Sandia National Laboratories

Exceptional service in the national interest

The Structural Simulation Toolkit (SST)

ISPASS 2024

Presented by: Patrick Lavin and Joseph Kenny
Sandia National Laboratories

Content by: SST Team

Indianapolis, Indiana

Sunday, May 5

SAND2024-02470C

Instructors

Patrick Lavin prlavin@sandia.gov

Joseph Kenny jpkenny@sandia.gov

Welcome!

Part 1: Introduction to SST	***8:00 – 9:30***
SST Overview	
Basic Simulation in SST	
A Tour of SST Elements	
Break	*9:30 – 9:45*
Part 2: Advanced Node-Level Simulation	*9:45 – 10:30*
Part 3: System-Scale Simulation	*10:30 - 12:00*

Learning Objectives – Part 1: Introduction to SST

This section of the tutorial will cover the following topics:

1. The basic structure of the SST project
2. How to run a simulation in SST
3. How to view the statistics from your simulation
4. How to find information about components with sst-info
5. A summary of the many available components
6. Where to get more info and help

Codespaces

- We will use Github Codespaces for this tutorial.
- Everyone with a Github account receives 180 free core hours and 15 free gigabytes of storage every month.
- https://github.com/sstsimulator/sst-tutorials
- Click “Code”
  - ”Codespaces”
  - “…”
  - “New with options”
  - Dev container configuration -> “ispass2024”
  - “Create codespace”
- Alternate instructions are available at https://github.com/sstsimulator/sst-tutorials/tree/master/ispass2024
- You may also want to open https://github.com/sstsimulator/sst-elements in another tab

Container

- The codespace automatically loads our container from Dockerhub
  - Includes SST, and the simulation components required for this tutorial

- The codespace also includes you a copy of the sst-tutorials repository, so you can edit the files we will work on

- Try it out: `sst`
- We didn’t give it an input file so it can’t simulate anything yet!

References

Websites

- Information on installing SST, and links to everything else you see here
  - http://www.sst-simulator.org/
- Documentation on SST's key interfaces, overview of all the elements, and more!
  - http://sst-simulator.org/sst-docs/
- Code
  - https://github.com/sstsimulator

Important Pages

- Configuration File Format: http://sst-simulator.org/SSTPages/SSTUserPythonFileFormat/
- Doxygen Documentation: *http://sst-simulator.org/SSTDoxygen/13.0.0_docs/html/*
- Developer FAQ: http://sst-simulator.org/SSTPages/SSTTopDocDeveloperInfo/
- Building SST
  - Quick start: http://sst-simulator.org/SSTPages/SSTBuildAndInstall_13dot1dot0_SeriesQuickStart/
  - Detailed: http://sst-simulator.org/SSTPages/SSTBuildAndInstall_13dot1dot0_SeriesDetailedBuildInstructions/

Part 1:
Introduction to SST

SST Overview

Motivation: Interoperability

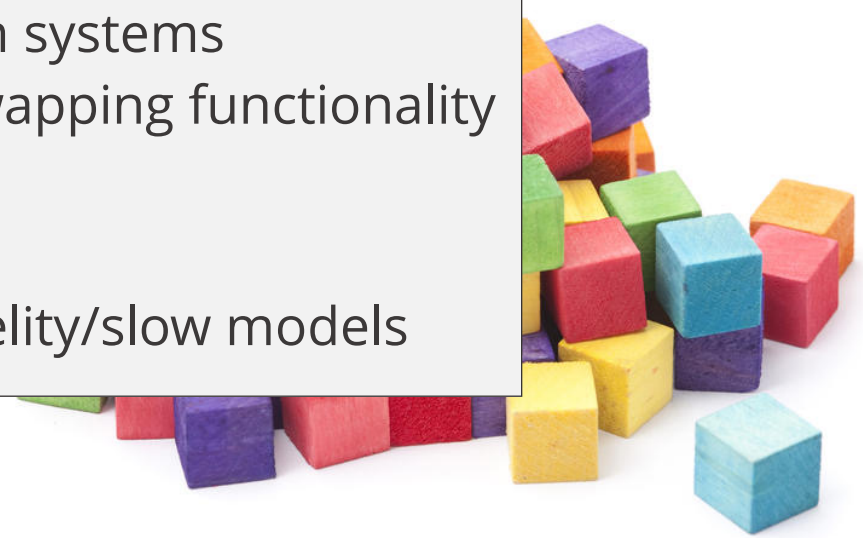
There is a rich selection of open-source simulators

But not a solid ecosystem for modeling systems

- Tightly-entangled components make modifications complex
  - E.g., assumptions about caching or address mapping are pervasive
- Most simulator integrations are ad-hoc, not lasting
- Significant performance problems with tying many simulators together

Wants:

- Enable “mix-and-match” of existing models to create custom systems
- Encourage disentangled models with clean interfaces for swapping functionality
  - Bricks not buildings
- Low effort, high performance parallel simulation
- Continuous path from low-fidelity/fast modeling to high-fidelity/slow models



The Structural Simulation Toolkit

Goals

- Create a standard architectural *simulation framework* for HPC*
- Ability to evaluate future systems on DOE/DOD workloads
- *Use supercomputers to design supercomputers*

Technical Approach

- **Parallel** Discrete Event core
  - With conservative optimization over MPI/Threads
- **Interoperability**
  - Node and system-scale models
- **Multi-scale**
  - Detailed (~cycle) and simple models that interoperate
- **Open**
  - Open Core, non-viral, modular

Status

- Parallel framework (*SST Core*)
- Integrated libraries of components (*Elements*)
- Current Release (13.1.0)
  - Two releases per year

Consortium

- "Best of Breed" simulation suite
- Used in industry, academia, and at labs

The SST Approach

Parallel Discrete-Event Simulator Framework (*SST Core*)

- Flexible framework enables multitude of custom “simulators”
- Demonstrated scaling to over 512 processors running a million+ components

Comes with many built-in simulation models (*SST Elements*)

- Processors, memories, networks

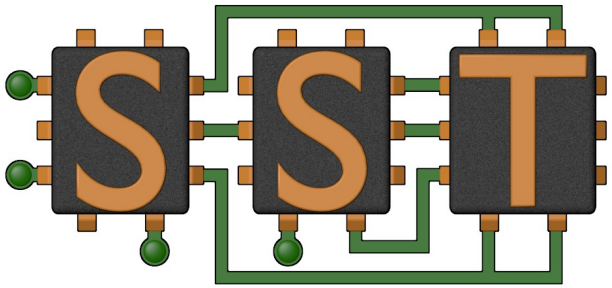
Open API

- Easily extensible with new models
- Modular framework
- Open-source core

Time-scale independent core

- Handles Micro-, Meso-, Macro-scale simulations

C++, Python



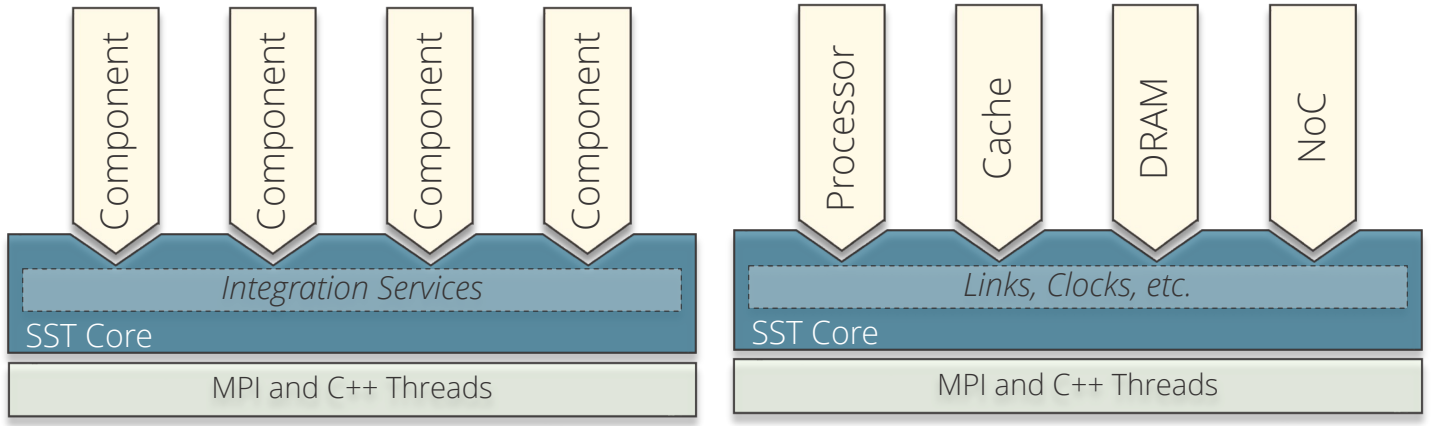
SST Architecture

SST **Core** framework

- The backbone of simulation
- Provides utilities and interfaces for simulation components (models)
  - Clocks, event exchange, statistics and parameter management, parallelism support, etc.

SST **Element** libraries

- Libraries of components that perform the actual simulation
- Elements include processors, memory, network, etc.
  - Includes many existing simulators: DRAMSim2, Spike, HMCSim, Ramulator, etc.



Building Blocks

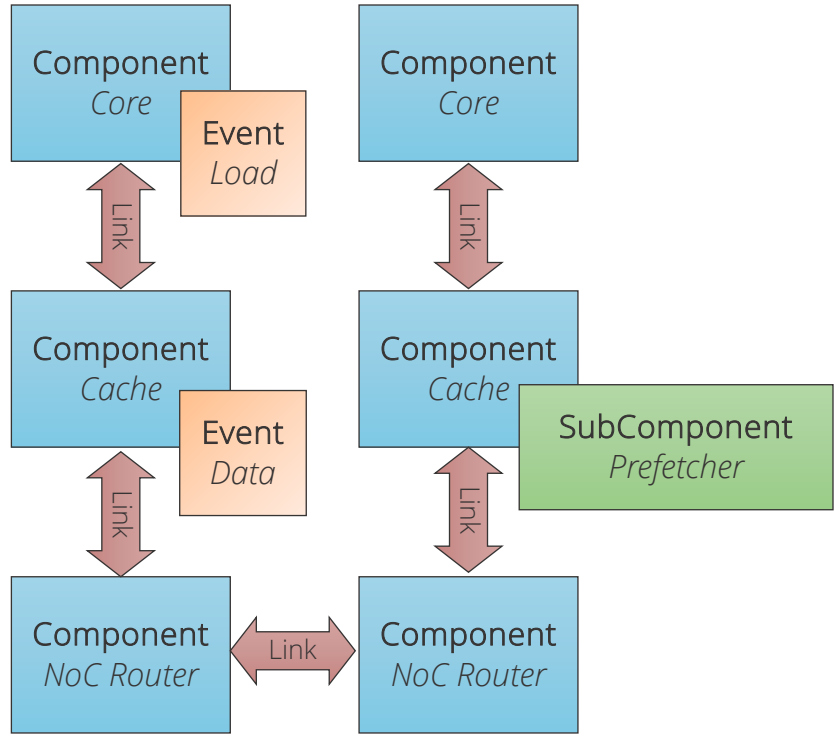
SST simulations are comprised of **components**

Components are connected by **links**

- Every link has a minimum (non-zero) latency
- Components define **ports** which are valid connection points for a link

Components communicate by sending **events** over the links

Components can use **subcomponents** and **modules** for customizable functionality



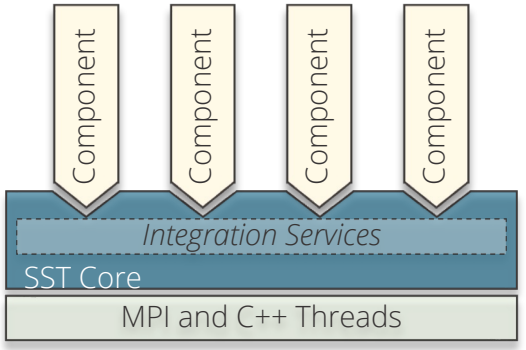
Element Library

A collection of components, subcomponents, and/or modules

SST Code Structure

SST Core and **SST Elements** are compiled separately

- Element libraries *register* with the core
- External elements (not part of SST Elements) can also be registered with the core
  - Example at github.com/sstsimulator/sst-external-element
- Core maintains a database of registered libraries
  - Can query database with *sst-info* utility



Source code for core:

- `sst-core/src/sst/core/`

Source code for elements

- `sst-elements/src/sst/elements/`
- Most elements have a tests/ directory
  - `sst-elements/src/sst/elements/SomeComponent/tests`
  - Often a good starting point for example configurations

Basic SST Simulation

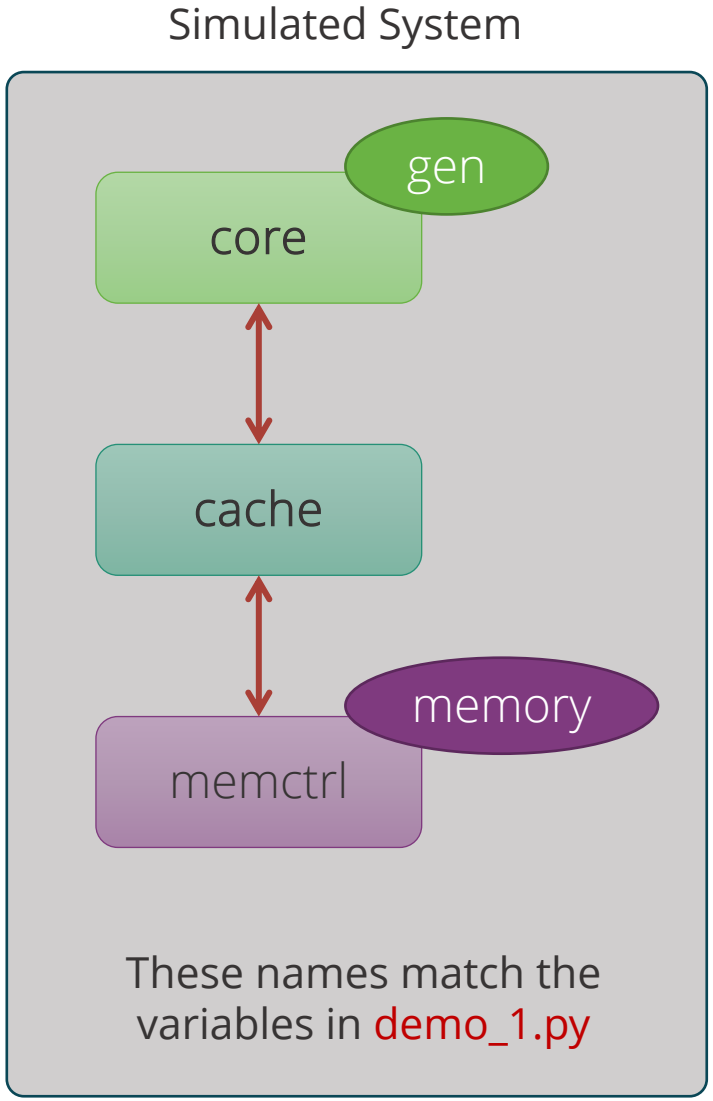
Our First Simulation – demo_1.py

We'll walk through how to configure a simulation and then run it

- Available at: (sst-tutorials → ispass2024 → exercises → single-node)

Element libraries in our example simulation

- **Miranda** - Simple core model that runs generated instruction streams
  - Generators produce memory access patterns (SubComponent)
- **memHierarchy –** Various cache/memory system related subcomponents and modules
  - Cache (Component) with coherence protocol SubComponent
  - Memory Controller (Component) that loads a memory timing model (SubComponent)



Configuration File: Global SST parameters

Set any global simulation parameters

SST Python API
User-defined string
SST argument

Other options

- Most command line options to SST are able to be set using setProgramOption()

```
import sst

##############################
## Define SST core options
##############################
# If this demo gets to 100ms, something
# has gone very wrong!
sst.setProgramOption("stop-at", "100ms")
```

Option	Definition
debug-file	File to print debug output to
heartbeat-period	If set, SST will print a heartbeat message at the specified period
print-timing-info	Tells SST to print timing information from the run
partitioner	Partitioner to use for parallel execution
output-partition	File to print partition to

Configuration File: Declare components

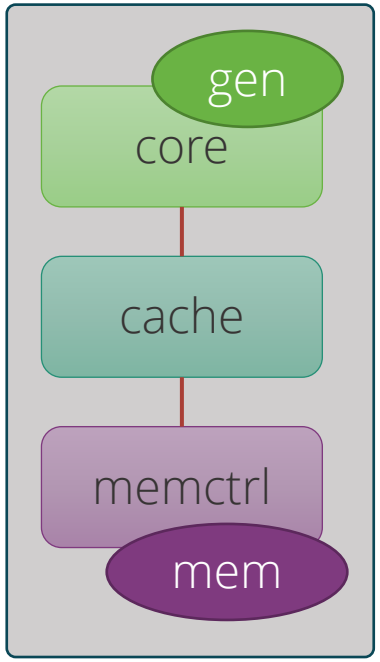
SST Python API
User-defined string
SST argument

Components: `sst.Component("name", "type")`

```
##########################
## Declare components
##########################
core = sst.Component("core", "miranda.BaseCPU")
cache = sst.Component("cache", "memHierarchy.Cache")
memctrl = sst.Component("memory", "memHierarchy.MemController")
```

Component name

Element library

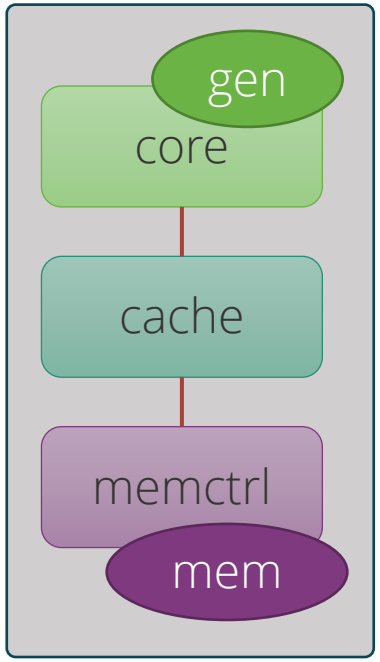


Configuration File: Configure the core

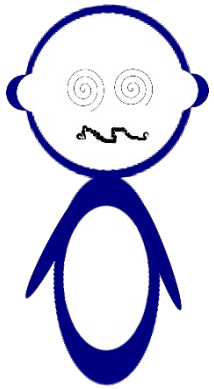
SST Python API
User-defined string
SST argument

Parameters: `addParams({ “parameter” : “value”, … })`

```
##########################################################
## Set component parameters and fill subcomponent slots
##########################################################
core.addParams({
    "clock" : "2.4GHz",
    "max_reqs_cycle" : 2,
})
```



How do I know what the options are?
Or even what elements I can pick from?



