Synthesis of Interfaces and Communication in Reconfigurable Embedded Systems

Michael Eisenring and Marco Platzner
Computer Engineering and Networks Lab
Swiss Federal Institute of Technology (ETH) Zurich

Address: Michael Eisenring, Marco Platzner
Computer Engineering and Networks Lab
Swiss Federal Institute of Technology (ETH) Zurich
ETZ G85, Gloriastrasse 32
8092 Zurich, Switzerland

Email: eisenring@tik.ee.ethz.ch,
platzner@computer.org

Short title: Communication Synthesis in Reconfigurable Systems

Abstract

Reconfigurable devices are used in different ways for the design of embedded systems. As reconfigurable computing is a rather new field, there is still a lack of design methodologies and tools. Previous work on design automation for reconfigurable systems focused on partitioning and scheduling. We present an approach to communication synthesis in reconfigurable embedded systems. Using the proposed methodology, the designer can focus on the functional behavior of the design and on the evaluation of different mappings. All the low-level details of the reconfigurable resource are hidden.

In this paper, we first classify communication types in reconfigurable systems, covering both compile-time and run-time reconfiguration. Then we describe our tool suite CORES/HASIS that generates interface circuitry to connect communicating tasks in reconfigurable systems. The tasks may be mapped onto the same or onto different FPGAs. The tools multiplex several logical communication channels over one physical channel, insert routing nodes to connect tasks on non-adjacent devices, and provide dedicated interfaces that solve the problem of interconfiguration communication. A case study demonstrates the usability and efficiency of our approach.

Synthesis of Interfaces and Communication in Reconfigurable Embedded Systems

1 Introduction

Reconfigurable devices, mostly Field-Programmable Gate Arrays (FPGAs), are used for embedded systems in different ways. Applications include so-called *compile-time reconfigurable* (CTR) systems where the goal is to combine ASIC performance with processor flexibility [1], emulation and rapid prototyping systems where FPGAs quickly implement hardware functions to allow for validation and design space exploration, and *run-time reconfigurable* (RTR) systems [2] where an embedded application is split into time-exclusive segments (configurations) that are executed sequentially. RTR can reduce cost significantly compared to an all-in-ASIC or an all-in-FPGA solution [3] [4]. According to the FPGA technology used, RTR systems further divide into fully and partially reconfigurable systems. Full reconfiguration means that the reprogramming process affects complete FPGA devices. Partially reconfigurable FPGAs allow to alter only parts of the circuitry, while the remaining parts remain executing.

As reconfigurable computing is a rather new field, there is still a lack of design methodologies and tools that reflect the characteristics of reconfigurable devices. Most proposed approaches follow a hardware/software codesign methodology, where a formal specification that includes the problem, the target architecture, and constraints is transformed into an implementation by partitioning and scheduling steps.

In our work, we focus on communication and interface synthesis for reconfigurable systems. We assume that a specification is partitioned onto several FPGAs and into

several configurations, and a configuration schedule is determined. After this, our communication and interface synthesis tool set CORES/HASIS automatically generates and inserts the necessary interfaces. CORES/HASIS relieve the designer from manual interface construction, which is extremely tedious for multi-FPGA systems. Our tool set also generates interfaces between objects that exist in different configurations, i.e., objects that do not necessarily exist at the same time. Hence, we offer a solution to the problem of interconfiguration communication in RTR systems [2].

In the remainder of this paper, we discuss related work and a taxonomy of basic communication types in reconfigurable systems. Then we explain our approach for communication synthesis, present a case study as a proof of concept, and discuss experimental results.

2 Related work

Communication synthesis is one of the major issues in hardware/software codesign [5]. Communication has been investigated for many different targets, ranging from processors connected via common buses [6] to intellectual property (IP) components [7]. Generally, there are two approaches to communication synthesis. The first approach instantiates interface components from a library of predefined communication primitives, e.g., [8]. The second approach treats communication synthesis as an explicit step in the system design flow and synthesizes interface circuitry from a formal specification. For example, in [9] communication synthesis is cast as an allocation problem and in [10] the communication protocol is a design parameter.

A number of codesign frameworks support communication synthesis: Chinook [11]

maps a behavioral description of communicating processes onto one or several processors and generates software drivers and glue logic to connect the processors with external devices. Cosyma [12] and Polis [13] are examples that map system specifications onto a processor-coprocessor architecture. HiPART [14] and ipChinook [15] target heterogeneous platforms consisting of processor, ASIC, and FPGA modules. CoWare [16] focuses on systems-on-a-chip.

For reconfigurable systems, [2] gives a classification into CTR and RTR systems and outlines the problem of interconfiguration communication. The usefulness of RTR has been demonstrated by several projects, including [17] and [18]. Recently, a number of approaches have been reported for the partitioning and scheduling of applications onto reconfigurable systems, e.g., [19] [20].

Communication synthesis for reconfigurable systems has not been investigated yet sufficiently. Although above frameworks synthesize interfaces for FPGAs operated in CTR mode, they focus on specific target architectures and specification formalisms and thus lack in flexibility. More importantly, none of the codesign frameworks focuses on RTR systems. Up to our best knowledge, there currently exists no communication synthesis tool that is able to generate interface circuits for both the CTR and RTR models. Our CORES/HASIS tool suite copes with both reconfiguration models as well as hardware/software interface generation for heterogeneous systems consisting of an arbitrary number of processors, FPGAs, and memories.

