Using a Reconfigurable Switch to improve MPI performance

Edson L. Horta, Sergio T.Kofuji
Laboratory of Integrated Systems - Department of Electronic Engineering
Escola Politécnica da Universidade de São Paulo, São Paulo, Brazil
Av. Prof. Luciano Gualberto, trav. 3, 158. CEP 05508-900 - São Paulo, SP, Brasil
Tel: (011) 818-5589 fax: (011) 211-4574. horta@lsi.usp.br, kofuji@lsi.usp.br

Abstract

This work describes how reconfigurable logic can be used to design a switch that can improve the performance of some MPI functions in a parallel machine. An introduction to the MPI Collective Operations is done, along with the various schemes used to build a reconfigurable system. We show that, with the reconfigurable switch, it could be possible to build an Active Network.

Introduction

The MPI Collective Operations constitute an important part of a Parallel Program execution and normally they are implemented by software. These functions depend also on how the communication network interconnects each one of the nodes in the machine.

Using reconfigurable logic to build a switch that interconnects the nodes will improve the performance of the MPI Collective Operations (MPI CollOps). This technology permits, through a reconfiguration "on the fly" of the switch, the implementation of the CollOps directly in hardware.

In the next sections it will be explained how some MPI CollOps work and the main aspects of a reconfigurable circuit. Then it will be shown how to modify a conventional switch architecture to support the reconfigurability in order to implement those CollOps.

MPI Collective Operations

MPI (Message Passing Interface) [MPI 95] is a message-passing application programmer interface that specifies the semantics and protocols used to implement the communication between processes of a parallel machine. These processes have to be grouped according to the user needs and can communicate in two formats: point-to-point and global. The second forms a subtype of functions defined as Collective Operations.

Some of Collective Operations defined in MPI Standard V.1.1 are :

a) MPI_Barrier : used to synchronize all the members of a group. The call returns only after all group members have entered the call.
b) MPI_Bcast (Broadcast) : used to send a message from one of the processes (called root) to all the others.
c) MPI_Gather : used to send different messages (with the same size) from each process to the root.
d) MPI_Scather : used to send different messages (with the same size) from the root to each process.
e) MPI_Reduce : used to combine different messages from all the process and send the result to the root.

All the Collective Operations assumes that there is a spanning tree structure to support the communication between the processes, in order to improve the performance [GL 96].

For example, if we had a group of three processes and would like to implement the MPI_Bcast from root to processes 1,2, and 3, they would be in a structure like this :

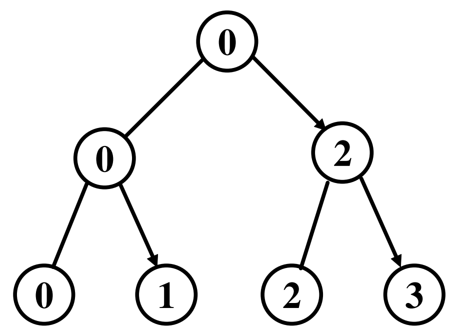

Fig. 1 - Spanning Tree of a Parallel Machine

The structure above needs only two stages to send the messages, otherwise the structure without the tree would need three.

Reconfigurable Logic

In order to build a more flexible switch it is reasonable to use Reconfigurable Logic. This is a technology that takes advantages from the flexibility of the FPGA devices. These devices can be reprogrammed an unlimited number of times, allowing the user to change the features of a product as soon as it needs them.

According to [HW 95], the implementation of a system that uses reconfigurability can be done in two ways: Compile-Time and Run-Time Reconfiguration.

In Compile-Time Reconfiguration (CTR) the FPGA does not change its configuration during the application lifetime. Each application has its specific functions that are loaded when the user starts it. This way is very similar to the use of an ASIC to implement a system. The main difference between them is the reprogrammability of the FPGA, that allows its re-utilization. When using CTR, it will be necessary to develop discrete images (partitions) for the application. Then each one of them will be programmed into a FPGA. These images will behave as static hardware during all the life of the project.

The Run-Time Reconfiguration (RTR) allows to change the functions of the system during the application execution. This approach introduces two new design problems: the partition of the system and the communication between them. When partitioning a system, one has to divide it into time-exclusive segments that have to run non concurrently, enabling each one to be programmed independently in the FPGA. This partition must be designed according to a temporal partition, visualizing the concurrent parts of the circuit and grouping them in each of the configurations loaded by the FPGA. Another important aspect in the project of a RTR system is how each configuration scheme will communicate with each other. It will be necessary to preview some procedures to store the intermediate values, wait for the reconfiguration process and retrieve the values to continue the process.