Aside: sst-info

Use sst-info to find information about all registered elements.

```
$ sst-info miranda.baseCPU

PROCESSED 1 .so (SST ELEMENT) FILES FOUND IN DIRECTORY(s)
[...]
================================================================================
ELEMENT LIBRARY 0 = miranda ()
Components (1 total)
   Component 0: BaseCPU
      Description: Creates a base Miranda CPU ready to execute an address
generator/access pattern
      ELI version: 0.9.0
      Compiled on: Dec  1 2023 14:33:14, using file: mirandaCPU.h
      Category: PROCESSOR COMPONENT
      Parameters (12 total)
         max_reqs_cycle: Maximum number of requests the CPU can issue per cycle
         (this is for all reads and writes)  [2]
         ...
```

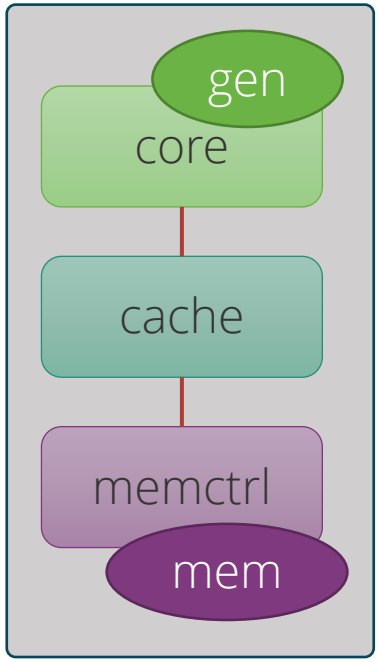
Configuration File: Add a subcomponent

SST Python API
User-defined string
SST argument

SubComponents: setSubComponent("slotname", "type")

- Recall: SubComponent is a *swappable piece of functionality*

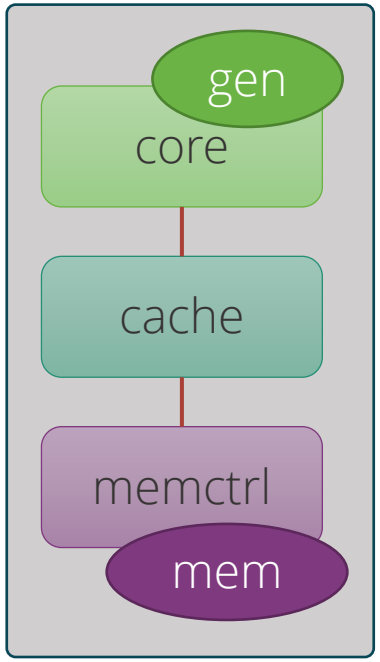
```
gen = core.setSubComponent("generator", "miranda.STREAMBenchGenerator")
gen.addParams({
    "n" : 1000, # Number of array elements
})
```



Configuration File: Configure the cache

SST Python API
User-defined string
SST argument

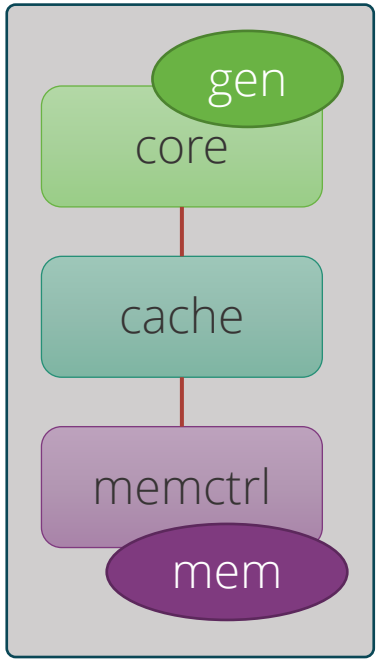
```
cache.addParams({
    "L1" : 1,
    "cache_frequency" : "2.4GHz",
    "access_latency_cycles" : 2,
    "cache_size" : "2KiB",
    "associativity" : 4,
    "replacement_policy" : "lru",
    "coherence_policy" : "MESI",
    "cache_line_size" : 64,
})
```



Configuration File: Configure the memory controller

SST Python API
User-defined string
SST argument

```
memctrl.addParams({
    "clock" : "1GHz",
    "backing" : "none", # No real memory values, just addresses
    "addr_range_end" : 1024*1024*1024-1,
})

memory = memctrl.setSubComponent("backend", "memHierarchy.simpleMem")
memory.addParams({
    "mem_size" : "1GiB",
    "access_time" : "50ns",
})
```

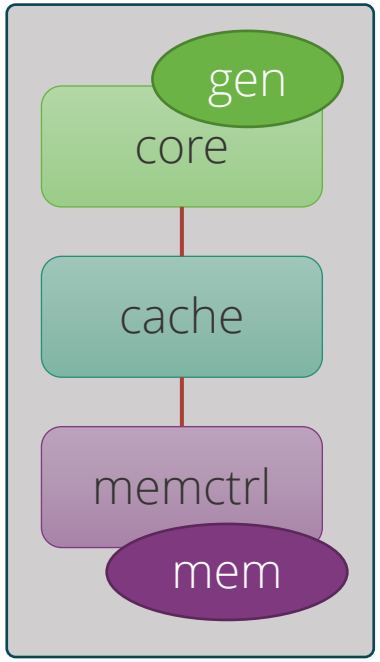


Configuration File: Declare and connect

Links: `sst.Link("name")`

```
#########################################
## Declare links
#########################################
core_cache = sst.Link("core_to_cache")
cache_mem = sst.Link("cache_to_memory")

#########################################
## Connect components with the links
#########################################
core_cache.connect( (core, "cache_link", "100ps"),
                    (cache, "high_network_0", "100ps") )

cache_mem.connect( (cache, "low_network_0", "100ps"),
                   (memctrl, "direct_link", "100ps") )
```

Link name



Configuration File: Connect links

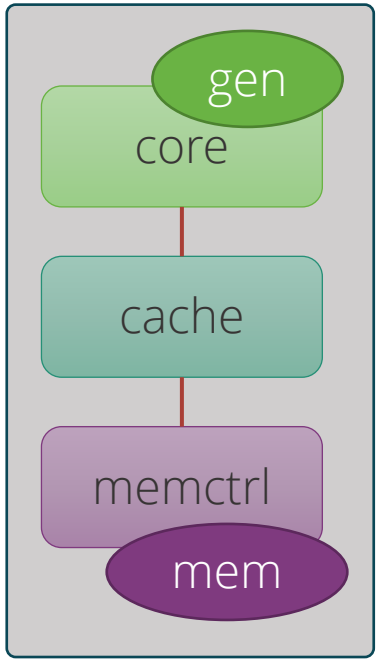
Connect components: `connect(endpoint1, endpoint2)`

- Where endpoint is: `(component, port, latency)`

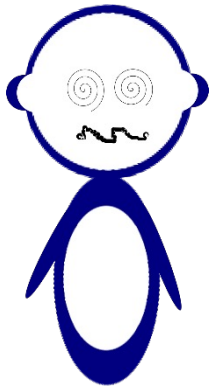
```
##########################################
## Connect components with the links
##########################################
core_cache.connect( (core, "cache_link", "100ps"),
                    (cache, "high_network_0", "100ps") )

cache_mem.connect( (cache, "low_network_0", "100ps"),
                   (memctrl, "direct_link", "100ps") )
```

Endpoint 1

Endpoint 2



How do I remember the port names?



SSTInfo: Getting component info

sst-info: utility to query element libraries

```
$ sst-info memHierarchy.Cache
PROCESSED 1 .so (SST ELEMENT) FILES FOUND IN DIRECTORY(s) /home/sst/build/lib/sst
Filtering output on Element = "memHierarchy.Cache"
========================================================================
ELEMENT LIBRARY 0 = memHierarchy ()
   Component 2: Cache
      Description: Cache controller
      Parameters (41 total)
         cache_frequency: (string) Clock frequency or period with units (Hz or s; SI units OK) [<required>]
         cache_line_size: (uint) Size of a cache line (aka cache block) in bytes.  [64]
…
      Ports (10 total)
         low_network_0: Port connected to lower level caches (closer to main memory)
…
      Statistics (43 total)
         TotalEventsReceived: Total number of events received,  (units = "events") Enable level = 1
…
```

Optionally filter for a specific library and/or component

Parameter

Definition

"REQUIRED" or default value

Port name

Definition

Name

Definition

Units

Enable Level

Running SST

Usage: `sst [options] configFile.py`

Common options:

`-h | --help`	Print complete list of command line options
`-v | --verbose`	Print information about core runtime
`--debug-file <filename>`	Send debugging output to specified file (default: sst_output)
`--partitioner <self | simple | rrobin | linear | lib.partitioner_name>`	Specify the partitioning mechanism for parallel runs
`-n | --num_threads <num>`	Specify number of threads per rank
`--model-options "<args>"`	Command line arguments to send to the Python configuration file. Any arguments after a final – will be appended to the model-options.
`--output-partition <filename>`	Write partitioning information to <filename>

`--output-dot <filename>`
`--output-xml <filename>`
`--output-json <filename>`
	Output the configuration graph in various formats to <filename>

Running a simulation

Launch simulation

```
$ sst demo_1.py
```

Output

```
Simulation is complete, simulated time:  6.80711 us
```

We probably want more information about what happened though

- Enable statistics!

Enabling statistics

Most Components and SubComponents define statistics

```
$ sst-info memHierarchy.Cache
    …
Statistics (43 total)
        TotalEventsReceived: Total number of events received,  (units = "events") Enable level = 1
        CacheHits: Total number of cache hits, (units = count) Enable Level = 1
        latency_GetS_hit: Latency for read hits, (units = cycles) Enable level = 1
```

Enable statistics in the configuration file

- `enableAllStatisticsForAllComponents()`
- `enableAllStatisticsForComponentType(type)`
- `enableAllStatisticsForComponentName(name)`
- `setStatisticLoadLevel(level)`
  - `enableStatisticForComponentName(name, stat)`
  - `enableStatisticForComponentType(type, stat)`

Configure output

- `setStatisticOutput("sst.output_type")`
- `setStatisticOutputOptions({"option" : "value", })`

Running with statistics enabled

Let's enable statistics for all components

- Caches have A LOT of statistics so send the output to a CSV file
- Other options: sst.statoutput*X* where X=
  - console
  - json
  - txt
  - hdf5

```
sst.setStatisticOutput("sst.statoutputcsv")

sst.setStatisticOutputOptions({ "filepath" : "stats.csv" })

sst.setStatisticLoadLevel(5)

sst.enableAllStatisticsForAllComponents()
```

Running a Simulation – Add Statistics

Copy configuration

```
$ cp demo_1.py demo_2.py
```

Add statistics to new configuration

```
What should you add?
```

Launch simulation

```
$ sst demo_2.py
```

Take a minute to look at the statistics

- Can you calculate the L1 memory bandwidth?

SST in parallel

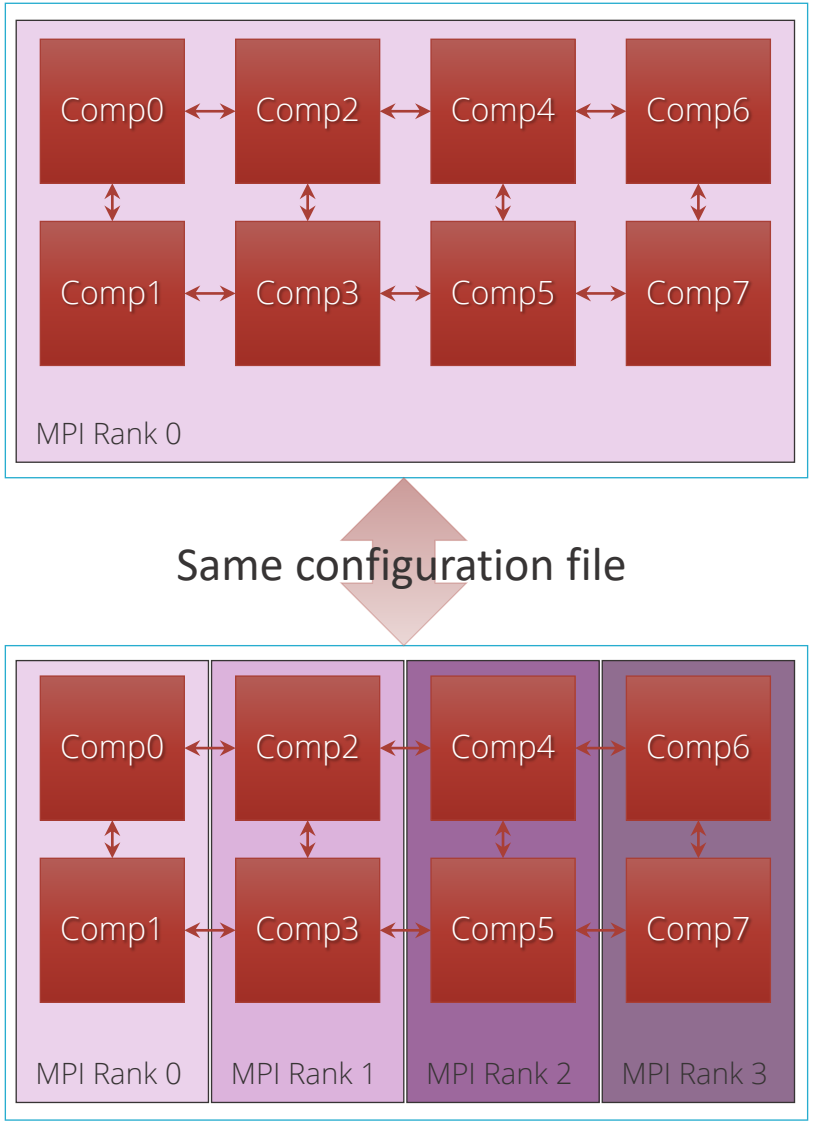
SST was designed from the ground up to enable scalable, parallel simulations

