PREDICTIVE LOAD BALANCING FOR INTERCONNECTED FPGAS

Jason D. Bakos, Charles L. Cathey, E. Allen Michalski

Department of Computer Science and Engineering
University of South Carolina
Columbia, SC 29208
e-mail: {jbakos, catheyc, michalsk}@cse.sc.edu

ABSTRACT

A Field Programmable Gate Array (FPGA), when used as a platform for implementing special-purpose computing architectures, offers the potential for increased functional parallelism over the alternative approach of software running on a general-purpose microprocessor. However, the increasing disparity between the logic speed and density of a state-of-the-art FPGA versus a state-of-the-art microprocessor has already begun to negate the benefits of this increased functional parallelism for all but a limited set of applications. We believe that the solution to this problem is to construct distributed multi-FPGA architectures to aggregate the parallelism of multiple FPGAs. Such a system would require a high-capacity interconnect, and thus we propose arranging the FPGAs onto a scalable direct network. This strategy requires each FPGA to contain an integrated router that must share the logic fabric with the application logic. In this paper, we propose a novel routing technique that can significantly boost such a network's capacity and be implemented into compact and efficient routers. We begin with an existing lightweight routing algorithm and augment it with a novel technique called *predictive load balancing*, where routers collect information about the blocking behavior on their output ports and use this information when making routing decisions.

1. INTRODUCTION

Direct networks are a popular technique for interconnecting the nodes of large-scale multiprocessor systems [1]. In this approach, instead of connecting each of the processing nodes to a global high-radix router, each node is directly connected to a low-radix router and each of these routers is connected to a small set of neighboring routers. Messages may be routed from any source node to any destination node through two or more routers. The move from a single high-radix router to a collection of several low-radix routers shifts the complexity growth of the interconnect from quadratic to linear at the cost of additional transmission latency. We have recently proposed the use of a wormhole-switched 2D mesh network as a viable option for constructing large-scale multiple-FPGA processing architectures [2]. In this architecture, each FPGA is configured such that it is logically partitioned into a processing element and a router. Each FPGA is connected to its neighbors through channels formed by its integrated multi-gigabit transceivers (MGTs).

A 2D mesh topology consists of a 2D array of routers with each router having five bidirectional ports. Four of these ports connect to the logically adjacent neighbor routers (north, south, east, west) while the fifth is used as an internal channel to the processing element (inject, eject). In wormhole switching, variable-length packets are subdivided into a sequence of fixed-length *flits* (flow control digits) which themselves are routed across the network in a store-and-forward fashion (the flits would undergo serialization and de-serialization when being transmitted across MGTs). The first flit in a packet is a header flit that contains a relative destination address for the packet. In the case of a 2D mesh, this address is represented as a pair of signed values that specify the packet's offset in both the X and Y dimension. The remaining flits are payload flits that contain the packet data as well as a flag bit that denotes if the flit represents the last flit of the packet. When a header flit enters an input port on a router, the router's internal crossbar switch is configured such that all flits entering this input are forwarded to an appropriate output port. Even if only minimal routes are allowed, the router must make a route decision consisting of up to two possible directions. For example, a flit entering a router that is destined for a node that is south-east of its current location may be forwarded to the south or east output port. Immediately after the last payload flit is forwarded, the router's internal switch is reset such that the output port is unassigned. The primary advantage of wormhole switching is the ability for the router designer to design the buffers at each input port at the flit-granularity as opposed to at the packet-granularity. A disadvantage of this technique is that the flits that make up an in-flight packet will occupy input ports at multiple routers along its path. This leads to blocking behavior caused by contention for input buffers (ultimately caused by contention for crossbar output ports). Another side-effect of wormhole switching is the possibility of deadlock caused by a circuit of blocked packets. Turn-based routing prevents deadlock by

```
route(block_hist[outputs],
      aval_dirs[], // set by O-E routing
      flit) {

   pref_dir = aval_dirs[0];
   nonpref_dir = aval_dirs[1];

   if size(aval_dirs) == 2 {
      if block_hist[aval_dirs[0]] >
         block_hist [aval_dirs[1]] then
         pref_dir = aval_dirs[1];
         nonpref_dir = aval_dirs[0];
   }

   if crossbar_output(pref_dir) is available
   {
      block_hist[pref_dir]--;
      configure crossbar and route packet;
      return;
   } else {
      block_hist[pref_dir]++;
   }
   if crossbar_output(nonpref_dir) is available
   {
      block_hist[nonpref_dir]--;
      configure crossbar and route packet;
      return;
   } else {
      block_hist[nonpref_dir]++;
   }
}

forward(block_hist[outputs]) {
   for each configured crossbar output i {
      attempt to forward flit from corresponding input
         port;
      if output i is blocked
         block_hist[i]++;
      else
         block_hist[i]--;
}
```

