Describing and Simulating Dynamic Reconfiguration in SystemC Exemplified by a Dedicated 3D Collision Detection Hardware

Dissertation
zur
Erlangung des Doktorgrades (Dr. rer. nat.)
der
Mathematisch-Naturwissenschaftlichen Fakultät
der
Rheinischen Friedrich-Wilhelms-Universität Bonn

vorgelegt von
Diplom Informatiker Andreas Raabe
aus Niederkassel

Bonn, April 18, 2008

Reviewer: Joachim K. Anlauf, Reinhard Klein
Date of oral exam: August 10, 2008
Published: 2008

This work was typeset with KOMA-Script and LaTeX.

For my parents, who provided me such a fine start in life.

And for Silke, who loves me how I am.

Contents

Acknowledgment

During the work on this dissertation a lot of people crossed my path, students as well as colleagues. Every single one of them is special in his ways and I want to thank all of them for the time we spent together, no matter if it was for educational purposes, plain work or fun.

During my work on both projects presented in this dissertation a lot of people helped bringing them to success and made it so much fun to work on it. Thus, I want to especially thank my co-authors of the publications concerning ReChannel and Collision-Chip [58–63] A. Felke, S. Hochgürtel, G. Zachmann, P.A. Hartmann, F. Zavelberg, J.K. Anlauf and B. Bartyzel. Also special thanks go to the students I supervised and who's diploma theses [48, 49, 73] did not (yet) result in publications A. Niers, K. von der Heyde, A. Nett, J. Tietjen, and those who participated in project-related work B. Bales, T. Becker, T. Loraing, M. Nolden, R. Reifenhäuser, U. Schuster, R. Thesen, and J. Wolf.

I also want to thank Prof. Dr. G. Zachmann for his support concerning CG specific problems and Prof. Dr. R. Klein for volunteering to be a referee on this work.

Last but not least I thank Prof. Dr. J.K. Anlauf for supervising this work and enabling it in the first place. I could always count on his opinion and support.

Part I

Introduction

1 Motivation

Due to the on-going micro-miniaturisation in chip production, hardware development has changed within the last decade. It evolved from plain circuit design into the development of complex heterogeneous systems with an increasing number of increasingly complex processing elements [68], that even contain embedded multicore processors. This on-going trend results in new challenges to the design community [41]. Not only that the productivity of hardware designers does not grow as fast as the number of available transistors per chip (productivity gap, see Figure 1.1), but the time to bring products to market reduces as well. This is due to the fact, that in important branches of the hardware market (e.g., mobile telecommunication) the rate of innovation has increased to a point where only the first product to the market makes profit. Additionally debugging and verifying has become increasingly complex due to the growing system complexity [10]. All this led to several new trends in design to cope with these and several other challenges in the field [76].

To fill the gap one obvious approach is reuse of own and externally produced components. Externally purchased components are usually provided closed source and they remain intellectual property of the vendor, hence they are usually referred to as IP-cores. This trend has led to a still growing market of IP-vendors with a huge range of products. IP reuse is widely regarded as one of the major motors of productivity in the contemporary chipdesign market.

A major obstacle on the way to a short time-to-market is the need to verify a chip's functionality before it is shipped. Different to software it is nearly impossible to update an erroneous hardware and a callback of a shipped chip is highly cost intensive and a major loss of prestige[1]. Hence nowadays a lot of effort is spend in verification, co-verification, cross-validation, testing, etc[2]... Still, the dominating way of early debugging is simulation based testing [81].

To cope with complexity higher levels of abstraction were introduced in system design and simulation. New languages were introduced that allow system level exploration of the design space to determine which parts of a system can be implemented in software and which ones need to be produced in hardware.

They also allow early estimation of time and hardware consumption. Especially the introduction of transaction level design to evaluate the speed impact of bus models proved to be highly efficient and productive in practise [54].

Concurrent development of hardware and the software to be running on or interacting with it decreases time-to-market effectively. This technique is called Hardware/Software CoDesign and allows simulation of hardware descriptions within the same development

[1] Who does not still remember the pentium bug?

[2] And this list does not even cover variants taking timing constraints into account.

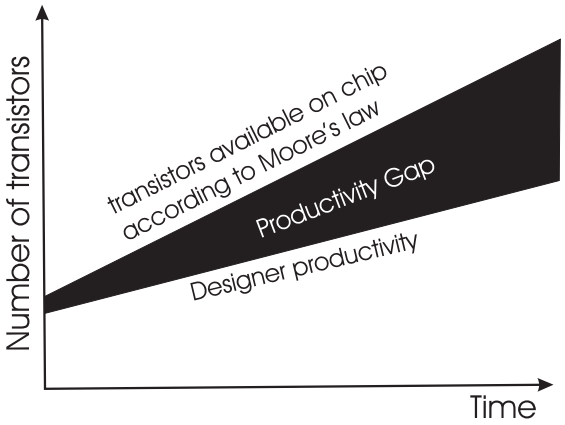


Figure 1.1: System designer's productivity grows slower than the number of available transistors. A gap in productivity results.

environment as the software code is running in. A survey of codesign methodologies and approaches can be found in [12].

As a consequence a large number of system description languages has been proposed since the late 1980s. Languages based on C/C++ are the dominating subspecies here. A brief overview of C-based languages and a discussion on their synthesisability can be found in [24]. Among these languages SystemC (see Sec. 5.1.1) became the most prominent one.

Configurable logic devices evolved considerably as well. State-of-the-art devices provide tremendous processing power and dynamic reconfiguration abilities[3]. This increased power even qualifies them to be used as cost effective alternatives in small amount production of low- and middle-end circuitry. Or seen from another point of view availability of integrated prototyping solutions with tremendous processing powers also opens possibilities in using them as highly parallel coprocessing units[4]. In fact, this is an ongoing trend in supercomputing [20, 45].

Moreover, the reconfiguration abilities are beneficial in different contexts other than prototyping. With configurable hardware it is even possible to provide (firmware) updates of the hardware description in use. Nearly the full flexibility of contemporary software update mechanism can be provided for hardware this way as well. It is only a question of effort the designer is willing to spend. Here increasing reconfiguration abilities were provided by the vendors to qualify their products for a new market. This led to dynamically reconfigurable devices as a side effect .

Still, in some branches of the market there is even more hunger for bigger chips to be

[3] As a rule of the thumb one can assume that high-end prototyping platforms are ten times slower than state-of-the-art ASIC technology. This is remarkable since they are implemented in ASIC technology and the configuration overhead is immense, taking into account that only rewiring takes large crossbar-switches on all wire crossings and that look-up-tables are nothing else but memory cells.

[4] Actually configurable hardware can be interpreted as some kind of very very very large instruction word processor.

filled with more functionality [68]. Partially this is due to the trend to pack complete chipsets onto a single platform resulting in systems-on-chip (SOC) and even complete networks-on-chip (NOC). The latter denote multiple systems-on-chip being interconnected via (usually TCP / IP based) networks.

In addition to increased reconfiguration abilities of certain platforms this led to increased research in run-time and dynamic reconfigurable designs and systems. As discussed in [70] they are now close to their commercial breakthrough.

2 Project Objectives

The market observations discussed in Sec. 1 immediately lead to the conclusion, that high-level design methodologies for development of reconfigurable hardware are vital in order to allow commercially relevant production of reconfigurable architectures. As will be discussed in Sec. 5.1.1 in depth, SystemC, the number one high-level design language, does not support modelling of dynamic reconfiguration natively. This is due to the fact, that changes to neither the module hierarchy nor to the interconnection properties of a system are allowed after the elaboration phase (see Sec. 5.1.1). As already discussed in the previous section IP reuse also is inevitable in recent hardware development.

Hence this work's major objective is the development of a language extension to SystemC that enables description and simulation of reconfigurable hardware on all levels of abstraction featured by the SystemC framework, while enabling IP reuse.

The subsequent sections principally consider which properties a simulation reconfiguration library for SystemC should have in detail. Objectives for this thesis are then deduced and formulated.

2.1 Standard Compliance Objective

Since there are multiple SystemC simulators and cosimulators on the market [14, 28, 50] it would be a major limitation to provide a solution that works with only one of them. Even concentrating on the OSCI's open source reference implementation [50] would lead to the need of permanent adaption of the library since SystemC is still under heavy development and newer versions tend to be internally quite different to older ones.

Hence modelling and simulation of reconfigurable systems should not require usage of a specially crafted or manipulated SystemC simulation kernel. There are many commercially available tools (e.g., for co-simulation), that include their own SystemC runtime implementation. Excluding their use, due to the reliance on a non-standard simulation kernel, would be an unwanted limitation for the designer.

Comply to the SystemC language standard.

2.2 IP Objective

As discussed in Sec. 1 component and IP reuse is a vital motor to the chip market, effectively bridging the productivity gap and reducing time-to-market. Hence it is unlikely that any approach requiring changes to modules to render them reconfigurable will be accepted by developers and/or the EDA community. Even changes to existing

(static) open source components would probably result in increased development costs. Since reconfiguration is neither necessarily initiated nor controlled by the reconfigurable components themselves, these changes are objectionable.

Component and even third-party-IP reuse should be supported.

2.3 Abstraction Objective

It is highly desirable to take possible (dynamic) reconfigurability schemes already at early stages of a system's design into account. Since using dynamic reconfiguration within a design can have high impact on chip utilisation and system performance, the possibility to study a system's behaviour in a reconfigurable context should be provided on all levels of abstraction within the SystemC design flow. Additionally, refinement of different modules (static *and* reconfigurable one's) should be possible independently of the inclusion and refinement of the system's reconfiguration properties (like scheduling techniques, development of a controller etc.). This enables independent and even concurrent refinement of a module's functionality and its reconfiguration behaviour. As a result, a language extension to SystemC, that allows modelling, simulation and refinement of a dynamically reconfigurable system, needs to support any, even custom-built channels natively. To integrate reconfiguration into a SystemC design *without* changes to the simulation kernel in use (see objective 2.1), the control of the reconfiguration process should be as flexible as possible. The designer should be free to model the reconfiguration controller as a module or even as a channel itself, so no limitation regarding the system's refinement is imposed.

Integrating reconfiguration into a SystemC design should be possible on all levels of abstraction.

2.4 Integration Objective

The hardware designer community is somewhat old fashioned. Introducing new languages or methods into existing company structures and production cycles is known to be very difficult. Hence every extension of language features should melt as seamlessly as possible with the existing infrastructure. Thus it is advisable to make a reconfiguration simulation library look as SystemC-like as possible to make it appear as "natural" to the designer as possible.

Seamless integration into SystemC language.

2.5 Synthesis Objective

Most systems are still refined manually by a designer in order to provide maximum utilisation of available resources. Even if automated synthesis of reconfigurable systems

might lead to a shorter path to hardware, such an approach unavoidably imposes design limitations. Hence it should be possible to refine and synthesise the system using standard techniques and tools independently of the reconfiguration properties.

"Handcrafted" refinement to hardware implementation should be possible using standard tools and techniques.

2.6 Connectivity Objective

As discussed in Sec. 5.1.2, realisation of reconfiguration abilities in different hardware platforms implies that there is more to reconfiguration than just exchange of functionality. It should at least in principle be possible to explicitly model connectivity changes separately from module functionality changes.

Allow connectivity changes separately from module functionality changes.

2.7 Case-Study Objective

Usually new methodologies are tested with toy problems to proof their effectiveness. Real world problems tend to impose higher demands on applicability of a methodology. Additionally, only well tested work flows have a chance to be incorporated into companies' productive every day work.

Prove applicability in real world case study.

3 Choice of Case Study

Sec. 2.7 demands application of the devised methodology in a real world case study. This implies application to a new area of usage, since "reinventing the wheel" is usually not done if avoidable.

On the other hand the problem itself should be "real world", meaning that application to some far out field is prohibitively far from reality. Therefore a computer graphic (CG) application was chosen. CG is an area of application that has a long history of developing special hardware for its purposes. This is probably due to the tremendous need of processing power in gaming, physical-based simulation, scientific visualisation and many other CG applications.

Physically-based simulation is more and more becoming a fundamental task in today's computational applications (e.g., gaming, virtual reality, augmented reality, etc.). As reported in [56] 95% of calculation time of physically-based simulation is spend on collision detection. This makes it a major bottleneck. Hence having dedicated hardware support is highly desirable, if not even indispensable. Still, collision detection algorithms are under heavy research and it is not yet clear what the final solution will be. It even is very likely that there will not be a single algorithm that will be used for all possible cases, but multiple different approaches coexisting. Especially the choice of primitives used for modelling of 3D environments is unlikely to be ultimately decidable. Hence, a collision detection hardware needs to remain flexible, in a way that different primitives can be used to model the underlying geometry. Even concurrent use of objects represented using different primitives is imaginable. This directly leads to the use of reconfigurable hardware that allows exchange of certain parts of the architecture to enable exchange of primitive intersection test architectures.

Within the field, hardware accelerated collision detection is a quite recent development.

The first special-hardware [1] for physically-based simulation in gaming-platforms was introduced to the market only recently. (The architecture presented in this dissertation had already been published.) Little detail was published and hence its inner structure is unknown to the author of this thesis and to the public. But it can be taken as proof on how commercially relevant and new research in this field is.

Additionally, a major reason for investigations on hardware acceleration for hierarchical collision detection for rigid bodies using k-DOPs (see Sec. 7.3) was the fact that good proof exists of its applicability and performance [86, 87].

4 Organisation

4.1 Document Structure

As will be discussed in Sec. 5.2 current description and simulation environments of the state-of-the art high-level modelling language SystemC do not allow development of a flexible architecture as the one outlined above. An according extension library had to be developed and is presented in this work, along with a collision detection architecture providing the required flexibility. Hence, ReChannel serves as a tool in the development of CollisionChip, while CollisionChip provides a real world case study. Therefore, this document is divided into four parts:

- *Introduction* (Part I),
- *Describing and Simulating Dynamic Reconfiguration in SystemC* (Part II),
- *A Dedicated 3D Collision Detection FPGA Architecture* (Part III) and
- *Putting It All Together* (Part IV)

Parts II and III are both divided into two sections: Related work and own contribution. In Part IV the ReChannel methodology is applied to the CollisionChip project. It also provides a concluding summary of this work and proposes further areas of research.

4.2 Specific Terms Used

Since not only terms connected to reconfiguration but also all kinds of hardware terms are often used differently in different contexts and abbreviations keep on changing meaning, it often comes to Babylonian confusion when discussing the subject. To avoid this, some of theses terms need to be defined and will be used exclusively as introduced here in the following.

- *Hardware Description* The way a hardware design is specified.
- *Hardware Modeling* The art of describing a hardware implementing a certain functionality.
- *Hardware Design / Model* A specific hardware description resulting from a modelling process.
- RTL(M) Register-Transfer-Level(-Model)
- BL(M) Behavioural-Level(-Model)
- TL(M) Transaction-Level(-Model)
- UTF(M) Untimed Functional-Level(-Model)
- TFL(M) Timed-Functional-Level(-Model)
- SL(M) System-Level(-Model)
- *Design / Model* Implementation of functionality on some (possibly heterogeneous) platform

- *Configurable hardware (CHW)* A most general term meaning every kind of hardware platform that does not require the functionality to be implemented to be completely fixed when it is shipped by the vendor.
- *Reconfigurable Hardware (RHW)* Every kind of hardware platform that does not only allow a single configuration, but to change functionality at least one time.
- Configuration Initial programming of functionality into a (re-)configurable hardware.
- *Reconfiguration* The process of changing the function executed/implemented by a reconfigurable hardware.
- *Run-time reconfiguration* Process of changing the function executed/implemented by a reconfigurable hardware without halting the overall design (which may be implemtented at least partly on a CPU or some other hardware different to the one being reconfigured).
- *Partial reconfiguration* Process of changing only parts of the design running on the RHW.
- *Dynamic reconfiguration (DR)* Partial run-time reconfiguration where parts of the hardware design keep on executing on the hardware being reconfigured during reconfiguration.
- *Dynamically reconfigurable hardware (DRHW)* Platform featuring partial run-time reconfiguration.
- *(Dynamically reconfigurable) hardware environment (DRHE)* Platform featuring partial run-time reconfiguration including its hardware environment (e.g., a prototyping carrier board with JDEG port).
- *Reset on (re-)configuration (ROC)* Variable initialisation after (re-)configuration and before start of execution.

Part II

Describing and Simulating Dynamic Reconfiguration in SystemC

5 Related Work

This section will give a brief overview of some basic related work in the field. A short SystemC recap is given, concentrating on details of the simulation engine, which mainly influence design and implementation of the work presented later on. Afterwards, the most interesting, commercially available reconfigurable hardware devices will be investigated.

Sec. 5.2 gives an overview and analyses of currently available approaches towards SystemC based reconfiguration simulation and synthesis. Varying methodologies will be recapitulated and it will be discussed how far they cover the objectives deduced in this work (see Sec. 2).

5.1 Basics

5.1.1 SystemC

SystemC [34, 50, 51] is an open source C++ library that allows component based modelling of hardware and software within the C++ framework. SystemC extends C++ with hardware modelling capabilities that enable the designer to write synthesisable RTL descriptions, transaction level models, functional hardware specifications and (parallel) software and co-simulate all of them. This allows an early design-space exploration to avoid implementation overhead at lower levels of abstraction. For an introduction to SystemC see [18, 31], a good quick reference is provided by [21]. For further language details refer to the language standard [34]. Advanced SystemC modelling techniques can be found in [47].

In the following a brief recap of SystemC's abstraction levels and the according design methodology is given. Afterwards, the SystemC simulation semantic is summarised.

A Brief SystemC Methodology Recap

The SystemC CoDesign methodology can be best visualised by an inverted Y-chart (see Figure 5.1). The more abstract the model the faster it can be simulated and the less effort it takes to be implemented. On the other hand does it take concrete models to receive high performance synthesis results[1].

A typical SystemC project starts with a specification of the system's functionality. This functionality is divided into concurrently running modules with point-to-point communication using FIFOs with blocking read/write access. This model of execution

[1] Some people would not even call implementations synthesised from high level models with today's compilers a *result*.

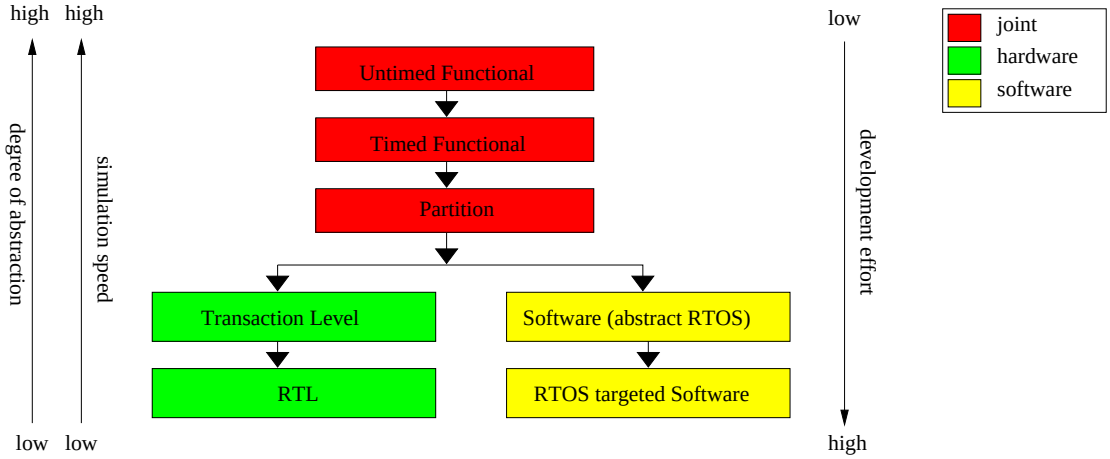


Figure 5.1: The SystemC design methodology as inverted Y-chart. The more concrete the description, the higher the development effort necessary. Additionally, more abstract models simulate faster.

is commonly known as a Kahn Process Network model [37] and eases restructuring of specifications or (e.g., imperative or object-oriented) algorithms into components and encapsulating their communication into channel (i.e., FIFO) accesses. This is called an Untimed Functional Model (UTFM) (see Figure 5.2) and is a first step towards parallelisation. Trying different divisions into modules accounts to design space exploration on the ”structural axis”.

Then a notion of time is added and a Timed Functional Model (TFM) results (see Figure 5.3). Here different execution times for (sub-) modules can be assumed and their impact on the overall performance can be evaluated. This is an easy and straight forward way to determine which modules need to be implemented in hardware to meet certain timing or even real-time constraints and which the designer can afford to run in software. This can be interpreted as design space exploration on the ”execution model axis” and results in a partition of the design into hardware and software parts. Of course more than these two classes can be distinguished, e.g., implementation in differently fast hardware (e.g., ASIC, FPGA,...) or software execution on different CPUs (e.g., on a host PC and an embedded CPU). A clever designer will even be able to distinguish different variants of a special FPGA or ASIC series.

Here joint view of hardware and software ends. Since this work is focused on hardware just a quick word on the software branch of the chart. Software supposed to be running on a PC can simply be (cross-) compiled to the according platform. Program parts meant to be running on an embedded CPU or specialised processor must be transcribed into the according machine language or (increasingly often) into ANSI-C to be compiled by special compilers. Then it can be run within an instruction set simulator (ISS) and thus get cosimulated with the hardware branch. If such an ISS is not provided by the CPU vendor it can be implemented within the SystemC framework. There even exist

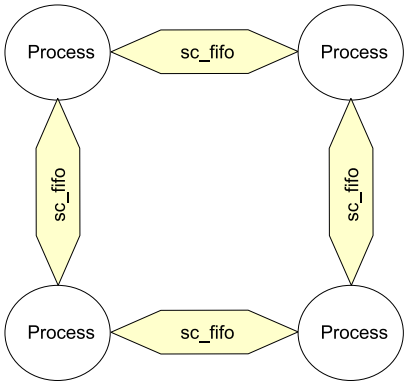


Figure 5.2: Example of an untimed functional model (Kahn process network).

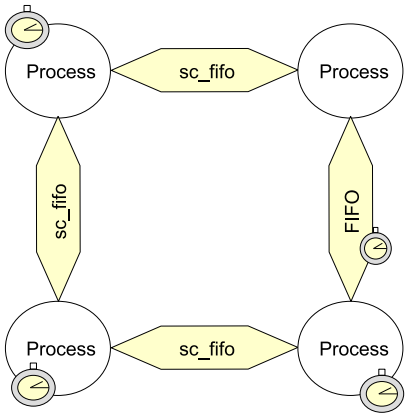


Figure 5.3: Example of a TF model.

special extensions [6, 7] that allow automated generation of assemblers for SystemC processor models. Furthermore special properties of the (real-time) operating system (RTOS) can be taken into account (if applies). Significant effort is currently spend within the SystemC community and the Open SystemC Initiative (OSCI) to improve SystemC's abstract RTOS and its RTOS simulation capabilities.

On the hardware branch it is necessary to investigate in the communications impact on run-time and the necessary inter-connectivity. This is done by refining the abstract FIFO point-to-point communication of the Kahn process network into blocking accesses (so called transactions) to random (mostly custom-built) channels. The main benefit is gained herein by introducing buses into the design. Impact of concurrent bus accesses and used scheduling techniques can be quantified and so a choice of communication models and paths between modules that meet the project constraints can be determined (communication space exploration). It is widely agreed upon that incorporating Transaction Level Modelling (see Figure 5.4) into the design flow is highly beneficial for time-to-market and performance-cost-efficiency of the devised hardware. Hence it became one of the main applications of SystemC. A TLM library is provided by the OSCI to standardise TLM channel interfaces.

To enable SystemC synthesis OSCI defined a synthesisable SystemC subset [66, 67], which is quite similar to RTL descriptions in HDLs like VHDL and Verilog. Analogously RTL descriptions in SystemC need to be cycle- and pinaccurate (see Figure 5.5). Due to the trend towards high level synthesis the property "pinaccurate" becomes more and more "floppy". A port of a fixed point type of compile time resolvable length of pre- and post-point part can with some right claim to be pin accurate and is even accepted by some tools [15]. Even if this was not covered by the term originally.

A SystemC introduction from beginner stage to a level of detail making it useful in the context of this work, would exceed any sensible extent. It can be found in the references named in the beginning of this section. Hence only a short summary of

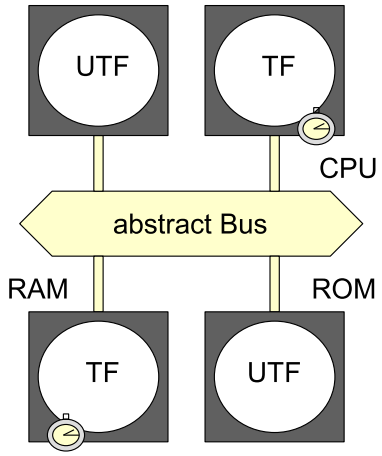


Figure 5.4: Example of a TL model.

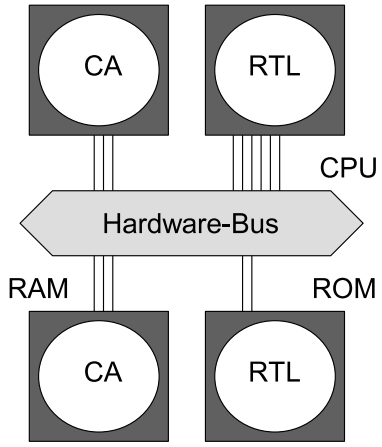


Figure 5.5: Example of a cycle- and pinaccurate RTL model.

the elaboration stage and the simulation semantics is given in the following, providing advanced knowledge of SystemC's insides.

SystemC Simulation Semantics

A formal definition of SystemC's simulation semantic was given in [46], but is already out-dated. So this section basically is a summary of the according chapter of the IEEE standard [34] extended by some figures and additional explanations.

Simulating a hardware design described in SystemC is done in two phases: *Elaboration* and *simulation* itself. During elaboration the module hierarchy is created according to the description of the presented design. When the simulation begins the event-driven scheduler is started. It takes care of process execution in accordance with the simulated design.

Elaboration SystemC primitives describing hierarchical properties of the design, such as `sc_module`, `sc_port`, `sc_export`, `sc_prim_channel`, etc. may only be instantiated during elaboration. Though not stated explicitly in the standard definition, this enforces execution of module constructors, since all the mentioned constructs can be instantiated from there. The callback function `before_end_of_elaboration` is called after the actual elaboration to allow hierarchy manipulations that depend on the hierarchy constructed so far. At this time not all bindings (see below) were processed. Callback function `end_of_elaboration` is called after all calls to `before_end_of_elaboration` are processed. The hierarchy is now complete and `end_of_elaboration` is not allowed to manipulate the hierarchy any more.

The OSCI reference implementation itself implements the elaboration by executing the `sc_main` (or `sc_main_main`) method that contains all top-level module instantiations. Hence, their constructors are called, instantiating their substructures,

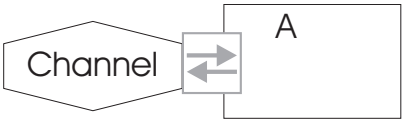


Figure 5.6: Channels within a module's environmental scope can be accessed via `sc_ports` from within the module. Therefore the port needs to be bound to the channel.

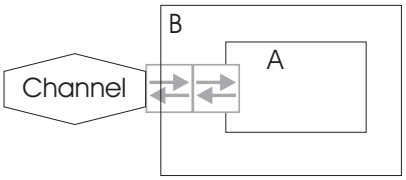


Figure 5.7: Binding a port of a submodule to a port bound to a channel allows access to the channel from within the submodule.

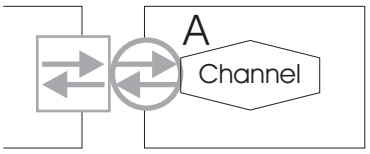


Figure 5.8: Channels inside a module can be made available to the module's environment by exporting their interface via an `sc_export`.

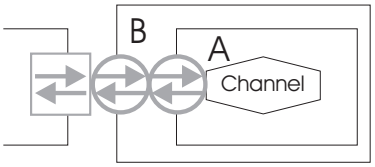


Figure 5.9: Binding the export of a father module to an export, accounts to exporting the channel interface to the father module's environment.

etc. When `sc_start` is called, the kernel completes all bindings and calls `before_end_of_elaboration` of all registered structures. `end_of_elaboration` is called afterwards. This way the designer is enabled to spread construction of the hierarchy over two separate stages of the elaboration phase.

"NOTE 1: Because these actions can only occur during elaboration, SystemC does not support the dynamic creation or modification of the module hierarchy during simulation, although it does support dynamic processes." [34], p.11.

This points out that any SystemC implementation that does not exceed the standard is incapable of simulating the process of reconfiguration. And worse, an implementation that *would* allow simulation of reconfiguration in the sense of modifying the module hierarchy during run-time would not conform to the standard any longer.

For modelling of hardware-like communication SystemC provides various channel types. These channels can be accessed directly by calling their access methods if the channel is within the caller's scope. Alternatively, channels in a module's environmental scope can be accessed via `sc_ports` from within the module. Therefore the port needs to be bound to the channel (Figure 5.6). Binding a port of a submodule to a port bound to a channel allows access to the channel from within the submodule (Figure 5.7).

Additionally, channels inside a module can be made available to the module's environ-

ment by exporting their interface via an `sc_export` (Figure 5.8). Binding the export of a father module to an export, accounts to exporting the channel interface to the father module's environment (Figure 5.9).

As will be shown in Sec. 6 rewiring communication is a possible solution for describing reconfiguration within a static module hierarchy. Still, the SystemC standard forbids changing (ex-)port binding during simulation:

"Port and export binding can occur during elaboration and only during elaboration." [34], p.14.

Simulation

Delta Semantics To simulate concurrency in a sequential environment the concept of delta-cycles is commonly used. It is based on the idea of applying a channel assignment after an infinitely small amount of time. For all succeeding actions meant to occur concurrently to the assignment, the channel's value appears to be still unchanged. All virtually concurrent processes "see" the same value. If no more processes need to be woken up in the current delta-cycle, it is finished, and the new channel values are applied. These changes can now trigger processes again, starting a new delta-cycle.

Within the SystemC environment classes derived from the class `sc_prim_channel` (prim-channel in the following) automatically possess a mechanism featuring this delta-delay. If a method of the prim-channel calls `request_update()` the kernel will call a callback method `update()` of the channel after the delta-cycle has ended.

Here usually `sc_event` notification will be executed to allow processes to be sensitive to the channel's value changes.

This way the user implementing the prim-channel is enabled to apply changes to externally visible features of the class only after a delta-delay.

Additionally, event notification comes in two flavours: immediate and delta-delayed notification.

Simulation Cycles Prior to the simulation itself all callbacks to `start_of_simulation` are processed. Afterwards the simulation is started. It proceeds in several simulation cycles illustrated in Figure 5.10 and detailed in the following.

Initialisation Phase In the beginning of the simulation all initial prim-channel values need to be applied, hence the *Update Phase* is run. According to their execution semantic all `sc_method` and `sc_thread` process instances are scheduled to be run in the *Evaluation Phase*. Afterwards the *Delta-Notification Phase* is run, scheduling all process instances sensitive to any of the initialised channels.

Evaluation Phase All scheduled process instances are woken up, one after the other. Only a single process instance may run at a time. A process may execute immediate notification on some event, then all process instances sensitive to it are evaluated within the same evaluation phase (again).

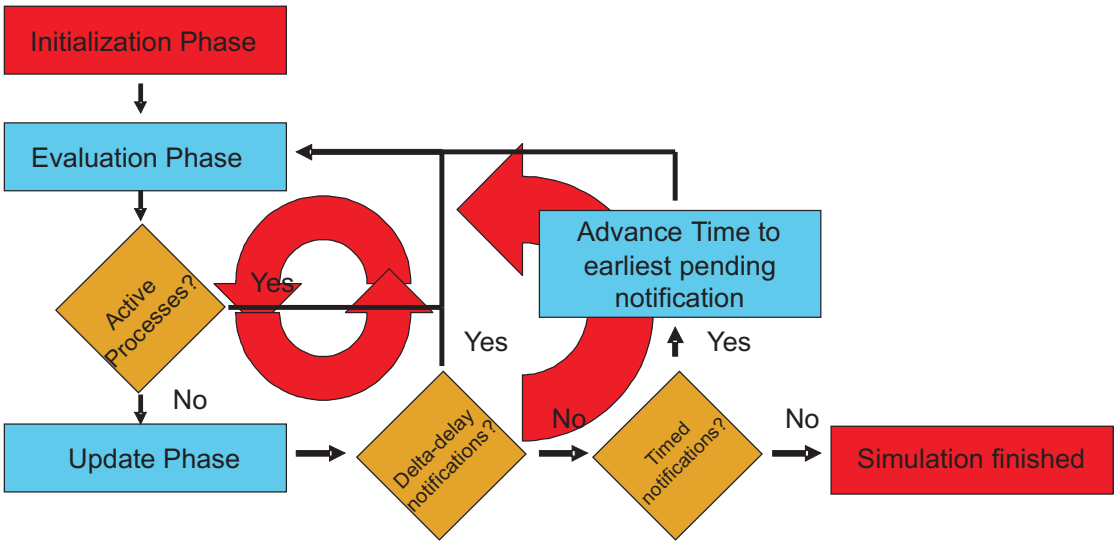


Figure 5.10: SystemC's delta notification and timed notification loop. The immediate notification loop is hidden within the evaluation phase.

Update Phase All prim-channels who's `request_update()` method was called during evaluation will be called back via their `update()` method.

Delta Notification The delta-cycle ends and pending delta-delayed event notifications are processed. Ending time-outs resulting from delta-delayed calls to `wait()` are notified. If any process instances were triggered go back to the *Evaluation Phase*, this is called *delta notification loop*.

Timed Notification Phase Advance simulation time to earliest timed notification or time-out. Schedule all process instances sensitive to these events and time-outs. If any process instances were triggered go back the *Evaluation Phase*, this is called *timed notification loop*.

5.1.2 Reconfigurable Platforms

As already discussed in Sec. 1 verifying chip functionality is a major obstacle on the way to a short time-to-market.

One way of more realistic testing than simulation is implementing the design under test (DUT) onto a prototyping platform. To avoid astronomic costs special prototyping platforms where developed that behave like parallel hardware and hence provide a (hopefully) realistic setting for the test hardware. These configurable[2] hardware devices (CHW) are direct descendants of PLA/PAL components.

[2] Sometimes the term programmable can be found in literate as well, but is not used in this work to avoid confusion with software terminology.

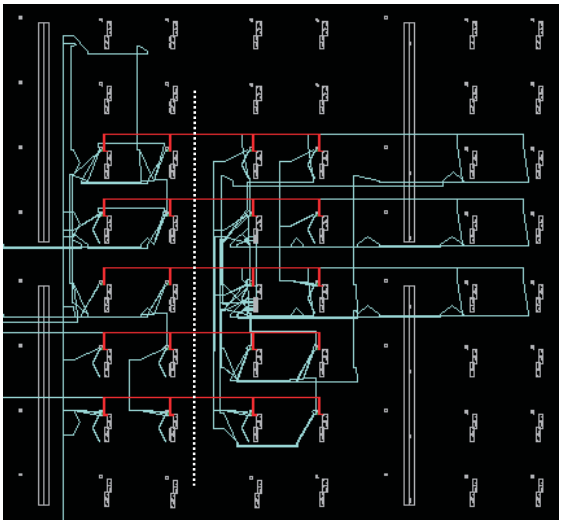


Figure 5.11: FPGA-Editor screen shot of a minimal design using reconfiguration abilities of the Xilinx Virtex II. The boundary between statical (left) and reconfigurable area (right) is marked by the doted line. Modules on different sides are connected via bus macros (red) only.

Since only those devices capable of partial run-time reconfiguration support the full scope of targeted applications in this work only those are discussed in the following.

In the following a brief overview of the most interesting and commercially relevant architectures is given. Special focus lay on choosing representative architectures with minimal in common. Afterwards a conclusion is drawn on how state-of-the-art realisation of partial reconfigurable platforms should influence high-level methodologies for describing partial reconfiguration.

Xilinx Virtex

When looking at contemporary commercially successful reconfigurable hardware, one cannot avoid to note that Xilinx is the leader in the field. Its Virtex series [77–80] is not only used for a vast number of commercial products, but also within research projects in the field of (dynamic-) reconfiguration. These devices are organised in Configurable Logic Blocks (CLBs) connected via different sorts of routing resources classified by their length. Boundaries between areas to be reconfigured (RA) and the rest design can be identified by placing special hard macros called bus macros on connections between RAs (see Figure 5.11).

These give the designer a defined way of communicating across reconfiguration boundaries.

By partitioning the design into parts on one side of the boundary and parts on the other side using placing constraints, a reconfigurable area can be defined. The bus macros provide a low-level communication interface to the rest design. Since CLBs are

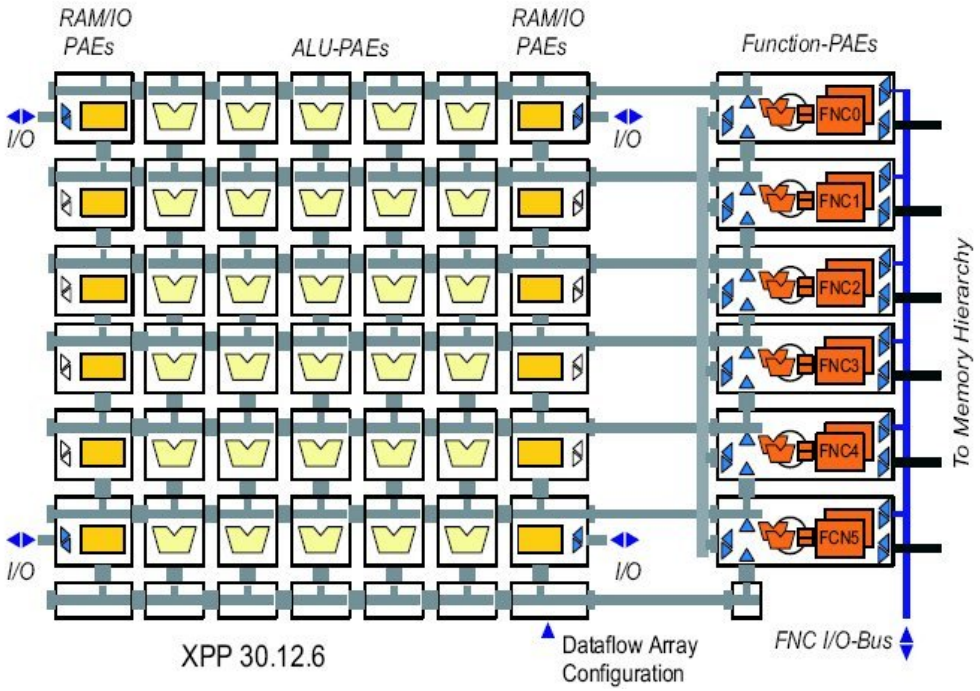


Figure 5.12: Structure of a sample XPP-III Core (taken from [53]).

rather small units the Virtex devices are classified as fine-grained or medium-grained architectures.

PACT XPP

ALU-Arrays are an interesting alternative to conventional reconfiguration hardware. They probably are the most promising coarse grained technology. The PACT XPP architecture [52,53] was originally designed with bearing partial run-time reconfiguration in mind. It is build of 3 different kinds of processing elements (PE) of which the ALU-PEs are the basic processing units. They consist of 3 ALUs each and are connected row-wise via horizontal routing buses (see Figure 5.12). These buses can be segmented by configurable switch objects. Thus reconfiguring this type of platform mainly persists of changing the ALU programmes and changing the data routing between them.

Conclusion - Architecture Specific Modelling

As the discussion above clearly indicates exchange of functionality is only one aspect of dynamic reconfiguration. Another important (and partially even more complex) issue is connectivity. Abstracting this aspect away restricts the designer to usage of a certain communication model (e.g., a certain interface as it is done by the DRCF-approach, as will be shown in Sec. 5.2.3) and hardware natively supported by the approach (e.g.,

JHDL, see Sec. 5.2.1, solely supports Xilinx Virtex II). Additionally it tends to large overhead where simpler schemes would suffice (i.e., self-reconfiguration of the device where host based reconfiguration control would result in leaner hardware).

Additionally, hardware designers often need to develop simulations (at least in the terminal phase of the project) that behave as much like the real hardware as possible to improve performance and to reduce hardware cost, but also for debugging purposes. Hence it is necessary to provide them with a methodology that is capable to model low-level reconfiguration behaviour of the chip in use. This enforces respecting connectivity issues and allows them to be explicitly modelled. This observation led to the introduction of the connectivity objective (see Sec. 2.6).

5.2 High-Level Reconfiguration Modelling

In the following an overview of existing approaches towards high-level reconfiguration modelling is given. Firstly, a brief recap of JHDL is provided, since it is a somewhat classical example of an early opponent of SystemC. It natively features modelling of reconfiguration.

Since there are only very few, but nevertheless highly relevant approaches targeting a SystemC reconfiguration methodology, they are presented in detail.

For the implementation of a reconfiguration extension process control language constructs are very useful, if not inevitable. Therefore two different works on adding process control to the SystemC language are discussed.

Each of these sections is concluded by a discussion on the applicability of the according approach.

5.2.1 JHDL

JHDL [9] is a somewhat classical example of a hardware description language natively featuring modelling of reconfiguration. It is an extension library for the programming language Java, that (not unlike SystemC does in C++) extends it with hardware modelling capabilities.

It basically allows modelling of synchronous and combinatorial circuits by structural description. Various hardware primitives are provided, that can be connected by `wire`s. Hence, a modelling language with capabilities comparable to VHDL or Verilog structural description results.

JHDLs main characteristic is that prior to simulation a complete memory model of the described hardware is built. During simulation this memory model is executed. This enables JHDL not only to simulate these descriptions, but also to extract them as a netlist. It also allows run-time construction of circuitry during simulation. This is a most intuitive way of interpreting reconfiguration.

Conclusion on JHDL

While JHDL certainly chooses the most intuitive way of describing reconfigurable hardware, it lacks a lot of other abilities. The complete absence of transaction and functional level description language constructs disqualifies it for early design space exploration. Especially, since even classical hardware modelling languages like VHDL and Verilog provide superior levels of abstraction.

Additionally, representing the complete model in memory and working on this structure, slows down simulation by several orders of magnitude compared to SystemC[3], rendering it unfit for even medium sized hardware projects.

As discussed in Sec. 5.1.1 SystemC is conceptually incapable of handling run-time module construction. Hence, this approach was discarded for the library presented in this work.

5.2.2 OSSS+R

OSSS+R [64] is certainly one of the most prominent approaches towards high-level design methodology for DPR-HW. It is based on OSSS (formerly known as SystemC-Plus), a language extension of SystemC that focuses on the introduction of object oriented modelling techniques into the hardware design cycle. OSSS allows usage of object oriented concepts such as polymorphism. OSSS+R exploits these capabilities by interpreting reconfiguration as a form of polymorphism. Therefore an introduction to OSSS will be given, before OSSS+R is detailed in the following.

OSSS

In [29] the synthesisable subset of SystemC is extended by adding constructs that enable modelling, simulation and synthesis of object oriented features. Basically three different such constructs are introduced: Polymorphic objects, shared objects and sockets. Explaining usage if polymorphism in hardware modelling is best done using an example. Figure 5.13 shows the somewhat "classical" example of an ALU being modelled using polymorphism. Addition and subtraction operations are derived from the pure virtual base-class ALUOp possessing a single function `executeCommand()`. An ALU module could now call `executeCommand()` on a member variable of type ALUOp representing the actually needed operation.

Other than in C++, within the OSSS-framework polymorphism is not based on pointers, due to the fact that pointer synthesis is known to be a difficult task that tends to result in large overhead [65]. Instead "tagged objects" are introduced. Listing 5.1 shows how this is done in practise. The basic difference to C++ is that assignments are done by copying attribute values instead of changing the reference. This way polymorphic objects possess a state space exclusively owned by themselves.

Different to polymorphism, shared objects and sockets have no counterpart in C++. Shared objects are objects that can be accessed from concurrent processes. This access

[3] Which is on RTL already approximately ten times slower then VHDL.

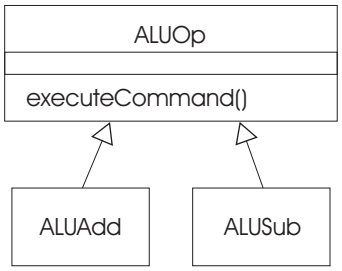


Figure 5.13: A "classical" example of object oriented hardware modelling. Two operations are derived from the pure virtual base-class ALUOp. A supposed module ALU could now call `executeCommand()` on a member variable of type ALUOp representing the actually needed operation.

```
class ALUOp
{
public:
    OSSS_TAG( ALUOp ) // tagging ALUOp
    ALUOp();
    void executeCommand();

    [ . .  . ]
}
```

Listing 5.1: Object tagging in OSSS is done by calling a tagging macro from within the header declaration.

```
SC_MODULE( BusUser )
{
 osss_shared<
        // instantiate m_mySharedBus as shared object
   myScheduler,     // use myScheduler to arbitrate access to m_mySharedBus
   myBus >  m_mySharedBus;
                      // m_mySharedBus is of type myBus

    void sendingProcess() {
        sc_bv<128 > randomNumber;
        while (true) {
            randomNumber = generateRandomBV();
            OSSS_SHARED_PC(m_mySharedBus, transmit(randomNumber) );
                            // make m_mySharedBus transmit randomNumber when
                            // able to. blocks otherwise.
        }
    }
    [ . . . ]
```

Listing 5.2: Instantiation of a shared object is done with the `osss_shared` template. Calling a function of this object is done via the macro `OSSS_SHARED_PC` that allows a call to a shared procedure.

is mutually exclusive and therefore needs to be arbitrated by a scheduler. Hence, it is necessary to nominate a scheduler when instantiating a shared object (Listing 5.2). Four different scheduling techniques are predefined, but own schedulers can be implemented as well. Calling a function of this object is done via the macro `OSSS_SHARED_PC` that allows a shared procedure call.

Declaring a class that can be instantiated as a shared object means declaring a standard C++-class with guarded function. A function is called guarded if it is only executed if a so called guard expression evaluates to true (Listing 5.3). A shared object can be bound to a submodule's shared object of the same kind. These two objects will behave as being only one. This allows usage of shared objects not only for interprocess communication, but also for communication between different modules.

Since sockets are a quite recent addition to OSSS and are of no further relevance in the context of reconfiguration, they will not be described in depth here. For the sake of completeness it suffices to say that they allow access to low level signals from within polymorphic and shared objects.

Modelling Reconfiguration with OSSS+R

The basic assumption of OSSS+R is that reconfiguration can be interpreted as an exchange of two objects sharing a common base type and can therefore be modelled with polymorphism. Assumable due to additional simulation necessities a reconfigurable object is not simply modelled as a polymorphic object in terms of OSSS. Instead, it is

```
class myBus                    // class to become a shared object
{
public:
    myBus();
    OSSS_GUARDED(              // mark function as guarded method
              void,            // return value
              transmit( const sc_bv< 128 > chunk ),
                               // actual function declaration
              m_BusFree        // guard expression must evaluate to true for
                               // transmission to be processed
    );

    [ . . . ]
```

Listing 5.3: Example of a shared object. Functions to be part of the interface needs to be marked as guarded. A guarded function is processed only if the guard-expression evaluates to true.

introduced as a special language construct named `osss_recon` container, but provides a mostly identical functionality. An `osss_recon< T >` must be instantiated with a specialising class type `T` that must be derived from `osss_context_base`. Such an `osss_recon` container represents a reconfigurable area of the target device. Now classes derived from `T` can be assigned to the container. This accounts to changing the object contained in it and therefore models a reconfiguration of the reconfigurable area. Listing 5.4 shows the somewhat artificial, but instructive example of a reconfigurable ALU. Calling member functions of `T` is done by procedure calls mostly alike those described in the previous Subsection *OSSS*.

