DART: A Programmable Architecture for NoC Simulation on FPGAs

Danyao Wang, *Member, IEEE,* Charles Lo, *Member, IEEE* Jasmina Vasiljevic, *Member, IEEE*
Natalie Enright Jerger, *Member, IEEE* and J. Gregory Steffan, *Senior Member, IEEE*

***Abstract*—The increased demand for on-chip communication bandwidth as a result of the multi-core trend has made packet-switched *networks-on-chip* (NoCs) a more compelling choice for the communication backbone in next-generation systems [1]. However, NoC designs have many power, area, and performance trade-offs in topology, buffer sizes, routing algorithms and flow control mechanisms—hence the study of new NoC designs can be very time-intensive. To address these challenges, we propose DART, a fast and flexible FPGA-based NoC simulation architecture. Rather than laying the NoC out in hardware on the FPGA like previous approaches [2], [3], our design virtualizes the NoC by mapping its components to a generic NoC simulation engine, composed of a fully-connected collection of fundamental components (e.g., routers and flit queues). This approach has two main advantages: (i) since it is virtualized it can simulate any NoC; and (ii) any NoC can be mapped to the engine without rebuilding it, which can take significant time for a large FPGA design. We demonstrate (i) that an implementation of DART on a Virtex-II Pro FPGA can achieve over $100\times$ speedup over the cycle-based software simulator Booksim [4], while maintaining the same level of simulation accuracy, and (ii) that a more modern Virtex-6 FPGA can accommodate a 49-node DART implementation.**

***Index Terms*—Network-on-chip, simulation, FPGA.**

I. Introduction

As more cores are incorporated into a single chip, packet-switched networks-on-chip (NoCs) have emerged as a compelling replacement of traditional bus-based on-chip interconnects. NoCs provide higher overall bandwidth, more efficient use of shared on-chip resources, and a modular design that is easier to design, verify and fabricate. NoC designs are sensitive to many parameters such as topology, buffer sizes, routing algorithms, and flow control mechanisms. Hence, detailed NoC simulation is essential to accurate full-system evaluation.

To study the performance trade-offs of NoCs, software simulation is used widely, both as stand-alone network simulators [4], [5] and as the interconnect component of large full-system simulators [6], [7]. Software simulation has the advantages of being very flexible, easy to program, fast to compile, and deterministic (making it amenable to debugging). However, software simulation of large NoCs is slow, which adds to the already burdensome computation required to perform detailed full-system simulation. To maintain reasonable simulation times, the user may need to simulate at a higher level of abstraction. For example, instead of a cycle-accurate model of the router's microarchitecture, using a simple ideal switch that does not model the resource contention within the router reduces the amount of computation required for simulation.

Department of Electrical and Computer Engineering, University of Toronto, ON.
E-mail: {wangda,locharl1,vasiljev,enright,steffan}@eecg.toronto.edu

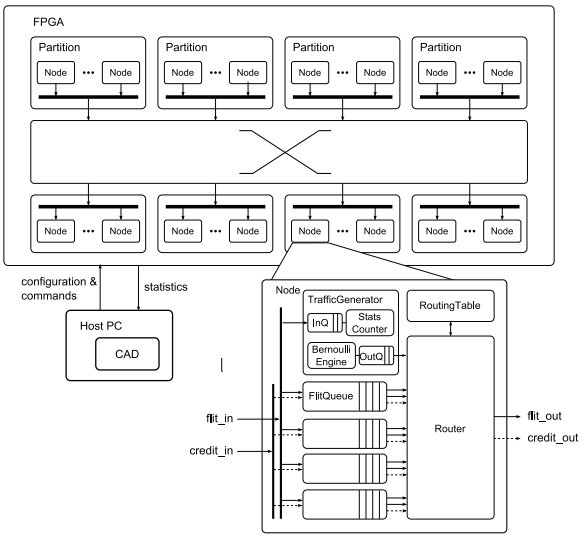


Fig. 1. DART Simulator architecture on the FPGA

The increased on-chip logic and memory capacities of FPGAs (Field Programmable Gate Arrays) allow an entire on-chip system to be prototyped or emulated on a single device. Several FPGA-based NoC emulators [2], [3], [8], [9] that reduce simulation time by several orders of magnitude have been proposed. These dramatic speedups are possible because the emulator is constructed by laying out the entire NoC on the FPGA, allowing the hardware to exploit all available fine and coarse grain parallelism between the emulated events in the NoC. However, this direct approach has three key drawbacks relative to software simulation: (i) any change in the simulated NoC requires manual redesign of the emulator HDL, (ii) redesign in turn requires complete compilation/synthesis of the FPGA design, which can take hours, or up to a day for a large design, and (iii) the maximum simulatable NoC size is determined by the FPGA capacity.

A. Flexible NoC Simulation Engine

We bridge the speed and usability gap between software and hardware approaches of NoC simulation by proposing a novel overlay architecture for FPGA-based NoC simulation. The simulator architecture, which we call *DART*, provides a

Digital Object Indentifier 10.1109/TC.2012.121

parametrized model of a generic NoC where the parameters can be set by software to simulate different NoCs without modifying the hardware simulator on the FPGA. Fig. 1 shows the organization of DART, which consists of a fully-connected collection of fixed-function components that model the building blocks of an NoC: traffic generators, routers and queues. Configurable parameters within each node allows behaviors of individual nodes to be altered to match nodes in the simulated NoC. The global interconnect provides all-to-all communication between DART nodes, thus allowing simulation of different topologies. In addition, the simulated time is decoupled from the FPGA cycles through the use of a global time counter. It is incremented once every simulated cycle after all network transfers for that cycle are simulated, which may take a variable number FPGA cycles. Virtualizing simulation time allows us to optimize the DART components for area efficiency. DART also supports virtualized router contexts to allow a larger number of simulated router nodes than physical nodes.

B. Contributions

This work makes the following contributions: (i) an abstraction model for NoC simulation on FPGAs and the demonstration of the feasibility of a software configurable FPGA-based NoC simulator with negligible area overhead relative to existing emulators that do not support configuration; (ii) evaluation of the implementation of a nine-node DART using a Xilinx Virtex-II FPGA, and a forty-nine-node DART using a Virtex-6 FPGA; (iii) a comparison of the performance of the DART simulator to the well-known software NoC simulator Booksim and the demonstration of over 100-fold speedup.

II. Related Work

A. Software NoC Simulators

Most NoC simulators are written in software so they are easy to develop and modify, and can be designed to be very accurate. Stand-alone network simulators such as Booksim [4] and SICOSYS [5] are used widely by researchers. Their modular design allows variations of network components such as routing algorithms and allocators to be easily incorporated. Both Booksim and SICOSYS use synthetic traffic, where packets are injected according to a random process. Synthetic traffic stresses network resources and provides a good estimate of the network's performance under worst-case traffic scenarios.

Full system simulators such as GEMS [10] and SimFlex [6] enable computer architects to study the interaction between processor architecture and other system components using real applications. They incorporate NoC simulators to model the communication fabric. Different simulators model the NoC at varying levels of detail. SimFlex has a simple network model that assumes perfect routers with infinite switching bandwidth and computes packet delay based solely on the topology of the network and the latency/bandwidth properties of the links. These simplified assumptions may lead to underestimates of network latencies when there is congestion. GEMS's Garnet [7] interconnect simulator provides a more accurate model of a classic five-stage pipelined router with virtual channels, and hence can provide better estimates of system performance.

Although software simulators offer many advantages, the most important being the ease to add and modify models, they can be slow. Typical simulation speeds range from the low KIPS (Thousands of Instructions per Second) to 100s of KIPS, depending on the detail level of the models [11]. Parallelizing the software simulators and leveraging modern multi-core processors to improve simulation speed is non-trivial, as NoC simulation is communication-intensive and requires fine-grained synchronization. Naïve parallelization can incur high synchronization cost. DARSIM [12] and HORNET [13] are parallel NoC simulators that achieve good scalability for up to four to six threads in cycle-accurate mode, which requires two global synchronizations per simulated cycle. Relaxing this constraint allows good scaling to eight to twelve threads at the cost of lower accuracy.

B. FPGA-based NoC Models

With advances in process technology, newer FPGAs offer enough logic and memory capacity to implement a complete digital system on a single chip. As a performance evaluation platform for NoCs, FPGAs have two advantages over software simulators. First, the fine-grained parallelism in NoC simulation can be exploited by a large collection of dedicated function units. Second, the high amount of communication that is expensive to implement in a coarse-grained thread model is easily accommodated by the abundance of wires available to connect functional units. As a result, FPGA-based NoC models can be orders of magnitude faster than software simulation.

Genko et al. [3] describe an emulation platform that consists of programmable traffic generators and receptors that drive a 6-switch NoC and is 2600-fold faster than a SystemC simulation of the same network. While this platform supports programmable traffic patterns and statistics counters, changing the configuration of the network requires re-synthesis of the emulator. DRNoC [2] circumvents this requirement by leveraging the partial reconfigurability of Xilinx FPGAs. The DRNoC host FPGA is divided into grids; each grid slot can be dynamically reconfigured to implement a new component to model different networks. However, partial reconfiguration requires a special design flow and incurs area overheads; it is also only available for select devices. In contrast, DART's configuration interface is based on a generic shift register and is portable to any FPGA.

