DESIGN OF A PROGRAMMABLE CONTROLLER FOR HYPERCYCLE BASED INTERCONNECTION NETWORKS

R. Sivakumar, V.V. Dimakopoulos and N.J. Dimopoulos

Department of Electrical & Computer Engineering,
University of Victoria, Victoria,B.C,
CANADA - V8W 3P6
E-mail: rsiva | dimako | nikitas @ece.uvic.ca

Abstract: In this work, we consider the problem of routing in hypercycles which are a class of multidimensional graphs that are generalizations of hypercubes. Hypercycles are products of circulant graphs with simple routing, incremental expandability, and range in complexity from simple rings to fully connected graphs. We present a novel framework and design methodology for a controller through a simple hardware design language called CoDeL (Controller Description Language). The functionality of the controller can be described by manipulating primitives and the design can be synthesized from VHDL. Using FPGA as the medium of implementation, the controller can be reprogrammed for a gamut of routing policies as the application demands.

1. INTRODUCTION

Message passing concurrent computers such as the Caltech's Cosmic Cube [1], MAX [2,3] Intel's iPSC [4] are examples of parallel computers that consist of several processing nodes that interact via messages exchanged over communication channels linking these nodes into one functional entity.

The interconnection used in implementing a concurrent computer can be modelled as a graph. Several topologies have been introduced and studied. Recently, a new class of multidimensional graphs called *hypercycles* [7,8] has been introduced. These graphs are products of circulants [9] and are generalizations of several popular interconnection networks such as *rings, toruses, binary n-cubes, k-ary n-cubes and generalized hypercubes*. Some examples of hypercycles are shown in Figure 1. Many properties and algorithms used for example in routing and processor allocation can be extended to the entire class of hypercycles making it possible to choose a topology that best suits the system requirements of a specific class of applications.

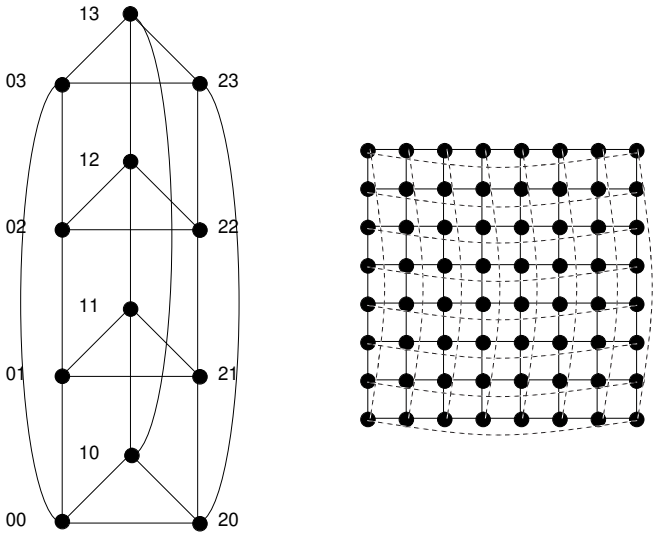

Figure 1. Example of hypercycles.
a:Hypercycle G_{34}^{11} *b: Torus* G_{88}^{11}

One of the crucial issues the designer of a concurrent machine faces is that of routing. Given that a computational node does not communicate directly with all the other nodes of a concurrent machine, sending a message from one computational node to any arbitrary node involves several intermediaries. The situation is even more complex given that several messages coexist and compete for communication resources at any given moment. Deadlocks and congestion may reduce the usable capacity of the interconnect to zero.

For hypercycle networks, we have devised several circuit-switching routing strategies that include deadlock-preventing and deadlock avoiding ones, and have reported the results in [8,9]. The implementation of these routing strategies takes the form

of a routing engine that is responsible for decoding, forwarding and generally managing the necessary operations for establishing a source to destination path and avoiding deadlocks. Because new routing strategies are developed or well known ones are adapted to new topologies, we are interested in developing a framework where a new routing engine can be developed and implemented and tested with the same ease that a new program is developed. In other words, we attempt to avoid the costly custom *one-of-a-kind* design and layout step.

The routing engine consists of a controller and several "helper" modules which implement specific functionality (e.g. a cross-bar, routers, encoders etc.). The "helper" modules can be thought of as implementing specific mechanisms required by the routing policy which is implemented by the controller. For any routing policy, the controller must be able to import specific message types (i.e. message headers) extract pertinent information from these messages (e.g. the destination address), perform some computation, and elicit the appropriate "helper" modules as need arises by exporting to them specific message types that can be understood and utilized by these modules. Since the specifics are application and policy dependent, the objective of this work is to develop a framework where the functionality of an arbitrary controller, which abides to the constraints outlined above, can be expressed and eventually embodied as a gate array or a VLSI circuit. In the following, we shall describe CoDeL, our *Controller Description Language* which can be used to specify the functionality of a specific controller, and shall present a design example of an e-cube controller for hypercycles.

