Communication Synthesis for Reconfigurable Embedded Systems

Michael Eisenring, Marco Platzner, and Lothar Thiele

Computer Engineering and Communication Networks Lab
Swiss Federal Institute of Technology (ETH)
email: {eisenring, platzner, thiele}@tik.ee.ethz.ch

Abstract. In this paper, we present a methodology and a design tool for communication synthesis in reconfigurable embedded systems. Using our design tool, 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, e.g., a multi-FPGA architecture, are hidden. After the application's tasks have been bound to FPGAs and a multi-configuration schedule has been generated, the communication synthesis step provides the necessary interfaces between tasks. Interface circuitry is automatically inserted to connect communicating tasks, whether they are mapped onto the same or onto different FPGAs. This includes multiplexing several logical communication channels over one physical channel, inserting routing tasks, and providing dedicated interfaces that solve the problem of interconfiguration communication.

1 Introduction

During the last years, reconfigurable computing has gained interest not only as a potentially new paradigm for general-purpose computing but also for embedded systems. Reconfigurable systems are often classified according to their reconfiguration model into *compile-time reconfiguration* (CTR) and *run-time reconfiguration* (RTR) [7]. In CTR, hardware compilation and reconfiguration, i.e., downloading the design onto a reconfigurable device, are done at compile time. In embedded systems, this reconfiguration model is mainly used for rapid prototyping. However, reconfigurable hardware can be a viable alternative to ASICs for the final system implementation when the expected volume is rather low and performance constraints are met. In RTR, the configuration of the reconfigurable device is changed while the application is running. There are two application scenarios for RTR in embedded systems: The first scenario are embedded systems where frequent hardware updates are expected, e.g., network switches that may be reconfigured to support different topologies [1]. Usually these systems operate in a time-critical execution mode and a not time-critical update mode. The second scenario are embedded applications that are split into time-exclusive parts. A reconfigurable resource is used to execute these parts sequentially. By this, cost can be significantly reduced compared to an all-in-ASIC or an all-in-FPGA solution [6] [13] [14].

To develop system design tools for embedded reconfigurable systems, some of the synthesis tasks have to be revisited: scheduling/binding and communication synthesis. For RTR systems, the scheduling/binding step must respect that communicating hardware tasks can be grouped into several configurations that are executed sequentially on the same resource. A reconfiguration schedule has to be found and a controller that initiates and controls reconfiguration has to be synthesized. This problem has already received some attention [2] [11]. The second major issue is communication synthesis. Communication synthesis has been attacked in several projects [8] [12]. However, for embedded reconfigurable systems the automatic generation of communication and interface circuitry leads to new problems and is of utmost importance for the following reasons:

First, the manual interface construction for *multi-FPGA* systems is extremely tedious. The designer is considered with all the low-level details, including the number and positions of available I/O pins, I/O pins reserved for system signals, and I/O pins connecting to other FPGAs and memories. If communicating tasks are mapped onto different FPGAs, the available number of wires connecting the FPGAs may be insufficient to implement all the required communication channels. In this case, some form of multiplexing logical channels over physical wires must be used. Communicating tasks mapped onto FPGAs that are not directly connected require the insertion of routing nodes. Second, a problem unique to RTR systems is interconfiguration communication [7]. If two communicating tasks are executed sequentially, a means of transferring the data from the sender task to the receiver task must be provided. Given a partially reconfigurable FPGA device, such as the Xilinx XC62xx, part of the FPGA area can be reserved to implement a communication buffer. In the case that the resources can only be reconfigured globally, data has to be stored temporarily in an external memory.

In this paper we present CORES/HASIS, a communication synthesis tool set for embedded reconfigurable systems. Using this tool, the designer can focus on the functional behavior of the design and on the evaluation of different mappings. Interface circuitry is automatically inserted to connect communicating tasks, whether they are mapped onto the same or onto different FPGAs. This includes multiplexing, routing, and dedicated interfaces that solve the problem of interconfiguration communication by temporarily buffering data in the system's memories.

2 Synthesis of Reconfigurable Systems

Generally, system synthesis starts from a *specification model* and refines this model iteratively until the *implementation model* is reached. The three tasks that usually comprise system synthesis are *allocation*, *binding*, and *scheduling*.

