecture element is introduced. *Abstract services* are used to promote encapsulation, reuse and flexibility, and to enable runtime connection dynamics and operational monitoring and debugging. Since this new approach introduces indirection into the software it can be assumed that performance degrades and tangibility is reduced. Advanced encapsulation and runtime connection dynamics are demonstrated below by reporting from a prototype implementation. Performance degradation will be quantified by measurements comparing prototypes of the new and the old approaches.

Figure 3.37 illustrates the consequences of introducing an abstract service operation into the device representations used throughout this paper and in the prototype implementations.

Here, the user requests actions from physical target devices by calling abstract service methods of the physical device's software-representations. The requests are transformed and/or modified within the software-representations of the target devices. This is done by the device's methods and attributes, which trigger one or more abstract services at the software-representations of intermediate or endpoint devices. An arbitrary number of intermediate stages can be involved until the transformed request(s) finally reaches one or more endpoint devices, and leaves the host system. The collaboration diagram 3.38 illustrates the travelling of a request through a chain of abstract service-providing classes. Different types of services may be required for implementing complex behaviour. Such may be *actions*, triggering an activity without invoking data, e.g. the generation of a clock pulse, *outputs* that carry data, e.g. setting a specific logic level on a signal trace, and *inputs* that request data from a device, like the locking status of a PLL. The class diagram 3.39 illustrates a device that serves multiple requests behind the abstract service interface.

A service identifier is used in the client-server interaction to address the different services. Transferral of service identifier from the requesting client to the

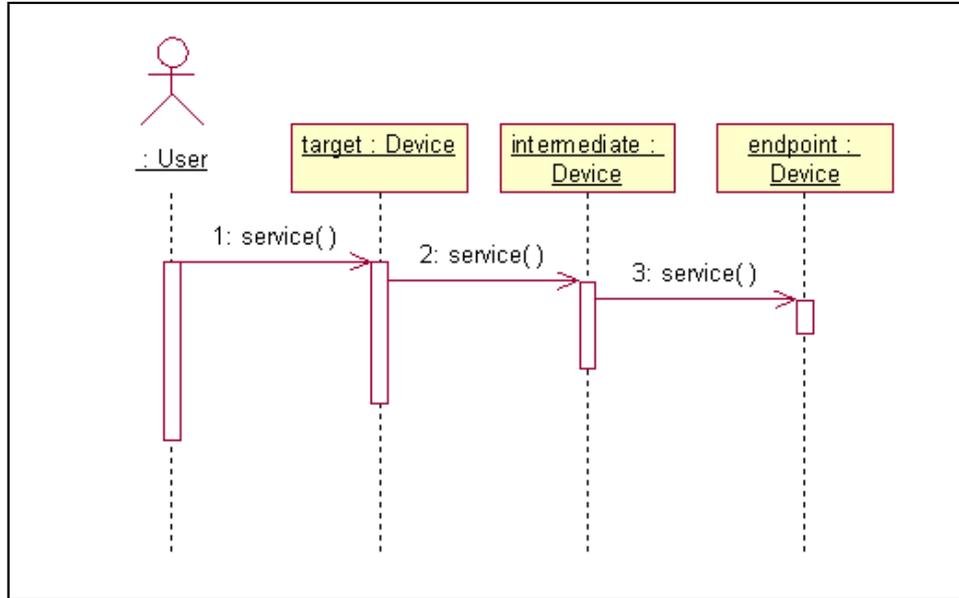


Figure 3.38: Abstract Services, Collaboration Diagram

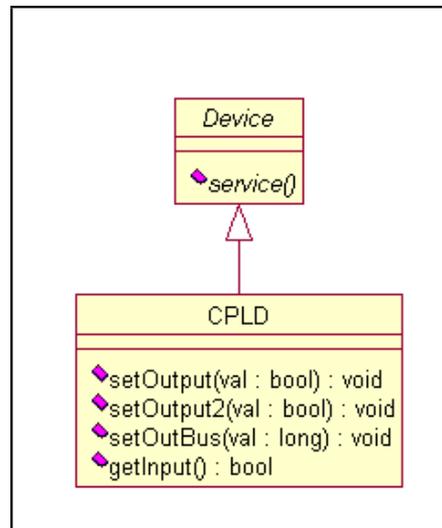


Figure 3.39: Multiple Services

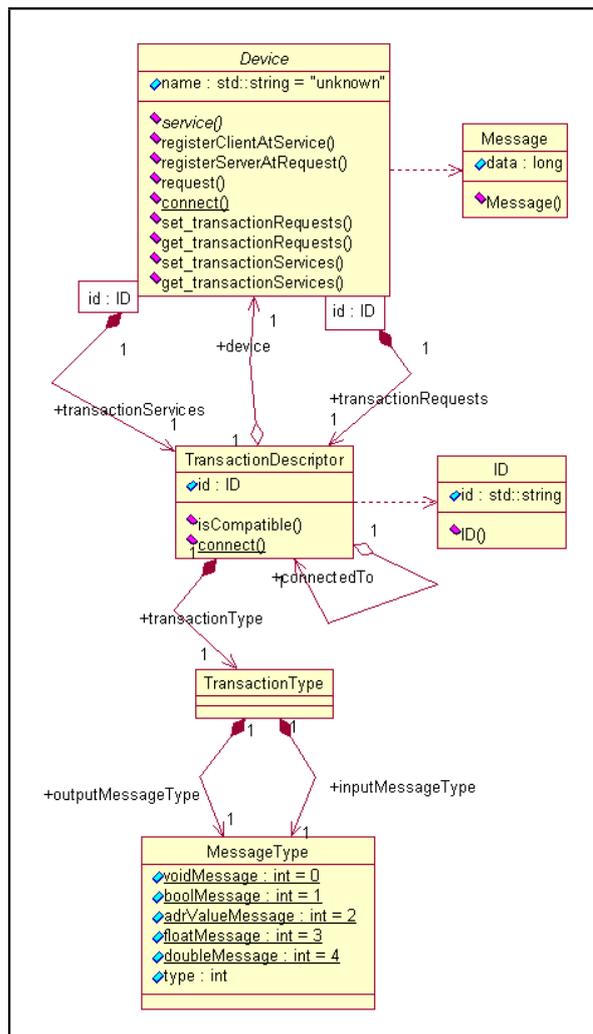


Figure 3.40: Device Base-Class and its Attributes

request server is implemented by passing an argument of type ID to the `service` operation of the server device. The server device selects the appropriate service, and transmits the request to that service.

3.4.5 Dynamically Connected Devices

As aforementioned the most important disadvantages in the presented collaboration framework are the strong association between the devices involved, the very poor re-usability, and the compile-time fixed connection scheme, whereas a run-time dynamic connection scheme is desired.

A run-time dynamic interconnection scheme would involve some far reaching modifications:

- provision of a uniform interface to the involved devices
- abstraction of a transaction interface
- ability to query the devices about available transaction interfaces
- dynamic connection of one device's request with another one's services,
- routing the transactions through the dynamic interconnection using the abstract device interface
- type checking to ensure that only suitable interconnections are set up

Class diagram 3.40 shows a pattern satisfying the above criteria. The implementation details of this pattern are described in the following.

