In Proceedings of the Third International Workshop on *Systems, Architectures, Modeling, and Simulation* (SAMOS'03)
Samos, Greece, July 21–23, 2003, ISBN 90-807957-1-2.

Basic OS Support for Distributed Reconfigurable Hardware

Christian Haubelt, Dirk Koch, Jürgen Teich
Department of Computer Science 12
University of Erlangen-Nuremberg
Email: {haubelt, dirk.koch, teich}@cs.fau.de

***Abstract*— While recent research is mainly focused on the OS support for a single reconfigurable node, this paper presents a general approach to manage distributed reconfigurable hardware. The most outstanding properties of these systems are the ability of reconfiguration, hardware task migration, and fault tolerance. This paper presents first ideas of an operating system (OS) for such architectures. Furthermore, a prototype implementation consisting of four fully connected FPGAs will be presented.**

I. Introduction

Distributed reconfigurable hardware platforms [1], [2] are becoming more and more important for applications in the area of automotive, body area networks, ambient intelligence, etc. The most outstanding property of these systems is the ability of hardware reconfiguration. In terms of system synthesis, this means that the binding of tasks to resources is not static, i.e., the binding changes over time. In the context of FPGAs, recent research focuses one OS support [3] by dynamically assigning hardware tasks to an FPGA.

In a network of connected FPGAs, it becomes possible to migrate hardware tasks from one node to another during the system operation. Thus, resource faults can be compensated by *rebinding* tasks to fully functional nodes of the network. The task of rebinding is also called *repartitioning* or *online partitioning*. A network of reconfigurable nodes that implements repartitioning will be termed *ReCoNet* in the following (see Figure 2(a) for an example of a ReCoNet).

This paper describes the basic problems to be solved for the implementation of a ReCoNet. The first two features that are provided by the OS of the nodes in a ReCoNet are 1) *rerouting* and 2) *repartitioning*. They deal with erroneous resources. Thus, we are able to compensate line errors by computing a new routing for broken communications and we can migrate tasks from one node in the network to another during the system operation. These two tasks are also necessary in all distributed systems, not only in reconfigurable hardware systems. However, based on these two problems, we define feature 3) *partial reconfiguration* as the process of merging different hardware tasks together on a single FPGA. Here, the reconfiguration of a single node, also called *ReCoNode*, is done partially for the whole network. One outstanding objective in this third process is to deliver a generous approach that is not necessary bounded to a specific FPGA architecture. In the context of our paper, we consider only small networks in the sense that the whole state of the ReCoNet is known and stored in each ReCoNode.

A lot of related work has been published in the area of routing and computer networks [4]. The novelty of our approach is to combine these approaches with the paradigm of hardware reconfigurability.

To the best of our knowledge, there is no implementation of a ReCoNet as described above. Reconfigurable Architectures like PACT [5] and Chameleon [6] are first approaches for coarse grained, networked processing units without providing support for online reconfiguration and optimization.

This paper is organized as follows: Section II discusses the three basic OS features needed to provide the necessary support for distributed reconfigurable hardware platforms. Section III focuses on the issue how to configure a single node in such a network while section IV presents our first simple prototype implementation of a ReCoNet. This prototype consists of four connected Altera FPGA boards.

II. The Basic OS Features

This section describes the basic features needed for running a distributed reconfigurable system. Before defining these features, we will take a closer look on the underlying architecture. In this paper, we consider "small" networks of hardware reconfigurable embedded systems. The main aspects of the hardware are:

- small: Each node in the network can store the current state of the whole network. The state of a network is given by its actually typology consisting of all available nodes, of available links, and of the distribution of the tasks in the network. In order to store the available connections in the ReCoNet, we use a so-called *incidence matrix* (see Figure 1). This matrix is of dimension $|V| \times |V|$, where $|V|$ is the number of nodes in the network. An element of the incidence matrix is non-zero if there is a direct connection between the two corresponding nodes.
- hardware reconfiguration: Allows the implementation of arbitrary functions in hardware. Thus, it accelerates the computation of the corresponding functions required in the network.
- embedded: requires the optimization of different objectives, like power consumption, cost, etc. simultaneously.