NoCem [8] improves emulation density over Genko et al.'s design [3] and implements a 9-node mesh network on a single FPGA by eliding the router pipeline details and virtual channels. Instead of sacrificing these important details, we employ a simple design for each DART Router: each has multiple input ports but only one output port, and models the all-to-all switching in a simulated router by routing one input port per DART cycle.

All of the aforementioned NoC evaluation platforms do not distinguish between the FPGA-based evaluation architecture and the architecture of the modeled NoC. As a result, to study a different NoC, the emulator must be modified and the FPGA

synthesis-place-route steps are repeated. This process is labor-intensive and time consuming. Moreover, they do not allow emulation performance to be traded off for other important criteria such as the implementation area.

Wolkotte et al. [9] allow performance/area trade-offs by virtualizing a single router on an FPGA. An NoC with multiple routers is simulated by iterating multiple contexts through the router model. An off-chip ARM processor stores *N* contexts for the router model and orchestrates the emulation of the *N*-node network. This approach allows the router model to be much more detailed, although changing the router configuration still requires hardware changes. In addition, the off-chip ARM/FPGA communication link is a performance bottleneck. DART also supports virtualization of routers but virtualization is achieved without requiring expensive off-chip memory accesses.

Papamichael [14] proposes two FPGA-based NoC simulator designs: a direct-mapped and a virtualized design. The direct-mapped approach directly instantiates the desired NoC onto the FPGA; hence new NoC designs require new Verilog, in turn requiring re-synthesizing the design which is time-intensive for large FPGAs. DART avoids this overhead through a software reconfigurable substrate. The virtualized implementation uses a single virtualized router to simulate all routers in the network; all routers are simulated before proceeding to the next cycle. This high degree of virtualization allows very large router designs to be simulated in conjunction with very large network sizes which would be infeasible in the direct-mapped approach due to resource limitations on the FPGA. DART virtualization maintains multiple router instances each with multiple contexts; this allows us to simulate large systems while paying a smaller performance penalty.

DART provides a flexible FPGA-based NoC simulator platform by decoupling the simulator architecture from the architecture of the simulated system. This technique is borrowed from FPGA-based processor simulators [11], [15], [16], [17]. ProtoFlex [15] is a functional simulator of a multi-core processor. It implements a SPARC V9 processor pipeline on the FPGA and uses multithreading to simulate multiple processors. RAMPGold [16], [18] also uses a virtualized functional model to simulate multiple processors. In addition, it contains a timing model and adds the ability to configure some system parameters, such as the number of simulated cores, cache configuration and simulated DRAM timing characteristics, after the simulator is implemented on the FPGA. A-Ports [11] proposes a method to abstract any synchronous system into components connected by queues. Each queue has some latency and bandwidth, and models the timing of the component it connects to. HAsim [17] applies time-division multiplexing (TDM) to both the cores and the routers in the on-chip network; they note that applying TDM to the cores is more straightforward as each core is independent (e.g., core 0 cannot change the register file of core 1). NoCs represent a more challenging case as there are cross-router dependences. HAsim proposes a novel permutation technique to handle these dependences. While DART employs the same high level strategy as these simulators, NoC architectures are significantly different from processor pipelines. An NoC is a distributed system and has more communication. We design the DART architecture to efficiently and accurately capture the characteristics of an NoC. We describe the DART architecture in the next section.

TABLE I
DESCRIPTOR FORMATS

Flit descriptor format (36 bits)		
Bit Range	Width	Description
35	1	Head flit boolean flag
34	1	Tail flit boolean flag
33	1	Measurement flit boolean flag
32:23	10	Timestamp to forward to next hop
22:15	8	Destination node address
14:5	10	Source node address (if this is a head flit) or injection timestamp
4:2	3	Next hop port ID
1:0	2	Next hop virtual channel ID
Credit descriptor format (12 bits)		
Bit Range	Width	Description
11:10	2	Virtual channel ID
9:0	10	Timestamp to forward to next hop

III. DART ARCHITECTURE

The basis of the DART architecture is to provide programmability by decoupling (i) the simulator architecture from the architecture of the simulated NoC, and (ii) DART cycles from simulated cycles. To provide a configurable functional model for NoC simulation, we abstract common NoC functionalities into three basic components: *Traffic Generators* (TGs), *Flit Queues* (FQs) and *Routers*. Components can be mixed and matched to model more complex NoC nodes. A given topology is then simulated by configuring the Routers to only forward to their simulated neighbors via the global interconnect; this configuration does not require re-synthesis.

Traffic Model A typical NoC carries two types of traffic: *flits (flow control units)* that carry data messages and *credits* that are exchanged between neighboring routers to enforce flow control [4]. DART models flits and credits using *descriptors* that contain only the information necessary to forward them from source to destination. A description of the descriptor formats is given in Table I. A 36-bit flit descriptor encodes the injection and next-transfer timestamps, source and destination addresses, and boolean flags for the flit type (head, tail, and warmup). Warmup flits are used to bring the network to steady state and hence do not have their latencies recorded. Not encoding the data payload saves area as fewer bits are stored and passed between DART nodes. We choose 36 bits to match the port width of embedded RAM blocks on the FPGA, which are used to implement the flit buffers. Anything wider doubles the RAM usage as two RAM blocks must be used in parallel to support the data width. A credit descriptor encodes only a timestamp and a virtual channel ID.

Timing Model To capture the timing of flit transfers, we use a global time counter to synchronize all network events. Each flit contains a *timestamp* that indicates when the next

```
N_through ++
if(T_enqueue>T_last_flit || N_through>=bandwidth)
    T_dequeue = max(T_enqueue, T_last_flit+1)
    N_through = 1
else
    T_dequeue = T_enqueue
T_dequeue += latency
```

Fig. 2. Algorithm to calculate the dequeue timestamp in an FQ

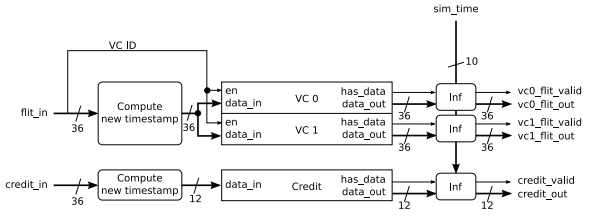


Fig. 3. Flit Queue datapath. The *Inf* block ensures that the flits at the head of the FIFOs are only dequeued according to their timestamps

transfer of this flit should happen. As a flit traverses the network, its timestamp is updated by intermediate DART nodes to reflect the delay due to pipeline latency and simulated contention. Credit transfers are timed similarly. Upon arrival at the destination TG, a flit's latency is computed by subtracting the injection timestamp from the arrival timestamp. The 10-bit timestamps allow DART to correctly compute latency provided a flit's latency does not exceed 1024 simulated cycles. We believe this is a reasonable compromise to keep the flit descriptors within 36 bits as most on-chip communication takes no more than a few hundred cycles. However, the maximum simulation length DART supports is not limited to 1024 cycles. By using signed subtractions to compare timestamps, we can correctly determine the chronological order of timestamps within 512 simulated cycles even when the global time counter wraps around. Since the DART design guarantees that timestamps of all flits traversing the global interconnect fall within a N-cycle window, where N is the simulated latency of the router pipeline and is smaller than 512, flits will always be delivered in correct simulation order.

Design Space Coverage The bit widths of the other descriptor fields are also chosen to be minimum size while still providing sufficient functionality coverage. The 8-bit node addresses, 3-bit port ID and 2-bit virtual channel (VC) ID allow DART to scale to 256 nodes, 8 ports per node and 4 virtual channels per port. Configurations that fit within these flit widths can be setup in software at run-time. These widths do not fundamentally limit the size NoC that DART can simulate; larger sizes can be accommodated through re-synthesis with only minor HDL changes.

A. Flit Queue (FQ)

The Flit Queue component models the VC buffers at a router's input port and the bandwidth/latency constraints of the wire link feeding the port. The buffers are independent FIFO (first-in-first-out) queues. They are implemented using a single block-RAM that is statically partitioned among the VCs. A Verilog parameter controls the number of VCs to incorporate (set to two in our current implementation).

TABLE II
PACKET DESCRIPTOR FORMAT (32 BITS)

Bit Range	Width	Description
31	1	Measurement packet boolean flag
30:21	10	Injection timestamp
20:19	2	Virtual channel ID
18:16	3	log_2(packet size) - 1
15:8	8	Destination node address
7:0	8	Source node address

Figure 3 shows the FQ datapath. For each incoming flit, the FQ computes the new dequeue timestamp to properly reflect the delay it experiences traversing the link due to latency and bandwidth constraints. The algorithm to compute the new timestamp is shown in Figure 2. Here $N_{through}$ counts the number of flits through the FQ during a simulated cycle. $T_{lastflit}$ is the dequeue timestamp of the previous flit less the link latency. Both the latency and bandwidth parameters are configurable per FQ.

After the timestamp is updated, the flit is queued according to its VC. It is forwarded to the next-hop Router when it gets to the front of the FIFO and the global simulation time is equal to its timestamp. This ensures all flits arrive at a Router in chronological order, which is required for correct simulation of resource contention at the routers. A separate FIFO is used for the credit channel. Similar to the flits, a credit can leave an FQ only during its scheduled dequeue time.

