BookSim 1.0 User's Guide

Brian Towles and William J. Dally

September 10, 2004

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 version of the simulator is available from `http://cva.stanford.edu` as a compressed tar archive. UNIX/Linux users can extract this archive using the tar utility

```
tar xvfz booksim-1.0.tar.gz
```

Windows users can use a compression program such as WinZip to extract the archive.

The simulator itself is written in C++ and has been specifically tested with GNU's G++ compiler (version $\geq$ 3). In addition, both a LEX and YACC tool (also known as FLEX and BISON) are needed to create the configuration parser. These are standard tools in any UNIX/Linux development environment. It is suggested that Windows users download the CYGWIN versions (`http://www.cygwin.com`) of these UNIX development tools to simplify their compilation process. The `Makefile` should be edited so that the first lines give the paths to the tools. At Stanford, for example, the compiler, YACC, and LEX are stored in the `/usr/pubsw/bin` directory. The `Makefile` reflects this:

```
CPP     = /usr/pubsw/bin/g++
YACC    = /usr/pubsw/bin/byacc -d
LEX     = /usr/pubsw/bin/flex
```

Then, the simulator can be compiled by running `make` in the directory that contains the `Makefile`.

2.2 Running a simulation

The syntax of the simulator is simply

```
booksim [configfile]
```

The optional 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 configuration is stored in `examples/torus88`.

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, to ensure deadlock-freedom of this routing function in the torus, two virtual channels are required. The `injection_rate` parameter is added to tell the simulator to inject 0.15 flits per simulation cycle per node. Because the simulator operates at the flit level, most parameters are specified in units of flits as is the case with the `injection_rate`. Also, any line of the configuration that begins with `//` is treated as a comment and ignored by the simulator. A detailed list of configuration parameters is given in Section 4.

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

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

// Routing
routing_function = dim_order;

// Flow control
num_vcs = 2;

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

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

```
%=================================
% Average latency = 6.02008
% Accepted packets = 0.11 at node 52 (avg = 0.147094)
% latency change    = 1
% throughput change = 1

...

% Warmed up ...
%=================================
% Average latency = 6.0796
% Accepted packets = 0.119 at node 5 (avg = 0.148266)
% latency change    = 0.00562457
% throughput change = 0.00379387

...

% Draining all recorded packets ...
% Draining remaining packets ...
====== Traffic class 0 ======
Overall average latency = 6.09083 (1 samples)
Overall average accepted rate = 0.149475 (1 samples)
Overall min accepted rate = 0.138551 (1 samples)
```

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

by the lowest throughput of all the destination in the network, but the average throughput is also displayed.

After the warm up periods have passed, 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 4.7.

3 Examples

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 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.

Figure 4 shows a configuration file that can be used to determine the distribution of packet latencies in a 2-ary 6-fly network that uses age-based arbitration. Note the use of the `priority` configuration parameter along with the `select` allocators that account for packet priorities. The simulator does not output latency distributions by default, but by editing `trafficmanager.cpp`, setting the configuration variable `DISPLAY_LAT_DIST` to true, and recompiling, the distribution will be displayed at the end of the simulation. This technique was used to produced the distribution shown in Figure 25.12 of PPIN.

As a final example, Figure 5 shows the use of the special single-node topology to test the performance of a switch allocator — in this case, the iSLIP allocator. The `in_ports` and `out_ports` options set up a simulation of an 8×8 crossbar.