3 Communication types

The *specification* for CORES/HASIS includes a problem graph and an architecture graph. The problem graph is a directed graph with two kinds of nodes: tasks and queues. Tasks are coarse-grained executable units, e.g., IP (Intellectual Property) blocks, that read data from and write data to ports [21]. Queues are buffers with FIFO semantics that connect tasks. Edges in the problem graph denote data flow. The architecture graph connects FPGAs, memories, and buses. In the *partitioning* step, each task and queue of the problem graph is mapped onto some architectural component. Tasks are always bound to FPGAs, whereas queues can be implemented in FPGAs or memories. In CTR systems, there exists only one configuration for each FPGA. Tasks are active after download and can be blocked on a read, if the required data is not yet available. In RTR systems, each FPGA may undergo a sequence of configurations. Here, a *scheduling* step is required that assigns a starting condition to each configuration. Usually, a new configuration can be loaded if all the tasks in the current configuration have completed. To that end, a task raises a done signal on completion. All the individual done signals are combined to form the configuration's done signal. During *communication synthesis*, interface and routing nodes are added to the problem graph. Interfaces nodes establish FPGA-FPGA, FPGA-memory, and FPGA-processor communication and can multiplex several logical communication channels over a limited number of physically available wires. Routing nodes are required when two communicating objects are mapped onto non-adjacent FPGAs. Interface- and routing nodes are always bound to FPGAs.

Figure 1 shows a taxonomy of communication types for the most basic problem graph, the task-queue-task system. This taxonomy results from the possible mappings of tasks

and queues, sketched along three orthogonal axes that determine i) if the tasks are bound to the same or different FPGAs, ii) if the queue is assigned to an FPGA or a memory, and iii) if the FPGAs undergo one or several configurations. CTR systems have only one configuration which restricts the communication types to *on-chip* and *off-chip* communication (cases 1, 2, 5, and 6 in Fig. 1). In RTR systems, tasks may communicate over configuration borders. Usually, the designer will include intermediate queues assigned to memories (cases 7 and 8) to hold data between subsequent configurations. However, if the design does not include such queues or the queues are assigned to FPGAs that undergo reconfiguration, either *interconfiguration communication with open channel* (case 4) appears or a partially reconfigurable FPGA is required (case 3). In the current version of our tool, we do not support partially reconfigurable FPGAs.

4 Communication synthesis

The CORES/HASIS tool suite forms the back-end of a design flow for reconfigurable systems (see Figure 2). Design capture, partitioning, and scheduling are done by front-end tools specific to the used specification formalism, e.g., [19] [20]. The input to CORES/HASIS is a problem specification consisting of a fully partitioned and scheduled problem graph and an architecture graph. This clear separation between front-end and back-end allows to map applications specified in a range of formalisms to arbitrary target architectures. In our implementation, front-end and back-end tools are coupled via text file interfaces.

CORES first refines the problem graph by inserting interface nodes into the data flow edges and then calls HASIS [21] [22], our hardware/software codesign tool for com-

munication synthesis. HASIS was developed to generate interfaces for heterogeneous systems, i.e., systems consisting of processors and FPGAs. Therefore, HASIS can handle all on/off-chip communication types. For RTR systems with several configurations, CORES inserts appropriate interface nodes and splits the problem graph into a series of subgraphs, one for each configuration. These subgraphs then contain only on/off-chip communication, which again is handled by HASIS. The result of the tool flow is a series of circuit descriptions in VHDL, one for each configuration and FPGA. These descriptions are synthesized for a specific FPGA target by commercial tools. Before this final synthesis step we do not know whether the circuits will actually fit onto the FPGAs. If this step fails, an error is reported. We provide, however, coarse area estimates for different types of interfaces. These estimates may be used by front-end tools to support design decisions and facilitate design space exploration.

4.1 On/off-chip communication

HASIS uses a hybrid approach between library-based interface instantiation and synthesis from a formal specification. Figure 3 shows the basic concept of HASIS' interface generation. For each interface type, HASIS provides a template and a corresponding generator method. An interface is constructed from an interface specification by combining parameterization of predefined templates with synthesis of additional control logic.

The core of HASIS is an object-oriented structure that combines classes of physical devices and interface generation methods. This class structure can easily be adapted to various target architectures. Figure 4 displays a part of the class inheritance structure with two kinds of classes: concrete classes which denote implementations of physical

devices and generator methods, and abstract classes which describe general properties of these implementations. An example for an abstract class is `memory` with the attribute `size`; an example for a concrete class is `XILINX XC4xxx`. Figure 4 shows four sections:

- **Devices** are the physical components that appear in the architecture graph, e.g., an FPGA XC4xxx or the SRAM MCM6206.
- **Component models** are abstractions of physical devices. For example, the XILINX XC4xxx is an FPGA and therefore a computing resource. One of the properties of computing resources is the capability of having interface generators.
- **Queue models** subsume queue types and according generator methods for different devices. For example, the concrete class FIFO MCM6206 is a generator method that provides FIFO queue implementations for the memory device MCM6206.
- **Interface models** are abstract and concrete classes that describe and implement different communication types. For example, the generator method MuxFF (**M**ultiplexer for **F**PGA-**F**PGA connection) produces an interface between two adjacent FPGAs that multiplexes several communication channels over a given number of I/O pins.

