3.2 Built gem5/Garnet2.0 extensions

Work started on the latest gem5 source code[4] as of April 27, 2018. The extensions listed below refer to this version of gem5 and Garnet2.0.

3.2.1 Added framework assist scripts

`./buildgarnet`: The Garnet_standalone simulation binaries (for use with the Garnet synthetic traffic injector) can be built by running this Bash script, which executes: `scons -j`*n* `build/NULL/gem5.debug PROTOCOL= Garnet_standalone` where *n* is the number of cores on the host machine.

`./buildx86`: The x86 simulation binaries (for use with both system-call emulation mode and full-system mode) can be built by running this Bash script, which executes: `scons -j`*n* `build/X86_MESI_Two_Level/gem5.fast PROTOCOL=MESI_Two_Level` where *n* is the number of cores on the host machine. The simulations will employ two cache levels with the MESI coherence protocol.

`./rungarnet`: The Garnet_standalone binaries and added extensions can be executed more conveniently by running this Bash script. The script calculates topology-specific parameters and generates a uniquely formatted output directory name within the `m5out` directory. Some parameters can be supplied from command line arguments; others have to be modified by editing the script. The script prints usage information by running it with no arguments.

`./runfft`: Runs the Splash2 FFT benchmark [38] in gem5 system-call emulation mode with a Garnet2.0 network, which can be specified in similar fashion to `./rungarnet`. Problem size M can be specified as the fifth parameter and clock frequency for all CPUs can be specified as the sixth parameter. The FFT benchmark will use 2^M complex doubles as data points. M must be an even number.

`./rundsent`: Takes one or more arguments, each of which should specify a simulation output directory. For each output directory, the script runs `util/on-chip-network-power-area-2.0.py` and writes the output to `dsent_out.txt` within the output directory.

`./grepnetworkstats.py`: Automatically run by `./rungarnet` and `./runfft`. Copies the relevant network statistics from the simulation's `stats.txt` to `network_stats.txt`.

`./grepdebug.py`: If `GARNETDEBUG=1` is set in `rungarnet`, is automatically run after a Garnet_standalone simulation. Lines matching the specified `grepdebug.py:greps[]` are copied from `debug.txt` to `debug_parsed.txt`. Optionally, a sorted output is written to `debug_parsed_sorted.txt`.

`./plotlatencythroughput.py`: Takes one argument: the root directory containing multiple simulation output directories. For each injection rate, gathers the reception rate (throughput) and the average packet latency from the simulation's `stats.txt`. The three statistics are appended to respectively formatted `*-latencythroughput.txt` files within the specified root directory.

3.2.2 Splash2 FFT benchmark for SE-mode x86 simulations

Newly implemented in `configs/example/fft_benchmark.py`, borrowing code from gem5's included `configs/example/se.py` and `configs/splash2/run.py`. The Splash2 benchmarks require compilation with a Pthreads implementation of the PARMACS macros. Unable to get this to work, I resorted to copying the Splash2 benchmarks included with the Sniper simulator to the `sniper_splash2` directory. Defunct hooks used by Sniper were removed, after which the FFT benchmark is working at expected. The `./runfft` script provides a convenient command-line option to specify primary Garnet2.0 topology parameters, as well as the FFT benchmark's problem size and the simulated CPU clock frequency.

[4] URL to the gem5 repository: `https://github.com/gem5/gem5`; URL to the latest commit used: `https://github.com/gem5/gem5/commit/5187a24d496cd16bfe440f52ff0c45ab0e185306` – accessed May 19, 2018.

3.2.3 Topology visualization with LaTeX/TikZ

Newly implemented in `configs/topologies/TikzTopology.py`. If parameter `--tikz` is used, will write LaTeX/TikZ code for visualizing a topology to `topology.tex` in the simulation's output directory. Example visualizations are shown in the following section. Links with a weight of 1 get a thicker edge than links with higher weight (lower order), to emphasize dimension order routing. For flattened butterfly topologies with more than 64 routers, only the first 64 routers' edges are drawn, due to LaTeX memory limitations.