Components distributed among MPI ranks/threads

- Link latency controls synchronization rate

Sadly, MPI is not supported in our container

```
# Two ranks
$ mpirun -np 2 sst demo_1.py

# Two threads
$ sst -n 2 demo_1.py

# Two ranks with two threads each
# This will give a warning since we only
# have 3 components across 4 ranks/threads
$ mpirun -np 2 sst -n 2 demo_1.py
```



A Tour of SST Elements

SST Element Libraries

Elements are libraries of related components

- Elements must be *registered* with the SST core
  - Tells SST where to find this set of components
  - Includes information on parameters and statistics for each component

SST provides a set of element libraries

- Processor, network, memory, etc.
- Tested for interoperability within and across libraries
- Many are compatible with external “components” such as Ramulator and Spike

You can also register your own elements

- Example: https://github.com/sstsimulator/sst-external-element

(A few) SST 13.1 Elements

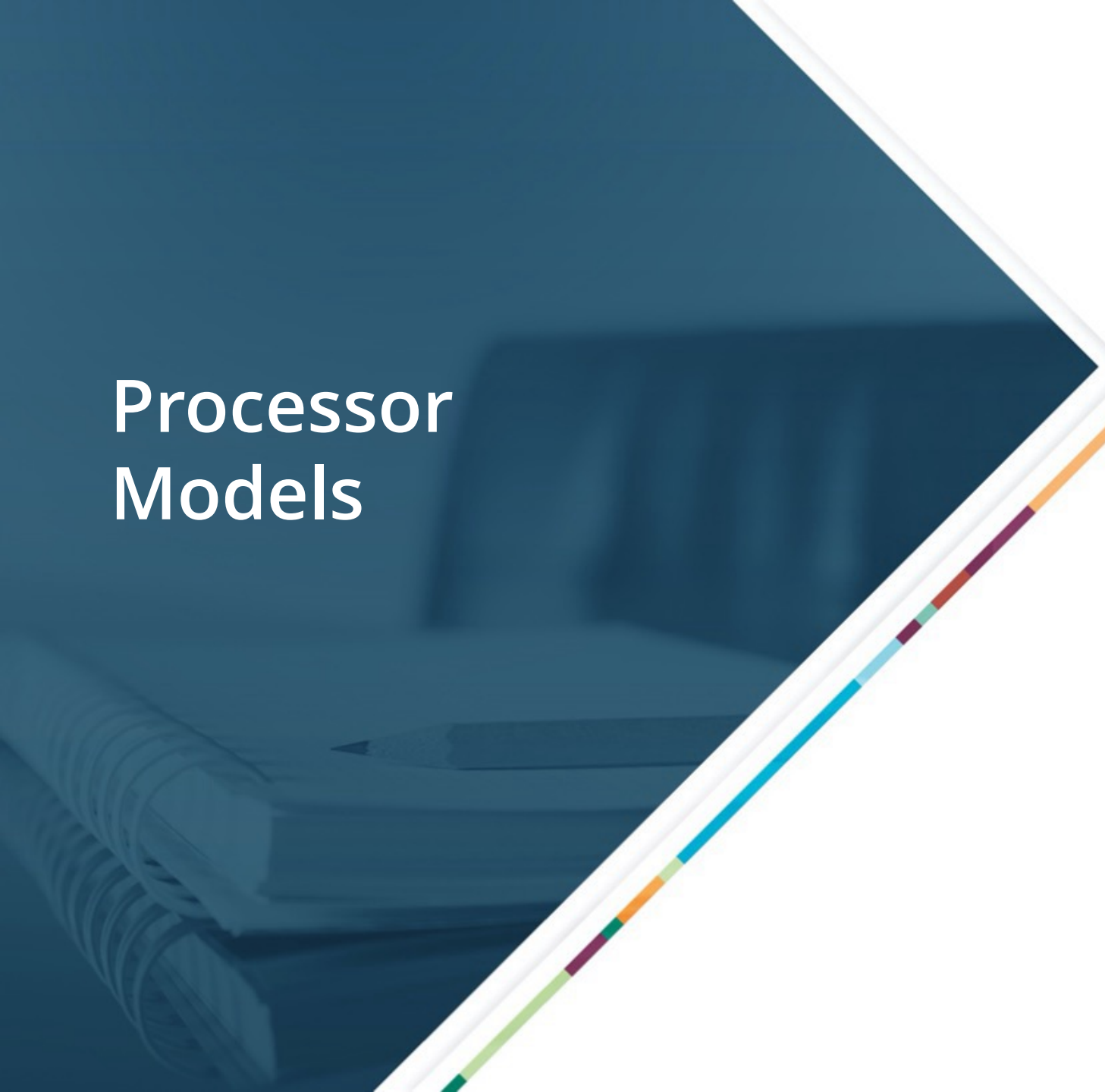
Processors
- **Ariel** – PIN-based
- **Prospero** – trace execution
- **Miranda** – pattern generator
- **Vanadis** – MIPS32 and RV64 pipeline

Memory Subsystem
- **MemHierarchy** – caches, directory, memory

Network drivers
- **Ember** – communication patterns
- **Firefly** – communication protocols
- **Hermes** – MPI-like driver interface

Networks/NoCs
- **Merlin** – flexible network modeling
- **Kingsley** – mesh NoC

Processor Models



Elements:

Ariel

Prospero

Miranda

Vanadis

Ariel: PIN-based processor

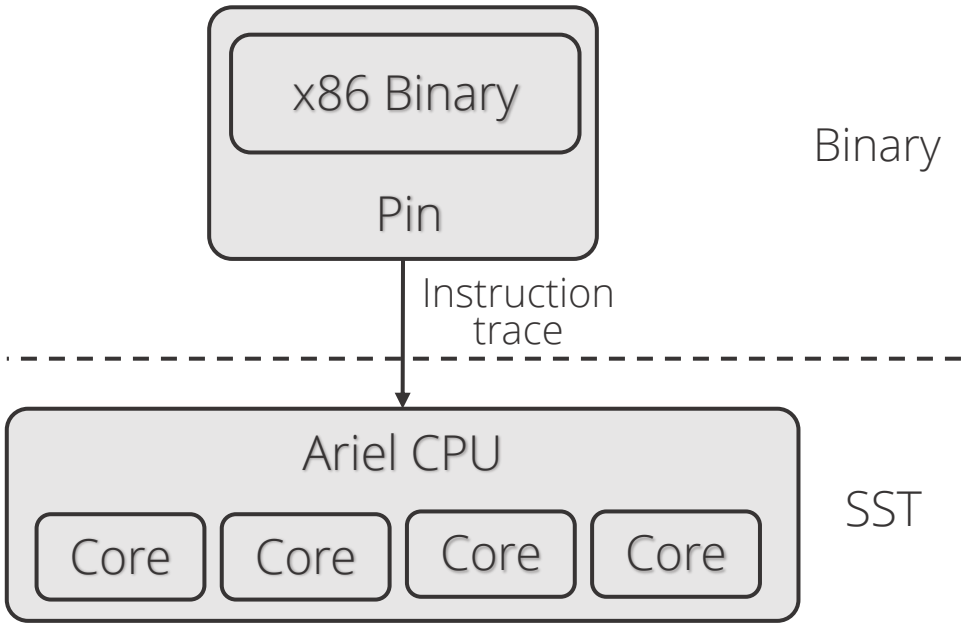
$ sst-info ariel.ariel

Lightweight processor core model

Uses Intel's PIN tools and XED decoders to analyze binaries

- Runs x86, x86-64, SSE/AVX, etc. binaries
- Supports fixed thread count parallelism (OpenMP, Qthreads, etc.)

Passes instructions to virtual core in SST



★ GPGPU-Sim Integration

Ariel: Details

$ sst-info ariel.ariel

Pintool communicates with Ariel via shared memory IPC

- Per-thread FIFO of instructions from pintool to Ariel's virtual cores
- Backpressure on FIFO halts the binary's execution

Ariel's virtual cores

- Memory instruction oriented: execute memory instructions; other ins. single cycle no-ops
- Clocked: Reads instruction stream in chunks but processes on clock
- Does *not* maintain dependence order or register locations
- Can map virtual-to-physical addresses internally or use external component

Key parameters

- Ops issued/cycle
- Load/store queue size

Uses SST standardMem interface

- Generates StandardMemRequests
- Compatible with memHierarchy

Ariel: The Tradeoff

$ sst-info ariel.ariel

Pros:

- Faster than more complex/pipeline models
- Reasonable approximation for studies on memory system performance
  - Especially for heavily memory-bound applications
- Reasonable model of thread interactions

Cons

- Non-deterministic results
  - Interactions between pintool, threads, etc.
  - Variation is low (O(1%))
- Not compatible with non-x86 binaries
- Reliant on Pin
  - Ongoing work to enable other frontends

Prospero: Trace-based processor

$ sst-info prospero.prosperoCPU

Trace-based processor model

- Like Ariel, memory instruction oriented
  - Reads memory ops from a file and passes to the simulated memory system
- “Single core” but can use multiple trace files to emulate threaded or MPI applications
- Supports arbitrary length reads to account for variable vector widths
- Performs “first touch” virtual to physical mapping

Comes with Prospero Trace Tool to generate traces

- Or can generate your own and translate to Prospero’s format

Prospero: The Tradeoff

$ sst-info prospero.prosperoCPU

Pros

- Faster than Ariel*
  - Provided you can get a trace

Cons

- Traces can be very large
  - Requires good I/O system to store and read the trace
- Traces are less flexible than actual execution
  - Capture a single execution stream using a single application input

Miranda: Pattern-based processor

$ sst-info miranda.BaseCPU

Extremely light-weight processor model

- Generates memory address patterns
- Supports request dependencies

Library patterns

- Strided accesses (single stream)
  - Forward and reverse strides
- Random accesses
- GUPS
- STREAM benchmark
  - In-order & out-of-order CPU
- 3D stencil
- Sparse matrix vector multiply (SpMV)
- Copy (~array copy)
- *Stake* interface to the Spike RiscV simulator
- Ongoing work to integrate Spatter patterns

Miranda: The tradeoffs

$ sst-info miranda.BaseCPU

Pros

- Very lightweight – no binary, no trace
- Good for applications whose address patterns are predictable
  - e.g., not much pointer-chasing
- Models instruction dependences

Cons

- Need a generator for the memory pattern of interest
  - Requires a good understanding of the pattern

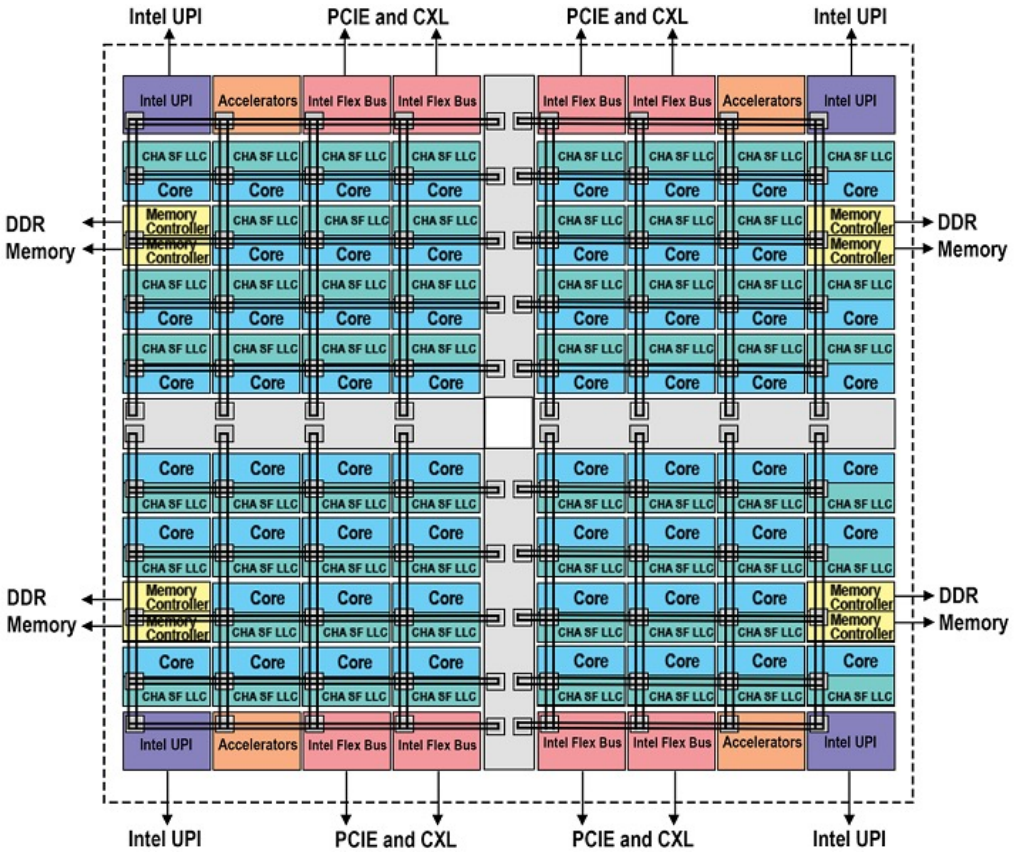
Vanadis: OOO Processor

$ sst-info vanadis.VanadisNodeOS

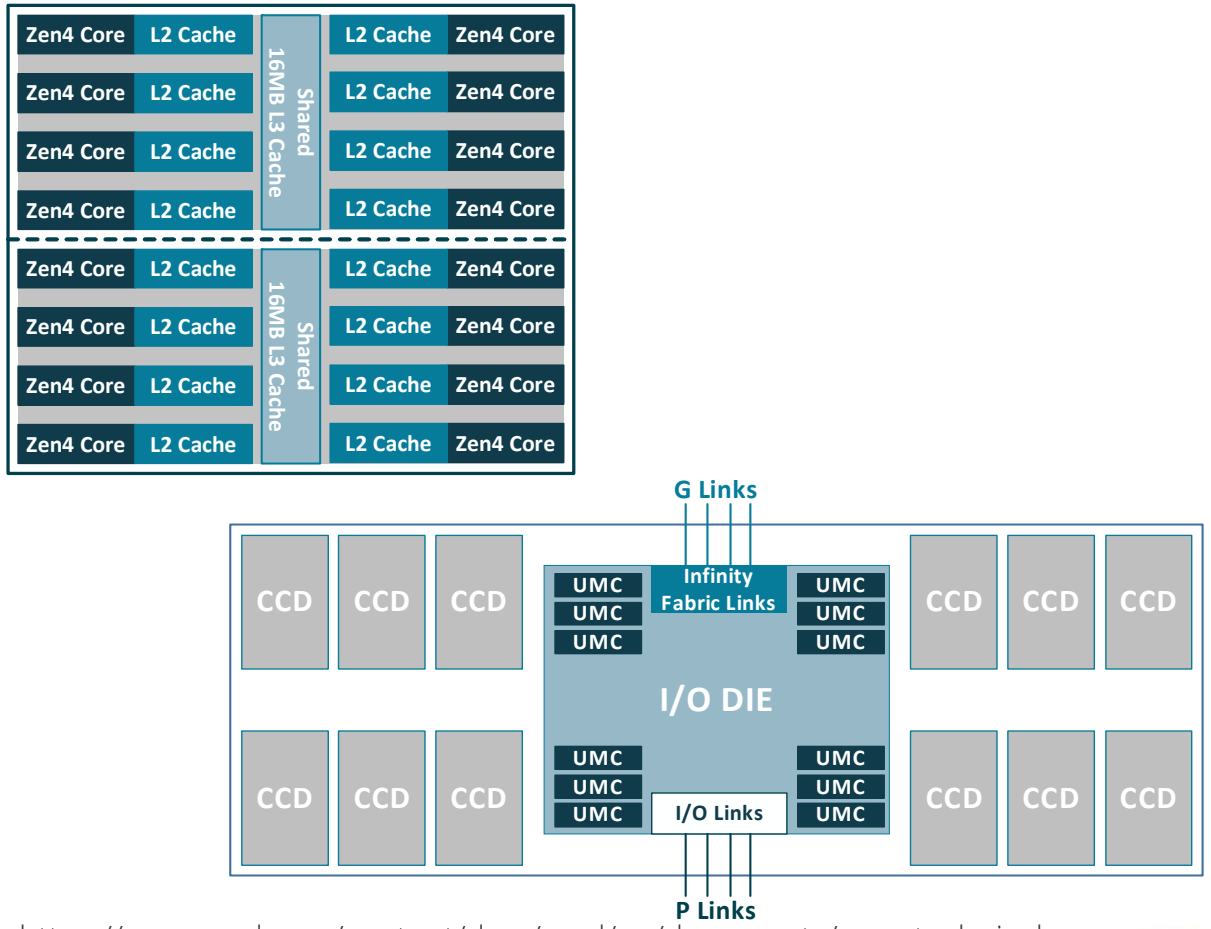
- MPIS32 and RV64 compatible processor model
- OOO model
  - Configure micro-architectural details
    - Instruction fetch/decode/retire rate
    - ROB size
  - Branch prediction
- Multi-threaded cores
- Musl libc used to cross-compile programs
  - Also tested with clang
- System-call emulation