The specification model consists of three parts, a *problem graph*, an *architecture graph*, and implementation *constraints*. The problem graph represents the functional objects of the application. In our methodology, the problem graph is a directed, acyclic graph with two kinds of nodes: tasks and queues. Tasks are

functional objects with an associated executable code/circuit that is stored in an implementation library. Tasks read data from and write data to ports. The ports are characterized by their bit width and use handshake signals to implement an asynchronous communication protocol [4]. On termination, a task raises its *done* flag. Queues are buffers with FIFO semantics that connect two tasks. Directly connected tasks denote unbuffered communication, queues allow for buffering. A queue has a size (width $\times$ depth) and an access mode (blocking, non-blocking). Edges in the problem graph denote *data flow*. The architecture graph has three kind of nodes: *FPGAs*, *memories*, and *buses*. The edges of the architecture nodes indicate the connections between FPGAs, memories, and buses. An *allocation*

	FPGA	memory	bus
task	$\surd$	—	—
queue	$\surd$	$\surd$	—
routing node	$\surd$	—	—
interface node	$\surd$	—	—
data flow edge	$\surd$	$\surd$	$\surd$

Table 1. Possible bindings

is a subset of the architecture graph, i.e., a set of FPGAs, memories and buses that is sufficient to implement the problem graph. A *binding* maps each object of the problem graph (tasks, queues) onto some resource of the architecture graph. The possible bindings are shown in Table 1. A binding induces also a mapping of data flow edges to paths in the architecture graph. A *schedule* assigns a starting time to each task. In pure CTR systems, there exists only one configuration for each FPGA. All the specified tasks and queues are loaded (configured) onto the FPGA before the application starts. Tasks are active from beginning and can be blocked by a read from a channel, if the required data is not yet available. In RTR systems, each FPGA may undergo a sequence of configurations. A new configuration can be loaded, if all the tasks in the current configuration have completed. The individual done flags from the tasks are combined to form the configuration's done flag. All the configurations' done flags are collected by the *reconfiguration controller*. This function, usually located at the host, executes the static reconfiguration schedule.

Communication synthesis is a synthesis step that is applied after allocation, binding, and scheduling. Our communication synthesis tool suite CORES/HASIS expects as input a refined specification graph, i.e., allocation, binding, and scheduling have already been done by either a system synthesis tool or manually by the designer. The CORES/HASIS tool suite splits the refined problem graph into several smaller problem graphs, one for each resource and configuration. Then *interface nodes* and *routing nodes* are inserted. Interfaces nodes establish FPGA-to-FPGA and FPGA-to-memory communication and multiplex several logical 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 bound to FPGAs, as shown in Table 1.

3 Communication Synthesis

Communication synthesis for reconfigurable systems deals with the connection of tasks and queues bound to arbitrary FPGAs and memories in one or several configurations. Fig. 1 presents a taxonomy of the different communication types for the basic task-queue-task system. In CTR systems, there is only *on/off-chip communication*. On-chip communication is the simplest communication type and occurs between tasks or tasks and queues. Off-chip communication involves two FPGAs or an FPGA and a memory. This communication type can induce routing and multiplexing. In RTR systems, *interconfiguration communication* is required.

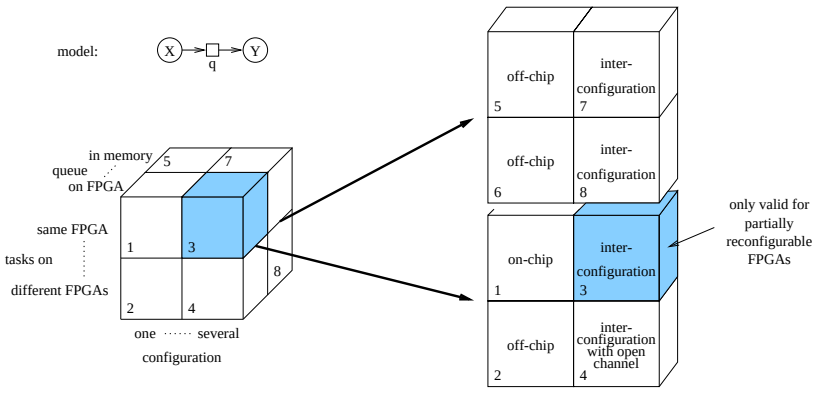

Fig. 1. Taxonomy of communication types for the basic task-queue-task system

3.1 On/off-chip communication

For on- and off-chip communication, we leverage on our object-oriented hardware/software codesign tool for communication synthesis, HASIS [3] [4]. HASIS copes with communication types 1, 2, 5 and 6 in Fig. 1. The key components of HASIS are configurable interface objects that generate dedicated VHDL interface code. HASIS maintains a set of object templates which can be easily extended by templates generated from scratch.