Transactions

To describe the different transactions that may be triggered on a particular device or which a device may request, objects of class `TransactionDescriptor` are used. An instance of `TransactionDescriptor` owns an attribute describing the type of transaction, and publishes methods to check if another instance has the same transaction type and to connect this descriptor to another one. To implement the request/service pair connection, a pointer to the device that possesses this transaction descriptor, and a pointer to a corresponding transaction descriptor are used.

Requests and Services

For a dynamic interconnection of the devices through a uniform interface it is necessary to create an abstract interface to the *ports* to be connected. Two stereotypes are created to facilitate this. First, a device may *request* a transaction from another one, second a device may *service* a transaction to another. Requests and services are identified using identifiers (ID). To establish a connection, a particular transaction request of one device is bound (connected) to a particular service of another device. This connection is realized using the `TransactionDescriptors` describing the devices requests and services.

Uniform Interface

A uniform interface is provided by declaring an abstract base class `Device` (see Figure 3.40). Implementations of device representations are supposed to inherit from `Device`. The following code shows the abstract base classes declaration and, as does Figure 3.40, the key properties of `Device`.

```
class Device
{
public:
    typedef TransactionDescriptor TD;
```

```

typedef std::map<ID, TD> TMap;
Device();
~Device();
virtual void service (const Message & message,
                     Message & messageOut,
                     const ID & serviceID) = 0;
void request (const Message & message,
             Message & messageOut,
             const ID & requestID) ;
static void connect (Device & requestDevice,
                   const ID & requestID,
                   Device & serviceDevice,
                   const ID & serviceID);

const std::string get_name () const;
void set_name (std::string value);

const TMap& get_transactionRequests () const;
TD& get_transactionRequests (const ID & id);
void set_transactionRequests (const ID & id,
                             const TD & value);

const TMap& get_transactionServices () const;
TD& get_transactionServices (const ID & id);
void set_transactionServices (const ID & id,
                             const TD & value);
private:
    std::string name;
    TMap transactionRequests;
    TMap transactionServices;
};

```

A Device maintains two collections of TransactionDescriptors, one for requests and the other for services. These collections are hashed with an ID to be uniquely identified.

Concrete Devices

The class diagram 3.41 shows examples of concrete devices derived from the abstract baseDevice. The stereotypes <<service>> and <<request>> are used to mark the methods that are implemented to build an interface to the actual devices implementation.

Children of Device will have to register their available requests and services, for example in the constructor, at their transactionServices and transactionRequests hash tables before these transactions can be used from other devices or the application.

The transaction can be routed through the device net after the request of one device has been connected with the service of another device. Request/service connection is accomplished by connecting the TransactionDescriptor describing the request or service.

The different device's <<request>> methods are responsible for calling the device's request method with the actual requests ID as a parameter. request is implemented in the base (Device) class. request finds the corresponding service by inspecting this requests TransactionDescriptor which is identified by its ID. The serving device's service method is then called. This service method is abstract (pure virtual) in Device and must be implemented

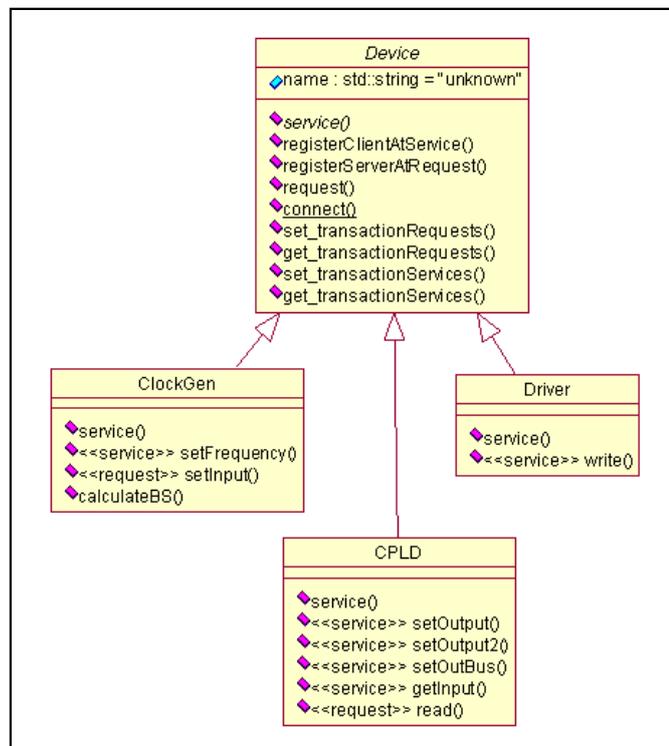


Figure 3.41: Class Diagram of Concrete Devices and Base Class

in every device derived from `Device`. `service` will then dispatch the request to the corresponding `<<service>>` method of the target device.

Runtime Connection

The following code illustrates the connection of devices at runtime and the triggering of a transaction.

```
Device * gc=new ClockGen;
Device * pld = new PLD;
Device * driver = new Driver;

Device::connect(
    &gc, "requestIn",
    &pld, "serviceOut");

Device::connect(
    &pld, "requestRead",
    &driver, "serviceWrite");

Message mIn;
double f=6e9;
mIn.set_data((long)&f);
Message mOut;

gc->service(mIn, mOut, "setClock");
delete gc;
delete pld;
delete driver;
```

The `connect` method works on `Device` pointers, ignoring their actual sub-type. Also the call of `gc->service` is a call to the (abstract) base class `Device`.

3.4.6 Performance Evaluation

In this section the transaction performance of the two connection schemes is compared by measurements. A prototype implementation of the *fixed* connection scheme as described in section 3.4.2 is opposed to a prototype implementation of the *dynamic* connection scheme as described in section 3.4.5. With both prototypes an imaginary transaction is modelled. This transaction involves the clock generator, the PLD and the driver, and is triggered by the user application requesting a service from the clock generator. Refer to figures 3.36 and 3.38 respectively. The operating system `ioctl()` call in the `Driver` class is replaced by the increment of a counter local to the `Driver` object. The counter value provides the transaction count used in the measurements.

For every transaction service counted by `Driver` a transaction originating from `clockGen` with destination `pld` as well as a transaction originating from `pld` with destination `driver` occurs. The execution time for the whole three - stage transaction chain is measured.

Table 3.5 summaries transaction chain execution time measurements. Execution time is given in units of processor clock cycles. Also the size of the source code for the two prototypes is compared.

As can be seen, the dynamically connected scheme runs approx. 10 times slower for both debug and optimised builds.