`./tex2png`: Automatically run from `./rungarnet` and `./runfft`. Converts `topology.tex` in the given output directory to PDF. If the `imagemagick` package is installed, the topology PDF is converted to `topology.png` as well. If `CLEAN_UP=1` is set, the LaTeX files will be removed, keeping only the PNG-file.

3.2.4 Topologies

A new parameter `--concentration-factor=`n was added, for all of the implemented topologies listed below. n denotes the number of CPUs per router. Each CPU is connected through one L1 cache controller node. The number of directory nodes can be specified with `--num-dirs=`d, with $d \leq$ the number of cache nodes. The directory nodes are distributed evenly across the routers. All DMA nodes are connected to the first router – Garnet_standalone does not employ any DMA nodes. Each topology is limited to two dimensions and each link is bidirectional.

Line

Newly implemented in `configs/topologies/Line.py`. Mimics a bus interconnect within a NoC. Constructed line topologies comprise a $1 \times n$ grid, where n is the number of routers.

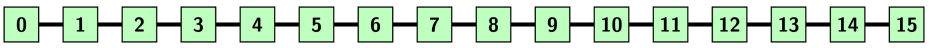

Figure 3.1: Diagram of the 16-router line topology.

Fully Connected

Newly implemented in `configs/topologies/FullyConnected.py`. Each router is connected to all other routers. Each link has the same dimension order routing weight of 1, since fully connected topologies inherently avoid deadlock [39]. Each link that is a straight edge is assigned a latency of 1 cycle and the same distance of l (square dimensions; 4 routers) or l' (odd dimensions; 8 routers and up) in DSENT link power and area modeling (see 3.2.5 on page 38). This implementation consequently restricts fully connected topologies to two mesh rows (grid rows). More elaborate implementations are forgone due to the inherent dire scalability of fully connected topologies, as well as time constraints. Each sloped link is assigned a latency of `ceil(proportional_distance)`, where `proportional_distance` is the distance proportional to a straight edge. Sloped links are set to distance of l or $l' \times$ `proportional_distance` in DSENT.

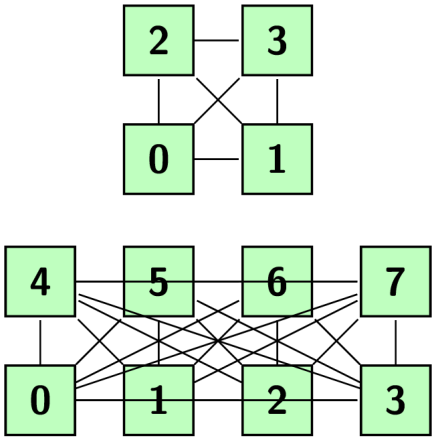

Figure 3.2: Diagrams of fully connected topologies for 4 and 8 routers. Each router is connected to all other routers. Links spanning more than two routers are omitted from the diagram. Each link has the same dimension order routing weight of 1, since fully connected topologies inherently avoid deadlock.

Ring

Newly implemented in `configs/topologies/Ring.py`. Constructed ring topologies comprise a $2 \times n$ grid, where n is half of the number of routers. The optional escape VC deadlock avoidance scheme, detailed in 3.2.7 on page 41, is not functioning correctly; many workloads will still deadlock. Therefore, this topology's implementation is deficient.

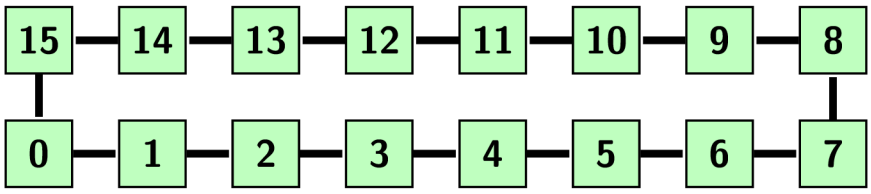

Figure 3.3: Diagram of the 16-router ring topology.