SOCs

What if you wanted something much more complex? Can SST simulate a core with SMT? What about something like a modern processor? SPR? EPYC?



https://www.intel.com/content/www/us/en/developer/articles/technical/fourth-generation-xeon-scalable-family-overview.html

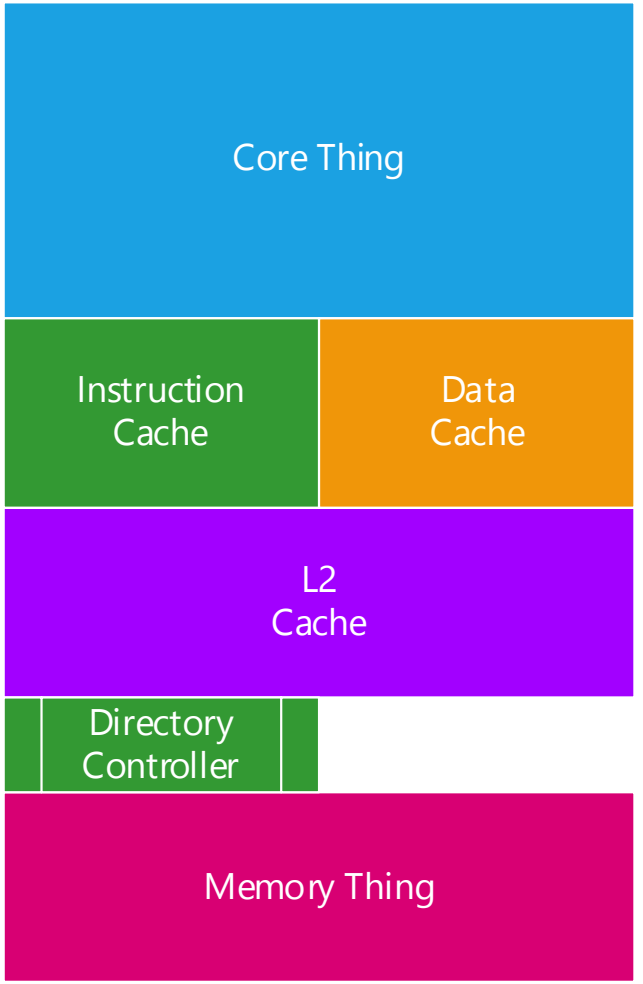


https://www.amd.com/content/dam/amd/en/documents/epyc-technical-docs/white-papers/58015-epyc-9004-tg-architecture-overview.pdf

Single-core Vanadis

Let's start simple…

- RISC-V core
- I-Cache
- D-Cache
- L2 Cache
- Memory



Single-core Vanadis

What attributes can we change in the core model?

```
$ sst-info vanadis
```

```
Parameters (35 total)
        verbose: Set the level of output verbosity, 0 is no output, higher is more output  [0]
        dbg_mask: Mask for output. Default is to not mask anything out (0) and defer to 'verbose'.  [0]
        start_verbose_when_issue_address: Set verbose to 0 until the specified instruction address is issued, then set to 'verbose' parameter  []
        stop_verbose_when_retire_address: When the specified instruction address is retired, set verbose to 0  []
        pause_when_retire_address: If specified, the simulation will stop when this address is retired.  [0]
        pipeline_trace_file: If specified, a trace of the pipeline activity will be generated to this file.  []
        max_cycle: Maximum number of cycles to execute. The core will halt after this many cycles.  [std::numeric_limits<uint64_t>::max()]
        node_id: Identifier for the node this core belongs to. Each node in the system needs a unique ID between 0 and (number of nodes) - 1. Used to tag output.  [0]
        core_id: Identifier for this core. Each core in the system needs a unique ID between 0 and (number of cores) - 1.  [<required>]
        hardware_threads: Number of hardware threads in this core  [1]
        clock: Core clock frequency  [1GHz]
        reorder_slots: Number of slots in the reorder buffer  [64]
        physical_integer_registers: Number of physical integer registers per hardware thread  [128]
        physical_fp_registers: Number of physical floating point registers per hardware thread  [128]
        integer_arith_units: Number of integer arithemetic units  [2]
        integer_arith_cycles: Cycles per instruction for integer arithmetic  [2]
        integer_div_units: Number of integer division units  [1]
        integer_div_cycles: Cycles per instruction for integer division  [4]
        fp_arith_units: Number of floating point arithmetic units  [2]
        fp_arith_cycles: Cycles per floating point arithmetic  [8]
        fp_div_units: Number of floating point division units  [1]
        fp_div_cycles: Cycles per floating point division  [80]
        branch_units: Number of branch units  [1]
        branch_unit_cycles: Cycles per branch  [int_arith_cycles]
        issues_per_cycle: Number of instruction issues per cycle  [2]
        fetches_per_cycle: Number of instruction fetches per cycle  [2]
        retires_per_cycle: Number of instruction retires per cycle  [2]
        decodes_per_cycle: Number of instruction decodes per cycle  [2]
        dcache_line_width: Width of a line for the data cache, in bytes. (Currently not used but may be in the future).  [64]
        icache_line_width: Width of a line for the instruction cache, in bytes  [64]
        print_retire_tables: Print registers during retirement step (default is yes)  [true]
        print_issue_tables: Print registers during issue step (default is yes)  [true]
        print_int_reg: Print integer registers true/false, auto set to true if verbose > 16  [false]
        print_fp_reg: Print floating-point registers true/false, auto set to true if verbose > 16  [false]
        print_rob: Print reorder buffer state during issue and retire  [true]
```

Single-core Vanadis

What attributes can we change in the core model?

```
$ sst-info vanadis
```

Default Parameter

```
Parameters (Purged a Few…)
        clock: Core clock frequency  [1GHz]
        reorder_slots: Number of slots in the reorder buffer  [64]
        physical_integer_registers: Number of physical integer registers per hardware thread  [128]
        physical_fp_registers: Number of physical floating point registers per hardware thread  [128]
        integer_arith_units: Number of integer arithemetic units  [2]
        integer_arith_cycles: Cycles per instruction for integer arithmetic  [2]
        integer_div_units: Number of integer division units  [1]
        integer_div_cycles: Cycles per instruction for integer division  [4]
        fp_arith_units: Number of floating point arithmetic units  [2]
        fp_arith_cycles: Cycles per floating point arithmetic  [8]
        fp_div_units: Number of floating point division units  [1]
        fp_div_cycles: Cycles per floating point division  [80]
        branch_units: Number of branch units  [1]
        branch_unit_cycles: Cycles per branch  [int_arith_cycles]
        issues_per_cycle: Number of instruction issues per cycle  [2]
        fetches_per_cycle: Number of instruction fetches per cycle  [2]
        retires_per_cycle: Number of instruction retires per cycle  [2]
        decodes_per_cycle: Number of instruction decodes per cycle  [2]
```

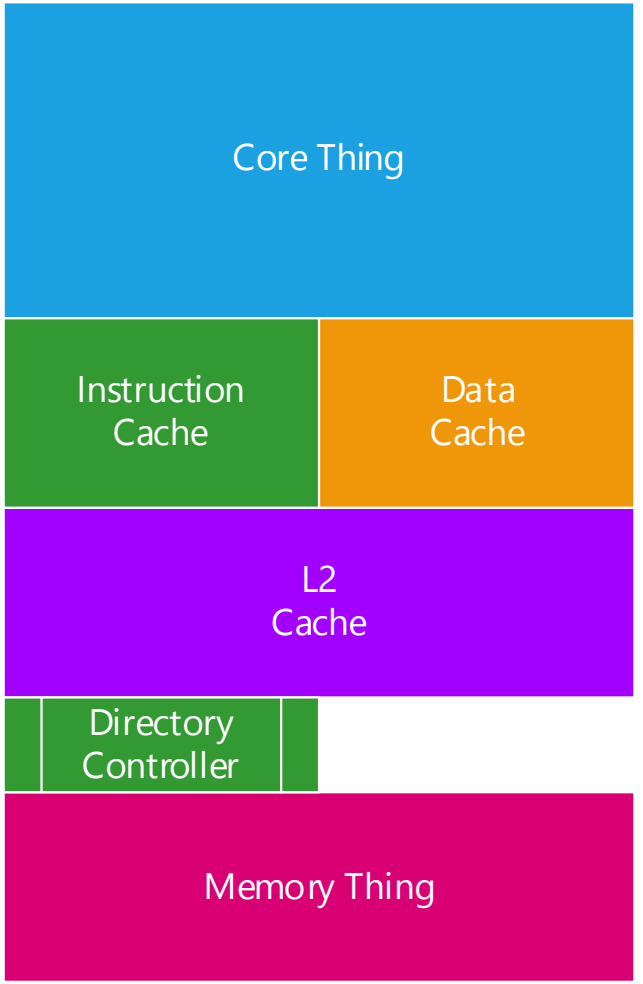
Single-core Vanadis

Let's start simple…

```
$ cat basic_vanadis.py
```

Ok, not so simple but the structure is similar to all SST configurations

```
$ sst no_rtr_vanadis.py
```



Vanadis: The tradeoffs

$ sst-info vanadis.VanadisNodeOS

Pros

- Runs binaries
- Detailed model of instruction dependencies

Cons

- Slower than other models

Memory

Elements:

memHierarchy

MemHierarchy: Memory system

```
$ sst-info memHierarchy | grep " Component"
```

Collection of interoperable memory system elements

- Caches
- Directories
- Memory controllers
  - Interfaces to memory models (DDR, HBM, HMC, NVM, etc.)
- Scratchpads
- NoC (network-on-chip) interfaces
- Buses

Components are cycle-accurate/cycle-level

Capable of modeling modern cache and memory subsystems

MemHierarchy: Cache modeling

$ sst-info memHierarchy.Cache

Highly configurable

- Arbitrary hierarchy depth, flexible topologies
- Cache inclusivity, coherence, private/shared, etc. configurable
- Single- and multi-socket configurations
- Prefetch via *Cassini* element library

Data movement

- Components support direct, bus, and on-chip network (NoC) communication

Event types: read/write, atomics, LLSC, noncacheable, custom memory, etc.

MemHierarchy: Memory modeling

$ sst-info memHierarchy.MemController

Interface to memory is the *MemController*

MemControllers implement *backends*

- These do the actual work of timing memory access
- Can be interfaces to other memory simulators
- More on the next slide

Support *custom memory instructions*

- Including ability to do cache shootdowns for coherence maintenance

MemHierarchy: Memory modeling

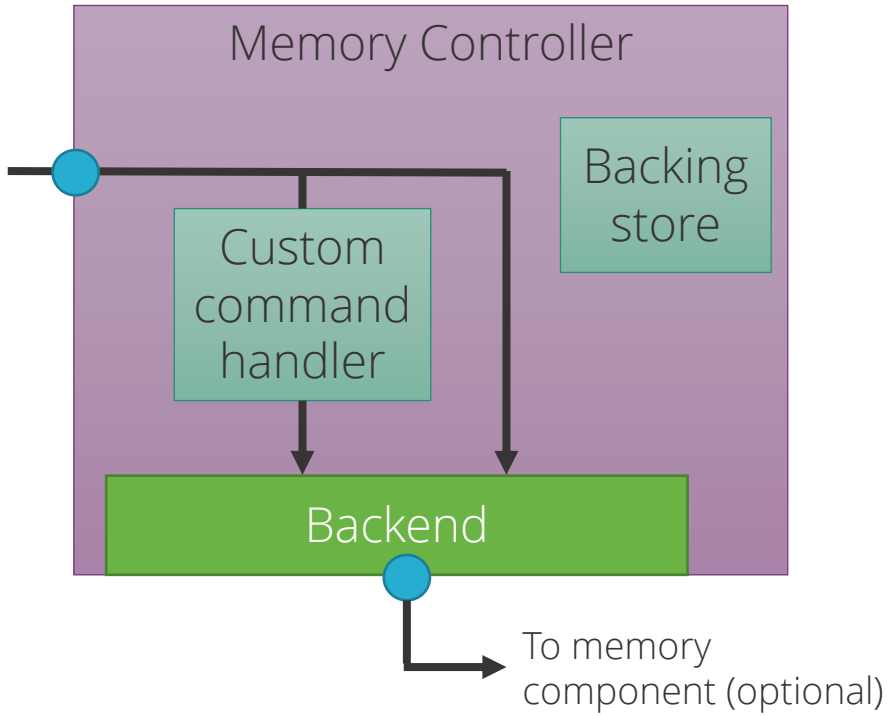
$ sst-info memHierarchy.MemController

Memory controller

- Manages data values if needed (backing store)
- Facilitates custom memory commands
  - Including cache shootdowns for coherence maintenance
- Passes events to *memory backend* subcomponent

Backend: the "real" memory controller and/or memory

- Implementations
  - Memory controller and model itself
  - Memory controller with interface to a memory component
  - Interface to another memory controller/memory component
  - Wrapper to an external simulator



MemHierarchy: SST 13.1 backends

Memory (external)

- CramSim (DDR, HBM)
- DRAMSim3 DDR, LPDDR, GDDR HBM, HMC, STT-MRAM)
- FlashDIMMSim (FLASH)
- HMCSim/GoblinHMC (HMC)
- Messier (NVRAM)
- Ramulator (DDR, HBM, HMC)
- SimpleDRAM (DDR)
- SimpleMem (constant latency)
- TimingDRAM (DDR)
- VaultSimC (HMC-like)

- Plus a few that can be used with other backends to reorder requests, add latency, etc.

Running a Simulation – Add Components, L2 Cache

Copy configuration

```
$ cp demo_2.py demo_3.py
```

Add an L2 cache between L1 and memory to new configuration

```
What should you add?
```

- What parameters are available for an L2 cache?
- What are appropriate values for the parameters?

Launch simulation

```
$ sst demo_3.py
```

- How did this affect your overall simulated time?
- How did this affect traffic to and from your backing store?

Running a Simulation – Switch Components, Timing DRAM

Copy configuration

```
$ cp demo_3.py demo_4.py
```

Switch the simpleMem subcomponent for timingDRAM

```
What should you change? Remember that sst-info is your pal!
```

Launch simulation

```
$ sst demo_4.py
```

- How do your results differ from the run with simpleMem?