2. CONTROLLER DESCRIPTION LANGUAGE (CODEL)

The controller's main function is to interact with a number of "helper" modules implementing specific functionality by importing, manipulating and exporting data and commands to them. The data the controller is asked to manipulate are embedded in bit-streams of specific formats and structure. Since the structure of these data as well as the way that the controller may obtain these data (i.e. the protocol of interaction) will be application specific, special attention has been paid in CoDeL to include powerful primitives that will allow the implementation of the anticipated functionality.

DATA TYPES

Data manipulated by a controller are organized in hierarchical structures called *frames*. Frames represent message headers or specific message types (e.g. module-setup, module-computation-request etc.).

A frame is recursively defined to consist of frames or bitfields at the lowest level, yielding thus a hierarchical structure. An example of a header frame is given in Figure 2.

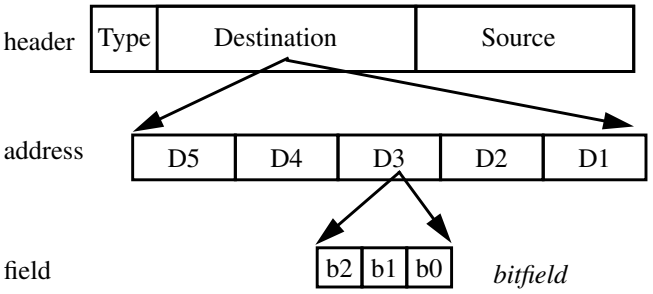

Figure 2. A typical frame for a message header. Both source and destination addresses are expressed as mixed radix numbers.

CoDeL uses hierarchical structures (`bitstruct`) to represent frames. A `bitstruct` (analogously to a frame) may be composed of collections of bits or `bitstruct`s. The example given in Figure 3. represents a structure composed of 28 bits and corresponds to the header in Figure 2.

```
bitstruct  field3
{
  (bits)  b[3];
}
bitstruct field2
{
  (bits)  b[2];
}
bitstruct address
{
  (field2)  d1;
  (field2)  d2;
  (field3)  d3;
  (field3)  d4;
  (field3)  d5;
}
bitstruct header
{
  (address)  source;
  (address)  destination;
  (bits)     type;
}
```

Figure 3. The bit structure corresponding to the frame in Figure 2.

Registers or ports are declared as having a particular structure through a declaration of the form

```
register(bitstruct_name)  register_name
inport   (bitstruct_name)  port_name with protocol_name
outport  (bitstruct_name)  port_name with protocol_name
```

The components of a register or port can be addressed at the bit level or the bitstruct level. We use a dot notation to identify the components under the convention that a sequence of dot separated fields represents a node in the tree representing the structure of the object and it consists of all the bits at the leaf nodes of the sub-tree rooted at the node. As an example, given the declaration

```
register (header) r1;
```

then the following are legal names and represent the corresponding bitfields.

`r1(5:0)`
the 1st to 6th bit are addressed (indices start from 0).

`r1.destination.d5`
represents the 14th to 26th bit (corresponding to the most significant field of the destination) are addressed.

PORTS AND PROTOCOLS

Frames must be imported from the environment (e.g. the network, the "helper" modules or the node itself) to the controller. Similarly, frames need be exported to the environment. Our model is that the controller interacts with its environment through specific protocols. Thus, an I/O operation is defined through a port, the definition of which specifies both the data path and the protocol to be followed. A port can be input or output. Once a port has been associated with a particular protocol, the interactions around the port are completely transparent to the user.

A port is an abstract data type that is accessed through a set of primitives which includes input, output and testing the state of the port. A port must be declared, and its declaration includes both its data path and the protocol that it uses.

```
inport   (bitstruct_name)  pi with protocol_name
outport  (bitstruct_name)  po with protocol_name
```

Protocols are supplied within a protocol library. CoDeL comes with a library of standard protocols, but the user can also develop additional protocols to account for the specific circumstances of a design. Protocols have been introduced so that the details of the I/O interaction are hidden.

