BookSim 2.0 User's Guide

Nan Jiang, George Michelogiannakis, Daniel Becker, Brian Towles and William J. Dally

March 8, 2010

Contents

1 Introduction

This document describes the use of the BookSim interconnection network simulator. The simulator is designed as a companion to the textbook "Principles and Practices of Interconnection Networks" (PPIN) published by Morgan Kaufmann (ISBN: 0122007514) and it is assumed that is reader is familiar with the material covered in that text.

This user guide is fairly brief as, with most simulators, the best way to learn and *understand* the simulator is to study the code. Most of the simulator's components are designed to be modular so tasks such as adding a new routing algorithm, topology, or router microarchitecture should not require a complete redesign of the code. Once you have downloaded the code, compiled it, and run

a simple example (Section 2), the more detailed examples of Section 3 give a good overview of the capabilities of the simulator. A list of configuration options is provided in Section 4 for reference.

2 Getting started

2.1 Downloading and building the simulator

The latest official release of the simulator can be checked out from Booksim's sourceforge subversion (SVN) repository. Make sure SVN is installed on your machine and run the following command:

```
svn co https://booksim.svn.sourceforge.net/svnroot/booksim
```

The simulator files should now be in the `booksim/trunk/src/` directory. The simulator itself is written in C++ and has been specifically tested with GNU's G++ compiler (version $\geq$ 3). The front end of the simulator uses LEX and YACC generated parser to process the simulator configuration file; however, unless you plan on making changes to the front end parser, LEX and YACC are not needed. The `Makefile` should be edited so that the first few lines reflect the correct paths to the tools for your particular system. The default `Makefile` should work on the Stanford Leland machines. Type `make` to build the simulator.

A note for Windows users: The above instructions have been tested to work with the newest version of Cygwin.

2.2 Running a simulation

The simulator is invoked using the following command line:

```
./booksim [configfile]
```

The parameter `configfile` is a file that contains configuration information for the simulator. So, for example, to simulate the performance of a simple 8×8 torus (8-ary 2-cube) network on uniform traffic, a configuration such as the one shown in Figure 1 could be used. This particular example configuration file can be found in the `examples/torus88` directory.

In addition to specifying the topology, the configuration file also contains basic information about the routing algorithm, flow control, and traffic. This simple example uses dimension-order routing and four virtual channels. The `injection_rate` parameter is added to tell the simulator to inject (on average) 0.15 packets per simulation cycle per node. Packet size defaults to a single flit. Also, any line of the configuration file that begins with `//` is treated as a comment and ignored by the simulator. A detailed list of configuration parameters is given in Section 4. Any parameters not specified by the user will take on default values. The default value for every parameter in the simulator is specified in the file `booksim_config.cpp`.

2.3 Simulation output

Continuing our example, running the torus simulation produces the output shown in Figure 2. Each simulation has three basic phases: warm up, measurement, and drain. The length of the warm up and measurement phases is a multiple of a basic sample period (defined by `sample_period` in the configuration). As shown in the figure, the current latency and throughput (rate of accepted packets) for the simulation is printed after each sample period. The overall throughput is determined by the lowest throughput of all the destination in the network, but the average throughput is also displayed.

```
// Topology
topology = torus;
k        = 8;
n        = 2;

// Routing
routing_function = dim_order;

// Flow control
num_vcs = 4;

// Traffic
traffic        = uniform;
injection_rate = 0.15;
```

Figure 1: Example configuration file for simulating a 8-ary 2-cube network.

After the warm up periods have passed (default to 3×sample_period), the simulator prints the "`Warmed up`" message and resets all the simulation statistics. Then, the measurement phase begins and statistics continue to be reported after each sample period. Once the measurement periods have passed, all the measurement packets are drained from the network before final latency and throughput numbers are reported. Details of the configuration parameters used to control the length of the simulation phases are covered in Section 4.8.

3 Example

One of the most basic performance measures of any interconnection network is its latency versus offered load. Figure 3 shows a simple configuration file for making this measurement in a 8-ary 2-mesh network under the transpose traffic pattern. This configuration was used to generate a single data point in Figure 25.2 in PPIN. The particular configuration accounts for some small delays and pipelining of the input-queued router and also introduces a small input speedup to account for any inefficiencies in allocation. By running simulations for many increments of `injection_rate`, the average latency curve can be found. Then, to compare the performance of dimension-order routing against several other routing algorithms, for example, the `routing_function` option can be changed.

4 Configuration parameters

All information used to configure a simulation is passed through a configuration file as illustrated by the example in Section 2.2. This section lists the existing configuration parameters — a user can incorporate additional options by changing the `booksim_config.cpp` file.