These are the fundamental properties of a distributed reconfigurable system that we call a *ReCoNet*. Furthermore, a ReCoNet

must support repartitioning of tasks in the network. Therefore, we have to implement three OS features. In order to compensate errors in the hardware infrastructure, we implemented the two OS features *rerouting* and *repartitioning*. Network connectivity faults are compensated by the computation of a new routing. And further, the fault of a complete node is compensated by migrating tasks to other nodes. Finally we implemented a third OS feature named *partial reconfiguration*.

In order to describe these OS Features mathematically, we need an appropriate model. The behavior of the system is given by n tasks $T = \{t_1, t_2, \ldots, t_n\}$ running on m possible nodes $V = \{v_1, v_2, \ldots, v_m\}$. Furthermore, the ReCoNet structure is given by l links $C\{c_1, c_2, \ldots, c_l\}$ between the nodes with:

$$C \subseteq V \times V.$$

Each task $t_i \in T$ can be mapped onto an arbitrary set of resources. Therefore, we model all possible bindings as a set β, where:

$$\beta \subseteq T \times V.$$

The OS features are triggered if a resource fault is detected or a new task becomes ready to run. In the following we reveal the basic OS features rerouting, repartitioning and partial reconfiguration in detail.

A. Rerouting

The first OS feature to be defined is the task of rerouting. Rerouting is required if a connection ($c_f \in C$) in the network fails. All communications done over this connection has to be rerouted. There are several publications dealing with this issue. Recent work was mainly focused on probabilistic approaches [7].

Here, we consider a high-level fault tolerant approach. Rerouting itself can be decomposed in three subproblems:

1) Line detection: Is a link $c_i = (v_j, v_k)$ between two nodes v_j and v_k available or not?
2) Network state distribution: If a connection between two nodes (v_j, v_k) fails, all nodes $v \in V \backslash \{v_j, v_k\}$ in the network not incident to connection c_f must be informed.
3) Routing of broken communications.

The first subproblem can be solved in several ways. The easiest implementation is to periodically send some predefined data over each connection. If we do not detect any signal changes at the end of this connection, either the line or the sending node may be defect. In both cases, we cannot use this connection any longer.

The second subproblem, the distribution of the network state, could also be solved in many ways. Again, we just mention the simplest one, the broadcast. The detecting node v_j just sends this new information to all its neighbors. These neighbors will relay this message until all nodes in the network are notified. The main problem in this solution is to keep track of already known messages which should not be relayed. Another problem is the time needed for the distribution. If an error is intermittent, there may be several different messages

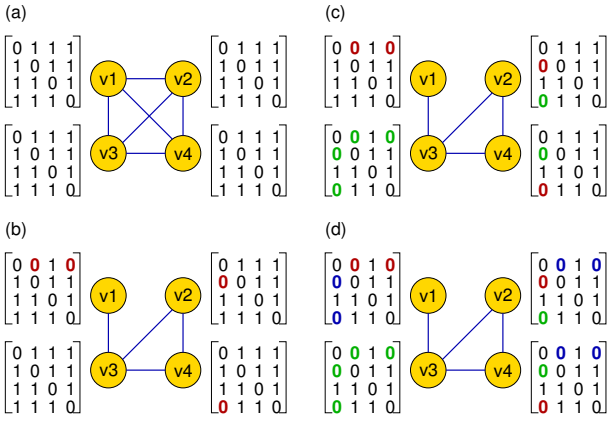

Fig. 1. Example of a ReCoNet. (a) The ReCoNet consisting of four ReCoNodes and the corresponding incidence matrices. (b) Two connections failed. (c) All directed neighbors are notified. (d) All indirect neighbors are notified.

are on their ways through the network all representing a different state of the network. Thus, also if we limit ourselves to applications without any time constraints, we need some time base in the network [8].

The last subproblem, the computation of a new routing can be done with any routing algorithm. Since we only consider small networks, we are able to store the state of all connections of the network in an incidence matrix. Based on this matrix, we can calculate the new routes using for example the shortest path algorithm (see [4]).

Figure 1 shows an example of the first two steps in the process of rerouting. Figure 1(a) shows a ReCoNet consisting of four fully connected ReCoNodes. The corresponding incidence matrices are also shown near each node. In Figure 1(b) the situation is shown where two connections fail. Each incident node detects this error and updates its incidence matrix.