The hardware design stored in a reconfigurable area is regarded as a context for the rest of the design. There are two kinds of contexts: first the anonymous context which is the one currently present in the reconfigurable area and so called named contexts of type `osss_context`. Referring to the anonymous context means referring to the design actually within the reconfigurable container. Referring to a named context instead, means loading this context into the reconfigurable area and then referring to it. Hence a named context needs to be bound to a reconfigurable container. Within this framework this binding stays statically the same.

To model the fact that when reconfiguring on a standard DPR-HW flip-flop states are not preserved automatically and therefore extra hardware effort is necessary, language constructs are provided that partition member variables into two classes: those that shall be preserved and those that may be discarded.

```
DURABLE_RECONFIGURABLE(ALUOp);   // all variables of ALUOp will be
                                 // preserved during reconfiguration
TRANSIENT_RECONFIGURABLE(ALUOp, myVar);
                                 // only myVar will be forgotten
```

```
class ALUOp :public osss_context_base
{
public:
    virtual void executeCommand();  // function defined elsewhere
};

class ALUAdd : public ALUOp {
    virtual void executeCommand();  // function defined elsewhere
};
class ALUSub : public ALUOp {
    virtual void executeCommand();  // function defined elsewhere
};

SC_MODULE(reconALU)
{
    sc_in< OpCodeType > OpCode;

    [ . . . ]

    osss_recon< ALUOp > m_Op;

    void reconf( ) {
        if (OpCode==ADD)            // check OpCode
            m_Op = ALUAdd();        // switch to according functionality
        else
            m_Op = ALUSub();
    }

    [ . . . ]
};
```

Listing 5.4: A reconfigurable micro ALU modelled with OSSS+R. Assume the different versions of `executeCommand()` to be defined in some other file.

The OSSS+R Reconfiguration Controller

Since objects are configured by either referring to them via a named context or by assigning them to a container (and being deconfigured by referring to another named context sharing the same container or assigning another object) the designer is not necessarily aware of concurrent accesses to different objects sharing an `osss_recon` container. Hence there need to be an instance not only controlling the process of reconfiguration for containers that are referred to, but these accesses need to be arbitrated as well. Therefore a reconfiguration controller is used[4]. Different scheduling techniques can be implemented and the ones already in existence (see Subsection *OSSS*) can be used as well.

```
ReconfigurationController< myScheduler > myController;
```

Since multiple different controllers may be present within a design a statement for binding objects to a controller is provided.

```
CONTROLLED_BY(reconALU.m_Op, myController);
```

Modelling reconfiguration timings is done via a simple statement that allows specification of save-and-restore and reconfiguration time.

```
OSSS_DECLARE_TIME(myController, ALUOp, sc_time(20,SC_US), sc_time(1,SC_MS));
    // object ALUOp controlled by myController needs 20 us to save and
    // restore its durable members and 1 ms to reconfigure
```

Conclusion

OSSS+R is an effective way to describe and simulate reconfigurable hardware. But since it is built upon OSSS it is not really a SystemC extension. Since OSSS demands a change in programming paradigm, from component based hardware design to object orientation, it must be regarded as a language of its own. OSSS+R exploits these object oriented features, therefore the vast number of IP-cores available for SystemC cannot be incorporated in OSSS+R designs.

Additionally, due to the most abstract modelling style demanded by OSSS and hence by OSSS+R the designer is not free to describe a design in a low abstraction closer to the resulting hardware, to optimise system performance.

Furthermore, due to the object oriented communication style of calling object functions to communicate with reconfigurable objects, only passive objects can be reconfigured. For hardware designers this is very contra-intuitive, since hardware's most intrinsic property is that every component has a thread of control of its own. Moreover, it is a major limitation since some behaviour can only be modelled with increased effort.

[4] Note that naming of the controller class and related topics may have changed. They are not mentioned in the newest publication, where a major renaming was done.

```
class bus_slv_if : public virtual sc_interface {
  public:
        virtual sc_uint<ADDW> get_low_add()=0;
        virtual sc_uint<ADDW> get_high_add()=0;
        virtual bool read(sc_uint<ADDW> add, sc_int<DATAW> *data)=0;
        virtual bool write(sc_uint<ADDW> add, sc_int<DATAW> *data)=0;
};
```

Listing 5.5: To enable DRCF candidate components to be automatically identified they need to implement `read()`, `write()`, `get_low_addr()` and `get_high_addr()` interface methods shown in this example interface implementation.

5.2.3 DRCF

DRCF Approach

An approach towards a reconfiguration extension to SystemC with less overhead is described in [55,57,69,74]. No explicit name is mentioned there, so it will be referenced as DRCF approach in the following. The abbreviation stands for dynamically reconfigurable fabric, the main component introduced by the approach. DRCFs are components that can contain multiple modules and are able to switch between them. This way the DRCF appears like one of these modules depending on the functionality needed.

DRCF is targeted towards transaction level (TL) description and simulation of reconfiguration aspects. It focuses on early evaluation of the performance impact of reconfiguration on the transaction level model. Therefore an automated tool is introduced that analyses static TL models and identifies components that could be made reconfigurable. The designer's task is now reduced to exploring the design space with respect to reconfiguration.

To enable this automated process the candidate components need to implement the `read()`, `write()`, `get_low_addr()` and `get_high_addr()` interface methods shown in Listing 5.5. This will enable the DRCF component, that was generated from the module, to capture and understand incoming messages directed to the module.

Components implementing these interface methods are then analysed with respect to their port interface. For this a script file specifying which candidate shall be moved into which DRCF component has to be provided. A tool called DRCF transformer then modifies the design accordingly. The transformation flow is shown in Figure 5.14.

Listing 5.6 shows a design ready for analysis. DRCF transformer substitutes the module instantiations of the candidate components against instantiations of the generated DRCF component (shown in Listing 5.8). Listing 5.7 shows the new top level module.

The `arb_and_instr()` (arbitrate and instrument) method shown in Listing 5.8 implements a context scheduler which is the main functionality of the DRCF component. It is part of the DRCF template used. Depending on calls to the interface methods shown in Listing 5.5 it decides which of the candidate modules is currently needed and activates

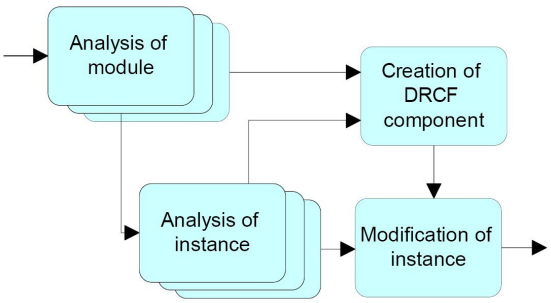


Figure 5.14: DRCF transformation flow.

```
SC_MODULE(top) {
    sc_in_clk   clk;
    bus         *system_bus;

    hwacc1      *hwa1;
    hwacc2      *hwa2;
    hwacc3      *hwa3;
    SC_CTOR(top) {
        system_bus = new bus("BUS");
        system_bus->clk(clk);

        hwa1 = new hwacc1("HWA1", HWA1_START, HWA1_END);
        hwa1 ->clk(clk);                          // instantiation and
        hwa1 ->mst_port(*system_bus);             // signal bindings for hwa1
        system_bus->slv_port(*hwa1);

        hwa2 = new hwacc2("HWA2", HWA2_START, HWA2_END);
        hwa2 ->clk(clk);                          // instantiation and
        hwa2 ->mst_port(*system_bus);             // signal bindings for hwa2
        system_bus->slv_port(*hwa2);

        hwa3 = new hwacc3("HWA3", HWA3_START, HWA3_END);
        hwa3 ->clk(clk);                          // instantiation and
        hwa3 ->mst_port(*system_bus);             // signal bindings for hwa3
        system_bus->slv_port(*hwa3);
    }
};
```

Listing 5.6: Original instantiation sequence of three parallel hardware accelerator modules connected to a bus. Assume that `hwacc1`-`hwacc3` are derived from the interface shown in Figure 5.5 and possess ports `sc_in_clk clk` and `sc_port<bus_mst_if> mst_port`.

```
SC_MODULE(top) {
    sc_in_clk clk;

    drcf *drcf_inst_1;
    bus *system_bus;

    SC_CTOR(top) {
        system_bus = new bus("BUS");
        system_bus->clk(clk);

        drcf_inst_1 = new drcf("DRCF1");
        drcf_inst_1 ->clk(clk);
        drcf_inst_1 ->mst_port(*system_bus);
        system_bus->slv_port(*drcf_inst_1);
    }
 };
```

Listing 5.7: Instantiation sequence as generated by DRCF transformer from Listing 5.6.

it accordingly.

Conclusion on DRCF

The DRCF approach is no complete methodology, but is meant to enable design space exploration on TL with respect to reconfiguration overhead. This is achieved by featuring automated design analysis and transformation as described above.

DRCF strictly constrains the designer to using certain interface methods. This constraint appears to be founded solely in the technology used.

That all examples shown by the authors and discussed here are exclusively using candidate modules that are bus slaves is not by accident. The methodology implicitly expects modules sharing a DRCF component to be connected to the same bus in the original design. Since reconfiguration can be seen as addressing a (very slow) bus this is an intuitive assumption, but it still limits the designer in his choice of communication interface implemented by the modules. Using IP seems to be principally possible, as long as an adaptor to the bus in use is provided by the designer. But no means to reset module states are provided.

It remains unclear which levels of abstraction the modules in question may be written in. But since no further detail on special handling of blocking accesses is given, it seems unlikely that any abstraction besides RTL is featured.

One major limitation is that no reset on configuration (ROC) is provided by the approach, but there seems to speak nothing against adding it in the future.[5]

None of the publications known to the author details on how the port interface is restricted by this approach or how DRCF transformer reacts if candidates with different

[5] Despite the fact that the approach does not seem to have been extended in the last 4 years.

```
class drcf: public sc_module,public bus_slv_if {
  public:
    sc_in_clk clk;
    sc_port<bus_mst_if> mst_port;

    hwacc *hwa1;
    hwacc *hwa2;
    hwacc *hwa3;

    SC_HAS_PROCESS(drcf);

    void arb_and_instr();

    sc_uint<ADDW> get_low_add();
    sc_uint<ADDW> get_high_add();
    bool read(sc_uint<ADDW> add, sc_int<DATAW> *data);
    bool write(sc_uint<ADDW> add, sc_int<DATAW> *data);

    SC_CTOR(drcf) {
        SC_THREAD(arb_and_instr);
        sensitive_pos << clk;

        hwa1 = new hwacc1("HWA1", HWA1_START, HWA1_END);
        hwa1 ->clk(clk);
        hwa1 ->mst_port(mst_port);

        hwa2 = new hwacc2("HWA2", HWA2_START, HWA2_END);
        hwa2 ->clk(clk);
        hwa2 ->mst_port(mst_port);

        hwa3 = new hwacc3("HWA3", HWA3_START, HWA3_END);
        hwa3 ->clk(clk);
        hwa3 ->mst_port(mst_port);
    }
};
```

Listing 5.8: DRCF component generated from Listing 5.6 by DRCF transformer.

port interfaces are designated to be moved into a common DRCF component.

Taking all this into account DRCF is an effective way to explore reconfiguration overhead at TL, but is conceptually too limited to provide a full featured methodology for high-level development of complex systems featuring dynamic reconfiguration in SystemC.

5.2.4 OCAPI-XL

[69, 74] present OCAPI-XL, an approach closely related to DRCF components. Since OCAPI-XL is closed source and even the documentation is not publicly available, but regarded as an internal document, very little of its usage or internals is known or even accessible to the public. Still, the authors have released numerous case studies using OCAPI-XL and it appears to allow productive development of dynamic reconfigurable applications. So it is presented here, being aware that the presentation is far from being complete.

OCAPI-XL can be regarded as a C++ like language of its own. Still it is (like SystemC) implemented in C++ using class definitions to create its own types and exploiting operator overloading to add syntactic sugar making it look like a whole new language. Technically seen an OCAPI-XL-hardware-description is a C++ program that generates a memory-representation of a hardware. After generation, simulation takes place on this memory model. Hence it follows the classical memory generation approach already described in Sec. 5.2.1, but different to JHDL the memory model is not executing itself, but is interpreted during simulation or code-generation. Interesting about OCAPI-XL is that OCAPI-XL-designs can be co-simulated with native C++ and even SystemC-code. Originally a so called Foreign-Language-Interface was provided, but proved to be insufficient in providing concurrency to processes with native C++ thread of control. Hence a threaded process extension was developed. OCAPI-XLs thread process extension is based on SystemC. And it now is a HDL interpreted by a kernel that is programmed using another HDL that is a class library of a compiled language[6]. And hence the co-simulation is merely a side-effect (see Figure 5.15)[7]. See [74] for further information on how OCAPI-XL threads can interact with the SystemC environment.

OCAPI-XL code can be of different process types:

- `procHLSW` for (possibly scheduled) software targets.
- `procANSIC, procMTHRC` for creation of ANSI-C Software
- `procHLHW` is a high level abstraction for hardware targets.
- `procOCAPI1` for FSMD hardware targets.
- `procSC` is a high level abstraction for integration with SystemC.

In the beginning a design is typically assigned to one of the high-level abstractions.

[6] C++ *is* a great thing, isn't it?

[7] Of course some work had to be spend here by the authors to exploit the benefits of this.

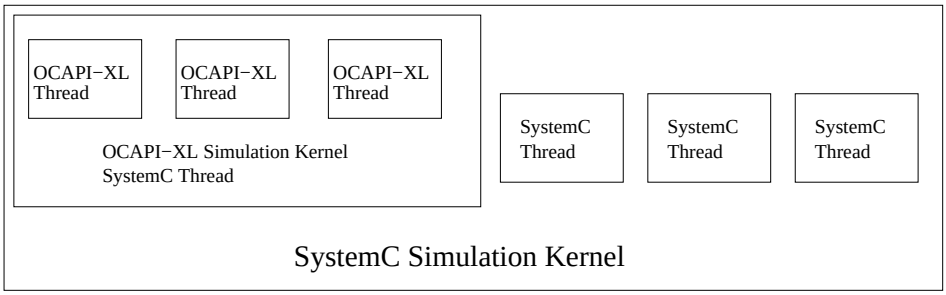


Figure 5.15: OCAPI-XLs thread process extension is based on SystemC. Some details are hidden here for the sake of simplicity. See [74] for further detail.

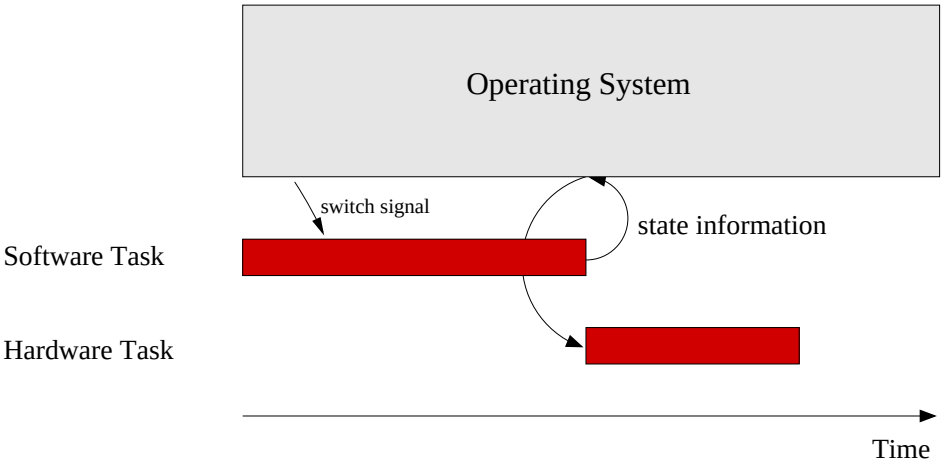


Figure 5.16: Task relocation in OCAPI-XL. When a switch signal is received from the operating system the task continues execution until a switch point is reached. Its state information is saved and it is reinitialised with this information within its new context.

Depending on the simulation obtained results, it can be refined towards one of the software directed types or the hardware directed type procOCAPI1.

Reconfigurable Context Switching

Dynamic reconfiguration is covered by OCAPI-XL as far as it can be viewed as a context switch (e.g., from software to hardware) or a re-scheduling of the task. Whenever a task receives a switch signal from the operating system it is relocated as soon as it reaches a switch point. Therefore state information is saved and the task reinitialises itself using this data when starting its execution within the new context. Tasks can contain multiple switch points and the state information saved can vary. It is up to the designer to implement the switch points in such a way that the state information to be saved is minimised. Figure 5.16 illustrates this.

Conclusion on OCAPI-XL

Too little is known about OCAPI-XL to draw final conclusions. According to the long list of examples of its successful application it appears to be effective in usage. Still it is of little academic relevance due to the information policy chosen by the authors[8].

Additionally, an obvious point for criticism is that OCAPI-XL is not SystemC and hence no standard high-level description language. Hence, the huge pool of already implemented SystemC designs cannot be used for reconfiguration without reimplementation in OCAPI-XL. Neither is clear if OCAPI-XL code can be provided closed source and so distributed as IP cores. As already discussed previously, the latter would be a major drawback.

5.2.5 Process Control

A well known deficiency of SystemC is the absence of process control statements [30]. Especially, since other system description languages (e.g., SystemVerilog) come with flavors of process control. These control capabilities are vital in RTOS (Real Time Operating System) -modelling, but also come in handy in other areas of application (e.g., test bench development, early design space exploration, etc.). Particularly implementing a reconfiguration extension would be simplified to a certain extend. As will also be discussed in this section, a very basic form of reconfiguration modelling can even be done using process control exclusively.

In [11] a modified SystemC kernel is presented providing process control statements. It features language constructs for suspending and resuming processes. If a suspended process is triggered by its sensitivity it will execute as soon as it is resumed. Disabling and enabling processes is also supported. The main difference is, that a disabled process will not care whether it was triggered or not.

Additionally, killing and resetting processes is featured. A `kill` immediately terminates the process, unwinds its function stack and destructs all local objects of the process. The process is not eligible to ever run again. `reset` performs an asynchronous reset by terminating the process in the same crude fashion as `kill` does. Additionally, all state information it might have build up is reset to the initial values. This even covers cancelling of dynamic sensitivity. Afterwards, the process is restarted. A process can also be switched to synchronous reset behaviour by `sync_reset_on` (and back using `sync_reset_off`). Afterwards the process is reset every time it is triggered by its sensitivity. These modifications are proposed for incorporation into the SystemC language standard.

A modified SystemC kernel featuring process control targeting reconfiguration modelling is presented in [2, 13]. It allows disabling and enabling [9] of all processes of a module by providing statements `dr_sc_turn_on(sc_module_name)` and

[8] or the copyright holder IMEC.

[9] The aforementioned notion is used here as well, for the sake of simplicity and clarity.

`dr_sc_turn_off(sc_module_name)`. Additionally, a statement `sc_add_constraint` for annotating time and resource consumption is provided. The timing constrained influences the time a `dr_sc_turn_on(sc_module_name)` is delayed before the according module's processes are disabled. The area constrained is not delayed any further. A very brief proof of concept code snipped is provided along with the "analysis" of a reconfigurable car window control application.

Concluding Remarks on Process Control Kernels

Both approaches need to modify the kernel in order to provide the user with process control. This (of course) is not compliant with the most recent SystemC language standard [34]. Neither of them provides any synchronisation statements for processes enabling them to group statements into primitive transactions that may not be interrupted by process control. This may not only render the design unstable, but (which might be worse if it remains unnoticed) even lead to unpredictable behaviour.

[2, 13] is presented as a reconfiguration library by the authors. The author of *this* thesis is reluctant to do so, despite the very interesting car window application. The presented features are neither simulation save (i.e., might render the simulation unstable), nor is any methodology provided that enables any kind of reset, decent reconfiguration controlling or anything else but plain process control and (added only recently) respecting time and annotating resource consumption. The authors refer to the part, that discusses the application, as "analysis" only, and obviously avoid the term "case study". This suggests that it was not developed using the modified kernel, but used afterwards to evaluate system features.

There are no other approaches known to the author, especially none that do not alter the underlying kernel implementation or that provide synchronisation statements. Hence ReChannel needs to provide its own process controlling as far as it is needed for providing the intended functionality. In case that the necessary process control capabilities will be standardised, the according functionality can (and will) be removed from the library, to improve interoperability.

6 The ReChannel Approach

As discussed in Sec. 5.1.1 SystemC lacks features for modelling reconfiguration. The previous Sec. 5.2 pointed out, that yet no approach exists, which complies to the objectives deduced in Sec. 2. Thus in the following a methodology will be derived that enables designers to conveniently describe and simulate reconfigurable systems in the number one system description language.

A SystemC extension library is proposed that respects the previously defined objectives (see Sec. 2). In the focus of this work and hence of ReChannel is the designer's convenience. Therefore this section begins with introducing the novel techniques implemented in ReChannel from a designer's point of view.

The following Sec. 6.1 presents the basic user interface of the library and the basic ideas that enable it to comply to this work's objectives. For this its main-components are presented along with code snippets illustrating their usage.

Sec. 6.2 discusses extended features necessary for full support of the current SystemC standard and to provide maximum convenience. It additionally introduces further language constructs for synchronising reconfiguration with data-flow. The latter extends ReChannel's modelling capabilities to functional levels of the abstraction hierarchy. Last but not least an extension for explicit reconfiguration modelling is presented, which provides its own process controlling without altering the SystemC kernel.

Sec. 6.3 explains how ReChannel integrates with the SystemC simulation cycles. This represents a different interpretation of its reconfiguration behaviour, as a state change of the overall system.

Sec. 6.4 proposes a methodology for developing and refining reconfigurable hardware on all abstraction levels supported by the original SystemC methodology. This is done to aid incorporation of ReChannel language primitives into practical hardware development. This methodology will be used in Part IV to introduce reconfigurable behaviour into the CollisionChip project.

6.1 ReChannel- Basic Features

As discussed in Sec. 2.3 having a unified way of modelling reconfiguration on all levels of abstraction is highly desirable, not only because it is convenient, but since it simplifies data refinement as well as structural refinement. Additionally, it is necessary to enable the designer to leave most of the design untouched, when rendering parts of it reconfigurable. Especially inside the reconfigurable modules no changes should be necessary. Otherwise integrating third-party IP, where only the interface is known, is simply impossible (see Sec. 2.2).

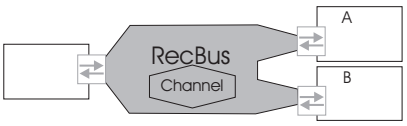


Figure 6.1: RecBus modelled as a channel.

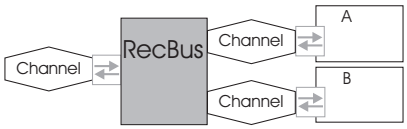


Figure 6.2: RecBus modelled as a module.

As discussed in Sec. 5.1.2 reconfiguration of programmable logic devices accounts to changing a system's inter-connectivity. In SystemC this kind of hardware property is usually described by binding module ports to channels. Hence, by changing binding of modules during simulation reconfigurable behaviour could easily be simulated. As already discussed in depth in Sec. 5.1.1 any kernel, that allows changing the port binding during run-time would no longer conform to the IEEE language standard.

The easiest and most intuitive way to circumvent this is to introduce special components decoupling binding and inter-connectivity. This is done by intercepting communication between static and reconfigurable modules at the channels, that interconnect those parts. Nowadays hardware designers usually use buses to intercept communication between static modules to let them appear reconfigurable. This is an easy and most intuitive way to model that only one module out of a set of modules is currently able to communicate via a certain channel. In this simple scheme the channel's arbiter fills the task of a reconfiguration controller. This approach is named *dynamic circuit switching* and was proposed in its simplest form in [40]. It comes in two different flavours: Firstly, the bus is modelled as a channel and substitutes the original channel (Figure 6.1). Secondly, it is made a module that connects to the original channel (Figure 6.2).

Both are not completely satisfying solutions, since some drawbacks come with them in practise:

High development effort For every SystemC channel type, that is used between static and reconfigurable modules, a custom "Reconfiguration Bus" (RecBus) has to be built from scratch, since the functional properties of the channel can differ considerably. In general the flexibility of such an approach will be very poor, if no extra effort is spent to allow connection of a random number of reconfigurable modules.

Reconfiguration cost The (dynamic re-)configuration of a system can usually not be performed instantaneously. The resulting delays might have an impact on the system's run-time behaviour or even its functional correctness. Hence they have to be considered during development. With manually crafted RecBuses, the designer has to model the delays separately, which is error prone and can be difficult.

Side effects If modelled as a channel the RecBus needs to substitute the original channel. Hence, it needs to mimic the original channel's behaviour, making it a full reimplementation. In addition to adding the switching capability, this makes it a time-consuming task even if simulation performance is ignored.
Modelling the RecBus as a module connected to the original channel unavoidably enforces that additional channels are used to connect the RecBus to the reconfigurable modules. This changes the system's topology and is error prone, since it is not necessarily clear which channel type is to be used or how it has to behave in case of reconfiguration. Furthermore, plugging another channel into the communication will in most cases change the system's timing behaviour, which might lead to unpredictable behaviour.

Automation Extending an architecture with Reconfiguration Buses can not be automated since the channel's behaviour needs to be respected in either implementation style.

In the following Sec. 6.1.1 a way to intercept communication at the channel-to-module border is presented, that resembles Reconfiguration Buses, but does not have their grave limitations described above.

6.1.1 Modelling Reconfiguration on All Levels of Abstraction

In this section portals are introduced as a framework to facilitate construction of specialised switches between channels and modules that do not cause any changes in simulation timing (not even delta-cycles) by forwarding the channel's events and the reconfigurable modules' channel accesses on C++ language level. Thus, they overcome the limitations imposed by Reconfiguration Buses while still maintaining their main advantage of being a most intuitive tool for any hardware designer.

The portals' state is controlled by the reconfigurable modules it is connected to. Reconfiguration delays, which are taken into account during simulation, can be modelled using `rc_module`s (see Sec. 6.1.2). The modules' state itself is controlled through a special simulation reconfiguration controller that can be used by the designer to model any custom controller and is presented in Sec. 6.1.3.

The current "configuration" is simulated by forwarding channel events only to the currently active modules. Additionally, only channel accesses of active modules are executed. If the modules are either sensitive to certain events or make use of blocking channel accesses, such a system will behave like a reconfigurable design during simulation.

Using Portals To Intercept Communication

A *portal* is a special switch, designed to connect a static channel to a port of a reconfigurable module, see Figure 6.3.

The portal's function is twofold:

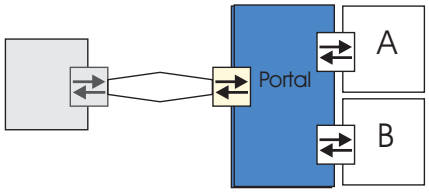


Figure 6.3: Plugging a portal between a port and its channel allows interception of their communication. Binding multiple ports of different modules to a portal allows switching data between them.

```
my_module_rc  mod;            // instantiate two reconfigurable modules
my_module2_rc mod2;

sc_fifo< int > fifo;          // instantiate (static) FiFo

rc_portal< sc_fifo_out<int> > portal;
                              // instantiate portal for  sc_fifo_out <int> port

portal.bind_static( fifo );   // connect the static channel
                              // to portal (named binding)

portal.bind_dynamic( mod1.out );
                             // bind reconfigurable module's ports to  portal
portal.bind_dynamic( mod2.some_other_out );
```

Listing 6.1: Integrating a portal into a design is done analogously to the integration of a channel. Here the usage of a fifo portal is shown. The ReChannel library predefines portals for the standard SystemC ports.

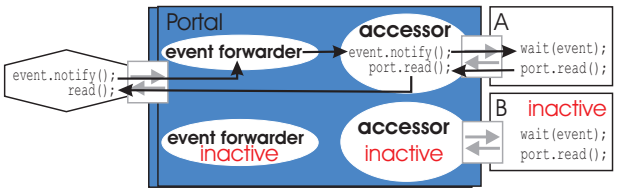


Figure 6.4: An example of a simulation sequence is shown, where a channel event is triggered by some outside source and is then forwarded to the accessor associated with the currently active module. A channel access within this module is triggered and is being executed via the accessor.

Channel Access Accesses of the active module to the channel need to be executed, while inactive modules should not be allowed to access the channel. Additionally, it is necessary to provide the module's port with an interface it is able to bind.
This is both done by binding a so-called accessor object, which is part of the portal, to the port. It needs to be derived from the interface the port can connect to and forward interface accesses to the channel.

Event Forwarding Any required events, the reconfigurable module is listening to (via sensitivity or dynamic `wait()` statements), need to be forwarded from the static channel to the currently active module.
Since the portal's accessors implement the interface the modules' port can bind, they also possess the events provided by the interface. These events are now registered with event forwarders inside the portal. These components listen to the channel's events and notify the according events inside the accessor associated with the currently active module.

Figure 6.4 shows an example of a simulation sequence, where a channel event is triggered by some outside source. It is then forwarded to the accessor associated with the currently active module. A channel access within this module is triggered and is being executed via the accessor.

During the simulation, the actual reconfiguration operations (Sec. 6.1.3) may change the data-flow through the portals depending on the reconfiguration state of the connected modules.

If all ports of a reconfigurable module are equipped with portals, no port can be triggered from outside, if the module is inactive (not configured). Therefore, no outbound traffic should occur any longer, since the module's processes are no longer triggered. Nevertheless, technically it is still possible that a module keeps on triggering itself (for instance by a member of type `sc_clock`). In this case outbound traffic is suppressed and a warning is reported to the designer. An approach capable of stopping processes, and hence effectively preventing a module from self-triggering will be presented in Sec. 6.2.5.

For standard SystemC port types corresponding portals are provided by the ReChannel library. Their usage is exemplary shown in Listing 6.1. So the common

ReChannel user will not need to construct portals himself.

Still, SystemC enables construction and usage of user-defined, possibly complex channels. Hence it is necessary to provide the user with an easy-to-use toolkit to devise portals for those custom channels. An approach that not only enables construction of portals for arbitrary channels but would also allow construction of a compiler that could do so is presented in the next subsection.

Using portals it is now possible to connect multiple modules to a channel and to switch the dataflow from one module to the other. Neither the modules nor the channel need to be altered in any way to achieve this. Therefore the IP reuse objective (Sec. 2.2) can be satisfied. Additionally, portals were implemented using standard SystemC language constructs, hence the standard compliance objective (Sec. 2.1) is also not harmed.

Furthermore, portals are motivated by *the* classic designer's approach for modelling reconfiguration, rendering them a most intuitive tool. Portal utilization is minimally invasive to the design, since neither the system's timing nor its topology are altered. In conjunction all this makes them comply to the integration objective (Sec. 2.4).

Additionally, they can be interpreted as a high-level version of bus-macros used in Xilinx Virtex reconfiguration design flows (see Sec. 5.1.2). Alternatively, portals utilization can be seen as re-channeling dataflow within the design, which is the mainly featured way of reconfiguring ALU-arrays (see Sec. 5.1.2).

In the following only blocks of portals will be controlled, but in principle it is possible to individually reroute data of every channel connected to a portal. This makes portals capable of modelling changes of inter-connectivity without altering the module hierarchy as demanded by the inter-connectivity objective (Sec. 2.6).

Creating Custom Portals

As already stated in the previous Subsection *Using Portals To Intercept Communication* it is not possible to provide portals for custom-built channels before these exist. Hence ReChannel provides the designer with an easy-to-use toolkit to devise portals for custom channels.

Basically, two steps have to be performed: Firstly, it is necessary to implement `accessor`s for the channel's interfaces, that set up the forwarding calls and register the required events. For registered events according event finders are set up automatically. Additionally, each accessor needs to provide the port with an interface it is able to bind. Therefore, the accessor has to be derived from the according interface type. Secondly, a so called `rc_port_traits` template specialisation is required, that specifies which interface/port/accessor type combination is needed for a given port. For both tasks, helper macros can be provided. Since macro usage comes with some limitations Subsection *Interface Wrapper* will discuss in how far custom accessor implementation can be facilitated by other means. Subsection *Reconfiguration Callbacks* will discuss an extension to the portal implementation providing the designer with certain hooks. The port traits' syntax will be influenced by this.

Forwarding Channel Accesses As discussed in the previous section, an `accessor` object

```
template< class T >
class my_accessor :                          // inherit from rc_accessor template
  public rc_accessor< my_channel_if<T> > {
  RC_EVENT( my_event );                      // register interface event
public:
  void bl_write( const T& data ) {           // enable forwarding of method
      RC_BLOCKING_ACCESS(
        this->channel->bl_write( data ));
  }
};
```

Listing 6.2: Create a custom accessor by implementing forwarding methods and registering required events.

has to be implemented for every interface. For the standard SystemC interfaces, these accessors are already part of the ReChannel library and can serve as reference for further accessor implementations.

The purpose of these accessor objects is forwarding of channel accesses and to provide the interface's events. They are additionally providing the ports of reconfigurable modules with interfaces they can bind.

Since blocking accesses might not return immediately, the portal must not be switched if such an access is still pending. Additionally, advanced users may want to be able to distinguish synchronisation behaviour depending on the type of access (reconfiguration synchronisation will be detailed in Sec. 6.2.4). Therefore *blocking* and *non-blocking* methods have to be distinguished, when implementing the access forwarding.

For both types of channel methods, the ReChannel library provides a macro, that takes care of the communication interception, if the calling module is currently inactive. As a result, the re-implementation is reduced to the choice of the correct macro and its usage around the "real" channel call. Listing 6.2 shows an example of the reimplementation of a blocking write function `bl_write`. Additionally, the *event finders*, i.e., the methods of the channel interface, that are used to access the channel's events, have to be re-implemented in the accessor as well. For convenience, the ReChannel library provides macros for this, so only the according event needs to be registered.

Forwarding Channel Events If the accessor is implemented for all corresponding ports (either the generic `sc_port<my_interface>`, or a specific custom port, e.g., `my_port`) of a given interface `my_interface`, the implementation of the corresponding portal `rc_portal<my_port>` is nearly ready.

All it takes, is the implementation of a template specialisation of some traits of the given port. `rc_port_traits` encapsulate the correct types of the interface/port/accessor combination and provide a static method, that specifies the

```
template< class T >
class rc_port_traits< my_port<T> > {
public:
  typedef my_channel_if<T> if_type;      // required type information
  typedef my_port<T>       port_type;
  typedef my_accessor<T>   accessor_type;

  static rc_event_list     events() {    // events to be forwarded
    return ( RC_EVENT( my_event ) );
  }
};
```

Listing 6.3: Specify interface, port and accessor types to create a new portal for a given port.

events, that are to be forwarded. An example is shown in Listing 6.3. As a result, the portal for the given port is ready to use. A slight limitation of this approach is the requirement of a fairly recent C++ compiler, that supports partial template specialisation.

This way introducing run-time reconfiguration is easy on all levels of abstraction featured by SystemC, especially on Transaction-Level where mainly custom-built channels are used. Therefore the refinement process for custom-built parts of the design can remain unchanged, with the single exception that reconfiguration needs to be taken into account (see Sec. 6.4).

Additionally, this approach not only enables construction of portals for arbitrary channels but would also allow construction of a compiler that could do so. This is due to the fact that portal construction does not depend on any creative coding of the designer, but merely is a repetition of facts known to the compiler, but not available via C++ language constructs (e.g., the type of the interface passed to `sc_port` as template parameter).

Interface Wrapper Using macros to simplify implementation of accessors as introduced in the previous Subsection *Creating Custom Portals* comes with some limitations. Firstly, methods returning values will need to be implemented using the quite unintuitive syntax shown in Listing 6.4.

Secondly, macro-code in general is hard to debug and maintain. Since the presented macros need to be quite complicated this is a real issue for further development of the library. Especially, if passing a return statement as macro parameter is supported.

Therefore, the macros were replaced by a class wrapping the channel interface, which will be referred to as interface wrapper in the following. It enables a simpler and more intuitive syntax for forwarding calls. The methods `rc_nb_forward` and `rc_forward` can now be used to forward the access instead of the original macro syntax. Additionally, utilising interface wrappers allows the use of synchronisation filters, which will be in-

```
const T& read()
{                              // enable forwarding of method
    RC_NON_BLOCKING_ACCESS(
      return (this->channel->read())
    );
}
};
```

Listing 6.4: Implementing an accessor method with return value using macros.

```
const T& read() const
{
    return this->rc_nb_forward(&if_type::read);
}
```

Listing 6.5: Implementing an accessor method with return value using an interface wrapper.

troduced in Sec. 6.2.4. They allow analysis and even modification of the data returned by the interface, for synchronising reconfiguration activities with data streams. This accounts for parameters passed to the interface as well, even in case of call-by-reference. A reimplementation of an access method, that returns a value, using the renewed syntax enabled by interface wrappers is shown in Listing 6.5.

With the release of the IEEE standard, SystemC prohibited multiple processes driving a single signal. Since portals are solely designed to connect multiple ports to an interface, and thus enable multiple processes to access the interface's channel, this needs to be coped with. Therefore, the interface wrapper provides dedicated driver access forwards. It internally creates driver objects that spawn as many driver processes of the according process type as are used by the reconfigurable modules. Thus, driver limitations of arbitrary channel types are supported. If any of the reconfigurable modules connected to the channel interface via the portal utilises more drivers than the channel allows, even the original SystemC error is reported to the designer.

Since significant overhead comes with driver objects one will not use it if not necessary. Thus special forwarding methods `rc_nb_forward_driver` and `rc_forward_driver` are provided additionally.

Reconfiguration Callbacks In many cases designers will want to include reconfiguration related functionality that is triggered by reconfiguration itself (e.g., ROC). Hence it is necessary for the ReChannel library to provide the architects with mechanisms enabling them to do so. Therefore, some methods are defined within `rc_portal` that may be overloaded and are called on different types of reconfiguration events. Reconfiguration, regarded from the portals point of view, accounts to opening or closing a

portal for a certain module's communication. Hence, callback methods `rc_on_open()` and `rc_on_close()` are provided.

After reconfiguration activities are completed it might be necessary to inform a newly activated module of channel activities it was not notified of while it was deactivated (e.g., that some value was written into a FIFO). To generate the necessary events `rc_on_refresh_notify()` can be overloaded.

In case that a portal is opened without an active accessor being present it will callback `rc_on_undef()`. Note that this is mentioned only for the sake of completeness and will never occur if the portals are controlled exclusively via the reconfigurable module's control functions, which will be introduced in Sec. 6.1.2. And is only provided for future usage in simulating mobility and explicit modelling of dataflow rechanneling.

There are two possible classes where these callbacks can be introduced into the ReChannel syntax: Within the portal itself or within the accessor. Letting the designer define callback methods within accessors would facilitate portal definition, since he would still only need to define port traits and accessors. Nevertheless, within the presented implementation the presented methods are located within the portal itself. This was done, because `rc_on_refresh_notify()` is a functionality that concerns event forwarders in the first place and would not be expected to be located within the accessor. The latter accounts even more for `rc_on_undef()`. `rc_on_open()` and `rc_on_close()` could be located in both classes without causing confusion. Still, they are provided within the portal as well, for the sake of regularity. This comes with the disadvantage, that the designer will not only need to define port traits and accessors, but to derive a portal himself to define these callbacks. If the designer does this derivation anyway, it is no longer necessary for port traits to provide the luxury of defining an event list. Instead, event forwarders can be defined far more easily within the portal derivation.

A portal will only need to instantiate a single type of accessor, which is unambiguously defined by the interface type. Therefore, the accessor typedef can be removed from the port traits definition. To further simplify portal definition, the mapping of ports to interfaces can be defined within the portal directly. Listing 6.6 illustrates the implementation of a custom accessor containing event forwarder definition and port-to-interface mapping. The previously discussed callback methods are also defined.

Note that instead of the direct typedefs port traits can also still be referenced. This is preferential, since it provides more flexibility, but is left out here for the clarity of presentation.

6.1.2 Rendering Own Components and Third-Party IP Cores Reconfigurable

Controlling the state of portals individually would be highly tedious and error-prone, since the designer would have to keep all states of portals connected to the same module consistent. Far more convenient is controlling the portals' state only implicitly, i.e., hiding it from the designer by extending the modules with reconfiguration control functions. These can now manipulate the portals' state. Therefore the module needs its own

```
template<class T>
class rc_portal<my_port<T> >
    : public rc_abstract_portal<my_port<T> >
{
       // typedef for convenience
    typedef rc_abstract_portal<myport_type<T> > base_type;
public:
       // required typedefinitions for port, interface and accessor
    typedef typename my_port<T>            port_type;
    typedef typename T                     if_type;

       // the accessor type only indirectly depends on the template
       // parameter
    typedef typename rc_accessor<if_type>  accessor_type;

    rc_portal(const sc_module_name& module_name)
        : base_type(module_name)           // calling the constructor of
                                           // the portal base class
    {
       RC_PORTAL_FORWARD_EVENT(event1);    // register events to be forwarded
        // [...]
    }

     virtual void rc_on_open()       // callback methods can now be defined
       {  // [...]
       }
     virtual void rc_on_close();
       { // [...]
       }
     virtual void rc_on_undef();
       { // [...]
       }
     virtual void rc_on_refresh_notify();;
       { // [...]
       }
};
```

Listing 6.6: Implementation of a custom accessor containing event forwarder definition and port-to-interface mapping. The callback methods are also defined.

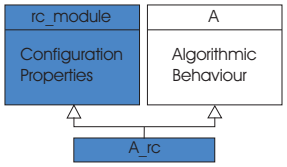


Figure 6.5: Deriving from `rc_module` and a static module `A` results a reconfigurable module `A_rc`.

reconfiguration state, which will be discussed later in this section.

Furthermore it is highly desirable to add reconfiguration related features to the modules in question (e.g., reconfiguration timings, bitfile size, etc.). Additionally, in general the designer will need to augment the modules with reconfiguration specific functionality (e.g., reset and handshaking behaviour).

Since the kernel independency objective prohibits altering kernel code all this needs to be done without altering the definition of `sc_module`. Therefore all the functionality discussed above is encapsulated into a single class `rc_module`.

Creating Reconfigurable Modules

In C++, like in most object oriented languages, adding new functionality and properties is usually done by specialisation. Hence the generic way to express, that a module `A_rc` is functionally a module of type `A` and a reconfigurable module of type `rc_module` is to derive it from both (see Figure 6.5). The original module is obviously not altered in the process, allowing incorporation even of closed source IP cores.

Since SystemC itself makes intensive use of inheritance this should be quite an intuitive way of doing things for any SystemC user. Deriving `A_rc` from `rc_module` makes it a module that can be registered with a reconfiguration controller (see Sec. 6.1.3), and (if it is equipped with portals, see Sec. 6.1.1) can be reconfigured.

The Module's States

At this point it is necessary to discuss which reconfiguration states a `rc_module` needs to provide. Therefore it is necessary to analyse a typical reconfiguration sequence.

Assuming that some initial configuration is active, the designer will first need to deactivate the module. The module's state is then `INACTIVE`. Then its internal data is preserved and it can be removed from the hardware rendering the module's state to be `UNLOADED`. Afterwards a new module can be loaded, which is then in state `LOADED`. In general there will be some internal data that was preserved before and needs to be written back, before it can be activated. It is then in state `ACTIVE`.

In average cases switching from `UNLOADED` to `ACTIVE` (and back respectively) will suffice. If a module is to be activated but was not loaded before, it is loaded automatically. This accounts for removal and deactivation functions analogously. Still, to enable de-

```
class A_rc: public rc_module, public A
{
protected:
 inline void rc_setup();

public:
 A_rc( sc_module_name name_ ): A(name_)
 {
   rc_init();        // Initialise  reconfiguration
                     // behaviour of the module

   rc_setup();       // call rc_setup
 }
};
```

Listing 6.7: Example of a manual generation of a reconfigurable module. Module `A_rc` is derived from `rc_module` and static module `A`. The constructor starts the finite state machine that takes care of the reconfiguration state of the module and calls `rc_setup()` to reset the module at start-up.

signers to model more complex scenarios like algorithm prefetching or manual variable preservation `INACTIVE` and `LOADED` are also provided.

For designers only the state changes are of any interest and hence only the according `action_type`s `RC_UNLOAD`, `RC_LOAD`, `RC_ACTIVATE` and `RC_DEACTIVATE` are visible from outside the `rc_module` class.

To enable the designer to assign the time necessary for loading, variable preservation and variable reconstruction, the function `rc_set_delay(action_type , sc_time)` is provided by the library. In general unloading will not consume any time, but for the sake of simplicity assigning a delay for it is also possible. Loading (or activating, deactivating, unloading respectively) a module now takes at least the loading (or activation, deactivation, unloading respectively) delay specified in the module's setup method `rc_setup()` (see Listing 6.7 and Listing 6.8).

Originally ReChannel allowed more than one active module per portal. This simplified signal distribution to several reconfigurable modules, since just a single portal was needed (e.g., for the clock). With the introduction of grouped binding, which will be presented in Subsection *Binding Groups of Switches*, this is not an issue any longer. Therefore the option of multiple active modules per portal was removed to simplify the library's implementation and maintenance.

Activation and deactivation can consume even more time if the module's state cannot be changed immediately. Since the SystemC standard does not allow (and the reference kernel does not enable) interruptions of pending channel accesses, they have to be executed. Due to these technical necessities a module will not be deactivated as long as it has any blocking accesses pending on any of its ports. Therefore the module's

```
virtual inline void A_rc::rc_setup()
{
  rc_resettable_var<int>(i,0);          // reset i to 0

  rc_preservable_var<sc_signal<int > > (j);
                                        // preserve j

  rc_set_delay(RC_LOAD, sc_time(20,SC_MS));
                                        // module needs 20
                                        // milliseconds to load

  rc_set_delay(RC_ACTIVATE, sc_time(1.5,SC_MS));
                                        // and 1.5 milliseconds to activate

  rc_set_delay(RC_DEACTIVATE, sc_time(2,SC_MS));
                                        // preserving variables takes some
                                        // time
}
```

Listing 6.8: Implementation of the `rc_setup()` method. In member function `rc_setup()` the integer member `i` is registered to be reset to zero and the member signal `j` to be preserved during reconfiguration. Modelling of configuration times is split into loading and activation delay.

```
RC_MODULE(A)
{
  rc_resettable_var<int>(i,0);           // reset i to 0

  rc_preservable_var<sc_signal<int > > (j);
                                         // preserve j

  // . . . .

  rc_set_delay(RC_DEACTIVATE, sc_time(2,SC_MS));
                                         // preserving variables takes some
                                         // time
}
```

Listing 6.9: A more convenient way of implementing a reconfigurable module using predefined Macros.

manipulation functions for the reconfiguration state need to be blocking.

A module cannot be activated if any of its portals still has another active module attached to it. This second module needs to be deactivated first.

Originally modules featured forced (de-)activation as well, which switched the module's portals without respecting pending accesses or any other simulation constraints. Since in most cases this leads to unexpected behaviour and may even render the simulation itself unstable, these functions were removed from the library. Instead, synchronisation capabilities were added that allow manipulating a module's reconfiguration state depending on its input-output behaviour. These will be discussed in depth in Sec. 6.2.4.

To complete generation of a reconfigurable module it is necessary to call the function `rc_init()` that initialises all reconfiguration properties from within the module's constructor. To simplify this process the macro `RC_MODULE()` can be used alternatively. Listing 6.9 shows that a condensed and elegant description results.

As can be seen in Listing 6.7 and Listing 6.9 `rc_module` generation mimics SystemC's `sc_module` instantiation and augments it with some extras. This was done to make ReChannel look like a native SystemC extension according to the integration objective (Sec. 2.4).

State Preservation

Still, there is another type of state the library has to cope with. A SystemC module will keep its state, i.e., its member variables will not be reset when removed and configured once again. A well known problem is, that hardware behaves differently. If not explicitly saved the module's state is lost after reconfiguration. Therefore a special mechanism is necessary to obtain correctness of simulation. As proposed in [64] (and recapitulated in Sec. 5.2.2) a feasible solution to this is to demand explicit description of every variable's behaviour. Therefore the keywords `rc_preservable_var` and `rc_resetable_var` are defined. Registering members to be preserved or reset is done in a special setup method `rc_setup()` (see Listing 6.7).