Task implementation HASIS assumes that each task is stored in a library and requires a task interface specification in VHDL. Figure 5a) shows a task implementation with one input and one output port. The core represents the task's functionality and is wrapped by small protocol state machines (FSMs) that establish data transmission on the input/output channels. On termination the task raises its done signal.

Interface and queue implementations HASIS provides interfaces and queues for different physical devices. For example, Figure 5b) shows the template for an interface that connects to queues in SRAM. The template wraps control logic for read/write and access control around a dedicated SRAM device. There are $n + 1$ read/write channels that compete for SRAM access. A scheduler resolves conflicts by a fixed priority scheme that is a parameter of the interface generator.

Routing and multiplexing Multiplexors are interfaces that multiplex several communication channels between FPGAs when the number of available I/O pins is limited. Routing is necessary if two communicating tasks are bound to non-adjacent FPGAs. A routing task acts like a blocking queue with depth one. The control logic of a routing task allows to enable/disable data transmission. An enabled routing task is fully transparent for both communicating tasks. Disabling a routing task blocks the sender immediately and the receiver the next time the queue is empty. This guarantees that the routing FPGA can be reconfigured without any data loss in the queue.

4.2 Interconfiguration communication

Figure 6a) shows a problem graph with the three tasks A, B and C bound to one FPGA. Task A forms configuration 1, tasks B and C are assigned to configuration 2, and the queues q_1 and q_2 are mapped to memory (case 7 in the taxonomy of Figure 1). CORES splits the problem graph into one subgraph per configuration and inserts interface nodes. Figure 6b) shows that interfaces I_1 and I_2 are inserted into configuration one and implement write channels to the queues mapped to memory. In configuration 2, the interface nodes I_3 and I_4 are inserted, which implement read channels. The write

and read interfaces are parameterized with the addresses and sizes of the queues. HASIS then generates circuitry for two configurations, each one having an SRAM interface such as that shown in Figure 5b).

If a queue connecting tasks in different configurations is bound to one of the reconfigured FPGAs or if there is no queue specified at all, interconfiguration communication with open channel occurs. This requires proper FPGA device characteristics, such as high-impedance states of I/O pins during reconfiguration. Figure 7a) shows an example problem graph with task A assigned to configuration 1 of FPGA 1 and tasks B and C assigned to configurations 1 and 2 of FPGA 2, respectively. The queue q_1 is assigned to configuration 1 of FPGA1 and queue q_2 to configuration 1 of FPGA2. The edges q_1-B and A-q_2 cross two configuration borders, which denotes that the underlying communication type will have to use an open channel (case 4 in the taxonomy of Figure 1). Again, CORES splits the problem graph into subgraphs and forces HASIS to generate the interface circuits. CORES/HASIS do not make any assumptions about blocking conditions or the sequence of read/write accesses for the queues. It is the responsibility of the front-end tools to ensure correct and dead-lock free operation.

5 Case study

As case study we have implemented a steganography [23] application on the Compaq/Digital Pamette board [24] (see Figure 8). The purpose of the case study was to provide proof of concept.

Steganography is the art of hiding a message, e.g., an ASCII text stream, within a stream of image data in an unsuspicious way (see Figure 9a). That means, one cannot *see*

that the image has been altered. The problem graph is presented in Figure 9b). The tasks V and T provide the image and text streams, respectively. The task S (steganography) hides the text in the images employing the Least Significant Bit (LSB) insertion method. The resulting data stream is compressed by a run-length coder C. The problem graph has two parallel processing lines in order to increase the throughput. The two output streams are finally merged by the task J (join).

We discuss two sample mappings, a CTR and an RTR mapping. For both cases, V and T are mapped to the PC and the resulting bitstream is directed to an external connector attached to FPGA2. In the CTR example, all tasks S, J, and C and queues $q1, \ldots, q6$ are bound to FPGA0. FPGA2 performs routing and the other two FPGAs are not used. In the RTR example, both FPGA0 and FPGA1 have two configurations. FPGA0 first implements S and $q2$, and then C; FPGA1 implements first S and then C. FPGA2 implements J; FPGA3 performs routing. All the queues, except $q2$, are mapped to memory.

Figure 10 presents the hardware cost in number of FPGA logic blocks (CLBs) for these mappings. In the CTR example, the queues dominate because all queues have been mapped to FPGA0. Hardware requirements for interfaces and routing nodes are very small. The overall hardware cost for the CTR mapping is 751 CLBs. For the RTR mapping the overall hardware cost is 681 CLBs. Considering the fact that here many queues are mapped to memory, one would expect a much smaller hardware cost. The reason for the relatively high hardware cost is that the SRAM interfaces that connect to the queues in memory are costly as they store queue addresses, rather long read/write pointers, and implement access control and arbitration among several channels. For

example, FPGA1 in configuration 1 implements an interface to SRAM1 with overall 5 channels. Two channels write data from tasks V and T (via routing tasks) into SRAM1, two channels read the data and feed it into task S, and one channel writes the result of S back into SRAM1.