Table 3.5: Performance Comparison

	clock cycles/transaction chain		lines of code
	debug	optimised	
dynamic	1130	265	970
fixed	90	28	71
ratio	13	9.5	13

The performance overhead due to the use of the dynamically connected scheme seems to be big enough to reason not to use this design pattern. But it is necessary to weigh up the software overhead to the usual bottleneck in a system consisting of a host and an FPGA based processor. Most FPGA based (co-)processors are connected to the controlling host using a PCI bus. Though a current PCI bus is clocked with 33 MHz, a typical programmed I/O (PIO) transaction on the PCI bus lasts about $0.5\mu s$ according to a transaction rate of 2MHz. In Table 3.5 an additional duration of about 250 processor clock cycles for the dynamically connected scheme is indicated. On a standard CPU running with 1 GHz this relates to $0.25\mu s$.

To summarise, the performance of a typical PCI hosted system would degrade to 66% using the dynamically connected scheme compared to the fixed scheme.

3.4.7 Dynamic Loading of Components

Introduction

The last sections described how one can realize a dynamic collaboration between objects that represent devices of a compound system. Although the topology of the collaboration can be set up and changed at run-time, the set of object-classes that participate in the collaboration is fixed at compile-time.

The following describes an extension with the capability to dynamically link at *run-time* to libraries containing additional component classes. The libraries are identified by their filenames, and software components defined in the libraries are made available to the system by *prototypes*. Thus, by retaining the abstract interfaces developed in the preceding sections, it is possible to compose a collaboration from a dynamic set of components at runtime. The composition is done without referring to implementations of the components as building blocks.

Accordingly, it is possible to extend and adapt the software at *run-time* according to the needs of reconfigured hardware, or to decorate or modify the collaboration e.g. to enable simulation of part of the system or to facilitate debugging and monitoring without recompiling the software. Additional components can be build independently of and parallel to the existing runtime.

This leads to the possibility to build the collaboration from a database describing the hardware's properties. Another possibility would be to use a graphical

editor to compose the components to a running system.

The performance overhead due to software indirection is monitored and the correctness of architecture is verified by implementing a functional prototype.

Relation to CORBA

CORBA [OMG], in conjunction with IDL, defines a framework that hides implementation details of software components even at a language independent way. CORBA aims at connecting objects in a network distributed system using central middle-ware services implemented by object request brokers (ORB). In contrast, the presented approach does not hide the implementation language (C++), nor does it address network-distributed systems. I carefully monitor potential performance degradation, whereas CORBA's performance is depending on proprietary ORB's and suffers from additional overheads.

Registering Implementations using Prototypes

Because I aim at operating system and compiler vendor independence I use standard c++ techniques to extend the application at runtime. Three main ingredients are used to achieve this: *shared libraries, static data, and prototypes*.

Many modern operating systems support the concept of **shared libraries** (SL), i.e. sharing common code among different applications. Different concepts exist for *exporting* symbols to clients. I do not rely on operating system dependent code for looking up symbols in the SL, since no symbols are exported from the SL. The way the SL is loaded and unloaded at runtime remains dependent on the OS.

Common infrastructures of SL's are implemented in a way that SL **static data** is initialised when the SL is loaded and finalised when unloaded just like it is done with application static data on start-up and exit. Initialisation and cleanup of static data local to the SL is used to register the SL's functionality within the application upon loading. The interface between the SL and the application is defined within a common library `Device`. All SL's and the application have to be linked against `Device`.

[GHJV95] describes the **prototype** design pattern as a way to extend applications at run-time. The prototype pattern is especially useful if the classes that are used to extend the application are not fixed at compile-time. In the prototype pattern objects are created by *cloning* from prototypes.

In this implementation the prototypes are instances of classes derived from `class Device`. These proto-instances are created as static data upon loading of the SL's that define the classes. During creation, the prototypes register themselves in a global static map that is available to the application, so that they can be looked up by name. After lookup the prototypes are to be *cloned* before they can be used by the application.

This prototype-self-registration makes it possible to access the functionality of the newly loaded classes without referring to their implementation (i.e. without the

need to export symbols from the SL).

Implementation

The following code shows the use of SL's in an application similar to the application presented in the previous sections. In contrast to the former example here the different devices' implementations are loaded dynamically at run-time from SL's and are not linked statically at compile-time. This is done by creating three `Component` objects using the SL's' base-names as parameters to their constructors. The SL's are unloaded upon destruction of the `Component` objects.

```
Component clockgen_component("clockgen_component");
Component driver_component("driver_component");
Component pld_component("pld_component");

Device * p_gc      = Device::getCloneByName("ClockGen");
Device * p_pld     = Device::getCloneByName("PLD");
Device * p_driver  = Device::getCloneByName("Driver");

Device::connect(
    p_gc, "requestIn",
    p_pld, "serviceOut");

Device::connect(
    p_pld, "requestRead",
    p_driver, "serviceWrite");

Message mIn, mOut;
double f=6e9;
mIn.set_data((long)&f);

p_gc->service(mIn, mOut, "setClock");
```

Performance measurements carried out with this application show similar results as compared to the results obtained in the previous sections.

Chapter 4

Discussion and Conclusions

The literature review (section 1.2), though being the excerpt of a comprehensive scan of current and recent publications and textbooks, showed that there is very little information available to which this thesis can be related. I assume that there are two reasons for this. First, the number of publications that deal with such a particular task as it is handled in this work is very low, what may result from the fact that most software that is used for controlling reconfigurable coprocessor is commercial and proprietary. Second, the task of creating such software may seem trivial. However, I hope that this work shows that there are many problems and properties that should be addressed if one tries to develop a stable, powerful, maintainable, extendable and effective control software, in particular if one considers the unstable but tight requirements that are caused by the hardware and the envisaged applications.

The most of the reviewed literature only indirectly relates to the work which is presented in this thesis. Some deal with, eventually distributed, real-time systems, [SK96]. Others address design flow [HA96, EM00, K⁺98, GLS99, Sys], co-synthesis [DJ98, BMBG02, Rüh01] or reconfigurable architectures [Hau98b]. Scheduling of configurations and integration into operating systems (OS) were addressed in [Sim01, WP04]. [HKT03] extends OS related issues to distributed systems. Software reuse is currently addressed with software components [Szy98], and is an everlasting topic in the literature. Related to components is the concept of middle-ware for connecting distributed, incompatible, or legacy software and computer systems [YBGL02, YCBL03, HPK03]. Software process and life-cycle is addressed in many textbooks. The effects of the use of software prototypes are quantified in [Kin95].

The Enable++ stand-alone FPGA processor is a powerful FCCM. While I was engaged in preparing the integration of Enable++ with the ATM testbed I learned that the maintenance of Enable++ is difficult due to its physical and logical complexity. Additional problems arose from previous fluctuations of manpower as it is common in a university environment. Also, the life-cycle of the Enable++ was depending on architectural requirements in the ATLAS second level trigger, which

changed significantly after developing Enable++, what finally led to abandoning development of stand-alone FPGA processors and triggered the development of a family of less complex FPGA coprocessors. I suggest an even stricter life-cycle planning for such a complex machine to reduce the risk of losing continuity in maintenance and development efforts.