Still, there might be subcomponents (e.g., submodules) and processes with inner states which are not resettable in the proposed fashion. In this case the designer has to take care for correctness by other means, e.g., setting a reset signal of the reconfigurable module. This is an unavoidable limitation in rendering re-used components reconfigurable. Still, any experienced designer will reset a third party module before using it, to ensure its correct initialisation, anyway. Thus, this can be regarded as a minor problem. Still, as will be discussed in Sec. 6.2.5 this problem can be avoided, if the module's source code can be manipulated.

6.1.3 Controlling Reconfiguration Simulation Control

To control reconfiguration it would be tedious for designers to switch all portals manually. Hence a reconfigurable module can be requested to activate itself. This request is passed down to the portals, which allow switching only if no other module is registered

as active. A module is only activated if all its portals can be switched. But this implicit control of the portal's state via `rc_module` is still not convenient enough.

Designers will in general need to implement some kind of scheduler managing different configurations of modules being present on the DRHW. The choice of configuration will usually depend on some data. E.g., consider the example of a music codec implemented in DRHW. Depending on the type of input stream (e.g., mp3 or ogg vorbis encoded music) the according codec module needs to be loaded. Here, multiple variants are imaginable. Different to other approaches this work does not want to impose any limitations on how this scheduler is to be modelled or how it is to be integrated into the design. Especially, since during a refinement process different methods might be advisable. Therefore, only a basic component is provided that enables the designer to control the reconfiguration properties of the design in question, but hides all simulation specific control tasks (e.g., waiting for pending blocking accesses) from him. This component is therefore called `rc_control`. It can be interpreted as a language extension to SystemC for modelling all kinds of custom configuration controllers (CCCs). Since it provides all kind of functionality necessary for controlling the simulation of reconfiguration, it will be referred to as simulation control object or as simulation controller.

Such a simulation control object provides registration and reconfiguration control functions for `rc_module`s. `rc_control` administrates the reconfigurable modules in the design and allows manipulation of their reconfiguration states. An example of instantiating a simulation control object, registration of several reconfigurable modules, as well as activation and loading of some of them via `rc_control` is shown in Listing 6.10.

Operating on Sets of Modules

To enable the designer to express that certain modules need to change their reconfiguration state concurrently `rc_control`'s reconfiguration control functions need to be able to operate on multiple `rc_module`s. This can be used to model reconfiguration of multiple independently reconfigurable logic devices. As illustrated in Listing 6.10 this additional feature is also provided with a most intuitive syntax. It is implemented by providing a module set class defining a constructor accepting `rc_module`s. Furthermore it overloads `operator+()` and `operator-()` with set union and intersection functionality. `rc_control`'s reconfiguration functions now only need to operate on `rc_module_set` to implement the according functionality for a whole set of modules at once. State manipulation functions on sets of modules take the time for manipulating the reconfigurable module with the maximum delay.

Intermediate Recap

Giving an overview of the methodology proposed so far in this doctoral thesis (and implemented in the ReChannel library) is best done considering the minimal example depicted in Figure 6.6. It shows a reconfigurable design with two alternatively present

```
rc_control ctrl;                          // create simulation control object

ctrl.add(mod_1+mod_2+mod_3+mod_4); // register four reconfigurable
                                          // modules with simulation control

ctrl.load(mod_1);                         // load module 1

ctrl.activate(mod_1);                     // activate module 1

ctrl.activate(mod_2+mod_3);               // load and activate modules 2 and 3
```

Listing 6.10: Example of instantiating a simulation control object, registration of several reconfigurable modules with it, as well as activation and loading of some of them via rc_control. After `mod_1` was activated, `mod_2` and `mod_3` are activated concurrently.

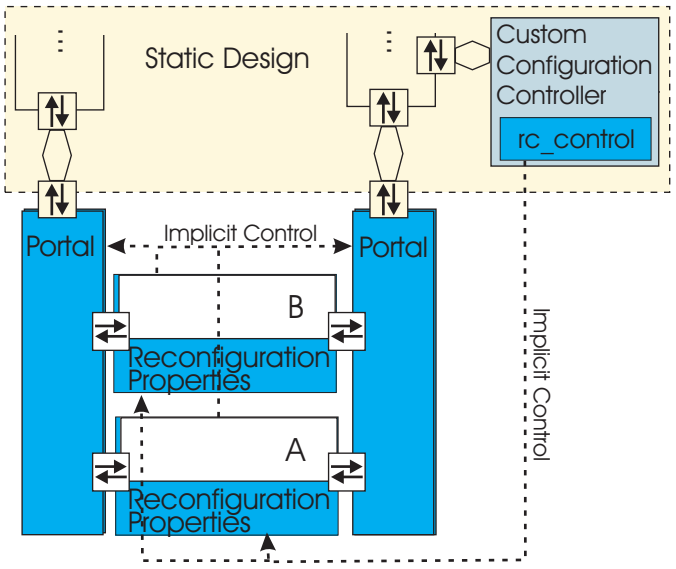


Figure 6.6: Design with two alternatively present modules. Their reconfiguration state is controlled by a custom configuration controller using an instance of `rc_control`.

modules. Their reconfiguration state is controlled by a custom configuration controller using an `rc_control` simulation control object. Both modules posses only two ports connecting them to the static part of the design. This communication is routed through portals. The portal's states interact with the modules' states only implicitly, i.e., invisible for the designer.

6.2 Advanced ReChannel Features

Intensive testing of the already presented features with toy applications revealed some limitations of the proposed methodology. In the following these limitations will be discussed and appropriate solutions will be proposed. In conjunction with rendering the library compliant to the (at that time novel) SystemC standard [34] the incorporation of these solutions became a full reimplementation of the library.

6.2.1 Reconfigurable Overhead In Static Applications

Due to these changes, which will be detailed in the subsequent sections, reconfigurable modules are no longer light-weight components. Because of the complicated structure of SystemC, C++ compilers will in general not be able to eliminate this memory overhead even if it is not used. Still, the designer will usually want to reuse the same component type in reconfigurable *and* non-reconfigurable contexts. This applies even more for modules explicitly designed for reconfiguration, which will be introduced in Sec. 6.2.5. Therefore the original `rc_module` was split into two different constructs: `rc_module` now provides all the language constructs for modelling reconfiguration, while the according behaviour and necessary interfacing capabilities are encapsulated into a novel class `rc_reconfigurable`. Instances of type `rc_module` will be referred to as *feature module* in the following. A *reconfigurable module* from now on denotes instances of classes derived from `rc_reconfigurable`.

Hence new macros for creating reconfigurable modules from feature modules and standard SystemC modules are required. Since it is of some benefit for some of the extended techniques detailed in the following, the original module is also wrapped by a template type `rc_reconfigurable_module<class>` which does the actual derivation from `rc_reconfigurable`. A reconfigurable version `M_rc` of a static module type `M` can now be generated by deriving from the wrapper class.

```
class M_rc:
    public rc_reconfigurable_module<M>
```

The resulting type `M_rc` is also of type `M` and `rc_reconfigurable`. To provide more convenience this process can further be automated by providing a macro `RC_RECONFIGURABLE_MODULE_DERIVED`, which accepts the static module type as parameter and a macro `RC_RECONFIGURABLE_CTOR_DERIVED`, where the user can define reconfiguration related properties (e.g., the module's loading delay).

```
RC_RECONFIGURABLE_MODULE_DERIVED(M_rc, M) {
 RC_RECONFIGURABLE_CTOR_DERIVED(M_rc, M) {
   rc_set_delay(RC_LOAD, sc_time(1, SC_MS));
}}
```

```
class virtex4module
{
  public:
    virtual const unsigned v4_bitfilesize() const =0;
    virtual ~virtex4module() {};
};

class v4A : public A_rc, public virtex4module
{
    v4A( const char* _name ) : A(_name) {};
    virtual const unsigned v4_bitfilesize() const {return 100;};
};
```

Listing 6.11: A Virtex-4 property class and a derived module.

6.2.2 Accuracy of Reconfiguration Delays

Modelling reconfiguration delays using estimates as it is described in Sec. 6.1.2 is an important first step in design space exploration. This way first information can be gained, if the model still meets its performance constraints when dynamic reconfiguration is used. But for a final decision in favour of reconfiguration, this will usually not suffice. More precise timing information will be needed to fine tune algorithms or to decide which hardware platform will be targeted.

Therefore ReChannel offers a mechanism that allows description of specialised simulation controllers that mimic reconfiguration timing of a target platform.

Let `A_rc` be a `rc_reconfigurable` and `PlatformProperty` be a platform dependent property type. `PlatformProperty` can now contain additional module properties that depend on the target platform (e.g., bitfile size on the according platform). Now `A_rc` can be equipped with the platform's properties (e.g., specify its bitfile size).

A specialised simulation controller `PlatformControl` can now be derived from `rc_control` that calculates reconfiguration timings based on these module properties. This can be done by overloading the member function `takes_time` of `rc_control`.

A reconfigurable module can even be equipped with properties of multiple platforms and hence behaves differently under control of different controllers. This enables investigation on the impact of different hardware platforms on the system's performance.

Using platform dependent properties accuracy of reconfiguration delays only depends on accuracy of the platform's behaviour model and the property estimation. The latter will usually still be a guess. But estimating a circuit's size for instance, will usually be much more precise than directly guessing reconfiguration delays.

Calculating reconfiguration delays out of properties will usually not be very difficult. For example, for a Xilinx Virtex-4 FPGA, the bitfile size has to be divided by the block size (1 or 4, depending on whether the internal configuration port (ICAP) is running in 32 Bit mode) and dividing the result by the clock frequency the ICAP is running on.

```
class virtex4ctrl : public rc_control
{
  // implement rc_control constructors and set member variables to sane defaults
  virtex4ctrl(  ) : rc_control(),Mode32(true),ICAPFreq(100) {};

  virtex4ctrl( const char* _name ) :
    rc_control(_name),Mode32(true),ICAPFreq(100){};

  const sc_time takes_time (const ReChannel::rc_reconfigurable& _mod,
                    ReChannel::rc_reconfigurable::action_type _action ) const
  {
    const virtex4module* v4mod = DCAST<const virtex4module*> (&_mod);
                                              // cast to Virtex-4 property type

    if (0!=v4mod) {                           // if module is Virtex-4 module
                                              // treat it accordingly
      switch ( _action ) {
          case (ReChannel::RC_LOAD):           // if module is loaded

              double block=1.0;               // respect 32Bit mode
              if (Mode32) block=4.0;

              return sc_time(                 // and calculate time
                   ((double)((*v4mod).v4_bitfilesize()) / block)
                   / ((double) ( ICAPFreq ) * 1000000.0 ), SC_SEC);

          default: // in all other cases use delay set with rc_set_delay
              return ( ReChannel::rc_control::takes_time(_mod, _action) );
       }
    } else // if module is no virtex4 module use delayed set with rc_set_delay
       return ( ReChannel::rc_control::takes_time(_mod, _action) );
  }

  void set32BitMode(bool i_Mode32) {Mode32=i_Mode32;};
       // switch ICAP's 32Bit Mode on/off

  void setICAPFreqMHz(unsigned i_ICAPFreq) {ICAPFreq=i_ICAPFreq;};
       // set frequency ICAP is running on

  // ....
};
```

Listing 6.12: Derivation of a controller that mimics Xilinx Virtex-4's reconfiguration behaviour from `rc_control`. The overloaded `takes_time` member function calculates the loading delay of modules from their Virtex-4 specific bitfile size. Loading time is additionally influenced by the frequency `ICAPFreq` the Virtex-4's configuration unit is running on and if it runs in 32bit mode `Mode32`.

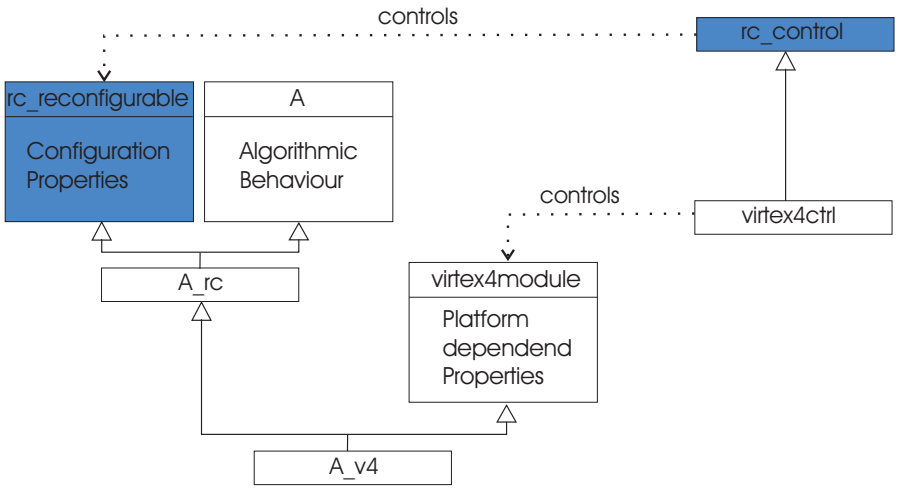


Figure 6.7: UML diagram of a module type and a controller type implementing platform dependent reconfiguration properties.

In Listing 6.12 a controller that calculates Xilinx Virtex-4's loading time with respect to the module's bitfile size is exemplarily derived from `rc_control`. Listing 6.11 shows the according Xilinx Virtex-4 property type and a module implementing its single property.

Figure 6.7 shows an UML diagram of a module type and a controller type implementing platform dependent reconfiguration properties.

6.2.3 Exportals

To simplify connecting pre-manufactured modules SystemC recently provides `rc_exports`. They can be used to export a member channel's interface. Another module's port can now be connected to this interface by binding it to the `rc_export`. Another option SystemC gives its users is to derive modules directly from `rc_channel` or from a class derived from `sc_interface`. The latter will be called interfaces in the following, despite the term's ambiguity[1].

A portal is a specially designed component to connect a channel of a design's static part to ports of reconfigurable modules. Hence it cannot be connected to interfaces, no matter if exported or not. To allow reconfigurable modules to provide interfaces as well as ports, the portal concept needs to be generalised. Therefore, its control interface was encapsulated into the `rc_switch` base class. `rc_portal` and the novel `rc_exportal` are derived from it. The latter will be referred to as exportals, since binding them to an exported interface certainly is the common case they will be used for. Additionally, if an interface is wrapped with an exportal, the latter fills the place of an export[2]. This way

[1] Seen from an C++-developer's point of view `sc_interface`s are nothing but C++-interfaces, hence this ambiguity is only apparent anyway.

[2] Interfacal sounds strange anyway.

implicit control from module to portal and from module to exportal is implemented using the same mechanism. Additionally, an exportal needs to forward channel events in the opposite direction than a portal does, from reconfigurable to static end. Nevertheless, the same techniques that are implemented in a portal can be used.

Interface accesses on the other hand are more difficult. Since they need to be forwarded from static parts of the design to reconfigurable ones, it can occur that *no* reconfigurable module is currently active to provide a channel implementing this interface to answer the request. Therefore a fallback interface needs to be supplied that answers the request. Pre-specifying a general fallback interface is obviously not possible, since the channel's interface methods must be specified. Hence the designer may specify a fallback interface individually for every exportal.

Implementing the fallback interfaces's access functions, two cases must be distinguished: If the access is blocking, the exportal can simply wait until a module is activated that can execute the request. But in case of a non-blocking access it must be executed immediately. If this occurs the design will in general be erroneous, but still the access has to be executed due to SystemC's execution semantic. The behaviour of the fallback interfaces provided by ReChannel for SystemC's standard interfaces is to issue a warning in this case.

In case of resolved signals some other options are available as well. Here the channel is explicitly constructed to model high-impedance, which can be interpreted as "unconnected". Hence the fallback interface simply returns "Z".

One could argue, that this component is not a plain interface, since it provides implemented methods. But since communication ends here it is not a channel in the strictest sense either. And since its choice is interface specific, it was named fallback *interface.*

6.2.4 Synchronisation Filters

Blocking accesses can cause problems, too. This type of access is usually only executed immediately if the channel is ready to do so. Otherwise a `wait` statement is reached, causing the calling process to suspend. As soon as the channel is ready to answer the request, it does so and re-awakens the process. Since portals are plugged between port and channel they can only gain control if an access is initiated, when it is finished or if a channel event is notified. This renders it difficult to control when the module is to be deactivated without input and output data becoming asynchronous.

E.g., a module reading from an input port `A` and writing to an output port `B` must in general not be deactivated if it has read the input, but no output was written yet.

Picking the right moment to initiate reconfiguration activities can also become very tricky, if it is to be synchronous with some datastream features.

Therefore it must be possible to either define blocks of code within the module's processes as atomic transactions, or to externally define synchronisation conditions depending on the module's communication behaviour. The former is the far more elegant solution and hence is supported and will be detailed in Sec. 6.2.5. Still, it rules out IP reuse, since internal processes of the modules need to be altered. Hence the latter

approach needs to be supported by ReChannel as well to comply to the IP objective (see Sec. 2.2).

Therefore an external synchronisation mechanism must be provided in order to allow non-intrusive synchronisation of the module's channel accesses with reconfiguration activities. As discussed in [17] adding synchronisation conditions after derivation is a difficult undertaking, due to the commonly known inheritance anomaly and can in case of parallel software be done using so-called guards[3]. Since usage of guards would imply changing at least the SystemC kernel, if not even the C++ compiler itself, they are not applicable in this case.

But external synchronisation is nevertheless possible, and thus another solution is proposed in this thesis. Within the ReChannel framework switches can be equipped with synchronisation filters, which allow bookkeeping of channel accesses and synchronisation with datastreams. As discussed in Sec. 6.1.1 the usage of interface wrappers as targets instead of the channel interface itself, allows filters to access and manipulate the data passed to the channel, even in case of call-by-reference.

Such a filter is an accessor whose access functions are overloaded with synchronisation behaviour. In the current ReChannel implementation accessors are derived from `rc_interface_filter`. The designer can now derive a concrete filter from a concrete accessor. Thus concrete filters need to be accessors that themselves provide the filter base interface.

The accessor itself is extended with the capability to forward accesses not only directly to the interface wrapper, but to a user defined filter instead. Since the filter is an accessor itself it is able to further forward the access to another filter or the interface wrapper. This way filters can be cascaded forming a filter chain of arbitrary length beginning with the accessor. Each filter in the chain adds its synchronisation condition to the synchronisation behaviour of the individual accessor it belongs to. Figure 6.8 illustrates this.

This way the inheritance anomaly is annulled, since filter chains can obviously be built and inherited independently from the module's behaviour as demanded by [17].

Still, filtering channel accesses does not suffice to provide a fully functional filter mechanism. Events also need to be filtered to enable the designer to manipulate all kind of communication between module and channel. But forwarding all event notifications through the filter chain would impair performance. Since an event is either getting notified or not it suffices to provide filters with a method with boolean return value, reporting whether the according filter lets the event notification pass or if it does not. In order to provide full flexibility filters are also able to generate these event notifications themselves. This enables them to fake any channel state.

[3] Please refer to [17] for further detail on guards and the inheritance anomaly.

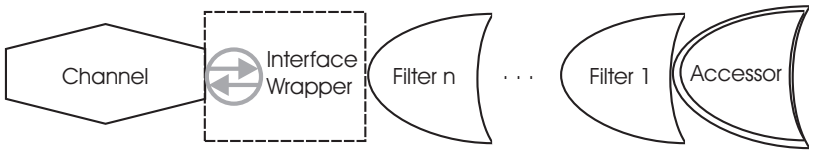


Figure 6.8: Filters can be cascaded forming a filter chain of arbitrary length beginning with the accessor. Each filter in the chain adds its synchronisation condition to the synchronisation behaviour of the individual accessor it belongs to. Usage of interface wrappers as targets instead of the channel interface itself, allows filters to access and manipulate the data passed to the channel. This way the inheritance anomaly is effectively annulled.

Transaction Counters

Different types of synchronisation requirements can be implemented using filters. The simplest, yet probably most common type is the necessity to synchronise data provided by different channels. This can easily be achieved by introducing transaction counters. Only if all counters equal zero the module can be deactivated. Reconfigurable modules may now equip their accessors with these filters and define synchronisation conditions using information supplied by the filters.

E.g., let `M` be the IP core with the previously described behaviour. If reading from input port `A` increases and writing to output port `B` decreases a transaction counter, then the module can only be deactivated if it performed equally many reads and writes. Listing 6.13 shows how this synchronisation might be implemented using filters, that solely manipulate transaction counters.

Filter Callbacks

For more complex synchronisation necessities another usage of synchronisation filters is provided as well. When instantiating the filter the designer may specify module member functions that are called if an access is initiated and if it is finished. This way synchronisation conditions have control over the module's members and can collect additional information about the module's state. This can be used for, e.g., synchronisation with incoming messages or internal state changes. Additionally, it enables the designer to synchronise thread processes of derived modules with each other, e.g., by waiting for an event.

Full Implementation of a Synchronisation Filter

Last but not least the designer has the possibility to overload all access functions of a filter. This will only in few cases be necessary. Still it is provided as a last resort, since for custom-built channels nearly every kind of behaviour is technically possible

```
RC_RECONFIGURABLE_MODULE_DERIVED(M_rc, M)
{

 RC_RECONFIGURABLE_CTOR_DERIVED(M_rc, M)
    filterA(tc,1),              // if data is read begin transaction
    filterB(tc,-1)              // if data is written end transaction
 {
    rc_set_filter(A, &filterA); // apply filters
    rc_set_filter(B, &filterB);
 }

 rc_transaction_counter tc;     // initially equals 0

 rc_fifo_in_filter<int>  filterA;
 rc_fifo_out_filter<int> filterB;

}
```

Listing 6.13: Example of a communication synchronised using transaction counters. Reading from input port `A` and writing to output port `B`, both inherited from base class `M`, are grouped into a transaction.

and hence every kind of synchronisation necessity might arise. Within the filter the designer has full control to interrupt as well as continue or delay an access whenever it is initiated or completed. Coupling certain accesses is also possible and was exemplary used in ReChannel to implement a `fifo_filter` featuring access limits. It enables the designer to provide the derived module, that was rendered reconfigurable, with faked information about the fill level of the connected FIFO. This way the user is given full control over the number of datapackets the module can possibly extract from (and input into) the FIFO.

A special variant of a full filter implementation is to derive from an accessor and augment its access functions by overloading them with synchronised equivalents. This is a very light weight filter implementation, since the access does not need to be forwarded to any explicit filter object. Still, it comes with the drawback of reduced comfort of usage, since it demands to specify the concrete accessor type the module port is to be bound to. For FIFO's one can take for granted, that the designer will not want an access to be blocked inside the FIFO in any case, but that it is preferential to block it already in the filter chain. Thus ReChannel provides a filtering FIFO accessor by default which prevents the channel from blocking. It identifies situations where the channel would block and invokes a specially prepared `wait()` method itself instead. The latter enables resetting and save deactivation of the module. This will be discussed in more detail in Sec. 6.2.5.

6.2.5 Explicit Description of Reconfiguration

As discussed in Sec. 6.2.4 grouping channel accesses into uninterruptable transactions is necessary to enable synchronisation of reconfiguration related activities with data. Using closed-source third-party IP cores as reconfigurable modules the designer has no choice but to utilise filters. If the module is built from scratch or if it is augmented with additional reconfiguration related behaviour, it is possible to provide the user with extended modelling capabilities, allowing him to define transactions inline.

Since this is not inheritance save with respect to the inheritance anomaly discussed in the previous section, in certain situations filter usage might still be preferential. But SystemC designs usually do not inherit from modules, due to the component based design style, which in general does not go together well with inheritance. Because of this and the development overhead that comes with filter design, designers will usually want to define synchronisation conditions inline.

A graver limitation concerns the reset of the module's state on reconfiguration. As described in Sec. 6.1.2 preserving and resetting of member variables explicitly marked using `rc_preservable_var` and `rc_resettable_var` can be achieved. Reset of arbitrary subcomponents or processes can only be achieved if these subcomponents themselves provide the according mechanism. Hence for originally static modules rendered reconfigurable using the according functionality presented in Sec. 6.1.2 the designer has to care for correct reset behaviour by other means (i.e., triggering the modules native reset mechanism by setting certain signals).

If a module is tailored to be reconfigurable this can be avoided. Therefore ReChannel language constructs possess an implicit reset mechanism being triggered on reconfiguration. These language constructs are primarily comprised of classes, functions and macros corresponding to a particular functionality already known from SystemC (e.g., `rc_signal`, `rc_fifo`, `rc_event`, `rc_semaphore`, etc). With the availability of resettable components and processes, both structure and behaviour of reconfigurable modules can be modelled in an intuitive way without the need to care about additional logic that deals with reset itself.

Resetting a module accounts to resetting its processes and its sub-components (variables, channels and sub-modules). Hence all of the sub-components and contained processes depend on a particular reconfigurable module preceding them in SystemC's object hierarchy tree (i.e., the first object among a component's parent list which is derived from class `rc_reconfigurable`). This object will be referred to as *context module* of its sub-components. If no such parent exists, a component is said to be used in a *non-reconfigurable context*. Resettable components and processes are designed for utilisation within context modules, but may also be employed in a non-reconfigurable context. In the latter case these components behave like their original SystemC counterpart.

In a reconfigurable context ReChannel's predefined components can be used without further knowledge of the underlying mechanism. How to custom-build resettable components will be outlined in the next Subsection *Resettable Components*.

To enable process reset, ReChannel provides its own process registry and process control API for internal management of resettable processes. The API directly builds

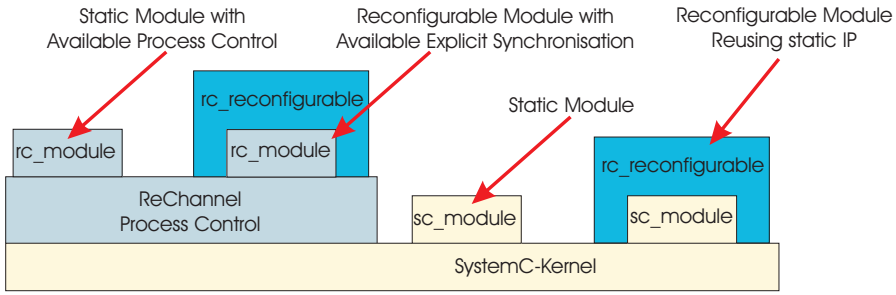


Figure 6.9: ReChannel's process control is layered on top of the SystemC kernel instead of altering it. It is therefore compliant to the IEEE 1666 standard [35] as demanded by the standard objective.

upon standard SystemC functionality and therefore can be interpreted as an additional layer on top of SystemC's process infrastructure. Figure 6.9 illustrates this. Hence it does not alter the SystemC kernel and is therefore compliant to the IEEE 1666 standard [35] as demanded by the standard objective.

Resettable Processes

Macros available for process declaration are `RC_METHOD`, `RC_THREAD` and `RC_CTHREAD`. They semantically correspond to the respective SystemC process types and can be used in the same manner.

The process registry needs to keep an account of whether a process is resettable or not. Thus, if a process has been declared through one of the aforementioned macros it is registered during its construction as resettable. SystemC processes are registered as soon as they call one of ReChannel's process control functions for the first time. E.g., if one of ReChannel's special wait statements within a filter is invoked. This way native SystemC and ReChannel processes can coexist in a single module. This is necessary to enable augmentation of a static module with additional dynamic behaviour after rendering it reconfigurable.

Primary reset condition is the deactivation of the context module. This way deactivation of the context module initiates a reset of all its processes and processes of submodules.

Somewhat as a byproduct additional reset conditions may be assigned by the user when declaring the process. The invocation of `reset_signal_is()` will result in the process being reset on the occurrence of an edge of a signal. The reset behaviour of a process may either be set to synchronous (default for `RC_THREAD`) or asynchronous (default for `RC_CTHREAD`). This way a fully-featured reset mechanism like the one introduced in [11] and already discussed in Sec. 5.2.5 is provided without altering the kernel. Listing 6.14 shows an example of a synchronously resettable thread declaration.

Classes of type `rc_reconfigurable_module` and `rc_prim_channel`, amongst others,

```
RC_RECONFIGURABLE_MODULE(M_rc)
{
    void proc;                          // functionality defined elsewhere

    RC_RECONFIGURABLE_CTOR(M_rc)
    {
        RC_THREAD(proc);                // registering proc as a thread process
        sensitive << clk.pos();         // it is sensitive to clk
        reset_signal_is(reset);         // and possesses a reset signal
        rc_set_sync_reset();            // be synchronously resettable with clk
    }
};
```

Listing 6.14: Example of a synchronously resettable thread declaration.

```
void proc() {
        [...]
  while(true) {
    wait(new_input_available);
        [...]                           // do something
  }
}
```

Listing 6.15: Example of the process function inside a module tailored to reconfiguration. While it is apparently not different to an according function in a static module, an overloaded `wait` is used. This enables cancellation of the process if its context module is deactivated. It will be restarted as soon as the context module is reactivated.

possess overloaded `wait()` and `next_trigger()` methods. These are tailored to check the reset conditions of a process. Consider the example of the process function shown in Listing 6.15 inside such a module.

If the module is blocked by the `wait` and a reset is triggered due to deactivation of the context module, the execution of the process is cancelled. It will be restarted as soon as the context module is activated again.

Still, it is required that the executed code can be cancelled safely in respect of algorithmic correctness and data consistency. Cancellation of transactions or blocking operations, not specially designed to be cancellable, would render a design highly unreliable with respect to simulation stability and correctness. This implies that a reset mechanism requires fine-grained control of where and when a process may be reset. This has already been discussed in-depth in Sec. 6.2.4 with respect to static modules rendered reconfigurable. In case of natively reconfigurable processes a different solution can be applied.

Here a macro `RC_TRANSACTION` can be provided to enable the designer to enclose blocks of code that must be finished before the reconfigurable context can deactivate.

```
[...]
   x = input.read();            // read input (blocking)
   RC_TRANSACTION {
                                // after data has been read
     y = calc(x);               // calculation must not be interrupted
     output.write(y);           // until output was written
   } // possible point of deactivation (if requested)
[...]
```

Listing 6.16: Example of transaction definition inside a resettable process.

Listing 6.16 shows an example of its application very similar to the one discussed in Sec. 6.2.4.

Still, it can occur that a resettable process calls external code, that is not intended for being cancellable at any time (e.g., by accessing a standard SystemC channel through a port). In this case the reset functionality has to be restricted: For the reset of thread processes to work, it is required that these processes have previously been suspended in one of ReChannel's prepared `wait()` methods. Whereas if a thread process calls a function or interface method, that uses SystemC's native `wait()` functionality, the reset mechanism will be temporarily unavailable. Due to this characteristic a reset condition can be considered to be locally bound, e.g., within the borders of a module.

ReChannel also supports resettable spawned thread processes. In contrast to non-spawned processes these are considered to be temporary, i.e., they will be physically terminated if their context module is deactivated. Thus their inner state is effectively "reset".

Reset signals can also be temporarily disabled for all process types. For this purpose ReChannel provides the macro `RC_NO_RESET`.

Resettable Components

ReChannel already provides resettable versions of all basic SystemC channels, (e.g., `rc_fifo`, `rc_signal`, `rc_signal_rv`,`rc_semaphore`, etc.) and the event class `rc_event`. Additionally, as already illustrated in Sec. 5.2.5 `rc_resettable_var()` allows declaration of resettable variables of arbitrary type.

A custom-built component can be easily rendered resettable by deriving it from `rc_resettable` and implementing the inherited abstract base interface.

The particular state such a component is reset to can be assigned beforehand during the construction phase. At start of simulation the callback method `rc_on_init_resettable()` is invoked once on all resettables, which now store their initial state after construction. The request of an immediate reset is propagated by a call to `rc_on_reset()`. Listing 6.17 shows an example of a custom resettable component

```
class myComponent : public rc_resettable
{
    [...] // implementation of myComponent

  // preservation of initial state
  virtual void rc_on_init_resettable()
  {
      p_reset_value = p_curr_value;
  }

  // definition of reset functionality
  virtual void rc_on_reset()
  {
    p_curr_value = p_reset_value;
  }

private:
  int p_curr_value, p_reset_value;
};
```

Listing 6.17: Example of a custom resettable component.

implementing `rc_resettable`'s abstract base interface.

A component derived from `rc_prim_channel` or `rc_module` is implicitly derived from `rc_resettable`.

For the reset mechanism to work, resettable components automatically register themselves with the current context module during construction. Hence the designer does not have to care about any further details.

6.2.6 Binding Groups of Switches

If a channel or export is connected to a module it is bound to a port by a single binding statement. If reconfiguration is used switches are bound to multiple module's ports at the dynamic end. In practise modules provide a vast number of ports, especially in RTL descriptions. In conjunction this results in nearly identical and long blocks of binding statements, which make the code difficult to read. Even worse is that implementing these binding blocks is error-prone, and that they are difficult to maintain.

To enable convenient use of ReChannel, port maps are provided, which group ports, channels and exports. As a counterpart switch connectors can be used to group switches. Switch connectors and port maps can now be bound using a single statement.

Moreover, port maps can be used to equip a module with multiple binding schemes. This allows e.g., to provide bit-vectors in little-endian or big-endian bit order. While the standard SystemC check for type compliance of bound objects is still provided, it is even extended with a check of port map compliance. E.g., a port map for little-endian order can not be bound to a switching connector which is defined for big-endian order.

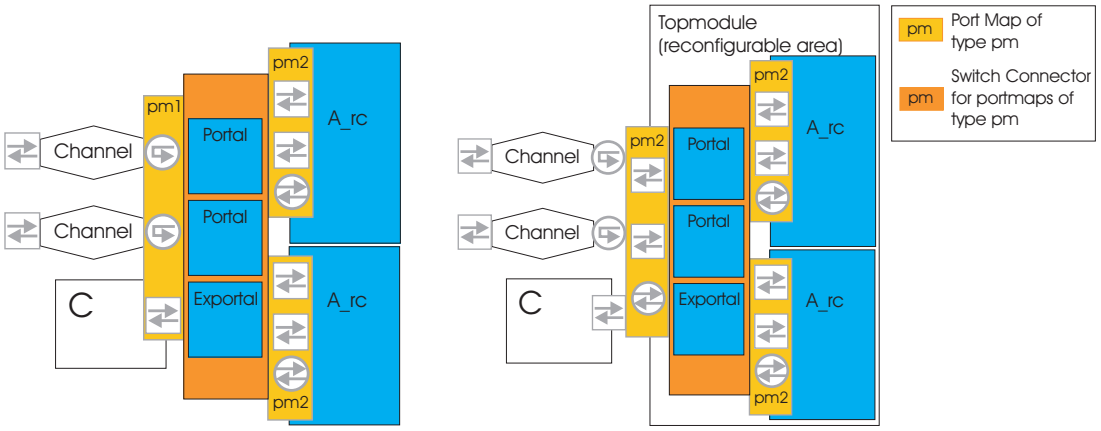


Figure 6.10: (a) Using port maps and switch connectors enables binding of complete modules to switches with a single binding statement. (b) `Topmodule` models the reconfigurable array explicitly and thus has the same ports and exports as the reconfigurable modules. Here only a single type of port map needs to be defined, to enable port-to-port and export-to-export binding.

Last but not least it is still possible to bind the modules ports (etc.) directly without using its port maps.

Figure 6.10(a) illustrates the use of port maps and switch connectors as it was previously discussed. In Figure 6.10(b) a more practical type of application is depicted. `Topmodule` models the reconfigurable array explicitly and thus has the same ports and exports as the reconfigurable modules. Here only a single type of port map needs to be defined, to enable port-to-port and export-to-export binding (compare Figure 5.6 (page 33) and Figure 5.9 (page 33)). A source code example of port map and switch connector utilisation in the CollisionChip simulation is presented in Sec. 9.4.1.

6.3 ReChannel Simulation Semantics

The syntax and functionality of portals and reconfigurable modules as introduced so far supports a highly flexible methodology for designing reconfigurable architectures on all levels of abstraction.

Still, there are some problems to cope with in its implementation. ReChannel needs to interact with the kernel and to fit into the simulation semantic as described in Sec. 5.1.1 and defined in [34].

Therefore it is necessary to define when and under which circumstances dataflow may be switched from one module to another. This means defining when a module may be deactivated, in the first place.

Here it is important to analyse, if all types of accesses that can be initiated concur-

rently from inside a reconfigurable module are compatible with reconfiguration and with the exact time the reconfiguration takes place. Within the SystemC environment this is exceptionally difficult, since not only modules can be described using different abstractions, but also a single module's processes can be modelled using various abstractions. Even worse is, that a single process can contain both, blocking *and* non-blocking accesses, rendering it a multi abstraction level description.

There are three important constraints that need to be respected:

- A user designing on RT-level will usually expect everything that is initiated to be executed between two delta-cycles. Different behaviour of a single component might lead to unexpected behaviour of the complete architecture.
- In high-level descriptions mainly blocking accesses are used. Here it is necessary to respect, that a module may only be deactivated if it has no more pending accesses. Otherwise the access might be executed while the module should be inactive. This was already discussed in Sec. 6.1.2.
- A module that received a deactivation command needs to deactivate.

These apparently simple constraints are hard to satisfy concurrently. Consider Figure 6.12. Timescale a) shows a simulation sequence with immediate deactivation. As soon as no more accesses are pending the module is deactivated in the middle of a delta-cycle. Timescale b) illustrates that simply delaying the deactivation by waiting for an event that is delayed by one delta-cycle will also not align it with the delta-cycle changes. Therefore a delta synchronisation object (DSO) is proposed, that exploits the `update` scheme of SystemC by implementing a `sc_prim_channel`. The deactivation is now requested by a call to the DSO's `request_update()`-method and executed when its `update()`-method is called by the kernel during the update phase (compare Sec. 5.1.1). So the Figure 5.10 (Sec. 5.1.1, page 35) can be extended with reconfiguration as it is illustrated in Figure 6.11.

As illustrated in Figure 6.12c) introducing this delta synchronisation can lead to deactivation requests that are never executed, if pending accesses are respected. This occurs if between two consecutive blocking accesses no delta delay ever elapses. This (of course) can only occur in untimed environments and hence might be considered a minor problem, since modelling reconfiguration in plainly functional descriptions will usually be done by simpler schemes (e.g., simply call another member function). Still, it can (and will) cause problems in mixed abstraction environments. Here another interpretation of synchronisation filters and explicit reconfiguration modelling shows that this is a problem already solved. Any type of synchronisation will delay the access a certain amount of time until its synchronisation condition is satisfied. This may only be a single delta-cycle, but as timescale d) shows this suffices to enable the DSO to deactivate.

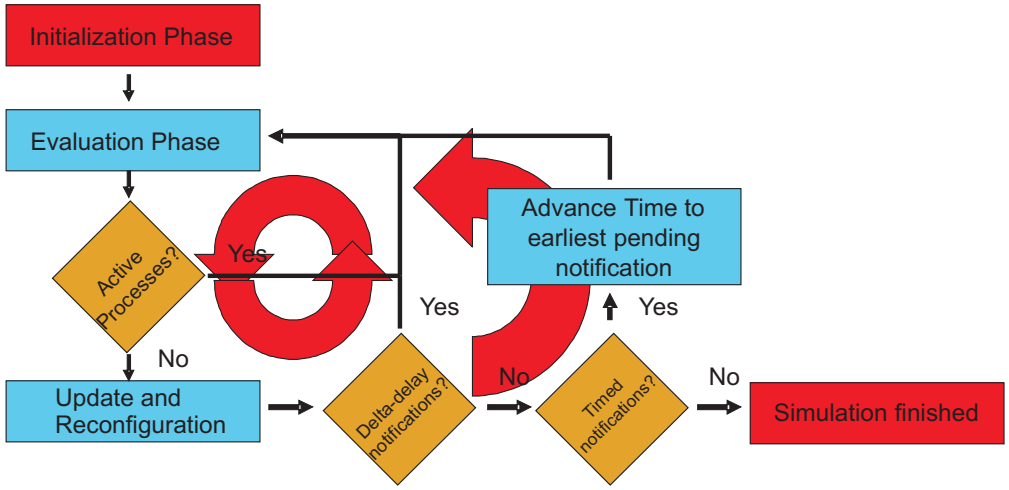


Figure 6.11: The introduction of the delta synchronisation object can be illustrated by augmenting SystemC's delta notification loop with reconfiguration.

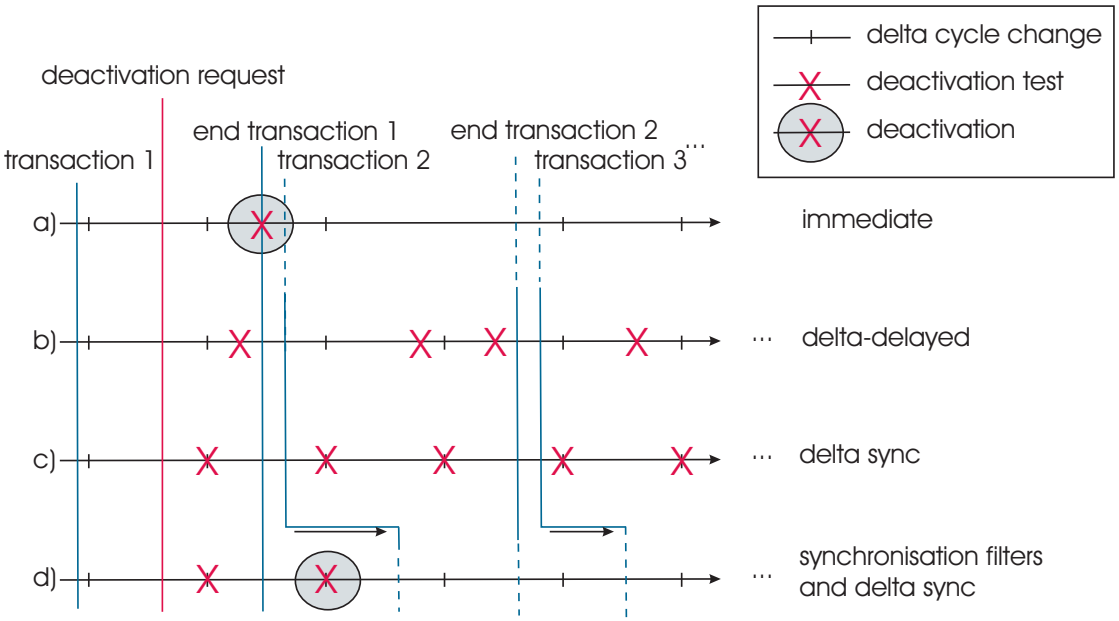


Figure 6.12: Timescale a) shows a simulation sequence with immediate deactivation. Timescale b) illustrates that simply delaying deactivation by waiting for an event that is delayed by one delta-cycle will not align it with the delta-cycle changes. Timescale c) shows that, if pending accesses are respected, delta synchronisation might lead to deactivation requests that are never executed. Timescale d) illustrates that synchronisation filter in conjunction with delta synchronisation objects (DSOs) enable a valid reconfiguration.

6.4 Integrating Reconfiguration into the Refinement Process

System design is a very demanding task, where many decisions have to be made depending on a huge variety of demands. Thus, it is not done in one major planning step and then implemented in its final realisation as e.g., it is mostly done in software development. Instead final assembly is preceded by a stepwise refinement of the system's features, where on each step different variations are explored with respect to their impact on system performance. For this SystemC features several refinement levels as discussed in Sec. 5.1.1.

To provide a really helpful tool an extension library with the objectives deduced in Sec. 2 needs to feature support for all these refinement stages as well. The following section is a discussion of how ReChannel can be integrated into a design on all levels of abstraction. Still, this is not enough. A complex tool such as this needs a tested methodology to be used effectively and efficiently. Therefore, the succeeding section also proposes a refinement methodology for reconfiguration aspects of a design that lends itself well to the SystemC refinement methodology. In Part IV of this work it is applied to the CollisionChip project presented in Sec. 8 to demonstrate its applicability.

6.4.1 Functional Level

As discussed in Sec. 2.3 introducing reconfiguration in early design stages is advisable. Since use of run-time reconfiguration often is prohibitively expensive, deciding if it will be integrated into the design in question needs to be done as early as possible in the design cycle.

If SystemC architecture refinement is done "by the book", a coarse approximation of the design's timing behaviour is generated at timed-functional level. With ReChannel this can be done easily for reconfiguration delays as well.

In a reconfigurable module's constructor the designer can set the time a reconfigurable module needs to be configured into or removed from the DRHW.

To control configuration on functional level it suffices to instantiate an object of type `rc_control` somewhere in the design and to register the reconfigurable modules with it as described in Sec. 6.1.3. Now the module's configuration state can be manipulated via function calls to `rc_control` whenever necessary.

If it turns out, that reconfiguration requests need to be processed very fast or that most requests are coming from modules to be implemented in hardware it might be necessary to implement a hardware reconfiguration controller as well.

Since functional level modelling is mainly used to determine the module structure, a possible next step is encapsulating the reconfiguration controlling into a custom configuration controller module instantiating `rc_control`.

This way requests for a certain module are modelled more explicitly, since modules requesting use of a reconfigurable module need to inform the reconfiguration controller of this. Using ReChannel, the user is completely free to choose a scheduling strategy and communication interface for the controller.

On functional levels mainly FIFO based communication is used. Therefore the main task related to reconfiguration will be implementing synchronisation filters defining transactions that must not be interrupted by reconfiguration. This will provide a good idea of the synchronisation requirements a custom configuration controller will have to respect on lower levels of abstraction.

6.4.2 Transactional Level

A slightly different approach is to use a hierarchical channel to encapsulate the controller (see Figure 6.13). This will be the canonical choice as soon as the refinement proceeds to Transaction-Level. Since requesting use of a reconfigurable module can be regarded as a request to a (sometimes very slow) bus, it is a very intuitive design. Maybe even more important is that it enables utilisation of standard techniques and tools for analysing TLM timing behaviour. It is vital to evaluate how reconfiguration influences system performance as early as possible in the design process to make an early decision in favour or against it. That ReChannel allows the use of standard tools makes this a convenient and intuitive undertaking. This is a direct result of ReChannel's satisfying the Integration objective (Sec. 2.4).

On Transaction Level usually custom-built channels are used. Thus, it is necessary to derive own switches (portals and/or exportals) for these channels' interfaces along with the necessary filters. For every custom-built channel's interface the switch derivation has to be done only once, and can then be reused.[4]

Nowadays it is more and more becoming good practise to use channels of the TLM library. This enables more general reuse of the derived switches. In future ReChannel versions support of the TLM library could be integrated as well.

Different to other approaches, that implement the complete reconfigurable area or the reconfigurable modules as buses the reconfigurable modules remain in their original scope and the surrounding design needs to be changed only very slightly. Obviously the topology of a static design remains basically unchanged if some parts are rendered reconfigurable. This is a major advantage over most related approaches (see Sec. 5.2).

6.4.3 Register Transfer Level

Evolving the design into a synthesisable hardware description can be done separately for each module out of a set of alternatively available reconfigurable modules and for the controller. The reconfigurable modules can be refined independently from their reconfiguration ability as demanded by the IP objective (see Sec. 2.2). The only exception here is the preservation of signal states during removal and reconfiguration the designer has to care for.

The synthesis objective (see Sec. 2.5) demands synthesisability without special tool support for ReChannel primitives, but using standard tools and techniques. Refin-

[4] Instead of implementing channels and switches separately, it is also possible to provide channels that themselves contain switches. This was not tested yet, since it reduces reusability of the switches and lows the channel down even in static designs. Still, in some cases it might be beneficial.