4.1 Topologies

The `topology` parameter determines the underlying topology of the network. There is also a set of numerical parameters that describes the size of the networks.

```
BEGIN Configuration File
Name: examples/torus88
// Topology
topology = torus;
k = 8;
n = 2;
// Routing
routing_function = dim_order;
// Flow control
num_vcs = 4;
// Traffic
traffic = uniform;
injection_rate = 0.15;

END Configuration File
0%================================
% Average latency = 10.3206
% Accepted packets = 0.113 at node 38 (avg = 0.149031)
lat(1) = 10.3206;
bins = [ 0 ...999];
freq = [ 0 0 145 4 550 52 988 137 1439 285 1656 428 1359 417 958 294 453 163 143
 49 10 7 1 0 ....

....

% Warmed up ...Time used is 3000 cycles

....

% Draining all recorded packets ...
% Draining remaining packets ...
Time taken is 6030 cycles
====== Traffic class 0 ======
Overall average latency = 10.2897 (1 samples)
Overall average accepted rate = 0.14834 (1 samples)
Overall min accepted rate = 0.129043 (1 samples)
Average hops = 4.99839 (57153 samples)
```

Figure 2: Simulator output from running the `examples/torus88` configuration file.

```
// Topology

topology = mesh;
k = 8;
n = 2;

// Routing

routing_function = dim_order;

// Flow control

num_vcs     = 8;
vc_buf_size = 8;

wait_for_tail_credit = 1;

// Router architecture

vc_allocator = islip;
sw_allocator = islip;
alloc_iters  = 1;

credit_delay   = 2;
routing_delay  = 1;
vc_alloc_delay = 1;

input_speedup     = 2;
output_speedup    = 1;
internal_speedup  = 1.0;

// Traffic

traffic                = transpose;
const_flits_per_packet = 20;

// Simulation

sim_type       = latency;
injection_rate = 0.01;
```

Figure 3: A typical configuration file (`examples/mesh88_lat`) for creating a latency versus offered load curve for a 8-ary 2-mesh network.

`k` Network radix, the number of routers per dimension

`n` Network dimension

`c` Network concentration, the number of nodes sharing a single router. Typically set to 1, ¿1 only in networks that has concentration (i.e. cmesh)

`x` (NoC simulations only) The number of routers in the X dimension. Used to calculate channel latency between routers.

`y` (NoC simulations only) The number of routers in the y dimension. Used to calculate channel latency between routers.

`xr` (NoC simulations only) For networks that have c¿1, the number of nodes in the x direction per router. Used to calculate channel latency between routers.

`yr` (NoC simulations only) For networks that have c¿1, the number of nodes in the y direction per router. Used to calculate channel latency between routers.

`limit` Emulate partially filled networks. Nodes greater or equal to limit does not generate or receive packets during simulation. Though depending on the routing algorithm the routers associated with those nodes may be use to route passing by traffic.

The channel latency of the network must be configured within the source code of the topology files. All topologies by default have channel latency of 1 cycles. Topologies available in BookSim are:

`fly` A k-ary n-fly (butterfly) topology. The `k` parameter determines the network's radix and the `n` parameter determines the network's dimension.

`mesh` A k-ary n-mesh (mesh) topology. The `k` parameter determines the network's radix and the `n` parameter determines the network's dimension.

`single` A network with a single node, used for testing single router performance. The number of input and output ports for the node is determined by the `in_ports` and `out_ports` parameters, respectively.

`torus` A k-ary n-cube (torus) topology. The `k` parameter determines the network's radix and the `n` parameter determines the network's dimension.

`cmesh` Concentrated mesh topology is a A k-ary n-mesh topology with multiple nodes sharing a single router. The `c` determines the concentration. The cmesh topology has the option that turns on "express channels" as described in by default these channels are turned off.

`fat tree` Fat Tree topology with 3 levels. Nodes are routers are arranged in a tree structure but the number of links between levels stays constant. At the bottom `k` nodes shares a level 0 router.

`flattened butterfly` A topology based on the paper "Flattened butterfly: a cost-efficient topology for high-radix networks" ISCA 2007

`quad tree` A quad tree topology.

`tree 4`

`cmo` Concentrated Multi-dimensional Octagon created by Sang Kyun Kim and Wook-Jin Chung from EE382C several years ago.

`MECS` A topology based on ***. Currently it only supports single-flit packets.

Both the `mesh` and `torus` topologies support the addition of random link failures with the `link_failures` parameter. The value of `link_failures` determines the number of channels that are randomly removed from the topology and are thus no longer available for forwarding packets. Moreover, the randomization for failed channels is controlled by selecting an integer value for the `fail_seed` parameter — a fixed seed gives a fixed set of failed channels, independent of other randomization in the simulation. Also, note that only certain routing functions support this feature (see Section 4.3).