Fig. 1. Pseudocode for Load-Balancing Algorithm.

restricting routes such that two of the eight possible turns are prohibited [4]. Turn-based routing is referred to as semi-adaptive routing, as it restricts packets to a subset of their possible minimum paths from source to destination. Such an algorithm constitutes the network's routing algorithm, which defines the set of possible paths that a packet from a given source/destination pair may follow through the network. A deterministic routing technique restricts this set of possible paths to one, while adaptive routing allows the size of this set to grow as an exponential function of the source/destination distance. However, this is not always an advantage, since bad routing decisions may lead to even worse performance than deterministic routing. Unfortunately, individual routers cannot make routing decisions based on knowledge of downstream network congestion. The reason for this is that while individual routers may make routing decisions based on the status of their own ports, they do not possess global knowledge of the current and future state of the network.

We believe the best solution to this problem is for each router to maintain historical information of blocks on its own output ports and make routing decisions based on this information. This allows each router to take advantage of "feedback" information to which the router has access

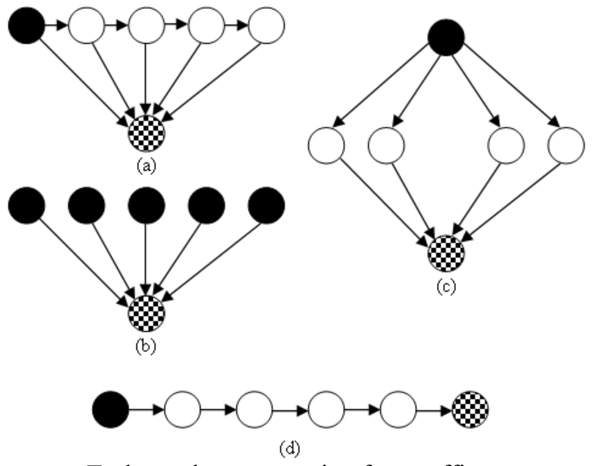

Fig. 2. Task graphs representing four traffic patterns: (a) linear fan-in, (b) fan-in, (c) diamond, (d) linear. Source tasks are black, sink tasks are checkerboard.

when forwarding payload flits from a packet that is blocked at a downstream router. This load-balancing technique must operate over an underlying routing algorithm that is adaptive and deadlock-free. We therefore chose the Odd-Even Turn Model because it achieves semi-adaptive routing without the requirement of high-overhead virtual channel flow control [5]. This routing algorithm guarantees freedom from deadlock by imposing the following restrictions: no 180-degree turns are allowed, packets entering from the west input port into a router located in an even column may not make a 90-degree turn (no WN or WS turns), and packets entering a router located in an odd column may not take a 90-degree turn to the west (no SW or NW turns). A router that implements this algorithm must also take special care to avoid dead-end routes. For example, if a packet is allowed to enter from the west into an intermediate router in an even column that contains the destination node, it will not be able to turn to the north or south to reach the destination.

2. PREDICTIVE LOAD BALANCING

Predictive load balancing operates on a principle analogous to branch predictors for a microprocessor. In this technique, each router keeps a running sum of the number of blocks that have occurred when forwarding flits to each output port. A block can occur when a router cannot *route* a header flit due to contention for the router's internal crossbar output ports (referred to as an internal block) or when a router cannot *forward* a payload flit to its corresponding output port due to an internal block within one of the downstream routers (referred to as a downstream block).

