In Proceedings of the
2nd International Workshop on Engineering of Reconfigurable Hardware/Software Objects (ENREGLE00),
pages 151-157, Las Vegas, USA, June 2000.

An Implementation Framework for Run-time Reconfigurable Systems

Michael Eisenring
Computer Engineering & Networks Lab
Swiss Federal Institute of Technology
Zurich, Switzerland

Marco Platzner
Computer Engineering & Networks Lab
Swiss Federal Institute of Technology
Zurich, Switzerland

Abstract *We present an implementation framework for run-time reconfigurable systems. The framework provides a methodology and a design representation which allow to plug in different design tools. Front-end tools cover design capture, temporal partitioning and scheduling; back-end tools provide reconfiguration control, communication channel generation, estimation, and the final code composition. This paper discusses two of the framework's main issues, the design representation and the hierarchical reconfiguration approach for multi-FPGA systems.*

Keywords: multi-FPGA systems, hierarchical run-time reconfiguration, communication channel synthesis

1 Introduction

During the last years run-time reconfiguration of hardware has gained interest as implementation approach for embedded systems [1] [2]. Run-time reconfigurable systems split a larger problem into temporally exclusive collections, so-called configurations, of smaller subproblems that are loaded onto FPGAs dynamically during the application's run-time. The design and implementation of these systems pose new research problems. The most pressing design problem is likely to be temporal partitioning, a step that groups a number of subfunctions or hardware objects into FPGA configurations [3] [4]. Important implementation issues are the control of FPGA reconfiguration and the generation of communication channels between hardware objects in arbitrary configurations [5].

We are developing an implementation framework for run-time reconfigurable systems. The main goal of this effort is to offer a methodology that simplifies and partly automates the design and implementation of reconfigurable multi-FPGA systems. The framework also provides an abstraction from the underlying hardware which increases portability of run-time reconfigurable applications to different target architectures. The systems we consider are built of communicating coarse-grained hardware objects, typically denoted as FPGA cores and increasingly traded as intellectual property (IP) blocks [6] [7] [8]. Typical applications are large signal processing tasks, e.g., processing of video and audio streams, that are executed on limited FPGA resources.

Figure 1 shows the structure of our framework. The heart of the framework is formed by the *data object repository* that captures design data which is stepwise refined from the original specification down to the final implementation. The framework's design flow provides pluggable tools that perform this refinement. The tools are divided into front-end and back-end tools. Front-end *capture* tools allow to formulate a design problem as a data object comprising a behavioral description of the system's functionality in form of a *problem graph* PG, and a target architecture in form of an *architecture graph* AG. Front-end *Partitioning* and *scheduling* tools update the design representation with i) a mapping M between PG and AG, ii) an assignment of PG nodes to FPGA configurations, and iii) determine execution orders based on *estimations* of IP cores and target components. A first back-end tool is *reconfiguration synthesis* which refines the specification by automatically introducing an appropriate hierarchical