4.2 Physical sub-networks

The `physical_subnetworks` parameter defines the number of physical sub-networks present in the network (defaults to one). All sub-networks receive the same configuration parameters and thus are identical. Traffic sources maintain an injection queue for each sub-network. The packet generation process is unaffected. It enqueues generated packets into the proper sub-network queue according to a division function in the traffic manager. At every cycle, flits at the head of each queue attempt to be injected. Traffic destinations can eject one flit from each sub-network each cycle.

4.3 Routing algorithms

The `routing_function` parameter selects a routing algorithm for the topology. Many routing algorithms need multiple virtual channels for deadlock freedom (VCDF). In addition to `routefunc.cpp`, some topologies source files include additional routing functions. Also, the simulator code is structured so that additional routing algorithms can be added with minimal changes to the overall simulator (see the `routefunc.cpp` file in the simulator's source code).

4.4 Flow control

The simulator supports basic virtual-channel flow control with credit-based backpressure.

`num_vcs` The number of virtual channels per physical channel.

`vc_buf_size` The depth of each virtual channel in flits.

`wait_for_tail_credit` If non-zero, do not reallocate a virtual channel until the tail flit has left that virtual channel. This conservative approach prevents a dependency from being formed between two packets sharing the same virtual channel in succession.

4.5 Router organizations

The simulator also supports two different router microarchitectures. The input-queued router follows the general organization described in PPIN while the event-driven router is modeled after the router used in the Avici TSR and described in U.S. Patent 6,370,145. The microarchitecture is selected using the `router` option. Also, both routers share a small set of options.

`credit_delay` The processing delay (in cycles) for a credit. Does not include the wire delay for transmitting the credit.

`internal_speedup` An arbitrary speedup of the internals of the routers over the channel transmission rate. For example, a speedup 1.5 means that, on average, 1.5 flits can be forwarded by the router in the time required for a single flit to be transmitted across a channel. Also, the configuration parser expects a floating point number for this field, so integer speedups should also include a decimal point (e.g. "2.0").

`output_delay` The processing delay incurred in the output queue of a router.

4.5.1 The input-queued router

The input-queued router (`router = iq`) follows the pipeline described in PPIN of route computation, virtual-channel allocation, switch allocation, and switch traversal. There are several options specific to the input-queued router.

`input_speedup` An integer speedup of the input ports in space. A speedup of 2, for example, gives each input two input ports into the crossbar. Access to these ports is statically allocated based on the virtual channel number: virtual channel v at input i is connected to port $i \cdot s + (v \mod s)$ for an input speedup of s.

`output_speedup` An integer speedup of the output ports in space. Similar to `input_speedup`

`routing_delay` The delay (in cycles) of route computation.

`hold_switch_for_packet`

`speculative` Enable speculative switch allocation (i.e., allow switch allocation to occur in parallel with VC allocation for header flits).

`filter_spec_grants` Determines how speculative grants are masked (`any_nonspec_gnts`: any non-speculative grant inhibits all speculative grants; `confl_nonspec_reqs`: speculative grants are inhibited by conflicting non-speculative requests; `confl_nonspec_gnts`: speculative grants are inhibited by conflicting non-speculative grants).

`alloc_iters` For the `islip`, `pim` and `select` allocators, allocation can be improved by performing multiple iterations of the algorithm; this parameter controls the number of iterations to be performed for both switch and VC allocation.

`sw_allocator` The type of allocator used for switch allocation. See Section 4.6 for a list of the possible allocators.

`sw_alloc_arb_type` If the switch allocator is a separable input- or output-first allocator, this parameter selects the type of arbiter to use.

`sw_alloc_iters` If the switch allocator is of type `islip`, `pim` or `select`, this parameter controls how many allocation iterations should be performed. This overrides the global `alloc_iters` parameter.

`sw_alloc_delay` The delay (in cycles) of switch allocation.

`vc_allocator` The type of allocator used for virtual-channel allocation. See Section 4.6 for a list of the possible allocators.

`vc_alloc_arb_type` If the VC allocator is a separable input- or output-first allocator, this parameter selects the type of arbiter to use.

`vc_alloc_iters` If the VC allocator is of type `islip`, `pim` or `select`, this parameter controls how many allocation iterations should be performed. This overrides the global `alloc_iters` parameter.

`vc_alloc_delay` The delay (in cycles) of virtual-channel allocation.

4.5.2 The event-driven router

The event-driven router (`router = event`) is a microarchitecture designed specifically to support a large number of virtual channels (VCs) efficiently. Instead of continuously polling the state of the virtual channels, as in the input-queued router, only changes in VC state are tracked. The efficiency then comes from the fact that the number of state changes per cycle is constant and independent of the number of VCs.

4.6 Allocators

Many of the allocators used in the simulator are configurable (see the input-queued router in Section 4.5.1) and several allocation algorithms are available.

`max_size` Maximum-size matching.

`islip` iSLIP separable allocator.

`pim` Parallel iterative matching separable allocator.