Networks

Elements:

Merlin

Kingsley

Merlin: Network simulator

$ sst-info merlin

Low-level networking components that can be used to simulate high-speed networks (machine level) or on-chip networks

Capabilities

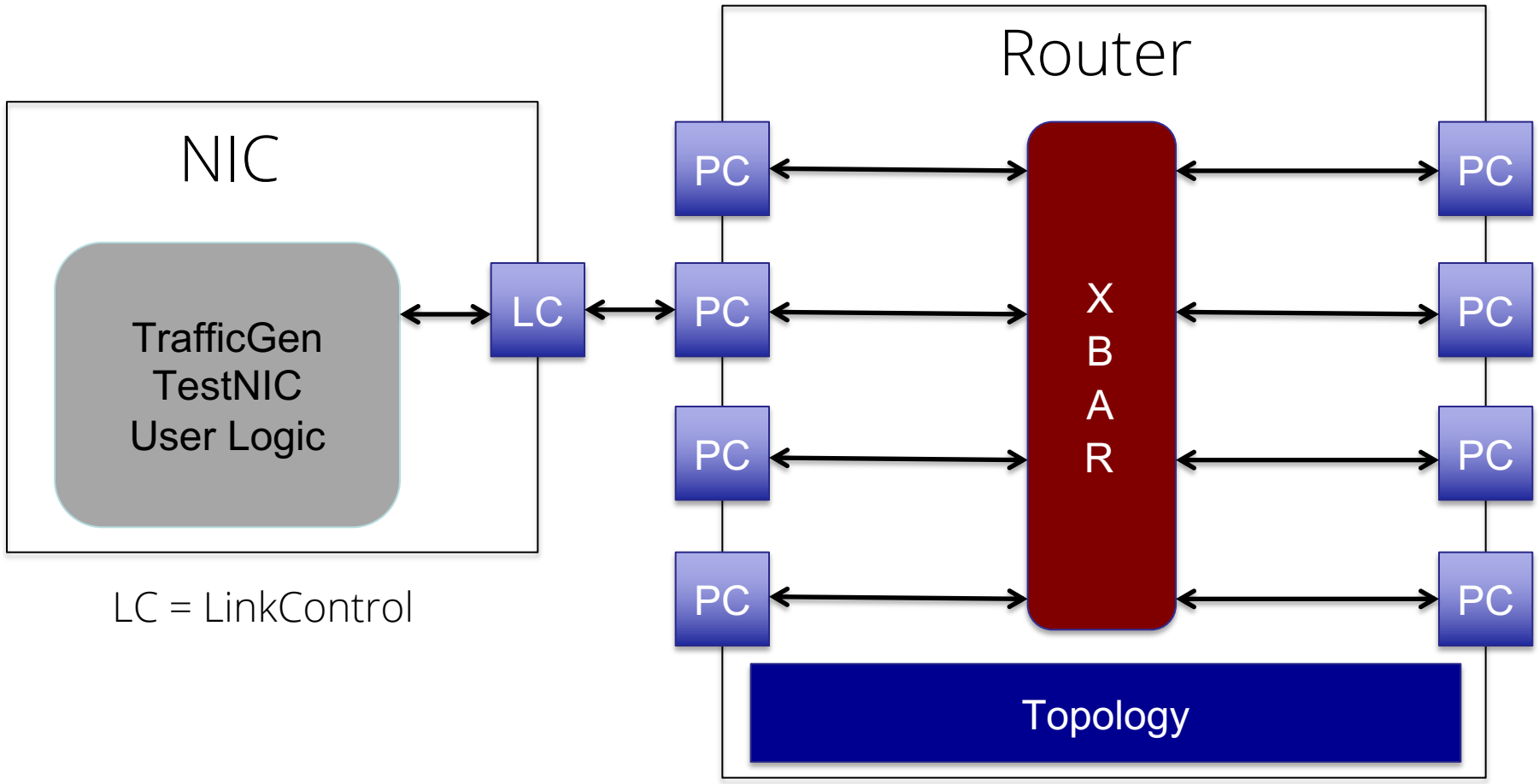
- High radix router model (hr_router)
- Topologies – mesh, n-dim tori, fat-tree, dragonfly,

Many ways to drive a network

- Simple traffic generation models
  - Nearest neighbor, uniform, uniform w/ hotspot, normal, binomial
- *MemHierarchy*
- Lightweight network endpoint models (*Ember* – coming up next)
- Or, make your own

Merlin: Organization

$ sst-info merlin



LC = LinkControl

PC = PortControl

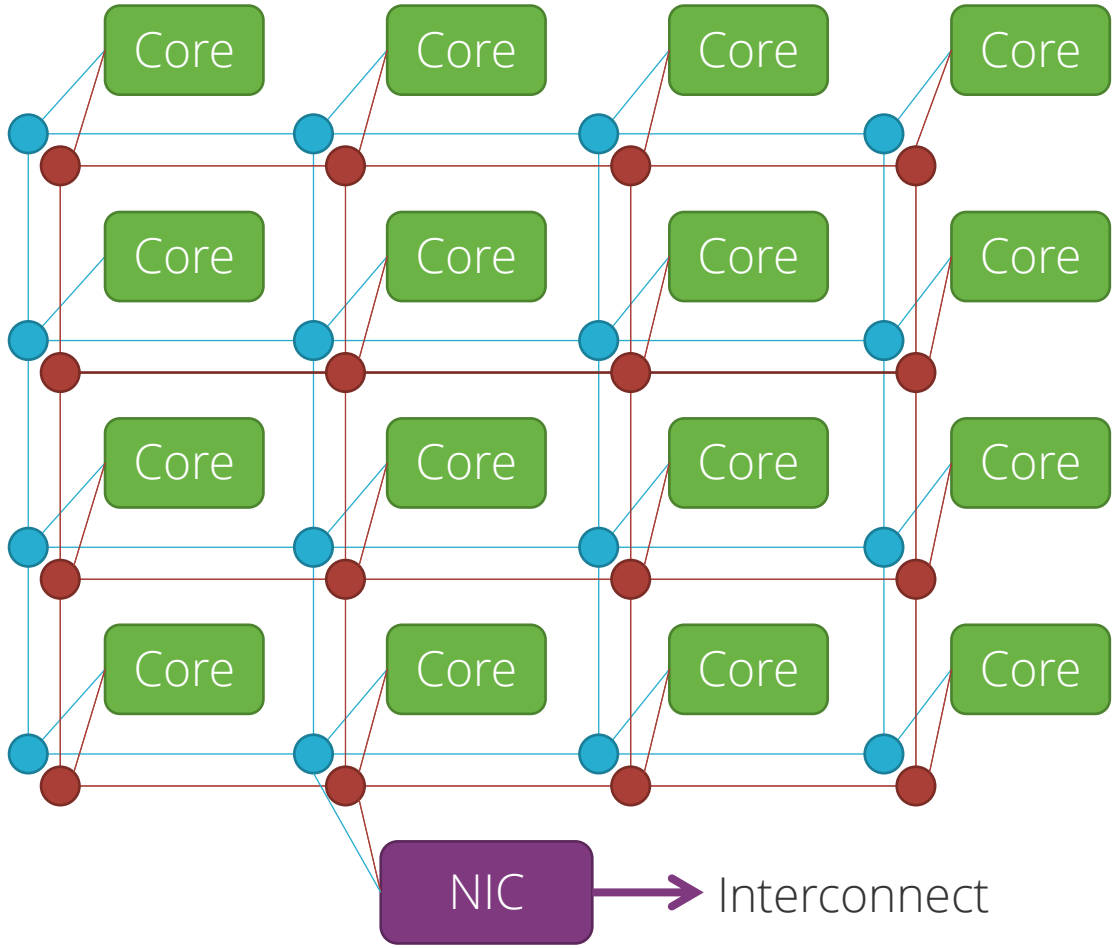
Kingsley: Mesh simulator

$ sst-info kingsley

Network-on-chip model; mesh configuration

Similar to Merlin but:

- No input queuing at routers
- Mesh topology only
- Not all ports need to be populated
- Possible to instantiate multiple unconnected networks
  - Multiple physical networks for coherence (e.g., request/response/ack/forward)
  - Kingsley NoC + Merlin/Kingsley system network



Network Drivers

Elements:

Ember

Firefly

Hermes

Ember: Network Traffic Generator

$ sst-info ember

Light-weight endpoint for modeling network traffic

- Enables large-scale simulation of networks where detailed modeling of endpoints would be expensive

Packages patterns as *motifs*

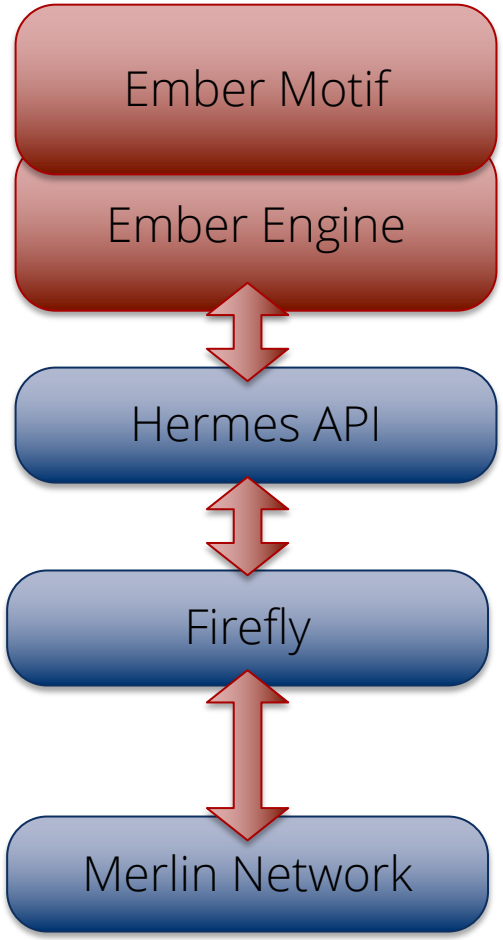
- Can encode a high level of complexity in the patterns
- Generic method for users to extend SST with additional communication patterns

Intended to be a driver for the Hermes, Firefly, and Merlin communication modeling stack

- Uses Hermes message API to create communications
- Abstracted from low-level, allowing modular reuse of additional hardware models

Ember: Overview

$ sst-info ember



High Level Communication Pattern and Logic
Generates communication events

Event to Message Call, Motif Management
Handles the tracking of the motif

Message Passing Semantics
Collectives, Matching, etc.

Packetization and Byte Movement Engine
Generates packets and coordinates with network

Flit Level Movement, Routing, Delivery
Moves flits across network, timing, etc.

Ember: Motifs

$ sst-info ember | grep Motif

Motifs are lightweight patterns of communication

- Tend to have very small state
- Extracted from parent applications
- Models as an MPI program (serial flow of control)
  - Many motifs acting in the simulation create the parallel behavior

Example motifs

- Halo exchanges (1, 2, and 3D)
- MPI collections – reductions, all-reduce, gather, barrier
- Communication sweeping (Sweep3D, LU, etc.)

Ember: Motifs (continued)

$ sst-info ember.EmberEngine

The EmberEngine creates and manages the motif

- Creates an event queue which the motif adds events to when probed
- The Engine executes the queued events in order, converting them to message semantic calls as needed
- When the queue is empty, the motif is probed again for events

Events correspond to a specific action

- E.g., send, recv, allreduce, compute-for-a-period, wait, etc.

Firefly: Network traffic

$ sst-info firefly

Purpose: Create network traffic, based on application communication patterns, at large scale

- Enables testing the impact of network topologies and technologies on application communication at very large scale

Scales to 1 million nodes

Supports multiple “cores” per Node

- Interaction between cores limited to message passing

Supports space sharing of the network

- Multiple “apps” running simultaneously

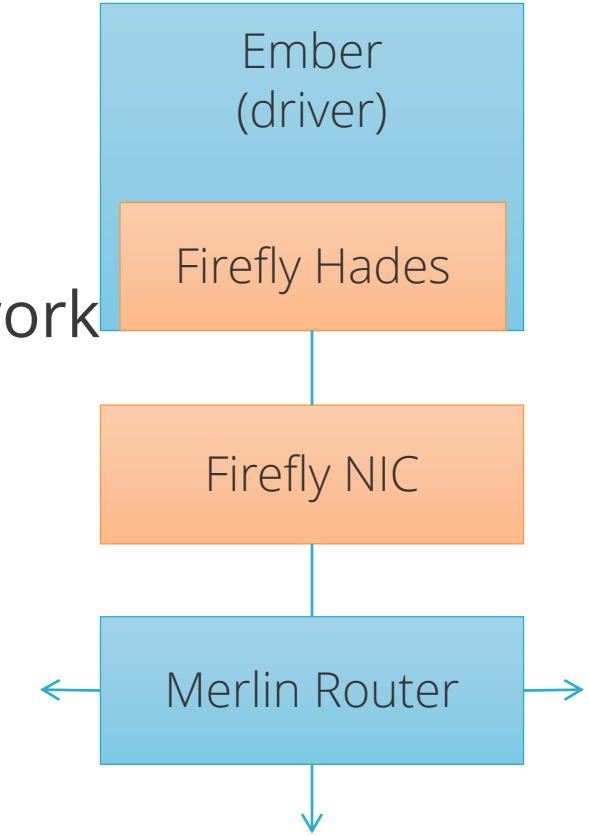
Firefly: Simulating large networks

$ sst-info firefly

A network node consists of
- Driver (the “application”)
- NIC
- Router

Nodes are connected together via routers to form a network
- Fat tree, torus, etc.

Firefly is the interface between the driver and the router
- Message passing library → Firefly Hades
- NIC → Firefly NIC



(More) SST 13.1 Elements

Processors
- **Ariel** – PIN-based
- **Prospero** – trace execution
- **Miranda** – pattern generator
- **Vanadis** – MIPS32 and RV64 pipeline

Memory Subsystem
- **MemHierarchy** – caches, directory, memory

Network drivers
- **Ember** – communication patterns
- **Firefly** – communication protocols
- **Hermes** – MPI-like driver interface

Networks/NoCs
- **Merlin** – flexible network modeling
- **Kingsley** – mesh NoC

Even More Elements

There are even more elements included, described here:

- http://sst-simulator.org/sst-docs/docs/elements/intro

A number of external simulators are compatible with SST.

- See section on optional dependencies: http://sst-simulator.org/SSTPages/SSTBuildAndInstall_13dot1dot0_SeriesDetailedBuildInstructions/
- Many memory timing simulators are supported by memHierarchy

Viewing Configuration Graph

Let's take a look at how SST views our system

Re-run demo_4 but add a command to dump the configuration graph

```
$ sst --output-dot=graph_demo_4.dot –dot-
verbosity=10 demo_4.py
```

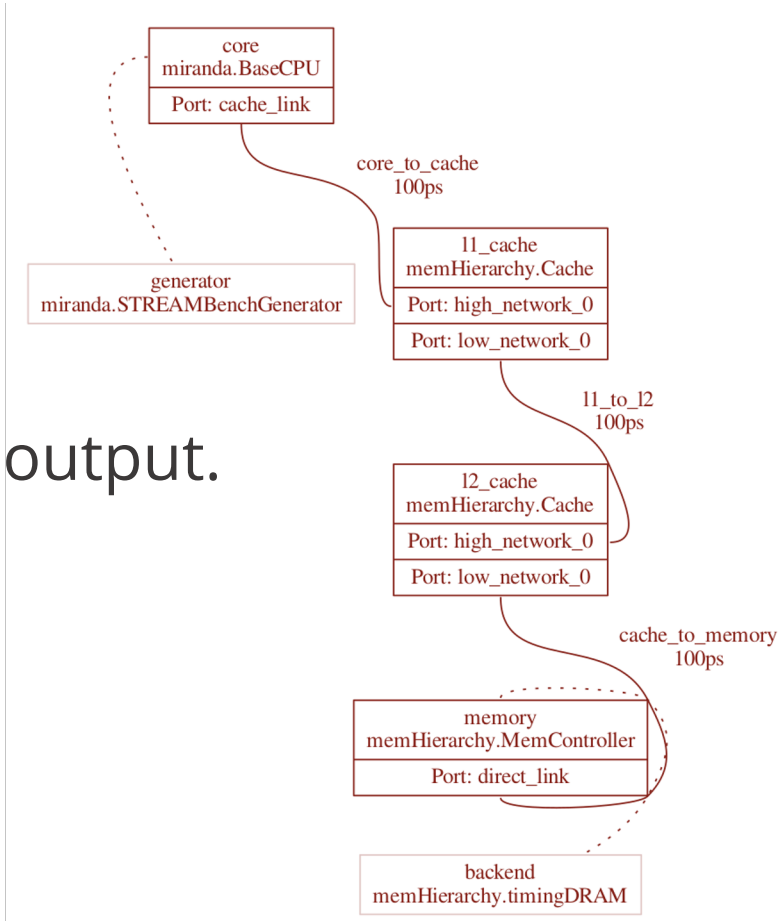
This gives you a GraphViz formatted file

```
$ dot –Tjpg graph_demo_4.dot –o graph_demo_4.jpg
```

You can install a JPEG previewer in VS Code to see the output.

Is this how you expected your system to look?