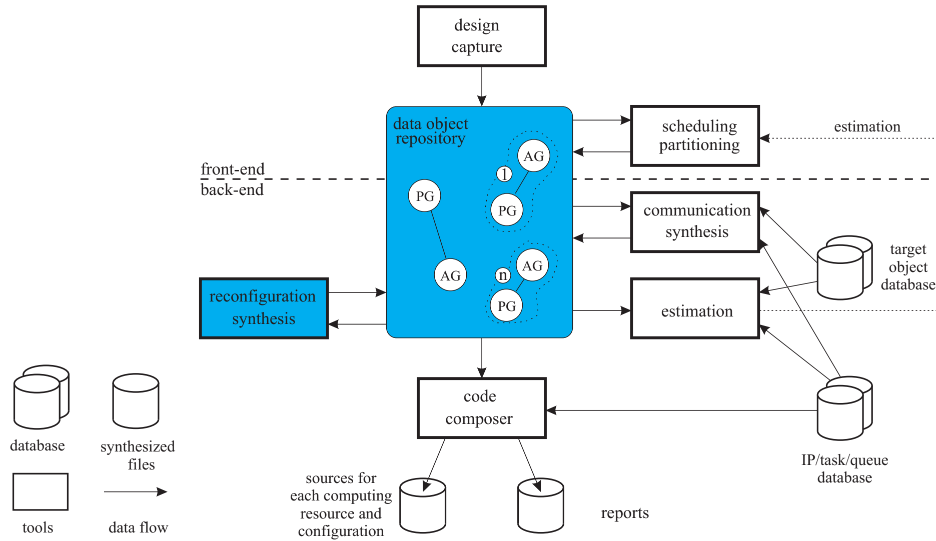

Figure 1: Implementation framework for run-time reconfigurable systems.

reconfiguration architecture that establishes FPGA reconfiguration while satisfying the given schedule. The result of this tool is a number of configurations per FPGA that contain all the necessary tasks and queues together with tasks responsible for reconfiguration control. The *target object database* provides object-oriented target models [9] of the supported architecture components and currently contains models for XILINX FPGAs, a host PC, a DSP TMS320C6701 (TI), memories, and interfaces. The *IP/task/queue database* stores PG node implementations (cores) released for synthesis. Cores for FPGAs are described as synthesizable VHDL/Verilog descriptions or netlists; cores for sequential computing resources (i.e., general/special purpose processors) are described in C. Each core has an associated property file describing its features, e.g., port descriptions, area estimations (CLBs for FPGA cores, program/data memory for general/special purpose processors), required files for synthesis (e.g., pin-out descriptions), etc. *Communication synthesis* tools establish communication links between nodes in arbitrary configurations and automatically generate the required interfaces [5] by using information about the connected cores and target objects. *Estimation* tools allow to assess a data object, e.g., the estimated overhead in terms of FPGA area (CLBs) required for establishing the hierarchical reconfiguration architecture. Finally, *code composer* tools assemble the cores, the generated reconfiguration architecture, and the interfaces for each configuration and FPGA.

Currently, the framework supports only complete FPGA reconfiguration and provides a command line shell interpreting CCSL [10], a language developed for communication channel synthesis. CCSL enables also access to tools and databases. A graphical user interface is in preparation.

In this paper we focus on two important issues of our framework (shaded areas in Figure 1). First, we describe the design representation comprising aspects of the problem, the target architecture, and the communication channels. Second, we discuss our hierarchical approach to reconfiguration control for multi-FPGA systems. The other parts of the back-end have been described in [5].

2 Design Representation

A problem graph $PG = (V_P, E_P)$ is a directed, not necessarily connected, graph consisting of a set of nodes V_P with input and output ports. Each port has assigned a communication protocol type. Directed edges $E_P \subseteq (V_P \times V_P)$ represent communication and connect output ports to input ports of the

same protocol type. Additionally, each node has a control port for receiving control commands (e.g., *start*) and replying status messages (e.g., *done*). Every node implementation contains a *node control* state machine that may set the node's execution status and read the node's internal status, depending on the commands received at the control port. A simple set of nodes which is sufficient for implementing dataflow models comprises *tasks* and *queues*. A task node (see Figure 2a) starts on receiving a *start* command on its control port, executes its function which computes a set of outputs from a set of inputs, and emits a *done* message at the control port on completion. A queue node (see Figure 2b) is a buffer with one input and one output port (besides the control port), a size attribute, and FIFO semantics. The queue shown in Figure 2b has also a control port, which allows to start and stop queue operation. All node implementations are stored in the IP/task/queue database (see Figure 1).