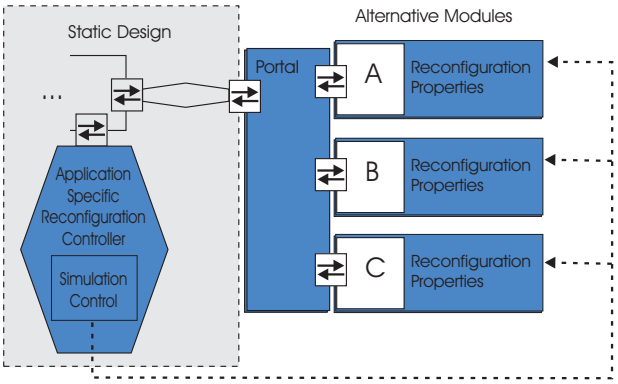


Figure 6.13: An overview of a reconfigurable design on Transaction-Level. The reconfiguration controller is modelled as a hierarchical channel. The reconfigurable modules remain in their scope.

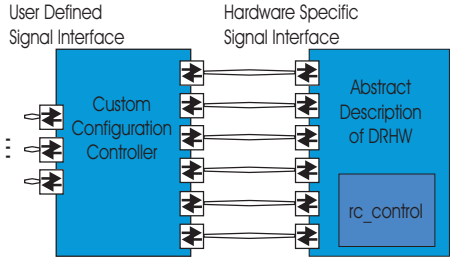


Figure 6.14: The application specific part of the controller is refined to pinaccurat RTL. Therefore a placeholder module for the reconfiguration behaviour of the underlying hardware is necessary. This can be built easily using the ReChannel simulation control `rc_control`.

ing the controller to a synthesisable description can be done in two stages. Firstly, the hierarchical channel is refined into a module with a pin accurate interface by standard techniques (e.g., adaptor insertion followed by adaptor inlining). Secondly, by encapsulating the properties of the DRHW in use into another module. The actual controller still contains the scheduling mechanism and communication with the remainder of the design. It can now be refined to synthesisability. The second module's interface consists of the program and reconfiguration pins that need to be addressed from inside the DRHW to initiate and control the reconfiguration. It will not be synthesised but serves as a placeholder for the real DRHW and the DRHE for simulation purposes. Hence it needs to behave like the DRHE, i.e., `rc_control` is instantiated here and its reconfiguration control functions are invoked, depending on the control signals triggered by the controller (see Figure 6.14).

This placeholder module can be reused as is for all designs using the same underlying DRHE.

The reconfigurable modules can be refined to synthesisability individually. Special synchronisation (e.g., handshaking) can now be investigated and respected within the environmental design. This will in general be necessary, since especially on RT-Level delays caused by reconfiguration need to be taken into account in protocol communication with the reconfigurable module (i.e., reset or start signals need to wait until the design is active, otherwise they will be ignored). Here it is very helpful, that ReChannel behaves like the real DRHW would. This way erroneous protocols can be identified in simulation already, instead of complicated on-chip debugging using logic-analysers, oscilloscopes or on-chip-scopes. After refinement all reconfigurable modules and the static part can be synthesised individually. Switches need to be replaced by bus macros (or according techniques of the technology in use). Further refinement (e.g., applying placement and timing constraints) now can proceed beyond the scope of SystemC and ReChannel.

Part III

A Dedicated 3D Collision Detection FPGA Architecture

7 Related Work

To proof the ReChannel approach and the methodology that goes along with it to be applicable and productive a viable case study is needed (compare Sec. 2.7). As was discussed in Sec. 3 in depth the development of a reconfigurable collision detection acceleration hardware satisfies all demands to such a case study. This Part of the thesis presents the preliminary development of a collision detection accelerator, which can then be extended with reconfigurable primitive tests. This extension will precede along the lines of the refinement methodology proposed in the previous Part II and will be presented in the subsequent Part IV.

One of the demands to the case study is the novelty of the architecture under construction. To proof this novelty an overview of the general field of collision detection is briefly discussed in Sec. 7.1–7.5 followed by the review of related approaches in hardware acceleration in Sec. 7.6 and Sec. 7.7. Sec. 8 will then propose the collision detection accelerator CollisionChip. It details the implemented algorithms and acceleration techniques and provides performance comparison with a state-of-the-art software implementation obtained in simulation.

To proof the design to be fit for synthesis and even hardware implementation it is synthesised and verified on-chip. Detailed analysis of the hardware consumption is also provided as an extra.

7.1 Collision Detection Overview

In CG applications objects are usually represented as sets of certain primitives (e.g., triangles, quadrangles, points, NURBS, etc.). Of theses variants triangle sets are probably the most common type. Testing two objects for intersection / collision can now naively be done by testing all primitives of one object against all primitives of the other one. This is in $O(n^2)$ with n denoting the number of primitives. Since collision detection is a fairly basic operation in many applications this is prohibitively expensive. There even exists empirical proof that collision detection is the major bottleneck in physically-based simulation. As reported in [56], 95% of calculation time of physically-based simulations is spent on collision detection. Therefore various algorithms have been proposed to accelerate these tests. These algorithms can be categorised by their applicability to three object classes of different generality: convex, rigid and deformable.

For convex objects, various algorithms were proposed that exploit convexity [16, 19, 32, 43]. Algorithms for this object-class are extremely fast, but due to their assumption of convexity very limited in their applicability.

For rigid bodies bounding-volume-hierarchies (BVH) proved to be most efficient. Since they are of major importance in the following they will be discussed in depth in the next

Sec. 7.2.

Due to their generality the class of deformable objects is the most interesting class of objects. Since an object's possible self-collision has to be taken into account as well, it is the most difficult class, too. Various different approaches have been proposed [25, 39] to check objects of this class for intersection. In some cases (e.g., small maximum deformation per timestep) bounding-volume-hierarchies (BVH) can be applied as well [33, 36, 39, 42]. The commercially available PhysX physics processor probably provides hardware acceleration for deformable objects. But to the author's knowledge there have not been any scientific publications concerning hardware implementability of any collision detection algorithms for the class of deformable objects yet. Despite being a very interesting subject investigating in hardware suitable algorithms for this class would tremendously exceed the scope of this thesis. Hence hierarchical collision detection of rigid bodies is tackled in the following.

7.2 Hierarchical Collision Detection

To avoid checking every primitive of a rigid object against all primitives of another rigid object when checking for collision, hierarchical collision detection (HCD) is used in the following. HCD is a divide-and-conquer technique that yields a linear or even logarithmic average runtime in realistic scenarios [38,75]. HCD uses hierarchies of bounding-volumes (BVs). BVs are used to group an objects primitive's together. A BVH is a tree with inner nodes corresponding to bounding-volumes containing all leafs of its subtree. Leafs correspond to primitives. Usually BVs of higher levels bound the BVs in their subtree as well.

Checking two objects for intersection can now be done by traversing both BVHs and testing their bounding volumes against each other. As shown in Figure 7.1 a test tree results. An intersection test of two objects can now be interpreted as traversal of the test tree. If two BVs do not intersect, the according subtree of the test tree does not need to be traversed any further (e.g., if in Figure 7.1 C and 2 do not intersect, this renders tests F4-G5 unnecessary, since the bounded geometry does obviously not intersect). Therefore one needs to be sure that no bounding volume separation is accidentally reported (false negative). False reports of bounding volume collision (false positive) do not lead to wrong overall results, as long as the primitive intersection test works correctly. This type of false report will only lead to more bounding volume tests in the succeeding calculation. As will be discussed in Sec. 7.4 and Sec. 8.1.2 this can be exploited in multiple ways.

Since two objects will usually intersect only locally in a very small number of primitives, HCD yields a significant speed-up in the average case. In practical cases, the complexity is in $O(\log n)$ (n = number of primitives) because only a small diagonal "slice" of constant width down the BVH needs to be visited.

In the worst-case depth-first search results in the testing sequence
A1-B2-D4 D5 E4 E5-**B3**-E6 E7 D6 D7-**C2**-G4 G5 F4 F5-**C3**-F6 F7 G6 G7.

If reused in the directly subsequent test not all nodes have to be fetched from memory. The fetching order for depth-first search then is as follows:

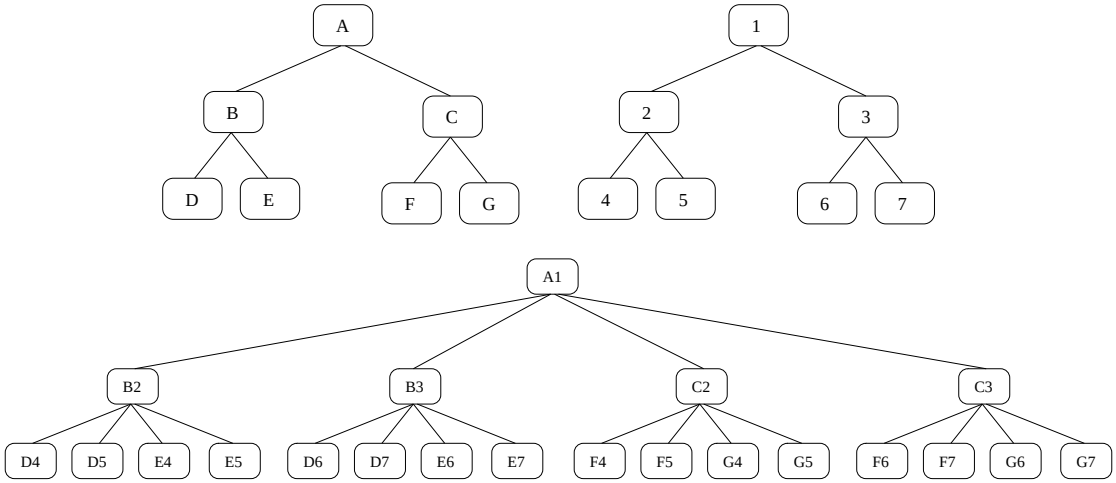


Figure 7.1: Bounding-volume hierarchies of two objects and the resulting test tree.

A,1,B,2,D,4,5,E,4,5,B,3,E,6,7,D,6,7,...

In the identical scenario breadth-first search results in the testing sequence
A1-B2 B3 C2 C3-D4 D5 E4 E5-D6 D7 E6 E7-F4 F5 G4 G5-F6 F7 G6 G7.
Here some further loading can be saved due to consecutive usage of B and C.

This can be optimised so that between testing two nodes of same depth only one node needs to be fetched from memory:
A1-B2 B3 C3 C2-D4 D5 E5 E4-D6 D7 E7 E6-F4 F5 G5 G4-F6 F7 G7 G6.
This will be called "optimised brotherhood" in the following and results in the following loading order:
A,1,B,2,3,C,2,D,4,5,E,4,D,6,7,E,6,F,4,5,G,4,F,6,7,G,6.
Software benchmarking of this can be found in [82].

Many different BVs, like Spheres, axes-aligned bounding-boxes (AABBs), oriented bounding-boxes (OBBs), etc. have been proposed. In the following k-DOPs will be used, because they were proven to yield very fast collision queries by extensive benchmarking in software [82]. Due to their regular structure they are likely to perform well in hardware, too. Therefore they will be introduced in the next Sec. 7.3. A first approach towards a hardware accelerated collision detection using k-DOPs was proposed in [86] and refined in [8, 59]. This version will be briefly recapitulated in Sec. 7.6.

7.3 k-DOPs

k-DOPs are defined over a fixed orientation matrix $\mathbf{D} = \left(\mathbf{D}_1, \ldots, \mathbf{D}_{k/2}, \mathbf{D}_{k/2+1}, \ldots, \mathbf{D}_k\right)$ of vectors in $\mathbb{R}^3$. Each vector $\mathbf{D}_i$ is antiparallel (pointing into the exact oposite direction) to $\mathbf{D}_{i+k/2}$.

An individual k-DOP is defined by k distances d_i, one along each vector $\mathbf{D}_i$, thus defining a half-space. These DOP coefficients $(d_1, \ldots, d_k)$ are the distances of the asso-

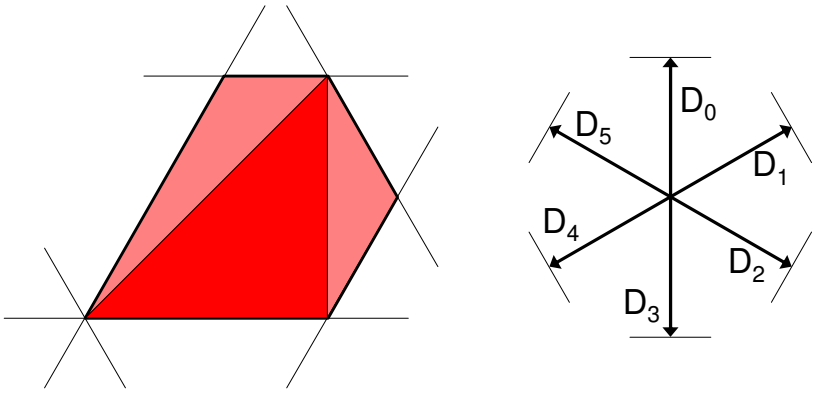


Figure 7.2: A single triangle enclosed by a 2-dimensional 6-DOP with its fixed set of orientations $\mathbf{D}_1, \ldots, \mathbf{D}_6$. Each vector $\mathbf{D}_i$ is antiparallel to $\mathbf{D}_{i+k/2}$.

ciated halfspaces to the origin. Note that the origin is neither necessarily the centre of the DOP nor even contained in it.

In the following wrap-around indexing of the DOP coefficients is used to improve readability of the equations. Thus, $d_{i+k/2}$ actually denotes $d_{((i+k/2) \mod k)}$.

The intersection of the halfspaces forms the BV:

$$\text{DOP} = \bigcap_{i=1,\ldots,k} H_i, \quad H_i : \mathbf{D}_i\mathbf{x} - d_i \leq 0 \tag{7.1}$$

Each of the $k/2$ pairs of DOP coefficients $(d_i, d_{i+k/2})$ form a so-called *slab*.

The orientation matrix D, consisting of all the vectors $\mathbf{D}_i$, is fixed and equal for all objects. This allows a very memory-efficient representation of k-DOPs: only the k coefficients d_i need to be stored. Figure 7.2 shows an example of a 2-dimensional 6-DOP enclosing a single triangle.

7.4 Separating Axis Test - SAT

In [27] it is shown that two convex polytopes are disjoint if and only if there exists a separating axis orthogonal to a face of either polytope or orthogonal to an edge of each polytope (Separating Axis Theorem).

To perform the test, both polytopes must be projected onto each of the candidate separating axes. For each axis, a pair of intervals on that axis results. If and only if one of these pairs is disjoint, then the polytopes are disjoint.

If only a subset of these axes is tested, false positives might occur, i.e., the polytopes are disjoint while the (incomplete) test reports an intersection. The complete SAT is always correct.

7.5 Primitives

In computer graphics multiple primitives are used to represent 3-dimensional objects. These primitives can be 3-dimensional themselves (e.g., voxels or polyhedra), 2-dimensional (e.g., triangles, b-splines or NURBs) or even simple points. Each of these primitives has its own advantages and disadvantages. Most commonly used are triangles, since they can be handled easily, while still providing good approximations to the object's surface. Hence standard computer graphics hardware is usually tailored to their usage. Because of the easy handling and their wide spread utilisation triangles are used in this thesis as primitives primarily. Still, as discussed in Sec. 3 collision detection was chosen as a case study, since reconfiguration of the primitive intersection subsystem is promising to provide real benefit. Thus a second primitive type is necessary. Since this work is not CG centred implementing most complicated intersection tests for complex primitives like NURBs or b-splines, was not seriously considered. Instead a most simple primitive needs to be chosen to avoid a change in focus. Therefore quadrangles will be utilised in the following. They can be regarded as a mere generalisation of triangles, while being still practically relevant. Their usage is preferred over other primitives, e.g., in virtual crash test simulations, since they enable providing a regular grid and thus simplify calculation of force propagation. As will be discussed in Sec. 8.7.1 most triangle intersection tests can be generalised to quadrangles, and thus little additional development overhead is imposed.

If used for approximating the surface of a 3-dimensional object, triangles and quadrangles are 2D objects embedded in a 3-dimensional space. Thus their apparent simplicity comes with a drawback. Different to native 3D objects two polygons can be coplanar, i.e., span a 2-dimensional subspace, instead of the 3D space they actually live in. As will be discussed in Sec. 8.7.1 in more depth, this causes multiple intersection algorithms problems. Non-coplanar, non-degenerated objects are said to be *in general position*.

7.6 An ASIC Targeted Approach

A first approach towards a collision detection hardware was proposed in [86] and refined in [8, 59]. In this last work the author of this Dissertation already participated. Since full fathership of this approach cannot be claimed it is presented in the related work section.

7.6.1 Bounding Volume Test

Each object O has its own reference frame (RF) $F(O)$ that describes its rotation R_O and translation T_O with respect to world coordinates. When an object moves, only R_O and T_O have to be updated. So checking two DOPs for intersection requires transformation of one of them into the RF of the other.

Let O and Q be two objects. Let $DOP_Q(Q)$ denote the minimum DOP with coefficients $(b_1, \cdots, b_k)$ which bounds Q with respect to the orientation matrix $\mathbf{D}$ in Q's own reference frame $F(Q)$.

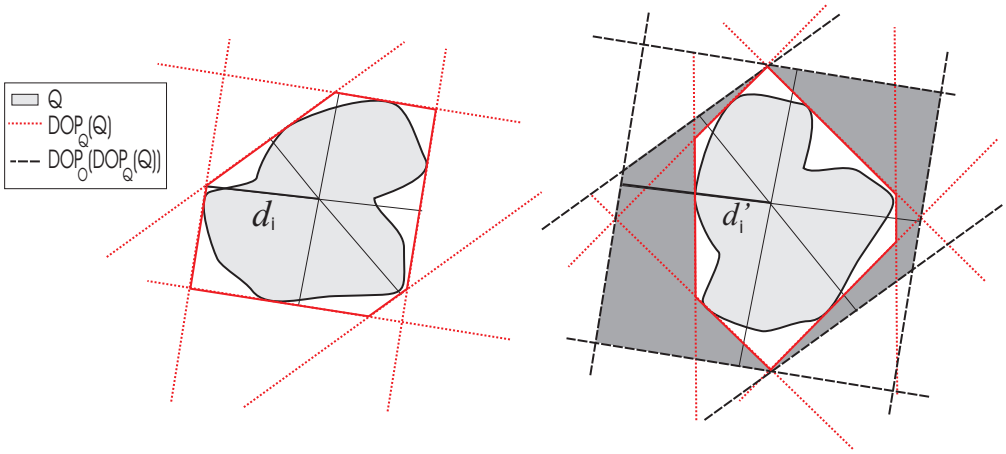


Figure 7.3: The described DOP overlap test gains its speed by transforming DOP(Q) into O's reference frame $F(O)$. The tightness loss is shown in dark grey. Obviously, each d' is determined by exactly three original ds.

$$DOP_Q(Q) = \bigcap_{j=1,\ldots,k} H_j, \quad H_j : \mathbf{D}_j\mathbf{x} - b_j \leq 0 \tag{7.2}$$

Let $DOP_O(O)$ be denoted respectively with coefficients $(a_1, \cdots, a_k)$.

For intersection testing a DOP bounding Q within O's reference frame is used. Since calculating $DOP_O(Q)$ is expensive, $DOP_O(DOP_Q(Q))$ is calculated, which is the minimum DOP in $F(O)$ bounding $DOP_Q(Q)$. Naturally, this causes a certain loss of tightness to the underlying geometric structure, which of course has a negative impact on the runtime of the algorithm. In software this is overcompensated by the speed gained through the simplicity of the test [82].

Let $\mathbf{M}$ be the rotation and $\mathbf{o}$ the translation which transforms $F(Q)$ into $F(O)$. Then, we need to find distances b'_i which bound $\mathbf{M} \cdot DOP_Q(Q) + \mathbf{o}$ minimally.

Applying $\mathbf{M}$ and $\mathbf{o}$ to Equation 7.2 yields

$$h_j : \mathbf{d}_j\mathbf{x} - b_j + \mathbf{d}_j\mathbf{o} \leq 0, \quad \text{where } \mathbf{d}_j = \mathbf{D}_j\mathbf{M}^{-1} \tag{7.3}$$

Let $DOP_O(DOP_Q(Q))$ be the intersection of

$$H_i : \mathbf{D}_i\mathbf{x} - b'_i \leq 0, \quad i = 1, \ldots, k \tag{7.4}$$

Then, each b'_i corresponds to exactly one vertex of $DOP_Q(Q)$ and therefore to three b_j (see Figure 7.3). These correspondences are identical for all nodes in an object's DOP hierarchy. So they can be pre-calculated at start-up.

Let j_l, $1 \leq l \leq 3$, be the indexes corresponding to b_i. Then,

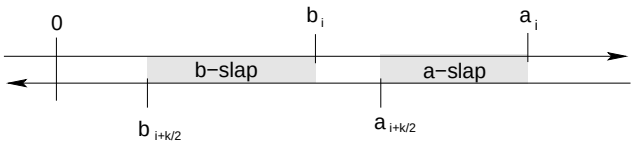


Figure 7.4: Two objects overlap, if and only if there is no axis on which their projections are non-intersecting. An axis consists of two anti-parallel normals of halfspaces.

$$\begin{pmatrix}\mathbf{d}_{j0}\\ \mathbf{d}_{j1}\\ \mathbf{d}_{j2}\end{pmatrix}\mathbf{x} - \begin{pmatrix}\mathbf{b}_{j0}\\ \mathbf{b}_{j1}\\ \mathbf{b}_{j2}\end{pmatrix} + \begin{pmatrix}\mathbf{d}_{j0}\\ \mathbf{d}_{j1}\\ \mathbf{d}_{j2}\end{pmatrix}\mathbf{o} = \mathbf{0} \tag{7.5}$$

$$\mathbf{D}_i\mathbf{x} - b'_i = 0 \tag{7.6}$$

Equating 7.5 and 7.6 yields

$$b'_i = \mathbf{D}_i\left(\mathbf{b}_j\mathbf{d}_j^{-1} - \mathbf{o}\right) \tag{7.7}$$

This can be reformulated to

$$b'_i = C_{ij_1}b_{j_1} + C_{ij_2}b_{j_2} + C_{ij_3}b_{j_3} + c_i \tag{7.8}$$

where C and c are chosen to be $C_{ij} := D_i d_j^{-1}$ and $c_i := D_i o$. Both, C and c, are equal for all nodes in an object's DOP hierarchy, thus can also be pre-calculated at start-up.

Checking $DOP_O(DOP_Q(Q))$ and $DOP_O(O)$ for intersection amounts to projecting them on the $k/2$ axes given by the D normals of the halfspaces. They overlap if and only if there is no axis on which they are non-intersecting. Since there are always two antiparallel normals, this needs to be taken into account (see Figure 7.4).

Putting it all together, the intersection test can be expressed as

$$\text{overlap} \Leftrightarrow \nexists i \in [1, \frac{k}{2}] : a_{i+\frac{k}{2}} > -b'_i \vee b'_{i+\frac{k}{2}} > -a_i \tag{7.9}$$

7.6.2 Triangle Intersection

Once the BVH traversal reaches two leaves, the enclosed primitives need to be tested for intersection. As primitives triangles are used exclusively. As discussed in Sec. 7.5 this is the canonical choice.

The triangle transformation algorithm from [85] was used, which is based on ideas from [3]. Let T_A and T_B be triangles with vertices $V^1, V^2, V^3 \in \mathbb{R}^3$, and $W^1W^2, W^3 \in \mathbb{R}^3$ respectively. They are assumed to be in general position to avoid the coplanarity problems discussed in Sec. 7.5. A rotation matrix $\mathbf{M}_A$ and translation V^0 are precomputed

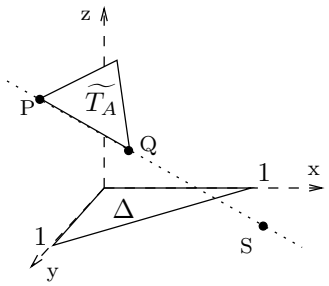


Figure 7.5: In [8, 59] triangle intersection testing is simplified by applying a precomputed affine transformation to both triangles T_B and T_A, so that T_B is mapped onto the unit triangle Δ.

so that the transformation $V \mapsto \mathbf{M}_A \cdot (V - V^0)$ maps T_A onto the unit triangle Δ with vertices $(0,0,0), (1,0,0), (0,1,0)$. The same is done for $\mathbf{M}_B$ and W^0 that map T_B into Δ respectively.

To test T_A and T_B for intersection, T_A's vertices V^i are transformed into $\widetilde{V^i}$ given by

$$\widetilde{V^i} = \mathbf{M}_B(V^i - W^0), \tag{7.10}$$

Let $\widetilde{T_A}$ be the resulting triangle. Figure 7.5 illustrates this.

Since this transformation maps T_B into the unit triangle it suffices to test $\widetilde{T_A}$ for intersection against Δ.

Consider an edge $\overline{PQ}$ of triangle $\widetilde{T_A}$. If both P_z and $Q_z \geq 0$ or both ≤ 0, then the edge cannot not intersect Δ and the next edge is processed. Otherwise, the intersection S of the line through $\overline{PQ}$ with the $xy-$plane needs to be computed:

$$S = P - \mathbf{r}\frac{P_z}{r_z}, \quad \text{with } \mathbf{r} = Q - P. \tag{7.11}$$

Since $r_z \neq 0$ and $S_z = 0$, only $S_x = P_x - \mathbf{r}_x \frac{P_z}{r_z}$, and S_y respectively need to be computed.

Overall, the following criterion results:

$$S_x < 0 \tag{7.12}$$

$$S_y < 0 \tag{7.13}$$

$$S_x + S_y > 1 \tag{7.14}$$

If condition 7.12, 7.13, or 7.14 holds, then edge $\overline{PQ}$ does not intersect Δ.

Substituting $\mathbf{r}$ in Eq. 7.11 and solving it results in a form of the criterion which does not contain divisions any longer. Therefore it can be calculated faster and with less hardware. Let

$$\begin{aligned} a &= P_x Q_z - Q_x P_z \\ b &= P_y Q_z - Q_y P_z \end{aligned} \tag{7.15}$$

If $r_z > 0$ the criterion now is:

$$a < 0 \tag{7.16}$$
$$b < 0 \tag{7.17}$$
$$(a) + (b) > r_z \tag{7.18}$$

To cover $r_z < 0$,

$$\text{sign}(r_z) \cdot a < 0 \tag{7.19}$$
$$\text{sign}(r_z) \cdot b < 0 \tag{7.20}$$
$$\text{sign}(r_z) \cdot (a + b) > \text{sign}(r_z) \cdot r_z \tag{7.21}$$

is used.

If no intersection is found the test is performed vice versa. These two tests could also be parallelised. This is avoided in order to save hardware, since this triangle test consumes a lot of hardware resources. Instead [8, 59] exploit that in nearly all practical cases far less than half of the leaves will be tested. This is due to the fact that in real world applications usually two objects will intersect only in a very small area that should already be singled out by the bounding-volume test. Hence an input FIFO is used that decouples bounding-volume and triangle intersection test to buffer triangle test tasks generated in quick succession.

7.6.3 The Architecture

The described architecture is targeted to a NEC UX5 CB-130 ASIC in 0.095μm-copper-technology. With a maximum of 61 gates in a row it can establish up to 800 MHz clock rate. Furthermore, the use of DDR2 memory modules is assumed. As BVs 24-DOPs are used, which are projected onto the 12 axes (= 24 pairwise antiparallel normals of its halfspaces) of one the DOPs. This was chosen because in extensive software benchmarking it has proved to be a good compromise of tightness and effort.

DOP Architecture

The DOP intersection test is a combination of criterion 7.9 with 7.8.

The choice of the correct correspondence is implemented within a hypermultiplexer. This amounts to the function

$$\widetilde{b'_i} = b_k C_{i,0} + b_m C_{i,1} + b_n C_{i,2} + c_i + a_{i+\frac{k}{2}} \tag{7.22}$$

This can be implemented as a three-staged macro-pipeline, called DOTADD unit shown in Figure 7.6. Single precision floating-point numbers are used to represent DOP coefficients and matrix entries.

The macro-pipeline stages were refined furthermore resulting in a pipeline of 15 stages and therefore an initialisation delay of 15 clock cycles.

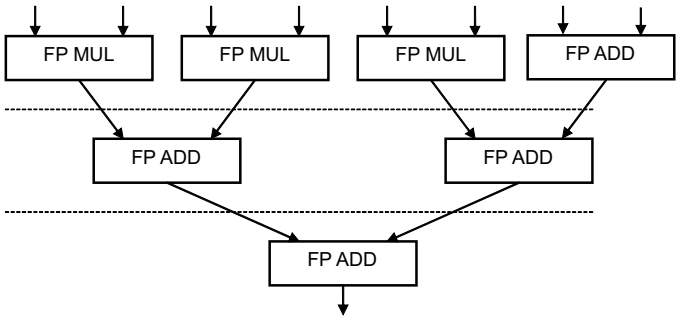


Figure 7.6: Three-staged macro-pipeline called DOTADD unit.

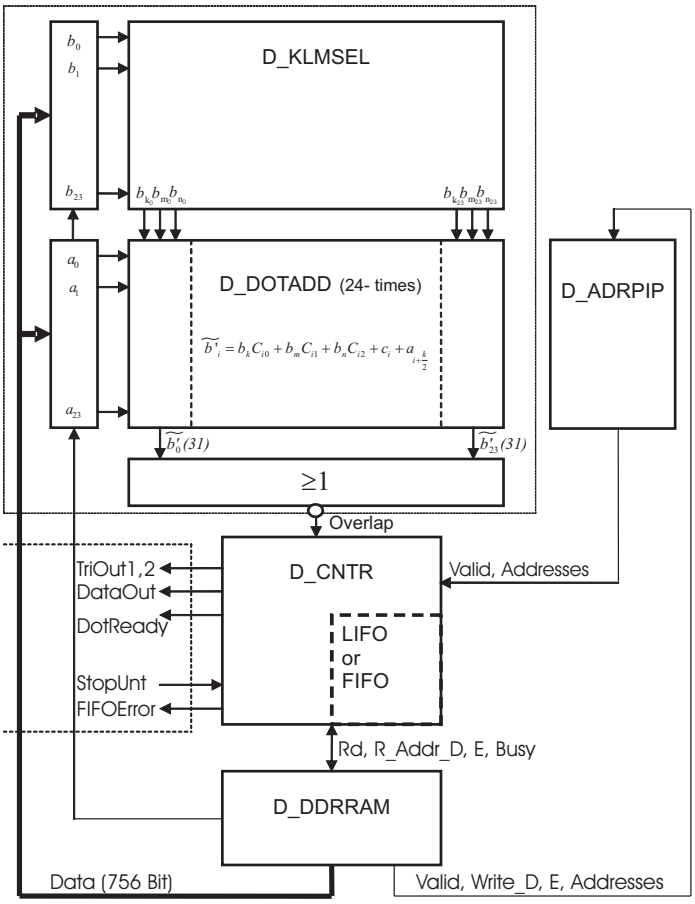


Figure 7.7: Architecture for DOP intersection testing presented in [8, 59].

24 DOTADD units in parallel are used, and their results are NOR-reduced to check the criterion. To fill the pipeline a 756-bit wide bus from the DDR-RAM is utilised. A hypermultiplexer (D_KLMSEL) routes the correct inputs of the DOP to the DOTADD units, which will then be transformed into the reference frame of the other DOP (see Figure 7.7).

Control

A D_CNTR unit controls which DOP pair is processed next. Here two different traversal schemes where compared (see Subsection *Performance Evaluation*): storing the sequence of BVs that need to be checked in a FIFO (breadth-first search) and storing them in a LIFO (depth-first search). Both times optimised brotherhood was used (compare Sec. 7.2).

Note that using a pipeline for intersection testing, inevitably causes a certain breadth in search. This is due to the fact that pipelined calculation is functionally equivalent to having a FIFO of the length of the pipeline prior to doing the whole calculation at once. Hence, traversal using pipelined execution of the intersection test will proceed along several paths down the tree in a depth-first manner.

Triangle Architecture

Using homogeneous coordinates, the affine transformations needed for the triangle intersection test discussed in Sec. 7.6.2 can be represented as 3×4 matrices.

The T_CHECK unit that performs the intersection test gets as input triangle $V_A^i = [x_i y_i z_i 1], 1 \leq i \leq 2$, the precomputed matrix

$$M_B := [m_{ij}] := \begin{bmatrix} m_{11} & m_{12} & m_{13} & m_{14} \\ m_{21} & m_{22} & m_{23} & m_{24} \\ m_{31} & m_{32} & m_{33} & m_{34} \end{bmatrix}, \tag{7.23}$$

and the matrix $M_{AB} := \left[b_{ij}\right]$ that transforms O's reference frame into Q's and V_B, M_A, and M_{BA} respectively.

Calculating $\widetilde{V_A^i} = M_B \times M_{AB} \times V_A^i$ is done in the first two of 5 macro pipeline stages. These are divided into micro stages in order to allow high clock frequencies.

1st macro stage: The calculation of $M_b = \left[b'_{ij}\right] = M_B \times M_{AB}$ is split into three sub-stages (marked with different brackets):

$$b'_{ij} = \{[(b_{i1}m_{1j}) + (b_{i2}m_{2j})] + (b_{i3}m_{3j})\} \tag{7.24}$$

$$b'_{i4} = \{[(b_{i1}m_{14}) + (b_{i2}m_{24})] + [(b_{i3}m_{34}) + (b_{i4})]\} \tag{7.25}$$

$$i, j = 1, \ldots, 3$$

The forth row of the resulting matrix is always [0001], since no perspective transformation is necessary here. These sub-stages are further divided into micro-stages to gain maximum clock frequency. With this refinement, the first macro stage consumes 36

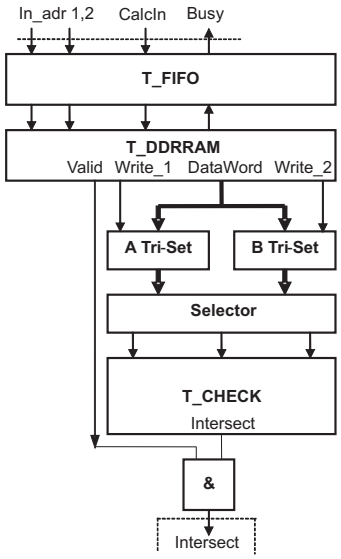


Figure 7.8: Architecture for Triangle Intersection

multiplication and 24 floating point addition units and takes 15 clock cycles delay to produce the first result.

2.nd macro stage: Calculating $\widetilde{V_A^i} = M_b \times V_A^i$ works very similarly. The resulting sub-stages are:

$$\widetilde{V^i} = \{[(b'_{i1}V_x^i) + (b'_{i2}V_y^i)] + [(b'_{i3}V_z^i) + (b'_{i4})]\} \tag{7.26}$$

Dividing this into micro-stages yields 27 multiplications, 27 additions, and another 15 clock cycles delay.

3rd macro stage: Before checking the edges of $\widetilde{T_A}$ for intersection with the unit triangle, a and b need to be calculated according to Eq. 7.15 for all three edges.

Therefore, P_xQ_z, Q_xP_z, P_yQ_z, and Q_yP_z are calculated first. Concurrently r_z is calculated. This takes 12 multiplications, 3 additions, and 8 clock cycles. Calculating a and b from these terms takes another 6 additions and 4 clock cycles.

4th macro stage: For all three edges $a + b - r_z$ are calculated. After division into micro-stages this consumes 6 additions and 8 cycles.

5th macro stage: Here, signs of a, b, and $a+b-r_z$ are checked for all edges. This needs one clock cycle.

Overall pipeline: Putting all stages together, a pipeline with 52 clock cycles delay results.

To fill the pipeline with data, triangle addresses are buffered, looked up in the DDR2-SDRAM which contains vertex data and the transformation matrices, and divide the data into two sets (because all edges of T_A have to be checked against T_B and vice versa). The whole TRI_UNIT is presented in Figure 7.8.

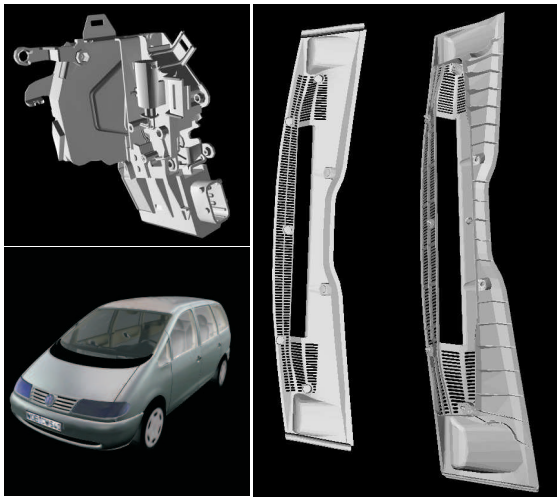


Figure 7.9: For benchmarking and testing, a number of different test objects with several polygon complexities were used.

Performance Evaluation

For benchmarking the architecture it was described in VHDL and simulated. As benchmarking application three different objects are used each of which in several polygon complexities (see Figure 7.9). For each of them, two copies are placed at different distances from each other and with different rotations. For each distance, the average collision detection query time is determined.

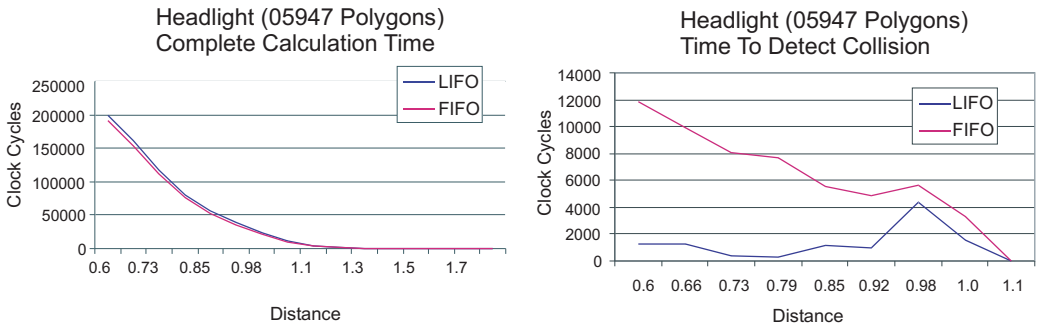
As shown in Figure 7.10(a) finding all intersecting primitives takes equally long using LIFO or FIFO control.

Since using a LIFO corresponds to depth-first search on the collision tree, finding the first collision is usually much faster than when using a FIFO. The simulation results verify this (see Figure 7.10(b)).

Another disadvantage of using a breadth-first search is the size needed for the memory structure. In the worst case, when all nodes need to be checked for intersection they all need to be stored in the FIFO before any leaves are checked and the queue size reduces.

With strict depth-first traversal, the LIFO would need to be only as large as the depth of the BVH. However, as explained in Subsection *Control*, the traversal scheme used is not strictly depth-first. Still, memory usage of the LIFO in the design seems to behave just as well (see Figure 7.11).

As Figure 7.12 shows the architecture is up to 1000 times faster than an according state-of-the art software implementation in determining all intersecting triangles of two objects. The software times were obtained on a 1GHz Pentium 3.



(a) Finding all intersecting primitives takes equally long.

(b) In finding the first intersecting pair of primitives LIFO control is superior to FIFO control.

Figure 7.10: Performance comparison of LIFO and FIFO controlled intersection testing.

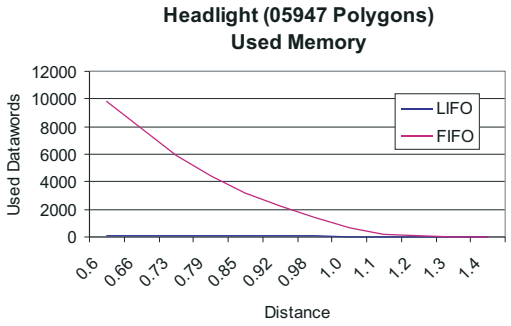


Figure 7.11: Although the architecture implements a traversal scheme that is no depth-first search in the strictest sense, this LIFO/pipeline combination still uses far less memory then breadth-first search using a FIFO.

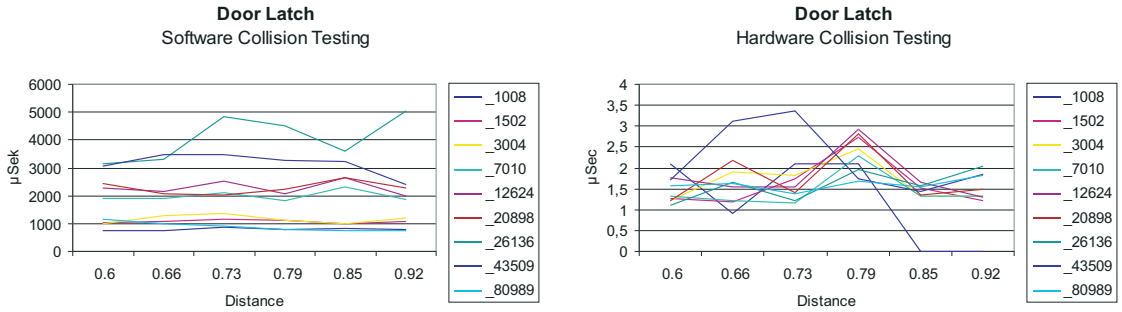


Figure 7.12: The simulated ASIC targeted collision detection architecture is up to 1000 times faster in determining all intersecting triangles of two objects than the software implementation.

7.6.4 Conclusion

[8, 59] showed that hardware acceleration is an effective way to speed up hierarchical collision detection. Using a high-end ASIC platform and a custom-made high-end board layout with extreme routing powers in conjunction with newest DDR2-SDRAM technology a speed-up of factor 1000 compared to a software implementation can be obtained.

Although nothing impossible is assumed, these assumptions are somewhat unrealistic. Especially requesting a fixed 756-bit wide bus from memory to collision detection unit that is capable of delivering one data-word each half-cycle makes the design inflexible and tremendously expensive. Not to mention development time necessary for a board layout of this complexity. Reducing this width would inevitably lead to a tremendous loss in performance since throughput of the memory interface is the main bottleneck in hardware collision detection as will be shown in Sec. 8.5.

The DOP transformation test used in [8, 59] can be interpreted as a separating axes test that always projects two 24-DOPs onto the 12 axes (= 24 normals of halfspaces, since they are pairwise anti-parallel) of one of both DOPs concurrently. There are several reasons why this is both, inflexible and a tremendous waste of resources. Firstly, the circuit size is linearly depending on the DOP dimension. Or putting it the other way around, the test's complexity cannot be varied depending on the target architecture in use. Secondly, it requires the memory bandwidth to be big enough to load at least one k-DOP each clock cycle. If it is not, the pipeline needs to be stalled. Thirdly, if any of the projections results in non-intersecting intervals it is unnecessary to test any more intervals. This is a very likely case, since the axes used are (of course) not linear independent. Due to the concurrent calculation this cannot be exploited.

Furthermore the design assumes that if 12 axes are a good compromise in software this necessarily applies for hardware as well. That this is not the case, will be shown in Sec. 8.3. Here things are more complicated since various factors can and need to be respected to maximise performance.

Additionally, the presented hardware design is very expensive with respect to resource consumption since it was optimised for speed only. After place and route the pipeline utilises a total of over 4 million gates targeting the named architecture. It is unfittable into any currently existing programmable hardware device even if the single precision floating-point representation is naïvely exchanged against 32Bit fixed-point numbers (which would definitely lead to false negatives, see discussion in Sec. 8.1.2). In this calculation pipeline controlling, triangle intersection, DDR controlling and host-to-accelerator communication was not even respected.

Neither DDR controlling, nor host-to-accelerator communication was respected in any of the simulations, but can be expected to slow it down considerably. Moreover, since the synthesis results could not be tested on a FPGA and were not realised in an ASIC on a custom-built board, final proof of the realisability, effectiveness and efficiency are missing.

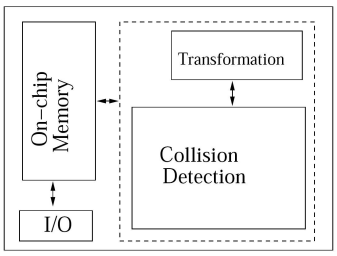


Figure 7.13: Rough division of the FPGA acclerated Möller intersection test of [4].

Taking all this into account the calculated speed-up can only be regarded as a best-case estimation. Still, it gives proof that collision detection is an area of application that can greatly benefit from hardware acceleration. Also using hierarchical collision detection with k-DOPs and triangles is proven to be promising.

7.7 FPGA-Accelerated Möller Triangle-Intersection Test

An alternative triangle intersection algorithm was implemented in hardware by [4, 5]. The authors use the well known algorithm from [44], which is commonly referenced as "Möller test", to check two triangles for intersection. It is also known as common-line interval approach and is detailed in Sec. 8.7.1.

In [4, 5] it is assumed that one triangle belongs to a moving object and the other is part of a completely static environment. A Virtex-4 XC4VLX200 was used as target architecture. As shown in Figure 7.13 the design is roughly separated into four parts: I/O, on-chip memory, a transformation unit and the collision detection unit itself.

7.7.1 Preprocessing, I/O and Memory Interface

Before start of calculation all coefficients of the triangle's vertices are normalised and rounded to 10 bit fixed-point numbers. This data is then sent via a RS-232 serial connection to the I/O unit of the architecture. There it is saved into the FPGA's Block-RAMs. The Block-RAM modules are organised in a way, that two triangles can be read in a single clock-cycle.

7.7.2 The Architecture

Since it is assumed that only pairs with one moving and one immobile triangle are checked only some triangles need to be transformed before the intersection test itself is started.

The Möller test considers three different cases:

- **half-space** One triangle's vertices all lie in the same half-space induced by the other triangle. It can be deduced that they do not intersect.

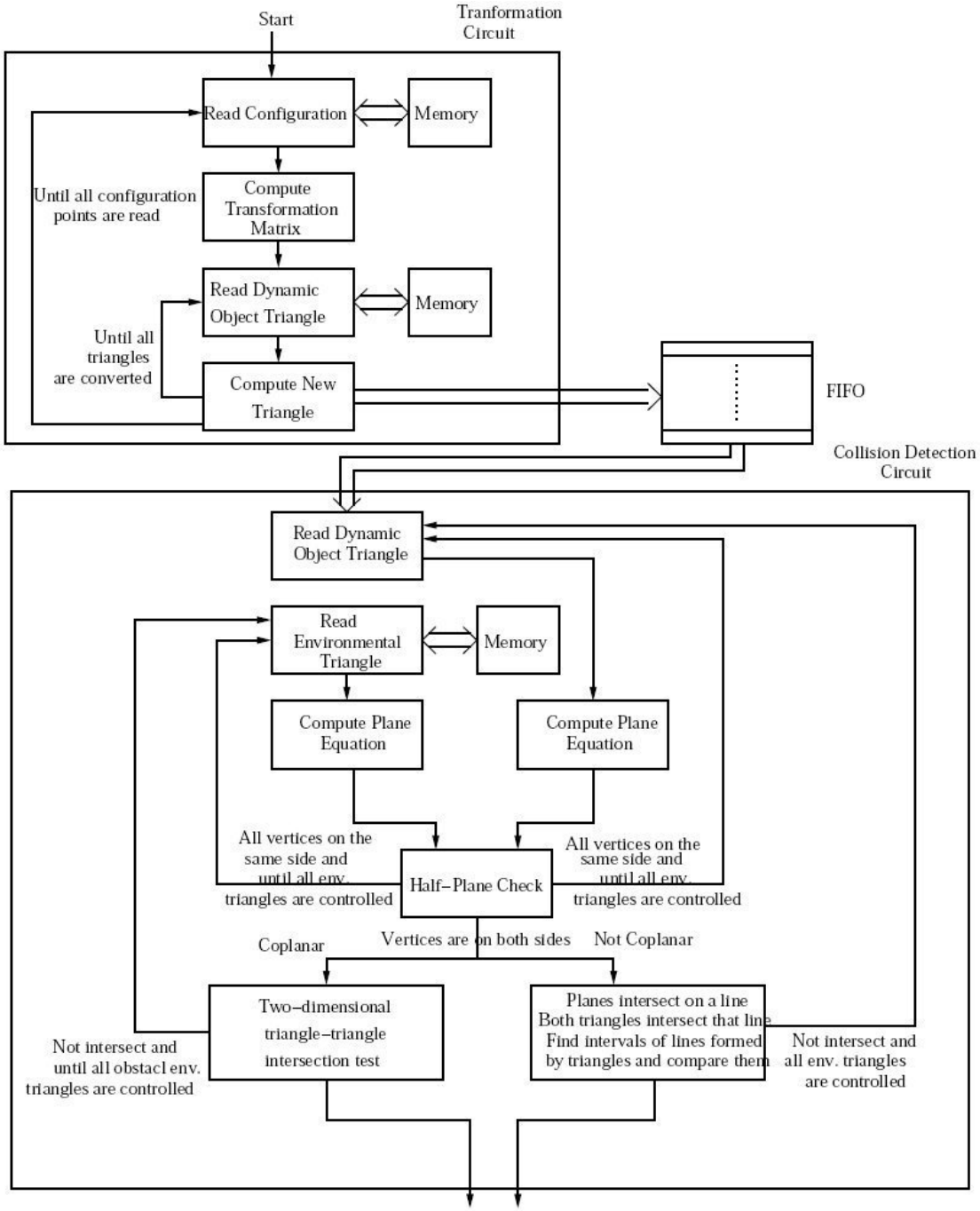


Figure 7.14: Blockdiagram of the triangle intersection circuit taken from [4].