B. Traffic Generator (TG)

When enabled, a Traffic Generator injects traffic in one of two modes: synthetic or dynamic. The former is useful for stress testing the simulated network. The latter provides an interface to incorporate DART into a full-system evaluation framework. The mode is configurable per TG.

In synthetic mode, a TG injects flits in bursts of fixed sized packets using a Bernoulli process. Packet size (minimum 2 flits), destination node address, and injection interval are configurable per TG.

In dynamic mode, a TG receives packet descriptors (Table II) from the host PC and injects packets according to the descriptors. Packet size can be varied between 2 and 256 flits in powers of 2. Packet descriptors can be generated from either a memory access trace or a processor simulator running concurrently with DART. Due to the required high volume of packets, the dynamic traffic is delivered onto the FPGA via a designated high-bandwidth link.

The traffic delivery path is implemented as a carry-chain which reaches all of the TGs in order. The packets are sent serially while each TG snoops the traffic path, and grabs its corresponding packet. The carry-chain implementation approach increases latency between the host-PC and the nodes. An alternative approach is to implement a low-latency, high fan-out mux distribution to all the TGs. Considering the large bus width required for each packet, this approach exhibits a significantly lower f_{max} and requires significantly more FPGA resources. This approach is discarded so as not to limit the maximum number of implementable nodes within DART.

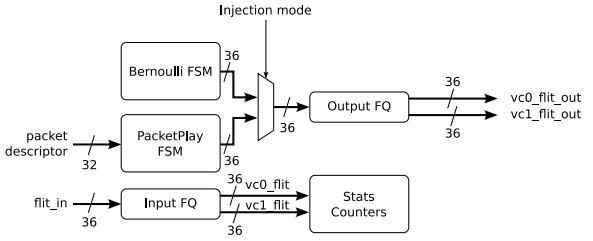


Fig. 4. Traffic Generator datapath

Each node has a small local FIFO holding its traffic data and feeding it to the PacketPlayFSM. The local FIFOs are necessary to prevent possible starvation of packets which can arise due to the carry-chain implementation and a particular order of packets in the traffic stream. We avoid this by allowing the FIFOs to store packets ahead of the current simulated cycle and ensure that starvation does not occur. If packets are delivered in order of the simulation cycle, then the maximum distance between two consecutive packets targeting the same node is the total number of nodes in the system. For this reason, the minimum FIFO depth is the total number of nodes, rounded up to a multiple of 2 (due to the use of BRAMs).

Figure 4 shows the TG datapath. The Bernoulli FSM and PacketPlay FSM handle the traffic injection of the synthetic and dynamic injection modes respectively. In addition, each TG also contains two FQs: the *input buffer* models the last-hop delay to the TG, and the *output buffer* models the source queue. We use the same technique from Dally and Towles [4] and allow the injection state machine to lag behind the current simulation time when the output buffer is full, to model an infinite source queue. TGs also serve as traffic sinks and record the number of packets received and the cumulative packet latency. More statistics counters can be easily added.

C. Router

State-of-the-art NoCs use the classic wormhole VC router, which is composed of per-VC flit buffers, routing logic, VC and switch allocators and a crossbar. Since the FQs model the flit buffers, the Router component only encapsulates the routing and allocation logic. Figure 6 shows the Router datapath. The number of ports is set to five in our current implementation, but can be changed by setting a Verilog parameter. We use table-based routing. Hence any deterministic routing algorithm can be implemented. The table contents are configurable without reprogramming the FPGA. The configuration of the routing table also facilitates the simulation of a wide range of topologies.

A 4-bit counter for each output VC is used to implement credit-based flow control. Initial credit values represent the number of entries in the input buffer at the downstream router. The counter is decremented when a flit is routed, and incremented when a credit is received. The values are configurable for each VC and Router.

1) Area-Speed Trade-off: The allocators and the crossbar in the classic router are complex structures [19]. A direct implementation is too area-consuming. Instead, the DART Router employs simple arbiters and a multiplexer to implement the same functionality by trading off simulation speed. Table III outlines the implementation costs of the allocators and crossbar in the classic router and their equivalents in the DART Router. A two-stage VC allocator (Figure 5a), which consists of $p_i v$ v-to-1 arbiters in the first stage and $p_o v$ $p_i v$-to-1 arbiters in the second stage is required because we allow flits to allocate any free VC once the output port is determined by the routing function. Figure 5b shows the DART Router VC allocator for comparison. DART's equivalent to the classic router's allocators and crossbar are much smaller.

TABLE III
ALLOCATOR AND CROSSBAR IMPLEMENTATION COST IN TERMS OF THE BASIC BUILDING BLOCKS REQUIRED IN THE CLASSIC ROUTER AND IN THE DART ROUTER (p_i IS THE NUMBER OF INPUT PORTS. p_o IS THE NUMBER OF OUTPUT PORTS. v IS THE NUMBER OF VCS PER PORT.)

Structure	Classic Router	DART Router
VC Allocator	$p_i v$ v-to-1 arbiters and $p_o v$ $p_i v$-to-1 arbiters	1 v-to-1 arbiter and 1 v-bit p_o-to-1 MUX
Switch Allocator	p_i v-to-1 arbiters and p_o p_i-to-1 arbiters	1 $p_i v$-to-1 arbiter
Crossbar Switch	$p_o v$ $p_i v$-to-1 MUXes	1 $p_i v$-to-1 MUX

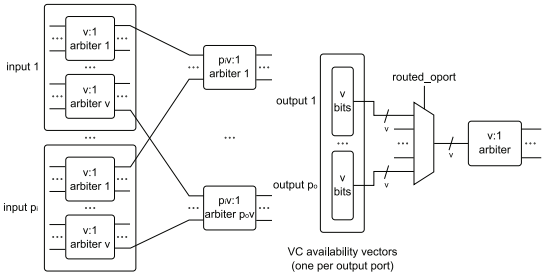


Fig. 5. VC allocator implementation: (a) classic router, (b) DART Router

As a result of this model simplification, the Router component can only route one flit per DART cycle. To model a 5-ported classic router, the input VCs are routed one at a time while the global time counter is stalled so all input ports appear to be routed in the same simulated cycle. A round-robin scheme selects an input VC to route in each DART cycle. If a flit cannot be routed due to failed VC allocation or lack of credits for the requested output VC, it remains in the FQ. It is considered again in the next simulated cycle. For every simulated cycle that a flit is unable to route, its timestamp is incremented to reflect the contention delay. When a flit is finally routed, its timestamp is incremented by a fixed pipeline latency. This pipeline latency is configurable per Router.

2) Send-Ahead Optimization: Once a flit is routed, it waits in the Router output queue. It is forwarded to the next-hop FQ at its scheduled dequeue time. The Router stalls when the output queue is full. If the queue is not large enough to cover the pipeline latency, deadlock may arise when global time is stalled because there are unprocessed input ports, but the output queue is not drained as the head flit waits for its dequeue time. To keep the output queue size small while avoiding deadlock, we let flits in the output queue proceed immediately to the global interconnect so the output queue can make forward progress even when global time stalls. The

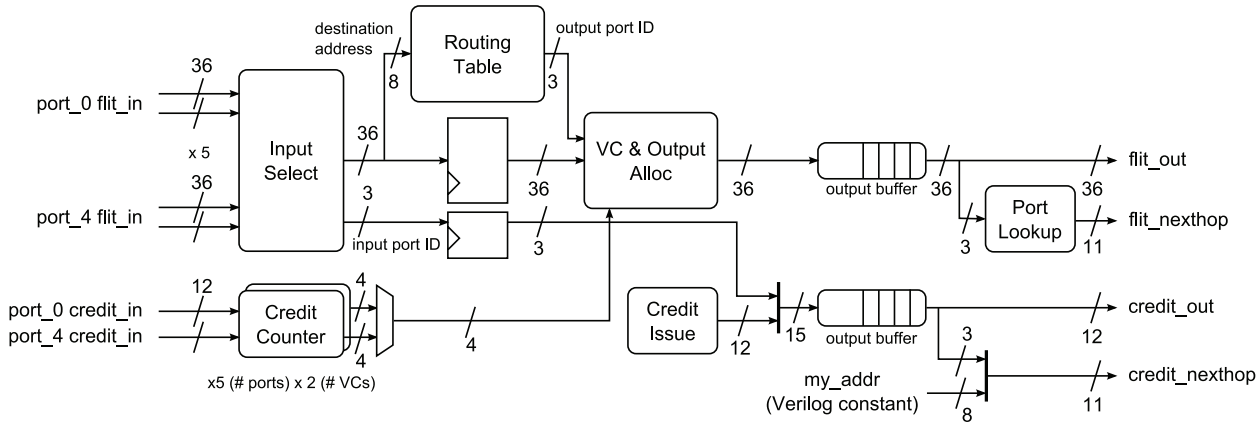


Fig. 6. DART Router datapath

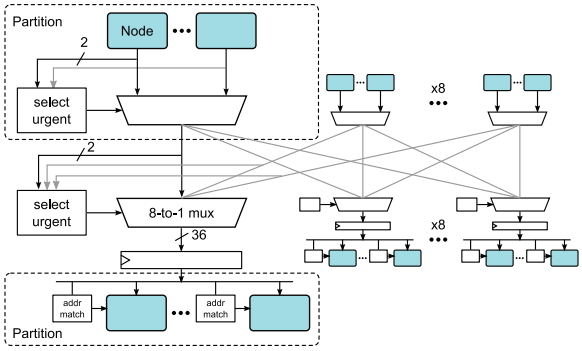


Fig. 7. DART's global interconnect. Nodes are grouped into partitions so the size of the crossbar needed is small. The source partitions are shown at the top and the destination partitions are shown at the bottom.

flits still arrive at the receiving FQ in order because they leave the Router in chronological order, despite being sent ahead. Sending ahead also avoids bursty requests to the global interconnect in the beginning of every simulated cycle. Overall interconnect utilization is improved.