Hierarchical ring

Newly implemented in `configs/topologies/HierarchicalRing.py`. No microarchitectural differentiation between the central ring and any of the sub-rings is implemented in Garnet2.0. No deadlock avoidance scheme is available. Therefore, this topology's implementation is deficient. For configurations with a number of mesh rows (grid rows) greater than 4, the number of directory nodes is limited to the number of mesh rows and the number of cores is limited to 128, due to unresolved deadlock issues.

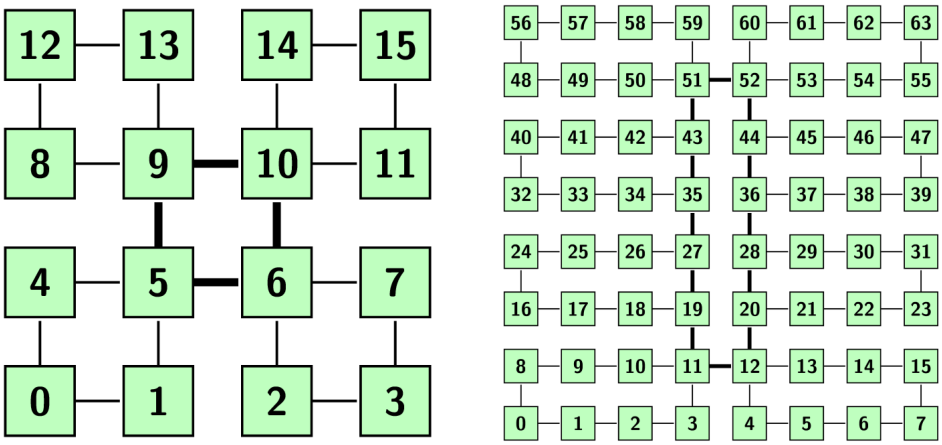

Figure 3.4: Diagrams of hierarchical ring topologies for 16 and 64 routers. The central ring connects multiple sub-rings.

Mesh and concentrated mesh

Mesh with XY dimension order routing is included with Garnet2.0 in `configs/topologies/Mesh_XY.py`. The topology's source code was modified to allow for a concentration factor, variable amounts of cache and directory nodes and TikZ visualization. XY dimension order routing is enforced by an assigned weight of 1 for each horizontal link and an assigned weight of 2 (lower precedence) for each vertical link.

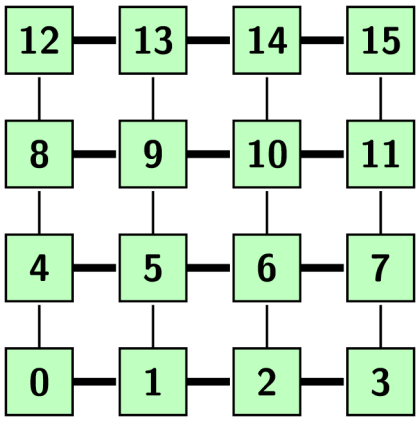

Figure 3.5: Diagram of the 16-router mesh topology. With a concentration factor of 4, the diagram shows a 16-router 64-core concentrated mesh topology.

Flattened butterfly

Newly implemented in `configs/topologies/FlattenedButterfly.py`. Each row and each column is fully connected. The flattened butterfly debut paper proposes a concentration factor of $n = 4$ for 64 cores [40]. For $n = 4$, each router has a radix of 10: each router connects to 3 neighboring routers in the x-dimension, 3 neighboring routers in the y-dimension and 4 cores. This implementation allows other concentration factors as well. The repeaters and pipeline registers for links connecting non-neighboring routers, suggested by the flattened butterfly paper, are not implemented. The repeaters and pipeline registers for these links suggested by the flattened butterfly paper are not implemented. Instead, the latency of these links is multiplied by the proportional distance between routers.