OPERATORS

A number of operations may need be performed on the bitfields composing a frame. The assignment statement places the results of a computation to an output port or a register or portions thereof as discussed above.

register_expression = *computation_expression*

A computation_expression is formed using a number of standard C operators, (these include additions, shifts and rotates, bitwise logical operations) which follow the usual C associativity and precedence rules. Normally, a sub-expression of the computation_expression constructs circuitry that uses the result of preceding sub-expressions. Parentheses affect the structure of the constructed circuit.

CONTROL STATEMENTS

CoDel includes loop, conditional and wait primitives with definitions that remind C. The wait primitive keeps the circuit to the current state until a certain condition is satisfied. It is used mainly for synchronizing with external signals.

COMPILATION

CoDeL is compiled to produce synthesizable VHDL code. The compiler, denoted as `cco`, consists of the parser and the VHDL generator. The VHDL generator implements the data path as an RCR (Register Combinational Circuit Register), where data are stored in registers, operations are effected by a combinational circuit or a sequence of combinational circuits and the results are stored back in a register. The assignment statement in CoDeL reflects this model.

Associated with the data path, the control path sequences which operations are to take place and when the results are to be stored in the registers. It is implemented as a sequential machine.

One of the premises in developing CoDeL was to avoid the explicit description of the control path. Rather, the control path is automatically synthesized from the algorithm itself. In a sequential environment, the control path is inherently described by the order of the operations in a program. This is exactly the view taken in CoDeL. The control path of the design is extracted based on the sequentiality of the algorithm, and it includes states which explicitly clock the registers included.

Having each assignment (i.e. register loading) clocked with a different state of the machine has a straightforward implementation but has also the undesirable feature that an excessive number of

states is generated, one state for each assignment in the CoDeL program. A closer look reveals that some assignments can be done during the same machine state under certain conditions, reducing thus the state count significantly, and yielding smaller and faster circuits. The current version of the `cco` compiler provides such a functionality which is in fact an automatic parallelization of assignment blocks (i.e. set of consecutive assignments). The parallelizer determines the *data dependencies* among the statements and the assignments are scheduled based on the constructed *dependency graph*.

3. CONTROLLER ARCHITECTURE FOR ROUTING ALGORITHMS

A routing engine present at each node of the network consists of three major blocks, namely, the Router, the Crossbar and the Controller. We shall assume that the communication is bit-serial and bidirectional with circuit switching. The crossbar essentially switches the incoming bit-streams to the corresponding outports as determined by the controller. the third component namely, the router module, implements a generalized deadlock-free, e-cube routing scheme for hypercycles [9].

With the above framework, the actions of the controller can be easily described to forward a message from source to destination. At system power-up, the router is configured by the controller with parameters that define the topology of the implemented hypercycle and the address of the node. From that point on, it expects a destination address and it returns the "network port" through which the circuit is to be continued. It is the responsibility of the controller to ascertain that the required "network port" is available and continue the circuit. The router that we have implemented can be programmed for hypercycles of up to four-dimensions with a maximum degree 16. The mixed radix number system is used to represent addresses in the router.

If the "network port" returned by the router is free, then the circuit can be extended. The controller uses the address of the "network port" and sets the appropriate switches in the crossbar so that it can "inject" the header of the message on both directions of the chosen link. At the same time, the collision detection hardware that is incorporated in the injector detects any concurrent attempt by the recipient node to use the same link. Upon detection of a collision, the controller implements a collision resolution strategy based on the lexicographical ordering of the nodes of the network.

If no collision was detected, then the controller sets the crossbar switches so that the circuit is extended and marks the "network port" as used.

The cycle continues with the controller sequentially testing all network or host ports for incoming traffic, and the status of the switches that implement the currently active circuits. If a closed switch has been opened because of a break (i.e. termination of the circuit) then the controller updates its internal state of the ports to reflect that the previously used port is now free and continues the cycle. It is assumed that the switches at the network part of the crossbar are paired with switches at the controller part so as once the circuit is dissolved, the incoming link is connected to its corresponding port while the outgoing link is disconnected. Any new messages just arriving, will be captured by the ports and initiate a new routing cycle by the controller.

IMPLEMENTATION

The control algorithm for the above e-cube routing policy has been written in CoDeL and the resultant VHDL code was targeted to a XILINX 4010 PG191 chip with 158 I/O pins. For this particular example, we assumed a message consisting of a 4-bit header and two 16-bit addresses for the source and destination. Besides, 17 serial-to-parallel ports (including the host) present messages to the controller for routing decisions. I/O interactions of the controller with the crossbar and router are controlled through a synchronous strobed protocol. The size of the source code in CoDeL is 168 lines which was translated into a 1765-line Mentor VHDL code. Fragments of the CodeL program are shown in Figures 4 and 5. The control path of the sequential machine consists of 79 states of which it takes 25 clock cycles for the initialization, 20 cycles for interaction with the router module, 22 cycles for transmitting the message assuming there is no collision or a block and 12 cycles for the final destination.