As an example, Fig. 2a) shows the implementation of a task. The core describes the node's functionality as defined in the problem graph. The wrapper around the core consists of FSMs that establish data transmission [5]. A queue implementation has a similar structure and consists of a container that is wrapped by FSMs for communication and access control. By using these interface objects, different protocols for synchronization and data transfer can

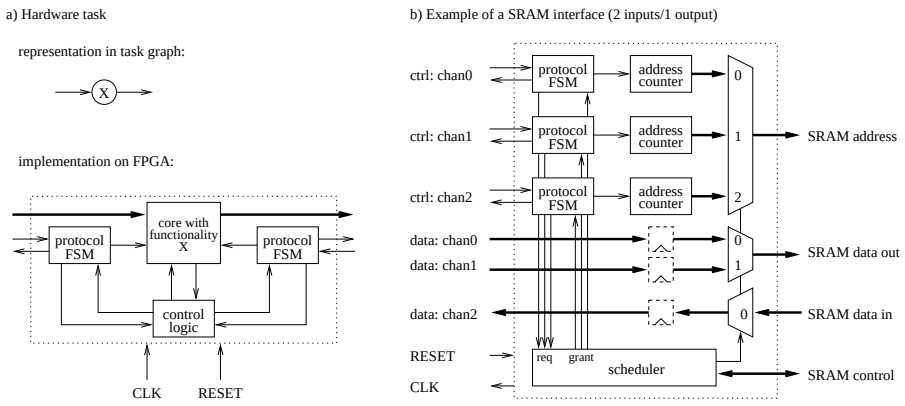

Fig. 2. a) Hardware task b) Example of an SRAM interface

be supported. Operations on queues may be blocking or non-blocking. Off-chip communication will be asynchronous, driven by handshake signals. This allows communication between FPGAs running at different clock speeds. Fig. 2b) shows a configured template of an SRAM interface with two write channels and one read channel. A scheduler manages the read/write requests. The access protocols are handled by FSMs.

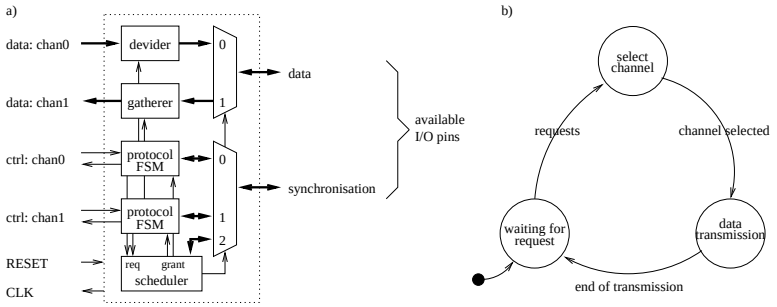

Fig. 3. a) Multiplexer template b) Main states of the scheduler

3.2 Routing and multiplexing

If two communicating tasks are bound to non-adjacent FPGAs, routing tasks with appropriate interfaces are inserted on intermediate FPGAs. A routing task acts like a blocking queue with depth one and is totally transparent to the communicating tasks. In larger systems, there may exist many alternative paths. In the current version of our tool, we expect that the synthesis tool or the designer chooses the path.

If the number of available pins that connect two FPGAs is insufficient to implement the required communication channels, a multiplexer interface will be inserted on both FPGAs. Fig. 3a) shows a multiplexer template configured for one read and one write channel. The available I/O pins are split into data and synchronization pins. Incoming communication requests are scheduled by a fixed priority; the data transfer is initiated by a protocol using the synchronization lines (see Fig. 3b).

3.3 Interconfiguration communication

Interconfiguration communication arises if two communicating tasks are assigned to different configurations, either on the same FPGA or on different FPGAs. If the two tasks are mapped onto the same FPGA and no queue has been specified between them, a partially reconfigurable FPGA is required (case 3 in Fig. 1). In the current version of our tool, this is not supported. Hence, we assume that either the designer or the system synthesis tool has inserted a queue of appropriate depth and mapped this queue to memory (case 7 in Fig. 1).