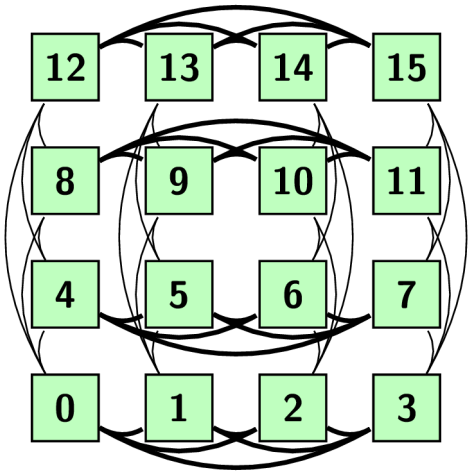

Figure 3.6: Diagram of the 16-router flattened butterfly topology. Each router can be connected to n CPUs, where n is the concentration factor. For the 4×4 2D mesh pictured, the flattened butterfly debut paper suggests concentration factors of $n = 4$ for 64 cores and $n = 8$ for 128 cores [40]. Links are drawn curved purely for visualization; the implementation assumes straight edges for all links.

3.2.5 DSENT – power and area modeling

NoC power and area models are generated by DSENT, version 0.91 (June 26, 2012), included in the gem5 commit referenced in footnote 4 on page 34.

`configs/topologies/TopologyToDSENT.py`: Contains a new class `TopologyToDSENT`, which is called within a topology's class. It writes topology-specific values to both a router and a link configuration file, `router.cfg` and `electrical-link.cfg` respectively, in the simulation's output directory. These values include the numbers of control buffers (pertaining to VNETs 0 and 1) and data buffers (pertaining to VNET 2), as well as the number of bits per flit. These configuration files serve as a scheme, to be later updated by `util/on-chip-network-power-area.py` for each individual router and link.

The router configuration file contains parameters and equations for calculating router power and area, based on statistics passed by the Python script. Similary, the link configuration file is used to calculate link power. To estimate the total area of a router, the release paper of ORION 2.0, a NoC power and area modeling tool, suggests taking the sum of the router's building blocks and adding 10% to account for global whitespace [41]. This factor is implemented in the router configuration file.

`util/on-chip-network-power-area-2.0.py`: Based on `util/on-chip-network-power-area.py`, which was outdated (2014) and included with gem5/Garnet2.0. The original script feeds limited Garnet1.0 simulation statistics to DSENT for modeling of NoC power and area. The resulting models assume a global injection rate for all routers and links. These limitations are eliminated in the newly created version 2.0 script, which is geared towards parsing of Garnet2.0 statistics. Passing of router- and link-specific arguments to DSENT was not functioning correctly for some arguments. This is fixed by instead running `sed -i` shell commands. The script is to be run with one argument: the simulation's output directory.

gem5's canonical implementation of DSENT defines the number of input ports of a router as the number of ports connecting unidirectional internal links (interconnecting a pair of routers). Similarly, the number of output ports are defined as the number of ports connecting unidirectional external links (between a router and NI-connected cache and directory controllers). The discrepancy with Garnet 2.0's bidirectional external links is accounted for. As explained in Section 2.11.1 on page 29, credit links carry VC buffer credits only and their influence on power and area results is deemed negligible. Therefore, the links considered in DSENT power and area modeling are limited to network links.

The injection rate in gem5's canonical implementation of DSENT is defined as the number of flits per cycle per port or link, in contrast to the Garnet synthetic traffic injector, which defines it as the number of packets per node per cycle. Injection rates for the flit buffers, crossbar and switch allocator are updated for each router, based on simulation statistics. Similarly, both the injection rate and wire length for each

link are updated. Individual injection rates are used in calculation of dynamic power.