In the next step (Figure 1(c)) all direct neighbors are informed about the topology change of the network. Note, that all nodes store different information of the state of the network. Finally, in Figure 1(d), all nodes have the same information about the state of the network. Based on this information, each node computes the new routing.
On every node this procedure of rerouting can be implemented in the following fashion:

```
if (line_state[i] != next_line_state[i])
  {  Update_Incidence_Matrix();
     Notify_all_Neighbours(line_fail);
     reroute()
  };
if(Receive_net_state! = net_state)
  {  Update_Incidence_Matrix();
     Notify_other_Neighbours(line_fail);
     reroute()
  }
```

B. Repartitioning

Rebinding describes the migration of hardware tasks $t_i \in T$ from one node v_j in the network to another v_k. Thus we

need $\{(t_i, v_j), (t_i, v_k)\} \in \beta$. Note, that if we configure the reconfigurable nodes v_i, v_k with a processor, it may be possible to migrate a hardware task (t_i^{hw}) to a software task (t_i^{sw}) and vice versa, too. For a node v_k this will further need $\{(t_i^{hw}, v_k), (t_i^{sw}, v_k)\} \in \beta$. The task of rebinding is also called *repartitioning* or *online partitioning* in the following.

But when should we migrate a task? In this paper, we just focus on the case of resource faults, i.e., if a node v_f in the network fails, all tasks running on v_f must be migrated to other nodes. Thus, the task of repartitioning can be divided into two subproblems:

1) Detection of resource errors and
2) rebinding of tasks to nodes.

The first subproblem again can be solved by using the incidence matrix of the actual network. If a node v_f has no working connection to any of its neighbors ($\{v_f, v_i\} \in C, \forall i$), it is called *isolated*. An isolated node cannot be used for process execution any more and all tasks bound to this node must be migrated.

An important question is how to perform a save task migration, i.e., how to keep track on the current state of a task. Here, we limit ourselves to stateless tasks which can be started in an appropriate state after a system failure. It is explained in [8] how to handle non-stateless tasks.

Furthermore, we must consider the question of on which node v_j should a task t_i be restarted. Again, there are several well-known approaches and the simplest one is to use a priority binding list $P : \beta \rightarrow \mathbb{N}$ for each task $t \in T$, i.e., each task is bounded to the node with highest priority in its priority list. If this node fails, it rebounds to the next available node with the highest remaining priority. Note that when rebinding tasks, we must recompute the routing as well.

C. Reconfiguration

In order to be as general as possible, we do not require partial reconfiguration as provided by some Xilinx FPGAs [9]. By disconnecting a single node v_i, the remaining network $V \backslash \{v_i\}$ should not be affected. Thus, we can reconfigure a disconnected node completely.

If we allow only fully hardware reconfiguration, we must ensure that all tasks ($t_j \in T \quad \forall (t_j, v_i) \in \beta$), that are implemented on the given node v_i are not needed for the time of the reconfiguration. If we are not sure about this, we first have to migrate these tasks to other nodes in the network. A simple implementation of this algorithm will be presented in Section IV.

Figure 2 shows a scenario where an additional hardware task is loaded into the system. The initial state of the network is shown in Figure 2(a). Each of the four nodes executes software (circles) and hardware (rectangles) tasks.Also, the configuration memory of each node is shown. Next, an additional hardware task (t_9) should be loaded into the system. Here, we assume that t_9 should be loaded on node ReCoNode4. Therefore, all tasks (t_7, t_8) running on ReCoNode4 are suspended and restarted on ReCoNode2 and ReCoNode3, respectively (see Figure 2(b)).