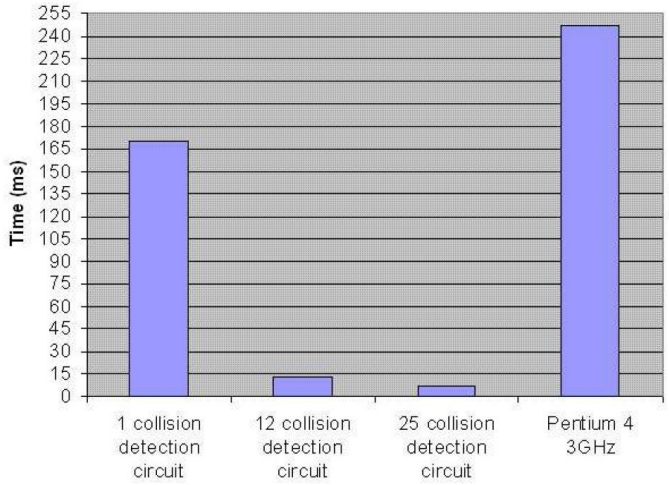


Figure 7.15: Results presented in [4].

- **coplanar** The triangles are coplanar and a two dimensional intersection test needs to be done.
- **not coplanar and not in one half-space** The planes induced by the triangles intersect in a common line L and both triangles intersect L. Intersection intervals of both triangles with L need to be calculated and checked for overlap.

Figure 7.14 shows a blockdiagram of the architecture. No further explanation on how the different parts of the architecture work is given except that the trigonometric functions necessary for the Möller test are implemented using CoreGen and that "multiplications where also generated with CoreGen". This probably accounts to usage of the embedded multipliers of Virtex-4's DSP slices. "Instead of divisions right shifts were used".

7.7.3 Performance Evaluation

For performance evaluation an object of 12 random triangles was placed in 20 different locations of an environment made up of 1600 random triangles. This was done in a ModelSim VHDL simulation processing 12 and 25 collision detection circuits in parallel, and on a real Virtex-4 XC4VLX200 running a single intersection test. Results are shown in Figure 7.15.

7.7.4 Conclusion

The approach might be effective in a very limited number of applications, still it can not be regarded as a fully featured triangle intersection test. Due to the strict memory limitation induced by the exclusive usage of Block-RAM as source of triangle data the complexity of movable and static object is strictly limited (and very low) as well.

Even worse is that there is no information given on how the positional information for the moving object is transmitted or how results are reported back to the host. The only communication channel to the host application that is mentioned is a RS 232 interface. This certainly does not suffice to feed the design with data fast enough (once per clock cycle).

Additionally, it is not detailed in the publication how the common line approach was implemented in FPGA logic, only a block diagram is given. As will be discussed in Sec. 8.7.1 the "Möller test" is a rather sequential approach. Thus it is probable that the resulting hardware design is very resource consuming. This is also suggested by the blockdiagram. A limitation resulting from this is the restriction to fixed-point numbers of only 10 bit length. Naïve rounding unavoidable leads to an increase in both, false positives *and* false negatives. This effect increases exponentially with decreasing the length of the fixed-point representation.

One can only speculate if the authors found an elegant way to implement the test, due to the quite dubious statement that right-shifts were used instead of divisions. It is also left to speculation if this imposes any further limitations.

8 CollisionChip: An FPGA-Based 3D Collision Detection Architecture

As discussed in Sec. 7.6 a major obstacle in implementing a hierarchical collision detection algorithm in FPGA technology is the tremendous space consumption. It mainly results from the usage of floating point arithmetic in order to prevent false negatives during the BVH traversal. Additionally, the more precise the calculation the less false positives occur during the traversal. This reduces the number of BV intersection tests and hence, reduces time to complete the collision query (see Sec. 7.2).

The DOP transformation test used in [8, 59] projects two 24-DOPs onto 12 axes of one of both DOPs concurrently. As already discussed in Sec. 7.6.4 this is inflexible and a waste of resources, since

- the circuit size is linearly depending on the DOP dimension,
- it requires the memory bandwidth to be big enough to load at least one 24-DOP each clock cycle (if it is not, the pipeline is stalled), and
- it cannot be exploited that if any of the projections results in non-intersecting intervals it is unnecessary to test any more intervals.

In the following Sec. 8.2 a more flexible test will be derived from the well known SAT theorem. It allows testing of an (nearly) arbitrary number of axes, while consuming minimal hardware resources. Afterwards, this work will investigate on the usability of fixed-point numbers to further reduce circuit size. Several theoretical and practical results of both, high-level simulation and synthesis will be presented in Sec. 8.3 and Sec. 8.4. Sec. 8.5 discusses the memory bottleneck and investigates different caching strategies to further speed-up collision queries while consuming minimum resources. The final synthesis results of the hierarchical bounding-volume intersection test design is presented in Sec. 8.6. In Sec. 8.7.1 several triangle and polygon intersection tests will be compared with respect to their suitability for hardware implementation and reconfiguration. The chosen primitive test is then integrated into the overall design in Sec. 8.7.2.

8.1 Space-Efficient Collision Detection

8.1.1 Efficient SAT for k-DOPs

In this work the so called Separating Axis Test (SAT) is used. It was introduced by [27,72] and was already discussed in Sec. 7.4.

In this section, an efficient Separating Axis Test for k-DOPs is derived. As discussed in Sec. 7.4 and illustrated in Figure 8.1 both k-DOPs are projected onto the candi-

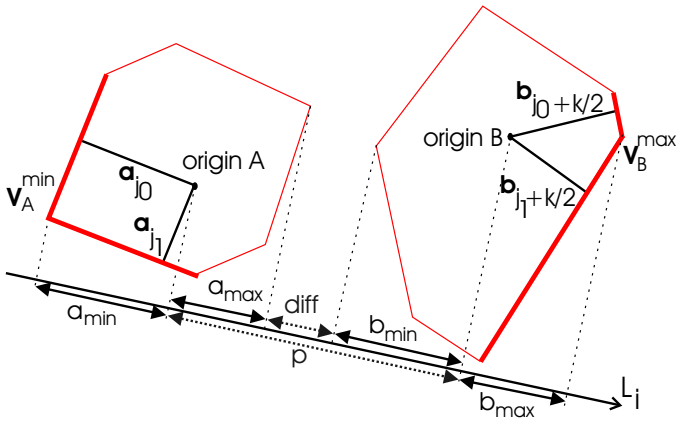


Figure 8.1: Two DOPs are projected onto test axis L_i. Since their images do not intersect L_i is a separating axis.

date separating axes. The k-DOPs intersect if at least one out of the set of candidate separating axes is a separating axis.

Afterwards, it is shown how the resulting overlap test can be done in fixed-point arithmetic such that no false negatives occur. Finally, a bound on the deviation of the projection of the fixed-point DOP with respect to the mathematically correct image is derived.

Precomputation

Since with DOPs the set of vectors $\{D_1, \ldots, D_k\}$ is fixed, it can be exploited that the set of face orientations of the DOPs are identical within a DOP-tree.

Assume object O is placed relatively to object Q by rotation $\mathbf{M}$ and translation $\mathbf{T}$. Let DT(O) and DT(Q) denote the DOP-trees of these objects. As described in Sec. 7.3, let $(\mathbf{A}_1, \ldots, \mathbf{A}_k)$ be the orientations of the DOPs' faces shared by all DOPs in DT(O) after applying rotation $\mathbf{M}$. Analogously, let $(a_1, \ldots, a_k)$ denote the DOP coefficients of DOPs in DT(O), let $(\mathbf{B}_1, \ldots, \mathbf{B}_k)$ denote the orientation vectors shared by all DOPs in DT(Q), and let $(b_1, \ldots, b_k)$ denote the corresponding DOP coefficients. In the following we will assume that all DOP coefficients are in the interval $[-1, 1]$. This can be achieved by normalising.

Note that everything independent of $(a_1, \ldots, a_k)$ and $(b_1, \ldots, b_k)$ is constant throughout the whole DOP-trees. Hence it can be precalculated at start-up to initialise the algorithm (and, later-on, the hardware). Precomputing as much as possible significantly reduces the resulting hardware costs, because it can be done in software prior to the hardware accelerated test. Since this is done only once per pair of DOP-trees, it is not time-critical.

Firstly, the n test axes $\mathbf{L}_i$ are precomputed. How this is done is detailed in Appendix B. All of the following is done for each $\mathbf{L}_i$, so for the sake of simplicity we omit the index i from now on.

Secondly, the projection $p = \mathbf{L} \cdot \mathbf{T}$ of the translation is precomputed.

Third, for each $\mathbf{L}$ a DOP has two vertices $\mathbf{v}_A^{\min}$ and $\mathbf{v}_A^{\max}$ whose projections onto $\mathbf{L}$ have maximum distance. Each of those vertices is formed by the intersection of three faces of the DOP. The correspondences $(j_{A,0}, j_{A,1}, j_{A,2})$ of the orientations whose faces meet in $\mathbf{v}_A^{\min}$ are calculated.

Fourth, and most important, in the actual projection

$$\begin{aligned} a_{\min} &= \mathbf{v}_A^{\min} \cdot \mathbf{L} \\ &= \begin{pmatrix} a_{j_{A,0}} & a_{j_{A,1}} & a_{j_{A,2}} \end{pmatrix} \cdot \begin{pmatrix} \mathbf{A}_{j_{A,0}} & \mathbf{A}_{j_{A,1}} & \mathbf{A}_{j_{A,2}} \end{pmatrix}^{-1} \cdot \mathbf{L} \end{aligned}$$

we can precompute the last dot product

$$\mathbf{P}_A := \begin{pmatrix} \mathbf{A}_{j_{A,0}} & \mathbf{A}_{j_{A,1}} & \mathbf{A}_{j_{A,2}} \end{pmatrix}^{-1} \cdot \mathbf{L} \tag{8.1}$$

$\mathbf{P}_B$ can be precomputed analogously. The mapping vectors for $\mathbf{v}_A^{\max}$ and $\mathbf{v}_B^{\max}$ are $-\mathbf{P}_A$ and $-\mathbf{P}_B$ respectively. This exploits that $k/2$ pairs of DOP orientations are anti-parallel. Note that this is an estimate to the correct solution, since not all possible combinations of DOP-coefficients share all maximum vertices. But it is impossible for any vertex made-up of the intersection of three faces to be inside the DOP, hence only false positives can result.

Intersection Testing

Using these precomputations, projecting onto the test axes can be processed very efficiently:

$$\begin{aligned} a_{\min} &= \begin{pmatrix} \mathbf{a}_{j_{A,0}} & \mathbf{a}_{j_{A,1}} & \mathbf{a}_{j_{A,2}} \end{pmatrix} \cdot \mathbf{P}_A \\ a_{\max} &= \begin{pmatrix} \mathbf{a}_{j_{A,0}+k/2} & \mathbf{a}_{j_{A,1}+k/2} & \mathbf{a}_{j_{A,2}+k/2} \end{pmatrix} \cdot (-\mathbf{P}_A) \end{aligned} \tag{8.2}$$

This is done for $b_{\min}$ and $b_{\max}$ analogously.

The condition for separation is straight-forward now. Let

$$\begin{aligned} \text{diff}_1 &:= (a_{\min} + p) - b_{\max} \\ \text{diff}_2 &:= b_{\min} - (a_{\max} + p) \end{aligned} \tag{8.3}$$

$$\text{diff} := \max(\text{diff}_1, \text{diff}_2) \tag{8.4}$$

then the intervals $[a_{\min}, a_{\max}]$ and $[b_{\min}, b_{\max}]$ are disjoint if and only if $\text{diff} > 0$. And from the Separating Axis Theorem we know that

$$(\text{diff} > 0) \Rightarrow \text{separation}. \tag{8.5}$$

Eqs. (8.2)–(8.5) have to be processed for every individual DOP test. Hence they cannot be precomputed.

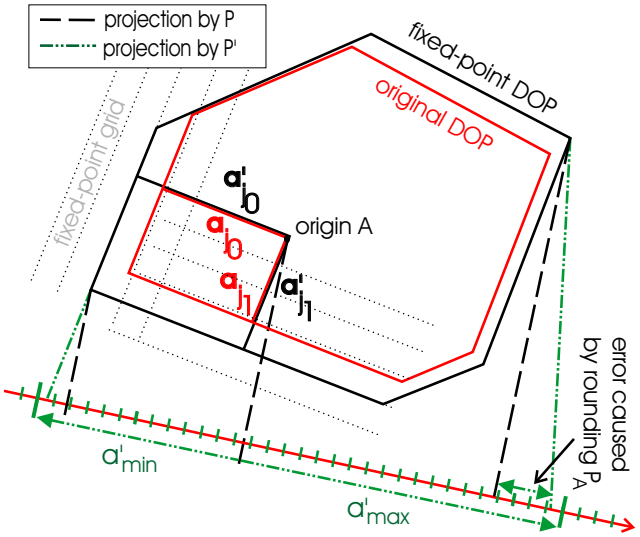


Figure 8.2: A DOP and its enclosing fixed-point equivalent. Both rounding the DOP to fixed-point numbers and projecting it with $\mathbf{P}'$ instead of $\mathbf{P}$, increases the DOP's image. Thus, it contains the according floating-point image. When checked for intersection false positives can occur.

8.1.2 Fixed-Point Arithmetic

As already discussed floating-point arithmetic is very expensive in hardware implementations considering circuit size and depth. Unfortunately, simply rounding DOP coefficients to fixed-point numbers would result in false negatives, because the intervals on the test axes could become smaller than the projection of the enclosed object. As discussed in Sec. 7.2 these false negatives are unacceptable, because collisions might be missed. Naïve rounding of the mapping vectors $\mathbf{P}_A$ and $\mathbf{P}_B$ would lead to even more false negatives since distance of the images could be overestimated. Hence rounding needs to be done in a manner such that each fixed-point DOP image contains the according floating-point image. Figure 8.2 shows an example of a correctly rounded fixed-point DOP and a projection.

Correct Fixed-Point Rounding

First, the smaller scale of fixed-point numbers needs to be handled by dividing all DOP coefficients of all DOPs by the largest absolute value of the DOP coefficients in the scenario. This way, 16-bit accuracy still allows concurrent use of DOPs the size of a skyscraper and of a 6mm screw. 36-bit even allow for DOPs the size of the sun and of a screw concurrently. This is regarded as sufficiently precise in the following.

In the following rounding of the DOP coefficients to b-bit after the point towards $+\infty$ is denoted by $a'_i = \lceil a_i \rceil$. Clearly, the rounded (i.e., fixed-point) DOP contains the original one. Then, $\varepsilon_i = a'_i - a_i$ is the resulting rounding error, with $0 \leq \varepsilon_i < 2^{-b}$.

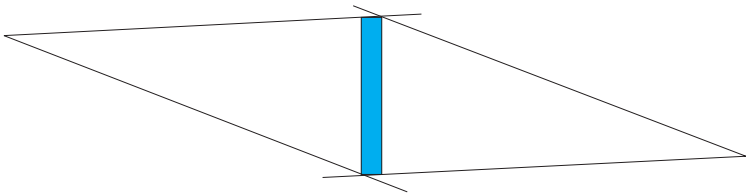


Figure 8.3: An oblong object bounded by a 4-DOP with acute angles.

By ensuring that the dihedral angle between all pairs of neighbouring faces of a DOP is larger than $\pi/2$, all $P_{A,i}$ are in the interval $[-1, 0]$. This is formally proven in Appendix A.1.

Abandoning angles larger than $\pi/2$ is not a hard restriction, since every well constructed DOP should not have acute angles to improve tightness of fit. This even accounts for oblong objects, since they can occur in random orientation. This is illustrated by Figure 8.3

Rounding $\mathbf{P}_{A,i}$ towards $-\infty$ to c-bit accuracy results in a rounding error $0 \leq \eta_i = \mathbf{P}_{A,i} - \mathbf{P}'_{A,i} < 2^{-c}$.

By simply truncating $\mathbf{P}_{A,i}$, the resulting image would become too small in case of negative DOP coefficients, whereas always rounding up would create the same problem with positive coefficients. The presented approach solves this problem during calculation simply by adding 2^{-c} to $\lfloor \mathbf{P}_{A,i} \rfloor$ before multiplication with negative DOP coefficients. This effectively compensates the error.

Let $\mathrm{sn}(\mathbf{x})$ be the sum of all negative x_i:

$$\mathrm{sn}(\mathbf{x}) := \sum_{\substack{i=0 \\ \mathbf{x}_i<0}}^{2} x_i$$

and let $\mathbf{a}' := (a'_{j_{A,0}}\ a'_{j_{A,1}}\ a'_{j_{A,2}})$, and $\mathbf{a}'_k := (a'_{j_{A,0}+k/2}\ a'_{j_{A,1}+k/2}\ a'_{j_{A,2}+k/2})$.

Then, correct rounding of the images amounts to:

$$\begin{aligned} a'_{\min} &= \mathbf{P}'_A \cdot \mathbf{a}' + 2^{-c}\,\mathrm{sn}(\mathbf{a}') \\ a'_{\max} &= -(\mathbf{P}'_A \cdot \mathbf{a}'_k + 2^{-c}\,\mathrm{sn}(\mathbf{a}'_k)) \end{aligned} \tag{8.6}$$

Finally, when computing diff_1, p can be simply truncated to z-bit ($p' = \lfloor p \rfloor$). This can create false positives exclusively, because a smaller p' only decreases the apparent distance between the two DOP images. For diff_2 it is necessary to round p up to $\lceil p \rceil$, which, again, can be done efficiently by adding 2^{-z} to $\lfloor p \rfloor$.

Overall, calculating the distances of the fixed-point DOP images amounts to

$$\begin{aligned} \mathrm{diff}'_1 &= (\mathbf{a}'_{\min} + p') - \mathbf{b}'_{\max} \\ \mathrm{diff}'_2 &= \mathbf{b}'_{\min} - (\mathbf{a}'_{\max} + (p' + 2^{-z})) \end{aligned} \tag{8.7}$$

$$\mathrm{diff}' = \max(\mathrm{diff}'_1, \mathrm{diff}'_2) \tag{8.8}$$

Now the condition for separation can be given analogously to Eq. 8.5:

$$((\text{diff}'_1 > 0) \text{ or } (\text{diff}'_2 > 0)) \Rightarrow \text{separation} \tag{8.9}$$

Thus, false negatives are prevented already by the way the algorithm was constructed. Now it remains to proof, that only a limited number of additional test is generated by the occurance of additional false positives.

Bound on Fixed-Point Deviation

In this section a bound on the deviation of the rounded image's distance on a separating axis from the mathematically correct distance is derived.

Let err denote this deviation (called *fixed-point error* in the following).

$$\text{err} := \text{diff} - \text{diff}' \tag{8.10}$$

Let

$$\begin{aligned} \text{err}_1 &:= \text{diff}_1 - \text{diff}'_1 \\ \text{err}_2 &:= \text{diff}_2 - \text{diff}'_2 \end{aligned} \tag{8.11}$$

Since diff is defined as $\max(\text{diff}_1, \text{diff}_2)$ and diff′ is defined analogously and we know that the images are non-intersecting it follows

$$\min(\text{err}_1, \text{err}_2) \leq \text{err} \leq \max(\text{err}_1, \text{err}_2) \tag{8.12}$$

Inserting Eqs. (8.2)–(8.4) and (8.6)–(8.8) into Eq. (8.11) yields

$$\begin{aligned} \text{err}_1 =& (\mathbf{P}_A \cdot \mathbf{a} - \mathbf{P}'_A \cdot \mathbf{a}') - 2^{-c} \cdot \text{sn}(\mathbf{a}') \\ &+ (\mathbf{P}_B \cdot \mathbf{b}_k - \mathbf{P}'_B \cdot \mathbf{b}'_k) - 2^{-c} \cdot \text{sn}(\mathbf{b}'_k) \\ &+ (p - p') \end{aligned} \tag{8.13}$$

and err_2 can be calculated analogously.

To calculate bounds on err_1 and err_2 the errors caused by products of mapping vectors and DOP coefficients need to be bound first. Since this is all very similar, it is done

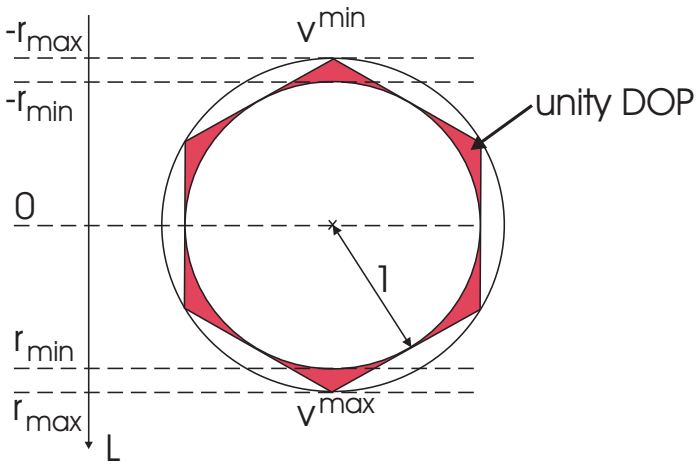


Figure 8.4: r_{max} is defined as the greatest distance and $r_{min} = 1$ as the smallest distance of the origin to any point on the surfaces of the unity DOP. This makes $-r_{max}$ a lower bound and $-r_{min}$ an upper bound on the cross sum of any mapping vector $\mathbf{P}$, since the unity DOP is the DOP with maximum coeffecients.

exemplarily for $\mathbf{P}_A \cdot \mathbf{a} - \mathbf{P}'_A \cdot \mathbf{a}'$.

$$\begin{aligned}
&\mathbf{P}_A \cdot \mathbf{a} - \mathbf{P}'_A \cdot \mathbf{a}' \\
=&(\mathbf{P}_A - \mathbf{P}'_A) \cdot \mathbf{a}' + \mathbf{P}_A \cdot \mathbf{a}' - \mathbf{P}'_A \cdot \mathbf{a}' \\
=&(\mathbf{P}_A - \mathbf{P}'_A) \cdot \mathbf{a}' + \mathbf{P}_A \cdot (\mathbf{a} - \mathbf{a}') \\
=&\sum_{i=0}^{2} (\mathbf{P}_{A,i} - \mathbf{P}'_{A,i}) \cdot \mathbf{a}'_{j_{A,i}} + \sum_{i=0}^{2} \mathbf{P}_{A,i} \cdot (\mathbf{a}_{j_{A,i}} - \mathbf{a}'_{j_{A,i}}) \\
=&\sum_{\substack{i=0 \\ a'_{j_{A,i}} \geq 0}}^{2} (\mathbf{P}_{A,i} - \mathbf{P}'_{A,i}) \cdot \mathbf{a}'_{j_{A,i}} + \sum_{\substack{i=0 \\ a'_{j_{A,i}} < 0}}^{2} (\mathbf{P}_{A,i} - \mathbf{P}'_{A,i}) \cdot \mathbf{a}'_{j_{A,i}} \\
&+ \sum_{i=0}^{2} \mathbf{P}_{A,i} \cdot (\mathbf{a}_{j_{A,i}} - \mathbf{a}'_{j_{A,i}})
\end{aligned} \tag{8.14}$$

The cross sum of any mapping vector can be interpreted as the image of a vertex of the maximum DOP (all $d_i = 1$).

$$\sum_{i=0}^{2} P_i = \begin{pmatrix} 1 & 1 & 1 \end{pmatrix} \cdot \mathbf{P} \tag{8.15}$$

Let r_{max} be the greatest distance and $r_{min} = 1$ be the smallest distance of the origin to any point on the surfaces of the maximum DOP, which is the unity DOP (see Figure 8.4).

Then $-r_{max}$ is a lower bound and $-r_{min}$ is an upper bound on the cross sum of any mapping vector $\mathbf{P}$.

$$-r_{max} \leq \sum_{i=0}^{2} P_i \leq -r_{min} = -1 \tag{8.16}$$

Let $\mathrm{sp}(\mathbf{x})$ (analogously to $\mathrm{sn}(\mathbf{x})$) be the sum of all $\mathbf{x}_i \geq 0$.

$$\mathrm{sp}(\mathbf{x}) := \sum_{\substack{i=0 \\ \mathbf{x}_i \geq 0}}^{2} x_i$$

With the known boundaries

$$\begin{array}{llll} -1 \leq \mathbf{a}'_i & \leq 1 & -2^{-b} \leq \mathbf{a}_i\text{-}\mathbf{a}'_i & \leq 0 \\ -1 \leq \mathbf{P}_{A,i} & \leq 0 & 0 \leq \mathbf{P}_{A,i}\text{-}\mathbf{P}'_{A,i} & \leq 2^{-c} \\ & & 0 \leq p\text{-}p' & \leq 2^{-z} \end{array}$$

all terms of the sum Eq. 8.14 can be bound:

$$0 \cdot \mathrm{sp}(\mathbf{a}') \leq \sum_{\substack{i=0 \\ a'_{j_{A,i}} \geq 0}}^{2} (\mathbf{P}_{A,i} - \mathbf{P}'_{A,i}) \cdot \mathbf{a}'_{j_{A,i}} \leq 2^{-c} \cdot \mathrm{sp}(\mathbf{a}')$$

$$2^{-c} \cdot \mathrm{sn}(\mathbf{a}') \leq \sum_{\substack{i=0 \\ a'_{j_{A,i}} < 0}}^{2} (\mathbf{P}_{A,i} - \mathbf{P}'_{A,i}) \cdot \mathbf{a}'_{j_{A,i}} \leq 0 \cdot \mathrm{sn}(\mathbf{a}')$$

$$-r_{min} \cdot 0 \leq \sum_{i=0}^{2} \mathbf{P}_{A,i} \cdot (\mathbf{a}_{j_{A,i}} - \mathbf{a}'_{j_{A,i}}) \quad \leq -r_{max} \cdot (-2^{-b})$$

Along with Eq. 8.14 this amounts to

$$2^{-c} \cdot \mathrm{sn}(\mathbf{a}') \leq \mathbf{P}_A \cdot \mathbf{a} - \mathbf{P}'_A \cdot \mathbf{a}' \leq 2^{-c} \cdot \mathrm{sp}(\mathbf{a}') + r_{max} \cdot 2^{-b} \tag{8.17}$$

Inserting Eq. 8.17 into Eq. 8.13 yields

$$\begin{aligned} \mathrm{err}_1 \leq & 2^{-c} \cdot \mathrm{sp}(\mathbf{a}') + r_{max} \cdot 2^{-b} - 2^{-c}\,\mathrm{sn}(\mathbf{a}') \\ & + 2^{-c} \cdot \mathrm{sp}(\mathbf{b}'_k) + r_{max} \cdot 2^{-b} - 2^{-c}\,\mathrm{sn}(\mathbf{b}'_k) + 2^{-z} \end{aligned} \tag{8.18}$$

and

$$\begin{aligned} \mathrm{err}_1 \geq & 2^{-c} \cdot \mathrm{sn}(\mathbf{a}') - 2^{-c}\,\mathrm{sn}(\mathbf{a}') \\ & + 2^{-c} \cdot \mathrm{sn}(\mathbf{b}'_k) - 2^{-c}\,\mathrm{sn}(\mathbf{b}'_k) + 0 = 0 \end{aligned} \tag{8.19}$$

Calculating bounds on err_2 can be done analogously and results in

$$\begin{aligned}\mathrm{err}_2 \leq & 2^{-c} \cdot \mathrm{sp}(\mathbf{b}') + r_{max} \cdot 2^{-b} - 2^{-c}\,\mathrm{sn}(\mathbf{b}') \\ & + 2^{-c} \cdot \mathrm{sp}(\mathbf{a}'_k) + r_{max} \cdot 2^{-b} - 2^{-c}\,\mathrm{sn}(\mathbf{a}'_k) - 0 + 2^{-z}\end{aligned} \tag{8.20}$$

and

$$\begin{aligned}\mathrm{err}_2 \geq & 2^{-c} \cdot \mathrm{sn}(\mathbf{b}') - 2^{-c}\,\mathrm{sn}(\mathbf{b}') \\ & + 2^{-c} \cdot \mathrm{sn}(\mathbf{a}'_k) - 2^{-c}\,\mathrm{sn}(\mathbf{a}'_k) - 2^{-z} + 2^{-z} = 0\end{aligned} \tag{8.21}$$

Ensuring the dihedral angles between all pairs of neighbouring faces exceed $\pi/2$, r_{max} is bounded by $\sqrt{3}$ (see Appendix A.2). Combining this with Eq. 8.18–8.21 and inserting the result into Eq. 8.13 yields the overall result

$$0 \leq \mathrm{err} \leq \sqrt{3} \cdot 2^{-b+1} + 6 \cdot 2^{-c} + 2^{-z} \tag{8.22}$$

This gives a bound on the deviation of the image size of a fixed-point DOP with respect to the exact image.

8.2 The Architecture

This section discusses how the theoretically obtained results can be efficiently implemented in a hardware design. It additionally presents how the basic architecture, that directly results from the previously presented equations, can be further optimised using various techniques and how the remaining free parameters need to be chosen to provide maximum performance.

8.2.1 The Pipeline

Combining Eqs. (8.6)–(8.9) results in the overlap condition

$$\begin{gathered}\mathbf{P}'_A \cdot \mathbf{a}' + 2^{\text{-}c}\,\mathrm{sn}(\mathbf{a}') + \mathbf{P}'_B \cdot \mathbf{b}'_k + 2^{\text{-}c}\,\mathrm{sn}(\mathbf{b}'_k) + p' > 0 \\ \text{or} \\ \mathbf{P}'_B \cdot \mathbf{b}' + 2^{\text{-}c}\,\mathrm{sn}(\mathbf{b}') + \mathbf{P}'_A \cdot \mathbf{a}'_k + 2^{\text{-}c}\,\mathrm{sn}(\mathbf{a}'_k)\text{-}(p' + 2^{\text{-}z}) > 0 \\ \Rightarrow \text{separation}\end{gathered} \tag{8.23}$$

Eq. (8.23) is divided into nine stages to enable pipelining.

Selection Stage one selects the 12 out of k DOP coefficients defining the outer (maximal) vertices for a given candidate separating axis based on the correspondences (j_A,j_B).[1] Correspondences j_A and j_B contain the indices of 6 of them. Due to the chosen k-DOPs representation the 6 remaining ones can be derived by simply increasing these indices each by $k/2$. Since wrap around indexing is used, this does not need any combinational hardware, but can simply be done by inputting the coefficients into the multiplexers in modified order.

[1] There are 2 k-DOPs, 2 maximal vertices per DOP, and 3 coefficients defining each vertex (see Sec. 8.1.1).

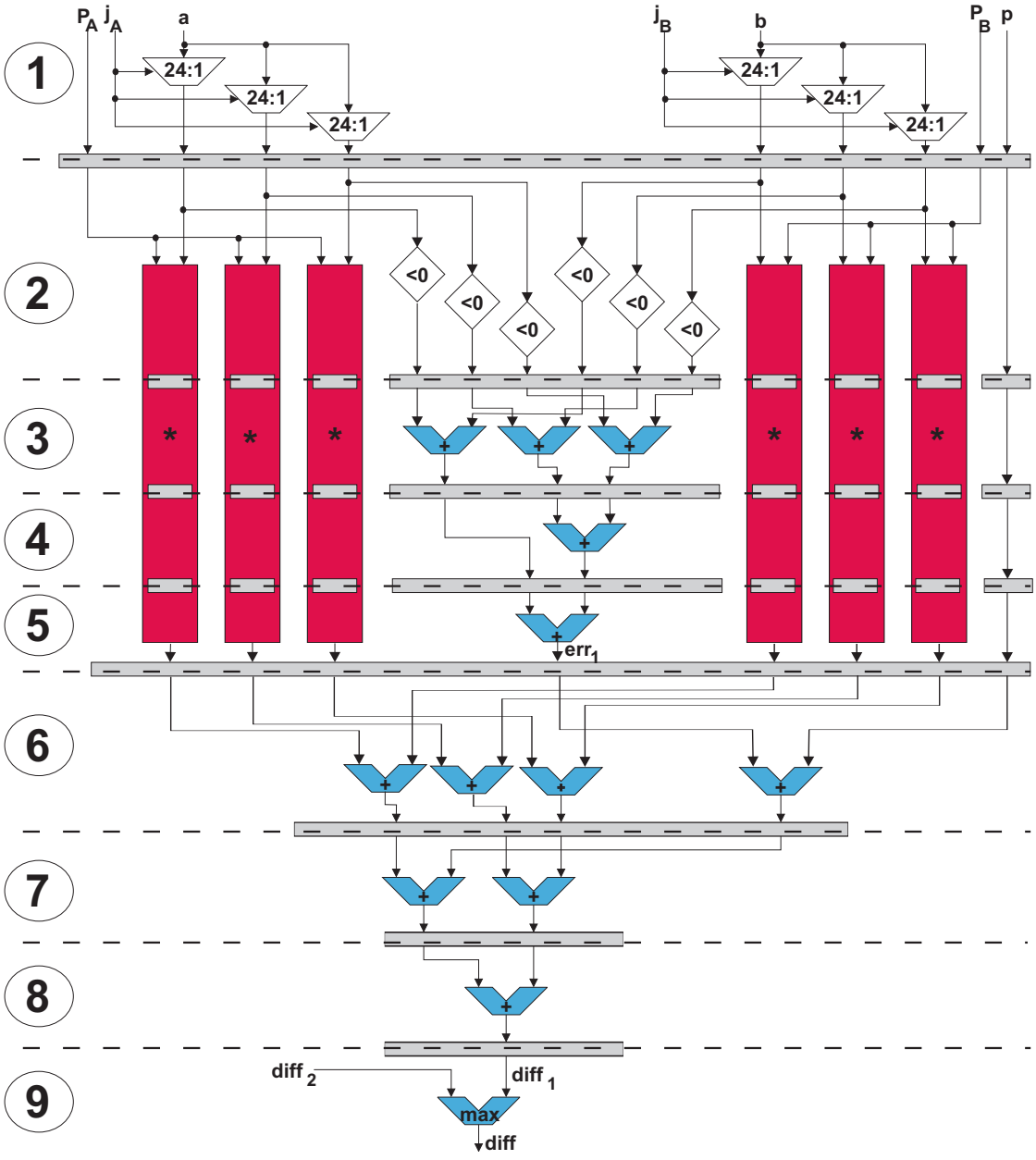


Figure 8.5: Blockdiagram of the pipeline. Stage one selects 12 out of k coefficients based on the correspondences (j_A,j_B). The adder-tree in the center calculates the error correction term err_1. Multiplications necessary for the actual projection are processed in the three-staged multipliers provided by the Virtex-II target architecture. This is both done in stages two to five. Stages six, seven and eight complete the calculation of diff_1 and combine it with err_1. diff, the maximum of diff_1 and diff_2, which is computed analogously to diff_1, is processed in stage nine.

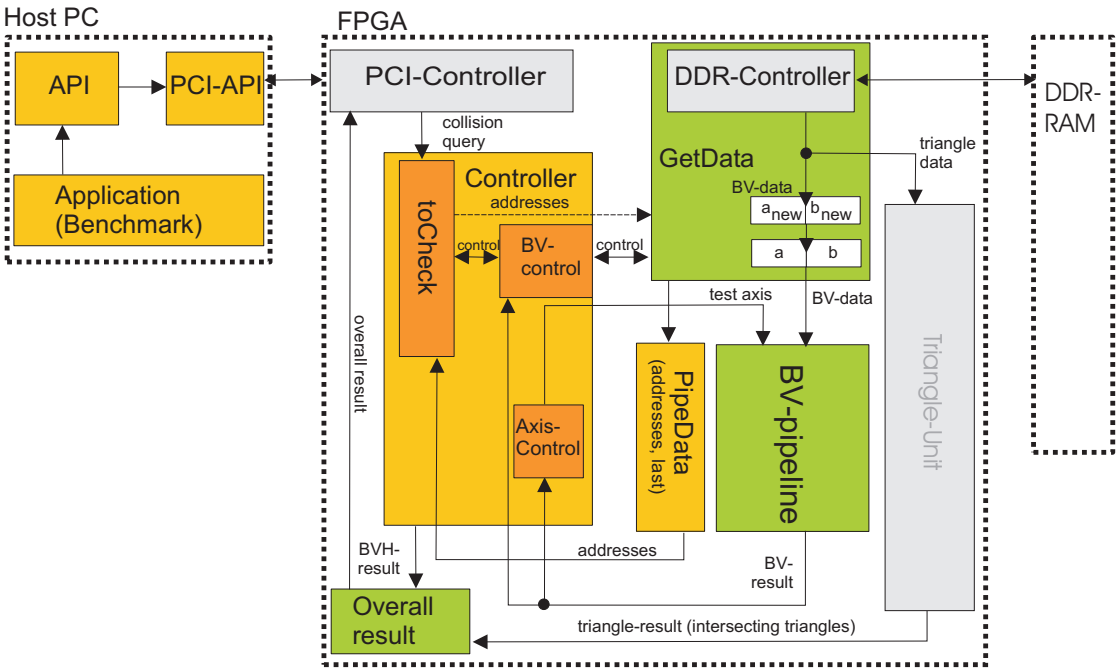


Figure 8.6: The basic architecture of the intersection test hardware.

Scalar Products and Fixed-Point Correction Stages two to seven implement the calculation of the scalar products and the fixed-point correction term. So, DOP coefficients have to be multiplied by $\mathbf{P}'$-vector entries and summed up by an adder tree. Additionally, p' $\big($or $-(p'+2^{-z})$ in case of diff'_2 respectively $\big)$ is added. Concurrently, negative DOP coefficients are selected and accumulated. The target architecture features three staged multipliers to allow maximising the clock frequency. As illustrated in Figure 8.5 this feature is utilized and concurrent calculations are synchronised with it.

Stage eight adds the results of both summations. Multiplying by 2^{-c} is done implicitly by shifting.

Result Testing $\max(\mathrm{diff}'_1, \mathrm{diff}'_2) > 0$ is done in a single stage by negating the conjunction of the sign bits.

8.2.2 Overall Design

The overall architecture is shown in Figure 8.6. The calculation is initialized by the host system by sending $(\mathbf{P}'_A, \mathbf{P}'_B, p', j_A, j_B)$ and the addresses of the DOP-trees to the hardware. A controller keeps track of DOP overlap tests that must still be executed and requests the needed DOP coefficients and triangle data. The module `GetData` reads them from memory concurrently to the current calculation. As soon as the parameters are loaded and the last calculation is finished, it feeds them into the pipeline (or the triangle-unit respectively). The pipeline receives not only the DOP coefficients but (from the

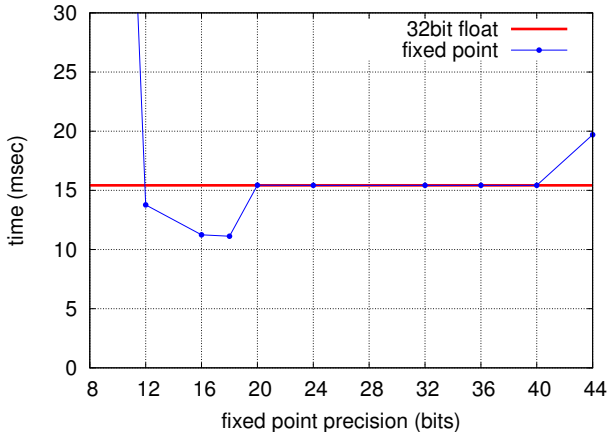


Figure 8.7: Speed of fixed-point arithmetic for different bit widths. Beyond 18-bit a second, and beyond 40-bit a third memory burst is needed.

controller) the data necessary for processing the next axis test. For each DOP pair, n axes are tested. A shift register `PipeData` holds additional bookkeeping information. For every pipeline stage it contains the indices of the processed DOPs and whether the contained calculation is the last axis test to be executed for the current DOP pair. If this last axis test leaves the pipeline and none of the test axes is a separating axis the controller schedules the child DOPs to be tested in a sequencing module `toCheck`. If a separating axis is found, the remaining calculations belonging to the same DOP-pair are obsolete. No new axis tests are initiated and the results of the calculations that are still in the pipeline will be ignored; no new DOP tests are scheduled.

As soon as `toCheck`, pipeline, and the `GetData` module are empty, and no intersecting triangles were found, the objects do not intersect and this is reported to the host application.

As was discussed in Sec. 7.6 scheduling DOPs in a stack is superior to queuing them and thus a stack is used in `toCheck`. How this can be further optimised is discussed in Sec. 8.3.

In Sec. 8.1.1 the fixed-point test was presented and it was discussed that accuracy influences speed of the overall design. Simulations done in SystemC early in the design process verified this. Figure 8.7 shows the inlfuence of the chosen bit-width. Below 18-bit accuracy, an increasing number of false positives occurs compared to the floating-point implementation and decreases calculation speed. Above 18-bit, a second memory burst is needed to fetch DOP coefficients from DDR-RAM.

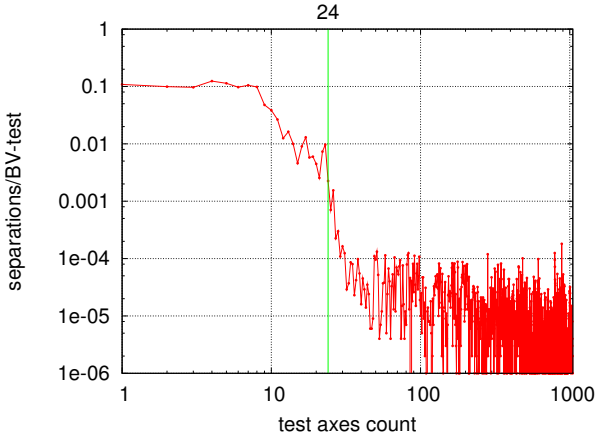


Figure 8.8: The more axes are tested for intersection the less probable it is for other axes to be separating.

8.3 Control

8.3.1 Push and Pull Control Architecture

As was previously discussed, a full separating axis test requires testing of all axes orthogonal to a face of either k-DOP or orthogonal to an edge of each k-DOP. As also discussed before, it is not necessary to test all axes $\mathbf{L}_i$ whether they are separating axes. Even more, [72] showed that if oriented bounding boxes are used as bounding-volumes, the probability of the BVs to be disjoint decreases rapidly with every non-separating test axis found so far. Thus it is not efficient to test all axes.

As shown in Figure 8.8 this applies for DOPs, too.

On the other hand, disjoint branches of the DOP trees should be eliminated as early as possible to reduce expensive loading of DOP coefficients. This can be achieved by evaluating which $n \leq N$ gives the best trade-off between axis-testing and parameter-loading. As Figure 8.9 shows, $n = 24$ yields the optimum performance for 24-DOPs and the given memory architecture. 24 axis-tests suffice to test all candidate separating axes generated from the 12 face-orientations of each DOP. Although this exceeds the time to load a complete set of DOP-coefficients (only 20 clock-cycles) by 4 cycles, testing 24 axes seems to reduce the number of false positives sufficiently to yield best performance. This form of control will be called a "pull control architecture" since the next pair of DOPs is read (pulled) from the registers when the test of the preceding pair is finished.

This leads to stalling the memory infrastructure whenever a test needs more time than loading the next data, e.g., if coefficients of both DOPs need to be loaded. Still, there is no reason to stop testing axes if the next DOP-pair is not completely loaded yet. This can occur, e.g., if the memory subsystem is occupied by loading triangle data. Both, stalling the pipeline and stalling the memory infrastructure can be avoided by starting a new calculation whenever a new pair of DOPs is ready to be tested. In the meantime

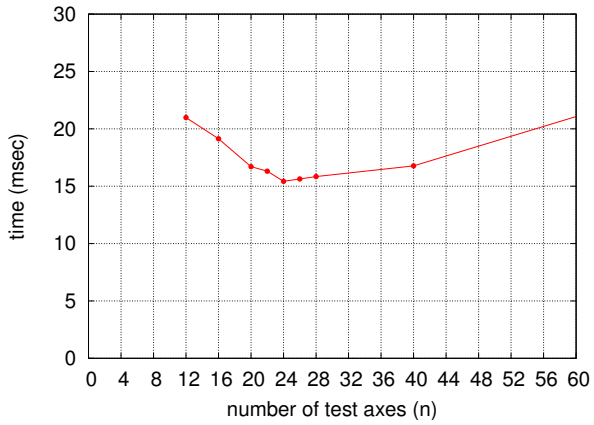


Figure 8.9: For fixed $k = 24$, the presented design performs best using $n = 24$ on the target architecture.

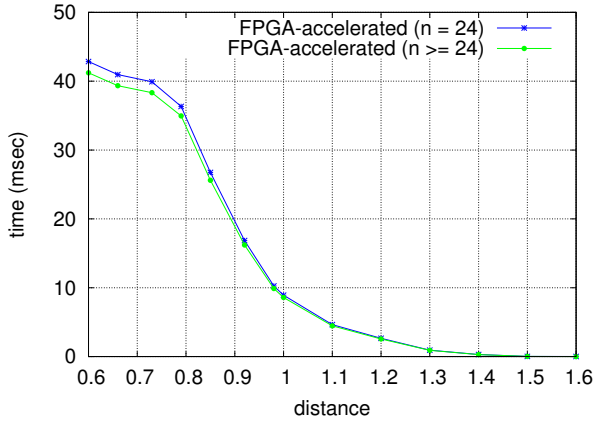


Figure 8.10: Testing further axes until next DOP-pair is loaded yields a small speed-up.

further axes can be tested for intersection. Since finding a separating axis prevents the algorithm from descending deeper into the test tree, the number of bounding-volume tests to be scheduled is decreased. Here a minimum number of 24 tests to be performed needs to be defined. Otherwise the test could run down the test tree without considering enough (or worse any) axes and this way reduce performance instead of speeding-up the calculation. As shown in Figure 8.10 continuing axis testing until the next set of DOP-coefficients is fetched from memory speeds-up calculation by about 2%. Since this requires only slight changes in control, the additional overhead in development time and circuit size is insignificant. This approach will be called “push control architecture” in the following, since new data is “pushed” into the pipeline whenever it arrives.