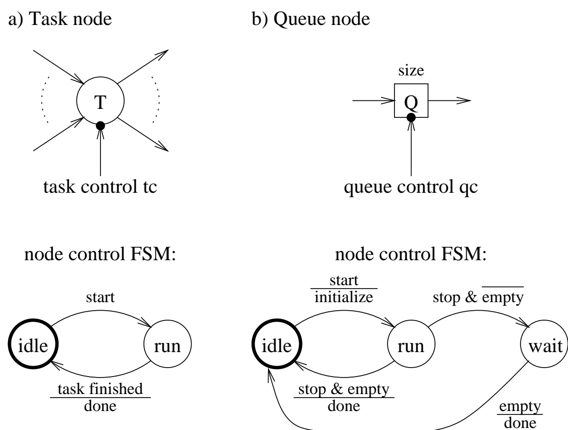

Figure 2: Node types: task and queues.

An architecture graph $AG = (C_A, B_A, E_A)$ is a bipartite graph consisting of a set of components C_A, a set of buses B_A, and a set of undirected connector edges $E_A \subseteq (C_A \times B_A) \cup (B_A \times C_A)$. Components comprise computing resources and memories. There exist two types of computing resources: processors and FPGAs. Processors execute task nodes programmed in C. FPGAs implement task and buffer nodes programmed in VHDL or given as a netlist. The framework relies on an object-oriented chip model [9] for the components C_A of the architecture graph modeling important properties, such as the input/output modules of a chip, as inner component objects. Figure 3 shows part of the model for XILINX FPGAs. Concrete classes (rectangular boxes) denote interface generators and chip models [9]. Abstract classes (rounded boxes) denote general properties. An example for a general property is that every communication module has assigned to a speed estimation value. The *chip model* (bold box) is an FPGA computing resource and may contain an arbitrary number of dedicated *Direct* (on-chip channel) or *MuxFF* (multiplexed off-chip channel) interfaces (only limited by the FPGAs' area). The actual interface implementations depend on the assignment of problem graph nodes to FPGA configurations and the hierarchical reconfiguration architecture. They are generated by the inner component objects (shaded boxes). Chip models are stored in the target object database.

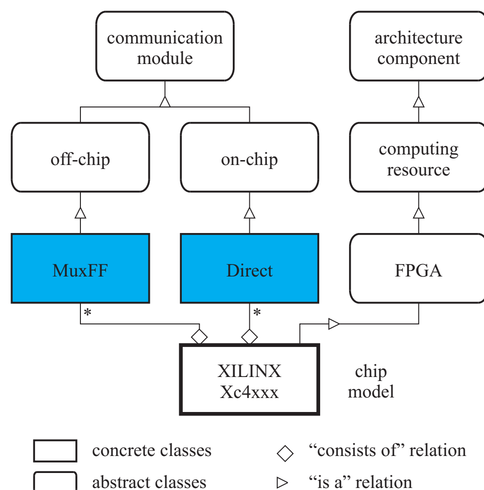

Figure 3: Part of the object-oriented model of a XILINX FPGA.

Example: 1 (SDF) *Figure 4a) shows a scheduled SDF model. The design representation of this application is graphically given in Figure 4b). The problem graph (PG) consists of two tasks and two queues and a node* d *(dispatcher) that implements the schedule by sending/receiving messages to/from the nodes' control ports. For simplicity,*

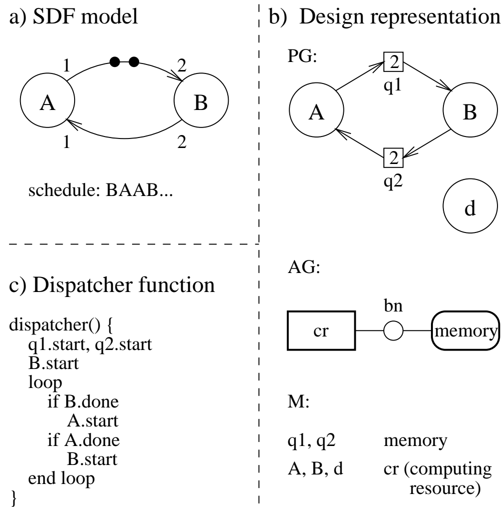

Figure 4: Representation of an SDF model.