The algorithm has also been adapted for a smaller 2-D network to reduce the I/O pin count and the resultant VHDL code has been mapped to a Xilinx 4010 PC84 chip. The code has been simulated for functional correctness and it is expected that a 20 Mhz clock or higher would provide commendable performance with a delay of no more than 2μs in routing a message.

```
bitstruct data_frame
# A 36-bit mixed_radix frame defn.
{
  (mixed_radix_4) source_address;
  (mixed_radix_4) destn_address;
  (bits)          header[4];
}
outport bus_enable[17];
inport  (data_frame) p1 with input_handshake;
inport  (npg_port_frame) p2 with input_ecube;
outport (data_frame) p3 with output_handshake;
outport p4[19] with output_ecube;
outport (cross_bar_frame) cf
                    with output_handshake;
inport  config;
inport  data_sent, collision;
inport  available_ports[17];
```

Figure 4. CoDeL fragment that defines the ports used in the example controller. Only the `data_frame` declaration is included here.

```
while (i <= 17) # sample each port
{
  shift_value = shift_value << 1;
  bus_enable  = shift_value;
  if (isready(p1))
  {
    # test for availability of data
    input(p1);
    destn = p1.destn_address;
    # extract the destination address from p1
    p4 = (5,available_ports);
    # concatenate available ports with 5
    output(p4);
    # send available ports to router
    p4 = (2,destn);
    output(p4);
    input(p2);
    # Receive computed port and control
    # signals (from router) in port p2
            ...
```

Figure 5. Fragment of the start of the main loop. Each of the Serial/Parallel (S/P) interfaces is polled sequentially, if it has valid data, the header is imported, the destination address is extracted and sent to the router, and then it inputs the resulting "network port".

4. CONCLUSION

In this work, we have formulated a special purpose hardware description language that provides elementary functions and data structures. These will serve as basic building blocks for transcribing the actions of the controller. Re-programmability is a major advantage of this approach when the user needs to change the controller's function for instance to have a different routing policy or when the network configuration changes. Hence fast silicon compilation is possible. A rigorous characterization, validation and performance analysis of the afore mentioned Xilinx implementations will be carried out shortly. In addition, optimization of the compiler, minimization of redundant states, introduction of loop indices and multi-module instantiation are areas for future work which will greatly augment the specification of more complex routing polices and produce efficient circuits.

REFERENCES

1. C. L. Seitz, "The cosmic cube", CACM, vol. 28, pp. 22 - 33, Jan 1989
2. R. D. Rasmussen, N. J. Dimopoulos, G. S. Bolotin, B. F. Lewis, and R. M. Manning "MAX: Advanced General Purpose Real-Time Multicomputer for Space Applications" *Proceedings of the IEEE Real Time Systems Symposium* pp. 70-78, San Jose, CA., Dec. 1987.
3. R. D. Rasmussen, G. S. Bolotin, N. J. Dimopoulos, B. F. Lewis, and R. M. Manning "Advanced General Purpose Multicomputer for Space Applications" *Proceedings of the 1987 International Conference on Parallel Processing* pp. 54-57, 1987.
4. iPSC User's Guide, No. 17455-3, Intel Corp., Portland, Ore., 1985.
5. Peterson, J.C., J. O. Tuazon, D. Lieberman, M. Pniel "The MARK III Hypercube -Ensemble Concurrent Computer" *Proceedings of the 1985 International Conference on Parallel Processing* pp. 71-73, 1985.
6. E. Chow, H. Madan, J. Peterson "A Real-Time Adaptive Message Routing Network for the Hypercube Computer" *Proceedings of the Real-Time Systems Symposium,* pp. 88-96, San Jose CA., 1987.
7. N. J. Dimopoulos, D. Radvan, K.F. Li "Performance Evaluation of the Backtrack to the Origin and Retry Routing for Hypercycle based Interconnection Networks" *Proceedings of the Tenth International Conference on Distributed Systems*, Paris, pp. 278-284, 1990.
8. R. Sivakumar, N. J. Dimopoulos, V. Dimakopoulos, M. Chowdhury, D. Radvan "Implementation of the Routing Engine for Hypercycle Based Interconnection Networks" *Proceedings of the 1991 Canadian Conference on Very Large Scale Integration* pp. 6.4.1-6.4.7, Kingston, 1991.
9. N. J. Dimopoulos, and R. Sivakumar, "Deadlock-preventing routing in Hypercycles", *The Canadian Journal of Electrical and Computer Engineering*, No. 4, vol. 19, Oct. 1994, pp. 193-199.
10. Mentor Graphics Corporation, AutoBlocks Reference Manual, v8.2_5, 1994