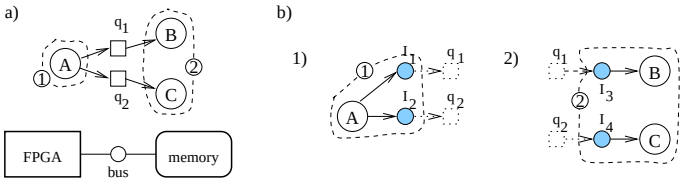

Fig. 4. a) Problem specification b) Splitted problem specification

If the two tasks are assigned to different configurations on different FPGAs, we have to distinguish two cases. In the first case, for all the interconfiguration communication memory-bound queues are inserted (case 8 in Fig. 1). In the second case, the queues - if specified at all - are bound to reconfigured FPGAs. This means, that at least one communication channel will stay open (unconnected) for a while (case 4 in Fig. 1). Using this technique requires FPGA technology that allows the physical implementation of open channels, e.g., tri-state I/O pins with pull-ups.

The example in Fig. 4a) shows a problem specification with 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 are mapped to memory. Each edge of the task graph crosses exactly one configuration border. CORES allocates memory for each of the two queues q_1 and q_2 and refines the specification graph by splitting it up into one graph per configuration and by inserting interface nodes. The interface nodes I_1 and I_2 are inserted into configuration one and implement write channels to the memory. In configuration 2, the interface nodes I_3 and I_4 are inserted, which implement read channels. The write and read

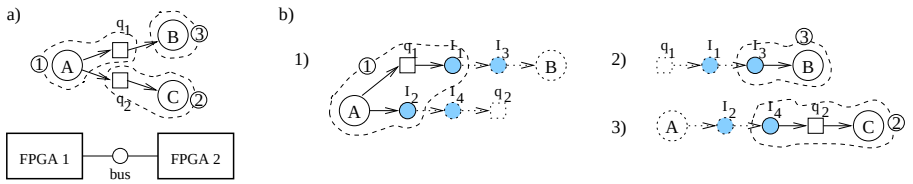

Fig. 5. a) Problem specification b) Splitted problem specification

interfaces are parameterized with the addresses of the queues in the memory and static priorities to resolve simultaneous requests.

The example in Fig. 5a) shows a problem specification with task A assigned to configuration 1 of FPGA 1 and tasks B and C assigned to configurations 2 and 3 of FPGA 2, respectively. The queue q_1 is assigned to configuration 1 of FPGA1, queue q_2 to configuration 2 of FPGA2 (FPGA2's first configuration). 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. Again, CORES allocates memory areas for the queues q_1 and q_2 and splits the problem specification into three graphs containing the necessary interface nodes.

4 Design tool flow

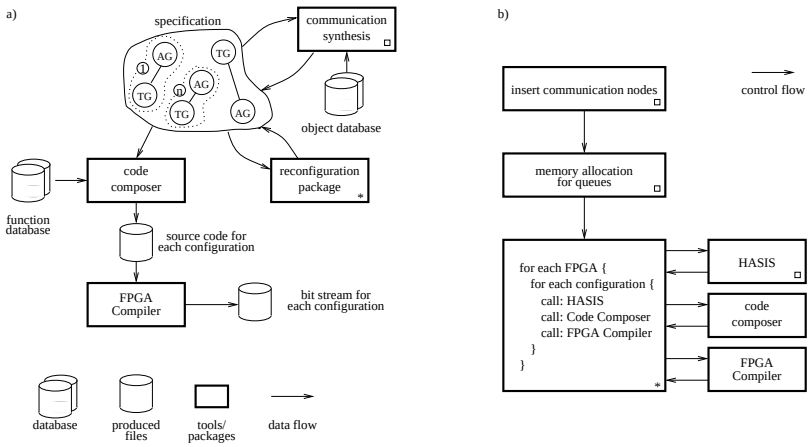

Fig. 6. a) Data flow view b) Control flow view

The input to CORES/HASIS is a problem graph, where the objects have been mapped to different resources and configurations. The design tool flow comprises a *data flow view* and a *control flow view*. The data flow view describes

the data flow between the tools and is shown in Fig. 6a). The control flow view describes the activation sequence of the associated tools and is shown in Fig. 6b). The first two phases of CORES prepare for the code synthesis of the individual configurations. First, communication nodes are inserted including routing nodes and interface nodes. Second, memory is allocated for the queues. The CORES main loop takes each configuration of each FPGA and calls HASIS for communication synthesis, the code composer to assemble the top entity from all the tasks functionalities and interfaces, and finally Synopsys FPGA compiler to generate the bit stream.