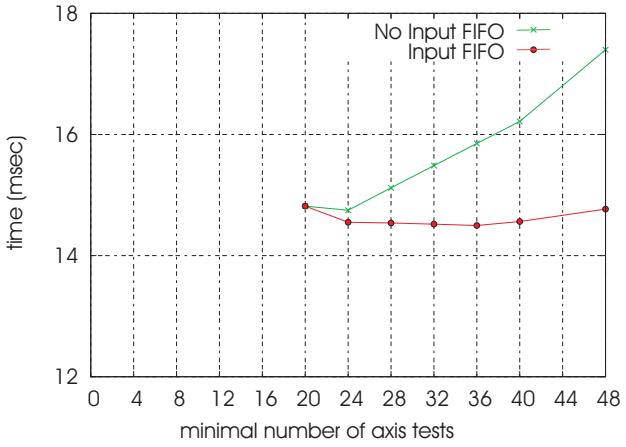


Figure 8.11: Buffering DOP coefficiens prior to inputting them into the BV test pipeline and increasing the minimal number of axis tests to be performed saves 2% of runtime.

8.3.2 Input FIFO

In order to avoid very intensive testing of a very small number of DOP pairs while waiting for the next data to arrive, it is possible to "spread" the additional axis tests over several DOP pairs. The easiest way to do this is implementing a FIFO that buffers DOP coefficients prior to inputting them into the BV test pipeline. Hence it will be called "input FIFO" in the following. Now the minimal number of axis tests to be performed can be increased. Figure 8.11 illustrates that this also saves only 2% of runtime. This speed-up does not justify the additional resource consumption of a FIFO implementation. As will be shown in Sec. 8.5 this changes if an additional cache is used.

8.3.3 Optimizing Tree Traversal

As discussed in Sec. 7.6.3 FIFO control results in a breadth-first traversal of the test tree and LIFO control proceeds in a depth-first manner along several paths. As was also discussed LIFO control consumes far less memory then FIFO control. This saves a lot of chip space, since this sequence structure needs to be very fast and hence has to be implemented on-chip. For the architecture presented in this dissertation LIFO control bests FIFO control in respect of resource consumption, too. This is shown in Figure 8.12. This is inevitable since in a balanced tree breadth-first traversal needs to store all test queries of the same depth before any of them is processed. Due to the pipelined processing of the queries LIFO control is not strictly depth-first, hence it also benefits from scheduling BV tests in optimised brotherhood order. Still, implementing `toCheck` as a FIFO results in 7% speed-up compared to LIFO control (see Figure 8.13). To further reduce resource consumption and combine it with faster processing a mixture of both is proposed. Storing the sons of an individual DOP in direct succession in the DDR-SDRAM allows saving only one of the DOPs addresses, while still being able to retrieve the other as well. Now the BV test queries are scheduled in a LIFO. Instead

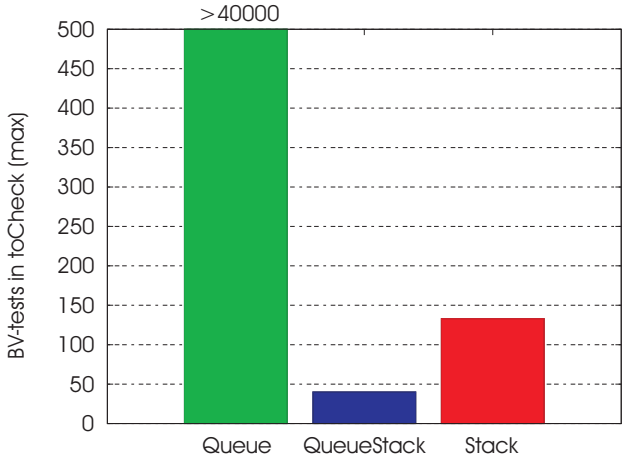


Figure 8.12: For the presented architecture it is also true that LIFO control consumes far less memory then FIFO control. The combined structure reduces resource consumption by a factor of 4.

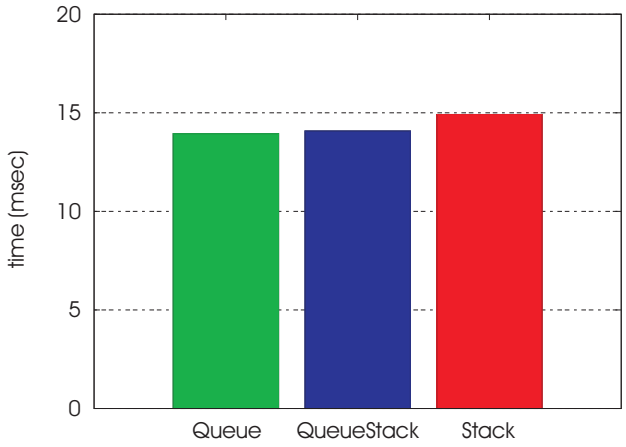


Figure 8.13: Implementing `toCheck` as a FIFO results in 7% speed-up compared to LIFO control. The combined structure is as fast as the FIFO.

of scheduling all four tests resulting from a single BV test result only one is stored as a placeholder. If it is popped it can be expanded to the full four tests, which are then stored in a FIFO with only four entries. From there they are popped. If the FIFO runs empty the next placeholder test is popped from the LIFO and is expanded. As also shown in Figure 8.12 and Figure 8.13 this technique effectively reduces the maximum number of BV tests to be stored by a factor of four compared to LIFO control, while providing the (small) speed-up of the FIFO traversal scheme.

8.4 Results of the Basic Architecture

The target architecture is a Xilinx Virtex II (XC 2V6000, speed grade -4) on an Alpha Data ADM-XRC-II board with 256 MB DDR-RAM at 100MHz connected via a 64 bit wide bus. The FPGA features 144 18-bit multipliers and 6 million gate equivalents. Synthesis, placing, routing, and mapping were done with Xilinx ISE 8.0.

8.4.1 Synthesis Results

Although 19-bit accuracy performs best on the test data with respect to calculation time (Figure 8.7), the pipeline was implemented for 32-bit fixed-point 24-DOPs to tolerate bigger differences in DOP size (see Sec. 8.1.2). Since the target architecture features 18-bit multipliers only, this results in two additional pipeline stages to implement pipelined 32-bit multipliers.

Overall, the pipeline utilises a total of 7278 out of 33792 slices (21% = 1,260,000 million gate equivalents). Maximum clock frequency is 111.117MHz.

8.4.2 Benchmarking

All results presented here were obtained in RT-level simulations. Two identical objects (the car headlight shown in Figure 7.9) with 5947 triangles [58] are placed at different distances from each other and with different rotations. For each constellation, the time to detect all intersecting triangles is determined. Figure 8.14 shows the comparison of the novel architecture with the state-of-the-art software DOP intersection test presented in [82] running on a 1 GHz Pentium III with 512 MByte main memory. Memory bandwidth and speed are nearly identical on both systems and hence allow for a direct performance comparison. This configuration is used for all following comparisons with software. The presented basic architecture yields a speed up of about factor 4.

8.5 Defying the Memory Bottleneck

In the previous sections a basic architecture for FPGA-based hardware acceleration of collision detection queries was introduced. The following section investigates on the architecture's interaction with the memory interface. As will be shown in the following the major bottleneck in hardware accelerated collision detection are memory accesses.

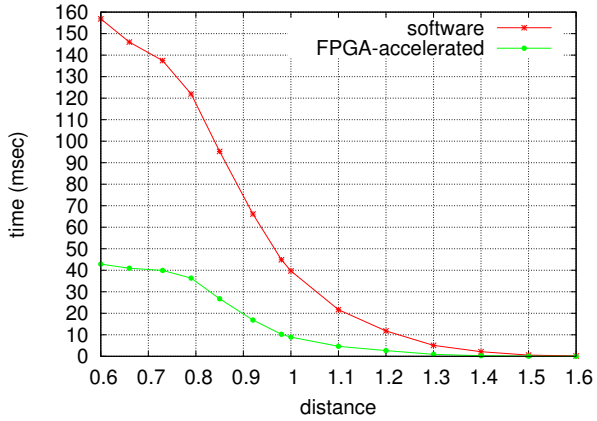


Figure 8.14: Driven by a 100MHz clock the presented architecture is approximately 4 times faster than a state-of-the-art software intersection test.

Hence special effort needs to be spent to minimise the number of memory accesses and to maximise performance of the memory hierarchy. A specialised tree-traversal algorithm is presented that exploits arbitrary memory interfaces optimally to minimise delay of collision queries. Along with this a novel caching technique is introduced that combines high-speed access to the bounding-volume hierarchy with minimal resource consumption. A very fast, yet hardware efficient collision detection hardware results.

8.5.1 Investigating on Benefits of Caching

As discussed in Sec. 7.2 using the optimised brotherhood traversal scheme provides some benefit regarding the loading order of DOP coefficients. The test order here is
A1-B2 B3 C3 C2-D4 D5 E5 E4-D6 D7 E7 E6-F4 F5 G5 G4-F6 F7 G7 G6.
This way it is possible to mostly ”keep” one DOP for the next calculation resulting in the following loading order:
A,1,B,2,3,C,2,D,4,5,E,4,D,6,7,E,6,F,4,5,G,4,F,6,7,G,6.
It can easily be seen that this still is not optimal, since almost half of the DOP loadings could still be saved. The optimal order is: A,1,B,2,3,C,D,4,5,E,6,7,F,G.

Hard wiring such an order requires unrealistic effort, hence this potential can only be exploited using caching techniques.

The results of a systematic investigation on the influence of the memory bandwidth are shown in Figure 8.15. It shows that speeding up the pipeline (or implementing multiple pipelines) without increasing memory bandwidth would not result in a significant speed-up of the calculation. However, increasing the memory bandwidth (which is done by increasing the memory clock in simulation here) has a much larger effect. When a standard FPGA board is used the memory interface to the DDR-SDRAM itself is fixed. Thus using caches is the only remaining option. As discussed in the preceding paragraph

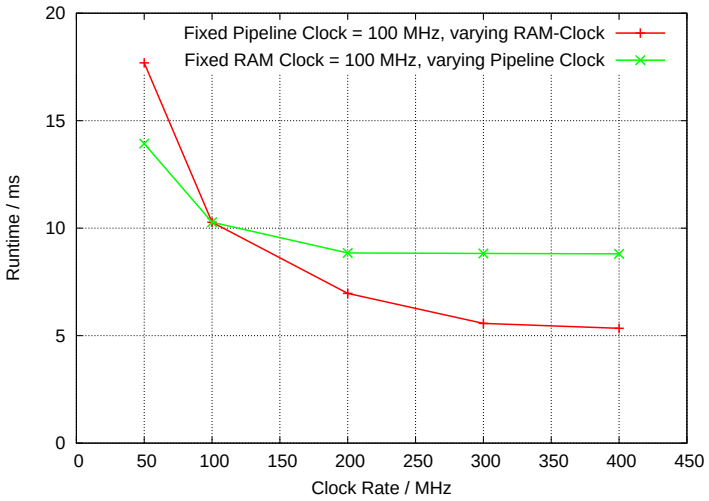


Figure 8.15: Influence of memory bandwidth and pipeline clock on the performance of the collision detection system obtained in simulation. Speeding up the pipeline does not result in a significant speed-up of the overall calculation. However, increasing the memory bandwidth (done here by increasing the memory clock rate of the simulation) has a much larger effect.

there still exists potential for optimising the loading order of DOPs from DDR-RAM that can be exploited using caching techniques.

8.5.2 Comparing Caching Techniques

For cache implementation the main challenge is to determine how to exploit spacial and temporal locality of data for the problem at hand.

Spacial Locality In case of spacial locality it suffices to evaluate the optimal block size. In case of the presented collision detection hardware this is easy, since only complete DOPs are used. Additionally, the memory infrastructure never runs idle in case of a push architecture. Hence, it is obvious that loading complete bounding-volumes in one burst and storing them as a cache block is the optimal way to exploit spacial data locality.

Temporal Locality To most efficiently exploit temporal locality it is necessary to determine the caching strategy best suited for the problem at hand.

Figure 8.16 shows a comparison of the influence of different caching techniques on the performance of a single DOP pipeline obtained in simulation. The span between *No Cache* and *Perfect Cache* indicates the amount of theoretically achievable savings.

The perfect cache saves any data it has ever seen and never needs to replace any of it. Implementing such a structure is somewhat pointless, since it would require a cache of the size of the main memory. It serves as a reference only. The figure shows that for the presented collision detection hardware with only a single collision detection pipeline over 35% speed-up can theoretically be achieved.

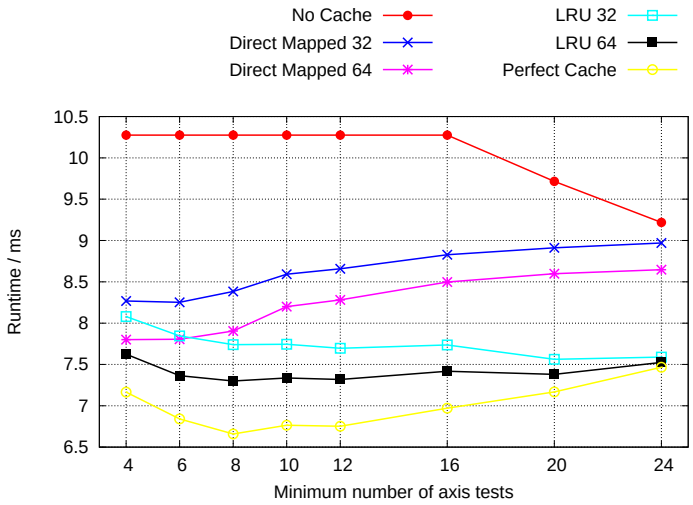


Figure 8.16: Comparison of the influence of different caching techniques on the performance of a single DOP pipeline. Numbers mark the cache size in number of cacheable 24-DOPs.

Also shown are two more realistic caching strategies. Firstly, a fully associative cache with least recently used (LRU) strategy is investigated. It qualitatively behaves like the perfect cache and yields good performance. But, to determine if the currently requested data block is present comparing all cache entries in parallel is necessary. Therefore, it consumes a lot of chip space. Secondly, the very simple direct mapped cache that maps each cache block to a unique entry is investigated. Though very hardware efficient it performs far below the optimum.

LTA Cache

To achieve good performance as well as space efficiency, this thesis presents a novel caching strategy for bounding-volume hierarchies. The basic structure is shown in Figure 8.18.

It implements a two-way set associative caching strategy. This is the minimum necessary to avoid two bounding-volumes to be tested for intersection to request the same cache entry. Additionally, it implements a FIFO that feeds data into the pipeline. This input FIFO is filled with data while the pipeline is processing the minimum number of axes necessary to avoid descending the test tree too quickly. These test requests can now be processed while the triangle intersection test loads its data from the DDR-RAM. This way the cache is able to load data that will be processed at some point in the future. This is done until some test data, which is still scheduled in the FIFO and needs to be processed by the pipeline first, has to be replaced.

To avoid double bookkeeping the FIFO only needs to contain pointers to the cache entries, not the actual data. The only additional overhead are counters of references to the

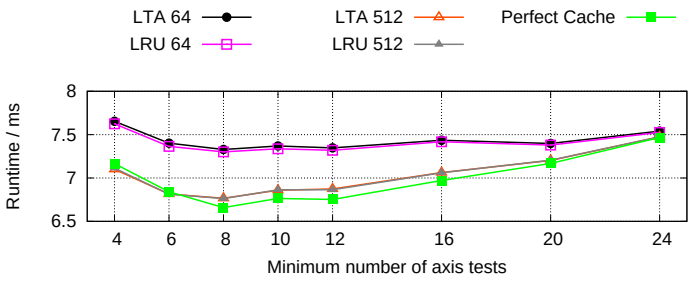


Figure 8.17: Comparison of a fully associative LRU cache and the lockable two-way set-associative cache (LTA) in simulation. LTA and LRU perform nearly equally well. With 512 cache entries both perform close to the theoretical optimum. An LRU implementation of this size would be prohibitively expensive.

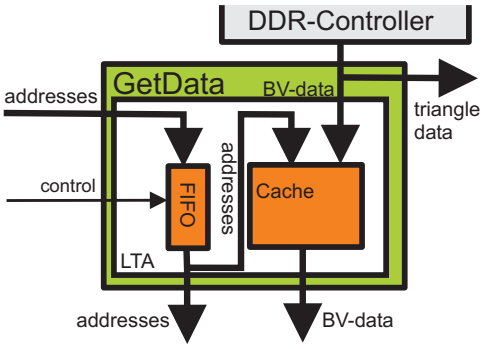


Figure 8.18: Basic structure of the LTA cache. To avoid double bookkeeping a FIFO contains pointers to the cache entries only. The cache itself is two-way set-associative. The cache entries' reference counters are not shown in the figure.

cache entries. If an entry's counter drops to zero the entry can be replaced, otherwise it is "locked". If a new entry needs to replace a locked one, the memory controller still has to be stalled. But this occurs in less than 0.0008% of the cases. Hence, no further speed-up is to be expected by increasing the associativity. Figure 8.17 shows that, though very simple and yet hardware efficient, this lockable two-way set-associative cache (LTA) performs nearly as well as the fully associative approach when providing the same number of entries. Increasing the number of entries to 512 shows that the LTA cache's performance is very close to the theoretical optimum. Implementing a fully associative cache of this size would be prohibitively expensive. The LTA cache's hardware consumption remains very modest, as will be shown in Sec. 8.5.3.

The presented caching strategy obviously collaborates efficiently with the push architecture introduced earlier in this thesis. As long as the FIFO is empty, more test axes are processed by the pipeline. This way the probability to find a separating axis is increased in case of a low bandwidth to the DDR-RAM (possibly temporarily caused by high emergence of triangle tests). This decreases the number of requested DOP data, since in case of separation this subtree of the test tree is not descended any further. This way the memory bottleneck is further relaxed.

8.5.3 Performance Evaluation and Synthesis Results of the LTA Cache

The LTA-cache was implemented in VHDL and synthesised for the target FPGA architecture. Integrating it into the collision detection architecture described in Sec. 8.2 results in an additional resource consumption of only 7% of the slices and 5% of the slice flip flops. Despite performing close to the theoretical optimum it consumes only 33 out of 144 Block RAMs.

Using the LTA cache in conjunction with the push architecture to effectively decrease the number of memory accesses during hierarchical collision detection not only speeds-up calculation by a factor of about 35% when using a single pipeline. It also enables parallel implementation of several pipelines in parallel. Figure 8.19 shows that a design with two pipelines and a DDR-RAM, both running at only 100MHz, is over 10 times faster than a standard PC with a comparable memory bandwidth, while still fitting onto a single FPGA.

8.6 Synthesis and Implementation

To prove applicability and realism of the provided case-study the CollisionChip was refined to synthesisability. It was then synthesised and tested on-chip. Since the target architecture available during the work on this project does not provide the necessary resources primitive test and bounding volume test are implemented and tested independently. This Section discusses the implementation of the bounding-volume hierarchy test. In Sec. 8.7.2 the primitive test subsystem is presented. Both sections provide detailed synthesis and performance results.

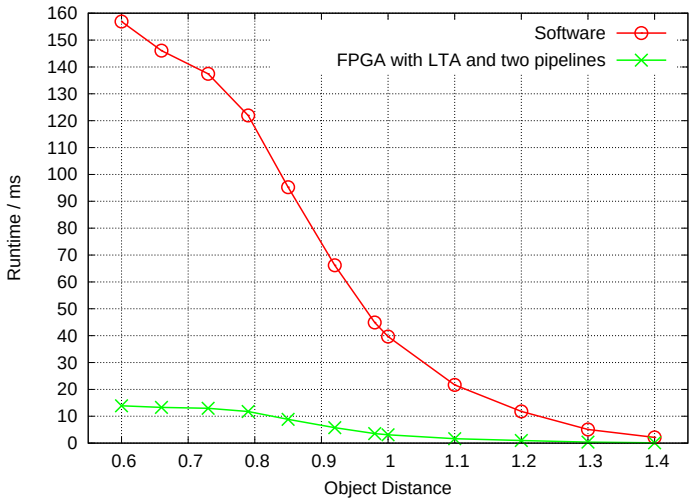


Figure 8.19: A design with two pipelines and a DDR-RAM, both running at only 100MHz, and implementing an LTA cache in conjunction with the push architecture is more than 10 times faster than a standard PC with a comparable memory bandwidth.

As already mentioned in Sec. 8.4 the target architecture is a Xilinx Virtex II (XC 2V6000, speed grade -4) on an Alpha Data ADM-XRC-II board with 256 MB DDR-RAM at 100MHz connected via a 64 bit wide bus.

Synthesis, Place, Route and Mapping were done with Xilinx ISE 8.0.

The individual components `controller`, `getData` (including the LTA cache), `pipe_data` and `pipeline` allow clock frequencies between 95MHz and 120 MHz according to Xilinx XST. If these components are combined, the overall design runs at only 10MHz. Intensive investigations have shown that this results from two different, negatively interacting effects. Firstly, an early optimisation on protocol level leads to very demanding timing constraints. Whenever the controller requests new data, `getData` notices this request and acknowledges it by reporting itself as busy. One clock cycle passes from the data request to noticing this busyness in the controller if both components react only on positive clock edges. This not only wastes one clock cycle, but more importantly, complicates controlling. To avoid requesting a new dataword during this cycle it demands that the controller keeps an account of its previous action.

A simpler method is to trigger `getData` on the negative clock edge. This enables it to report its busyness earlier and the controller receives this acknowledgement in the consecutive clock cycle. Still, this scheme demands that processing the request and acknowledging it is done in half a clock cycle. Thus it requires nearly optimal placing-and-routing to provide high clock frequencies.

Secondly, the DDR controller contained in `getData` demands very strict placing and timing constraints. This deprives the placing and routing tool Xilinx PAR of multiple degrees of freedom in locating the subcomponents. These conflicting demands cannot be satisfied concurrently and hence result in the low frequency.

Nevertheless, technically it is no problem to change `controller` and `getData` in a way that the additional clock cycle is spent to improve overall timing. This local intervention will solve the problem effectively. Moreover, in conjunction with the LTA cache it will leave the number of clock cycles spent for a complete hierarchy traversal nearly completely unaffected. Still, this is tedious, error-prone and time consuming work, which will not result in any additional scientific insight. As discussed earlier this dissertation does not aim to result in a collision detection hardware fit for production, but exclusively uses it as a real world case study. Therefore this optimisation is left to future projects.

8.7 The Primitive Test Subsystem

8.7.1 Triangle Intersection Test Review

The investigations presented in the previous sections clearly show, that the effective speed of hierarchical collision detection is directly determined by the speed of the memory access. Since, bounding volume and triangle intersection test need to share memory bandwidth, it is vital to choose a triangle test that has modest bandwidth demands.

But not only the memory interface needs to be shared by the two stages of the hierarchical design, but also the available hardware resources. Therefore, the triangle test needs to be as modest in its hardware consumption as possible.

Since in the presented work the major goal is to exchange two primitive tests against each other in simulation, it is also necessary to feature a second primitive type. Implementing most complicated intersection tests for complex primitives is far beyond the scope of this doctoral thesis. Therefore the second primitive used is the quadrangle, as was already discussed in Sec. 7.5. Since most triangle intersection tests can be generalised to quadrangles, little additional development overhead is imposed.

In [85] various triangle intersection test algorithms are discussed. Some can be discarded without in-depth analysis, which is discussed in the next Subsection 8.7.1. Some of them will be analysed with respect to their suitability for hardware implementation in Subsection *Triangle Transformation*, *Common Line Intervals*, and *SAT for Triangle Intersection Testing*. Subsection *Choice of Triangle Intersection Test* discusses which of the reviewed intersection tests is best suited for implementation in the CollisionChip.

2×2 Linear Equation System, Configuration Space and Determinants Approaches

The 2×2 *Linear Equation System* algorithm proposed by [Badouel90] tests edges of one triangle against the supporting plane of the other. Therefore a 2×2 linear equation system is solved to determine the barycentric coordinates of a possible intersection point. This is basically the same idea as it is realised in an optimised form in the triangle transformation algorithm, which was already discussed in Sec. 7.6.2 and will be analysed in the next Subsection *Triangle Transformation*.

Configuration space approaches are omitted in [85] even for software based intersection testing, because of their high dimensionality and their complex nature. Thus they are not taken into account here as well, to avoid a shift of focus.

Triangle Transformation

A hardware implementation of the triangle transformation algorithm was presented in [8, 59] and already discussed in Sec. 7.6.2. Hence only its costs are calculated here to enable an overall comparison.

Initially both triangles are transformed (Eq. 7.10, page 100), so that one is mapped into the unit unit triangle. Since perspective projections are not required in the given context, this transformation can be performed using 27 multiplications and 27 additions. Checking three triangle edges for intersection with the xy-plane needs 6 comparisons. Processing the criterion Eq. 7.19–7.21 for all edges consumes $3 \cdot 4 = 12$ multiplications, $3 \cdot 4 = 12$ additions/subtractions and $3 \cdot 3 = 9$ comparisons. Multiplying by $\text{sign}(r_z)$ is practically free.

This makes an overall of 39 multipliers and 48 adders if comparisons and subtractions are implemented using adders.

Sequencing the test into 3 triangle-line intersection tests results in 31 multipliers. Since in this scheme some vertex coefficients are sign checked multiply 38 adders are utilised.

The aforementioned variants always need to load both triangles and if the vice-versa check is performed also both transformation matrices. In our target architecture only 128-bit data words can be loaded. Using 32-bit matrix and triangle coefficients a total of 11 datawords needs to be loaded in the worst case.

If the test is performed vice-versa in parallel the according total of multipliers and adders doubles.

[85] details how this approach can be generalised to quadrangles.

Common Line Intervals

The common line interval approach was proposed in [44] and is commonly known as the "Möller test". Since it is a very well known approach it is only outlined in the following.

Initially, the distance of both triangles to the supporting plane of the other triangle is calculated. If one is greater then zero the triangles obviously do not intersect. Now the basic idea of the "Möller test" is to calculate the line parameters of the intersection points of both triangles on the common line of the planes. The line intervals of the triangles intersect if and only if the triangles intersect. Figure 8.20 illustrates the triangles and their supporting planes.

Exploiting the invariance of these parameters against translation and projection the line itself does not need to be determined explicitly for this. Note that in a hardware implementation this would need to be thought over in depth. Simplifications such as these usually lead to numerical instabilities in some rare scenarios. In software they can be identified and treated separately. Implementing such a special treatment in hardware comes at tremendous extra cost, thus it is generally preferential to choose the numerically stable solution right from the start. But since this is one of the basic ideas of the approach the following will assume that it is not an issue and approximate the resource consumption for the simplified algorithm.

[85] details an optimisation of the original algorithm, that allows implementation of

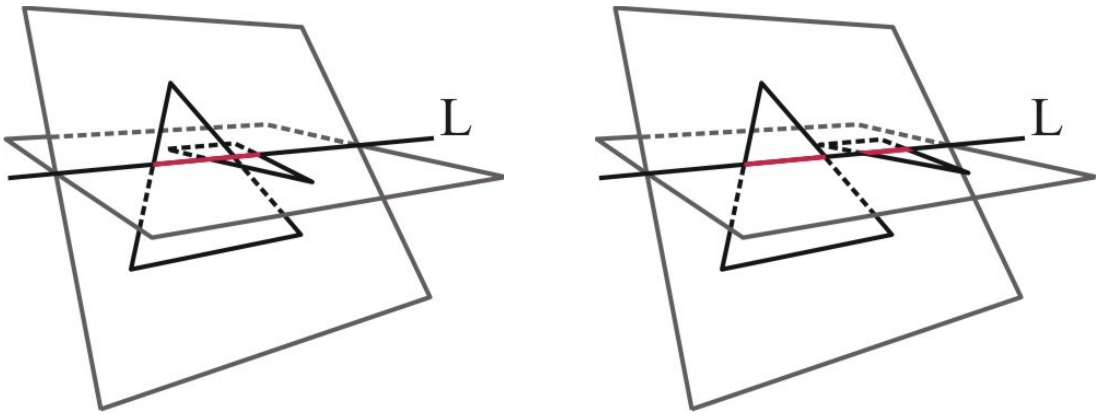


Figure 8.20: Due to earlier tests we know that the distance of both triangles to the supporting plane of the other triangle is greater than zero. Then the line intervals of the triangles intersect if and only if the triangles intersect.

the approach without any divisions. The target architecture does not provide any divider hard-cores. Since implementing a soft-core division is prohibitively expensive only this variant is taken into account. It requires 60 multipliers and 66 adders if the triangles' normals are loaded from memory. In conjunction with the triangle data a total of 7 datawords à 128 bit results. If the normals are calculated on-chip, both, the multiplier count and the number of adders increases to 72, while the number of datawords to load is decreased to 5. The approach can be generalised to quadrangles straight forward.

SAT for Triangle Intersection Testing

As discussed in Sec. 7.4 the separating axis test is a general way to test two convex polytopes for intersection. Despite the fact that triangles are 2 dimensional objects embedded in 3D they can be interpreted as (degenerated) convex polytopes. In case they are in general position it suffices to test 11 axes according to the separating axis theorem [26]. These are the normals and the cross edges.

Let T_A and T_B be triangles with vertices $V^0, V^1, V^2 \in \mathbb{R}^3$, and $W^0, W^1, W^2 \in \mathbb{R}^3$ respectively. Again wrap-around indexing is used for the vertices. Let k, l$\epsilon\{0,1,2\}$. Normals and cross-edge-axes are then calculated using

$$\begin{aligned} n^A &:= \left(V^1 - V^0\right) \times \left(V^2 - V^0\right) \\ n^B &:= \left(W^1 - W^0\right) \times \left(W^2 - W^0\right) \\ C_{k,l} &:= \left(V^{k+1} - V^k\right) \times \left(W^{l+1} - W^l\right) \end{aligned} \tag{8.24}$$

Then the triangles' points are projected onto any test axis L out of this set by

$$\begin{aligned} p_{V^k} &:= V^k * L \\ p_{W^l} &:= W^l * L \end{aligned} \tag{8.25}$$

The projection intervals on the axis are

$$\begin{aligned} I_A &:= [\min\{p_{V^0}, p_{V^1}, p_{V^2}\}, \max\{p_{V^0}, p_{V^1}, p_{V^2}\}] \\ I_B &:= [\min\{p_{W^0}, p_{W^1}, p_{W^2}\}, \max\{p_{W^0}, p_{W^1}, p_{W^2}\}] \end{aligned} \tag{8.26}$$

Due to definition 8.24 every test axis is orthogonal to at least two triangle edges. Thus the projections of the two triangle points defining this axis are identical. This can be used to reduce the number of necessary projections and comparisons.

Further simplifications, e.g., projecting the test axis L_i onto the coordinate axis a most parallel to it ($a = \arg\max_i\{L_i\}$), are avoided. They either lead to numerical instabilities or increase the resource consumption.

To avoid coplanarity problems the triangles can be extruded to 3 dimensional quadrihedra of thickness ε by treating the cross products of normals and edge orientations as additional normals [22, 23]. This results in 6 additional axis tests and is a very simple way to consider coplanar triangles as well. Moreover, it comes with almost no extra development effort.

Since the SAT approach is applicable to all convex polytopes it can be applied to quadrangles as well.

This approach can easily be divided into sequences with identical calculations. One for each axis. Thus in the worst case it takes as many clock cycles to process the overall result as axes are considered. It then consumes 6 multipliers and 9 adders for generation of the axis under test. 3 multipliers and 2 adders for projecting one triangle point onto this axis are needed. Since only 4 points need to be projected this totals to 12 multipliers and 8 adders. Comparing the resulting intervals consumes 2 more adders.

If all axes are tested concurrently there is not much information, that could be reused. Thus in this case the hardware consumption can simply be multiplied by the number of considered test axes.

Choice of Triangle Intersection Test

Table 8.1 summarises the hardware consumption of the previously discussed triangle intersection algorithms. Since the target architecture can load 128-bit datawords exclusively, the number of memory accesses is considered instead of the de-facto data size. As can be seen in the table the triangle transformation algorithm has the most modest hardware consumption if a full test is to be performed each cycle. But since the target architecture is incapable of loading the necessary data fast enough, such an implementation is a tremendous waste of resources. Additionally, it has the greatest demand of data to be loaded and thus would de-facto result in the slowest query. The architecture would be waiting for data permanently.

The algorithm with the smallest resource consumption is the SAT approach if only a single axis is tested at a time. It also requires only 5 datawords to be loaded and thus can be expected to interact well with the bounding volume test. Since both tests compete against each other in memory usage this is a very important issue.

Name of Test	Multipliers	Adders	128-bit Datawords	Supports Coplanarity	Quadrangle Extension Available	Run-time in Clock Cycles
Triangle Transformation						
double, parallel	78	96	11	–	√	1
single, parallel	39	48	11	–	√	2
double, sequential	62	76	11	–	√	3
single, sequential	31	38	11	–	√	6
Common Line						
loading normal	60	66	7	–	√	1
calculating normal	72	72	5	–	√	1
SAT						
1 axis per cycle	18	19	5	increased run-time	√	11 / 17
11 axes per cycle	198	209	5	–	√	1
17 axes per cycle	306	323	5	√	√	1

Table 8.1: Comparison of triangle intersection test algorithms with respect to hardware consumption.

The common line interval approach does not seem to compete in hardware implementation with the other two approaches discussed. The least that can be said is, that parallelising it is not straight-forward and it would demand high development effort to decrease its hardware consumption.

If integrated into the overall design for every triangle pair that is tested at least two k-DOPs need to be loaded before. Since 24-DOPs with 32-bit coefficients are used they need 7 memory accesses to be loaded. Considering its own loading time the sequential SAT approach can complete its calculation until the next pair of triangles will be available on-chip if 11 axes are tested. Hence, it is the best alternative to choose the smallest and yet most bandwidth efficient intersection test for the implementation. Additionally, it is the approach that can be extended most easily to consider triangles that are not bound to be in general position.

8.7.2 Integrating the Primitive Test into the Overall Design

Untimed Functional Implementation of SAT for the Primitives

Since the bounding volume test was already refined to RT-level, it is necessary to insert adaptors to enable usage of a high-level interface to the primitive test. The input adaptor is split in two: The implementation of the communication protocol and a serial-to-parallel converter, which re-groups the data-packets provided by `get_Data` into complete triangle data. The output adaptor only needs to provide the low-level protocol between result module and primitive test.

Both low level interfaces are not detailed here, since they are of no further relevance and are straight forward. Still, there is one exception. Since the primitive intersection changes during run-time, the size of the primitive, that needs to be loaded from memory changes as well. This was already respected in the design of the `get_Data` module. Therefore it is necessary that the currently active primitive test can inform `get_Data` of the size of the primitives in use. This is done by the signals `prim1_size` and `prim2_size`, which contain the number of memory accesses necessary to load the according primitive. Although this might appear artificial at first, it is a quite common design even in static architectures. In nearly any architecture different types of data are loaded from memory and thus the memory controller needs to be informed of the data volume it needs to transmit.

The primitive tests can be split in two: application of the matrix specifying the relative position of the objects and implementation of the intersection test itself as it was described in Sec. 8.7.1.

Figure 8.21 shows the primitive subsystem for triangle objects after these preparations were applied. Both, the transformation module and the primitive test are straight forward software implementations using object oriented implementation techniques.

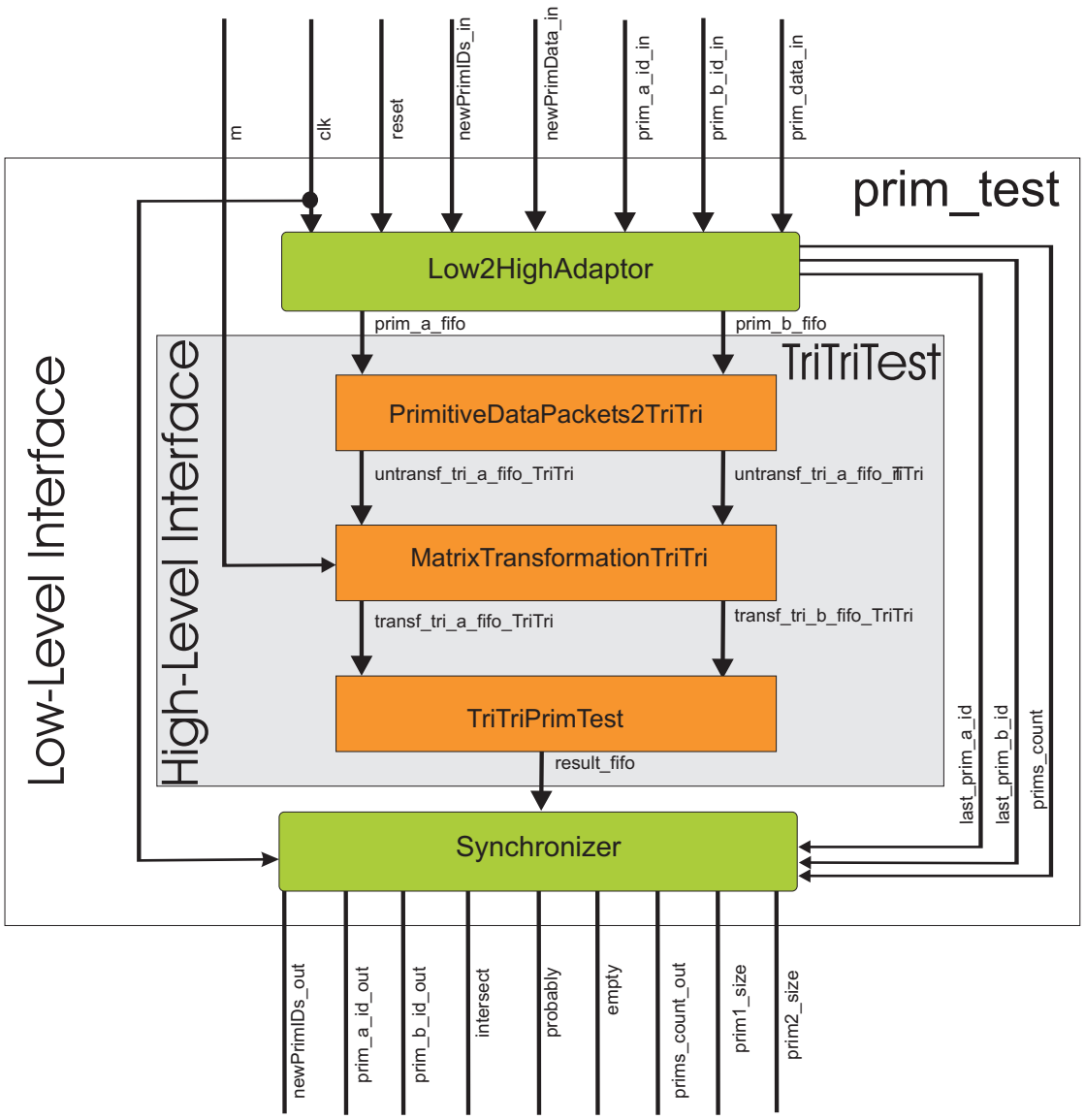


Figure 8.21: Using adaptor insertion enables the UTF implementation of the primitive test to provide a low level interface to the hierarchical collision detection design. The primitive test itself is implemented using object orientation.

Primitive SAT on RT-Level

To provide as much realism as possible, the primitive tests were refined to RT-level. Since floating-point arithmetic is prohibitively expensive in this context the calculations where implemented in fixed-point precision. This might lead to both, false positives and false negatives. This can be avoided using very similar rounding techniques as were proposed in Sec. 8.1.2 and is subject to current investigations. Since it is of no real relevance in the context of reconfiguration, this issue is not discussed in the following.

Figure 8.22 shows the primitive test subsystem on RTL. The transformation module `Tri_Pipeline_MTansf` sequentially transforms each point of the first triangle into the reference frame of the second one according to transformation matrix m and propagates them to the pipeline controller. The latter one keeps track of the current test axis. This information is passed to the primitive intersection test pipeline along with the primitive data. The test pipeline outputs the number of the currently processed test axis, how many axes are still to be processed, whether the primitives' projection intervals intersect on this axis and if the current output is valid. The pipeline implementation is detailed in the next Subsection *The Intersection Test Pipeline*. The result output adaptor collects the intersection results of the individual axes and combines them into an overall result.

The Intersection Test Pipeline Figure 8.23 shows a simplified illustration of the triangle-triangle intersection test pipeline. Here the test axes need to be generated according to Eq. 8.24 (page 138). Therefore, the triangle points used for the generation of the current axis are chosen. Since the test axes are generated from triangle points, there are always two pairs of points whose projections are identical. Thus only four out of the six triangle points need to be projected per axis test. Since they differ according to the axis under test they also need to be multiplexed. This is done in the first macro stage of the triangle pipeline (see Figure 8.23).

In the pipeline's second macro stage the test axes are actually generated from the points singled out in the previous stage.

The third stage implements 4 vector multipliers used to project the four triangle points onto the test axes.

Stage four compares the results. Due to the projection of only four points, the line intervals need to be generated differently for different axes. This is done with respect to the current test axis.

The implementation of the axis generator and the vector multiplier modules are straight forward hardware implementations of the according equations. They are further divided into ten substages each to maximise pipeline throughput.

Generation of the remaining signals depicted in Figure 8.22 is very straight forward and is left out here for clarity of presentation.

The implementation of the triangle-quadrangle and the quadrangle-quadrangle tests are very similar to the implementation of the triangle-triangle intersection test pipeline and are also left out.

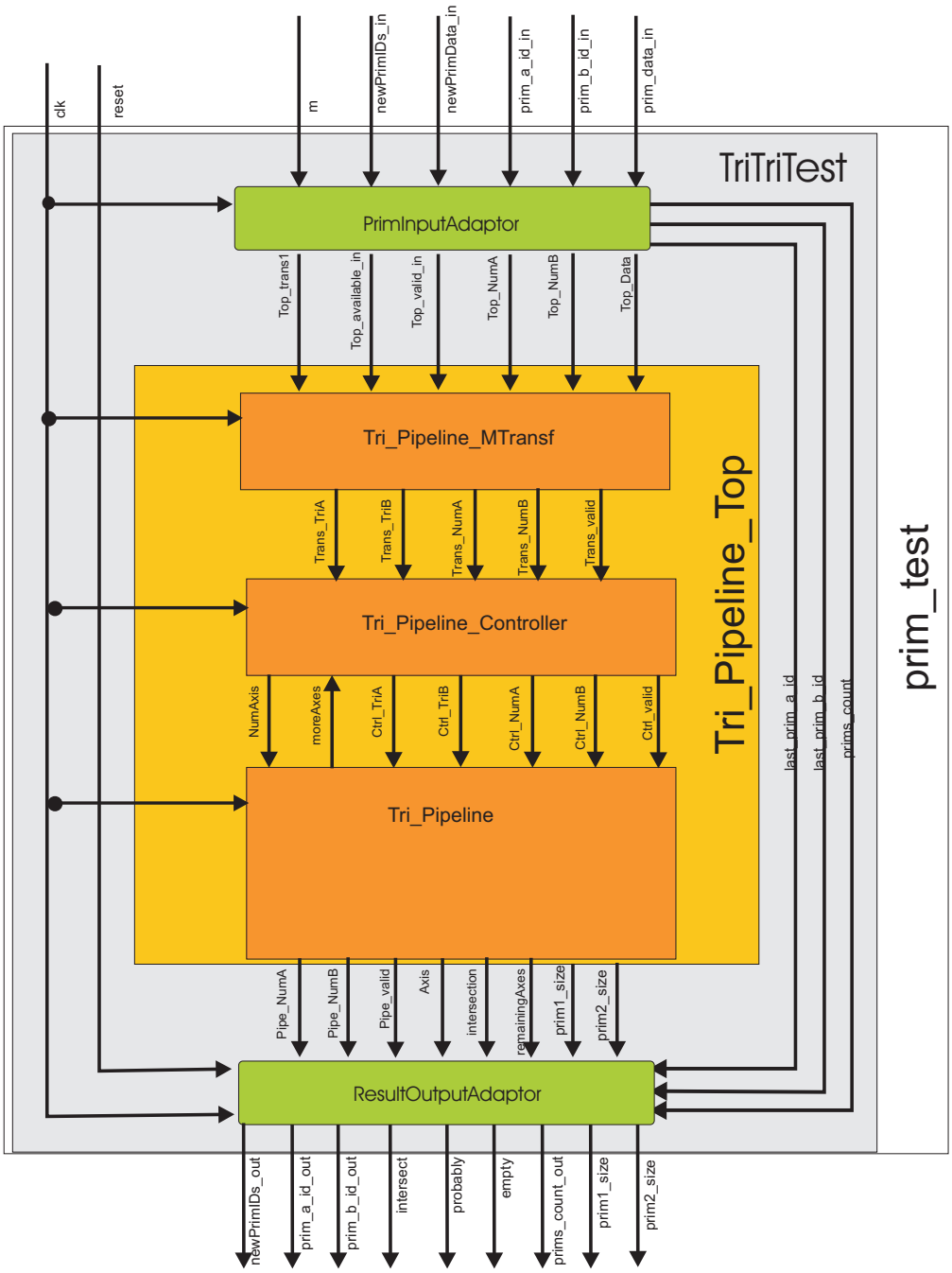


Figure 8.22: The primitive test subsystem on RTL.

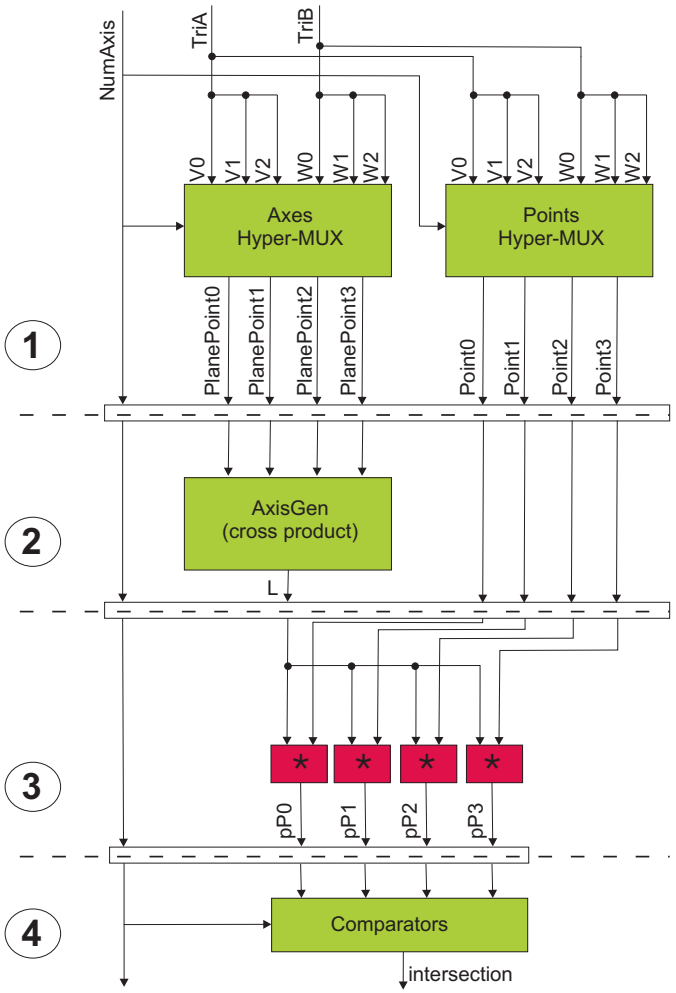


Figure 8.23: A simplified illustration of the triangle-triangle intersection test pipeline. The triangle data is pre-processed in the first macro stage in order to enable optimised generation of the test axes in the second and projection of the triangles in the third macro stage. In stage four the projection intervals are compared.

Synthesis Results of the Primitive Intersection Test The primitive subsystem was implemented in 32-bit and in 18-bit fixed-point arithmetic. It was implemented, synthesised and tested, both, in SystemC and VHDL. The VHDL 32-bit implementation of the triangle-triangle intersection test consumes 9% of the available Flip-Flops and 132 18-bit multipliers. Using 18-bit precision, the multiplier consumption drops below 50%. This enables co-implementation with the hierarchical bounding-volume test.