The implementation of RTR can be done in two ways: global or local.

The global RTR reconfigures all the FPGA when it changes the configuration. In order to have an efficient partition of configuration, one has to go through an iterative process that divides the system into segments using as many as reconfigurable resources as possible. This is an iterative process because no one knows the physical size of a partition before it has been placed and routed. After the partition process it is necessary to design the interfaces between the configurations and the communication between them. Because all the FPGA is used during the reconfiguration, it is necessary to have an area to store the common data and each configuration has to treat this data correctly. One of the disadvantages of global RTR is the time spent to change the configuration data in the FPGA. The advantage of this method is its simplicity, making it possible with the use of available commercial tools.

The local RTR consists of programming only a part of the FPGA, reducing the reconfiguration time. The partition process of this implementation is based on a functional division of the system. The functional parts used in all the application lifetime are programmed only once, during the initialization process. Then, when the application needs a specific function, the part used to perform it is loaded in place of a part no more useful. To have the reconfigurable resources utilized in an efficient manner, it's necessary to eliminate all the idle circuit or reduce it to a minimum. One problem that comes with local RTR is concerned to the physical space utilization. Because the FPGA will support multiple configurations at a time, it's necessary to predict the available space for them, preventing from the multiple occupation of the same physical space during an application execution.

A FPGA must have a reconfiguration time very fast to be utilized as a RTR device and the ability of maintaining some of its internal states, when reconfigured.

Some examples of CTR systems are SPLASH (a Reconfigurable Linear Logic Array) [GH&al 94] and PAM (Programmable Active Memories) [PDJ 93] [PH 94] an some examples of RTR systems are Imputer (Imaging Computer) [RVT 93] and RRAN 2 (run-time reconfigurable artificial neural network) [HH 95].

The Reconfigurable MPI Switch

It is possible to implement the MPI CollOps directly in hardware, using a reconfigurable switch. A switch is a set of input and output ports that interconnects users, other switches or other network elements. Its architecture can be seen in the figure 2. According to [CL 95] it can be divided in: SM (System Management), CAC (Connection Admission Control), IM (Input Modules), OM (Output Modules), and CSF (Cell Switch Fabric).

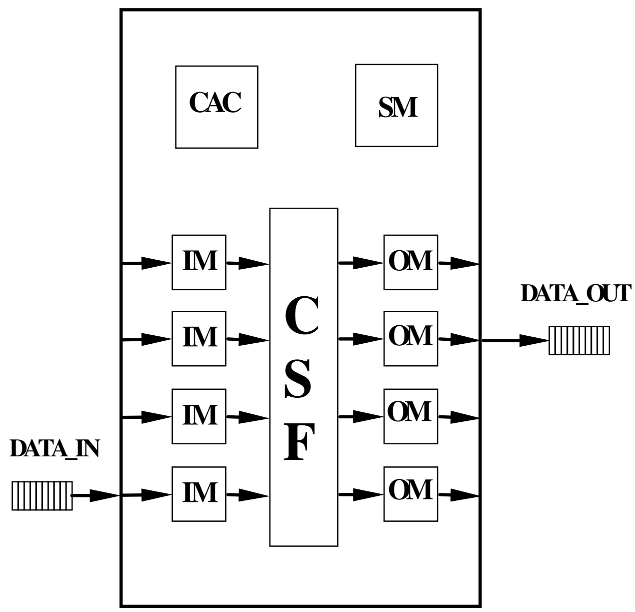

Figure 2 - Switching System

The System Management (SM) module is responsible for performing all the management and traffic control functions to ensure the correct and efficient operation of the switch. One can divide this function in six types:

- Fault management;
- Performance management;
- Configuration Management;
- Accounting management;
- Security management;
- Traffic management.

All these functions may be distributed to the other blocks of the switch. What will dictate this distribution is the performance and the costs involved in order to achieve it

The Connection Admission Control (CAC) module processes and interprets signaling information and decides about the connection admission or rejection. This block is responsible for the interaction with other users and other switches through signaling, and resource allocation decisions for requested connection.

The Input Modules (IM) treat the incoming cells and prepare them for routing through the CSF (explained later). It's possible, also, to have some functions of CAC and/or SM integrated on the IMs. This integration yields a better performance, but it's more expensive to implement.

The Output Modules (OM) receives cells from the CSF and prepares them for physical transmission on the output links. This block is simpler than the IM and may have some functions of CAC and/or SM integrated on it.