6 Conclusion

We outlined a design methodology and discussed different types of communication in embedded reconfigurable systems. We presented the communication synthesis tool suite CORES/HASIS, which forms the back-end of a reconfigurable system design flow, and experimental results from a case study.

The main advantage of our tool suite is the greatly shortened and simplified design process for reconfigurable systems. We replace the time consuming and error-prone manual interface implementation by a flexible and fully automated generation of interfaces. This will facilitate sound design space exploration for run-time reconfigurable systems.

A manual implementation of the case study would certainly use interface circuits more specialized to the problem at hand. Although we did not compare our results with a manual design, an inspection of the generated circuitry indicated that the hardware overhead induced by automatic interface generation is reasonable. This is mainly due to our approach to interface synthesis that combines the flexibility of synthesis with instantiation of pre-optimized library components.

A lesson learned through experimenting is that besides partitioning and scheduling, the mapping of queues plays a crucial role for the quality of the final design. Generally, queues in memory are more efficient than in FPGAs in terms of hardware area. This

means that longer queues should preferably be mapped to memory. However, when several queues reside in one memory and are accessed over one memory bus, additional circuitry is required for access control and channel arbitration. This trade-off again points out the importance of a sound design space exploration for reconfigurable embedded systems.

Acknowledgment

We would like to thank Corneliu Tobescu for his contributions to this project and the fruitful discussions about interface implementations.

References

[1] BUELL, D. A. and POCEK, K. L. Custom Computing Machines: An Introduction. *The Journal of Supercomputing*, (9):219–229, 1995.

[2] HUTCHINGS, B. L. and WIRTHLIN, M. J. Implementation Approaches for Reconfigurable Logic Applications. In *International Workshop on Field-Programmable Logic and Applications*, pages 419–428, 1995.

[3] LEMOINE, E. and MERCERON, D. Run Time Reconfiguration of FPGA for Scanning Genomic DataBases. In *IEEE Symposium on FPGAs for Custom Computing Machines*, pages 90–98, 1995.

[4] SCHONER, B. and JONES, C. and VILLASENOR, J. Issues in Wireless Video Coding using Run-time-reconfigurable FPGAs. In *IEEE Symposium on FPGAs for Custom Computing Machines*, pages 85–89, 1995.

[5] O'NILS, M. and JANTSCH, A. Communication in Hardware/Software Embedded Systems - A Taxonomy and Problem Formulation. In *15th NORCHIP Seminar, Copenhagen, Denmark*, pages 67–74, November 1997.

[6] ORTEGA, R. B. and BORRIELLO, G. Communication Synthesis for Distributed Embedded Systems. In *IEEE/ACM Int'l Conf. on Computer-Aided Design*, pages 437–444, San Jose, CA, November 1998.

[7] ORTEGA, R. B. and LAVAGNO, L. and BORRIELLO, G. Models and Methods for HW/SW Intellectual Property Interfacing. In *NATO Advanced Study Institute on System-level Synthesis*, 1998.

[8] VAHID, F. and TAURO, L. An Object-Oriented Communication Library for Hardware-Software CoDesign. In *5th Int'l Workshop on Hardware/Software Codesign*, pages 81–86, Braunschweig, Germany, March 1997.

[9] DAVEAU, J. M. and ISMAIL, T. B. and MARCHIORO, G. and JERRAYA, A. A. Protocol selection and interface generation for HW-SW codesign. 5(1):136–144, March 1997.

[10] KNUDSEN, P. V. and MADSEN, J. Integrating Communication Protocol Selection with Partitioning in Hardware/Software Codesign. In *11th International Symposium on System Synthesis ISSS'98, Hsinchu, Taiwan*, pages 111–116, December 1998.

[11] CHOU, P. and ORTEGA, R. and BORRIELLO, G. Interface co-synthesis techniques for embedded systems. In *IEEE/ACM Int'l. Conf. on Computer-Aided Design*, pages 280–287, San Jose, CA, November 1995.

[12] ERNST, R. and HENKEL, J. and BENNER, T. and YE, W. and HOLTMANN, U. and HERRMANN, D. The COSYMA environment for hardware/software cosynthesis of small embedded systems. *Microprocessors and Microsystems*, 20(3):159–166, May 1996.

[13] BALARIN, F. and GIUSTO, P. and JURECSKA, A. and PASSERONE, E. and SENTOVICH, E. and TABBARA, B. and CHIODO, M. and HSIEH, H. and LAVAGNO, L. and SANGIOVANNI-VINCENTELLI, A. and SUZUKI, K. *Hardware-Software Co-Design of Embedded Systems: The Polis Approach.* Kluwer Academic Publisher, 1997.

[14] HOLLSTEIN, T. and BECKER, J. and KIRSCHBAUM, A. and GLESNER, M. HiPART: A new hierarchical semi-interactive hw/sw partitioning approach with fast debugging for real-time embedded systems. In *Sixth International Workshop on Hardware/Software Codesign*, pages 29–33, Seattle, WA, March 1998.

