DARSIM Design Overview

Mieszko Lis

0.06

Contents

List of Figures

1 Introduction

This document provides an overview of DARSIM parallel cycle-level network-on-chip (NoC) simulator [DS], including the major components and their interfaces as well as a trace of a packet as it transits the system.

2 Design Overview

2.1 Simulating registered hardware

Since DARSIM simulates a parallel hardware system inside a partially sequential program at cycle level, it must reflect the parallel behavior of the hardware: all values computed within a single clock cycle and stored in registers become visible simultaneously at the beginning of the next clock cycle. To simulate this, most simulator objects respond to `tick_positive_edge()` and `tick_negative_edge()` methods, which correspond to the rising and falling edges of the clock; conceptually, computation occurs on the positive clock edge and the results are stored in a shadow state, which then becomes the current state at the next negative clock edge. (See Section 2.5 for more details).

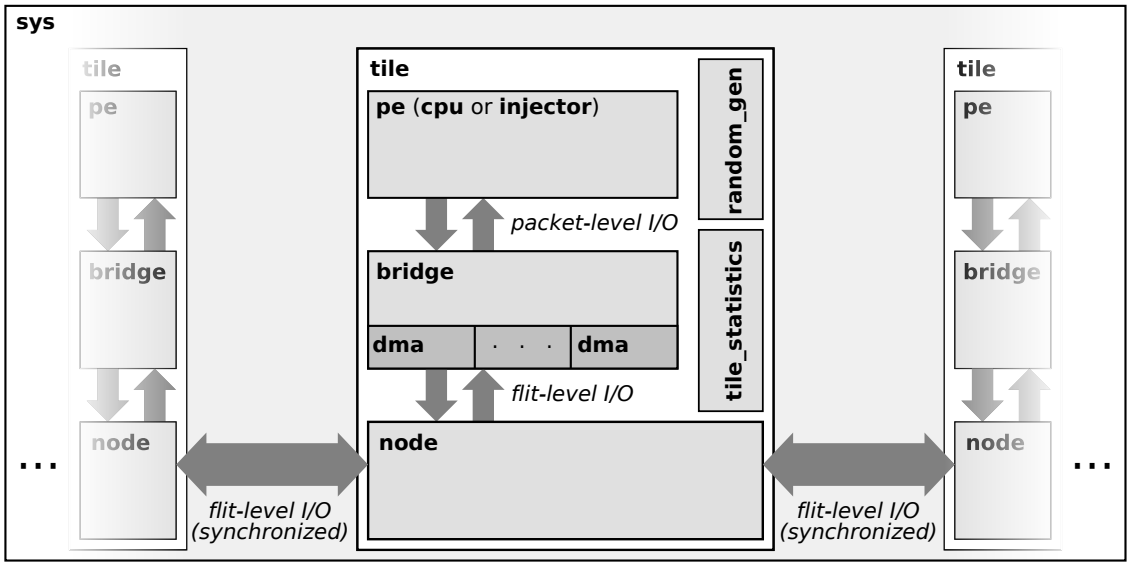
For cycle-accurate results in a multi-threaded simulation, a the simulation threads must be barrier-synchronized on every positive edge and every negative edge. A speed-vs-accuracy tradeoff is possible by performing barrier synchronization less often: while per-flit and per-packet statistics are transmitted with the packets and still accurately measure transit times, the flits may observe different system states and congestions along the way and the results may nevertheless differ (cf. Section 2.7).

2.2 Tile-based system model

The DARSIM NoC system model (defined in `sys.hpp` and `sys.cpp`) is composed of a number of interconnected tiles (defined in `tile.hpp` and `tile.cpp`). As shown in Figure 1, each tile comprises a processing element (PE), which can be a MIPS CPU simulator or a script-driven injector, a bridge that converts packets to flits, and, finally, the network switch node itself.

Since each tile can be run in a separate thread, inter-tile communication is synchronized using fine-grained locks (see Section 2.5). To avoid unnecessary synchronization, each tile has a private independently initialized Mersenne Twister random number generator and collects its own statistics; at the end of the simulation, the per-tile statistics are collected and combined into whole-system statistics.

Figure 1 A system as simulated by DARSIM



2.3 The PE and the bridge

In addition to the base `pe` class (defined in `pe.hpp` and `pe.cpp`), DARSIM includes a cycle-level MIPS CPU simulator with a local memory (`cpu.hpp` and `cpu.cpp`) and a script-driven injector (`injector.hpp` and `injector.cpp`).