The Cell Switch Fabric (CSF) routes the user data cells from input to output ports and possibly route signaling and management cells between the other functional blocks in the switch. Besides this main task (routing), the CSF is responsible for functions, such as:

- Cell buffering;
- Traffic concentration and multiplexing;
- Redundancy for fault tolerance;
- Multicasting or broadcasting;
- Cell scheduling based on delay priorities;
- Selective cell discarding based on loss priorities;

To transform the structure above in a reconfigurable one it is necessary to include a Reconfigurable Control Block, a Configuration Memory block and make the CSF programmable (see figure 3).

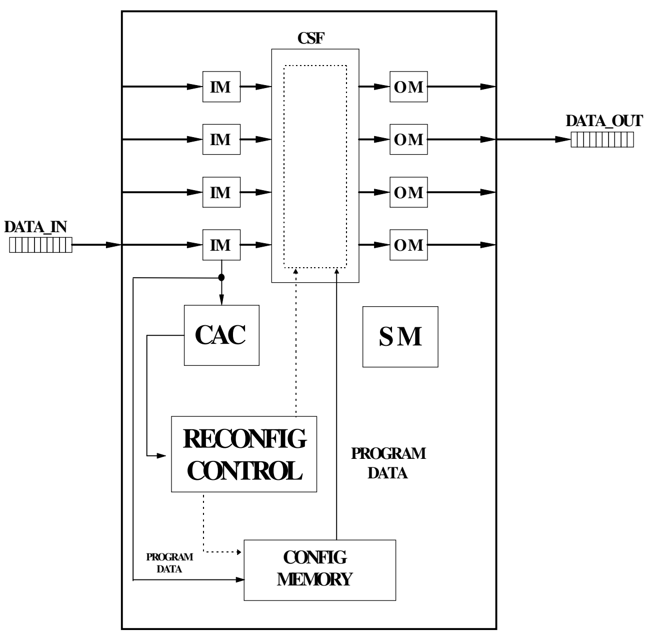

Figure 3 - Reconfigurable Switch

The CAC block will monitor the data entering the switch by each one of its input ports. When some special headers of data appear, the CAC sends them to the RECONFIG CONTROL.

This block has some status registers, to monitor each one of the ports. This register is shown in the figure 4.

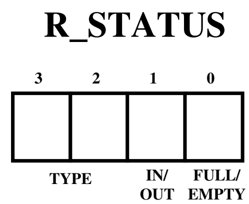

Figure 4 - Port Status Register

According to the figure above, each port of the switch can be one of four types (bits 3 and 2), each one can be an input or an output port (bit 1) and the last bit of the status register indicates whether the buffer port is empty or full. There is also a configuration flag that will indicate if the switch is in the default or in the reconfigured state.

When a port is inactive, the bits 3 and 2 will be zero. The RECONFIG CONTROL block will monitor these bits and, if all of them, for each port are zero and the configuration flag is 1, it will reconfigure the switch to the default state.

This block will wait for an opcode that comes from the CAC and, according to the state of each port, will choose one of the configuration data from the CONFIG MEMORY and will generate the signals necessary to reconfigure the CSF.

If a special opcode is transmitted to port 3, the RECONFIG CONTROL block will begin to store the data in the CONFIG MEMORY, allowing the users to change the possible reconfigurable states of the switch.

In order to implement the COllOps in a Reconfigurable Switch some assumptions have to be assumed:

- the MPI_Barrier will have to be called before any other CollOp;
- there must be only one active CollOp at a time;
- each node can have only one process at a time;
- the tree structure will be based on the switches;
- there must be special sends and receives inside the MPI CollOp routines that encapsulates opcodes in the messages to the switches, in order to them to know what configuration file to use;
- the MPI CollOp routines will have only to worry about sending/receiving messages to/from the switches, not doing any manipulation of data;
- when a communicator is created, it won't be possible the communication among the process outside it, until the MPI CollOp finalizes;
- the switch will have two possibilities of in/out ports: three in and one out; one in and three out;
- it will be necessary to define the maximum values for the number of entries in each operation, in order to allocate the space in the switch;
- each communicator must have at least three processes;

The switch that will be explained is supposed to be used alone and has four ports. It will have two states: normal and reconfigured. During the first one it will work as a simple Switch (sending messages from its input ports to its output ports). In the second mode it will have some predefined configurations that will be loaded according to the MPI Collective Operation functioning at the moment.

Each switch will have two buffers: IN and OUT. These buffers will be connected with each port buffer and will transfer data with each other according to the MPI CollOp. Sometimes it will be necessary to have a combinatorial logic between these buffers, in order to implement the MPI CollOp.