the control port connections are not shown. The architecture graph (AG) specifies a computing resource cr *connected to a* memory *via the bus bn. The mapping (M) given for this simple example has only one configuration for the computing resource* cr*. Pseudo-code for the dispatcher is shown in Figure 4c).*

Example: 2 (Dataflow) *Figure 5 shows another example of a dataflow-based formalism, a model that is often denoted as task graph. The problem graph has been partitioned onto one host processor* H*, two FPGAs* F1 *and* F2*, and memories* mem1, mem2*, and* mem3*. Each of the two FPGAs runs three configurations (dashed areas).* F1*, for example, repetitively executes first configuration 1 (queue* q1*, task* B*), then configuration 2 (tasks* C *and* D*), and finally configuration 3 (task* E*).*

3 Reconfiguration Control

Our reconfiguration architecture for multi-FPGAs is hierarchical and consists of two layers. On the top layer, one or several *configurator* nodes *cfg* supervise a set of FPGAs by downloading and starting complete configurations. On the bottom layer, a dispatcher *d* is assigned to each configuration and starts and stops individual nodes of the configuration. The dispatcher assembles the nodes' done events, determines the end-of-configuration and transmits a configuration *done* message together with an optional return value to the supervising configurator. A configurator task may be implemented on any computing resource of the architecture graph that is able to download configurations to each of the supervised FPGAs.

Example: 3 (Reconfiguration) *Figure 6a) shows the top reconfiguration layer for example 2 (see Figure 5). The configurator task* cfg *communicates with all the configurations of the two FPGAs. Note that only one configuration per FPGA is active at a time. Figure 6b) shows the bottom reconfiguration layer for the first configuration of FPGA* F1*. Three additional nodes (*R_{IJ}*,* R_{2c}*, and* d_{11}*) have been automatically inserted into this configuration. The dispatcher* d_{11} *controls the other nodes and communicates with the configurator* cfg*. The architecture graph shows that any communication between FPGA* F2 *and the host has to be routed via FPGA* F1*. Consequently, the node* R_{IJ} *routes task* I *on FPGA 2, configuration 3 to task* J *on the host and the node* R_{2C} *routes the dispatchers of FPGA* F2 *to the configurator on the host. In certain cases optimizations can be applied. One such optimization is the use of only one routing node* R *to connect the configurator to all dispatcher nodes of FPGA* F2*. Figure 6c) shows a possible configurator function where each FPGA cycles through 3 configurations.*

The two reconfigurator layers together determine the overall schedule of nodes. This overall schedule is either fully determined by the front-end tools at system design time (as shown in Example 1) or dynamically by the dispatchers and configurators at run-time (as will be shown in Example 1). In any case, we can identify three sources of constraints for the scheduler. First, the schedule has to reflect the execution semantics of the used specification formalism, e.g., precedence relations in task graphs. Then, the schedule must resolve nondeterminism in the specification either at compile time or at run-time. Finally, the schedule must respect configuration borders given by the partitioning. It has been shown [11] that careless partitioning may even lead to infeasible schedules.

Formally, a configurator executes one or several *configurator finite state machines* $cFSM$, each de-