D. Global Interconnect

The global interconnect provides uniform-latency communication between all DART nodes. By configuring the routing tables appropriately, DART can simulate any topology. The maximum node radix is limited by the number of ports configured in the Router components. Figure 7 shows the interconnect organization. Each DART node consists of one TG, one Router, and four FQs to implement a 5-ported wormhole router. The number of FQs to incorporate is controlled by a Verilog parameter. We choose not to use a full crossbar for the global interconnect because crossbar area increases quadratically with the number of input and output ports. It will not scale as we incorporate more nodes into DART. Instead, DART nodes are grouped into partitions and the partitions are connected by a small crossbar. A separate, narrower but otherwise identical interconnect is used to carry the credit traffic.

A flit crosses the global interconnect in two stages. First, it arbitrates for the source partition output, which is connected to a crossbar input port. Upon winning the arbitration, the flit arbitrates for the desired crossbar output at the destination partition. Both intra- and inter-partition arbitrations use simple round-robin arbiters, with priority given to flits with timestamps equal to the current simulation time (*urgent flits*). These flits must be forwarded first before ticking the global time counter to prevent late flits, which may cause out-of-order flits at the next Router. Flits with a timestamp ahead of the current simulation time (*future flits*) can be forwarded out-of-order across the global interconnect because flits destined for each FQ remain in order. Within each source and destination partition, the priority is implemented by having two separate arbiters for the urgent flits and future flits. The result of the urgent arbiter always takes precedence. When there are no valid requests for the urgent arbiter, it indicates that all urgent flits have crossed the interconnect. The global time counter is then incremented. Because it takes a cycle to detect this condition, each simulated cycle takes at least 2 DART cycles. The global interconnect is on the timing critical path. Hence, it is pipelined to improve maximum achievable system clock frequency (f_{max}) during implementation. The pipeline registers are inserted after the source partitions.

The partitions are the throughput bottleneck because only one flit can be sent and received by a partition per DART cycle. For a fixed number of DART nodes, varying the size of the partition trades off the global interconnect throughput for implementation area. We discuss this in more detail in Section V-D. For our current 9-node implementation, we use 8 partitions connected by an 8×8 crossbar. In general, the largest crossbar that fits in the device once the nodes are implemented should be chosen.

E. Virtualization

A given FPGA can only support a limited number of physical NoC nodes, limiting the size of the simulated NoC. To allow the simulation of a NoC with more simulated nodes than physical, DART supports node virtualization such that several simulated NoC nodes can be represented using multiple *contexts* on a physical DART node. The concept of applying virtualization to FPGA simulators has been used in previous approaches [14], [17]; in this section, we describe how to

apply virtualization specifically to DART and novel techniques used to improve DART's virtualized performance. In Section V-E we explore the impact of virtualization on DART. Each additional context uses significantly less resources than would additional full DART nodes; however, the operation of virtual contexts is necessarily serialized which reduces the performance of the system.

To minimize the performance impact of this serialization, the different DART components are able to switch between contexts independently. Within the router, a rotating priority encoder is used to select between contexts with valid flits. Likewise, an arbiter that favors urgent flits is used to select between valid contexts at the source partition of the global interconnect. However, the synthetic traffic generator, due to its random nature, must touch all contexts during each simulation cycle to test for flit injection. To accommodate the extra logic delay associated with context selection, additional pipeline stages were added to the router and traffic generator.

The primary incremental area cost of adding virtual contexts to DART nodes is the memory required to store the extra state information. Since the operation of virtual contexts is serialized, few read and write ports are required for these memory structures. To maintain high memory density, block-RAMs were used for flit queue and routing table storage while distributed RAMs were used to hold other state information.

To ensure correct ordering among packets between virtualized routers, Papamichael [14] implements double-buffering. This double buffering isolates events belonging to different target cycles. HAsim [17] sets the simulation order of nodes to ensure proper routing of data between them. In contrast, correct ordering of packets in virtualized DART is maintained naturally using the same timestamp mechanism as non-virtualized DART. For correctness, our virtualized DART architecture must ensure that all pending actions for virtual contexts in the current simulation cycle are handled as described previously.

F. Configuration and Data Collection

As described in previous sections, each DART node is highly configurable. Table IV lists the parameters by node type. With the exception of the routing table, the parameters are chained in a 16-bit shift register. The configuration byte-stream is received from the host PC and shifted into the chain. The RAM-based routing tables are connected to the input end of the shift register. Each table has a finite state machine that captures a segment of the configuration bits to populate the table. An enable signal is asserted to start the simulation when configuration completes.

Similar to the configuration registers, performance counters are read back by shifting them through a 16-bit-wide chain. Currently three counters are incorporated per TG to record the number of injected and received packets (32 bits) and the cumulative packet latency (64 bits). More counters can be easily added to this shift register chain.

G. Software Tools

The DART software tools run on a host PC connected to the FPGA where the hardware simulator resides. They allow the dynamic reprogramming of the hardware simulator after it is implemented on the FPGA. Fig. 8 shows the software flow. The components are described in this section.

TABLE IV
CONFIGURABLE PARAMETERS IN DART NODES

Node	FF-based	RAM-based
Traffic Gen	Destination node address Bernoulli threshold Taus RNG seeds Packet size (in flits)	
Flit Queue	Latency Bandwidth	
Router	Initial credit counts Pipeline latency	Routing table

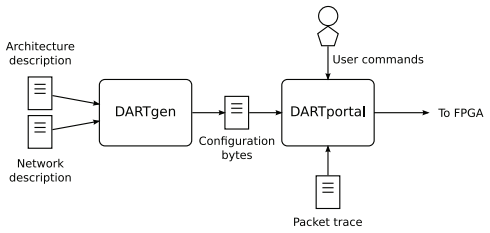


Fig. 8. DART software flow. DARTgen creates the configuration byte-stream from user specifications. DARTportal provides an interactive interface for the user to control the hardware simulator.

```
SourcePartitions 8
DestPartitions 8

SwitchPorts 5
NumVCs 2

SP 0 DP 0 NumNodes 2
SP 1 DP 1 NumNodes 1
SP 2 DP 2 NumNodes 1
SP 3 DP 3 NumNodes 1
SP 4 DP 4 NumNodes 1
SP 5 DP 5 NumNodes 1
SP 6 DP 6 NumNodes 1
SP 7 DP 7 NumNodes 1
```

Fig. 9. Architecture description file for a 9-node DART simulator

1) Configuration Generation: DARTgen creates the configuration byte-stream from an Architecture Description File (ADF) and a Network Description File (NDF)—an excerpt of the ADF is shown in Fig. 9. The ADF specifies the on-chip architecture by listing the number of partitions on the interconnect and the number of nodes within each partition. Each partition is identified by its source partition ID (SP) and destination partition ID (DP). The NDF describes the user network to simulate, including the topology, routing tables, traffic pattern, and properties of individual links and routers.

The DARTgen flow consists of three stages: network extraction, placement, and byte-stream generation. In network extraction, DARTgen constructs a graph of the user network from the NDF. Each vertex represent a router and its host node. The edges represent the links between routers. The vertices and edges are annotated with properties specified in the NDF. A model of the on-chip partition hierarchy is also constructed from the ADF.

In placement, vertices and edges are mapped to Routers

and Flit Queues, with the host node in each vertex mapped to the Traffic Generator connected to the corresponding Router. We use a round-robin scheme to balance the number of used nodes across different partitions. This provides sufficient load balancing because the on-chip communication bottleneck is within each partition, where the nodes contend for the shared access to the inter-partition crossbar. Intra-partition contention can be further reduced by grouping neighbors of a common node in the user network into one partition. Because the common node can only send to one member of the group in a simulated cycle, there is less competition within the group to use the crossbar output.

Finally in byte-stream generation, the configuration bytes for the architecture nodes are printed to a file in the order that they are connected in the configuration chain on-chip. This file is used by the front-end tool DARTportal to program the on-chip simulator.

2) Front-end Interface: DARTportal provides a command-based interactive interface to configure, run and collect data from the simulator. It is implemented in two layers for portability. The top layer implements all functionalities exposed by the command protocol of the DART off-chip interface. The bottom layer contains driver code to directly communicate with the physical interface used to connect to the FPGA. Only the bottom layer needs to be modified when a different physical interface is used for DART's off-chip interface.