The slowest primitive test is the quadrangle-quadrangle test, which runs at 40.2 MHz. Here the multipliers are the limiting element, which could be eliminated by using pipelined multipliers as was done for the bounding-volume test (see Figure 8.5, page 122). But since the bounding volume design runs at rates below this, further optimising the primitive test would not improve the overall performance.

Part IV

Putting It All Together

9 Applying ReChannel To CollisionChip

This section presents a case study of how ReChannel can be applied to the real world problem of hardware accelerated collision detection. A full refinement of a triangle-triangle intersection test algorithm, which is exchanged against a triangle-quadrangle or a quadrangle-quadrangle test during run-time is presented. These tests provide the second stage of a hierarchical collision detection hardware accelerator. The overall design is simulated on all levels of abstraction featured by the classical SystemC design flow. The system's reconfiguration aspects are described and simulated using ReChannel. Thus the effectiveness of the ReChannel refinement methodology proposed in Sec. 6.4 is demonstrated.

The primitive tests were developed with the single purpose to be reconfigured. But the refinement solely concerns the interfaces and the reconfiguration controlling since these are the only parts of the design that should be affected by the introduction of reconfiguration. One of ReChannel's main objectives is the support of closed source

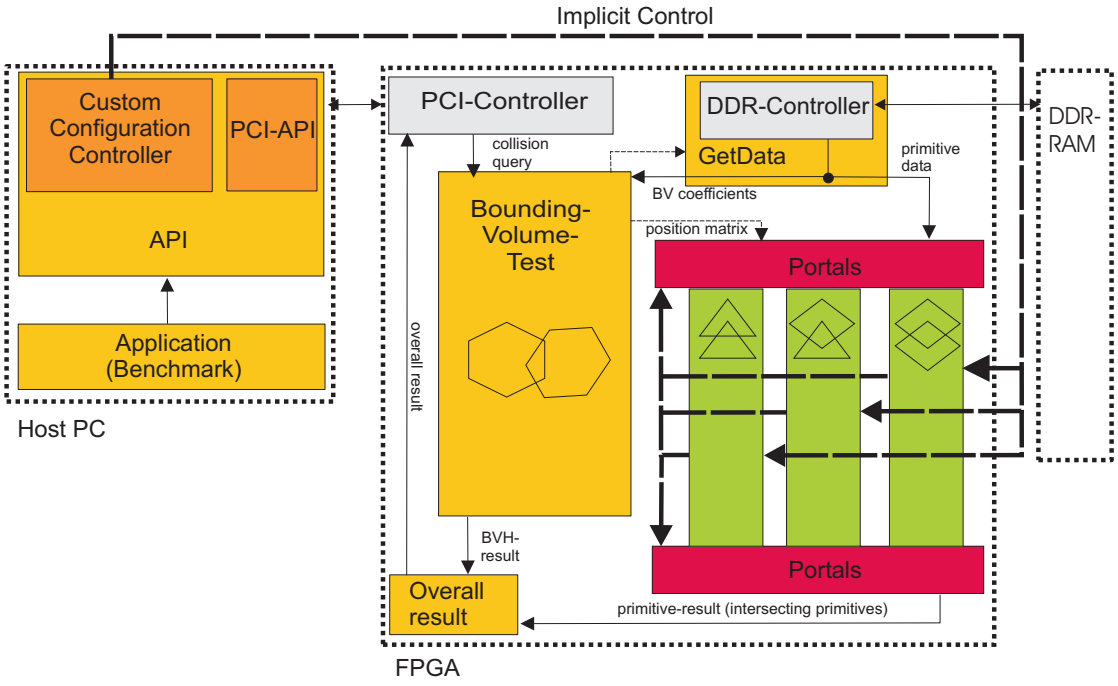


Figure 9.1: Overall design of the CollisionChip simulation using dynamic reconfiguration to enable exchange of primitive tests. The primitive tests are decoupled from the static design by portals.

IP components provided by a third party. Therefore the primitive test designs will be treated like an IP core of which only the interfaces are known to the user in the following.

Figure 9.1 gives an outline of the reconfigurable design as it is to be developed. The benchmarking application controls the collision detection accelerator by an API, which itself contains a PCI-API and a custom configuration controller (CCC).

If a collision query of two objects constructed of e.g., triangles is requested, the API instructs the CCC to configure the triangle-triangle intersection test onto the FPGA. The CCC checks if this test is already configured and if it is not the CCC initiates a reconfiguration. This is to be simulated using ReChannel language constructs, and thus the primitive tests are decoupled from the static part of the design by portals. The latter are implicitly controlled by the CCC.

The PCI-API transports the object data (root nodes, transformation matrix and test axes of the BV-test) to the PCI-Controller on-chip and thus, starts the intersection test. If the test is completed the overall result is also reported to the host via the PCI communication.

Since hardware design is an error-prone task, it is uncommon to implement a design directly in a synthesisable fashion. Instead, it is refined from a very software like implementation into a synthesisable architecture. The SystemC refinement methodology was already discussed in Sec. 5.1.1 and extended with the ReChannel features in Sec. 6.4.

The hierarchical bounding-volume test and the primitive tests were refined individually prior to the incorporation of reconfiguration. The results of this refinement process (e.g., determining the number of test axes, improving the memory interface etc.) are presented in Sec. 8. The subsequent sections will apply the extended refinement methodology to the CollisionChip architecture to extend it with reconfigurable primitive intersection tests.

9.1 Untimed Functional Level

9.1.1 Reconfigurable Topology

First of all the reconfigurable modules need to be constructed from the static ones. For this, according modules are derived as proposed in Sec. 6.1.2 and Sec. 6.2.1. The original static modules have many template parameters to enable usage in different contexts and thus can be reused. They basically enable simulation of calculation in different fixed-point precisions. Therefore the modules need to be explicitly derived instead of utilising the according macros. Listing 9.1 shows the derivation code. This is done for the triangle-quadrangle and the quadrangle-quadrangle test accordingly.

These modules can now be instantiated instead of their static counterparts (see Figure 9.2) and can be connected to portals. Since the design is modelled on UTF level as a Kahn Process Network, the primitive test exclusively uses FIFOs to communicate to the static design. Figure 9.3 shows the binding of the output FIFO. The reconfigurable topology depicted in Figure 9.1 results, while only minimal changes had to be made to

```
template
 < typename SAT_INTERN_TYPE,
   class ID_TYPE, int PRIMITIV_DATA_PACKET_BIT_WIDTH,
   unsigned POSITIONS_AFTER_DECIMAL_POINT, int POINT_BIT_WIDTH,
   int MAT_ENTRY_BIT_WIDTH, unsigned MAT_POSITIONS_AFTER_DECIMAL_POINT>

class TriTriTestTF_rc
 : public ::ReChannel::rc_reconfigurable_module<TriTriTestTF<SAT_INTERN_TYPE,
    ID_TYPE, PRIMITIV_DATA_PACKET_BIT_WIDTH, POSITIONS_AFTER_DECIMAL_POINT,
    POINT_BIT_WIDTH, MAT_ENTRY_BIT_WIDTH, MAT_POSITIONS_AFTER_DECIMAL_POINT> >
{
public:
TriTriTestTF_rc(const sc_module_name& name)
 : rc_reconfigurable_module<TriTriTestTF<SAT_INTERN_TYPE,
    ID_TYPE, PRIMITIV_DATA_PACKET_BIT_WIDTH, POSITIONS_AFTER_DECIMAL_POINT,
    POINT_BIT_WIDTH, MAT_ENTRY_BIT_WIDTH, MAT_POSITIONS_AFTER_DECIMAL_POINT> >
           (name) { }
};
```

Listing 9.1: Definition of a reconfigurable triangle test module type. Since the original static module has template parameters explicit derivation is used instead of macro utilisation.

```
[...]

    // Instantiation of the reconfigurable primitive intersection tests
TriTriTestUTF_rc<[...]>   tri_tri_test_UTF_rc;
TriQuadTestUTF_rc<[...]>  tri_quad_test_UTF_rc;
QuadQuadTestUTF_rc<[...]> quad_quad_test_UTF_rc;

    // Instantiating the result channel and its portal
sc_fifo< IntersectionResult* > result_fifo;
rc_fifo_out_portal< IntersectionResult* >  result_fifo_out_portal;

[...]
```

Listing 9.2: Instantiation of the reconfigurable primitive test modules and one of the portals they use to communicate. The template parameterisation is left out here for clarity of presentation.

```
[...]

    // Bind channel to static end of the portal
result_fifo_out_portal.static_port(result_fifo);

    // Bind modules' ports to portal
result_fifo_out_portal.bind_dynamic(tri_tri_test_UTF_rc.result_fifo_out);
result_fifo_out_portal.bind_dynamic(tri_quad_test_UTF_rc.result_fifo_out);
result_fifo_out_portal.bind_dynamic(quad_quad_test_UTF_rc.result_fifo_out);

[...]
```

Listing 9.3: An example of binding a portal to a static channel and multiple ports.

the original design.

9.1.2 Reconfiguration Control

To control the configuration state of the design, i.e., which primitive test is currently active, a configuration controller needs to be integrated. Which test needs to be configured onto the hardware needs to be decided only if a new collision query is requested for a pair of objects. Thus it is intuitive to integrate configuration control into the API. As was discussed earlier, the CCC can be interpreted as a bus with varying access times and thus is modelled as a hierarchical channel. Since the PCI-communication is a bus anyway, and the API basically consists of these two components, the latter is modelled as a hierarchical channel itself.

On UTF level the CCC's only functionality is to activate the correct intersection test. For this it needs to decide, which intersection test is needed, depending on the objects' primitive types, if the correct test is already active, and if it is not, to load and activate it accordingly.

To manipulate the modules' reconfiguration states an instance of type `rc_control` is used and the primitive tests are registered with it. `sc_find_object` is used to request the intersection test module's pointer. Since `rc_control`'s methods implicitly cast to a reconfigurable set, single modules need to be casted to reconfigurable objects explicitly. Listing 9.4 shows the complete registration process of the triangle-triangle test with the `rc_control` instance `ctrl`.

Figure 9.2 shows the API and its interaction with the CollisionChip as it was implemented.

9.1.3 Synchronisation

Contradicting the name *untimed* functional level, timing is not completely irrelevant here. While the exact delays of the modules in use can be neglected, the sequence of events cannot. Thus it is necessary to decide when a reconfiguration is allowed to take

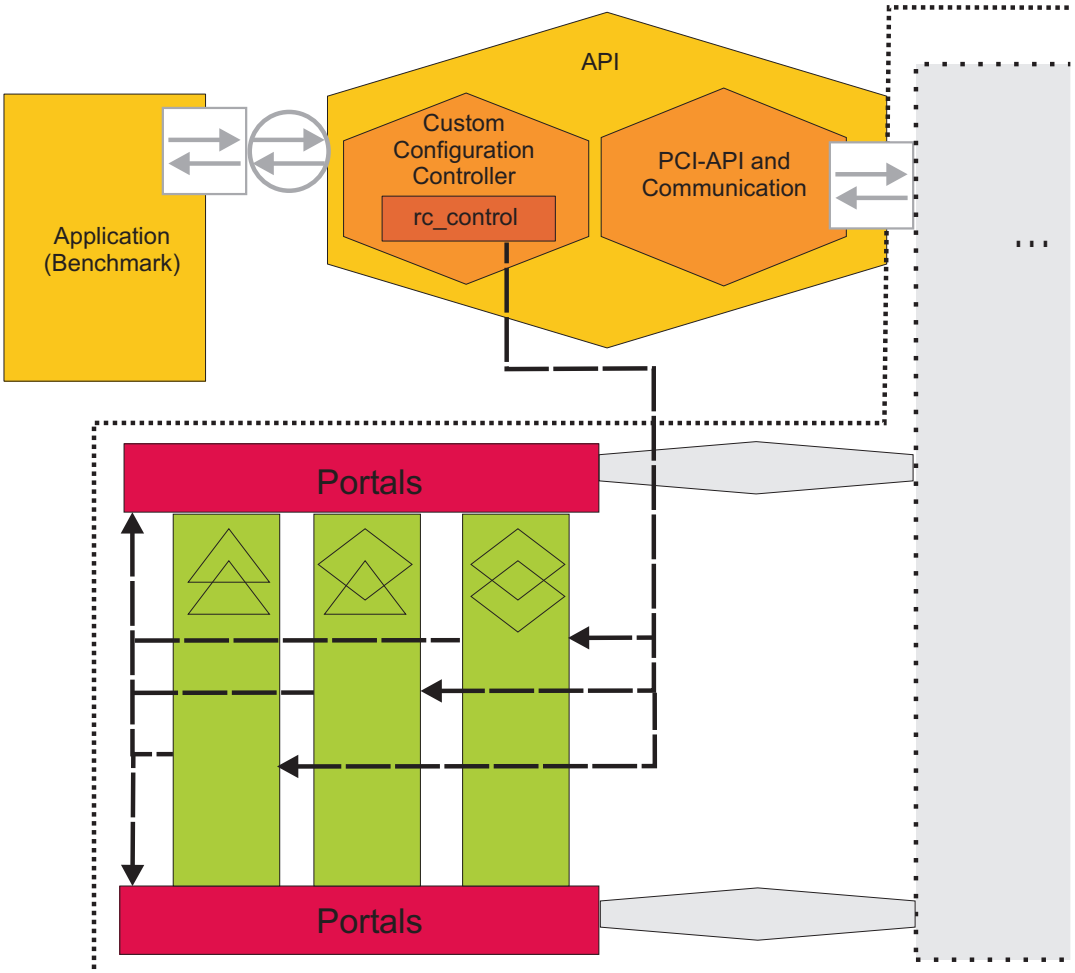


Figure 9.2: The API and its interaction with the CollisionChip as it was implemented. The API basically consists of two components: The PCI-API and the custom configuration controller. The PCI-API is a purely functional implementation of a PCI Communication. The CCC is implemented using ReChannel language primitives, i.e., instantiating `rc_control`. See Listing 9.4 for detail.

```
[...]

  // Request pointer of intersection test module
  // and assert that a module was really found
sc_object *obj = sc_find_object(Sim.priTt.tri_tri_test_TF_rc);
assert( obj!=0 && "Find object failed. No triangle-triangle-test found.");

  // Cast to rc_reconfigurable and check if object is really of type rc_reconfigurable
rc_reconfigurable *triTest = dynamic_cast< rc_reconfigurable* >( obj );
assert( triTest!=0 && "Dynamic_cast from *sc_object to *rc_reconfigurable failed .");

  // Register test with rc_control-instance ctrl
ctrl.add(*triTest);

[...]
```

Listing 9.4: To manipulate the modules' reconfiguration states an instance of type `rc_control` is used and the primitive tests are registered with it. Since `rc_control`'s methods implicitly cast to reconfigurable set, single modules need to be casted to reconfigurable modules explicitly. The primitive test is registered with the `rc_control` instance `ctrl`.

place. In the CollisionChip framework a primitive test can be triggered at any time, while the hierarchical bounding volume test is in operation. On the other hand it is necessary to change the primitive test present on the FPGA only if a new collision query of two objects is requested, i.e., not during the hierarchy traversal. This enables a very simple controlling scheme, as it was discussed in the preceding Sec. 9.1.2.

Still, it demands usage of synchronisation filters for the reconfigurable modules. Since the currently active primitive test is connected to the static design parts via a SystemC FIFO and it is not explicitly designed for reconfiguration, it will proceed to the next `read()` statement, when the last test of the object pair currently under test has finished. Actually it does not even "know" that this test was special in some way. Since there will be no further primitive tests, the current collision query was completely processed, the result is reported back to the host application, which now might initiate a new test. If this test requires a reconfiguration, the CCC will initiate it accordingly.

This course of events will inevitably cause errors in the processing sequence, since the last primitive test is still blocked within the `read()` statement. Therefore the accessor implemented FIFO filter is used, that was described in Sec. 6.2.4. It prevents the access from blocking within the FIFO by calling a specially prepared `wait()` method, and thus blocks itself instead. This `wait()` enables resetting and thus deactivation of the module as proposed in Sec. 6.2.5. Using this mechanism effectively allows deactivation of the module whenever all FIFOs it is connected to are empty. This comes at practically no extra development effort. Any developer not familiar with ReChannel's internals would not notice the implicit usage of a filter at all. Note that although this might often

be the case, the designer will have to define filters for himself in general settings.

9.2 Timed Functional Level

When the refinement proceeds to timed functional level the design's topology remains unchanged. This refinement step only concerns its timing behaviour and thus the primitive tests are extended with delays.

In the CollisionChip context this basically accounts to an explicit modelling of a module latency caused by the pipeline stages described in Sec. 8.7.2 and their sub-stages This latency is modelled explicitly by delaying the result of the high-level test already used on the previous refinement level, by a parameterisable number of clock cycles.

Experimental results show, that this latency does not have any practical effect on the overall timing behaviour of the design. This is an expected result, since the primitive intersection is tested concurrently to the hierarchical bounding volume test and the result of the primitive test does not influence the hierarchy traversal at all. Solely the primitive tests at the end of the overall calculation add a negligible delay to the overall time consumed by the object test.

Already using clock cycle counts and thus providing a cycle accurate simulation on TFL is not refining "by the book". But it does not change the outcome in any way to express the design's delay in nanoseconds instead of cycles. Moreover, this way a mixed level description results, which is far more often the case in realistic scenarios then a strictly separated simulation on different refinement levels.

9.3 Transaction Level

When it comes to transaction level, two different aspects of the refinement need to be distinguished: Refinement of the controller, which enables to interpret reconfiguration as a bus (see Sec. 6.4), and the refinement of a design's bus interfaces.

Probably, there is no architecture, that covers all aspects of system design. Thus it is not possible to provide a case study testing and illustrating all uses of ReChannel. Using CollisionChip as an example as was done in the previous sections, there are no buses in use, neither in the primitive test's interface nor anywhere else in the design. Solely the `getData` module could be interpreted as some kind of arbitrated bus, but due to its central role in the system's controlling and its cache functionality this appears a bit far fetched.

Since there are plenty of specialised portals already implemented within the ReChannel library and their use is demonstrated both, in previous and subsequent sections, there is nothing to be learned from any artificial toy problems as well.

A most interesting extension of ReChannel would be to provide special support for channels of the TLM library. But since this is an extension library of SystemC itself it is not considered in this work either. Thus no transaction level experiments concerning the refinement of bus interfaces are provided here.

The following Subsection *Communication Latency* and Subsection *Reconfiguration Delay* discuss the controller refinement.

9.3.1 Communication Latency

The target hardware communicates with the design via the PCI bus. For this the PCI controller chip on-board forwards the data via the local bus to the FPGA where a vendor supplied PCI controller module is used in the synthesisable VHDL design to provide the CollisionChip design with it. To enable realistic simulation of communication latencies already in the SystemC simulation, the bus was modelled on transactional level as well. It is located within the PCI-API and delays the data transfer according to the system's specifications. The communication via the board's local bus is the limiting element in the aforementioned communication, thus its delay is simulated. The transmission delay $D_{transmission}$ of a data packet of size P-bit of data at the maximum bus frequency of 66MHz can be approximated with the following formula, since the local bus transmits data in 32-bit chunks.

$$D_{transmission}\ \text{[ns]} \approx \frac{P\ \text{[Bit]}}{32\ \text{[Bit]}} \cdot \frac{1}{66\ \text{[MHz]}} \tag{9.1}$$

9.3.2 Reconfiguration Delay

As was discussed in Sec. 6.4 simulating the delay caused by reconfiguration is vital for a decision in favour or against the use of it. Timing experiments such as this are usually made on TFL. But since reconfiguration is widely excepted to be modelled as bus accesses it is performed on transaction level here. Note that this does not necessarily mean that it can occur only late in the design process. It is just a different way of modelling it and can (of course) be mixed with (timed) functional descriptions any time. Thus the CollisionChip simulation is extended with it on this level of abstraction.

The target architecture provides means to (re-)configure the on-board FPGA using API calls, which transmit the according bitstream via PCI and local bus. Still, the transmission delay of the module's bitstream via the bus can be neglected, since the on-board configuration controller is the limiting element. The configuration itself is executed in 8-bit chunks at a maximum frequency of 50 MHz and thus the configuration delay $D_{configuration}$ of a bitstream of size B can be approximated as follows:

$$D_{configuration}\ \text{[ns]} \approx \frac{B\ \text{[Bit]}}{8\ \text{[Bit]}} \cdot \frac{1}{50\ \text{[MHz]}} \tag{9.2}$$

In the following two different types of data transit are compared. This is done to give an example of how ReChannel can be used on TFL to compare two different reconfigurable platforms with respect to their impact on the overall system performance. The first is the target architecture, which provides only a single PCI interface. Thus the data's transition delay and the reconfiguration delay sum up to an overall initialisation

delay.

$$D_{\text{one bus}} = D_{transmission} + D_{configuration} \tag{9.3}$$

The second is a hypothetical architecture, which provides an extra bus for reconfiguration requests. Hence the overall initialisation delay is the maximum of the individual delays.

$$D_{\text{two busses}} = \max\{D_{transmission}, D_{configuration}\} \tag{9.4}$$

Considering $D_{configuration}$ is implemented in a specialised simulation controller `virtex2ctrl`, that models the according timing behaviour. This simulation controller is very similar to the example in Listing 6.12 (page 73). It differs solely in the calculation of the loading delay and is used within the CCC instead of its base class `rc_control` in the following. Furthermore, a `virtex2module` is implemented, which is quite similar to Listing 6.11 (page 72). The primitive tests are now derived from `virtex2module` and the according reconfigurable test module.

Interface Modification

As soon as reconfiguration delays are considered by the simulation an unexpected interface problem occurs. In the static CollisionChip design three signals are used to report the architecture's current state back to the host. The `bv_ready` signal is true if and only if the bounding volume subsystem is ready for a new collision query. `prim_ready` behaves accordingly for the primitive test. `bvh_ready := bv_ready` $\wedge$ `prim_ready` results from those two signals. If a positive edge of `bvh_ready` occurs, a collision test is finished and the result is reported back to the host.

Using reconfigurable intersection tests the `prim_ready` signal is set false during the actual reconfiguration rendering the overall test non-ready. This is necessary to avoid initialising a new test while the reconfiguration is in progress. As soon as a primitive test was activated the `prim_ready` signal is set back to true, signalling that it is ready for new test queries. Using the aforementioned protocol this causes a falling and a rising edge of `bvh_ready`.

As was discussed in Sec. 9.1.2 a reconfiguration is initiated by the API only if a new intersection test was requested. Due to the concurrent behaviour and the architecture's latency simulated in the design, `bvh_ready`'s additional rising edge will cause the design to signal the end of the test before it has actually begun. This effectively prevents the query from being processed at all. This problem needs to be solved inside the API, since this is the only instance informed that a reconfiguration was requested and that receives the according report. Thus solving the problem is simple as soon as it is recognised.

It is unclear if the observation, that introducing reconfiguration leads to a change in the interface's timing specification, can be generalised to all projects. But it obviously proofs that this *can* occur and that it is not possible to provide a general solution on library level, since timing behaviour greatly varies in different architectures. Thus it is clear that the designer will have to deal with it himself. In the presented case, the resulting problem is solved in the instance controlling the reconfiguration. It is very intuitive to solve the problem where it was caused and thus it is likely that that is

Design	Bitfilesize
Triangle-triangle intersection test	5,504,088-bit
Quadrangle-quadrangle intersection test	5,409,976-bit

Table 9.1: Bitfilesize of static triangle-triangle and quadrangle-quadrangle test as reported by Xilinx ISE.

Simulation	Delay $[ns]$
Summed delay of two static designs	274,948,800
Reconfiguration using a single Bus	274,887,700
Reconfiguration using two Buses	274,887,580

Table 9.2: Comparison of three architectures: First: Two static designs that were started independently and their delays were added. Second: A reconfigurable design using a single bus for transmitting data to the CollisionChip and the bitstream. Third: A reconfigurable design using separate buses for collision query and bitstream. Two triangle objects were tested for intersection, followed by two quadrangle objects.

a result, that can be generalised. But to provide a valid methodology to prevent or circumvent changes in interface timing extensive investigations on multiple architectures of realistic dimensions are needed. This exceeds the scope of this thesis by far, thus the problem is only documented. Since this is not a problem resulting from the use of ReChannel but of the introduction of reconfiguration in general, it can be taken for granted that the approaches presented in Sec. 5.2 will also suffer from interface changes in similar settings. Still, to the author's knowledge neither reports nor solutions were published yet.

Impact of Reconfiguration Delays on System Performance

For comparison of the architectures using one and two buses a simple test sequence was used: First two triangle objects were tested for intersection, followed by two quadrangle objects. As a reference a third scenario was simulated: Two static designs that were started subsequently and their delays were added. No bistream had to be transmitted. Since the primitive tests were already synthesised the bitsream sizes of the static tests (see Table 9.1) were used as approximated for the reconfiguration bitstream. Table 9.2 shows the results. It is obvious that the hierarchical intersection test dominates the measurement. Hence reconfiguration can be used safely within the CollisionChip framework. It will not spoil answering delays of the architecture. As was to be expected introducing a special reconfiguration bus does not significantly accelerate the overall calculation and even the performance loss caused by using reconfiguration will be reduced by less than 11%. This will usually not justify the cost of a second bus.

9.4 Register Transfer Level

Another aspect of reconfigurable architecture design which is not discussed here is the description of synthesisable reconfiguration controllers. Such a development would make sense only, if the controller would afterwards be really synthesised and used in practise, to proof the description to be realistic and applicable[1]. As detailed in Sec. 9.3.2 reconfiguration does not significantly increase the system's performance. Thus development of a reconfigurable CCC is not necessary within the given framework. Furthermore, applying a hardware reconfiguration controller inevitably needs a reconfigurable system in hardware to control. Development of such a system demands tremendous implementation effort and solving of problems below SystemC language level (i.e., implementation of bus macros, application of the Xilinx modular design flow, etc.). It might even concern problems within the field of electrical engineering. Thus it is complex enough to justify a doctoral thesis of its own. Hence it exceeds the scope of this work by far and is therefore left to future projects.

Still, this section will refine the reconfigurable parts of the CollisionChip design to a synthesisable SystemC description to enable such a development in the future and to further show ReChannel's applicability on RT level.

9.4.1 System Topology on RTL

Proceeding to RT-level the interface of the intersection tests changes significantly. The FIFO communication is substituted with signals as it is discussed in Sec. 8.7.2. Thus it is necessary to exchange the portals in use accordingly. In general RT level interfaces require a vast number of signals. As it is illustrated in Figure 8.22 (page 144) this applies to the primitive intersection test's RT-level interface, too. A direct consequence is the requirement of as many portals, which not only need to be instantiated, but need to be bound, too. Since multiple reconfigurable modules need to be bound to the dynamic end of theses switches, the number of binding statements multiplies accordingly. This is tedious and error-prone work and the resulting code is difficult to understand and maintain. Listing 9.5 shows binding of a single primitive test to the according portals and their binding to the according static channels. Syntactically the latter binding is no direct binding to channels, but to the inside of ports. Technically this accounts to a binding to the channels these ports are connected to. This may appear rather complicated at first, but simplifies implementation of the topology in multiple ways. Thus it results from the fact, that this is a real application and not solely developed for presentation.

Therefore it is advisable to reduce the number of necessary binding statements. This insight motivated the development of `rc_portmap` and `rc_switch_connector` as they are discussed in Sec. 6.2.6. Introducing these structures the source code depicted in Listing 9.6–9.8 results. Listing 9.6 defines the port map type needed for the binding. Listing 9.7 shows how the triangle-triangle test is extended to provide an according port

[1] This is the same reason, why the CollisionChip was really implemented in (static) hardware. It proofs the realism of the setting itself.

```
1      // Binding of static design to portals
2  clk_in_portal.static_port(clk);
3  reset_in_portal.static_port(reset);
4  m_in_portal.static_port(m);
5  newPrimIDs_in_portal.static_port(newPrimIDs_in);
6  newPrimData_in_portal.static_port(newPrimData_in);
7  prim_a_id_in_portal.static_port(prim_a_id_in);
8  prim_b_id_in_portal.static_port(prim_b_id_in);
9  prim_data_in_portal.static_port(prim_data_in);
10
11 newPrimIDs_out_portal.static_port(newPrimIDs_out);
12 prim_a_id_out_portal.static_port(prim_a_id_out);
13 prim_b_id_out_portal.static_port(prim_b_id_out);
14 intersect_out_portal.static_port(intersect);
15 probably_out_portal.static_port(probably);
16 empty_out_portal.static_port(empty);
17 primsCount_out_portal.static_port(primsCount_out);
18 prim1_size_out_portal.static_port(prim1_size);
19 prim2_size_out_portal.static_port(prim2_size);
20
21     // Binding of reconfigurable triangle-triangle test to portals
22 clk_in_portal.dynamic_port(tri_tri_test_RTL_rc.clk);
23 reset_in_portal.dynamic_port(tri_tri_test_RTL_rc.reset);
24 m_in_portal.dynamic_port(tri_tri_test_RTL_rc.m);
25 newPrimIDs_in_portal.dynamic_port(tri_tri_test_RTL_rc.newPrimIDs_in);
26 newPrimData_in_portal.dynamic_port(tri_tri_test_RTL_rc.newPrimData_in);
27 prim_a_id_in_portal.dynamic_port(tri_tri_test_RTL_rc.prim_a_id_in);
28 prim_b_id_in_portal.dynamic_port(tri_tri_test_RTL_rc.prim_b_id_in);
29 prim_data_in_portal.dynamic_port(tri_tri_test_RTL_rc.prim_data_in);
30
31 newPrimIDs_out_portal.dynamic_port(tri_tri_test_RTL_rc.newPrimIDs_out);
32 prim_a_id_out_portal.dynamic_port(tri_tri_test_RTL_rc.prim_a_id_out);
33 prim_b_id_out_portal.dynamic_port(tri_tri_test_RTL_rc.prim_b_id_out);
34 intersect_out_portal.dynamic_port(tri_tri_test_RTL_rc.intersect);
35 probably_out_portal.dynamic_port(tri_tri_test_RTL_rc.probably);
36 empty_out_portal.dynamic_port(tri_tri_test_RTL_rc.empty);
37 primsCount_out_portal.dynamic_port(tri_tri_test_RTL_rc.primsCount_out);
38 prim1_size_out_portal.dynamic_port(tri_tri_test_RTL_rc.prim1_size);
39 prim2_size_out_portal.dynamic_port(tri_tri_test_RTL_rc.prim2_size);
```

Listing 9.5: Binding of a single primitive test to the according portals and their binding to the according static channels. Syntactically the latter binding is no direct binding to channels, but to the inside of ports. Technically this acounts to a binding to the channels these ports are connected to. For any further primitive intersection test lines 21-39 need to be repeated accordingly. The template parameters are left out for the sake of readability.

```
typedef
rc_portmap< possiblePrimTests,
            sc_in<bool>,                          sc_in<bool>,
            sc_in< sc_bv < Max3x4MatBits > >,     sc_in<bool>,
            sc_in<bool>,                          sc_in<BV_ID_TYPE>,
            sc_in< BV_ID_TYPE >,                  sc_in< sc_bv < 128 > >,
            sc_out<bool>,                         sc_out<BV_ID_TYPE>,
            sc_out<BV_ID_TYPE>,                   sc_out<bool>,
            sc_out<bool>,                         sc_out<bool>,
            sc_out< sc_uint < MaxTC_ADDR > >,     sc_out<OBJ_WORD_TYPE>,
            sc_out<OBJ_WORD_TYPE> >    possiblePrimTestsPortMap;
```

Listing 9.6: Portmap definition for primitive intersection test interfaces.

map to the environmental design. A static port map of the same type is initialised in Listing 9.8 and bound to the switching adaptor, which is then bound to the reconfigurable modules itself. This latter binding is effectively simplified by the use of port maps. Any additionally added intersection test can now be bound to the portals by a single additional line of code.

9.4.2 Reset on Configuration

As discussed in Sec. 6.1.2 an inevitable limitation in rendering static closed source components reconfigurable is the necessity for an externally initiated reset. If the designer does not consider this, the component will behave unpredictably.

In case of the previously detailed architecture the design works correctly until the first reconfiguration is executed. While no test is activated the default value zero is set on the signals connected to the test's output. This is the signal portal's default behaviour. If another test is activated it is not reset on activation and thus does not rewrite its outputs. Hence the signal values at the modules output remain zero. Since the `prim1_size` and `prim2_size` signals are fed back into the `getData` module this component assumes a primitive size of zero and does not forward any data to the primitive test. Due to `getData`'s actual implementation's assumption of primitive sizes greater than zero the architecture deadlocks. This deadlock could be avoided, by simply transmitting no data to the primitive test. Alternatively the signal portals could also be switched to keeping the old value, but this does not solve the problem, since it might lead to incomplete data being transferred to the test. Both results in wrong behaviour as well and makes debugging only more difficult.

Thus an externally initiated reset is the only possible solution. As was previously pointed out this is inevitable, if closed source components are used for reconfiguration. For this an additional process needs to be implemented after the derivation of the primitive tests from `rc_reconfigurable`. Here `RC_METHOD` and `RC_THREAD` can be used, since the full functionality of explicit modelling is available.

The additional process generates a reset signal, which is externally fed back into the

```
template < [...] >
class TriTriTestRTL_rc
    : public ::ReChannel::rc_reconfigurable_module<TriTriTestRTL< [...] > >
{
    [...]

      // Declare port map.
    possiblePrimTestsPortMap pm;

    TriTriTestRTL_rc(sc_module_name name )
        : rc_reconfigurable_module<TriTriTestRTL< [...] > > (name),
        // Initialise port map with own interface
        pm( this->clk,            this->reset,
            this->m,              this->newPrimIDs_in,
            this->newPrimData_in, this->prim_a_id_in,
            this->prim_b_id_in,   this->prim_data_in,
            this->newPrimIDs_out, this->prim_a_id_out,
            this->prim_b_id_out,  this->intersect,
            this->probably,       this->empty,
            this->primsCount_out, this->prim1_size,
            this->prim2_size),
       [...]
    {
        // Provide port map to outside design.
     this->rc_add_portmap(pm);
    }
}
```

Listing 9.7: Instantiation of the port map type defined in Listing 9.6 inside the reconfigurable triangle-triangle test. The test initialises the port map with its own interface and provides it to the environmental design to enable simplified binding. By providing a port map of type `rc_portmap< possiblePrimTests, [...]` the module identifies itself as a primitive test with the according interface.

```
template < [...] >
SC_MODULE (prim_test)
{
   [...]
   possiblePrimTestsPortMap pm;      // Declare port map

    // Declare switching adaptor
   rc_switch_connector<possiblePrimTestsPortMap> sconn;
   [...]

   template < ... > prim_test< ... >::prim_test(sc_module_name name)
    : sc_module(name),  [...]
        // Initialise port map with static interface
      pm(            this->clk,              this->reset,
                     this->m,                this->newPrimIDs_in,
                     this->newPrimData_in,   this->prim_a_id_in,
                     this->prim_b_id_in,     this->prim_data_in,
                     this->newPrimIDs_out,   this->prim_a_id_out,
                     this->prim_b_id_out,    this->intersect,
                     this->probably,         this->empty,
                     this->primsCount_out,   this->prim1_size,
                     this->prim2_size,       this->roc_out),
        // Initialise switching adaptor with portals
      sconn("sconn", clk_in_portal,          reset_in_portal,
                     m_in_portal,            newPrimIDs_in_portal,
                     newPrimData_in_portal,  prim_a_id_in_portal,
                     prim_b_id_in_portal,    prim_data_in_portal,
                     newPrimIDs_out_portal,  prim_a_id_out_portal,
                     prim_b_id_out_portal,   intersect_out_portal,
                     probably_out_portal,    empty_out_portal,
                     primsCount_out_portal,  prim1_size_out_portal,
                     prim2_size_out_portal ),
      [...]
   {
     // Bind static interface
   sconn.bind_static(pm);

     // Bind intersection tests
   sconn.bind_dynamic(tri_tri_test_RTL_rc);
   sconn.bind_dynamic(tri_quad_test_RTL_rc);
   sconn.bind_dynamic(quad_quad_test_RTL_rc);
   [...]
   }
} ;
```

Listing 9.8: Definition and binding of a port map for the static design part's interface to the intersection tests. With the preliminary initialisations of port map and according switching adaptor binding of a primitive test to the portals is done in a single line.

```
virtual void write_roc()
{
    roc_out.write( true );
    wait(); // wait two clock cycles
    wait();
    roc_out.write( false );
}
```

Listing 9.9: A ROC signal is generated within an `RC_THREAD` inside the primitive intersection test after derivation from `rc_reconfigurable`. The `write_roc()` thread is sensitive to the clock's positive edge.

reset port of the primitive test. Listing 9.9 shows the implementation of the simple enough ROC generator, while Figure 9.3 illustrates how it integrates with the primitive test. As can be seen, this causes only a minimum amount of additional logic and development effort.

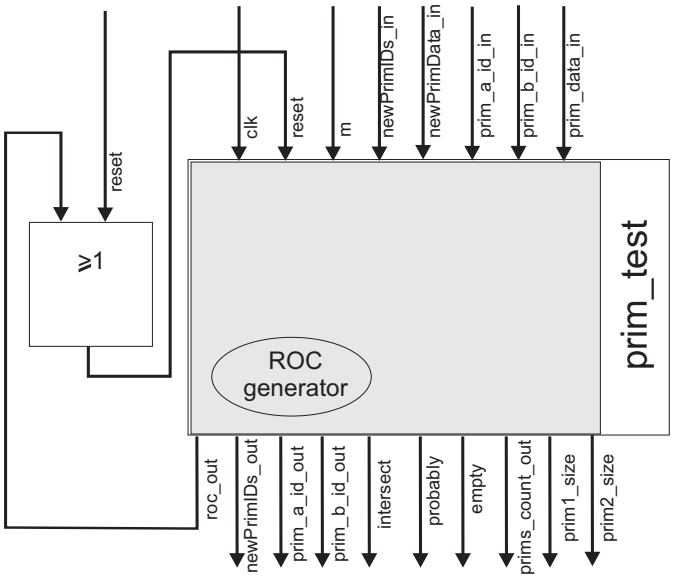


Figure 9.3: The ROC signal is generated within the primitive intersection module and is externally fed back into the primitive test's reset port.

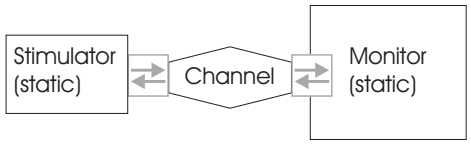
10 Simulation Performance

While it is most important to provide as much flexibility and a most intuitive syntax to the designer, it is almost as important to enable a fast simulation. Even the most elaborated simulation tool will not stand a chance at the market if it slows down development significantly. To provide a full featured overview over ReChannel's run-time, designs of varying complexity and reconfiguration behaviour are used for measurement. Firstly, Sec. 10.1 measures the increase of simulation run-time if reconfiguration is integrated into a design. Secondly, Sec. 10.2 provides a comparison to a handcrafted multiplexer solution. Sec. 10.3 finally concludes the measurements by providing delays caused by ReChannel within the CollisionChip case study, which was presented in Sec. 8.

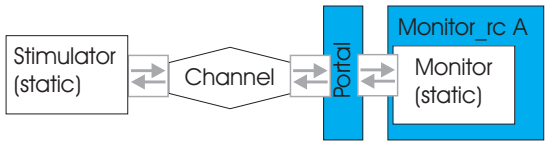
10.1 Simulation Delay Costs of Using Reconfiguration

Firstly, three very simple architectures are investigated. The static design shown in Figure 10.1(a) is used for reference. It consists of a single channel which is bound to ports of a stimulator and a monitor module. To enable analysis of how the overall delay is caused by the various ReChannel components this design is extended with the infrastructure necessary for reconfiguration (Figure 10.1(b)). The third design (Figure 10.1(c)) integrates a second monitor module, which can now be reconfigured. This architecture is used in two different scenarios: Firstly, only one of the monitor modules is used. Secondly, the monitors are alternately activated. Reconfiguration control is implemented within the stimulator using an `rc_control` instance. All designs are implemented using a signal and a FIFO. They are also stimulated in two different ways: Firstly, the stimulator writes only twice to the according channel. The reconfigurating architectures are switched as soon as they complete the first read access. Secondly, the stimulator performs $100,000$ writes, requests a reconfiguration (if applies) and performs $100,000$ additional writes. The measurement is started after the construction phase. All measurements were made on an Intel Pentium 4 CPU 2.8 GHz with 2 GB of main memory using Ubuntu Linux with a 2.6.15-29-386 kernel and gcc version 4.0.3. Table 10.1 and Table 10.2 show the measurement results. Median, average and minimum over 100 measurements are reported in Table 10.1. Since they behave qualitatively and quantitatively nearly equally, in the following only the median is discussed. Minimum and average are discarded in subsequent tables.

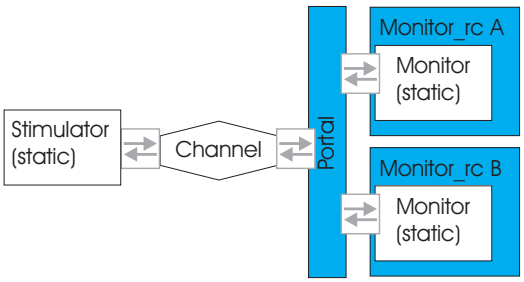
Consider Table 10.1 first. Extending the static design using a signal does not change the simulation's runtime. If the FIFO design is extended with ReChannel infrastructure it suffers from 33% performance loss. Since only very little communication is performed, the time consumed by SystemC's design elaboration dominates the measurement. In small designs such as this the elaboration delay is minimal, and in large



(a) Static design for reference.



(b) Static design with reconfiguration infrastructure.



(c) Reconfigurable design with modules of identical type.

Figure 10.1: Test architectures for measuring ReChannel's reconfiguration delay.

Architecture	FIFO Simulation Run-time Median [μs]	FIFO Simulation Run-time Average [μs]	FIFO Simulation Run-time Minimum [μs]	FIFO Factor Median	Signal Simulation Run-time Median [μs]	Signal Simulation Run-time Average [μs]	Signal Simulation Run-time Minimum [μs]	Signal Factor Median
Static reference (Fig. 10.1(a))	9.00	9.02	8.00	1.00	7.00	7.17	6.00	1.00
Static with ReChannel infrastructure (Fig. 10.1(b))	12.00	11.59	11.00	1.33	7.00	7.43	7.00	1.00
One out of two monitors active (Fig. 10.1(c))	12.00	11.73	11.00	1.33	7.00	7.05	6.00	1.00
Reconfigurating (Fig. 10.1(c))	292.00	293.90	285.00	32.44	280.50	282.12	273.00	40.07

Table 10.1: Comparison of the simulation run-times of the test architectures shown in Figure 10.1. The single channel is implemented both, as a FIFO and a signal. The designs perform only two writes to and two reads from the according channel. The reconfigurating architecture is switched as soon as a read access is completed. The measured run-time is the median/average/minimum over 100 simulation runs. The factor indicates the run-time relatively to the static reference.

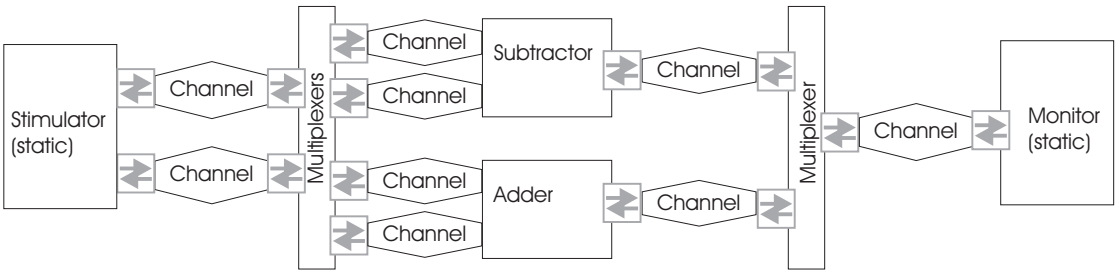
Architecture	FIFO		Signal	
	Simulation Run-time Median [μs]	Factor	Simulation Run-time Median [μs]	Factor
Static reference (Figure 10.1(a))	24,590.00	1.00	93,692.50	1.00
Static with ReChannel infrastructure (Figure 10.1(b))	82,159.50	3.34	112,899.50	1.21
One out of two monitors active (Figure 10.1(c))	85,922.00	3.49	111,955.00	1.19
Reconfigurating (Figure 10.1(c))	85,518.50	3.48	115,532.00	1.23

Table 10.2: Comparison of the simulation run-times of the test architectures shown in Figure 10.1. The architectures' single channel is implemented both, as a FIFO and a signal. The designs perform 200,000 writes to and reads from the according channel. The reconfigurating architecture is switched as soon as a read access is completed. The measured run-time is the median over 100 simulation runs. The factor indicates the run-time relatively to the static reference.

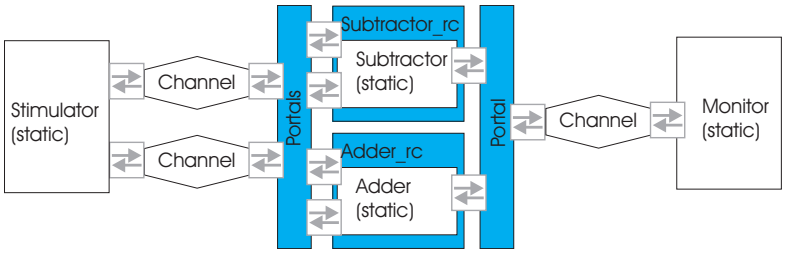
designs it will nearly always be negligible if communication delay is considered, too. Thus this aspect is irrelevant here. This factor is ruled out in the measurement tabulated in Table 10.2. The additional simulation run-time of 234% and 21% of the design with ReChannel infrastructure is caused by communicating via portals. Thus factors 3.34 and 1.21 apply for designs which equipped each of their FIFOs / signals with portals and do not have time consuming calculations. This may appear much at first, but is an expected result, since the use of switches causes additional communication overhead, e.g., notification of events and additional method calls.

Adding a second monitor module does not change the run-time in any of the test cases. This measurement is primarily provided to disable compiler optimisations concerning the previously discussed overhead. This especially enables a valid comparison with the reconfigurating designs.

Actually reconfiguring the design slows simulation down by a factor of about 32 and 40 respectively, for the designs with low communication requirement. This is also an expected effect, since ReChannel was designed to optimise communication, and thus a lot of calculation overhead was moved into the methods implementing the reconfiguration itself. If a lot communication occurs in the reconfigurable design the performance remains practically unchanged at a factor of about 3.5 even for FIFO communication. Defining a worst case of a channel in use is hard, since they can be arbitrarily complex. FIFOs are the most complex channel type predefined by SystemC as far as ReChannel is concerned, thus they are provided with synchronised accessors by default (see Sec. 6.2.4).



(a) An adder and a subtractor module reconfigured using multiplexers.



(b) An adder and a subtractor module reconfigured using ReChannel.

Figure 10.2: Architectures used to compare ReChannel's simulation performance with the "classical" multiplexer solution.

Still, the factor of 3.48 is too large to be acceptable in designs of realistic size. But in the vast majority of realistic design studies large areas of functionality are exchanged. Thus usually an amount of communication through portals comparable to the test designs will not be required. This is underlined in Sec. 10.3, which provides run-time measurements for the simulation of the CollisionChip project.

10.2 Comparing ReChannel with Multiplexers

The previous Sec. 10.1 aims at providing boundaries of additional simulation delay caused by adding reconfigurable components to a static design using ReChannel. This is unsatisfying, since it does not examine how ReChannel compares to other reconfiguration simulation techniques. To provide a valid comparison of simulation delay to the techniques discussed in Sec. 5.2 is not possible. These libraries and languages are too different from ReChannel and from each other, and each requires its own description style. Thus there is no single design, or class of architectures that could be used for reference.

Instead ReChannel is compared to the most "classic" approach: The multiplexer solution, which is the simplest form of a RecBus (see Sec. 6.1). This very "low-tech" approach does not respect any reconfiguration delays and does not provide any syntactical sugar. Therefore it is expected to out-perform any library, that does.