The choice of a spiral software process (iterative prototyping) was justified by the need for the early delivery of tools and the unstable software requirements. Resulting from the use of the spiral process were small iterations of the software with high visibility, low development risk, and good compliance to schedules. However, it became only lately visible that the process can lead to poor overall design because the small iterations can cover possibly better approaches. This became obvious in the `registry` package, that breaks encapsulation, and whose maintenance tends to be cumbersome. The process favours ad-hoc solutions, which may establish in the software. Such ad-hoc solutions can become a burden for maintenance, tangibility, and further development in general.

Bottom-up design is used for most of the software. I favoured bottom-up over top-down because of different reasons. First, the bottom-up design reflects problem analysis. Such analysis was done by decomposing the hardware model into distinct and independent components. Then, the first application of the software was the bootstrapping of the Atlantis ACB. Bootstrapping a newly developed hardware usually requires low level access to the components of the hardware for monitoring and debugging. Also, for some components, an initial configuration has to be made, e.g. by uploading configuration data for PLD's. For doing that, tools need to be implemented. Finally, the software shall be configurable and extendable to cope with firmware and design update of a specific hardware, to enable integration of hardware expansion modules into the software, and to facilitate support for future hardware.

Several results suggest that the bottom-up design successfully addresses the aforementioned issues. First, the software integrates support for different FPGA coprocessors through software extension and without redesign. Also, the support for newer hardware does not break support for older hardware although a great part of the software massively exploits reuse. And many of the supported FPGA coprocessors have been bootstrapped using tools composed from the low level software components that resulted from the bottom-up design. A critical problem with bottom-up design is the poor definition of stable user API's. Such API's were only lately introduced into the software in an ad-hoc manner. I assume that a good design approach would have to use bottom-up design *together* with top-down design, however causing enhanced effort for the design phase.

The bottom-up design is largely accompanied with a close mapping between hardware components and their software models. Resulting is a strong encapsulation of knowledge that is specific to a particular hardware component. This encapsulation supports distributed development and reuse. This becomes very clear with the software representations of the system integration PLD's that are used on all coprocessors. Since the system integration PLD controls and synchronises

the functions of many devices contained in a coprocessor and provides sensible functions to the whole system, it is very specific to a particular coprocessor implementation. The functionality of a PLD is defined by its configuration, and the development of a PLD's configuration (*design*) requires significant amount of work and domain specific knowledge. Accordingly, most of the software models for the different system integration PLD's were implemented by the respective hardware programmers, reusing their domain specific knowledge. Though being mostly successful, especially considering knowledge encapsulation, this approach imposes problems regarding software quality because hardware programmers are not necessarily software programming experts.

Besides the abovementioned capture and encapsulation of hardware device specific knowledge and functions, the according software models (classes) are designed in such a way that they not only serve as user-visible service providers, but also in a way that the same classes implement the media that transparently transform and transfer service requests between the client, e.g. user application, occasionally intermediate classes, and finally the point where the request leave the software, which is usually the device-driver interface. In following this design principle, it is possible, and necessary, to chain classes together to implement the functions of the software. The collaboration diagram 3.12 shows such a chain and the transformation and mediation of a service request. This design contrasts well with designs like the one implemented by Schumacher et al [PSSL00], where two orthogonal class hierarchies are used to implement and to access functions. A principal problem arising from the identity of service-providing and service-mediating classes is that such classes depend on the declarations of other classes to which transformed services are passed on upstream the service chain. This dependency breaks encapsulation and inhibits reuse. One example of a resulting strong coupling is the `AcbClkPld` class whose definition depends on the declaration of the `AcbCtrlPld` class. However, reuse of `AcbCtrlPld` is very unlikely since `AcbCtrlPld` is unique to the ACB. On the other hand, e.g. JTAG capability is present on many devices on different coprocessors. In this case reuse is enabled by generalising software functions supporting JTAG in a base class, and specialising this base class in order to adapt to upstream, mediating classes, as it is done some ten times in the software. The results show that this approach is powerful and effective, and the code needed to subclass and adapt is very compact and straightforward (see JTAG on page 78 and figure 3.2.2). Performance issues that may be caused by chaining of services are discussed below. Also, an approach for decoupling the involved classes, which was presented in section 3.4, is discussed below.

For every coprocessor a *control* class is provided which is the top level model of its respective coprocessor and implements the coprocessor's application interface. The control class references a tree-like structure of lower level components and implements the algorithm to populate that tree. The control class is, like the system integration PLD class, coprocessor specific. These two classes reuse common components and functionality from the core software and are generally the only specialised software components that are needed to support a specific coprocessor.

Since the software allows applications to use several coprocessors simultaneously, even if they are of differing type, a bookkeeping scheme is provided. The scheme enumerates all coprocessors found in the system and provides services to identify coprocessors based on different criteria. Because the scheme has to know about every supported coprocessor type, it breaks encapsulation, e.g. the scheme has to be modified for every coprocessor-support-package that is added to the software, even if the coprocessor is very similar to one that is already supported by the software.

The results show (see section 3.2.4) that support for six different coprocessors was added successively to the software. This adding was carried out mostly seamless and increased the code-base by more than a factor of two during a time-span of about four years. I believe that there are two main reasons for the fact that the evolution of the software was so steady, did not break the initial software architecture, and took advantage of software reuse. First, the bottom-up design approach for low level software entities was creating a rich set of building blocks, which were reused, adapted, and composed in order model the added coprocessors. Second, the encapsulation of the classes that represent coprocessor-specifics, like the abovementioned PLD and control classes, enabled the decoupling of the coprocessor-supporting software in order to partition it into independent packages. These support packages make up about half of the total code-size of the software, which indicates the great amount of reuse in the software.

However, the long-lasting evolution of the software showed some deficiencies of the software. To me, the most annoying of these is the lack of good design in some parts of the software that resulted from inaccuracy and hastiness in following the spiral process. Such hastiness often turned the process into a code-and-fix process, introducing faint design into the software. According problems with the `registry` package have been discussed above. As also mentioned above, the spiral process itself shows a tendency to veil wrong design decisions.

The layered structure used to access the coprocessor implies some execution overhead compared to a flat software structure, which may be further enhanced due to overhead introduced by using C++ instead of C as implementation language. However, the very most operations that travel through the software are not in the *performance path* i.e. their performance is not critical to overall system and application performance. Also, single transactions on the PCI system bus usually last long compared to the overhead introduced by additional software layers, which has been measured to be very small. Wherever possible, DMA is used to transfer bulk data between the coprocessor and the host. Although DMA transfers reach much higher rates, compared to PIO transfers, with large data blocks, the price of an increased overhead must be paid. The actual overhead and saturated transfer rate are determined by host and coprocessor architecture rather than by software. Consequently, applications have to be careful with choosing a transfer mode depending on block size and host and coprocessor architecture.