Any internal or downstream block will result in the block count being incremented for the corresponding output port. On the other hand, any flit that is successfully *routed* or *forwarded* through the router's internal crossbar results in the corresponding block count being decremented. In order to test the upper bound of performance for our load balancing technique, we currently do not consider "wrap-around" conditions for the block count value (when the value exceeds the maximum possible value given the bit-width).

When more than one routing option is available, the router consults the number of blocks that have previously occurred before it makes a routing decision. This allows the routers to choose preferred routing paths in order to evenly distribute traffic across the network. This technique works best for applications that exhibit consistent, periodic traffic patterns.

Pseudocode for predictive load balancing is shown in Figure 1. Our load balancing algorithm consists of two main components. The first component, **route**, performs load balancing and records blocking history for header flits. In this routine, **aval_dirs[]** is set by the underlying routing technique and defines the allowed output ports for the packet. The second component, **forward**, records blocking history for payload flits.

3. SYSTEM AND TRAFFIC MODEL

Our system model consists of a 16 x 16 bidirectional mesh of interconnected *nodes*, where each node represents a single FPGA. Each node consists of a single *processing element (PE)* and a single *router*. Each input port of each router has a buffer capacity of one flit. Each processing element is configured to hold a hardware *task*.

In order to model traffic characteristics of a generic hardware application, we assume that multiple independent hardware *applications* are mapped onto the sample multi-FPGA processing architecture. Each application is represented as a directed acyclic graph (DAG), referred to as a *task graph*, where each vertex represents a task and each edge represents a data dependency. As specified by the task graph, each task accepts one or more inbound packets from one or more other tasks. Each packet has a length of 20 flits. Each task must receive all of its inbound packets before it may begin performing its execution. The execution time of each task is fixed at 2000 network clock cycles (100x the transfer time of an unblocked packet though a router). After its execution time has elapsed, the task transmits one or more outbound packets to one or more other tasks. Each task graph contains a set of one or more tasks that are designated as *source tasks*, which is not required to wait for source packets before beginning execution. The source task(s) repeat execution and resultant transmission of outbound packet(s) every *p* network clock cycles, where *p* is a predefined *period time*. This behavior is continued indefinitely. Each application also has a set of one or more tasks that are designated as *sink tasks*, which do not transmit packets when they complete their computation. At the instant when all sink tasks for a given task graph have received their inbound packets and their required execution time has elapsed, the task graph is considered to have completed an application execution.

For our experiments, we pseudo-randomly mapped each of the tasks from 8 independent task graphs of 32 tasks each onto each of the PEs, without allowing any two or more tasks from any single task graph to be mapped onto a single PE. The periods of each task graph are equal, and the period time for the graphs is varied in order to simulate varying network load. Figure 2 illustrates four classes of task graphs that are used to simulate traffic patterns (the task graphs shown contain 6 tasks each, whereas the tested graphs contain 32 tasks). These graphs were chosen to model traffic patterns from common multi-processor applications. In order to measure the effectiveness of our load balancing technique, we ran a series of network simulations that measures the average application execution time over all task graphs over 500 executions of each graph. We tested the following routing/load balancing algorithms. For these algorithms, when multiple header flits are waiting for a single output channel, they are serviced in the order of their arrival.

- *OEN: odd-even routing with naïve load balancing*

This routing algorithm is minimal and semi-adaptive, meaning that intermediate routers may chose to route packets in up to two possible directions. At least one direction is always allowable, and two directions are allowable to routers that do not occupy the same row or column as the destination node and when neither direction results in an illegal turn or would lead to an inevitable dead-end condition. When two routing directions are both allowable and available (crossbar output is unbound and unblocked), the router chooses the output along the Y dimension (north/south).

- *OEP: OE routing /predictive load balancing*

In this technique, each router uses the odd-even routing rules to determine a set of allowable routing directions in order to prevent deadlock and dead-end routes. When two directions are both allowable and available, the router utilizes the load balancing algorithm shown in Figure 1 to choose an output port.