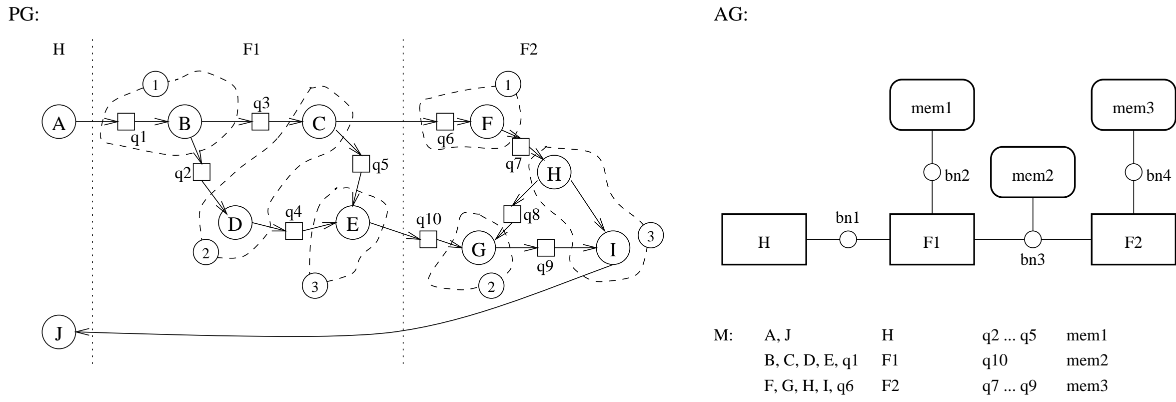

Figure 5: Representation of a task graph model consisting of a problem graph (PG), an architecture graph (AG) and a mapping (M).

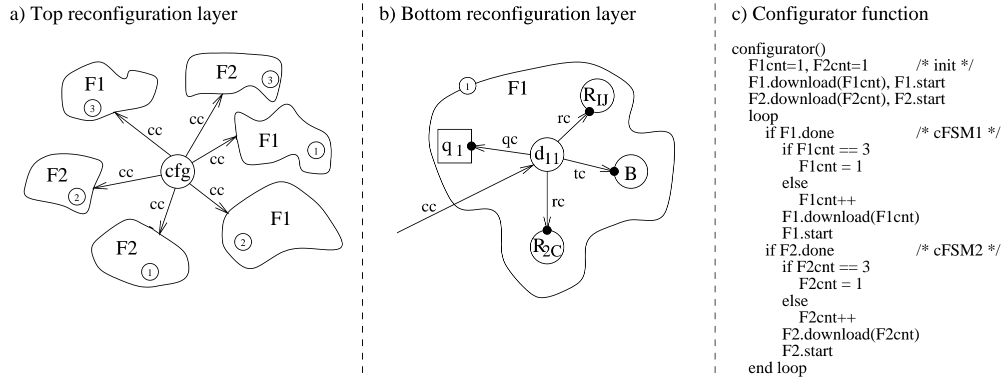

Figure 6: Reconfiguration architecture for Example 2.

scribing a single FPGA's run-time configuration sequence. A $cFSM = (S, E, v_0)$ consists of a set of states S which represent device configurations, a set of edges $E \subseteq S \times S$, which represent device reconfigurations (configuration switches), and an initial configuration v_0. Each edge $e \in E$ has a condition and an action assigned to it. The condition is composed of `done(X)` and `rval(X)` predicates. `done(X)` is set true when configuration `X` has completed; `rval(X)` is configuration X's optional return value. We require that i) edges that are outgoing from a state `X` contain `done(X)` in their conditions, and ii) only one out of possibly several outgoing edges for a specific state evaluate to true.

Example: 4 (cFSM1) *Figure 7 shows a cFSM of an FPGA implementing four configurations. The initial state is A. Assume that the cFSM is in state B. The next state depends on the return value of B. If the return value is 1, the cFSM restarts the current configuration. If the return value is 2 or 3 the next configuration of the FPGA is C or D, respectively.*

Generally, configuration switches may depend on other FPGA's run-time conditions. This allows to implement dynamic configuration scheduling between several devices. To enable such an advanced scheduling technique, the cFSMs of the FPGAs must be coupled in a meaningful way. A simple approach is that `rval(X)` predicates may refer to return values of other FPGA's configurations. An `rval(X)` predicate will only return with

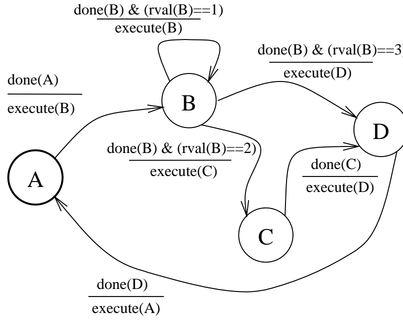

Figure 7: Example of a cFSM.

a value if configuration X has been completed once. `rval(X)` for a running configuration will either return the value of X's last execution or block the evaluation of the condition and hence device reconfiguration.