With this in mind, let's add a second Miranda core…



Running a Simulation – Add Components, Second Miranda

Copy configuration

```
$ cp demo_4.py demo_5.py
```

Add an a second Miranda generator

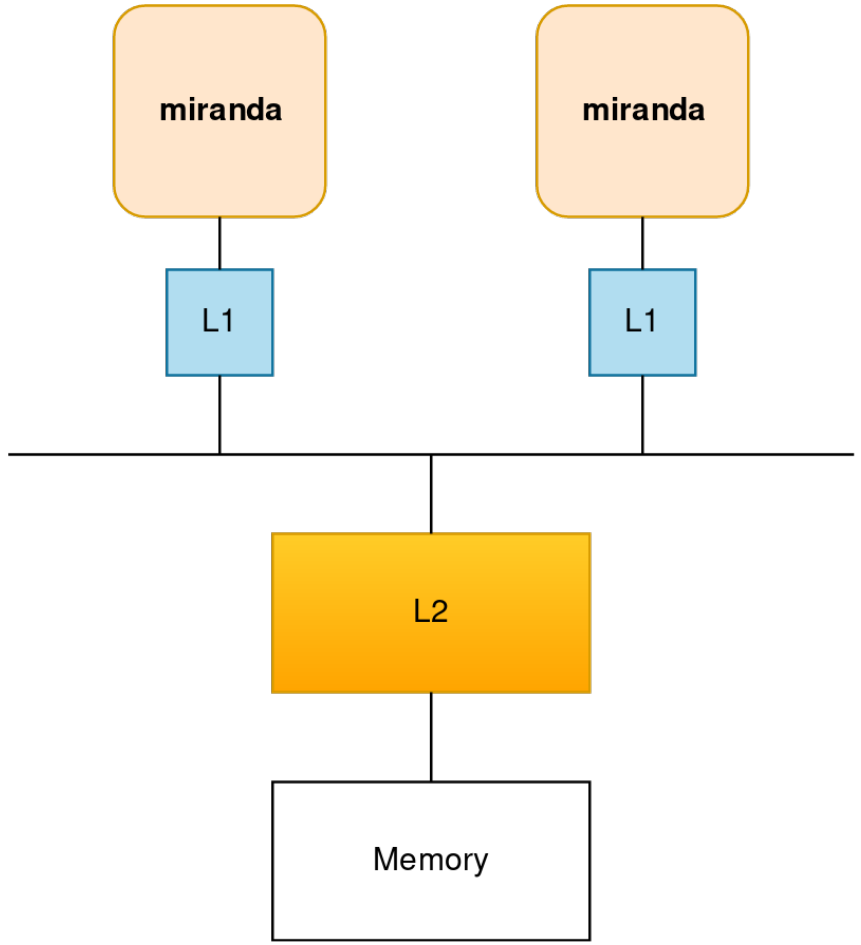
- What else might you need?

```
• How about another L1 cache?
• How are you going to wire everything
  together? How about a bus?
```

Dump the wiring diagram to verify the model

Launch simulation

```
$ sst demo_5.py
```



Running a Simulation – Add Components, Second L2

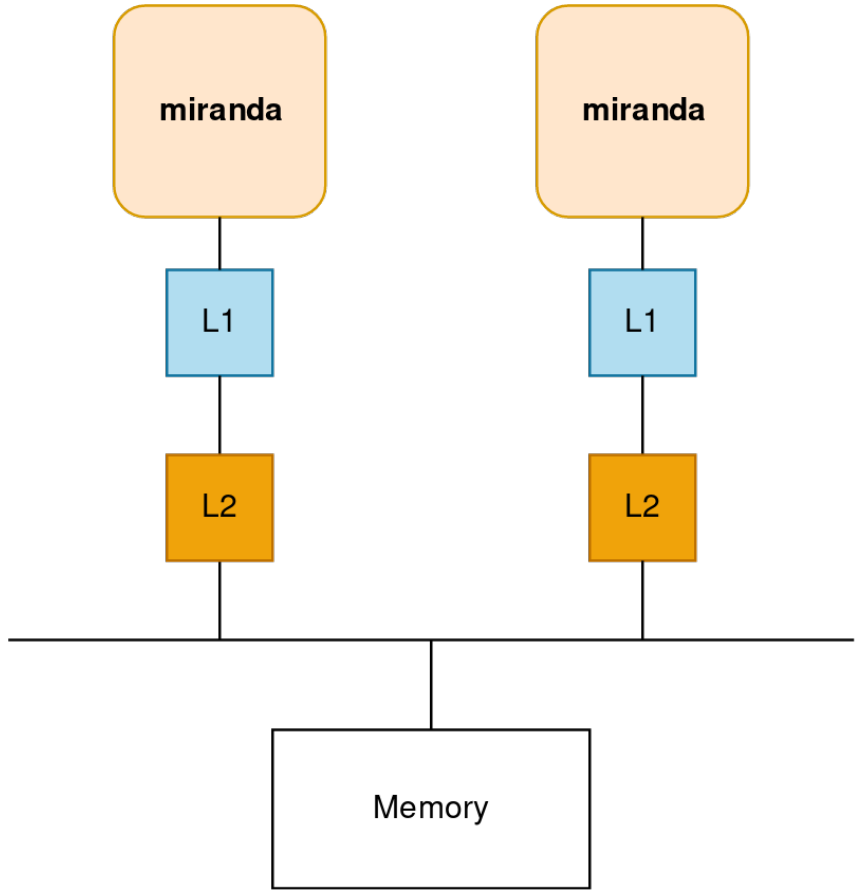
Copy configuration

```
$ cp demo_5.py demo_6.py
```

Add L2 cache and move both above the bus

Launch simulation

```
$ sst demo_6.py
```



Running a Simulation – Add Components, Secondary Memory

Copy configuration

```
$ cp demo_6.py demo_7.py
```

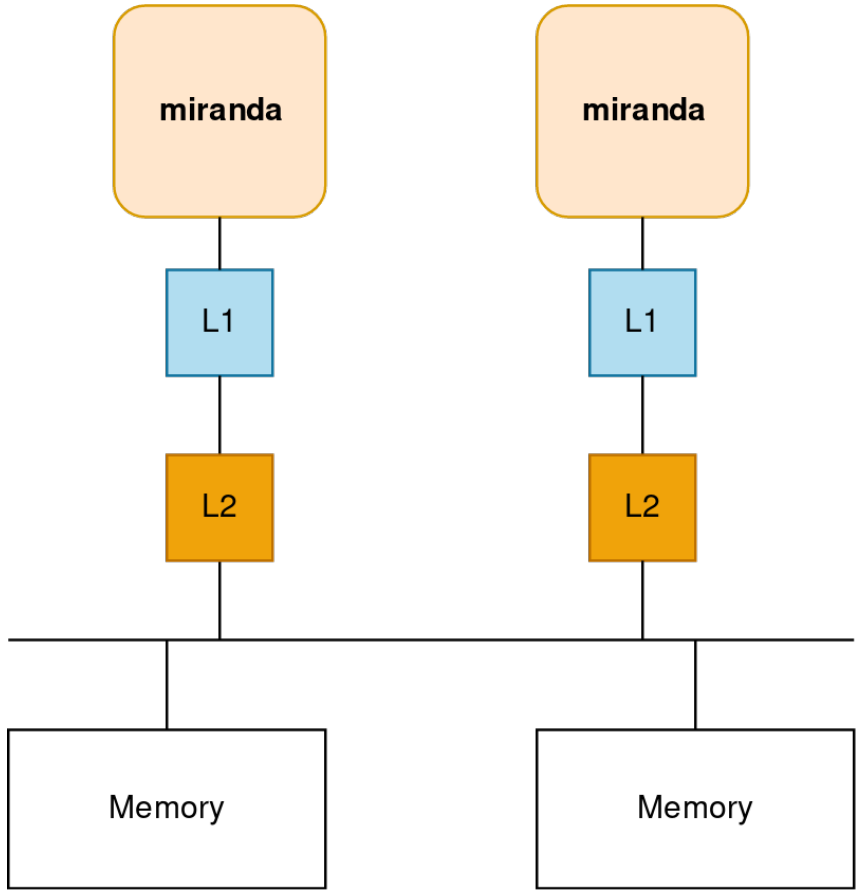
Add an a second L2 and memory controller

- Think carefully about how addressing should will work…

```
Can you still use a bus?
Can the talk directly to the memory controller?
```

Launch simulation

```
$ sst demo_7.py
```



Getting Help & Extending SST

Extending SST

SST was designed for extensibility

- Components/subcomponents can be added without touching SST Elements
  - Example: write a new prefetcher and have memH caches use it → *no changes* to memHierarchy
- SST-Core APIs are stable → one year deprecation period
  - Element APIs may be less so but generally try to keep them consistent
- Many users start with SST Elements and then build their own customized libraries
  - Partially or completely replacing SST Element functionality

Many approaches to using SST

- Core only: Write your own components from scratch
- Start from existing Elements and replace components/subcomponents to meet your needs
- Wrap existing simulators and insert as components or subcomponents

Extending SST: Resources

Example element library

- Components demonstrating links, ports, clocks, event handling, etc.
- sst-elements/src/sst/elements/simpleElementExample/

simpleSimulation

- Simulates a car wash (a little more complex than example elements)

Example external element library

- Demonstrates building and registering a new element library
- https://github.com/sstsimulator/sst-external-element

Finally: Getting help

SST website contains lots of information (www.sst-simulator.org)

- Downloading, installing, and running SST
- Element libraries and external components
- Guides for extending SST
- Information on APIs
- Information about current development efforts
- Past tutorial slides and exercises

SST Github

- Current development
- Issues track user questions as well as development plans, bugs, etc.

Part 1 wrap-up

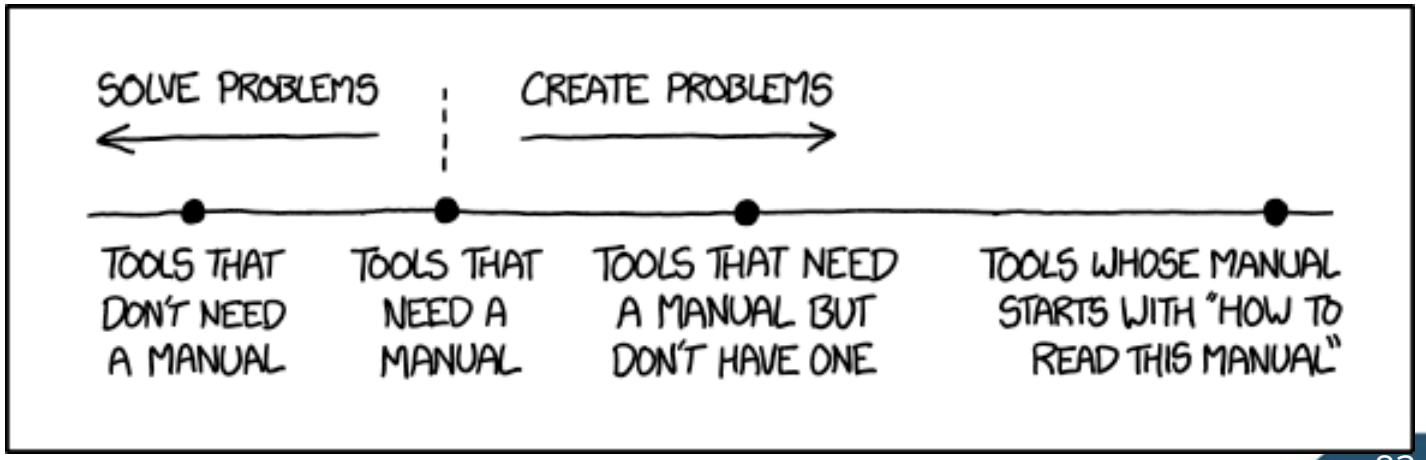
SST is a parallel, flexible simulation framework

- Can simulate many systems at many granularities
- Capable of simulating modern architectures
- Modular design for extensibility

Please keep us posted on your uses of SST as well as any capabilities you've added or would like to see added

The SST team wants to help you!

- Documentation?
- Examples?
- Kittens?



Welcome!

Part 1: Introduction to SST	8:00 – 9:30
SST Overview	
Basic Simulation in SST	
A Tour of SST Elements	
Break	*9:30 – 9:45*
Part 2: Advanced Node-Level Simulation	*9:45 – 10:30*
Part 3: System-Scale Simulation	*10:30 - 12:00*

BREAK

Return at 9:45

Welcome!

Part 1: Introduction to SST SST Overview Basic Simulation in SST A Tour of SST Elements	8:00 – 9:30
Break	*9:30 – 9:45*
Part 2: Advanced Node-Level Simulation	***9:45 – 10:30***
Part 3: System-Scale Simulation	*10:30 - 12:00*

Learning Objectives – Part 1: Advanced Node-Level Simulation

This section of the tutorial will cover the following topics:

1. How to wire up a more modern memory hierarchy
2. How to simulate an OOO processor with Vanadis

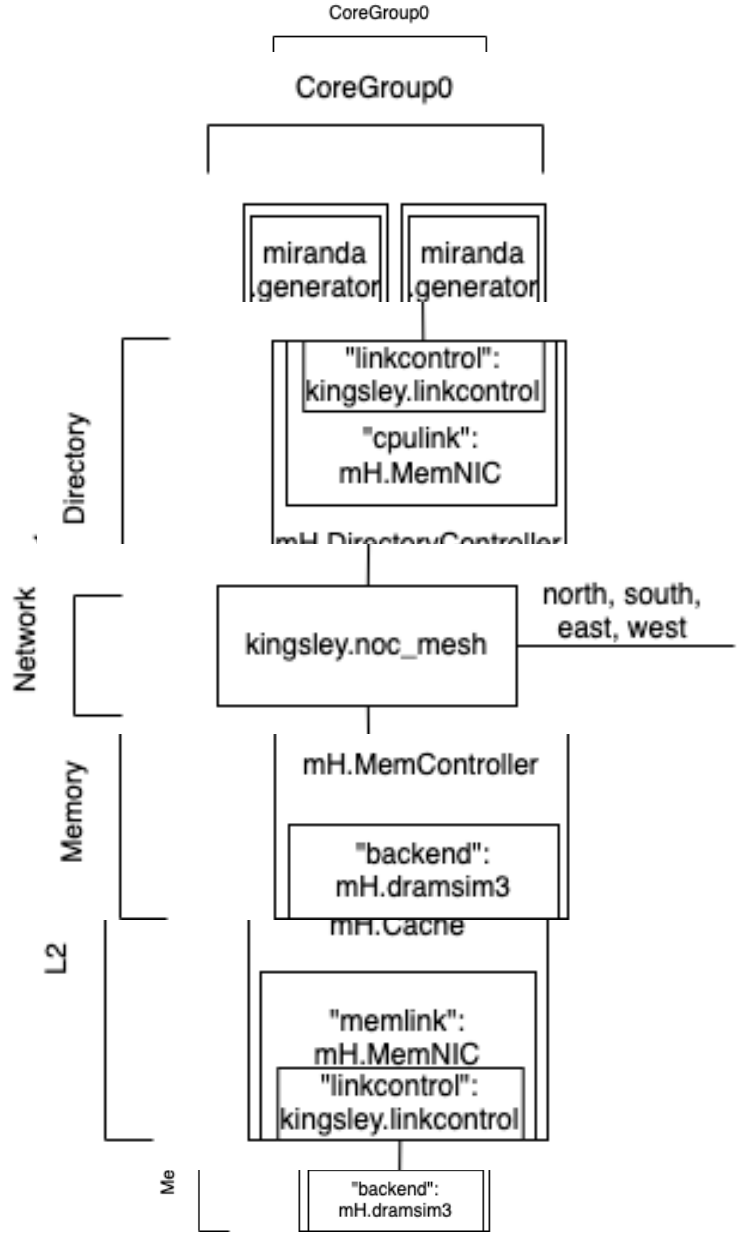
Advanced memHierarchy

Advanced memHierarchy Configuration

- Lets do something a bit more realistic.
- Let’s model a processor with
  - Split LLC
  - Multiple NUMA domains
  - 2D mesh network
- We will keep the simple core model, Miranda

numa.py

- To the right is a single core group
- A core has its own L1D cache
- Each group of cores shares an L2
- In the middle, the NOC router will connect to others
- At the bottom, the memory needs a directory controller



System-Level Modeling

Welcome!