IV. Implementation

To demonstrate the functionality of DART and to obtain real measurement of simulation speed, we implement a 9-node DART on a Xilinx University Program Virtex-II Pro Development System [20]; note that the Virtex-II FPGA uses 4-input Look-Up Tables (LUTs) as programmable logic. In addition, we implement a scaled 49-node DART on a Virtex-6 FPGA using a ML605 Development System (ML605) [21]. The newer and larger Virtex-6 FPGA uses 6-input LUTs as programmable logic—consequently, the scaled 49-node results are measured in terms of 6-LUTs, and the *correctness* results are shown using a 4-LUT-based device. We design the DART components in Verilog HDL, and use the Xilinx ISE 12.3 software suite for synthesis and implementation. Device-specific constructs are avoided whenever possible so that the simulator core can be implemented on different FPGA systems with minimal changes.

A. 49-node DART on a Virtex-6

The ML605 platform contains a Virtex-6 XC6VLX240T FPGA that has 37,680 slices and 416 embedded RAM blocks. Each block-RAM has a capacity of 36Kb and has two read-/write ports each with maximum width of 36 bits. Table V shows the resource breakdown of the DART components as implemented on the ML605 platform. Because every two Routers share a dual-ported routing table implemented using a dual-port block-RAM, each Router uses 0.5 block-RAMs on average. The maximum number of DART nodes that fit on this FPGA is 49. Each node consists of one Traffic Generator (TG), one Router with 5 ports, and four Flit Queues (FQs) with two VCs each. We use 16 partitions in the global interconnect, which is the largest that fits on the FPGA. The final implementation runs at 50MHz.

TABLE V
Resource utilization breakdown of a 49-node DART on a Virtex-6 FPGA

Module	6-LUTs	Registers	% of Total (Slices)
Traffic Generator	22,981	10,535	25%
Flit Queue	6,272	2,156	15%
Router	9,114	3,087	15%
Global Interconnect	2,096	3,776	7%
Control Unit	88	56	0.1%
PCIexpress Core	1,236	132	1%

TABLE VI
DART Scalability and Resource Utilization on a Virtex-6 FPGA

DART Size	Partition Size (SPxDP)	6-LUTs	Registers	Block RAMs	% of Total (Slices)
9 node	4x4	10,630	6,850	12	9%
9 node	8x8	23,061	14,104	22	22%
16 node	4x4	37,265	22,400	36	34%
16 node	8x8	38,209	22,973	36	35%
16 node	16x16	42,912	24,213	36	38%
25 node	4x4	56,124	33,854	54	49%
25 node	8x8	56,838	34,459	54	51%
25 node	16x16	62,204	35,723	54	55%
36 node	4x4	79,123	47,856	76	69%
36 node	8x8	80,366	48,436	76	70%
36 node	16x16	84,915	49,552	76	72%
49 node	8x8	108,166	64,957	102	87%
49 node	16x16	112,434	66,184	102	89%
Total Available		150,720	301,440	416	100%

1) RAM Optimization: DART uses FIFO buffers extensively. FIFOs can be implemented either in block-RAMs or LUT-based shift registers on an FPGA. When implemented in a RAM, a FIFO consumes both ports of the RAM block to allow simultaneous read and write operations. In general, implementations which approach the upper limit of an FPGA's resource availability, such as logic cells and block RAMs, become increasingly harder to implement due to placement and routing restrictions imposed on the compiling tools. As a result, large designs typically require manual customization to successfully fit onto the FPGA.

In DART's case, the customization was three-fold. First, because a FQ can receive at most one flit and send one flit per DART cycle, the VC buffers in the FQ can share a single statically partitioned RAM block for storage. The FIFO output registers and control logic must be replicated. Second, some FIFOs were mapped onto block RAMs, while others onto distributed logic with the attempt to reduce the burden of the place and route step in the Xilinx ISE tools. Third, we implement the shallow output buffers in the Routers using LUT-based SRL16 shift registers [22]. Further, because routing tables are read-only during simulation, we can pack two into each dual-port block-RAM to utilize both ports. One of the ports is configured to be read/write and is used to configure the table contents.

TABLE VII
3×3 MESH BENCHMARK CONFIGURATION PARAMETERS

Topology	3 × 3 mesh
Link latency	1 flit cycle
Router architecture	Input queue
Routing algorithm	Dimension-order (XY)
# of VCs per port	2
VC Allocation	Round-robin
Input VC buffer size	5
Router pipeline latency	5 flit cycle
Traffic pattern	Permutation traffic
Packet size	2 flits

B. Scalability

DART's design allows for a convenient scalability depending on the amount of available resources on the FPGA accessible to the user. For example, the Virtex-II FPGA on the XUPV2P development board can fit a 9-node simulator, while a Virtex-6 FPGA on the ML605 development board can accommodate a 49-node simulator. The paths connecting the nodes are pipelined in a way as to minimize their impact on degrading scalability. For example, the global interconnect, the configuration carry-chain, and the dynamic traffic carry-chain are all pipelined. As a result, the critical path lies within the node itself, thus accommodating larger simulator sizes. Consequently, as DART scales the f_{max} remains at 50 MHz. Table VI shows the resource breakdown of the scalable DART simulator on the ML605 platform.

V. Analysis

In this section we validate DART's simulation results using Booksim as a reference. Because Booksim is widely used among NoC researchers, we hope this choice of baseline provides more confidence in DART's correctness and performance potential. We measure DART's speedup over Booksim using our Virtex-II Pro implementation. We also investigate the performance cost of a programmable simulator architecture and DART's scalability.

A. Correctness

We developed a cycle-accurate DART architecture simulator in C++ prior to building the FPGA architecture to explore different design options and to verify the correctness against Booksim. The architecture simulator also serves as the design specification during hardware implementation. Results shown in this section are obtained from this architecture simulator.

We simulate a 9-node mesh network and compare the measured average latency reported by Booksim and DART (Fig. 10). Table VII shows the parameters of the simulated network. To investigate the accuracy loss DART incurs by not modeling the delay through each stage separately, we simulate two router configurations in Booksim: *booksim* has a 5-cycle routing delay and zero switch and VC allocation delay, while *booksim2* has a 4-cycle routing delay, 1-cycle switch allocation delay and zero VC allocation delay. We simulate 15,000 warm-up cycles, 30,000 measurement cycles and a draining phase. The flit injection rate is varied from 0.01 until saturation. DART tracks Booksim closely at low injection rates. At higher injection rates, the one-stage pipeline in the Router results in a less accurate latency measurement. This is evident in that DART latency is enveloped by the two Booksim configurations that have the same overall router latency but different latencies at each stage. To further investigate the mismatch, Fig. 11 shows the distribution of packet latencies at 0.4 flits per cycle. The peaks at 8, 14, 20, 26, and 32 correspond to the zero-load latencies for 0, 1, 2, 3, and 4-hop paths. The lower peaks reflect the queuing delay and resource contention packets experience at the routers. Booksim has a much longer tail than DART. Because all contentions (buffer, VC, and switch) are modeled in one stage in the DART Router, DART may under predict the latency for a flit to acquire all resources. However, the similar overall shapes of the two distributions increases our confidence that DART produces useful predictions of network performance trends.

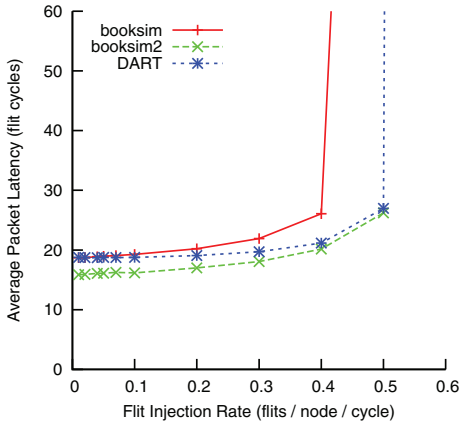


Fig. 10. Average packet latency for Booksim and DART for a 3×3 mesh

B. Speedup vs. Software Simulation

In Fig. 12 we evaluate the 3×3 mesh benchmark described in Table VII on the XUPV2P DART implementation and compare the simulation speed to Booksim. For the Booksim baseline, we measure the execution time of the main loop, excluding network setup, on a 2.66 GHz Core 2 Quad Linux workstation. Each data point is an average over 20 runs. We measure DART's execution time in DARTportal from the sending of the "Run" command and until the end-of-simulation signal is received back from the simulator. Configuration time is excluded. The speedup is the ratio of the number of cycles simulated per second in DART to that in Booksim. We observe that Booksim's simulation speed decreases with increasing injection rate. DART's speed is roughly constant, with all measured run-times falling within 3% (0.528 ms) of the average (20.5 ms for 50,000 cycles)—this is because DART's execution time increases slowly with traffic, and is largely masked by the high IO overhead to send and read-back commands to/from the simulator, which accounts for over 50% of the measured time (about 11.7 ms). As a result, DART achieves greater speedup at higher packet injection rates. The IO overhead can also be amortized in longer-running simulations of networks larger than the 3×3 mesh used here.