Example: 5 (cFSM2) *Figure 8 shows two coupled cFSMs. FPGA1 alternates execution of two configurations,* $A1$ *and* $B1$*. FPGA2's cFSM executes first* $A2$ *and then either* $B2$ *or* $C2$*, depending on the return value of* $A1$*. The initial states are* $A1$ *and* $A2$*.*

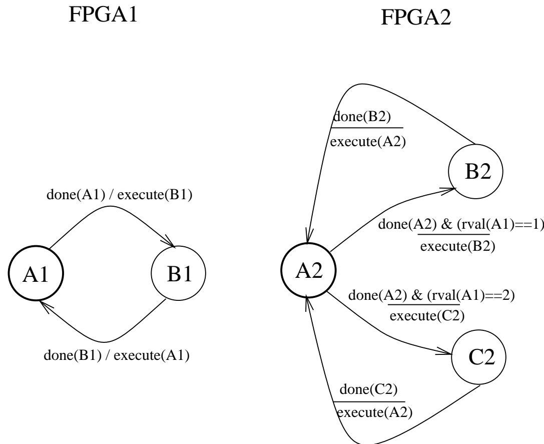

Figure 8: Example of two coupled cFSMs.

The cFSM of each FPGA is specified using dedicated CCSL commands. The dispatchers of the individual configurations are generated automatically.

4 Run-time Overheads

The back-end tools reconfiguration control and communication channel synthesis introduce overheads in terms of execution time and hardware area. An estimation of this overhead is important to the front-end tools that rely on this data for efficient partitioning and scheduling.

Each node of a cFSM may be annotated with the run-time of its configuration. Note that this parameter is only known when the schedule executed by the dispatcher is static and the single task run-times are given (as is usually the case for, e.g., SDF problems). The edges e_{XY} are annotated with estimates for the time required to switch from configuration X to Y, tr_{XY}. This time is the sum of t_d, the time required by the dispatcher to wait for the end of configuration, t_t, the transmission delay between the dispatcher and the corresponding configurator, t_s, the scheduling time of the cFSM, and t_{conf}, the time for downloading and starting the new configuration.

$$tr_{XY} = t_d + t_t + t_s + t_{conf} \tag{1}$$

For specification models that do neither require cooperating cFSMs nor complex dispatcher schedules, t_s and t_d will be small. Assuming moderately sized multi-FPGA systems, t_t will be small as well, as communication channels between dispatchers and configurators are not routed over many FPGAs. For such a scenario, the configuration switch time will be dominated by the technology-dependent device reconfiguration time.

The area overhead consists of the generated interfaces for the tasks and queues (depending on the chip models) [9] and circuitry for the hierarchical reconfiguration. Preliminary measurements on XILINX XCV-1000 show that the introduced area overhead due to dynamic reconfiguration is quite small. For example, in the first configuration of FPGA $F1$ (see Figure 2b), all additional nodes (R_{IJ}, R_{2C}, and d_{11}) consumed 10 Virtex slices.

5 Conclusion

In this paper, we have discussed an implementation framework for run-time reconfigurable sys-

tems. We have presented the internal system representation and the hierarchical reconfiguration architecture. This reconfiguration architecture consist of two layers, a dispatcher for each configuration and a configurator for each FPGA. Dispatchers and configurators together can implement a range of static as well as dynamic scheduling techniques.

Although this approach allows for sophisticated scheduling, we are currently investigating specific applications and formalisms (e.g., SDF problems) to motivate the need for such advanced scheduling.

References

[1] 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.

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

[4] 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.

[5] 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.

[6] 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.

[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] 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.

[9] 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.

[10] EISENRING M. CCSL, Communication Channel Synthesis Language. TIK Report No. 80, Computer Engineering and Networks Laboratory, ETH Zurich, Switzerland, 1999.

[11] DICK, ROBERT P. and JHA, NIRAY K. 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.