Part 1: Introduction to SST SST Overview Basic Simulation in SST A Tour of SST Elements	8:00 – 9:30
Break	*9:30 – 9:45*
Part 2: Advanced Node-Level Simulation	9:45 – 10:30
Part 3: System-Scale Simulation	***10:30 - 12:00***

Learning Objectives – Part 3: System-Level Modeling

This section of the tutorial will cover the following topics:

1. Detailed description of Merlin, our system-level network simulator
2. Network simulation interfaces
3. How to write a new SST element
4. How to add a new network topology
5. How to extend the SST python interface to support the new topology

High-level View

Merlin: Network Simulator

Low-level, flexible networking components that can be used to simulate high-speed networks (machine level) or on-chip networks

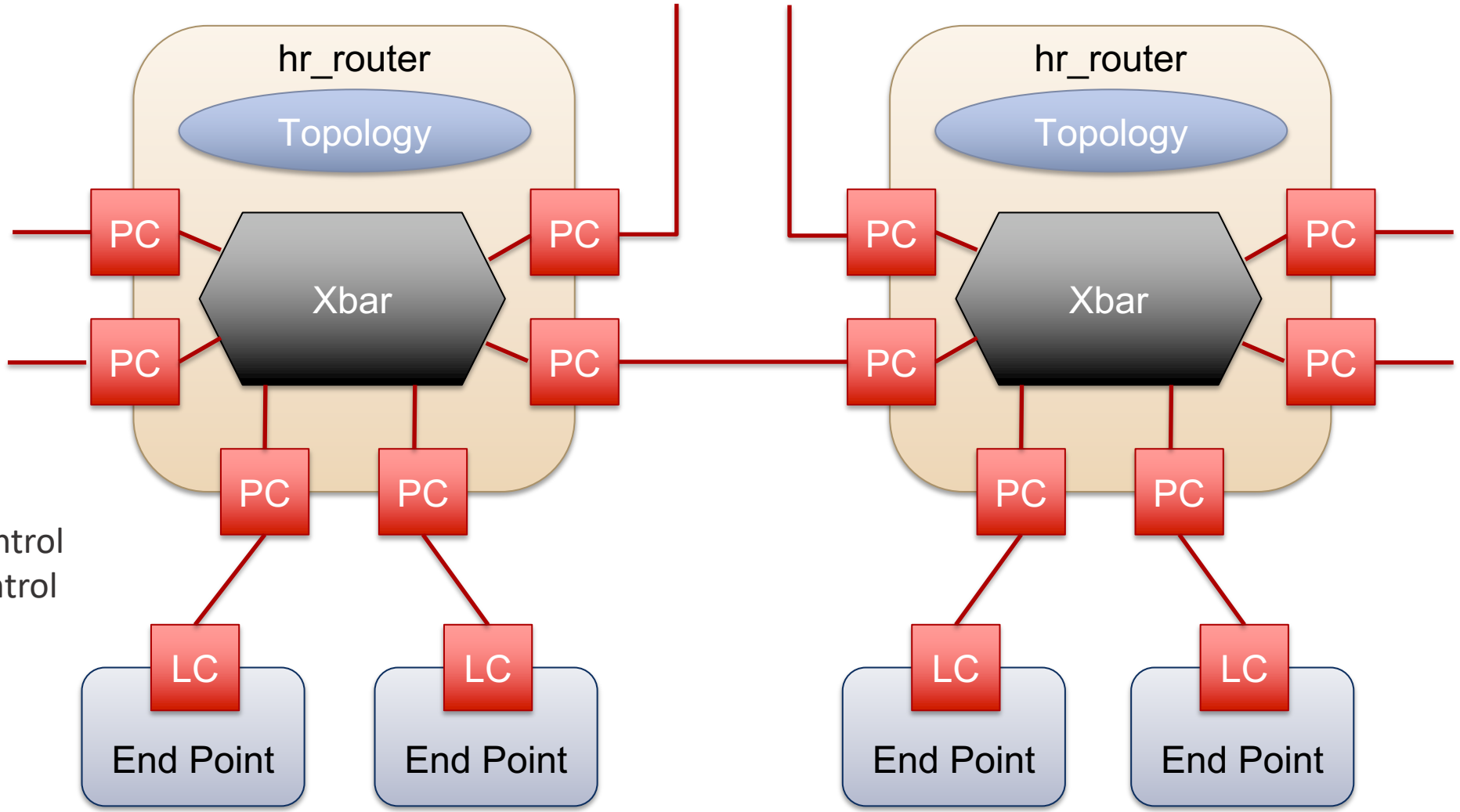
Capabilities

- High radix router model (hr_router)
- Topologies – mesh, n-dim tori, fat-tree, dragonfly, hyperx

Many ways to drive a network

- Simple traffic generation models
- *MemHierarchy*
- Lightweight network endpoint models (*Ember*)
- Or, make your own

Merlin High Level Overview



PC = PortControl
LC = LinkControl

General Parameters

General

- link_bw – bandwidth of the network links (in B/s or b/s)

hr_router

- xbar_bw – per port bandwidth into router crossbar (in B/s or b/s)
- flit_size – size of a flit (in B or b)
- input_latency – input latency of router (in s)
- output_latency – output latency of router (in s)
- xbar_arb – crossbar arbitration unit to be used

hr_router/LinkControl

- input_buf_size – size of input buffer for router/NIC (in B or b)
- output_buf_size – size of output buffer for router/NIC (in B or b)

LinkControl – Endpoint Facing

Inherits from SST::Interfaces::SimpleNetwork

```
bool send(SST::Interfaces::SimpleNetwork::Request* req, int vn);
bool spaceToSend(int vn, int flits);

SST::Interfaces::SimpleNetwork::Request* recv(int vn);
bool requestToReceive( int vn ) { return ! input_buf[vn].empty(); }

void sendUntimedData(SST::Interfaces::SimpleNetwork::Request* ev);
SST::Interfaces::SimpleNetwork::Request* recvUntimedData();

void setNotifyOnReceive(HandlerBase* functor) { receiveFunctor = functor; }
void setNotifyOnSend(HandlerBase* functor) { sendFunctor = functor; }

bool isNetworkInitialized() const { return network_initialized; }
nid_t getEndpointID() const { return id; }
const UnitAlgebra& getLinkBW() const { return link_bw; }
```

SimpleNetwork::Request

```
class Request : public SST::Core::Serialization::serializable {
public:
    nid_t  dest;         /*!< Node ID of destination */
    nid_t  src;          /*!< Node ID of source */
    int    vn;           /*!< Virtual network of packet */
    size_t size_in_bits;  /*!< Size of packet in bits */
    bool   head;         /*!< True if this is the head of a stream */
    bool   tail;         /*!< True if this is the tail of a steram */

private:
    Event* payload;      /*!< Payload of the request */
public:
    inline void givePayload(Event *event);
    inline Event* takePayload();
    inline Event* inspectPayload();

protected:
    TraceType trace;
    int traceID;
```

LinkControl-PortControl Interactions

LinkControl and PortControl share data and negotiate various parameters during the init() phase

- Each LC/PC pair will negotiate link bandwidth. It is set to the minimum of the two set bandwidths
- PortControl will notify the LinkControl of the network ID used to address it
- PortControl will report FLIT size to the LinkControl
- LinkControl notifies PortControl of the desired Virtual Networks to be used
  - This is a deprecated feature. Number of VNs is now set directly through the router
- The LinkControl and PortControl objects manage send credits

PortControl/LinkControl Statistics

packet_latency (LC only)
- For each packet, adds latency to statistics object (side effect is that it counts the number of packets received at an endpoint)

send_bit_count
- For each packet, adds the size in bits to the statistics object (side effect is that it counts number of packets sent on a port)

output_port_stalls
- For each interval where data is present, but can't be sent due to lack of send credits, add the time stalled to statistics object

idle_time (PC only for now)
- For each interval where no data is present to be sent, add total idle time to statistics object

ReorderLinkControl

Inherits from SST::Interfaces::SimpleNetwork

Contains a SimpleNetwork interface object to talk to the “physical” layer (this is typically just a LinkControl).

Puts a sequence number on each packet and reorders packets on the receive-side before giving them to endpoint

- Allows endpoint models that can’t handle out of order receipt of packets to use network models that don’t guarantee ordering

Currently assumes “infinite” resources for buffer and reordering

Crossbar Arbitration

xbar_arb_rr
- Round robin allocation – first across ports, then across virtual channels

xbar_arb_lru
- Least recently used – across all port/VC pairs

xbar_arb_age
- Oldest packets get higher priority

xbar_arb_rand
- Random priority assigned to each port/VC pair each arbitration cycle

Topology Module

Router knows nothing about topology/routing without loading a topology module

- Can use static and/or dynamic routing

Available topologies

- SingleRouter
- Mesh
- Torus
- Fattree
- HyperX/Flattened Butterfly
- Dragonfly
- PolarFly/PolarStar

Topology Class – More on this later

```
virtual void route_packet(int port, int vc, internal_router_event* ev) = 0;
virtual internal_router_event* process_input(RtrEvent* ev) = 0;

virtual PortState getPortState(int port) const = 0;
bool isHostPort(int port) const;
virtual std::string getPortLogicalGroup(int port) const;

virtual void routeInitData(int port, internal_router_event* ev, std::vector<int> &outPorts) = 0;
virtual internal_router_event* process_InitData_input(RtrEvent* ev) = 0;

virtual int computeNumVCs(int vns);
virtual int getEndpointID(int port);

virtual void recvTopologyEvent(int port, TopologyEvent* ev);
```

Configuring a Merlin Simulation

Merlin/Ember Python Modules

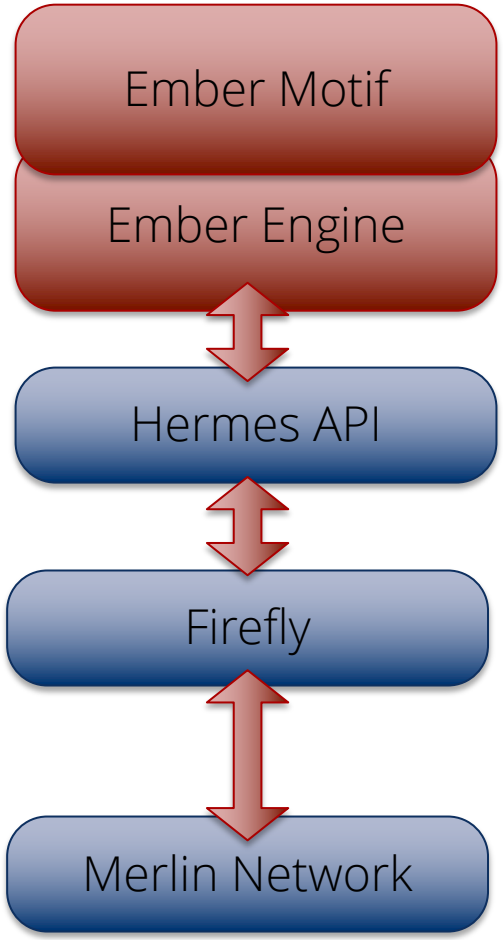
Merlin and Ember provide built-in Python modules to make configuring simulations easier

- Wires up the topology based on user supplied parameters
- Allows simulated jobs (endpoint models) to be easily allocated to the network

The Python modules are built off of five primary base classes:

- System – Overall system with functions to allocated jobs and build the simulation
- Topology – Controls the network topology and parameters
- RouterTemplate – Allows the ability to swap in different router models
  - In practice, there is only one current model: hr_router
- Job – Jobs are mapped to the system using the job allocation functions and contains all the parameters for running the given job on the system
- PlatformDefinition – allows you to capture all or part of the platform parameters and load them into a simulation instead of having to repeat this information in every input file

Ember – Lightweight Network Endpoint



High Level Communication Pattern and Logic
Generates communication events

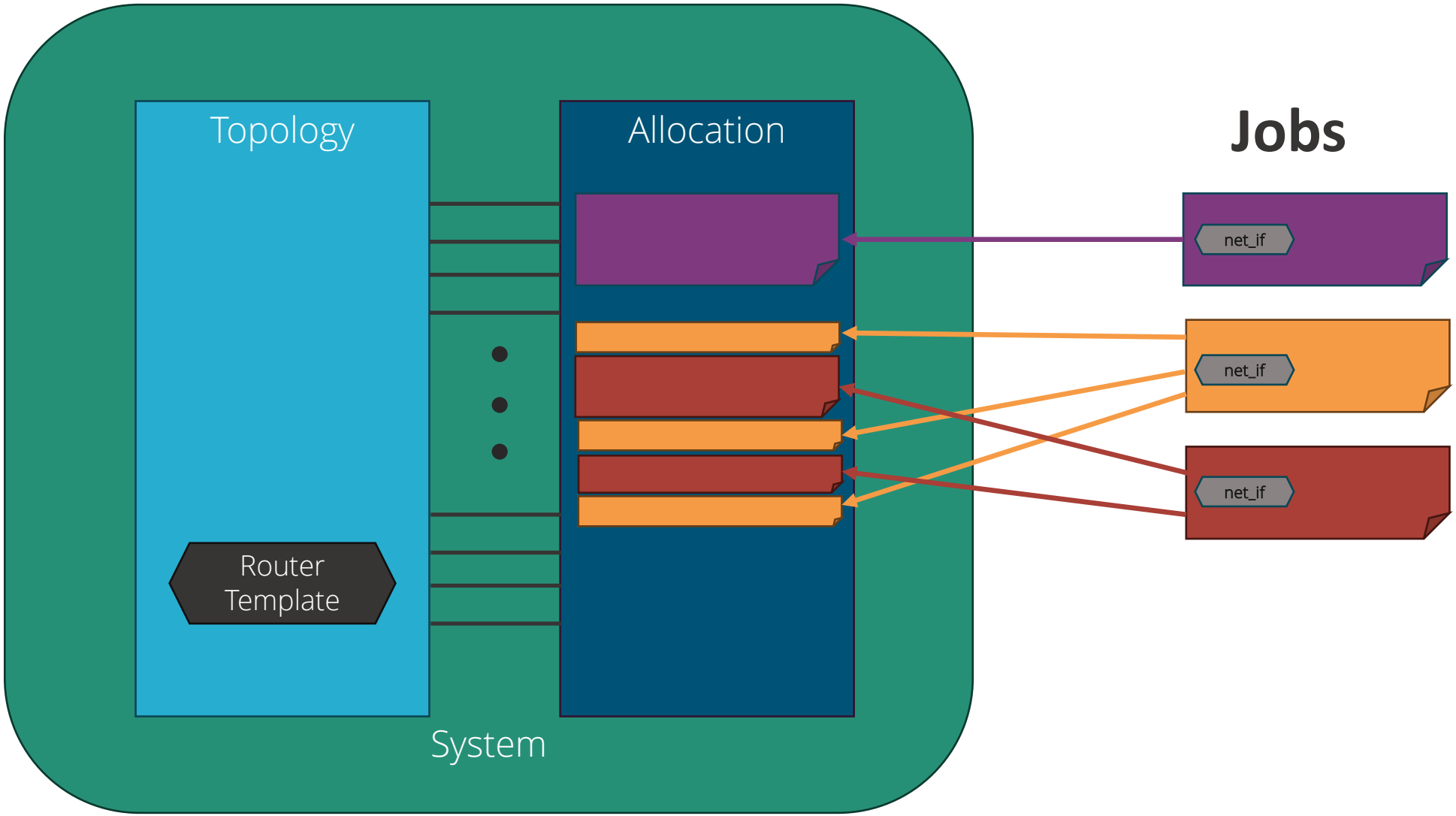
Event to Message Call, Motif Management
Handles the tracking of the motif

Message Passing Semantics
Collectives, Matching, etc.

Packetization and Byte Movement Engine
Generates packets and coordinates with network

Flit Level Movement, Routing, Delivery
Moves flits across network, timing, etc.