[15] CHOU, P. and ORTEGA, R. and HINES, K. and PARTRIDGE, K. and BORRIELLO, G. ipChinook: An Integrated IP-based Design Framework for Distributed Embedded Systems. In *36th Design Automation Conference*, New Orleans, LA, June 1999.

[16] VERCAUTEREN, S. and LIN, B. and DE MAN, H. Constructing Application-Specific Heterogeneous Embedded Architectures from Custom HW/SW Applications. In *33rd Design Automation Conference*, pages 521–526, June 1996.

[17] SANCHEZ, E. and SIPPER, M. and HAENNI, J. O. and BEUCHAT, J. L. and STAUFFER, A. and PEREZ-URIBE, A. Static and Dynamic Configurable Systems. *IEEE Transactions on Computers*, 48(6):556–564, June 1999.

[18] ELDREDGE, J. G. and HUTCHINGS, B. L. Run-Time Reconfiguration: A Method for Enhancing the Functional Density of SRAM-based FPGAs. *Journal of VLSI Signal Processing*, 12(1):67–86, January 1996.

[19] KAUL, M. and VEMURI, R. Optimal temporal partitioning and synthesis for reconfigurable architectures. In *Design, Automation and Test in Europe*, pages 389–396, 1998.

[20] PURNA, K. and BHATIA, D. Temporal Partitioning and Scheduling Data Flow Graphs for Reconfigurable Computers. *IEEE Transactions on Computers*, 48(6):556–564, June 1999.

[21] EISENRING, M. and TEICH, J. Domain-Specific Interface Generation from Dataflow Specifications. In *Sixth International Workshop on Hardware/Software Codesign*, pages 43–47, Seattle, WA, March 1998.

[22] EISENRING, M. and PLATZNER, M. and THIELE, L. Communication Synthesis for Reconfigurable Embedded Systems. In *9th International Workshop on Field-Programmable Logic and Applications*, pages 205–214, Glasgow, UK, August/September 1999.

[23] JOHNSON, N. F. and JAJODIA, S. Steganography: Seeing the unseen. *IEEE Computer*, pages 26–34, February 1998.

[24] MOLL, L. and SHAND, M. Systems performance measurement on PCI Pamette. In *IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 125–133, April 1997.

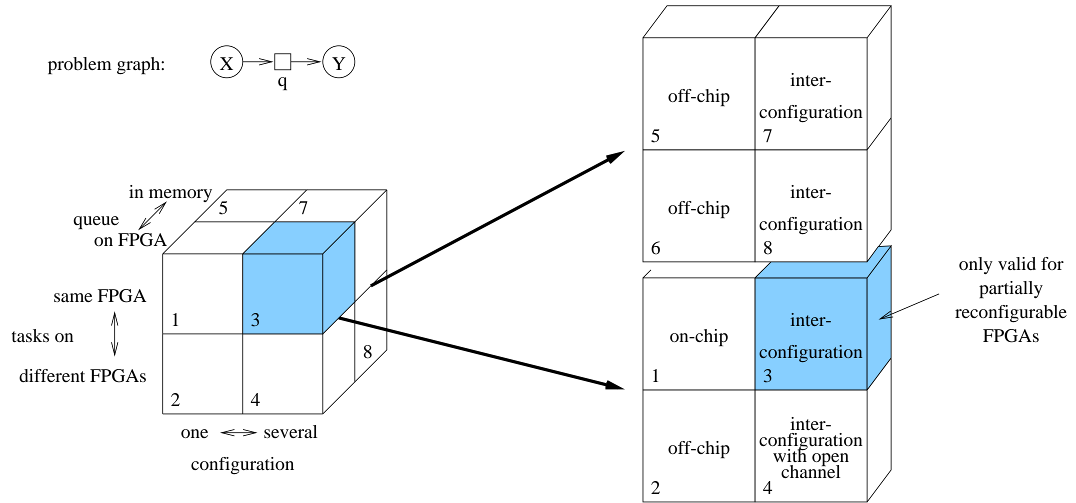

Figure 1: Taxonomy of communication types in reconfigurable systems. The problem graph consists of two tasks X and Y connected by a queue q.