The types of port (see previous section) used to implement the MPI CollOps will be:

- inactive (00);
- process (01);
- root (10);
- not used (11);

The MPI_Barrier call will program all the ports used by the communicator as 01 (process). After this, when any process calls the MPI_Bcast function, there will be inside it a call to a special send, called here **sendroot,** that will identify the root and program its port as type 10 and as an output port (bit 1=1). The sendroot will also send the opcode to the switch.

When the switch notes that the opcode is different from zero, it starts the process of reconfiguration, programming the CSF with the data stored in the CONFIG MEMORY.

For the time being, each port of the switch is ready to deal with the special receives functions (recBcast, for example) of the other processes. When each one of the processes reaches the end of transmission/reception, it programs the status registers of the ports as inactive, allowing the CONFIG CONTROL block to reconfigure the switch back to the default mode as soon as all the ports become inactive.

The switch will send data only to the nodes that will use it, otherwise it will send only a status signal. For example: in a MPI_Bcast, the root will only send data.

If the interconnection of the nodes from a parallel machine is made with switches that uses reconfigurable logic, this network is known as an Active Network (AN) [TS&al 97]. An AN is made of intelligent nodes and data packets that contain encapsulated programs[TW 96]. Through these programs it is possible to have, for example, a specific fault management feature for each group in the network. Or, if one wants to program a different protocol between two or more users, it will not be necessary to wait for a standardization process to use this protocol.

Conclusions

It was shown that is possible to modify the structure of a switch to permit its reconfiguration through the contents of a special message traversing it. With this kind of switch one is capable to build an interconnection network for parallel machine based in an Active Network.

The specification of the switch was obtained, along with the improvement in one MPI Collective Operation.

Another feature that has to be highlighted is the flexibility of a reconfigurable switch, yielding the implementation of new protocols in hardware, saving much of developing time, wasted in the process of standardization.

Acknowledges

The authors thank: University of São Paulo and FEI (Brazilian Academic Centers) for their technical support and CNPq and FINEP of Brazil for their financial support.

References

CL 95 T. M. Chen, and S. S. Liu, ATM Switching Systems, 1995 , Norwood, MA : Artech House, Inc, 1995.

GH&al 94 M. Gokhale, Bill Holmes. Andrew Kopser, Dick Kunze, Dan Lopresti, Sara Lucas, Ron Minnich, Peter Olsen, SPLASH : A Reconfigurable Linear Logic Array, January 1994

GL 96 William Gropp, Ewing Lusk, Nathan Doss, and Anthony Skjellum, "A High-Performance, Portable Implementation of the MPI Message Passing Interface Standard," Preprint MCS-P567-0296, July 1996.

HH 95 J.D. Hadley, B. L. Hutchings. "Designing a partially reconfigured system," in Field Programmable Gate Arrays (FPGAs) for Fast Board Development and Reconfigurable Computing, John Schewel, Editor, Proc. SPIE 2607, pp. 210-220 (1995).

HW 95 B. L. Hutchings, M.J. Wirthlin. "Implementation Approaches for Reconfigurable Logic Applications", In 5th International Workshop on Field Programmable Logic and Applications, pp 419-428, August 1995, Oxford, England.

MPI 95 Message Passing Interface Forum. MPI: A Message Passing Interface Standard V.1.1. University of Tennessee. June 1995. cp 1 - 175

PDJ 93 Bertin Patrice, Roncin Didier, and Vuillemin Jean, Programmable Active Memories: a Performance Assessment, Research on Integrated Systems: Proceedings of the 1993 Symposium, G. Borriello and C. Ebeling, editors, pp. 88-102, MIT Press, 1993.

PH 94 Bertin Patrice and Touati Hervé. PAM Programming Environments: Practice and Experience, IEEE Workshop on FPGAs for Custom Computing Machines, D. A. Buell and K. L. Pocek, editors, Napa, CA, USA, 1994.

RVT 93 D. Ross, O. Vellacott, and Turner. An FPGA-based hardware accelerator for image processing. In W. Moore and W. Luk, editors, More FPGAs : Proceedings of the 1993 International workshop on field-programmable logic and applications, pages 299-306, Oxford, E

TS&al 97 D. L. Tennenhouse, J. M. Smith, W. D. Sincoskie, D. J. Wetherall, and G. J. Minden, "A Survey of Active Network Research", IEEE Communications Magazine, Vol. 35, No. 1, pp80-86. January 1997.

TW 96 D. L. Tennenhouse, D. J. Wetherall, Towards an Active Network Architecture. Computer Communication Review, Vol. 26, No. 2, April 1996.