DSENT includes four electrical technology models, all of which are LVT (Low Voltage Threshold): “bulk”, at 45 nm, 32 nm and 22 nm process sizes and tri-gate (multi-gate), at a 11 nm process size. The 11 nm tri-gate technology model is used in all experiments detailed in Chapter 4 on page 42, since this model represents the most novel lithography.

DSENT returns dynamic power, leakage power and area statistics for each building block of a router. The primary contributing building blocks of a router are the flit buffers, crossbar switch, switch allocator and clock distributor. Simply adding the total area for each of the routers together would not illustrate topology-specific influences on the CPU die area. `on-chip-network-power-area-2.0.py` features a new extension to calculate the approximate CPU die area, which is presented in the following paragraphs.

Simulation core count	**Scale to model**	**Model**	**Model core count**	**Model die size (mm^2)**
1-4	No	14nm Skylake-D Server (quad-core)	4	122.6
5-10	Yes	14nm Skylake Server (LCC)	10	325.44
11-12	No	""	""	""
13-20	Yes	14nm Skylake Server (HCC)	18	485.0
21-28	Yes	14nm Skylake Server (XCC)	28	694.0
29-63	No	""	""	""
64-76	Yes	14nm Knights Landing (XCC)	76	682.6

Table 3.1: Five existing Intel x86 server CPUs used as comparative models in DSENT core area estimation. If the simulated CPU die size is not scaled to the model die size, the model die size is simply adopted.

For five existing 14 nm Intel x86 server CPUs, the die area in mm^2 and the number of cores are defined in the `getCoreAreaForCoreCount(num_cpus)` function, as shown in Table 3.1. Since DSENT does not include a 14 nm technology model and the comparative die size models are used merely for a rough estimation of CPU area, differences from the tri-gate technology model are not accounted for. For numbers of simulated cores, “num_cpus”, that are congruent with existing server CPUs, the die size is assumed to scale linearly with core count and is therefore calculated by (3.1). For other numbers of simulated cores, the die size is not scaled and the model die size is used. Thus, CPU area for simulations with a much larger core count than 76 is assumed to scale down significantly, corresponding to the continued shrinking of future process sizes.

$$\text{proportional\_die\_size} = \text{model\_die\_size} \times \frac{\text{num\_cpus}}{\text{model\_core\_count}} \tag{3.1}$$

The estimated area of a single CPU in m^2 is then calculated by (3.2):

$$\text{cpu\_area} = \frac{\text{proportional\_die\_size} \times 10^{-6}}{\text{num\_cpus}} \tag{3.2}$$

The uncore part of the die, including NoC(s), is assumed to take up 30% of the CPU area. Therefore, the area of a single core is set to (3.3):

$$\text{core\_area} = 0.7 \times \text{cpu\_area} \tag{3.3}$$

Cache and directory controllers are assumed to be located at a 45° angle from the router at a distance of $0.1\times$ the square root of the core area. This takes into account a maximum concentration factor of 4 CPUs per router. Thus, the external link (interconnecting router and CPU through an NI) wire lengths k are set to (3.4):

$$k = 0.1 \times \sqrt{\text{core\_area}} \tag{3.4}$$

The lateral space between a router and a CPU is therefore $\frac{k}{\sqrt{2}}$.

The area of the silicon die used is calculated by (3.5):

$$\begin{aligned}\text{Area} =&(\text{nrows} \times (\frac{k}{\sqrt{2}} + \sqrt{\text{router\_area}}) + \text{num\_vertical\_cpus} \times \sqrt{\text{core\_area}}) \\ &\times (\text{ncols} \times (\frac{k}{\sqrt{2}} + \sqrt{\text{router\_area}}) + \text{num\_horizontal\_cpus} \times \sqrt{\text{core\_area}})\end{aligned} \tag{3.5}$$

where "nrows" is the number of rows in the topology, "num_vertical_cpus" is the number of CPUs placed along the y-axis, "ncols" is the number of columns in the topology, "num_horizontal_cpus" is the number of CPUs placed along the x-axis and "ncpus" is the number of CPUs in the mesh. For concentrated meshes two CPUs per router are placed along the y-axis. For square mesh-based topologies (3.5) simplifies to:

$$\text{Mesh\_area} = (\text{nrows} \times (\frac{k}{\sqrt{2}} + \sqrt{\text{router\_area}}) + \sqrt{\text{ncpus}} \times \sqrt{\text{core\_area}}))^2 \tag{3.6}$$