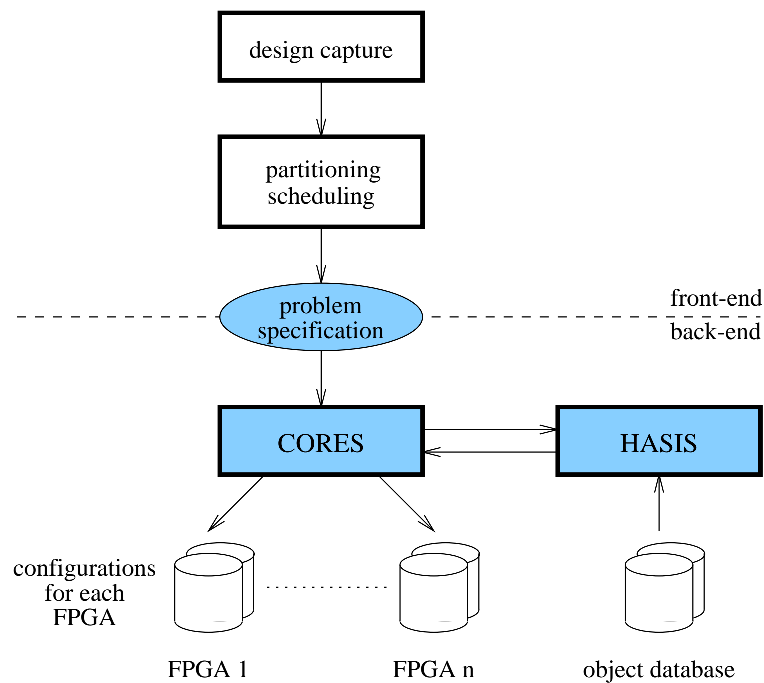

Figure 2: CORES/HASIS tool suite.

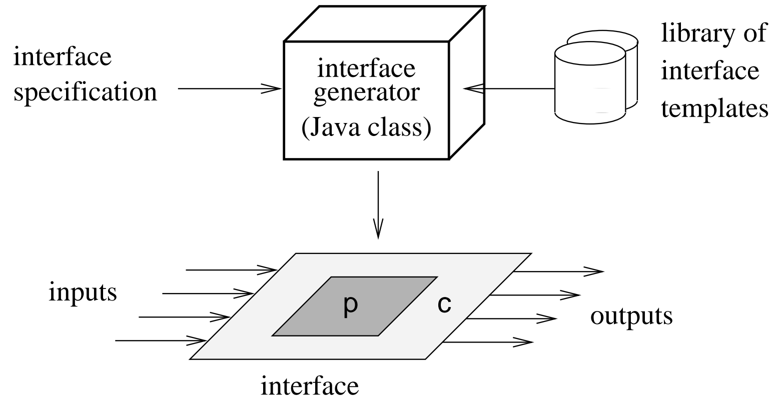

Figure 3: Concept of hybrid interface generation. The interface consists of parameterized templates (p) surrounded by synthesized control logic (c).

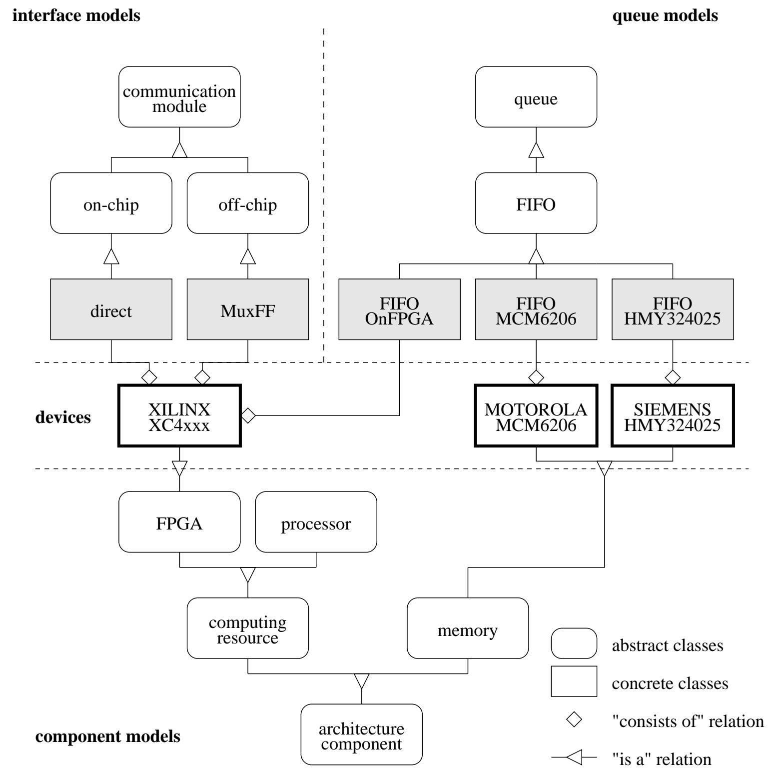

Figure 4: HASIS class tree structure. Bold boxes denote physical implementations, shaded boxes are methods that generate interfaces and queues, respectively.