The support of the software for reconfiguration of the coprocessors is divided into two parts. First, the software supports reconfiguration of the FPGA resources

on the coprocessors at runtime. FPGA reconfiguration is initiated by user applications and, more important, the consequences of the reconfiguration of the FPGA, i.e. changed behaviour of the FPGA, are to be handled entirely by the user applications. On the other hand, the software does support reconfiguration of the system integration PLD's, or modification of configuration EEPROM's through the tools provided. Such reconfiguration may require modifications and recompilation of the software.

Concluding, the software offers only limited support for run-time reconfiguration. I addressed this problem with an approach that allows run-time reconfiguration of the software itself (see section 3.4). While retaining a bottom-up design, capturing device or function specific knowledge in separate classes, the new approach implements a generalisation of these classes by a common base class. The interface of the base class provides access to the service functions (ports) of its concrete child classes by a naming scheme. Thus, the classes become software *components*. The collaboration between different elements is enabled at run-time by connecting communication ports of the elements pairwise using the common interface. Additionally, the components reside in shared libraries that can be loaded and unloaded at runtime (see section 3.4.7). The libraries and the contained classes are identified by a simple naming scheme, too.

Wang and Shin [WS02] present a software architecture in which the behaviour of a reusable component is determined by a *control plan* that is executed by a finite state machine (FSM) driver embedded in the component. Thus the definition of the component's behaviour is separated from the definition of the functions which the component provides, and can be reconfigured at run-time. They also provide means for structural reconfiguration at run-time. However, according to the real-time requirements of their system platform, structural reconfiguration is restricted to special system states, e.g. start-up. The structural composition is based on ports that receive external events of predefined types. Wang and Shin contrast their approach with component technologies like CORBA. They argue that such technologies heavily depend on middle-ware and their approach would need fewer resources and supports more predictable execution.

Although the target of the architecture proposed by Wang and Shin is a real-time system, whereas the execution platform of my architecture is an ordinary time-sharing operating system, both approaches try to implement systems that show high execution performance and consume few computing resources. Both approaches consider the performance overhead caused by middle-ware to be unacceptable. Wang and Shin also argue that middle-ware would make the performance unpredictable, which is true for my approach, too. Wang and Shin do not provide performance measurements whereas I have measured that the performance overhead caused by my approach is significant, but may be acceptable. Also, I assume that the performance of my architecture can be greatly enhanced by optimisations since I used a very simple naming lookup scheme based on hash tables. Wang and Shin separate behaviour from functions by using FSM drivers that interpret behaviour descriptions. My approach does not support such a separation of behaviour since

behaviour and function is fixed in the components. Configuration of behaviour in my architecture requires structural changes or modification of components, which is supported at run-time by the dynamic collaboration between components and the loading and unloading of components contained in shared libraries.

Stewart and Khosla [SVK97] propose an approach that is similar to the one followed by Wang and Shin. They too use FSM interpreters to define behaviour, yet their communication architecture is based on shared memory whereas Wang and Shin use event-based communication. In contrast, my scheme resembles a function-call-like connection architecture.

Abbreviations

API application programming interface

ASIC application specific integrated circuit, hardware device that is designed and manufactured for specific applications.

ATLAS general purpose particle detector at the LHC, CERN.

ATM asynchronous transfer mode, channel based networking technology with integrated quality of service, often used in telecommunication industries.

CERN European organisation for nuclear research, located at Geneva.

CORBA common object broker architecture, standard for component interoperability middle-ware.

COTS commodity of the shelf, computers and devices with high availability and thus low price.

CPCI compact PCI, often used in telecommunication industries.

CPLD complex programmable logic device, see PLD.

CPU central processing unit, the device that controls a computer system and executes algorithms on data.

CSRC control software for reconfigurable coprocessors, the topic of this work.

DAQ data acquisition, system for collecting and combining data from the detector.

DMA direct memory access, method to transfer data between the components of a computer system by using a specialised device (DMA controller), relieving the CPU.

FCop FPGA based coprocessor.

FPGA field programmable gate array, hardware device that can be configured to emulate arbitrary logic behaviour.

- HLT** high level trigger, complex distributed computing systems used for data reduction in high energy physics experiments.
- IA32, IA64** Intel architecture 32, Intel architecture 64, two CPU families.
- LVL2** second level trigger, HLT excluding the third level trigger / event filter.
- JTAG** joint test action group, standard interface and infrastructure for accessing complex devices besides their designated functions, e.g. for system debugging.
- LHC** large hadron collider, accelerator/collider at CERN.
- OS** operating system, basic program managing a computer.
- PCI** peripheral component interconnect, bus for connecting devices to a computer, widely adopted in COTS systems.
- PIO** programmed input-output, CPU controlled transfer of data to peripheral components of a computer system.
- PLD** programmable logic device, hardware device that can be configured to execute arbitrary logic functions.
- RAM** random access memory
- RH** reconfigurable hardware, any system or device with configurable behaviour based on arbitrary logic.
- ROM** read only memory
- RS** reconfigurable systems, systems with reconfigurable devices.
- S-Link** simple link, a CERN standard for a unidirectional high speed data link.
- T2REF** second level trigger reference software, software framework used to conduct studies for the second level trigger.
- TDAQ** trigger and data acquisition, system for collecting and selecting data from the detector.
- TRT** transition radiation tracker, particle detector, here a sub-detector of the inner detector of ATLAS.
- VME** versa module eurocard bus

Acknowledgments

My warmest thanks go to my family for their almost endless patience. I thank Denis Calvet and Irakli Mandjavidze from CEA for their help in integrating Enable++ and Atlantis into testbeds. I thank my colleagues Dr. Matthias Müller and Andreas Kugel, who both provided significant input to the software. My special thanks go to Andrei Khomich for the hours of discussion about the TRT-LUT-Hough algorithm. I thank Harald Simmler for proofreading this thesis, and for all his kind help. I thank the Bundesministerium für Forschung und Technik, that provided the funding for this thesis, and Henning Schmidt and Hewlett-Packard who donated an IA64 workstation. Special thanks go to Andrea Seeger for her help, and to Prof. Dr. Reinhard Männer, who made this thesis possible.