5 Case study

As proof of concept, we have implemented a steganography [10] [9] application. A message in form of an ASCII text is hidden within a stream of image data (see Fig. 7a)) by the Least Significant Bit (LSB) insertion method. The resulting data stream is then compressed by run-length coding.

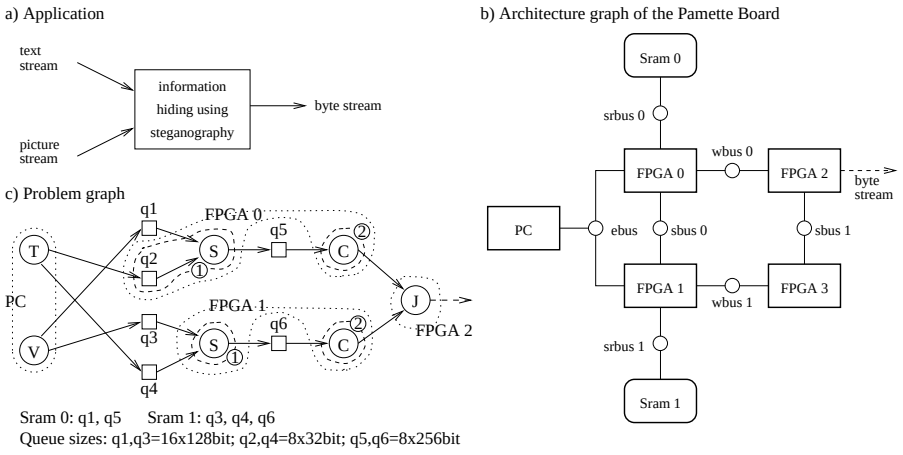

Fig. 7. Steganography application: a) principle b) Architecture graph for the Pamette board c) Problem graph

Figure 7b) shows the architecture graph of the target, the PCI-based multi-FPGA board Pamette from COMPAQ/DEC connected to a PC. The Pamette contains four FPGAs (FPGA0 ... FPGA3) of type Xilinx XC4028, two SRAM banks (SRAM0, SRAM1), and a number of buses. Fig. 7c) presents the problem graph for the steganography example. The tasks S (steganography), C (compress), and the queues q1, q2, and q5 are sufficient to implement the application. To increase the throughput we have doubled this processing line. The input data comes from two tasks running on the PC, V (image stream) and T (text), which

write images and text to the queues q1, q2 and q3, q4 respectively. The output streams of the two processing lines are merged by task J (join).

We present results for two different mappings. In the first case, a CTR example, all tasks – except V and T – and queues are bound to FPGA0; FPGA2 performs routing. In the second case, an RTR example, FPGA0 and FPGA1 have two configurations, FPGA2 contains task J, and FPGA3 performs routing. This can be seen in Fig. 7b). The queue q2 is bound to FPGA1, all other queues are mapped to memory. The queues q1 and q3 are of size 16 $\times$128, queues q2 and q4 of size 8 $\times$ 32, and queues q5 and q6 of size 8 $\times$ 256. Table 2 summarizes the number of required CLBs for each FPGA for the RTR and CTR example. For each FPGA configuration the number of required CLBs is given, divided into interfaces (I), queues (Q), and tasks (T). Different configurations are separated by a comma, the task names are given in parentheses. R8(R16) denotes a 8(16)bit routing task.