`loa` Lonely output allocator.

`wavefront` Wavefront allocator.

`separable_input_first` Separable input-first allocator.

`separable_output_first` Separable output-first allocator.

`select` Priority-based allocator. Allocation is performed as in iSLIP, but with preference towards higher priority packets (see `priority` option in Section 4.7).

4.7 Traffic

4.7.1 Injection mode

The rate at which packets are injected into the simulator is set using the `injection_rate` option. The simulator's cycle time is a flit cycle, the time it takes a single flit to be injected at a source, and the injection rate is specified in packets per flit cycle. For example, setting `injection_rate = 0.25` means that each source injects a new packet in one out of every four simulator cycles. The unit of `injection_rate` can optionally be changed to flits per cycle by setting `injection_rate_uses_flits` to 1.

The injection process can further be specified as either Bernoulli process (`injection_process = bernoulli`) or an on-off process (`injection_process = on_off`). The burstiness of the latter is controlled via the `burst_alpha` and `burst_beta` parameters. See PPIN Section 24.2.2 for a description of the on-off process and its parameters.

4.7.2 Request-reply traffic

By default, all packets that are injected into the network have the same, fixed length. The number of flits per packet is set using the `const_flits_per_packet` option.

Alternatively, the simulator can be configured to generate request-reply traffic. In this mode, the traffic manager injects read and write requests into the network; when a destination node receives such a request packet, it returns a reply packet of the same type. Injection of reply packets has priority over injection of new requests packets. Consequently, the effective overall network load is generated by both the injected requests and the automatically generated replies. Request-reply traffic ignores the `const_flits_per_packet` option; instead, packet sizes are determined by the {`read|write`}_{`request|reply`}_`size` options. Furthermore, the mapping of packet types to VCs can be customized using the {`read|write`}_{`request|reply`}_{`begin|end`}_`vc` options.

4.7.3 Traffic patterns

The simulator also supports several different traffic patterns that are specified using the `traffic` option. To describe these patterns, we use the same notation of PPIN Section 3.2: s_i (d_i) denotes the i^{th} bit of the source (destination) address whereas s_x (d_x) denotes the x^{th} radix-k digit of the source (destination) address. The bit length of an address is $b = \log_2 N$, where N is the number of nodes in the network.

`uniform` Each source sends an equal amount of traffic to each destination (`traffic = uniform`).

`bitcomp` Bit complement. $d_i = \neg s_i$.

`bitrev` Bit reverse. $d_i = s_{b-i-1}$.

`shuffle` $d_i = s_{i-1 \mod b}$.

`transpose` $d_i = s_{i+b/2 \mod b}$.

`tornado` $d_x = s_x + \lceil k/2 \rceil - 1 \mod k$.

`neighbor` $d_x = s_x + 1 \mod k$.

`randperm` Random permutation. A fixed permutation traffic pattern is chosen uniformly at random from the set of all permutations. The seed used to generate this permutation is set by the `perm_seed` option. So, randomly selecting values for `perm_seed` gives a random sampling of permutations while a fixed value of `perm_seed` allows the same permutation to be used for several experiments.

4.8 Simulation parameters

The duration and other aspects of a simulation are controlled using the set of simulation parameters.

`sim_type` A simulation can either focus on `throughput` or `latency`. The key difference between these two types is that a `latency` simulation will wait for all measurement packets to drain before ending the simulation to ensure an accurate latency measurement. In `throughput` simulations, this final drain step is eliminated to allow simulation of networks operating beyond their saturation point.

`sample_period` The sample period is expressed in simulator cycles and is used as a multiplier when specifying the warm-up length of a simulation and the maximum number of samples. Also, intermediate statistics are displayed once every `sample_period` cycles. This is only applicable in injection mode.

`warmup_periods` The length of the simulator warm up expressed as a multiple of the `sample_period`. After warming up, all statistics counters are reset. This is only applicable in injection mode.

`max_samples` The total length of simulation expressed as a multiple of the `sample_period`. This is only applicable in injection mode.

`latency_thres` If the sampled latency of the current simulation exceeds `latency_thres`, the simulation is immediately ended.

`sim_count` The number of back-to-back simulations to run for the given configuration. Useful for creating ensemble averages of particular statistics.

`seed` A random seed for the simulation.

`print_activity` At the end of a simulation using iq_router, print out the activity for buffer, switch, and channel of the network.

`viewer_trace` The simulator will generate very verbose print out of all activity inside the network. This print out should be fed into noc_viewer for a graphic display of the activity inside the network. Currently not working.

`watch_file` Specific flits can have their "watch" status turn on. Require input a file which has flit id listed. 1 id per line.

A Random number generation

The simulator uses Knuth's integer and floating point pseudorandom number generators. These algorithms and their explanations appear in "The Art of Computer Programming: Seminumerical Algorithms".