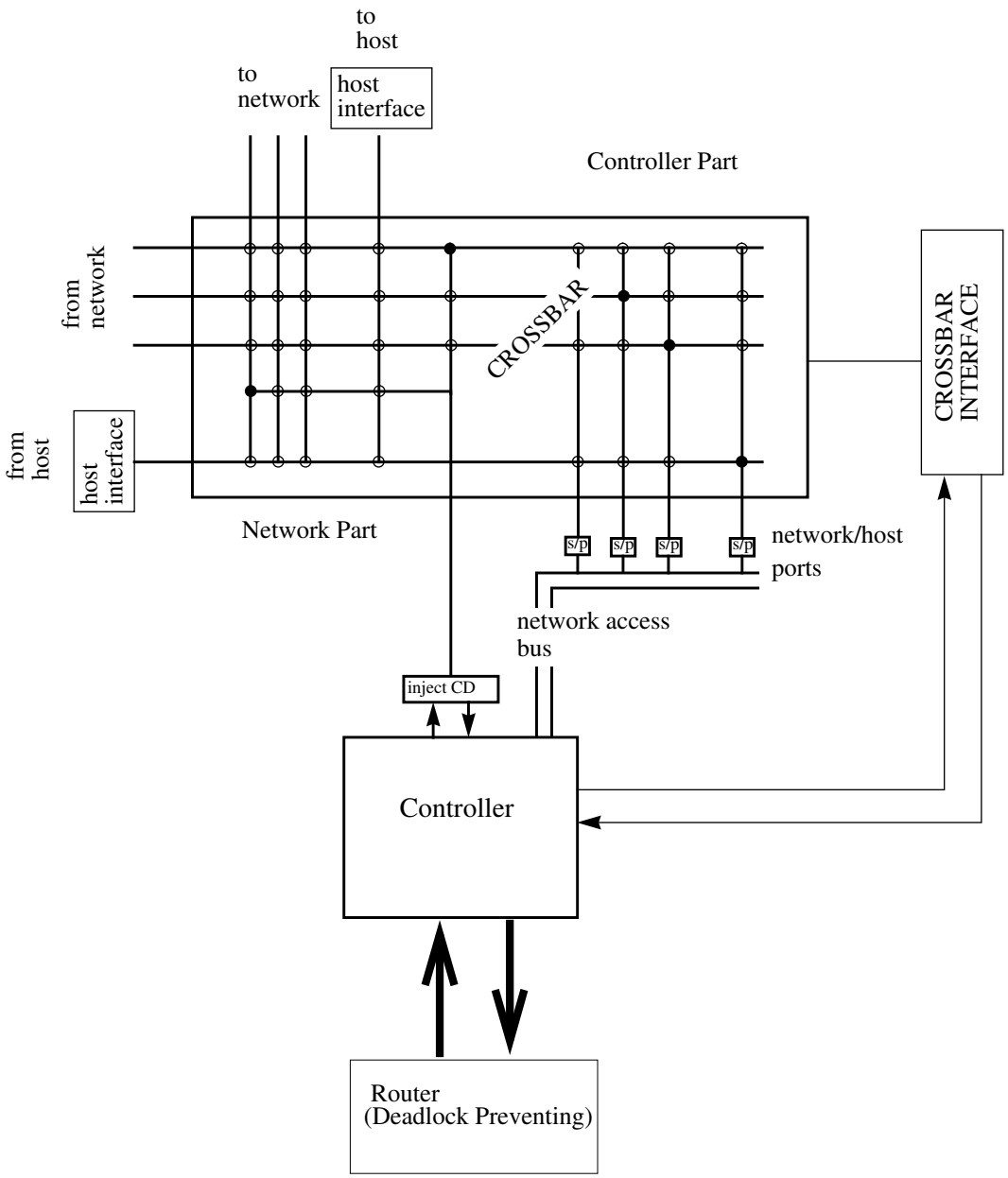

Figure 6. Structure of a circuit switching routing node in a hypercycle network.

- ● Switch that is closed
- ○ Switch that is open

S/P Serial to Parallel