Bibliography

- [ALI95] ALICE Collaboration. A large ion collider experiment, technical proposal. CERN/LHCC/95-71, CERN, December 1995.
- [APP98] Maris Abolins, Beatriz Gonzáles Piñero, and Peter F. Peterson. S-Link transmission measurements. ATL-DAQ-98-131, CERN, December 1998.
- [ATL94] ATLAS Collaboration. ATLAS technical proposal for a general-purpose pp experiment at the large hadron collider at CERN. CERN/LHCC/94-43 LHCC/P2, CERN, December 1994.
- [ATL98] ATLAS Collaboration. ATLAS DAQ, EF, LVL2, and DCS. technical progress report. CERN/LHCC/98-16, CERN, June 1998.
- [ATL03] ATLAS Collaboration. ATLAS high-level trigger, data acquisition and controls. CERN/LHCC/2003-022 ATLAS TDR 016, CERN, October 2003.
- [Aus98] Matthew H. Austern. *Generic Programming and the STL*. Addison-Wesley Publishing Company Reading, Massachusetts, 1998.
- [B⁺02] R. Blair et al. The ATLAS level-2 trigger pilot project. In *IEEE Trans. Nucl. Sci.*, volume 49, pages 851–857, 2002.
- [BDK⁺98] O. Brosch, P. Dillinger, K. Kornmesser, A. Kugel, M. Sessler, H. Simmler, H. Singpiel, S. Ruehl, R. Lay, K.-H. Noffz, and L. Levinson. Microenable, a reconfigurable FPGA coprocessor. *Fourth Workshop on Electronics for LHC Experiments*, pages 402–406, 1998.
- [BDS99] John Baines, Reinier Dankers, and Sergey Sivoklov. Performance of a lvl2 trigger feature extraction algorithm for the precision tracker. ATL-DAQ-99-013, 1999.
- [BHH⁺00] O. Brosch, J. Hesser, C. Hinkelbein, K. Kornmesser, T. Kuberka, A. Kugel, R. Männer, H. Singpiel, and B. Vettermann. ATLANTIS - a hybrid FPGA/RISC based re-configurable system. In *7th Reconfigurable Architectures Workshop (RAW 2000)*, May 2000.

- [BHK⁺03] John Baines, Christian Hinkelbein, Andrei Khomich, Andreas Kugel, Reinhard Männer, and Matthias Müller. Timing measurements of some tracking algorithms and suitability of fpga's to improve the execution speed. ATL-DAQ-2003-026, CERN, September 2003.
- [BMBG02] Mihai Budiu, Mahim Mishra, Ashwin R. Bharambe, and Seth Copen Goldstein. Peer-to-peer hardware-software interfaces for reconfigurable fabrics. In *Proceedings of the 10th IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM 2002)*, 22-24 April 2002, Napa, CA, pages 57–66, 2002.
- [BMvdB97] O. Boyle, R. McLaren, and E. van der Bij. The S-Link Interface Specification. <http://hsi.web.cern.ch/HSI/s-link/spec>, March 1997.
- [Boo94] Grady Booch. *Object-Oriented Analysis and Design*. Benjamin / Cummings Publishing Company, Inc., 1994.
- [Boo99] Grady Booch. *The Unified Modeling Language User Guide*. Addison-Wesley Publishing Company Reading, Massachusetts, 1999.
- [CH02] Katherine Compton and Scott Hauck. Reconfigurable computing: A survey of systems and software. *ACM Computing Surveys*, 34(2), 6 2002.
- [Cha01] Stephen P.G. Chappell. Rapid development of reconfigurable systems. In *Proc. of the 12th Int. Workshop on Rapid System Prototyping (RSP'01)*, 2001.
- [CMS94] CMS Collaboration. The compact muon solenoid. CERN/LHCC/94-38, CERN, December 1994.
- [Com] Community. GNU. <http://www.gnu.org>.
- [Com99] K. Compton. Programming architectures for run-time reconfigurable systems, 1999.
- [CVHM03] H. Chen, B. Vettermann, J. Hesser, and R. Männer. Innovative computer architecture for real-time volume rendering. *Computer & Graphics*, 27(5):715–724, 2003.
- [DCG⁺99] P. Le Dû, D. Calvet, O. Gachelin, M. Huet, I. Mandjavidce, R. Blair, J. Dawson, J. Schlereth, M. Abolins, and Y. Ermoline. The ATLAS high level trigger ATM testbed. In *11th IEEE NPSS Real Time Conference*, pages 559–562, June 1999.
- [DFH⁺99] J.-P. Dufey, M. Frank, F. Harris, J. Harvey, B.Jost, P. Mato, and H. Mueller. The LHCb trigger and data acquisition system. In *11th IEEE NPSS Real Time Conference*, pages 49–53, June 1999.

- [DJ98] Robert P. Dick and Niraj K. Jha. CORDS: hardware-software co-synthesis of reconfigurable real-time distributed embedded systems. In *ICCAD*, pages 62–67, 1998.
- [Dou99] Bruce Powel Douglass. *Real Time UML*. Addison-Wesley Publishing Company Reading, Massachusetts, 1999.
- [DS99a] A. Daneels and W. Salter. Selection and evaluation of commercial SCADA systems for the controls of the CERN LHC experiments. In *Proc. 7th Int. Conf. on Accelerator and Large Experimental Physics Control Systems*, Trieste, Italy, Oct 1999.
- [DS99b] A. Daneels and W. Salter. What is SCADA. In *Proc. 7th Int. Conf. on Accelerator and Large Experimental Physics Control Systems*, Trieste, Italy, Oct 1999.
- [DSB03] Holger Froening David Slognat, Patrick R. Haspel and Ulrich Bruening. The ATOLL system area network (SAN). IEEE Task Force Cluster Computing Newsletter, Sep 2003.
- [EM00] M. Eisenring and M. Platzner. An implementation framework for runtime reconfigurable systems. In *Proceedings of the 2nd International Workshop on Engineering of Reconfigurable Hardware/Software Objects (ENREGLE00)*, pages 151–157, June 2000.
- [Eva95] L.R. Evans. The large hadron collider. CERN-AC-95-02-LHC, CERN, February 1995.
- [GG95] S. Guccione and M. J. Gonzalez. Classification and performance of reconfigurable architectures. In W. Moore and W. Luk, editors, *Field-Programmable Logic and Applications. 5th International Workshop on Field-Programmable Logic and Applications*, volume 975 of *Lecture Notes in Computer Science*, pages 439–448, Oxford U.K., August 1995. Springer-Verlag.
- [GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison-Wesley Publishing Company Reading, Massachusetts, 1995.
- [GJP⁺03] B. Gorini, M. Joos, J. Petersen, A. Kugel, R. Männer, M. Müller, M. Yu, B. Green, and G. Kieft. A RobIn Prototype for a PCI-Bus based Atlas Readout-System. In *9th Workshop on Electronics for LHC Experiments*, Amsterdam, Netherland, September 2003.
- [GLS99] S.A. Guccione, D. Levi, and P. Sandararajin. Jbits; a java-based interface for reconfigurable computing. In *2nd Annual Military and Aerospace Applications for Programmable Devices and Technology*, 1999.