Again a very minimalistic example architecture is used to enable valid measurement results: An adder module is exchanged against a subtractor. Figure 10.2(a) shows the

Architecture	FIFO		Signal	
	Simulation Run-time Median [μs]	Factor	Simulation Run-time Median [μs]	Factor
Multiplexer	58,586,400	1.00	62,963,800	1.00
ReChannel	71,324,700	1.22	81,451,750	1.29

Table 10.3: Simulation run-times of the designs shown in Figure 10.2.

multiplexer and Figure 10.2(b) the ReChannel architecture. Reconfiguration control is again implemented within the stimulator. Obviously the ReChannel design has a much simpler topology.

Both designs were simulated using FIFOs and signals as only channel type. Since the multiplexers switch practically instantaneously a comparison only makes sense if a mixture of communication and reconfiguration is used. Thus the stimulator produces $2,000,000$ data sets, every $100,000$ the architecture is reconfigured. Table 10.3 shows the measurement results.

ReChannel provides a lot of comfort, respects reconfiguration delays, is far more flexible since it can be used on all abstraction levels and with any kind of channel, and does not alter the system's timing behaviour, while no reconfiguration is requested. Neither of this applies to the multiplexer solution. Still, the architecture using ReChannel is less than 30% slower than the one implemented using multiplexers.

10.3 ReChannel Simulation Delay in the CollisionChip

The CollisionChip was presented in Sec. 8 to provide a real world case study of ReChannel's applicability. Here both, the functional as well as the RT-Level implementation are re-used to provide a realistic framework for measuring the simulation delay. For this a triangle-triangle primitive intersection test is exchanged against a module of the same type. This enables comparison of the reconfigurable architecture with a static reference design of identical functionality. As was done in Sec. 10.1 a static design equipped with the ReChannel infrastructure is also provided as an extended reference. Two identical headlight objects were tested for collision, the design is reconfigured, and the test is repeated. Table 10.4 presents the measured simulation run-times.

The designs on both abstraction levels only suffer from minimal performance loss. The functional implementation has an increased simulation run-time of only 3%. Even if one respects that this might be a bit underestimated, since the hierarchy traversal is implemented on RTL, this is a very good result. But even more important is the run-time on RTL, since SystemC itself is critically low-performing here compared to other HDLs. The RT-Level implementation using ReChannel is only 11% slower than the static design, without any additional reconfiguration infrastructure. This exceeds any preliminary expectations.

Architecture	UTF-Level		RT-Level	
	Simulation Run-time [h]	Factor	Simulation Run-time [h]	Factor
Static reference	1.2612	1.0	2.6375	1.00
Static with ReChannel infrastructure	1.2991	1.03	2.8715	1.09
Reconfigurating	1.3011	1.03	2.9303	1.11

Table 10.4: Simulation run-time measurement results of the CollisionChip being simulated using ReChannel. A triangle-triangle intersection test module is exchanged against an identical module to enable comparison to static design implementations.

11 Conclusion

After a brief discussion of the general field of electronic design automation and hardware design this thesis deduced that an extension of SystemC with language constructs for reconfiguration is a valuable contribution. Thus objectives for this project were defined to maximise its applicability and its benefit for the design community.

In Part II (Sec. 5.2) related approaches are reviewed and analysed. Although some are without doubt academically interesting, while others are of greater practical relevance, none of them provides a satisfying SystemC extension considering all objectives. Thus Sec. 6 introduces the ReChannel library, which is the main contribution of this work.

ReChannel extends SystemC with the according language constructs for modelling and simulating reconfiguration. Since it does not alter the SystemC kernel and makes exclusive use of standardised SystemC features, it complies to the language standard and can be used with any standard compliant SystemC implementation. The concept of portals was introduced and generalised to switches. It decouples binding and inter-connectivity, by intercepting communication between interfaces and ports. Since ReChannel provides the switch abstraction it also separates connectivity changes from functionality changes. Reconfigurable modules result by derivation from static ones and can then be extended with reconfigurable properties. This enables reuse even of closed source, third party IP-cores as reconfigurable components. Multiple extensions to the ReChannel framework are proposed, which render it a most convenient and elegant tool. It was discussed in-depth how ReChannel integrates with the SystemC simulation cycle and its delta-delay semantic. To enable integration of ReChannel into the SystemC design flow, a refinement methodology for reconfigurable components and their integration into a system description was proposed. The proposed ReChannel primitives were designed with great care to make them look as "SystemC-like" as possible and to allow both, handcrafted as well as automated refinement to synthesisability.

To proof ReChannel's effectiveness and applicability a realistic case study was presented in Part III. Hierarchical collision testing was chosen, since it is a novel field of application, which demands tremendous processing speed and is both, academically and economically interesting. Sec. 7.1 gives a brief overview of hierarchical collision detection and discusses the state-of-the-art in hardware accelerated intersection testing.

In Sec. 8 the CollisionChip architecture was introduced. It is this work's second key contribution. A novel fixed-point intersection test for k-DOPs, based on the well known separating axis test, was derived. It guarantees that no false reports of non-collision (false negatives) occur during hierarchy traversal. A bound on the fixed-point deviation was derived, which also bounds the number of occurring false reports of collision (false positives). A pipeline architecture for this test was proposed and implemented along with the necessary controlling and infrastructure in SystemC and VHDL. A performance

analysis was presented, which shows that the memory interface is the main bottleneck in hardware accelerated collision detection. Thus the intersection test hardware was augmented with a specialised caching module, which was proposed and compared to approaches known from literature in Sec. 8.5. It performs as well as a fully-associative cache with least-recently used strategy, while consuming far less resources. This allows implementing a large cache and thus enables it to perform close to the theoretical optimum. In conjunction with a novel hierarchy traversal scheme two pipelines can now be combined to outperform a state-of-the-art software implementation of a hierarchical bounding-volume test by an order of magnitude if a clock rate of 100 MHz is assumed.

To enable a fully featured intersection test various triangle intersection tests were reviewed with respect to their hardware implementability and their generalisability to quadrangles. The separating axis test was found to be best suited. It was implemented and tested in SystemC and VHDL for triangle-triangle, triangle-quadrangle and quadrangle-quadrangle intersection testing.

To proof the implementation to be realistic, the whole design was synthesised and tested on the target architecture. Individually, each of the modules allows clock rates of at least 50 MHz, most allow more than 100 MHz. If combined, the possible clock rate decreases to 10 MHz. The source of this decrease was identified. It can be eliminated if sufficient development time is spent. Since performance proof in simulation and correct behaviour on-chip suffice to prove the case studies realism, this elimination was found to be out of scope of this thesis.

Part IV integrates reconfiguration into the CollisionChip's SystemC simulation using ReChannel. The triangle-triangle, the triangle-quadrangle and the quadrangle-quadrangle test are exchanged during run-time of the simulation. The integration was performed on all levels of abstraction featured by the SystemC refinement methodology. It followed the ReChannel methodology as it was proposed in previous sections. While speed of simulation is not part of the objectives, it was measured and presented. If applied to the CollisionChip ReChannel increases the simulation's run-time only by 11% even for the RTL implementation. Thus it exceeds expectations in this respect by far if applied to realistic scenarios.

ReChannel was applied very successfully within the CollisionChip project. It does not only allow simulation of reconfiguration, but was effectively used to gain information on the usefulness of reconfiguration within the CollisionChip project itself. The primitive intersection test modules were treated as closed source, and thus not altered in the process. Hence, it can be concluded that ReChannel effectively enables IP reuse in reconfigurable modules on all levels of abstraction. No non-compliance to any of the objectives was detected in the process either. Thus it can be concluded that ReChannel provides a most convenient framework for describing and simulating reconfiguration, while imposing minimum additional development overhead.

12 Future Work

12.1 Future Work on ReChannel

No case study can cover all aspects of system design. Therefore it was not possible to test all aspects of ReChannel within the CollisionChip project. Probably the topic of highest importance here concerns transaction level description. TL modelling is commonly accepted as a major source of productivity in contemporary chip design, because of its tremendous flexibility. This also makes it a very wide field. The CollisionChip does not contain any bus communication and thus no bus interfaces were found at the border from static to reconfigurable design parts. Therefore it was not possible to include specialised examinations on how to integrate reconfiguration into bus interfaces. Thus it would be most important to provide case studies, which cover this field.

Moreover, it would be highly interesting and important to extend ReChannel with native support for the SystemC TLM library. This would provide strong evidence, not only of its flexibility, but also of its applicability in projects of larger dimensions.

Despite the additional simulation delay caused by ReChannel in the CollisionChip simulation was far lower than one could have expected, there might still be modes of usage were it is too large. Thus optimisation efforts should be undertaken to improve its runtime. E.g., the current ReChannel implementation provides an additional management layer implementing its own miniature process controlling. This causes significant overhead. Since process controlling is very likely to be incorporated in the SystemC standard soon, this native SystemC process controlling should then be used.

Some aspects of dynamic reconfiguration were only slightly touched in this work, e.g., rechannelling of communication in a running system with fixed module topology. Although ReChannel was designed keeping this in mind right from the beginning (hence the name), it lacks support for this exciting (yet uncommon) form of reconfiguration.

Moreover, this work did not cover any mobility aspects, i.e., not only exchanging modules, but even moving complete modules within the hierarchy. This can be interpreted as a form of reconfiguration as well. ReChannel in its current state of development is already able to simulate some aspects of mobility, e.g., rechanneling data or de-registering modules with their controller, while re-registering them with another. Still, it completely lacks support for other aspects as, e.g., extraction and transport of a module's state.

12.2 Future Work on CollisionChip

The individual components of the CollisionChip perform very well. But as was already discussed in Sec. 8.6 they only allow clock rates below $20MHz$ if implemented in conjunction on-chip. The source of this performance loss was identified in this work, but it still needs to be fixed.

The architecture effectively and efficiently detects collisions of rigid bodies if provided with precalculated bounding-volume hierarchies. Extending the design in a way, that it generates and updates theses hierarchies itself would allow the architecture to detect collisions between deformable objects, too.

The primitive test is implemented in fixed-point arithmetic. No special attention was paid to rounding issues, although the solution proposed for k-DOPs needs to be adapted only slightly. This work is currently in progress.

Another very important topic is the intersection test between objects built of more complex primitives. This would open new fields of application.

And last but not least it is highly desirable to have a fully featured realtime demonstrator. For this it is necessary to integrate the CollisionChip's API with a graphic package providing rendering and calculation of collision answers.

13 List of Tables

14 List of Figures

15 List of Listings

Bibliography

[1] Ageia. White paper, May 2005.

[2] Alisson V. De Brito, Elamr U. K. Melcher, and Wilson Rosas. An open-source tool for simulation of partially reconfigurable systems using SystemC. In *IEEE Computer Society Annual Symposium on Emerging VLSI Technologies and Architectures (ISVLSI'06)*, pages 434–435, 2006.

[3] J. Arenberg. Re: Ray/triangle intersection with barycentric coordinates. *Ray Tracing News*, 1(11), 1988.

[4] N. Atay and B. B. John W. Lockwood. A Collision Detection Chip on Reconfigurable Hardware. Technical report, Washington University in St. Louis, 2005.

[5] N. Atay and B. B. John W. Lockwood. A Collision Detection Chip on Reconfigurable Hardware. In *13th Pacific Conference on Computer Graphics and Application*, 2005.

[6] R. Azevedo, S. Rigo, M. Bartholomeu, G. Araujo, C. Araujo, and E. Barros. The archc architecture description language and tools. *Int. J. Parallel Program.*, 33(5):453–484, 2005.

[7] A. Baldassin, P. C. Centoducatte, and S. Rigo. Extending the archc language for automatic generation of assemblers. In *SBAC-PAD '05: Proceedings of the 17th International Symposium on Computer Architecture on High Performance Computing*, pages 60–68, Washington, DC, USA, 2005. IEEE Computer Society.

[8] B. Bartyzel. Hardwarearchitektur für Kollisionserkennung, 2003.

[9] P. Bellows and B. Hutchings. JHDL - An HDL for Reconfigurable Systems. In *FCCM, IEEE Symposium on FPGAs for Custom Computing Machines*, page 175, 1998.

[10] J. Bergeron. *Writing Testbenches: Functional Verification of HDL Models*. Kluwer Academic Publishers, Norwell, MA, USA, 2000.

[11] B. Bhattacharyya, J. Rose, and S. Swan. Language Extensions to SystemC: Process Control Constructs. In *DAC '07: Proceedings of the 44th annual conference on Design automation*, pages 35–38, New York, NY, USA, 2007. ACM Press.

[12] G. Bosman. A Survey of Co-Design Ideas and Methodologies. Master's thesis, vrije Universteit amsterdam, 2003.

[13] A. V. Brito, M. Kuhnle, M. Hubner, J. Becker, and E. U. K. Melcher. Modelling and Simulation of Dynamic and Partially Reconfigurable Systems using Systemc. *ISVLSI*, 00:35–40, 2007.

[14] Cadence. *NC-SystemC Simulator - Datasheet*, 2003.

[15] Celoxica. *Agility manual*.

[16] J. D. Cohen, M. C. Lin, D. Manocha, and M. K. Ponamgi. I-COLLIDE: An interactive and exact collision detection system for large-scale environments. In P. Hanrahan and J. Winget, editors, *1995 Symposium on Interactive 3D Graphics*, pages 189–196. ACM SIGGRAPH, Apr. 1995. ISBN 0-89791-736-7.

[17] L. Crnogorac, A. S. Rao, and K. Ramamohanarao. Inheritance anomaly - a formal treatment. In *FMOODS '97: Proceeding of the IFIP TC6 WG6.1 international workshop on Formal methods for open object-based distributed systems*, pages 319–334, London, UK, UK, 1997. Chapman & Hall, Ltd.

[18] David C. Black and Jack Donovan. *SystemC From The Ground Up*. Kluwer Academic Publishers, 2004.

[19] D. P. Dobkin and D. G. Kirkpatrick. A linear algorithm for determining the separation of convex polyhedra. *J. Algorithms*, 6:381–392, 1985.

[20] C. Donninger and U. Lorenz. The hydra project. *XCell Journal*, 2005.

[21] Doulos. *SystemC Golden Reference Guide*.

[22] D. Eberly. Dynamic Collision Detection using Oriented Bounding Boxes. December 2002. `http://www.geometrictools.com`.

[23] D. Eberly. Intersection of Convex Objects: The Method of Separating Axes. October 2007. `http://www.geometrictools.com`.

[24] S. A. Edwards. The Challenges of Hardware Synthesis from C-Like Languages. In *DATE '05: Proceedings of the Conference on Design, Automation and Test in Europe*, pages 66–67, Washington, DC, USA, 2005. IEEE Computer Society.

[25] S. Fisher and M. Lin. Fast penetration depth estimation for elastic bodies using deformed distance fields. In *Proc. International Conf. on Intelligent Robots and Systems (IROS)*, 2001.

[26] S. Gottschalk. Separating Axis Theorem. Technical Report TR-96-024, 1996.

[27] S. Gottschalk, M. Lin, and D. Manocha. OBB-Tree: A hierarchical structure for rapid interference detection. In H. Rushmeier, editor, *SIGGRAPH 96 Conference Proceedings*, pages 171–180. ACM SIGGRAPH, Addison Wesley, Aug. 1996. held in New Orleans, Louisiana, 04-09 August 1996.

[28] M. Graphics. *ModelSim UserâĂŹs Manual*, 2007.

[29] E. Grimpe and F. Oppenheimer. Aspects of Object Oriented Hardware Modelling With SystemC-Plus. In *System on Chip Design Languages. Extended papers: Best of FDL'01 and HDLCon'01.*, pages 213–223. Kluwer Academic Publ., 2002.

[30] Grötker. Modeling Software with SystemC 3.0, 6th European SystemC User Group Presentation, 2002.

[31] T. Grötker, S. Liao, G. Martin, and S. Swan. *System Design with SystemC*. Kluwer Academic Publishers, 2002.

[32] L. J. Guibas, D. Hsu, and L. Zhang. H-walk: Hierarchical distance computation for moving convex bodies. In W. V. Oz and M. Yannakakis, editors, *Proc. ACM Symposium on Computational Geometry*, pages 265–273, 1999.

[33] S. Huh, D. N. Metaxas, and N. I. Badler. Collision resolutions in cloth simulation. In *IEEE Computer Animation Conf.*, Seoul, Korea, Nov.2001.

[34] IEEE Standards Association ("IEEE-SA") Standards Board. *IEEE Std 1666 -2005 Open SystemC Language Reference Manual*, 2005.

[35] IEEE Standards Association Standards Board. *IEEE Std 1666 -2005 Open SystemC Language Reference Manual*.

[36] D. L. James and D. K. Pai. BD-Tree: Output-sensitive collision detection for reduced deformable models. *ACM Transactions on Graphics (SIGGRAPH)*, 23(3), Aug. 2004.

[37] G. Kahn. The Semantics of Simple Language for Parallel Programming. In *IFIP Congress*, pages 471–475, 1974.

[38] J. Klein and G. Zachmann. The expected running time of hierarchical collision detection. In *SIGGRAPH 2005, Poster*, Los Angeles, Aug. 2005.

[39] T. Larsson and T. Akenine-Möller. Collision detection for continuously deforming bodies. In *Eurographics*, pages 325–333, 2001. short presentation.

[40] P. Lysaght and J. Stockwood. A Simulation Tool for Dynamically Reconfigurable Field Programmable Gate Arrays. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 4(3):381–390, 1996.

[41] H. D. Man. On Nanoscale Integration and Gigascale Complexity in the Post.Com World. In *DATE*, page 12, 2002.

[42] J. Mezger, S. Kimmerle, and O. Etzmuss. Hierarchical techniques in collision detection for cloth animation. In *Journal of WSCG '2003*, University of West Bohemia, Plzen, Czech Republic, Feb.3–7 2003.

[43] B. Mirtich. V-clip: Fast and robust polyhedral collision detection. *ACM Trans. Graph.*, 17(3):177–208, July 1998.

[44] T. Möller. A Fast Triangle-Triangle Intersection Test. *journal of graphics tools*, 2(2):25–30, 1997.

[45] K. Morris. Cray goes fpga - algorithm acceleration in the new xd1. *FPGA and Programmable Logic Journal*, 2005.

[46] W. Mueller, J. Ruf, D. Hofmann, J. Gerlach, T. Kropf, and W. Rosenstiehl. The simulation semantics of systemc, 2001.

[47] W. Müller, W. Rosenstiel, and J. Ruf, editors. *SystemC: Methodologies and Applications*. Kluwer Academic Publishers, Norwell, MA, USA, 2003.

[48] A. Nett. Dynamisch rekonfigurierbare Primitivschnitttests für eine FPGA-basierte Kollisionserkennung, August 2006.

[49] A. H. Niers. Evaluierung des ReChannel-Workflows zur Synthese von dynamisch rekonfigurierbaren SystemC-Modellen, December 2007.

[50] Open SystemC Initiative (OSCI). *SystemC*. `http://www.systemc.org`.

[51] Open SystemC Initiative (OSCI). *Draft Standard SystemC Language Reference Manual*, 25 Apr. 2005.

[52] PACT XPP Technologies. Reconfiguration on XPP-III Processors, 2006.

[53] PACT XPP Technologies. XPP-III Processor Overview, 2006.

[54] S. Pasricha, N. Dutt, and M. Ben-Romdhane. Extending the transaction level modeling approach for fast communication architecture exploration. In *DAC '04: Proceedings of the 41st annual conference on Design automation*, pages 113–118, New York, NY, USA, 2004. ACM Press.

[55] A. Pelkonen, K. Masselos, and M. Cupak. System-Level Modeling of Dynamically Reconfigurable Hardware with SystemC. *Proceedings of International Symposium on Parallel and Distributed Processing (Reconfigurable Architecturs Workshop)*, Apr. 2003.

[56] E. Plante, M.-P. Cani, and P. Poulin. A layered wisp model for simulating interactions inside long hair. In N. M.-T. Marie-Paule Cani, Daniel Thalmann, editor, *Computer Animation and Simulation 2001Proceeding*, Computer Science. EUROGRAPHICS, Springer, Sept. 2001. Proceedings of the EG workshop of Animation and Simulation.

[57] Y. Qu, K. Tiensyrja, K. Masselos, and K. System-level modeling for dynamically reconfigurable co-processors. In *Field-Programmable Logic and its application (FPL)*, 2004.

[58] A. Raabe, B. Bartyzel, J. K. Anlauf, and G. Zachmann. Hardware Accelerated Collision Detection — An Architecture and Simulation Results. In *Design Automation and Test (DATE)*, pages 130–135, Munich, Germany, Mar.7–11 2005. IEEE Computer Society.

[59] A. Raabe, B. Bartyzel, G. Zachmann, and J. K. Anlauf. Hardware Accelerated Collision Detection – An Architecture and Simulation Results. In *8th WSEAS International Conf. on SYSTEMS*, pages 487–321, Vouliagmeni, Athens, Greece, July12–14 2004.

[60] A. Raabe, P. A. Hartmann, and J. K. Anlauf. Rechannel: Describing and Simulating Reconfigurable Hardware in SystemC. *ACM Transactions on Design Automation of Electronic Systems*, 13(1):1–18, 2008.

[61] A. Raabe, S. Hochgürtel, G. Zachmann, and J. K. Anlauf. Hardware-Accelerated Collision Detection using Bounded-Error Fixed-Point Arithmetic. *Journal of WSCG '2006*, pages 17–24, 2006.

[62] A. Raabe, S. Hochgürtel, G. Zachmann, and J. K. Anlauf. Space-Efficient FPGA-Accelerated Collision Detection for Virtual Prototyping. In *Design Automation and Test (DATE)*, pages 206–211, Munich, Germany, 2006.

[63] A. Raabe and F. Zavelberg. Defying the Memory Bottleneck in Hardware Accelerated Collision Detection. In *WSCG '2008*, University of West Bohemia, Plzen, Czech Republic, 2008.

[64] A. Schallenberg, F. Oppenheimer, and W. Nebel. Designing for dynamic partially reconfigurable FPGAs with SystemC and OSSS. In *Forum on Specification and Design Languages*, Lille, France, Sept. 2004.

[65] L. Semeria, K. Sato, and G. D. Micheli. Resolution of dynamic memory allocation and pointers for the behavioral synthesis form c. In *Design Automation and Test (DATE)*, pages 312 – 319, Paris, France, 2000.

[66] Synopsys. *Describing Synthesizable RTL in SystemC*, 2002.

[67] Synthesis Working Group of Open SystemC Initiative. *SystemC Synthesizable Subset*, 2004.

[68] Y. Tanurhan. Processors and FPGAs Quo Vadis? *IEEE Computer*, 39(11):108–110, 2006.

[69] K. Tiensyrja, Y. Qu, Y. Zhang, C. Miroslav, L. Rynders, G. Vanmeerbeeck, K. Masselos, K. Potamianos, and M. Pettisalo. Systemc and ocapi-xl based system-level design for reconfigurable systems-on-chip. In *Forum on Design Languages (FDL)*, 2004.

[70] N. Tredennick and B. Shimamoto. The Rise of Reconfigurable Systems. In T. P. Plaks, editor, *Engineering of Reconfigurable Systems and Algorithms*, pages 3–12. CSREA Press, 2003.

[71] S. Trenkel, R. Weller, and G. Zachmann. A benchmarking suite for static collision detection algorithms. In V. Skala, editor, *Inter'l Conf. in Central Europe on Computer Graphics, Visualization and Computer Vision (WSCG)*, Plzen, Czech Republic, Jan. 29 – Feb. 1 2007. Union Agency.

[72] G. J. A. van den Bergen. *Collision Detection in Interactive 3D Computer Animation*. PhD dissertation, Eindhoven University of Technology, 1999.

[73] K. von der Heyde. Steuerung, Integration und Synthese eines FPGA-basierten 3D-Kollisionserkennungssystemes, October 2007.

[74] N. S. Voros and K. Masselos. *System Level Design of Reconfigurable Systems-on-Chip*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.

[75] R. Weller, J. Klein, and G. Zachmann. A model for the expected running time of collision detection using aabb trees. In *Eurographics Symposium on Virtual Environments (EGVE)*, Lisbon, Portugal, May8–10 2006.

[76] W. Wolf. A decade of hardware/software codesign. *Computer*, 36(4):38–43, 2003.

[77] Xilinx. Virtex-4 Configuration Guide, 2006.

[78] Xilinx. Virtex-II Platform FPGA User Guide, 2006.

[79] Xilinx. Virtex-4 User Guide, 2007.

[80] Xilinx Application Note 290. Two flows for partial reconfiguration: Module based or Difference based, 2004.

[81] J. J. Yi, L. Eeckhout, D. J. Lilja, B. Calder, L. K. John, and J. E. Smith. The future of simulation: A field of dreams. *IEEE Computer*, 39(11):22–29, 2006.

[82] G. Zachmann. Rapid collision detection by dynamically aligned DOP-trees. In *Proc. of IEEE Virtual Reality Annual International Symposium; VRAIS '98*, pages 90–97, Atlanta, Georgia, Mar. 1998.

[83] G. Zachmann. Optimizing the collision detection pipeline. In *Proc. of the First International Game Technology Conference (GTEC)*, Jan. 2001.

[84] G. Zachmann. Minimal hierarchical collision detection. In *Proc. ACM Symposium on Virtual Reality Software and Technology (VRST)*, pages 121–128, Hong Kong, China, Nov.11–13 2002.

[85] G. Zachmann. Triangle and quadrangle intersection tests. 2004.

[86] G. Zachmann and G. Knittel. An architecture for hierarchical collision detection. In *Journal of WSCG '2003*, pages 149–156, University of West Bohemia, Plzeň, Czech Republic, Feb.3–7 2003.

[87] G. Zachmann and G. Knittel. High-performance collision detection hardware. Technical Report CG-2003-3, University Bonn, Informatik II, Bonn, Germany, Aug. 2003.

Part V

Appendix

Appendix A

Lemmas for Bounding the Fixed-Point Error

A.1 Bounding Mapping Vectors

In Sec. 8.1.2 the dihedral angle between all pairs of neighbouring faces of a DOP is restricted to being smaller than $\pi/2$ to ensure that all $P_{A,i}$ are in the interval $[-1, 0]$. That this indeed is a direct consequence needs still to be proven.

Since only a single k-DOP is used respecting an additional orientation matrix R as is done in the preceding chapters is not necessary. Assume the DOP orientations to be manipulated accordingly.

Without loss of generality let orientations and test-axes be normalised and the orientations be a right-hand system:

$$|\mathbf{D}_{j_0}| = |\mathbf{D}_{j_1}| = |\mathbf{D}_{j_2}| = |\mathbf{L}| = 1 \tag{A.1}$$

$$\left(\mathbf{D}_{j_0}{}^T \cdot (\mathbf{D}_{j_1} \times \mathbf{D}_{j_2})\right) > 0 \tag{A.2}$$

Further let $\mathbf{v}^{min}$ be the intersection of the edges $\mathbf{E}_{1,2}$, $\mathbf{E}_{2,0}$ and $\mathbf{E}_{0,1}$. And let edge $E_{m,n}$ be the intersection of the faces defined by $\mathbf{D}_{j_m}$ and $\mathbf{D}_{j_n}$.

$$\mathbf{E}_{1,2} := -\frac{(\mathbf{D}_{j_1} \times \mathbf{D}_{j_2})}{|\mathbf{D}_{j_1} \times \mathbf{D}_{j_2}|} \quad \mathbf{E}_{2,0} := -\frac{(\mathbf{D}_{j_2} \times \mathbf{D}_{j_0})}{|\mathbf{D}_{j_2} \times \mathbf{D}_{j_0}|} \quad \mathbf{E}_{0,1} := -\frac{(\mathbf{D}_{j_0} \times \mathbf{D}_{j_1})}{|\mathbf{D}_{j_0} \times \mathbf{D}_{j_1}|} \tag{A.3}$$

Using the convexity property of DOPs ensuring $\mathbf{v}^{min}$ to be the vertex with minimal image on $\mathbf{L}$ can be done by demanding that its edges point into the direction of $\mathbf{L}$.

$$\mathbf{L}^T \cdot \mathbf{E}_{1,2} \geq 0 \quad \mathbf{L}^T \cdot \mathbf{E}_{2,0} \geq 0 \quad \mathbf{L}^T \cdot \mathbf{E}_{0,1} \geq 0 \tag{A.4}$$

Interpreting $\mathbf{v}^{min}$ as intersection of the face defined by $\mathbf{D}_{j_0}$ and $\mathbf{E}_{1,2}$, let α_0 be the angle between $\mathbf{D}_{j_0}$ and $\mathbf{E}_{1,2}$. And let β_0 be the angle between $\mathbf{E}_{1,2}$ and $\mathbf{L}$. Using Eq. A.2-A.4 results

$$\cos(\alpha_0) := \mathbf{D}_{j_0}{}^T \cdot \mathbf{E}_{1,2} \leq 0 \tag{A.5}$$

$$\cos(\beta_0) := \mathbf{L}^T \cdot \mathbf{E}_{1,2} \geq 0 \tag{A.6}$$

Let α_i and β_i be defined analogously.
In the following it will be shown that

$$-1 \leq P_0 \leq 0 \tag{A.7}$$

The same accounts for P_1 and P_2 and can be shown analogously.
Now $\mathbf{v}_{min}$ can be interpreted as the projection d_0' of the plane defined by d_0 and $\mathbf{D}_{j_0}$ onto $\mathbf{E}_{1,2}$.

$$\begin{aligned} -d_0' &= \frac{d_0}{\cos(\pi/2 - \alpha_0)} \\ \Leftrightarrow d_0' &= \frac{d_0}{\cos(\alpha_0)} \end{aligned} \tag{A.8}$$

Additionally, the projection of $\mathbf{v}_{min}$ onto $\mathbf{L}$, can be interpreted as the projection d_0'' of the plane through $\mathbf{v}_{min}$ with normal $\mathbf{E}_{1,2}$ onto $\mathbf{L}$.

$$d_0'' = d_0' \cdot \ cos(\beta_0) = d_0 \cdot \frac{\cos(\beta_0)}{\cos(\alpha_0)} \tag{A.9}$$

Hence P_0 can be rephrased using Eq. A.8-A.9 and bounded by 0 using Eq. A.5-A.6.

$$P_0 = \frac{d_0''}{d_0} = \frac{\mathbf{L}^T \cdot \mathbf{E}_{1,2}}{\mathbf{D}_{j_0}{}^T \cdot \mathbf{E}_{1,2}} \leq 0 \tag{A.10}$$

As discussed before it is no hard restriction to demand that the dihedral angle between all pairs of neighbouring faces of a DOP is smaller than $\pi/2$. To prove -1 to be a lower bound of P_0, it therefore remains to show that this is a consequence of the bound on the orientation angles.

$$\begin{aligned} &((\mathbf{D}_{j_1} \cdot \mathbf{D}_{j_2} \geq 0) \ \wedge \ (\mathbf{D}_{j_2} \cdot \mathbf{D}_{j_0} \geq 0) \ \wedge \ (\mathbf{D}_{j_0} \cdot \mathbf{D}_{j_1} \geq 0)) \\ &\Rightarrow \forall_{\mathbf{L}} : (P_0 \geq -1) \end{aligned} \tag{A.11}$$

For simplification purposes we rotate DOP and test axis by a matrix R_0.

$$\begin{aligned} \mathbf{R}_0 &:= \begin{pmatrix} \mathbf{D}_{j_0} & -\mathbf{E}_{0,1} \times \mathbf{D}_{j_0} & -\mathbf{E}_{0,1} \end{pmatrix}^T \\ \mathbf{D}'_{j_0} &:= \mathbf{R}_0 \cdot \mathbf{D}_{j_0} = \begin{pmatrix} 1 & 0 & 0 \end{pmatrix}^T \\ \mathbf{D}'_{j_1} &:= \mathbf{R}_0 \cdot \mathbf{D}_{j_1} = \begin{pmatrix} x_1 & y_1 & 0 \end{pmatrix}^T, y_1 \geq 0 \\ \mathbf{D}'_{j_2} &:= \mathbf{R}_0 \cdot \mathbf{D}_{j_2} = \begin{pmatrix} x_2 & y_2 & z_2 \end{pmatrix}^T, z_2 \geq 0 \\ \mathbf{L}' &:= \mathbf{R}_0 \cdot \mathbf{L} = \begin{pmatrix} x_L & y_L & z_L \end{pmatrix}^T \end{aligned} \tag{A.12}$$

Using this transformation statement target (Eq. A.11) can be rephrased using Eq. A.3 and Eq. A.10.

$$\begin{aligned}
P_0 = \frac{\mathbf{L}^T \cdot \mathbf{E}_{1,2}}{\mathbf{D}_{j_0}{}^T \cdot \mathbf{E}_{1,2}} &= \frac{\mathbf{L}^T \cdot (\mathbf{D}_{j_1} \times \mathbf{D}_{j_2})}{\mathbf{D}_{j_0}{}^T \cdot (\mathbf{D}_{j_1} \times \mathbf{D}_{j_2})} \geq -1 \\
\Leftrightarrow \mathbf{L}^T \cdot (\mathbf{D}_{j_1} \times \mathbf{D}_{j_2}) &\geq -\mathbf{D}_{j_0}{}^T \cdot (\mathbf{D}_{j_1} \times \mathbf{D}_{j_2}) \\
\Leftrightarrow (\mathbf{L} + \mathbf{D}_{j_0})^T \cdot (\mathbf{D}_{j_1} \times \mathbf{D}_{j_2}) &\geq 0 \\
\Leftrightarrow \left(\mathbf{L}' + \mathbf{D}'_{j_0}\right)^T \cdot \left(\mathbf{D}'_{j_1} \times \mathbf{D}'_{j_2}\right) &\geq 0 \\
\Leftrightarrow \begin{pmatrix} x_L + 1 & y_L & z_L \end{pmatrix} \cdot \begin{pmatrix} y_1 \cdot z_2 \\ -x_1 \cdot z_2 \\ x_1 \cdot y_2 - y_1 \cdot x_2 \end{pmatrix} &\geq 0 \\
\Leftrightarrow (x_L + 1) \cdot y_1 \cdot z_2 - y_L \cdot x_1 \cdot z_2 + z_L \cdot (x_1 \cdot y_2 - y_1 \cdot x_2) &\geq 0 \\
\Leftrightarrow (x_L + 1) \cdot (y_1 \cdot z_2) - (x_1) \cdot (y_L \cdot z_2 - z_L \cdot y_2) - (z_L \cdot y_1) \cdot (x_2) &\geq 0
\end{aligned} \tag{A.13}$$

Hence it suffices to prove

$$\begin{aligned}
& (x_L + 1) \cdot (y_1 \cdot z_2) \geq 0 \\
\wedge\ & (x_1) \cdot (y_L \cdot z_2 - z_L \cdot y_2) \leq 0 \\
\wedge\ & (z_L \cdot y_1) \cdot (x_2) \leq 0
\end{aligned} \tag{A.14}$$

Using Eq. A.4 it is clear that

$$\begin{aligned}
& \mathbf{L}^T \cdot \mathbf{E}_{2,0} \geq 0 \Leftrightarrow \mathbf{L}^T \cdot (\mathbf{D}_{j_2} \times \mathbf{D}_{j_0}) \leq 0 \Leftrightarrow \mathbf{L}'^T \cdot \left(\mathbf{D}'_{j_2} \times \mathbf{D}'_{j_0}\right) \leq 0 \\
\Leftrightarrow\ & x_L \cdot (0) + y_L \cdot (z_2) + z_L \cdot (-y_2) \leq 0
\end{aligned} \tag{A.15}$$

$$\begin{aligned}
& \mathbf{L}^T \cdot \mathbf{E}_{0,1} \geq 0 \\
\Leftrightarrow\ & x_L \cdot (0) + y_L \cdot (0) + z_L \cdot (y_1) \leq 0
\end{aligned} \tag{A.16}$$

Since we ensured the dihedral angle between all pairs of neighbouring faces to be larger than $\pi/2$ in Eq. A.11 and assumed the DOP-orientations to be a right-hand system with Eq. A.2 it results:

$$\mathbf{D}_{j_0}{}^T \cdot \mathbf{D}_{j_1} \geq 0 \Leftrightarrow \mathbf{D}'_{j_0}{}^T \cdot \mathbf{D}'_{j_1} \geq 0 \Leftrightarrow x_1 \geq 0 \tag{A.17}$$

$$\mathbf{D}_{j_2}{}^T \cdot \mathbf{D}_{j_0} \geq 0 \Leftrightarrow \mathbf{D}'_{j_2}{}^T \cdot \mathbf{D}'_{j_0} \geq 0 \Leftrightarrow x_2 \geq 0 \tag{A.18}$$

$$\mathbf{D}_{j_0}{}^T \cdot (\mathbf{D}_{j_1} \times \mathbf{D}_{j_2}) \geq 0 \Leftrightarrow \mathbf{D}'_{j_0}{}^T \cdot \left(\mathbf{D}'_{j_1} \times \mathbf{D}'_{j_2}\right) \geq 0 \Leftrightarrow y_1 \cdot z_2 \geq 0 \tag{A.19}$$

The normalisation of L results

$$\begin{aligned}
|\mathbf{L}| = \left|\mathbf{L}'\right| = x_L^2 + y_L^2 + z_L^2 = 1 \\
\Rightarrow x_L^2 \leq 1 \Rightarrow x_L + 1 \geq 0
\end{aligned} \tag{A.20}$$

Combining Eq. A.15-A.20 with Eq. A.14 proves $-1 \leq P_i$. And with A.10 it follows that

$$-1 \leq P_i \leq 0 \tag{A.21}$$

A.2 Bounding Cross Sums of Mapping Vectors

In Sec. 8.1.2 an upper bound for the fixed-point error is calculated. The proof constraints all dihedral angles between all pairs of neighbouring faces to exceed $\pi/2$. It is claimed that this ensures that the distance r_{max} of the origin to the outermost vertex v_{min} of the unity-DOP (which is the DOP with maximum coefficients) is bounded by $\sqrt{3}$. This itself implies a bound on the cross sum of any mapping vector used.

Let $\mathbf{D}_{j_0}$, $\mathbf{D}_{j_1}$ and $\mathbf{D}_{j_2}$ to be the DOP-orientations whose according faces meet in $\mathbf{v}^{min}$.

It remains to proof the claim

$$\begin{aligned}&((\mathbf{D}_{j_1} \cdot \mathbf{D}_{j_2} \geq 0) \ \wedge \ (\mathbf{D}_{j_2} \cdot \mathbf{D}_{j_0} \geq 0) \ \wedge \ (\mathbf{D}_{j_0} \cdot \mathbf{D}_{j_1} \geq 0)) \\ &\Rightarrow r_{max} \leq \sqrt{3}\end{aligned} \tag{A.22}$$

One more time it is assumed that all orientations and DOP coefficients are normalised.

$$\begin{aligned}|\mathbf{D}_{j_0}| = |\mathbf{D}_{j_1}| = |\mathbf{D}_{j_2}| &= 1 \\ -1 \leq a_i' &\leq 1\end{aligned} \tag{A.23}$$

As shown in Figure A.1 the three DOP-orientations form a three-sided pyramid. Its base is a triangle connecting the ends of the orientation-vectors. r_{max} can be interpreted as the greatest distance from the origin to any point $\mathbf{v}^{min}$ on the unity-DOP surface (all DOP-coefficients equal one). And since all face-orientations have the same length, the vector from the origin to $\mathbf{v}^{min}$ intersects the base-triangle in the circumcentre $\mathbf{C}$. Let d be the distance from a triangle vertex to $\mathbf{C}$ and w be the distance from a triangle vertex to $\mathbf{v}_{min}$. Furthermore let r and c be the length of the vectors $\mathbf{v}_{min}$ and $\mathbf{C}$:

$$\begin{aligned}d &:= |\mathbf{D}_{j_i} - \mathbf{C}| \\ w &:= \left|\mathbf{D}_{j_i} - \mathbf{v}^{min}\right| \\ c &:= |\mathbf{C}| \\ r &:= \left|\mathbf{v}^{min}\right|\end{aligned} \tag{A.24}$$

Now $r = c + (r - c)$ and hence calculating r amounts to:

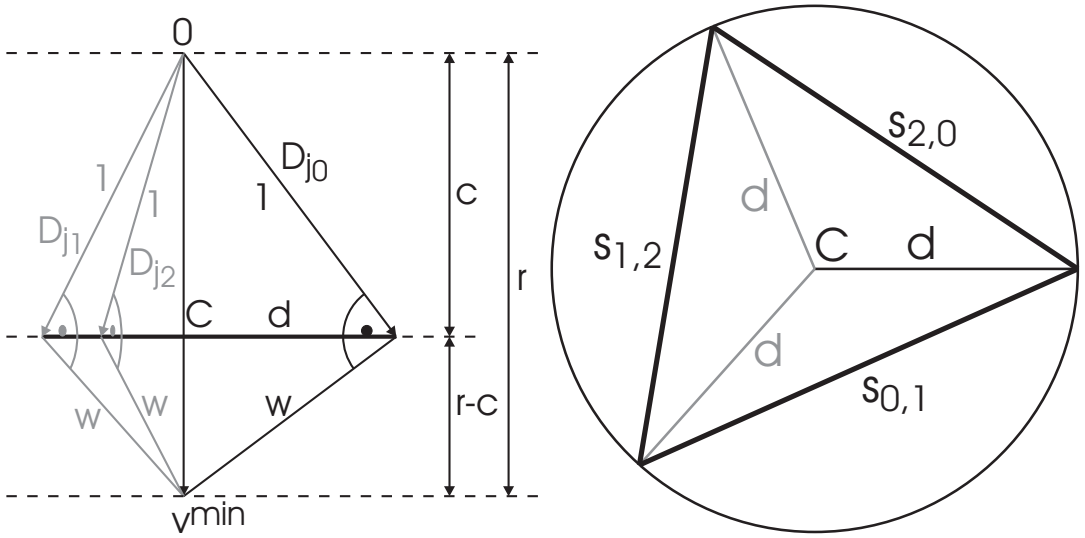


Figure A.1: Three DOP-orientations form a three-sided pyramid. Its base is a triangle between the ends of the orientation-vectors.

$$
\begin{aligned}
& 1 + w^2 = r^2 \ \wedge\ c^2 + d^2 = 1 \ \wedge\ (r-c)^2 + d^2 = w^2 \\
\Rightarrow & 1 + w^2 = r^2 \ \wedge\ c = \sqrt{1-d^2} \ \wedge\ r - c = \sqrt{w^2 - d^2} \\
\Rightarrow & 1 + w^2 = r^2 \ \wedge\ r^2 = 1 - d^2 + 2 \cdot \sqrt{(1-d^2)\cdot(w^2-d^2)} + w^2 - d^2 \\
\Rightarrow & 1 + w^2 = r^2 \ \wedge\ 0 = 2 \cdot \sqrt{(1-d^2)\cdot(w^2-d^2)} - 2 \cdot d^2 \\
\Rightarrow & 1 + w^2 = r^2 \ \wedge\ d^4 = w^2 - d^2 - w^2 \cdot d^2 + d^4 \\
\Rightarrow & 1 + w^2 = r^2 \ \wedge\ w^2 = \frac{d^2}{1-d^2} \\
\Rightarrow & r^2 = \frac{1}{1-d^2}
\end{aligned}
\tag{A.25}
$$

Note, that r increases monotonically with d.

Claim: The circumcentre of the corresponding triangle lies inside that triangle.

Proof by contradiction: Let the circumcentre be outside the triangle. It follows that it exists a normal $\mathbf{n}$ of a pyramid-face pointing out of the pyramid, but is directed like $\mathbf{v}^{min}$ ($\mathbf{v}^{min} \cdot \mathbf{n} > 0$). Since this normal is the normalised cross-product of two orientations, there is an edge $\mathbf{E}$ of the DOP that has the same direction as $\mathbf{n}$. Now let $\overline{\mathbf{v}} := \mathbf{v}^{min} + \mathbf{E}$ be the DOP-vertex at the other end of the edge. Then its distance to the origin is greater than that of v_{min}, contradicting the definition of $\mathbf{v}^{min}$ to have maximum distance to the origin.

$$\begin{aligned} \mathbf{v}^{min} \cdot \mathbf{n} &> 0 \\ \Rightarrow \mathbf{v}^{min} \cdot \mathbf{E} &> 0 \\ \Rightarrow \left(\mathbf{v}^{min}\right)^2 + 2 \cdot \mathbf{v}^{min} \cdot \mathbf{E} + \mathbf{E}^2 &> \left(\mathbf{v}^{min}\right)^2 \\ \Rightarrow \left(\mathbf{v}^{min} + \mathbf{E}\right)^2 &> \left(\mathbf{v}^{min}\right)^2 \\ \Rightarrow \left|\mathbf{v}'\right| = \left|\mathbf{v}^{min} + \mathbf{E}\right| &> \left|\mathbf{v}^{min}\right| \end{aligned} \tag{A.26}$$

Next assume that the angle between any pair of the three orientations is bounded by α_{max}. This yields an upper bound on the sides $s_{0,1}$,$s_{1,2}$,$s_{2,0}$ of the triangle between the orientations.

$$\begin{aligned} &\mathbf{D}_{j_0} \cdot \mathbf{D}_{j_1} \geq \cos(\alpha_{max}) \\ &\Rightarrow s_{0,1} \leq 2 \cdot \sin\left(\frac{\alpha_{max}}{2}\right) = \sqrt{2 - 2 \cdot \cos\left(\alpha_{max}\right)} \end{aligned} \tag{A.27}$$

Under these preconditions, the triangle with the greatest circumcircle-radius is equilateral with maximum side-length. Since greater circumcircles yield farther DOP-vertices, calculating r_{max} amounts to

$$\begin{aligned} s = s_0 = s_1 = s_2 &= \sqrt{2 - 2 \cdot \cos(\alpha_{max})} \\ \Rightarrow d_{max} = \frac{s}{\sqrt{3}} &= \sqrt{\frac{2 - 2 \cdot \cos(\alpha_{max})}{3}} \\ \Rightarrow r_{max} = \sqrt{\frac{1}{1 - d_{max}^2}} &= \sqrt{\frac{3}{1 + 2 \cdot \cos(\alpha_{max})}} \end{aligned} \tag{A.28}$$

Ensuring the dihedral angle between all pairs of neighbouring faces to exceed $\pi/2$, it results

$$r_{max} = \sqrt{\frac{3}{1 + 2 \cdot \cos(\pi/2)}} = \sqrt{3} \tag{A.29}$$

Appendix B

Generation of Test Axes

As discussed in Sec. 8.1.1, a full separating axis test requires testing of all axes orthogonal to a face of either k-DOP or orthogonal to an edge of each k-DOP.

Since a k-DOP has $v_k = 8 + 2(k-6) = 2k-4$ vertices (8 for a basic 6-DOP, and additional 2 for every further face) this results in $v_{24} = 44$ vertices for 24-DOPs.
Using the Euler characteristic this results in 66 edges. Due to the DOP property of antiparallel faces for each edge there also is an antiparallel one. This results in 12 face orientations and 33 edge orientations per DOP.
Hence a full separating axis test for 24-DOPs requires $N = 12 + 12 + 33^2 = 1113$ test axes. Since these axes are precomputed this is not time critical and hence is done brute force.[1]
Firstly, all face intersections are computed and the real vertices are selected by testing which are not outside the halfspaces spanning the DOP.
Secondly, all vertices sharing 2 faces are selected to form the edges. Antiparallel ones are abandoned.
Thirdly, cross-products of edges are calculated to result the test axes.
Since they are usually not all different within the given precision of 32-bit an additional optimisation is to test them pairwise for equality. The discussion in Sec. 8.3.1 shows that the vast majority of test axes is not used anyway. Hence this of no practical relevance, and is mentioned only for the sake of completeness.

[1] The source code uses parts of the *CollDet library* [71, 83, 84].