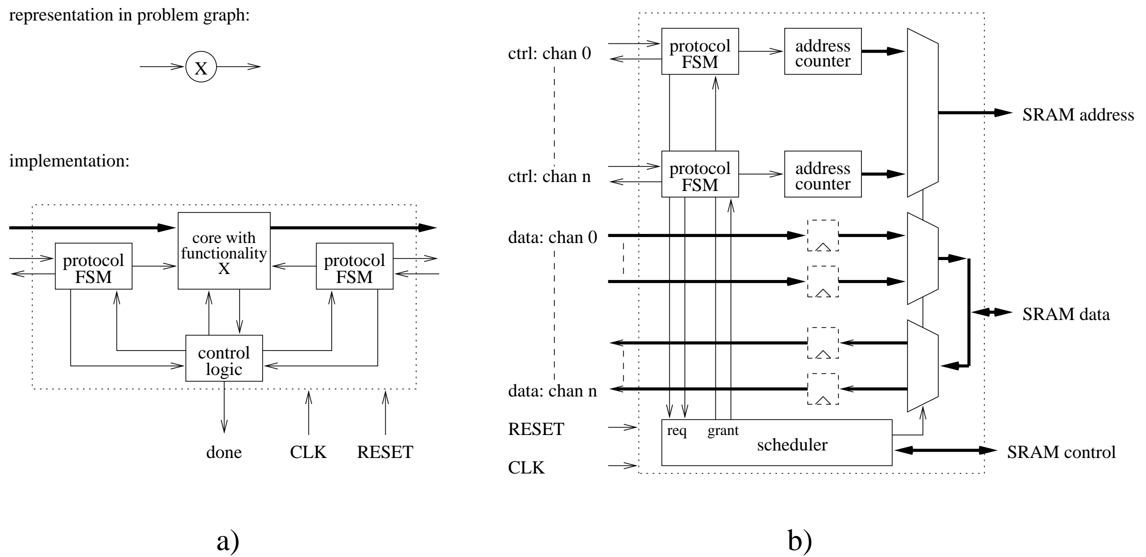

Figure 5: a) Task implementation b) Interface template for queues in SRAM.

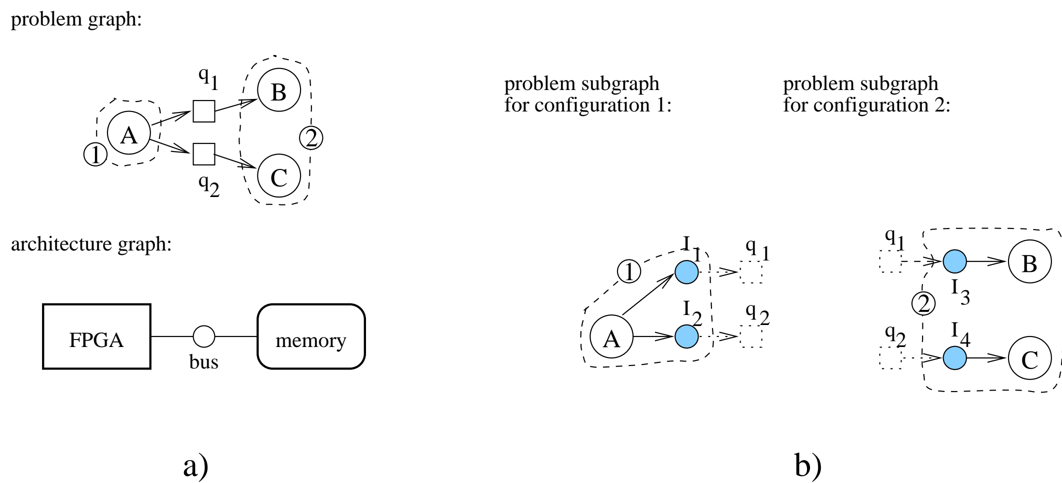

Figure 6: a) Problem specification consisting of a partitioned problem graph and an architecture graph b) Problem subgraphs corresponding to the configurations.

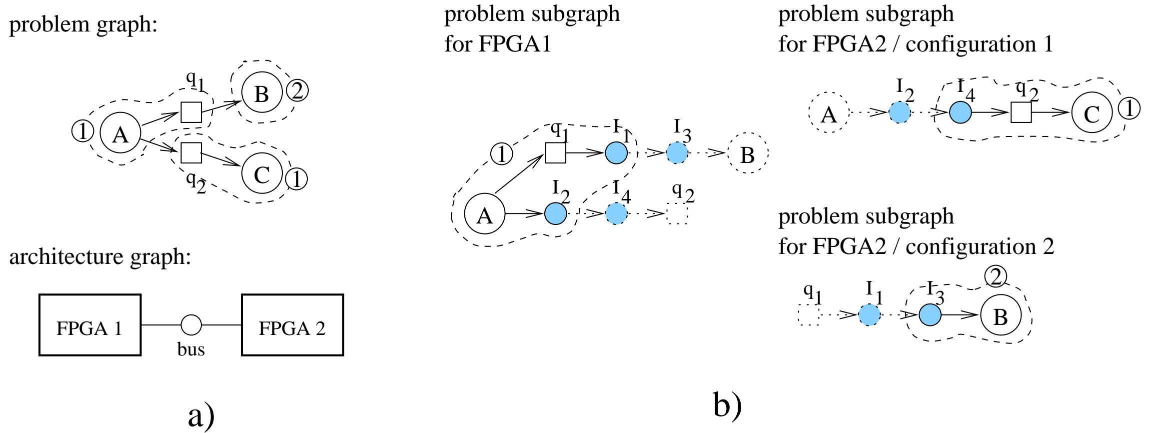

Figure 7: a) Problem specification consisting of a partitioned problem graph and an architecture graph b) Problem subgraphs for FPGA1 and the two configurations of FPGA2.