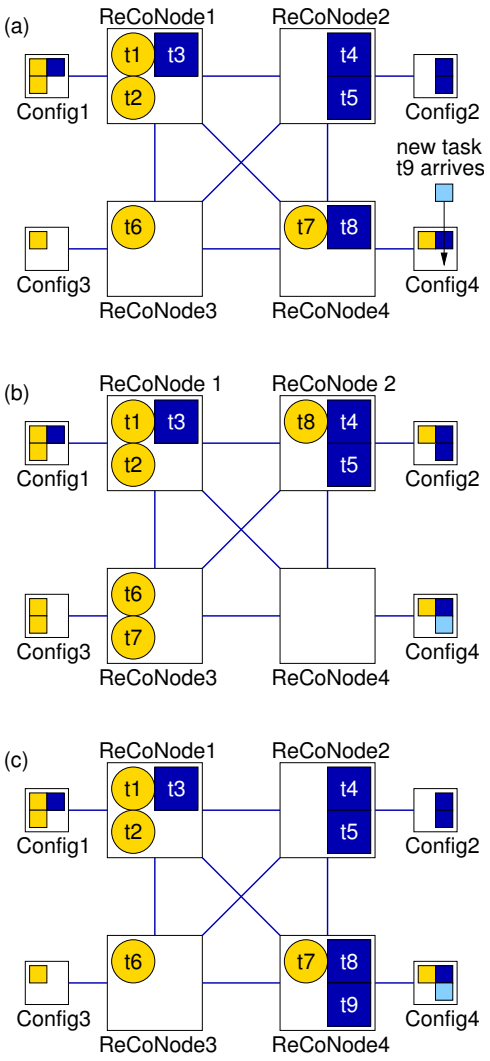

Fig. 2. Example of Repartitioning. (a) shows the original ReCoNet configuration. There are four ReCoNodes. Each node is configured with software (circles) and hardware (rectangles) tasks. The configuration memory for each node is also displayed. (b) To configure ReCoNode4 with an additional hardware task (t_9), we must move all running tasks of this node to other nodes in the network. At the same time the configuration memory (Config4) is updated with the additional task. (c) After reseting ReCoNode4 the new configuration is loaded.

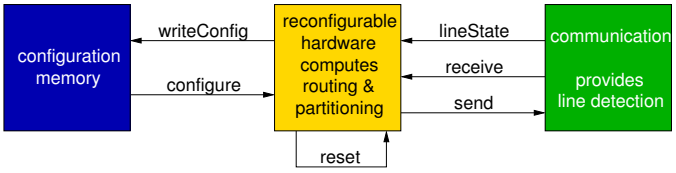

Fig. 3. Example of a ReCoNode. The reconfigurable hardware computes the routing and partitioning of the distributed application. The communication module provides information about local connections.

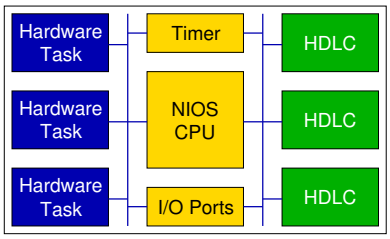

Fig. 4. Configuration of a single ReCoNode. Beside the communication modules (HDLC) and the hardware tasks, there is a NIOS CPU with additional timers and I/O ports.

Note, that task t_8 is implemented in software. Furthermore, the new configuration for ReCoNode4 is stored into the configuration memory.

In a last step (Figure 2(c)), the reset signal is issued on ReCoNode4. ReCoNode4 starts with the new configuration. The task t_7 and t_8 on ReCoNode2 and ReCoNode3 are suspended.

III. Configuration of a ReCoNode

A ReCoNode should implement the three basic functions as described in Section II. Figure 3 shows a simple example of a ReCoNode.

The reconfigurable hardware computes the actual routing and partitioning based on its information about the state of the network. Note, that the hardware could be configured with a processor (as will be presented in the next section) to do the required computation. The communication module is needed for sending and receiving packages over the network. Furthermore, it observes the state of each directly connected line. If the state of a line changes, the reconfigurable hardware will be notified. As described in Section II, the reconfigurable node will then compute the new incidence matrix and will notify all other ReCoNodes in the network about the change.

To perform the reconfiguration of the hardware, the ReCoNode itself can write the configuration memory and generate a self reset on the FPGA. That way, our solution does not require any partial reconfiguration of the FPGAs. In the simplest scenario, we just migrate all tasks to other nodes of the network, write the desired configuration into the configuration memory, and trigger a reset on the FPGA (see also Figure 2). For non-stateless tasks, additional work is needed to ensure a proper migration of such tasks inside the ReCoNets.

Figure 4 shows the configuration of a single ReCoNode as used in the sample implementation described in the next section. Beside the communication modules (HDLC), there is also a CPU and user-defined hardware. By using a CPU core, we have the possibility to run tasks in hardware as well as in software or any mixed implementation.

IV. A Prototype Implementation

In order to test our new approach, we have constructed a distributed reconfigurable prototype consisting of four Altera Excalibur development boards [10]. With a single Excalibur board, the user has the ability to build a hardware-software-system by using Altera's SoPC-builder [11]. Such a systems is composed of Altera's NIOS processor [12], user-specified hardware, and user-specified software. The user-specified software is given in C/C++. The user-specified hardware and the NIOS-processor are connected by using a hardware description language like VHDL or Verilog. After synthesis of the hardware design, it is downloaded to the FPGA on the Excalibur board.