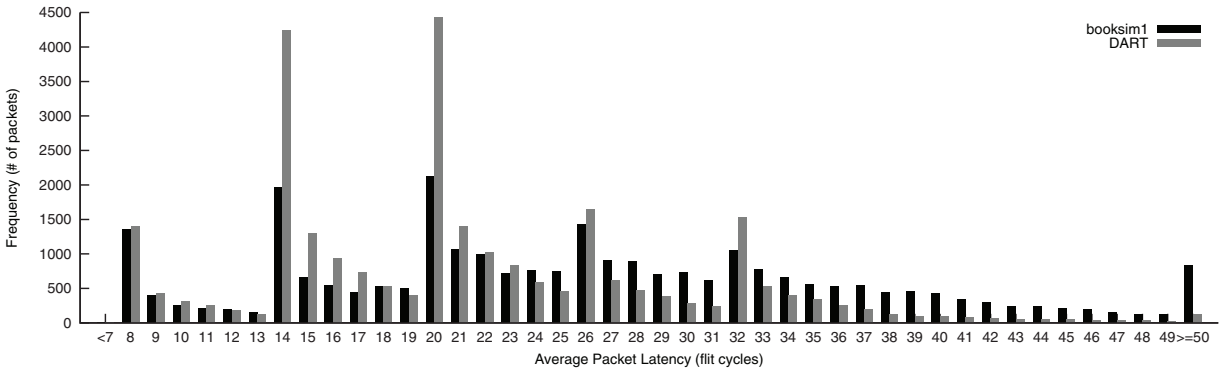
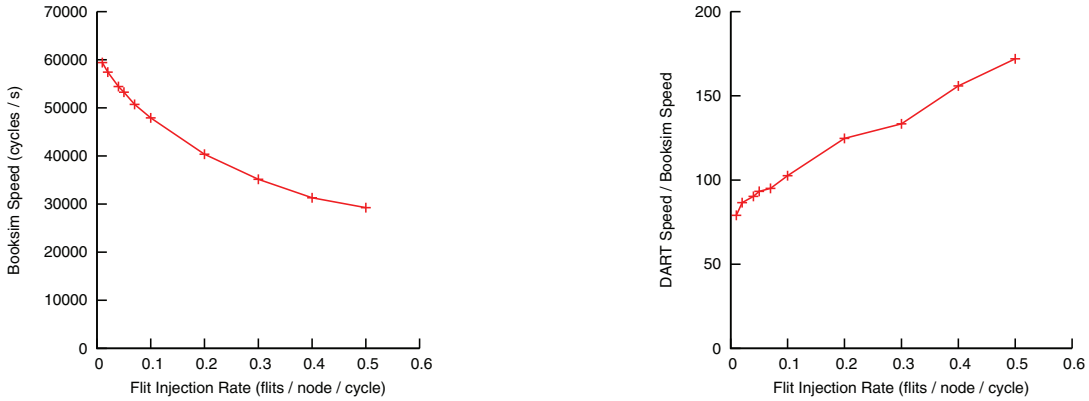


Fig. 11. 3×3 Mesh. Packet latency distribution measured by Booksim (booksim1) and DART at flit injection rate = 0.4



(a) (b)

Fig. 12. 3×3 mesh DART performance: (a) Booksim simulation speed, (b) Speedup achieved by DART vs. Booksim

C. Cost of Programmability

The main alternative to DART's programmable architecture is one that is directly laid-out in the FPGA fabric. In this section we measure the performance cost of DART's programmability by measuring the overhead (extra cycles required) of DART's global interconnect and the simplified Router model, relative to a model with a dedicated interconnect and full routers. As shown in Fig. 13, we measure for a 9-node and a 64-node DART all combinations of the two types of interconnect and two types of router:

- **dedicated**: Baseline interconnect with dedicated links between connected ports on neighboring nodes
- **global**: DART interconnect with 8 partitions
- **5port**: True 5-ported router
- **1port**: DART Router that routes 1 flit per DART cycle

Fig. 13a shows the number of DART cycles required per simulated cycle for the 3×3 mesh benchmark from Table VII. The baseline (dedicated+5port) has a constant cycles per second (CPS) of 1 as it corresponds to a direct mapping of the 9-node mesh NoC. Global+5port shows the performance loss due to the DART interconnect. The timer increment bubble, described in Section III-D, limits the minimum CPS to 2. Increased traffic causes more contention over the interconnect and lower CPS. Dedicated+1port shows the performance loss due to the serial processing of input VCs in the Router. CPS increase with network traffic, as each Router has more input VCs in use. Global+1port shows that for 9 nodes, because of the small number of nodes and low throughput of the Router, the global interconnect is not the performance bottleneck. However, Fig. 13b shows that with more nodes, contention increases for the global interconnect and it can become the bottleneck. An appropriate interconnect size should be chosen for each DART implementation. DART's interconnect uses more area than dedicated links, but the overhead is compensated for by the simplified Router. Thus, the overall area cost is comparable to published results from existing direct mapped emulators [3], [8]. We believe the performance penalty is a worthwhile trade-off for the ability to reconfigure the simulator at run-time without any hardware modification.

D. Scalability

We explore the scalability of DART beyond 9 nodes on a larger FPGA using the architecture simulator. The predicted runtime does not include communication overhead to and from the host PC.

1) Performance Scaling: Fig. 14 highlights the different scaling trends of Booksim and DART for four mesh networks of different sizes. The aggregated flit transfers per simulated cycle is the product of the flit rate, average number of hops between source and destination pairs and number of nodes. It measures the overall amount of in-flight traffic that traverses the network every cycle. Booksim's simulation speed depends on both the size of the simulated network and the

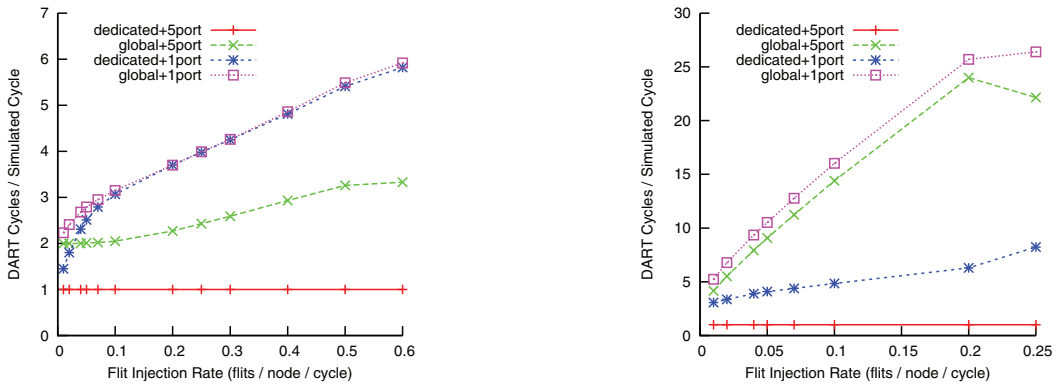


Fig. 13. Overhead of the DART interconnect and simplified Router model for a (a) 9-node and (b) 64-node DART. 3×3 and 8×8 mesh with random permutation traffic simulated.

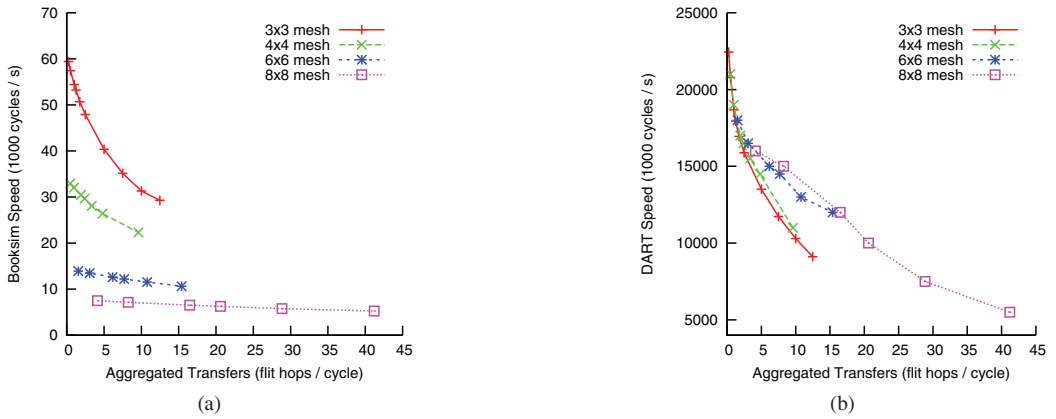


Fig. 14. Booksim (a) and DART (b) simulation speed for different network sizes

amount of network activity because it must simulate every cycle including those when the simulated network is idle. This overhead dominates simulation time for large networks and the network activity becomes an insignificant factor for performance. DART's simulated time advances faster when the simulated network is idle. Its simulation speed thus depends only on the amount of network activity. As a result, DART's speedup over Booksim varies from 300× for the 3×3 mesh to 2000× for the 8×8 mesh. These estimates are higher than the measured speedup from Section V-B due to the overhead of sending commands to the FPGA. In long-running simulations, this overhead can be amortized. The design focus for DART is on improving area efficiency so more simulator nodes can be implemented on a given FPGA.