4. RESULTS

In order to measure the performance of our load balancing technique, we have implemented the above network model as well as a corresponding flit-level simulator in Java [5]. The simulator operates on a network-cycle granularity, where one network cycle is required to transfer a flit across a router-to-router, router-to-PE, or PE-to-router channel. One network cycle is also required for a router to perform a routing operation. The network model was simulated over a range of period times, allowing each traffic pattern to be

Table 1. Average time to execute task graphs with **hybrid linear fan-in** traffic pattern.

period	OEN	OEP	OEP/OEN
700	341223	324091	0.9498
750	253493	241928	0.9544
800	185461	141163	0.7611
850	140704	92935	0.6605
900	106519	68820	0.6461
950	73313	67263	0.9175
1000	67927	66976	0.9860

Table 2. Average time to execute task graphs with **fan-in** traffic pattern.

period	OEN	OEP	OEP/OEN
1700	593639	6490	0.0109
1750	357764	6090	0.0170
1800	283319	5950	0.0210
1850	114548	5877	0.0513
1900	29515	5897	0.1998
1950	8861	5851	0.6603
2000	5829	5734	0.9837

Table 3. Average time to execute task graphs with **diamond** traffic pattern.

period	OEN	OEP	OEP/OEN
800	209010	156276	0.7477
850	168090	139291	0.8287
900	145740	117860	0.8087
950	120365	95485	0.7933
1000	98804	82559	0.8356
1050	80919	74126	0.9161
1100	66158	68646	1.0376

Table 4. Average time to execute task graphs with **linear** traffic pattern.

period	OEN	OEP	OEP/OEN
100	394806	398027	1.0082
150	214202	206466	0.9639
200	123940	122923	0.9918
250	81205	85196	1.0491
300	65585	66450	1.0132
350	65412	65517	1.0016
400	65360	65434	1.0011

operated over a range of network load conditions. Lower periods correspond to higher network load. Each simulation is run until all task graphs complete at least 500 executions. We estimate performance by computing the average execution time of each task graph over all execution periods. Lower execution times reflect less blocking behavior and lower packet latencies.

Tables 1-4 list the average execution times required by each of the task graphs assuming OEN and OEP routing over a period window of 7 with a granularity of 50 cycles. Also shown is the relative performance of OEP to OEN, by dividing the execution time of OEP by OEN. These particular result windows were selected by fixing the highest period value to the period time at which the performance of both routing algorithms converges, meaning that the period is sufficiently high that there is insignificant network contention and thus insignificant blocking behavior (when the ratio exceeds .95, except in the linear case).

For the **linear-fan-in** traffic pattern, OEP exhibits a reduction in execution time over OEN by up to 36%. For the **fan-in** traffic pattern, OEP exhibits a reduction in execution time over OEN by up to 99% for the periods range selected. For the **diamond** traffic pattern, OEP exhibits a reduction in execution time over OEN by up to 21% for the periods range selected. For the **linear** traffic pattern, OEP does not exhibit any appreciable difference in performance over the OEN algorithm. From the results, it is clear that predictive load-balancing yields significant improvement over naïve load balancing for traffic patterns that contain a fan-in communication behavior. This includes hybrid patterns that contain both fan-in and linear or fan-in and fan-out (diamond pattern) behavior.

5. CONCLUSION

This paper describes a novel load-balancing technique for 2D meshes that may be implemented within area efficient integrated routers. It has been verified to achieve higher performance relative to traditional approaches for applications that exhibit fan-in traffic behavior.

6. REFERENCES

[1] L.M. Ni, P.K. McKinley, "A survey of wormhole routing techniques in direct networks," IEEE Computer, Volume 26, Issue 2, Feb. 1993 Page(s): 62 - 76.

[2] Charles L. Cathey, Jason D. Bakos, Duncan A. Buell, "A Reconfigurable Distributed Computing Fabric Exploiting Multilevel Parallelism," IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM 2006), April 24-26, 2006.

[3] C.J. Glass, L.M. Ni, "The Turn Model for Adaptive Routing", Proc. 19th Ann. Int'l Symp. Computer Architecture, pp. 278-287, May 1992.

[4] G.-M. Chiu, “The Odd-Even Turn Model for Adaptive Routing,” IEEE Transactions on Parallel and Distributed Systems, Vol. 11, No. 7, July 2000, Pages(s): 729-738.

[5] NoCsim, available at http://sourceforge.net/projects/nocsim.