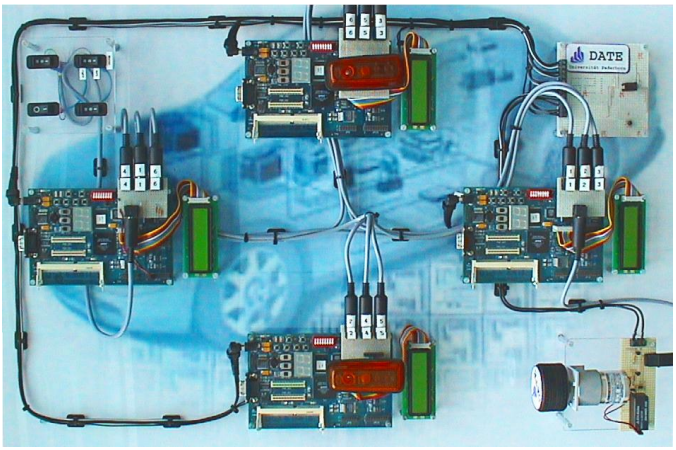

Fig. 5. A prototype implementation of a distributed reconfigurable system that supports repartitioning. The prototype is composed of four Altera Excalibur development boards.

A single node of the prototype implementation is configured as described in section III with a NIOS processor and three communication modules (see also Figure 4). For the communication we have chosen the standard HDLC (High-level Data Link Control) protocol. In order to detect line errors, we have implemented the physical layer with a Manchester encoding. See Figure 4 for the configuration of a single ReCoNode.

The prototype implementation of the ReCoNet is shown in Figure 5. Several automotive applications are implemented on this ReCoNet. The three basic OS features as described in section II are implemented as follows:

- Rerouting: Each node stores the current state of the network in an incidence matrix. If a line error is detected the incidence matrices are updated as shown in Figure 1. In a last step, a new routing for all applications is computed by the use of Dijkstra's shortest-path-algorithm. The routing itself is done by the NIOS CPU on each node.
- Repartitioning: Each ReCoNode stores the binding priority list of each task. If a ReCoNode fails, all tasks currently executed on this node are migrated to the next available ReCoNode with the highest remaining priority. If there is no copy of the migrating task available on this ReCoNode, we must perform a reconfiguration step by either copying a software implementation or a complete configuration file to this node. Therefore, the ReCoNode has to know where to find this configurations. In our prototype, only the two nodes ReCoNode2 and ReCoNode3 switch between two different configurations.

These configurations are stored in the configurations memory of ReCoNode1 and ReCoNode4. Again, the selection of a configurations file is statically programmed. The transfer of the configuration streams is done over the network.

- Reconfiguration: Reconfiguration is established by writing a configuration stream to the configuration memory. The copying of the configuration file is again done by the NIOS CPU. It requests the configuration from one of its neighbors and stores the data in the configuration memory. The error free transfer of the reconfiguration content is guaranteed by the utilized HDLC protocol. Further, a special bit inside the configuration bitstream is used to indicate valid configurations in the memory. This bit is analyzed by an additional PLD that controls the reconfiguration process after the triggered reset. By writing this bit as the last one to the configuration memory, we can ensure that only valid configurations are transferred to the FPGA. This mechanism is archived by the *configurator* PLD that loads a default configuration on determining an invalid configuration memory content. When all data is written correctly to the configuration memory, the NIOS CPU issues the reset signal to itself. The ReCoNode that sends the reconfiguration bitstream remembers the last read request until a proper reconfiguration has been achieved. This allows the default configuration to initialize a broken reconfiguration process again (e.g. during a power failure in the reconfiguration download phase).

In order to visualize the state of the network as well as the distribution of the applications, we have written a small monitoring JAVA program. This program communicates to a single Excalibur board over the debug port. A screenshot of the monitoring program is shown in Figure 6.