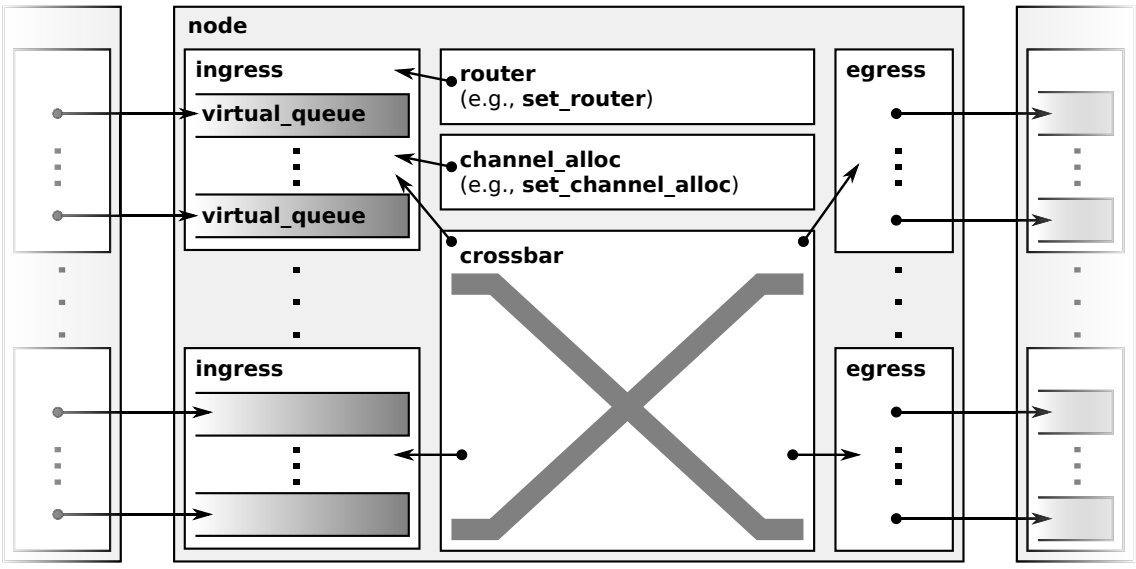
The PE interacts with the network via a bridge (`bridge.hpp` and `bridge.cpp`), which exposes a packet-based interface to the PE and a flit-based interface to the network. The bridge exposes a number of incoming queues which the PE can query and interact with, and provides a number of DMA channels for sending and receiving packet data. The processing element can check if any incoming queues have waiting data using `bridge::get_waiting_queues()` and query waiting packets with `bridge::get_queue_flow_id()` and `bridge::get_queue_length()`; it can also initiate a packet transfer from one of the incoming queues with `bridge::receive()`, start an outgoing packet transfer via `bridge::send()`, and check whether the transfer has completed with `bridge::get_transmission_done()`.

Once the bridge receives a request to send or receive packet data, it claims a free DMA channel (defined in `dma.hpp` and `dma.cpp`) if one is available or reports failure if no channels are free. Each DMA channel corresponds to a queue inside the network switch node, and slices the packet into flits (`flit.hpp` and `flit.cpp`), appending or stripping a head flit as necessary. The transfer itself is driven by the system clock in `ingress_dma_channel::tick_positive_edge()` and `egress_dma_channel::tick_positive_edge()`.

2.4 The network switch node

The interconnected switch nodes (see Figure 2) that form the network fabric are responsible for delivering flits from the source bridge to the destination bridge. The node (defined in `node.hpp` and `node.cpp`) models an ingress-queued wormhole router with highly configurable, table-based route and virtual channel allocation and a configurable crossbar.

Figure 2 A DARSIM network node



The `ingress` class (`ingress.hpp` and `ingress.cpp`) models a router ingress port; there is (at least) one ingress for each neighboring network node and one ingress for each connected bridge; each ingress manages a number of virtual queues (`virtual_queue.hpp` and `virtual_queue.cpp`). Egresses (`egress.hpp` and `egress.cpp`) contain no buffering and only hold references to the corresponding neighbor-node ingresses.

Next-hop routes and flow IDs are allocated by `router::route()` (in `router.hpp` and `router.cpp`) whenever a head flit arrives at the front of a virtual queue; the router holds references to all virtual queues

in the node and directly modifies them by invoking `virtual_queue::front_set_next_hop()`. Specific router implementations inherit from the `router` class (for example, the configurable table-driven `set_router` in `set_router.hpp` and `set_router.cpp`). Similarly, next-hop virtual channel allocation is handled by `channel_alloc::allocate()` via a call to `virtual_queue::front_set_vq_id()`, and specific channel allocator implementations like `set_channel_alloc` in `set_channel_alloc.hpp` and `set_channel_alloc.cpp`. Each virtual queue remembers its next-hop assignments until the last flit of the current packet has left the queue.