- [GS03] Igor Gavrilenko and Sergey Sivoklov. TRTxK: a high level trigger algorithm for the trt detector. COM-DAQ-2003-044, 2003.
- [HA96] S. Hauck and A. Agarwal. Software technologies for reconfigurable systems, 1996.
- [Hau98a] S. Hauck. The future of reconfigurable systems, 1998.
- [Hau98b] Scott Hauck. The roles of fpgas in reprogrammable systems. *Proceedings of the IEEE*, 86(4):615–638, April 1998.
- [Hau00] Reiner Hauser. The ATLAS level 2 reference software. ATL-DAQ-2000-019, CERN, March 2000.
- [HGKM02] Stefan Hezel, D. M. Gavril, Andreas Kugel, and Reiner Männer. FPGA-based Template Matching using Distance Transforms. In *Proc. FCCM 2002*, pages 89–97, Napa Valley, California, 2002.
- [HKK⁺04] C. Hinkelbein, A. Khomich, A. Kugel, R. Maenner, and M. Mueller. Using an fpga coprocessor for improving execution speed of trt-lut - one of the feature extraction algorithms for atlas lvl2 trigger. In *12th ACM International Symposium on Field-Programmable Gate Arrays*, page 247, 2004.
- [HKL⁺95] H. Högl, A. Kugel, J. Ludvig, R. Männer, and R. Zoz. Enable++: A second generation fpga processor. IEEE Symposium on FPGA's for Custom Computing Machines, 1995.
- [HKM⁺99] C. Hinkelbein, A. Kugel, R. Männer, M. Müller, M. Sessler, H. Singpiel, J. Baines, and M. Smizanska. Global pattern recognition in the TRT for the ATLAS b-physics trigger. ATL-DAQ-99-012, CERN, September 1999.
- [HKM⁺00] C. Hinkelbein, A. Kugel, R. Männer, M. Müller, M. Sessler, H. Simmler, and H. Singpiel. LVL2 full TRT scan feature extraction algorithm for b physics performed on the hybrid FPGA/CPU processor system ATLANTIS: Measurement results. ATL-DAQ-2000-012, CERN, March 2000.
- [HKT03] Christian Haubelt, Dirk Koch, and Jürgen Teich. Basic OS support for distributed reconfigurable hardware. In *Proceedings of the Third International Workshop on Systems, Architectures, Modeling, and Simulation*, pages 18–22, Samos, Greece, July 2003.
- [Hou58] Paul V. C. Hough. Machine analysis of bubble chamber pictures. In *International Conference on High Energy Accelerators and Instrumentation*, pages 554–556. CERN, 1958.

- [HPK03] Jérôme Hugues, Laurent Pautet, and Fabrice Kordon. Contributions to middleware architectures to prototype distribution infrastructures. In *Proc. of the 14th Int. Workshop on Rapid System Prototyping (RSP'03)*, 2003.
- [IEE] IEEE. *IEEE Standard Test Access Port and Boundary Scan Architecture*. IEEE. <http://standards.ieee.org/reading/ieee/std/testtech/1149.1-1990.pdf>.
- [Inta] Intel. *IA-32 Intel® Architecture Software Developer's Manual*. <ftp://download.intel.com/design/Pentium4/manuals/25366{5,6,7,8}13.pdf>.
- [Intb] Intel. *Intel® Itanium® Architecture Software Developer's Manual*. <ftp://download.intel.com/design/Itanium/manuals/24531{7,8,9}.pdf>.
- [Iwa] Iwanski. *E-SLink Users Manual*. University of Krakow. <http://www.ifj.edu.pl/~iwanski/e-slink/uman.html>.
- [Jun] Jungo technology windriver. <http://www.jungo.com>.
- [K⁺98] Klaus Kornmesser et al. Simulating FPGA-coprocessors using the FPGA development system CHDL. *Proc. PACT Workshop on Reconfg. Comp.*, pages 78–82, 1998.
- [KHM⁺00] A. Kugel, Ch. Hinkelbein, R. Männer, M. Müller, and H. Singpiel. ATLANTIS - a modular, hybrid FPGA/CPU processor for the ATLAS readout system. In *6th Workshop on Electronics for LHC Experiments*, pages 429–433, October 2000.
- [Kin95] R.M. Kinmond. Survey into the acceptance of prototyping in software development. In *Proc. of the 6th Int. Workshop on Rapid System Prototyping (RSP'95)*, 1995.
- [KKL⁺98a] A. Kugel, K. Kornmesser, R. Lay, J. Ludvig, R. Männer, K-H. Noffz, S. Rühl, M. Sessler, H. Simmler, H. Singpiel, V. Dörsing, W. Erhard, P. Kammel, A. Reinsch, and T. Schober. ATLAS level-2 trigger demonstrator-a activity report part 2: Demonstrator results. ATLAS Internal Note DAQ-NO-084, CERN, March 1998.
- [KKL⁺98b] A. Kugel, K. Kornmesser, R. Lay, R. Männer, K-H. Noffz, S. Rühl, M. Sessler, H. Simmler, H. Singpiel, V. Dörsing, W. Erhard, P. Kammel, and A. Reinsch. ATLAS level-2 trigger demonstrator-a activity report part 3: Paper model. ATLAS Internal Note DAQ-NO-101, CERN, June 1998.
- [KKL⁺98c] A. Kugel, K. Kornmesser, R. Lay, R. Männer, K-H. Noffz, S. Rühl, M. Sessler, H. Simmler, H. Singpiel, V. Dörsing, W. Erhard, P. Kammel, A. Reinsch, L. Levinson, R. Bock, W. Iwanski, K. Korcyl, J. Olszowska, D. Calvet, J. R. Hubbard, P. Le Dû, I. Mandjavidze, and

- M. Smizanska. ATLAS level-2 trigger demonstrator-a activity report part 1: Overview and summary. ATLAS Internal Note DAQ-NO-085, CERN, March 1998.
- [KKM⁺99] T. Kuberka, Andreas Kugel, Reinhard Männer, Holger Singpiel, R. Spurzem, and R. Klessen. AHA-GRAPE: Adaptive hydrodynamic architecture - GRAvity PipE. In *Proc. Int. Conf. on Parallel and Distributed Processing Techniques and Applications*, pages 1189–1195, 1999.
- [Kug02] Andreas Kugel. Race-1. <http://www-li5.ti.uni-mannheim.de/fpga/>, 2002.
- [L⁺01] P. Leong et al. Pilchard - a reconfigurable computing platform with memory slot interface. In *IEEE Symposium on Field-Programmable Custom Computing Machines, FCCM*, 2001.
- [LHC95] LHC Study Group. The large hadron collider: conceptual design. CERN-AC-95-05-LHC, CERN, October 1995.
- [LHC98] LHCb Collaboration. LHCb technical proposal. CERN/LHCC/98-4, CERN, 1998.
- [Lie03] G. Lienhart. Implementing Hydrodynamic N-Body Codes on Reconfigurable Computing Platforms. In *International Conference on High Performance Scientific Computing*, Hanoi, Vietnam, March 2003.
- [LKM02] Gerhard Lienhart, Andreas Kugel, and Reinhard Männer. Using floating-point arithmetic on FPGAs to accelerate scientific n-body simulations. In *Proc. IEEE Symposium FCCM*, 2002.
- [Lud98] Jozsef Ludvig. *Enable++: Ein universeller FPGA-Triggerprozessor für das ATLAS-Experiment*. PhD thesis, Lehrstuhl für Informatik V, Universität Mannheim, 1998.
- [McC96] Steve McConnell. *Rapid Development*. Microsoft Press, Redmond, Washington, 1996.
- [ME02] David Mosberger and Stéphane Eranian. *ia-64 linux kernel*. Prentice Hall, New Jersey, 2002.
- [Mey96] Scott Meyers. *More effective C++*. Addison-Wesley Publishing Company Reading, Massachusetts, 1996.
- [Mey97] Scott Meyers. *Effective C++*. Addison-Wesley Publishing Company Reading, Massachusetts, 1997.