2) Performance vs. Resource Utilization: The global interconnect is the main performance bottleneck in the DART architecture because nodes within the same partition compete for access to the input and output ports on the inter-partition crossbar. Fig. 16 shows the performance impact of various crossbar configurations for different DART sizes, where *SPx_DPy* denotes a global interconnect with an $x \times y$ crossbar. Each data point is evaluated by simulating a torus network using the DART global interconnect. We use permutation traffic for each simulated network. The results for

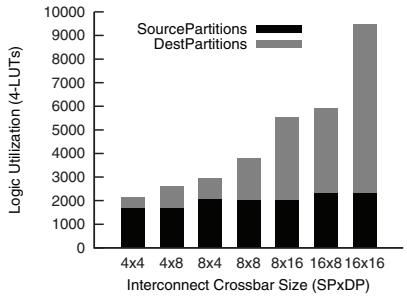


Fig. 15. Resource utilization of different interconnect sizes

the SP4_DP8 and SP8_DP16 configurations are not shown here because their performance is similar to that of the SP8_DP4 and SP16_DP8 configurations respectively. Relative to a square crossbar, doubling the number of either the input ports or output ports only improves performance slightly as the asymmetric configurations do not fully remove the contention within partitions.

Fig. 15 shows the total resource utilization of the global interconnect for a 64-node DART. The source partitions encapsulate the arbitration logic to allow sharing of the inter-partition crossbar inputs; the area of each source partition grows roughly linearly with the number of nodes in the

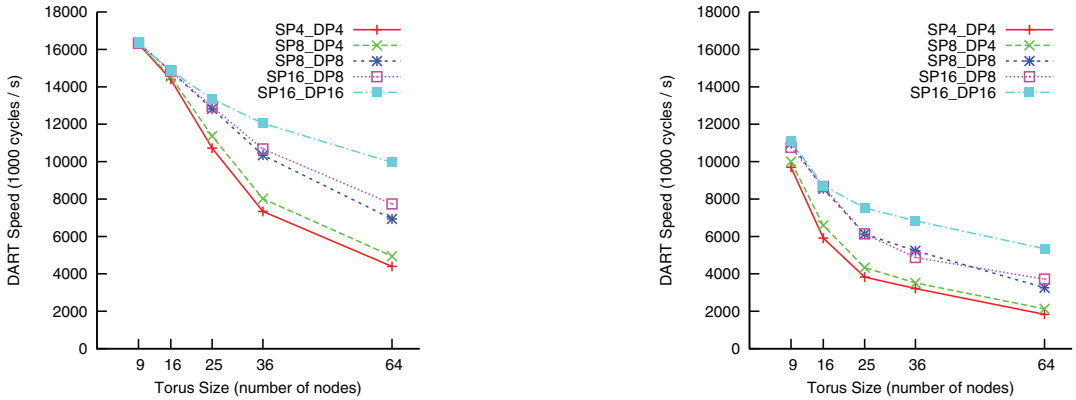


Fig. 16. Performance impact of global interconnect sizes, evaluated using torus networks (a) flit injection rate = 0.1 (b) flit injection rate = 0.5

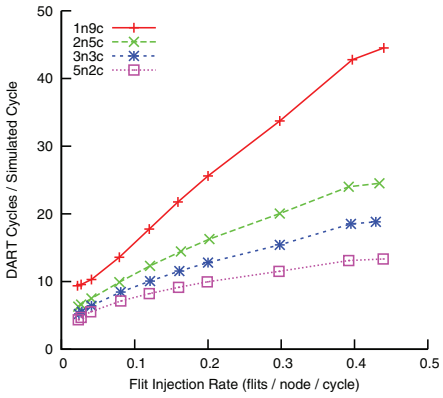


Fig. 17. Performance impact of virtualizing nodes in a single partition, 9-node system. The different configurations are written as xnyc where x is the number of DART nodes instantiated and y is the number of contexts available to each node.

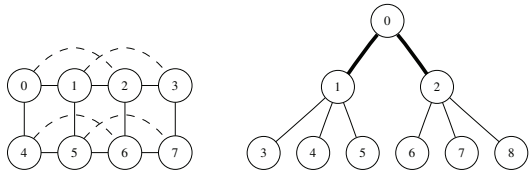


Fig. 18. Two microbenchmarks: (a) 4×2 mesh with express links, and (b) 2-level tree

partition. As a result, for a fixed number of DART nodes, the aggregated resources used by the source partitions are roughly constant, irrespective of the number of source partitions. The destination partition contains the multiplexers that implement the crossbar and the broadcast logic that enable the sharing of the inter-partition crossbar outputs, and it grows linearly to the number of source partitions. For the range of interconnect sizes considered here, because the source partitions start off as a significant portion of the total area, the overall LUT usage of the interconnect grows roughly linearly with the number of ports. For best performance, the largest square crossbar that meets the area constraint should always be used.

E. Virtualization

Virtualization presents another option for trading off performance and resource utilization. Since context information is densely packed on to the FPGA memory structures, each additional context requires much fewer LUTs relative to a fully replicated DART node. Furthermore, block-RAMs are only replicated once context storage exceeds the maximum RAM depth instead of for every DART node. Thus while the ML605 platform described in Section IV-A supports up to a 49-node DART configuration, much larger networks may be simulated by using one DART node per partition with multiple virtual contexts. In particular, 64-node and 81-node simulators may be configured running at 50 MHz and 46 Mhz respectively on the ML605; these configurations are not possible without virtualization.

The penalty for these area benefits comes in the form of serialized operation of the contexts. Fig. 17 demonstrates the performance of a single partition DART system, simulating a 3×3 mesh, as physical nodes are replaced with virtual contexts. At low injection rates, the minimum number of DART cycles is limited by the synthetic traffic generator since it must examine each context in turn during every simulation cycle. Performance is further reduced from a non-virtualized system due to the extra pipeline stages in the router and source partition required to support context selection. As the injection rate increases, the serialization penalty becomes apparent with performance decreasing roughly proportional to the number of contexts used. Despite the loss in simulator performance, virtualization provides the flexibility for DART to simulate large networks when working with limited hardware resources.

F. Case Studies

We choose two examples (Fig. 18) to demonstrate DART's ability to simulate different network configurations without re-synthesis. The results presented in this section are simulated using a randomly generated permutation traffic pattern. All configurations are implemented on the same 9-node DART described in Section IV.

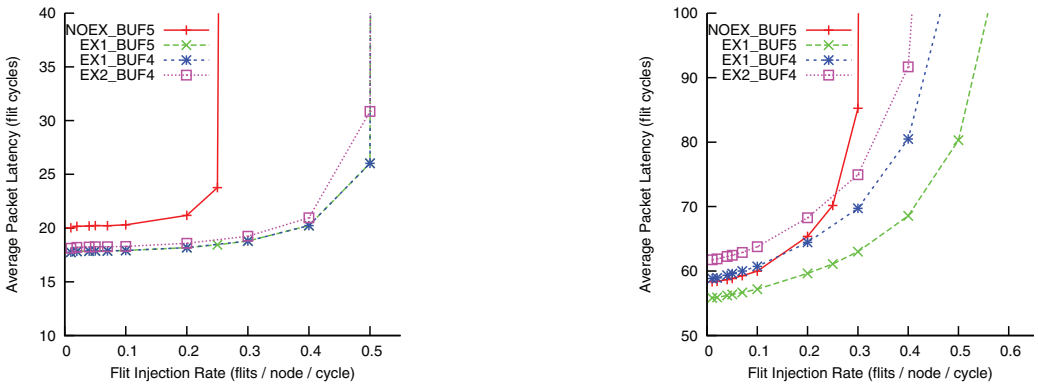


Fig. 19. Express links performance: (a) 2-flit packets and (b) 16-flit packets

1) Mesh with Express Links: Fig. 18a illustrates a simplified version of an express cube [23]. The solid lines represent local links and the dashed lines represent express links that allow non-local traffic to bypass intermediate nodes. Fig. 19 shows the average packet latency for the following configurations:

- **NOEX_BUF5**: No express link, 5 flits/VC input buffer
- **EX1_BUF5**: With express links, 5 flits/VC input buffer
- **EX1_BUF4**: With express links, 4 flits/VC input buffer
- **EX2_BUF4**: EX1_BUF4 with 2-cycle express links

For all packet sizes, the express links reduce packet latency because flits traverse fewer hops. The increased bisection bandwidth afforded by the added links also allows the network to accept higher load before saturating. To compensate for the additional area the added express link port incurs in each router, we reduce the input buffer size from 5 flits to 4 flits (EX1_BUF4 vs. EX1_BUF5). The resulting performance degradation is progressively more pronounced for traffic with larger packet sizes because they are more bursty, hence more sensitive to buffer space in the router. Because the express links span two hops, we increase their latency to 2 cycles while keeping the latency of other links at 1 cycle (EX2_BUF4). This configuration causes higher latency for large packets because credits take longer to replenish on the slower express links. However, the network still saturates later than the NOEX_BUF5 baseline.