Virtual queues with valid next-hop assignments compete for crossbar transfers to the next-hop node or bridge. In each clock cycle, `crossbar::tick_positive_edge()` examines the competing ingress queues and invokes `virtual_queue::front_pop()` of the winning queues and `virtual_queue::back_push()` of the corresponding next-hop queues until crossbar bandwidth for that cycle is exhausted.

2.5 Synchronized communication

The virtual queue (defined in `virtual_queue.hpp` and `virtual_queue.cpp`) models a virtual channel buffer, and, as the only point of inter-thread communication, is synchronized in multi-thread simulations. The fine-grained synchronization ensures correctness even if the simulation threads are only loosely synchronized.

Synchronization is achieved via two mutual exclusion locks: `virtual_queue::front_mutex` and `virtual_queue::back_mutex`. During a positive edge of the clock, the two ends of the queue are independent, and operations on the front of the queue (e.g., `virtual_queue::front_pop()`) only lock `front_mutex` while operations on the back (e.g., `virtual_queue::back_push()`) only lock `back_mutex`; because no communication between the two occurs, flits added to the back of the queue are not observed at the front of the queue in the same cycle (so, for example, `virtual_queue::back_is_full()` can report a full queue even if a flit was removed via `virtual_queue::front_pop()` during the same positive edge. During the negative clock edge, the changes made during the positive edge are communicated between the two ends of the queue, and both locks are held.

2.6 Initialization

At startup, DARSIM reads a binary *image file* which describes the components of the network, how they are configured, and how they are connected; for tiles that include a CPU, the image also describes memory contents, the initial program counter, and the initial stack pointer. The image is generated from a text-based configuration file, and, optionally, MIPS executables, by the tool `darimg`, and parsed in `sys::sys()`.

If the system was configured to include injectors, they are programmed using one or more text-based *event files*. The event files specify packets to be sent in each cycle for each flow; they are read by `event_parser::event_parser()` and distributed to the injectors where the flows originate by calling `injector::add_event()`.

2.7 Statistics

DARSIM collects various statistics during the simulation. To avoid synchronization in a multi-threaded simulation, each tile collects its own statistics (class `tile_statistics` in `statistics.hpp` and `statistics.cpp`); a `system_statistics` object keeps track of the `tile_statistics` objects and combines them to report whole-system statistics at the end of the simulation.

To accurately collect per-flit statistics when the simulation threads are only loosely barrier-synchronized, some counters (for example, the elapsed flit transit time) are stored and carried together with the flit (see `flit.hpp`), and updated during every clock cycle.

3 The Life of a Packet

3.1 Injection from the source PE

The packet is generated either by an injector (in `injector::tick_positive_edge()`) or by a system call executed by the MIPS CPU simulator (in `cpu::syscall()`) via a call to `bridge::send()` with

the flow ID, packet length, and a pointer to the memory region to be transmitted. If a DMA channel is available, this in turn invokes `egress_dma_channel::send()`, which stores the packet details in the `egress_dma_channel` object for later transmission.

3.2 Transport to network via DMA

Next, the packet contents are split into flits and a head flit containing the flow ID and the number of data flits that follow is prepended. In every clock cycle, `egress_dma_channel::tick_positive_edge()` transmits as much of the packet as possible flit by flit by repeatedly calling `virtual_queue::back_push()` until the relevant node ingress queue is full or the available port bandwidth has been exceeded.

3.3 Node-to-node transport

Once in a node ingress queue, flits are transmitted when `node::tick_positive_edge()` is invoked during every cycle.

For every head flit at the head of a virtual channel queue, `router::route()` determines the next-hop node ID and flow ID and calls `virtual_queue::front_set_next_hop()` to assign them to the packet. Next, `channel_alloc::allocate()` computes the next-hop virtual channel assignment and sets it using `virtual_queue::front_set_vq_id()`.

Finally, each flit of the packet competes for crossbar bandwidth and transits to the next-hop node in `crossbar::tick_positive_edge()`, which in turn calls `virtual_queue::front_pop()` and `virtual_queue::back_push()` to effect the transfer.

3.4 Delivery to the destination PE

At the destination node, the router and channel allocator direct the packet to a queue in the connected bridge, and the crossbar completes the transfer. The relevant bridge queue is then reported as ready when `bridge::get_waiting_queues()` is invoked and the PE can retrieve it with `bridge::receive()`.

4 References

[DS] Mieszko Lis, Keun Sup Shim, Myong Hyon Cho, Pengju Ren, Omer Khan and Srinivas Devadas (2010). DARSIM: a parallel cycle-level NoC simulator. In Proceedings of MoBS 2010.