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 and the simulator supports four basic topologies:

`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.

```
// 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 = 0.1;
```

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

```
// Topology

topology = fly;
k = 2;
n = 6;

// Routing

routing_function = dest_tag;

// Flow control

num_vcs     = 8;
vc_buf_size = 8;

wait_for_tail_credit = 1;

// Router architecture

vc_allocator = select;
sw_allocator = select;
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                = uniform;
const_flits_per_packet = 20;
priority               = age;

// Simulation

sim_type       = latency;
injection_rate = 0.1;
```

Figure 4: A configuration file (`examples/fly26_age`) for finding the distribution of packet latencies using age-based arbitration.

```
// Topology

topology  = single;
in_ports  = 8;
out_ports = 8;

// Routing

routing_function = single;

// Flow control

vc_allocator = islip;
sw_allocator = islip;
alloc_iters  = 2;

num_vcs     = 8;
vc_buf_size = 1000;

wait_for_tail_credit = 0;

// Simulation

sim_type       = latency;
injection_rate = 0.1;
```

Figure 5: A single-node configuration file (`examples/single`) for testing the performance of a switch allocator.

`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.

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 4.2).

4.2 Routing algorithms

The `routing_function` parameter selects a routing algorithm for the topology. Many routing algorithms need multiple virtual channels for deadlock freedom (VCDF).

`dim_order` Dimension-order routing. Works for the `mesh` topology (1 VCDF) and for the `torus` topology (2 VCDF).

`dim_order_bal` Dimension-order routing for the `torus` topology with a more balanced use of VCs to avoid deadlock (2 VCDF).

`dim_order_ni` A non-interfering version of dimension-order routing. Works on the `torus` or `mesh` topology and requires one VC per network terminal.

`min_adapt` A minimal adaptive routing algorithm for the `mesh` topology (2 VCDF) and for the `torus` topology (3 VCDF).

`planar_adapt` Planar-adaptive routing for the `mesh` topology (2 VCDF). Supports routing around failed channels.

`romm` ROMM routing for the `mesh` (2 VCDF). Load is balanced by routing in two phases: one from the source to a random intermediate node in the minimal quadrant and a second from the intermediate to the destination.

`romm_ni` A non-interfering version of ROMM routing for the `mesh` that requires one VC per network terminal.

`single` A dummy routing function used for the `single` topology.

`valiant` Valiant's randomized routing algorithm for the `mesh` (2 VCDF) and `torus` (4 VCDF) topology.

`valiant_ni` A non-interfering version of Valiant's algorithm for the `torus` that requires 4 VCs per network terminal.

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).

4.3 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 in flits.

`voq` If non-zero, use virtual-output queuing. With virtual output queuing, a separate virtual channel is assigned to each destination in the network. This option is most useful when used with a non-interfering routing algorithm (Section 4.2).

`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.

4.4 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.

4.4.1 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.

`sw_allocator` The type of allocator used for switch allocation. See Section 4.5 for a list of the possible allocators.

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

`vc_allocator` The type of allocator used for virtual-channel allocation. See Section 4.5 for a list of the possible allocators.

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

4.4.2 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.

4.5 Allocators

Many of the allocators used in the simulator are configurable (see the input-queued router in Section 4.4.1) 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 matching.

`select` Priority-based allocator. Allocation is performed as in iSLIP, but with preference towards higher priority packets (see `priority` option in Section 4.6).

Allocation can also be improved by performing multiple iterations of the algorithm and the number of iterations is controlled by the `alloc_iters` parameter.

4.6 Traffic

The rate at which flits 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 flits per flit cycle. For example, setting `injection_rate = 0.25` means that each source injects a new flit one of every four simulator cycles. The injection process can also be specified as either Bernoulli (`injection_process = bernoulli`) or an on-off process (`injection_process = on_off`). The burstiness of the latter injection process is controlled via the `burst_alpha` and `burst_beta` parameter. See PPIN Section 24.2.2 for a description of the on-off process and its parameters.

The unit of injection is packets, which may be comprised of many flits. The number of flits per packet is set using the `const_flits_per_packet` option. Each packet may also have an associated priority, either age-based (`age`) or none (`none`), as specified by the `priority` option.

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 permutation while a fixed value of `perm_seed` allows the same permutation to be used for several experiments.

4.7 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.

`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.

`max_samples` The total length of simulation expressed as a multiple of the `sample_period`.

`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.

`reorder` A non-zero value indicates that packet order should be maintained and reordering time is accounted for in the overall latency.

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".