		FPGA 0	FPGA 1	FPGA 2	FPGA 3
RTR	I	180, 29	210,29	0	0
	Q	38,0	0, 0	0	0
	T	37 (R16,S),38 (C)	42 (R8,R16,S),38 (C)	35 (J)	5 (R8)
CTR	I	65	-	0	-
	Q	514	-	0	-
	T	167 (S,S,C,C,J)	-	5 (R8)	-

Table 2. Amount of required CLBs for each FPGA configuration

The purpose of this case study was to proof the concepts of communication synthesis. Hence, the examples were chosen to include a variety of communication types; by no means these mappings are optimal with respect to performance. The results in Table 2 show that for CTR a considerable number of CLBs has been allocated for the queues. This is necessary, because all queues of the problem graph are mapped to FPGA0. In the RTR example, the number of interface CLBs is quite remarkable. This stems from the fact, that for queues mapped to memory, the interface implements the signaling protocol to the SRAM as well as the queue management, i.e., write and read access control. For example, FPGA1 in configuration 1 implements an interface to SRAM1 with 5 channels. Two of these channels write data from the tasks V and P (via routing tasks) into SRAM1, two channels read the data and feed it into task S, and the last channel writes the result of tasks S into SRAM1.

6 Future Work

Future work will include the investigation of system synthesis tasks for RTR, especially binding and scheduling. Communication synthesis plays an important role for system synthesis by providing good and reliable estimates for the final implementation. Based on the combined system and communication synthesis, a more accurate design space exploration for RTR systems will be possible.

Acknowledgment

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

References

1. Saad AlKasabi and Salim Hariri. A Dynamically Reconfigurable Switch for High-speed Networks. In *IEEE 14th Annual International Phoenix Conference on Computers and Communications*, pages 508–514, 1995.
2. Robert P. Dick and Niray K. Jha. CORDS: Hardware-Software Co-Synthesis of Reconfigurable Real-Time Distributed Embedded Systems. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 62–68, 1998.
3. M. Eisenring and J. Teich. Domain-specific interface generation from dataflow specifications. In *Proceedings of Sixth International Workshop on Hardware/Software Codesign, CODES 98*, pages 43–47, Seattle, Washington, March 15-18 1998.
4. M. Eisenring and J. Teich. Interfacing hardware and software. In *8th International Workshop on Field-Programmable Logic and Applications, FPL'98, Lecture Notes in Computer Science, 1482*, pages 520 – 524, Tallinn, Estonia, August 31 - September 3 1998.
5. M. Eisenring, J. Teich, and L. Thiele. Rapid prototyping of dataflow programs on hardware/software architectures. In *Proc. of HICSS-31, Proc. of the Hawai'i Int. Conf. on Syst. Sci.*, volume VII, pages 187–196, Kona, Hawaii, January 1998.
6. Bernard Gunther, George Milne, and Lakshmi Narasimhan. Assessing Document Relevance with Run-Time Reconfigurable Machines. In *IEEE Symposium on FPGAs for Custom Computing Machines*, pages 10–17, 1996.
7. Brad L. Hutchings and Michael J. Wirthlin. Implementation Approaches for Reconfigurable Logic Applications. In *International Workshop on Field-Programmable Logic and Applications*, pages 419–428, 1995.
8. T. Ismail J. Daveau and A Jerraya. Synthesis of system-level communiction by an allocation-based approach. In *8th Int. Symp. on System Synthesis*, pages 150–155, September 13-15 1995.
9. Neil F. Johnson and Sushil Jajodia. Staganalysis of images created using current steganography software. In *Workshop on Information Hiding, Lecture Notes in Computer Science, 1482*, volume 1525, pages 273–289, Portland, Oregon, USA, 15 - 17 April 1998.
10. Neil F. Johnson and Sushil Jajodia. Steganography: Seeing the unseen. *IEEE Computer*, pages 26–34, February 1998.
11. M. Kaul and R. Vemuri. Optimal temporal partitioning and synthesis for reconfigurable architectures. In *Proceedings of Design, Automation and Test in Europe*, pages 389–396, 1998.
12. A. Kirschbaum and M. Glesner. Rapid prototyping of communication architectures. In *8th IEEE Int. Workshop on Rapid System Prototyping*, pages 136–141, June 24-26 1997, Chapel Hill, North Carolina.
13. Eric Lemoine and David Merceron. Run Time Reconfiguration of FPGA for Scanning Genomic DataBases. In *IEEE Symposium on FPGAs for Custom Computing Machines*, pages 90–98, 1995.
14. Brian Schoner, Chris Jones, and John Villasenor. Issues in Wireless Video Coding using Run-time-reconfigurable FPGAs. In *IEEE Symposium on FPGAs for Custom Computing Machines*, pages 85–89, 1995.