- [Mye99] D. R. Myers. The LHC experiments' joint controls project, JCOP. In *Proc. 7th Int. Conf. on Accelerator and Large Experimental Physics Control Systems*, Trieste, Italy, Oct 1999.
- [Nev03] Pavel Nevski. Heavy ion collisions with the ATLAS detector. *ATL-COM-CONF-2003-027*, 2003. ATLAS internal document.
- [Nof96] Klaus-Henning Noffz. *Ein FPGA-Prozessor als 2nd-Level-Trigger für ATLAS*. PhD thesis, Lehrstuhl für Informatik V, Universität Mannheim, 1996.
- [NZK⁺95] K.-H. Noffz, R. Zoz, A. Kugel, F. Klefenz, and R. Männer. Results of on-line tests of the ENABLE prototype, a 2nd level trigger processor for the TRT of ATLAS/LHC. Reihe Informatik 17/95, Universität Mannheim, 1995.
- [OMG] OMG. *CORBA*. <http://www.omg.org/>.
- [PCI] PCISIG. *PCI*. <http://www.pcisig.com/home>.
- [PLXa] PLX Technology. *PCI 9080 Data Book*. PLX Technology. <http://www.plxtech.com>.
- [PLXb] PLX Technology. *PCI 9656 Data Book*. PLX Technology. <http://www.plxtech.com>.
- [PLXc] PLX Technology. *PLX SDK Reference*. PLX Technology. <http://www.plxtech.com>.
- [PSSL00] U. Pfeiffer, V. Schatz, C. Schumacher, and M.P.J. Landon. HDMC: An object-oriented approach to hardware diagnostics. In *6th Workshop on Electronics for LHC Experiments*, pages 464–468, October 2000.
- [Qua99] Terry Quatrany. *Visual Modelling with Rational Rose 2000 and UML*. Addison-Wesley Publishing Company Reading, Massachusetts, 1999.
- [R⁺93] R.K.Bock et al. Status report: Embedded architectures for second-level triggering (east). CERN/DRDC 93-12, RD11 Status Report (EAST note 93-08), CERN, May 1993.
- [Rüh01] Stephan Rühl. *Programmierung von FPGA-Prozessoren mittels aktiver Komponenten*. PhD thesis, Lehrstuhl für Informatik V, Universität Mannheim, 2001.
- [S⁺03] S.Armstrong et al. An overview of algorithms for the ATLAS high level trigger. In *IEEE Real Time 2003*, 2003.

- [SA99] Tom Shanley and Don Anderson. *PCI System Architecture*. Addison Wesley, 1999.
- [SD02] Alexander Staller and Peter Dillinger. Implementation of the jpeg 2000 standard on a virtex 1000 fpga. In *Proc. 12th Int. Conf. on Field Programmable Logic and Applications*. Springer-Verlag, Berlin, Heidelberg, New York, 2002.
- [Ses96] Matthias Sessler. Entwicklung des i/o-boards für den second-level-triggerprozessor enable++. Master's thesis, University of Heidelberg, 1996.
- [Ses00] Matthias Sessler. *Algorithms on CPUs and FPGAs for the ATLAS LVL2 Trigger*. PhD thesis, Lehrstuhl für Informatik V, Universität Mannheim, 2000.
- [Sim01] Harald Simmler. *Preemptive Multitasking auf FPGA-Prozessoren*. PhD thesis, Lehrstuhl für Informatik V, Universität Mannheim, 2001.
- [Sin00] Holger Singpiel. *Der ATLAS LVL2-Trigger mit FPGA-Prozessoren*. PhD thesis, Lehrstuhl für Informatik V, Universität Mannheim, 2000.
- [SK96] David B. Stewart and Pradeep K. Khosla. The chimera methodology: Designing dynamically reconfigurable and reusable real-time software using port-based objects. *International Journal of Software Engineering and Knowledge Engineering*, 6(2):249–277, June 1996.
- [Sof] Silicon Software. microEnable. <http://www.silicon-software.com>.
- [Str00] Bjarne Stroustrup. *The C++ Programming Language and Reference*. Addison-Wesley Publishing Company Reading, Massachusetts, third edition, 2000.
- [SvG00] Jochen Seemann and Jürgen Wolff von Gudenberg. *Software-Entwurf mit UML*. Springer-Verlag, Berlin, Heidelberg, New York, 2000.
- [SVK97] David B. Stewart, Richard A. Volpe, and Pradeep K. Khosla. Design of dynamically reconfigurable real-time software using port-based objects. *Software Engineering*, 23(12):759–776, 1997.
- [Sys] SystemC. *SystemC*. <http://www.systemc.org/>.
- [Szy98] Clemens Szyperski. Emerging component software technologies - a strategic comparison. *Software - Concepts & Tools*, 19(1):2–10, 1998.
- [Tak03] H. Takai. Heavy ion collisions with the ATLAS detector. SN-ATLAS-2003-035 CERN, 2003.

- [TC04] Linus Torvalds and Community. Linux. <http://www.kernel.org/pub/linux>, 1991-2004.
- [TL02] David. B. Thomas and Wayne Luk. Framework for development and distribution of hardware application. In *Reconfigurable Technology: FPGAs and Reconfigurable Processors for Computing and Communications IV, Proceedings of the SPIE*, volume 4867, 2002.
- [TOT99] TOTEM Collaboration. Total cross section, elastic scattering and diffraction at the LHC. CERN/LHCC/99-7 LHCC/P5, CERN, March 1999.
- [vdBH] Erik van der Bij and Stefan Haas. S-SLink general information. <http://hsi.web.cern.ch/HSI/s-link/>.
- [Ver00] J. Vermeulen. A SHARC based ROB Complex : design and measurement results. *ATL-DAQ-2000-021*, March 2000.
- [WP04] Herbert Walder and Marco Platzner. Reconfigurable hardware os prototype. <http://e-collection.ethbib.ethz.ch/show?type=incoll&nr=990>, 2004.
- [WS02] Shige Wang and Kang G. Shin. Constructing reconfigurable software for machine control systems. *Transactions on Robotics and Automation*, 18(4):475–486, 2002.
- [YBGL02] Paul Young, Valdis Berzins, Jun Ge, and Luqi. Using an object oriented model for resolving representational differences between heterogeneous systems. In *Proceedings of the 2002 ACM Symposium on Applied Computing (SAC), March 10-14, 2002, Madrid, Spain*, pages 976–983. ACM, 2002.
- [YCBL03] Paul Young, Nabendu Chaki, Valdis Berzins, and Luqi. Evaluation of middleware architectures in achieving system interoperability. In *Proc. of the 14th Int. Workshop on Rapid System Prototyping (RSP'03)*, 2003.