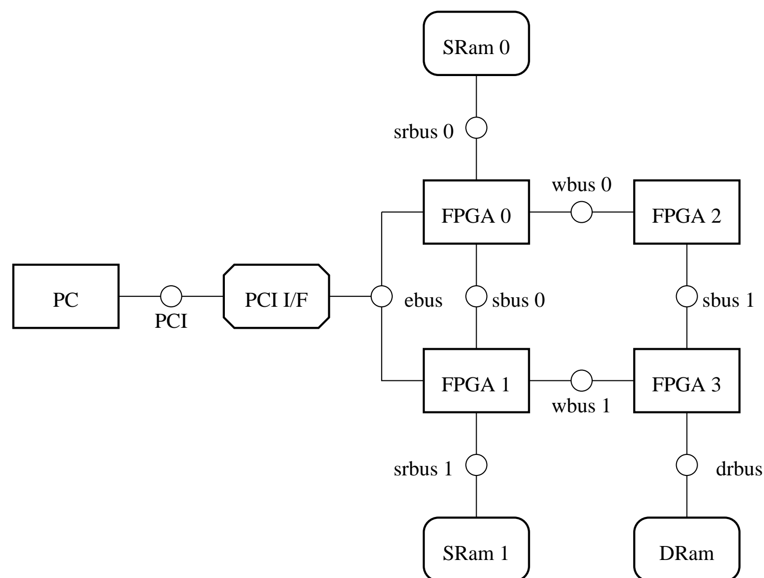

Figure 8: Architecture graph showing the Pamette board connected to a PC. The Pamette consists of a PCI interface and four user-programmable FPGAs, SRAM and DRAM memory chips, and various buses.

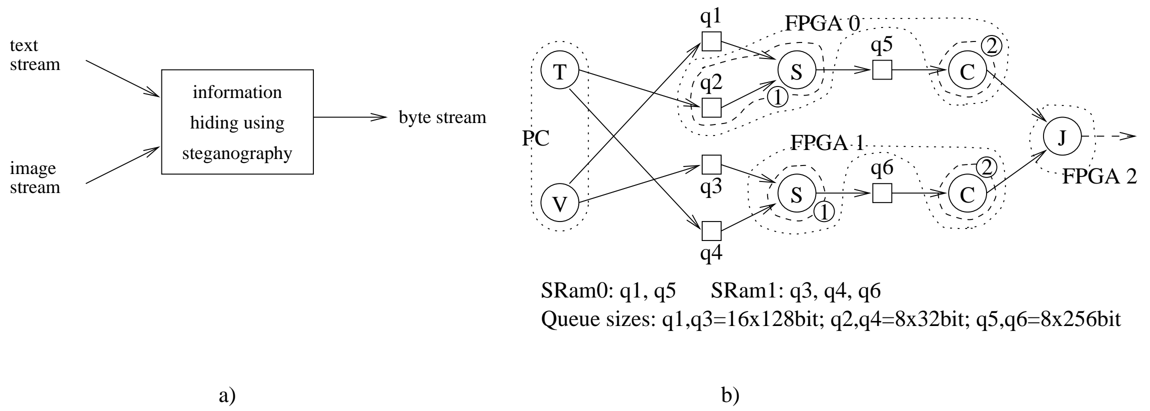

Figure 9: Steganography application: a) Principle b) Problem graph.

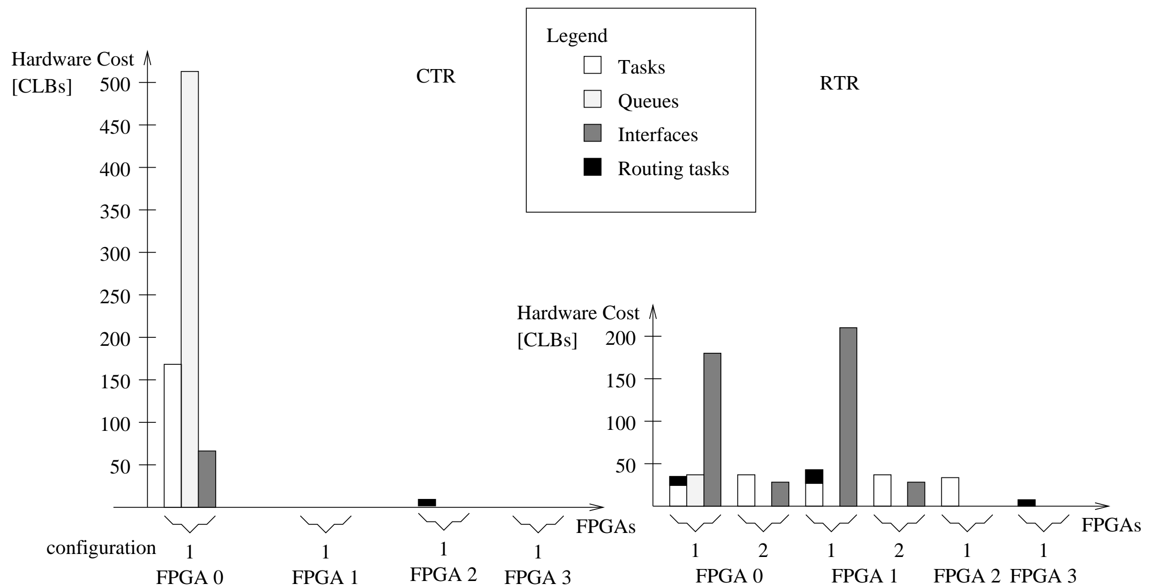

Figure 10: Hardware cost for each configuration and FPGA.