For square mesh-based topologies internal link (interconnecting a pair of routers) wire lengths l are set using (2.5):

$$l = \frac{\sqrt{\text{Mesh\_area}}}{\sqrt{\text{ncpus}} - 1} \tag{2.5 reiterated}$$

For odd-dimensional topologies internal link wire lengths l' are set using (3.7):

$$l' = \frac{\text{Area\_xymax}}{\text{cpus\_xymax} - 1} \tag{3.7}$$

where "Area_xymax" = `max`(y-component of Area, x-component of Area) and "cpus_xymax" = `max`(num_vertical_cpus, num_horizontal_cpus). Out of all considered topologies, only the flattened butterfly has internal links that span non-neighboring routers. The wire lengths of these links are set to $n \times l$, where n is the proportional distance between routers. The repeaters and pipeline registers for these links suggested by the flattened butterfly paper are not implemented.

Since wire length calculation requires the router area, the link power results rely on the router area results. Wire delay is set according to 2011 International Technology Roadmap for Semiconductors (ITRS) projections [2], as interpreted by Al Khanjari and Vanderbauwhede [42]. For core counts up to 76, 14 nm CMOS is assumed, for which ITRS projected a global wire delay of 1 ns/mm, which amounts to one clock cycle per millimeter of wire for a 1 GHz CPU frequency. For core counts greater than 76, future many-core 10 nm architectures are assumed, for which ITRS projected a global wire delay of 33.8 ns/mm.

`ext/dsent/interface.cc`: Included with gem5/Garnet2.0. This file is fed Garnet2.0 simulation statistics by `util/on-chip-network-power-area*.py`. It was modified to accomodate the new `util/on-chip-network-power-area-2.0.py`.

3.2.6 Routing algorithms

Implemented in `src/mem/ruby/network/garnet2.0/RoutingUnit.cc`. gem5/Garnet2.0 comes with the following routing algorithms, which can be chosen by parameter `--routing-algorithm=`*i*:

0: Table based routing. Calculates the shortest paths between nodes. In case of multiple possible paths, the path with the minimum weight is chosen. In case of multiple equally weighted paths, one of the paths is chosen at random.

1: XY routing for mesh topology. Will always take the shortest path. Forces a flit to always take an x-direction path firstly and a y-direction secondly.

Added routing algorithm parameters are:

2: Random routing. Ignores shortest paths and instead chooses a random possible direction for the flit. Will deadlock quickly in most trials.

3: Adaptive routing. Not implemented thusfar.

3.2.7 Deadlock avoidance

Ring topology – escape VC

Attempted implementation in `src/mem/ruby/network/garnet2.0/OutputUnit.cc`. Required for a ring topology to not deadlock. Unfortunately, it is not functioning correctly, as many workloads will still deadlock. If parameter `--escapevc` is used, VC 0 will be used as an escape VC. Requires a number of VC's ≥ 2. Requires a shortest path routing algorithm. VC 0 will provide an acyclic escape path, while other VC's are allowed to be cyclic. Flits requesting to cross the eastmost vertical link on the ring are prohibited access to VC 0.

Hierarchical ring topology

Unfortunately hierarchical rings with a number of mesh rows (grid rows) greater than 4 suffer unresolved deadlock issues. For these topologies, the number of directory nodes is limited to the number of mesh rows and the number of cores is limited to 128.