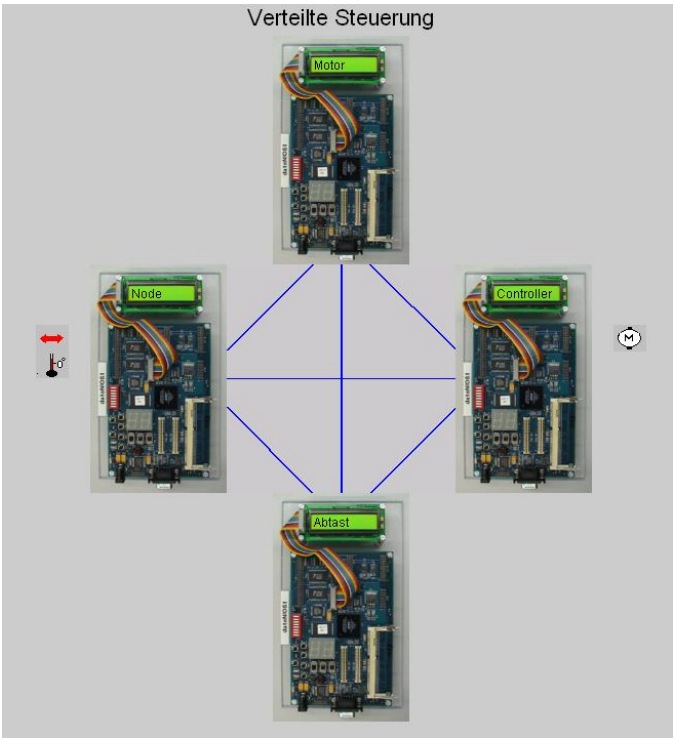

Fig. 6. Monitoring tool for the prototype implementation. One can see the active connections as well as the distribution of the processes.

V. Conclusions and Future Work

In this paper, we have presented the implementation of basic OS features for running a distributed reconfigurable hardware system, a so-called *ReCoNet*. Two of these tasks deal with the fault tolerance of such a system. While the *rerouting* is used to compensate communication errors, we can compensate the defect of a node by *repartitioning*. These two features together with the ability of *hardware reconfiguration* allows us to build dynamical hardware-software systems. A first implementation of these basic tasks in distributed reconfigurable system consisting of four Altera FPGAs was presented here.

Important issues in the future are to provide support for real-time and non-stateless applications. Furthermore the optimization (online/offline) of distributed reconfigurable systems must be considered in future work.

References

[1] R. Dick and N. Jha, "CORDS: Hardware-Software Co-Synthesis of Reconfigurable Real-Time Distributed Embedded Systems," in *Proceedigns of ICCAD'98*, 1998, pp. 62–68.

[2] I. Ouaiss, S. Govindarajan, V. Srinivasan, M. Kaul, and R. Vemuri, "An Integrated Partitioning and Synthesis System for Dynamically Reconfigurable Multi-FPGA Architectures," in *IPPS/SPDP Workshops*, 1998, pp. 31–36.

[3] H. Walder and M. Platzner, "Online Scheduling for Block-partitioned Reconfigurable Devices," in *Proceedings of Design, Automation and Test in Europe (DATE03)*, Mar. 2003, pp. 290–295.

[4] A. Tanenbaum, *Computer Networks*. Prentice Hall PTR, 2002.

[5] V. Baumgarte, F. May, A. Nückel, M. Vorbach, and M. Weinhardt, "PACT XPP - A Self-Reconfigurable Data Processing Architecture," in *ERSA*, Las Vegas, Nevada, June 2001.

[6] Chameleon Systems, *CS2000 Reconfigurable Communications Processor, Family Product Brief*, 2000.

[7] T. Dumitraş, S. Kerner, and R. Mărculescu, "Towards On-Chip Fault-Tolerant Communication," in *Proceedings of the Asia and South Pacific Design Automation Conference 2003*, Kitakyushu, Japan, Jan. 2003.

[8] H. Kopetz, *Real-Time Systems - Design Principles for Distributed Embedded Applications*. Norwell, Massachusetts 02061 USA: Kluwer Academic Publishers, 1997.

[9] Xilinx, 2003, `http://www.xilinx.com`.

[10] Altera, "Excalibur development kit data sheet," June 2000, http://www.altera.com.

[11] Altera, "Quartus programmable logic development system & software data sheet," May 1999. [Online]. Available: `http://www.altera.com`

[12] Altera, "Nios soft core embedded processor data sheet," June 2000. [Online]. Available: `http://www.altera.com`