2) Tree: Fig. 18b illustrates a tree where two sub-trees are linked by a root router, where the bold lines represent global links. This organization captures the essence of building blocks in a hierarchical on-chip network. Only the leaf nodes generate and receive traffic, and 50% of the generated traffic crosses the root router. Fig. 20 shows the average packet latency for the following configurations:

- **BUF5_BW1**: Unit latency and bandwidth for all links, 5 flits/VC input buffer
- **BUF5_BW2**: BUF5_BW1, bandwidth = 2 flits/cycle on global links
- **BUF10_BW1**: BUF5_BW1 with 10 flits/VC input buffer
- **BUF10_BW2**: BUF5_BW2, with 10 flits/VC input buffer

For all packet sizes, increasing the global link bandwidth (BUF5_BW2 vs. BUF5_BW1, BUF10_BW2 vs. BUF10_BW1) does not significantly improve packet latency because all flits crossing the global links must be first stored in the buffers of the gateway routers (nodes 1 and 2), which form the performance bottleneck. Increasing the buffer space to 10 flits significantly reduces latency. Again the reduction is greater for large packets because bursty traffic is more sensitive to buffer sizes. Moreover, Fig. 20a and 20b show that once the buffer space bottleneck is removed, increasing global link bandwidth can provide additional performance improvement.

VI. Conclusions

We have presented a software-programmable overlay architecture for NoC simulation on FPGAs. By decoupling the simulator architecture from the architecture of the simulated NoC and virtualizing simulation time, DART improves upon existing FPGA-based emulators by eliminating the high cost of modifying and resynthesizing the hardware emulator when simulating different NoCs. At the same time, DART is significantly faster than software NoC simulators. Using an implementation of a 9-node DART simulator on a Virtex II Pro FPGA, we demonstrate over 100-fold speedup over Booksim while maintaining a similar level of accuracy. Virtualization allows DART to support more simulated NoC nodes than physical. Through two examples, we also show that irregular NoCs can be easily set up and simulated on DART. Finally, we demonstrate that a 49-node DART can be supported on a Xilinx Virtex-6 FPGA.

References

[1] W. Dally and B. Towles, "Route packets, not wires: on-chip interconnection networks," in *Proc. Design Automation Conference*, 2001.

[2] Y. Krasteva, F. Criado, E. de la Torre, and T. Riesgo, "A Fast Emulation-Based NoC Prototyping Framework," in *Proc. Int'l Conf. on Reconfigurable Computing and FPGAs*, Dec. 2008.

[3] N. Genko, D. Atienza, G. De Micheli, J. Mendias, R. Hermida, and F. Catthoor, "A complete network-on-chip emulation framework," in *Proc. Design, Automation and Test in Europe*, March 2005.

[4] W. Dally and B. Towles, *Principles and Practices of Interconnection Networks*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2003.

[5] V. Puente, J. Gregorio, and R. Beivide, "SICOSYS: an integrated framework for studying interconnection network performance in multiprocessor systems," in *Euromicro Workshop on Parallel, Distributed and Network-based Processing*, 2002.

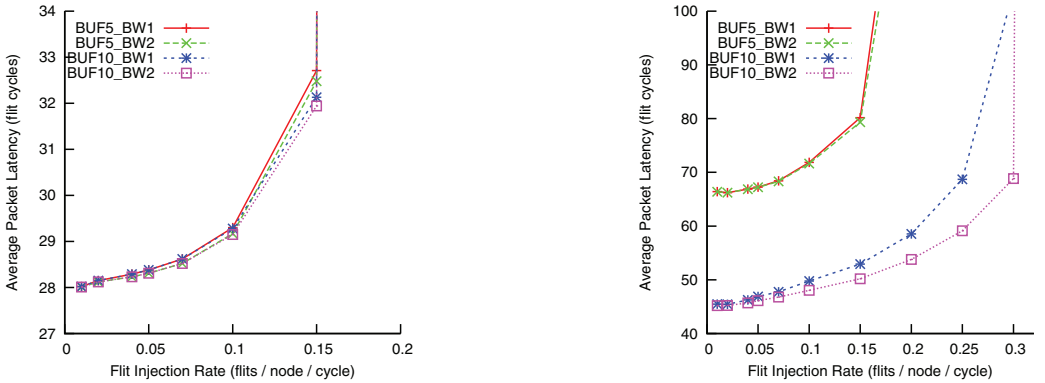


Fig. 20. Tree performance: (a) 2-flit packets and (b) 16-flit packets

[6] N. Hardavellas, S. Somogyi, T. F. Wenisch, R. E. Wunderlich, S. Chen, J. Kim, B. Falsafi, J. C. Hoe, and A. G. Nowatzyk, "SimFlex: a fast, accurate, flexible full-system simulation framework for performance evaluation of server architecture," *SIGMETRICS Perform. Eval. Rev.*, vol. 31, no. 4, pp. 31–34, 2004.

[7] N. Agarwal, T. Krishna, L.-S. Peh, and N. Jha, "GARNET: A detailed on-chip network model inside a full-system simulator," in *Proc. Int'l Symp. on Performance Analysis of Systems and Software*, April 2009.

[8] G. Schelle and D. Grunwald, "Onchip Interconnect Exploration for Multicore Processors Utilizing FPGAs," in *2nd Workshop on Architecture Research using FPGA Platforms*, 2006.

[9] P. Wolkotte, P. Holzenspies, and G. Smit, "Fast, Accurate and Detailed NoC Simulations," in *Proc. Int'l Symp. on Networks-on-Chip*, May 2007.

[10] M. M. K. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood, "Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset," *SIGARCH Comput. Archit. News*, vol. 33, no. 4, pp. 92–99, 2005.

[11] M. Pellauer, M. Vijayaraghavan, M. Adler, Arvind, and J. Emer, "A-Ports: An Efficient Abstraction for Cycle-Accurate Performance Models on FPGAs," in *Proc. Int'l Symp. on Field Programmable Gate Arrays*, Feb. 2008.

[12] M. Lis, K. Shim, M. Cho, P. Ren, O. Khan, and S. Devadas, "DARSIM: a Parallel Cycle-Level NoC Simulator," in *6th Annual Workshop on Modeling, Benchmarking and Simulation*, June 2010.

[13] M. Lis, P. Ren, M. Cho, K. Shim, C. Fletcher, O. Khan, and S. Devadas, "Scalable, accurate multi-core simulation in the 1000-core era," in *Int'l Symp. on Performance Analysis of Software and Systems*, April 2011.

[14] M. K. Papamichael, "Fast scalable FPGA-based network-on-chip simulation models," in *Proc. of Int'l Conf. on Formal Methods and Models for Codesign*, 2011.

[15] E. Chung, E. Nurvitadhi, J. Hoe, B. Falsafi, and K. Mai, "PROToFLEX: FPGA-accelerated Hybrid Functional Simulator," in *Proc. Int'l Parallel and Distributed Processing Symposium*, March 2007.

[16] Z. Tan, A. Waterman, H. Cook, S. Bird, K. Asanović, and D. Patterson, "A Case for FAME: FPGA Architecture Model Execution," in *37th Proc. Int'l Symp. on Computer Architecture*, May 2010.

[17] M. Pellauer, M. Adler, M. Kinsy, A. Parashar, and J. Emer, "HAsim: FPGA-based high-detail multicore simulation using time-division multiplexing," in *Proc. Int'l Symp. on High Performance Computer Architecture*, 2011.

[18] Z. Tan, A. Waterman, R. Avizienis, Y. Lee, H. Cook, D. Patterson, and K. Asanović, "RAMP Gold: An FPGA-based architecture simulator for multiprocessors," in *Proc. Design Automation Conference*. ACM, 2010, pp. 463–468.

[19] L. Peh and W. Dally, "A delay model and speculative architecture for pipelined routers," in *Seventh Proc. Int'l Symp. on High-Performance Computer Architecture*, 2001.

[20] Xilinx, Inc., "Xilinx university program Virtex-II pro development system hardware reference manual," 2008. [Online]. Available: http://www.xilinx.com/univ/XUPV2P/Documentation/ug069.pdf

[21] ——, "Xilinx ML605 development system hardware reference manual," 2011. [Online]. Available: http://www.xilinx.com/support/documentation/boards_and_kits/ug535.pdf

[22] ——, "Using look-up tables as shift registers (srl16) in Spartan-3 generation FPGAs," *Appl. Note XAPP465*, May 2005.

[23] W. Dally, "Express cubes: Improving the performance of k-ary n-cube interconnection networks," *IEEE Transactions on Computers*, pp. 1016–1023, 1991.

Danyao Wang received the B.A.Sc. degree from the Engineering Science program at the University of Toronto, and the M.A.Sc. degree from the Electrical and Computer Engineering Department at the same university. She is currently employed at Google Waterloo.

Charles Lo received the B.A.Sc. degree from the Engineering Science program at the University of Toronto, and is currently a M.A.Sc. candidate in the Electrical and Computer Engineering Department at the same university.

Jasmina Vasiljevic received the B.A.Sc. and M.A.Sc. degrees from the Electrical and Computer Engineering Department at Ryerson University. She is currently a Ph.D. candidate in the Electrical and Computer Engineering Department at the University of Toronto.

Natalie Enright Jerger received the B.A.Sc. degree from the Department of Electrical and Computer Engineering at Purdue University, and the M.S.E.E. and Ph.D. degrees from the Department of Electrical and Computer Engineering, University of Wisconsin - Madison. She is currently an Assistant Professor in the Electrical and Computer Engineering Department at the University of Toronto.

J. Gregory Steffan received the B.A.Sc. and M.A.Sc. degrees from the Electrical and Computer Engineering Department at the University of Toronto, and the Ph.D. degree from the Department of Computer Science, Carnegie Mellon University. He is currently an Associate Professor in the Electrical and Computer Engineering Department at the University of Toronto.