Block Diagram of Python Network Configuration Classes



System

After configuring the System, the build() function is called to generate the configuration graph. Configuration options (some of these can be set in a PlatformDefinition file):

- topology: set the topology to use. See next slide for available topologies
- allocation_block_size: sets the number of contiguous endpoints that will be used for allocation. Only currently used when simulating multiNIC nodes
- Allocate jobs to system: jobs are allocated using the allocateNodes() function. The following allocation algorithms can be used, and each job can use a different allocation using the remaining nodes in system:
  - Random – randomly allocate ranks to the available nodes
  - Linear – allocate nodes in order based on number used by topology
  - Random-linear – allocate linearly in a randomly selected subset of nodes
  - Interval – allocate N nodes every M endpoints (this is handy for reserving nodes for things like I/O nodes in the network)
  - Indexed – provide a list of ordered endpoints to allocate the job to (all endpoints in the list must still be unallocated in the system)

Topology

Controls the topology and all the high level network parameters (for example link bandwidth). The build function is called during the System.build() function

Can set the type of router to use by setting MyTopology.router, or can use the default (hr_router)

Supported topologies:

- Dragonfly – uses 1D all-to-all groups
- Hyperx - can be of any dimension, with any number of links per dimension
- Fattree – built using individual routers (no support for consolidated routers)
- Mesh/Torus – can be of any dimension, with any number of links per dimension
- SingleRouter – just a single router (used mostly to mimic crossbars on chip)

Topology: Mesh/Torus

Implements an N-dimensional torus or mesh

shape
- Gives the number of routers in each dimension
- Examples: 16x16x16, 4x4x4x4x4

width
- Gives number of links between routers in each dimension
- Example: 1x1x1, 4x8x4, 3x3x3x3x3

local_ports
- Gives number of hosts connected to each router

```
// Create a Torus topology
topology = topoTorus()

// Set shape to 3D torus with 16 routers per
// dimension
topology.shape = "16x16x16"

// 8 links between routers in each dimension
topology.width = "8x8x8"

// 16 endpoints per router
topology.local_ports = 16
```

Topology: HyperX/Flattened Butterfly

Implements and N-dimensional hyperX/flattened butterfly

shape
- Gives the number of routers in each dimension
- Examples: 16x16x16, 4x4x4x4x4

width
- Gives number of links between routers in each dimension
- Example: 1x1x1, 4x8x4, 3x3x3x3x3

local_ports
- Gives number of hosts connected to each router

```
// Create a hyperX topology
topology = topoHyperX()

// Set shape to 3D torus with 16 routers per
// dimension
topology.shape = "16x8x16"

// Keep same bisection BW for each dimension
// by doubling links in small dimension
topology.width = "1x2x1"

// 16 endpoints per router
topology.local_ports = 16
```

Topology: Dragonfly

Implements a dragonfly topology with fully connected local group

hosts_per_router
- Number of hosts connected to each router

routers_per_group
- Number of routers per local group

intergroup_links
- Number of links between each pair of groups

intragroup_links
- Number of links between each router pair in a group

num_groups
- Total number of groups in the topology

algorithm
- Routing algorithm to use
  - minimal, min-a, valiant, ugal

```
// Create a dragonfly topology
topology = topoDragonFly()

// Set network parameters
topology.hosts_per_router = 16
topology.routers_per_group = 16
topology.intragroup_links = 2
topology.intergroup_links = 8
topology.num_groups = 32

// Set the routing algorithm
topology.algorithm = "ugal"

// Get the total number of nodes
num_nodes = topology.getNumNodes()
```

Total number of hosts is:
- hosts_per_router * routers_per_group * num_groups

Topology: Fattree

Implements an N-level fattree

shape

- Specifies the number of up and down ports at each level of the fattree. Equal up and down links indicates no bandwidth tapering, while unequal numbers indicated tapering. Highest level only specified down links:
  - Specified down,up:down,up:down,up:down
    - Total number of nodes is computed by multiplying all the ”downs” together
  - Examples:
    - 18,18:18,18:36 (largest 3 level fattree using 36 port router with 11,664 hosts)
    - 24,12:18,18:18 (3 level fattree with 50% bandwidth taper out of first level with 7,776 hosts)

These logically match typical fattrees, but are not necessarily physically the same (for example, does not consolidate routers at top level when not all ports are used in each router)

RouterTemplate

Controls which router model is used when the topology is built.

Router parameters are set on the RouterTemplate object. For hr_router, the main parameters are:

- link_bw
- flit_Size
- xbar_bw
- num_vns
- input_latency
- output_latency
- input_buf_size
- output_buf_size

If using the default router, you can access the parameters of the RouterTemplate object as follows:

- MySystem.topology.router.link_bw = “100 Gbps”
- MySystem.topology.router.input_buf_size = “12 kiB”

Job

The Ember Python module uses the Job abstraction to easily configure Ember jobs using the Merlin network models.

To configure a set of motifs using MPI, use EmberMPIJob

- Constructor takes the following parameters:
  - job_id – unique ID to identify the job
  - num_nodes – number of network endpoints to use in the system
  - numCores – number of cores (MPPI ranks) per node to simulate
  - nicsPerNode – number of NICs to simulate per node
    - Each rank will get mapped to only one NIC, and numCores must divide evenly by nicsPerNode
    - The System allocation block size must be a multiple of the nicsPerNode
- Must set the nic parameter to be the network interface corresponding to the router used in topology (for hr_router either LinkControl or ReorderLinkControl)
- Add motifs to the job using the addMotif() function. Examples:
  - myJob.addMotif("Halo3D26 pex=32 pey=32 pez=32 nx=20 ny=20 nz=20 iterations=1")
  - myJob.addMotif("Allreduce")
- There are a large number of other parameters which are best set in a PlatformDefinition file (see next slide)

PlatformDefinition

The various objects in the simulation will “subscribe” to a set of named parameters and class types that can be used to configure the simulation objects, which can set using the PlatformDefinition interface:

- PlatformDefinition.loadPlatformFile(”firefly_platform")
  - Loads a python file that contains platform definitions
  - Platform files can contain more than on platform definition
  - Multiple platform files can be loaded
- PlatformDefinition.setCurrentPlatform(”firefly-platform-base")
  - Set the specified definition as the current platform (can only load one platform at a time)
- PlatformDefinition.compose(platformName, ...)
  - Allows you to compose multiple PlatformDefinitions into a new PlatformDefinition
    - For example, ember parameters and network definition can come from different platform files and be composed together since you can only have one active platform

Platform files can be provided by Sandia, vendors, etc.

Example Configuration using Ember

Example Configuration: merlin-ember-example.py

```
# Import base sst module
import sst

# Import necessary merlin modules
from sst.merlin.base import *
from sst.merlin.endpoint import *
from sst.merlin.interface import *
from sst.merlin.topology import *

from sst.ember import *

# Include the firefly defaults to get default
# parameters for NIC and network stack
PlatformDefinition.setCurrentPlatform(
   "firefly-defaults")
```

```
# Setup the topology
topo = topoDragonFly()
topo.hosts_per_router = 2
topo.routers_per_group = 4
topo.intergroup_links = 2
topo.num_groups = 4
topo.algorithm = ”ugal"]
topo.link_latency = "20ns"

# Set up the routers
router = hr_router()
router.link_bw = "4GB/s"
router.flit_size = "8B"
router.xbar_bw = "6GB/s"
router.input_latency = "20ns"
router.output_latency = "20ns"
router.input_buf_size = "4kB"
router.output_buf_size = "4kB"
router.num_vns = 1
router.xbar_arb = "merlin.xbar_arb_lru"

# Add router template to topology
topo.router = router
```

Example Configuration: merlin-ember-example.py

```
# Set up the network interface
networkif = ReorderLinkControl()
networkif.link_bw = "4GB/s"
networkif.input_buf_size = "4kB"
networkif.output_buf_size = "4kB"

# Create the ember job
ep = EmberMPIJob(0,topo.getNumNodes())
ep.network_interface = networkif
ep.addMotif("Init") # required first motif
ep.addMotif("Allreduce")
ep.addMotif("Fini") # required last motif
ep.nic.nic2host_lat= "100ns"

# Create the system object
system = System()
system.setTopology(topo)

# Allocate the job using linear placement
system.allocateNodes(ep,"linear")

# Build the system
system.build()
```

```
% sst merlin-ember-example.py
Allreduce: ranks 32, loop 1, 1 double(s), latency 5.192 us
Simulation is complete, simulated time: 18.7055 us
```

Hands On with Merlin and Ember

Things to try:

- Change the ember motif being run; possible other motifs to run:
  - Halo3d26
  - sweep3D
- Change some of the network parameters
  - link_bw, input/output latencies, etc
- Use a different routing algorithm
  - minimal, min-a, or valiant
- Try random allocation of the job
- Double the size of the network and add a second job
  - Try different allocations (linear, random, random-linear)

Extending the functionality of Merlin

Creating a new topology for Merlin

Two approaches to adding functionality to merlin

- Add features to the merlin library directly and submit changes for merging
- Create an external element library with the changes and distribute library as a third party plug-in
  - Merlin installs all header files necessary to create a new implementation of its SubComponent APIs

When adding a new topology, there are two main pieces of functionality to implement:

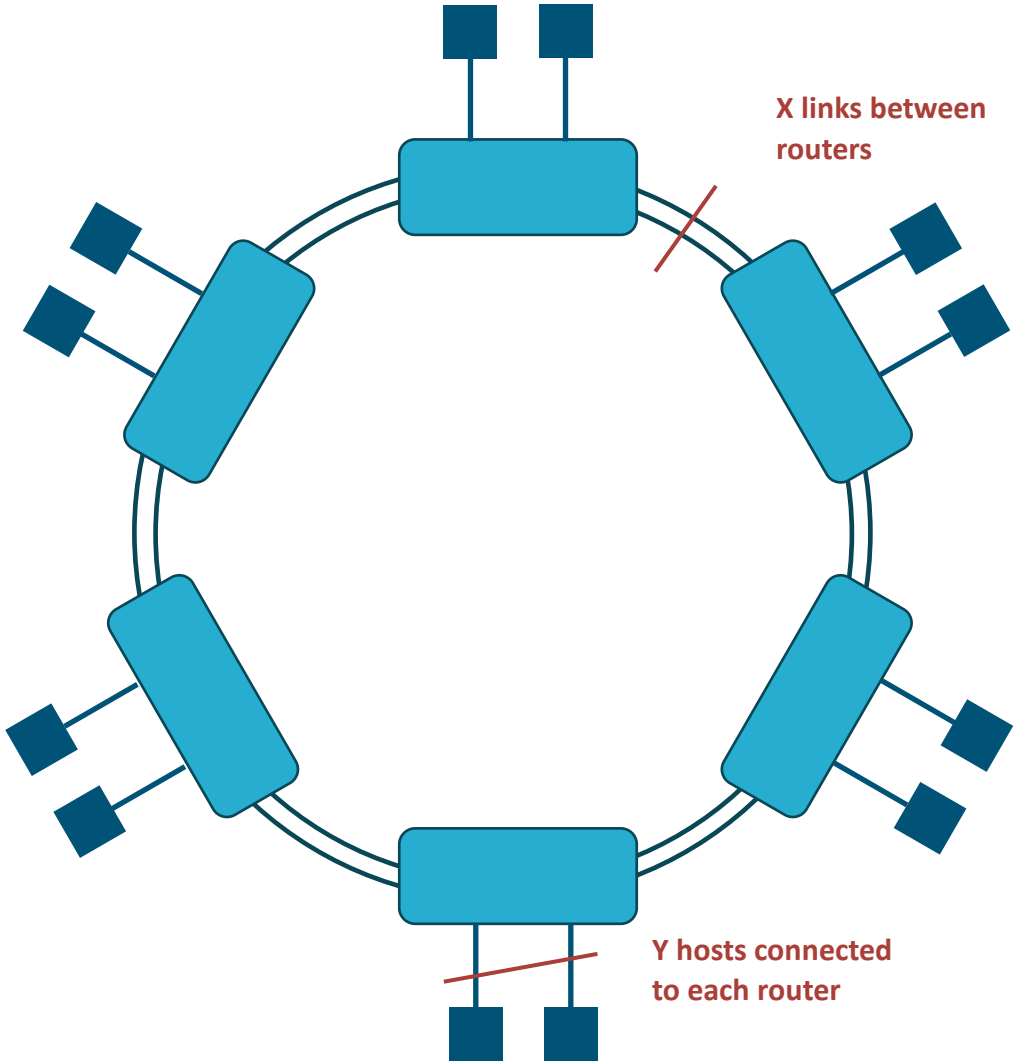
- C++ simulation model inheriting from Merlin::Topology
  - Defined in sst/elements/merlin/router.h
- Python configuration help class inherited from merlin.base.Topology
  - Defined in sst/elements/merlin/pymerlin-base.py
    - Making this Python module available to the interpreter through the ELI requires you to create a C++ class inheriting from SST::SSTElementPythonModule
      - Defined in sst/core/model/element_python.h

Example: Ring Topology

Simple ring topology with multiple links between routers and multiple endpoints connected to each router

Topology Parameters:

- num_routers – total number of routers
- interrouter_links – number of links between routers
- local_ports – number of hosts per router



Let's look at the code: archimedes element library

Makefile – Makefile to build the libarchimedes.so element library

topo_ring.h – Header file for the ring topology object and events

topo_ring.cc – Source file for the ring topology implementation

- Also includes code to register the archimedes python module with SST core

pyarchimedes.py – Implementation of the archimedes Python module

ring_test.py – Input file to test the ring topology

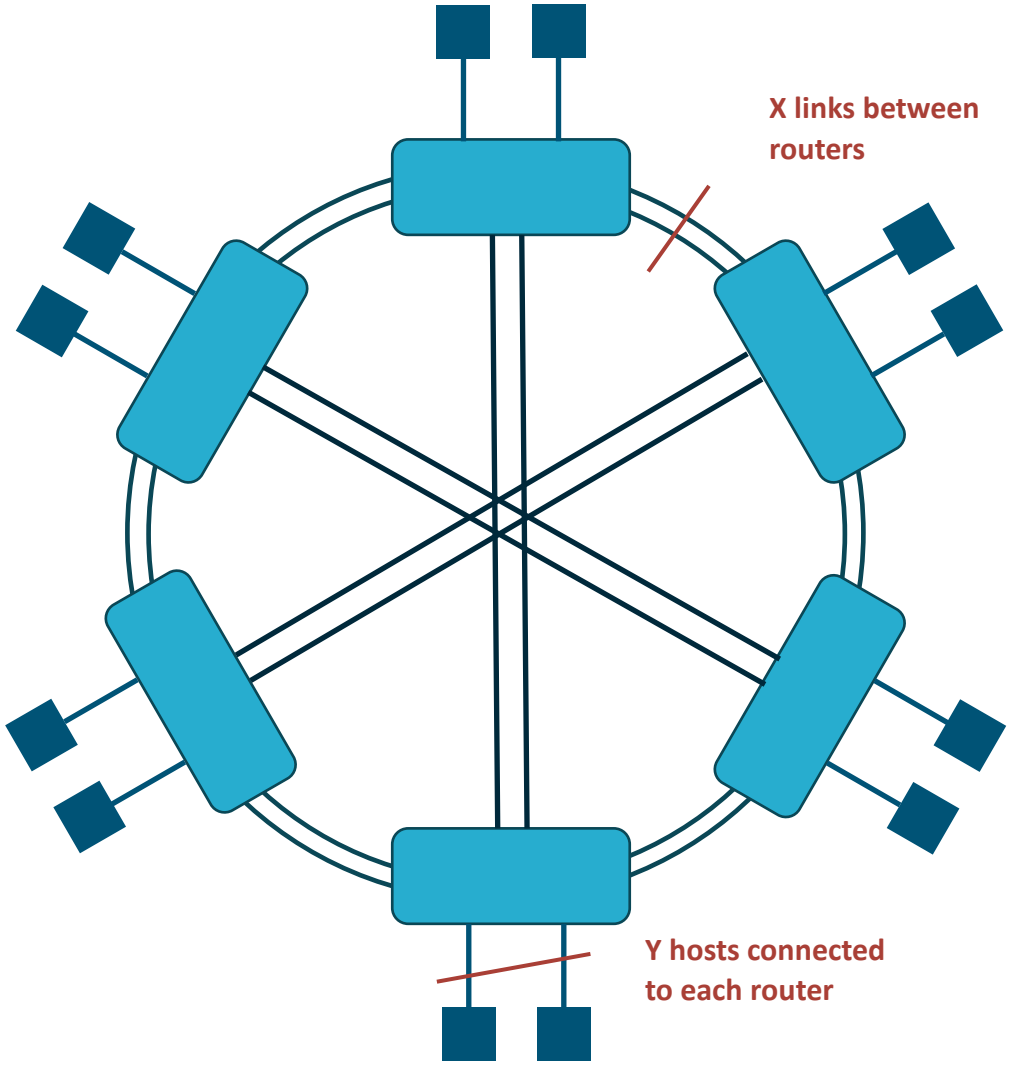
Hands on: Modify topology to include cross links

Try adding cross links to the topology to cut average hop count in half.

- May want to limit topology to even number of routers only

Will need to modify:

- Routing function in SubComponent model
  - Depending on implementation, may need to modify some of the other functions in topo_ring class.
- _build_impl() function in the Python template class



Sandia National Laboratories

Thanks!

Exceptional Service In The National Interest

Join Us!

We are always looking for new staff and new collaborations…

- Design challenges in the post-exascale era requires that we draw from a diverse pool of talent across multiple disciplines!

If you're interested, check out https://www.sandia.gov/careers/career-possibilities/career-opportunities/computer-science/, or contact Clay at chughes@sandia.gov

Backup

Running a Simulation – Add Components, Simple Node

Copy configuration

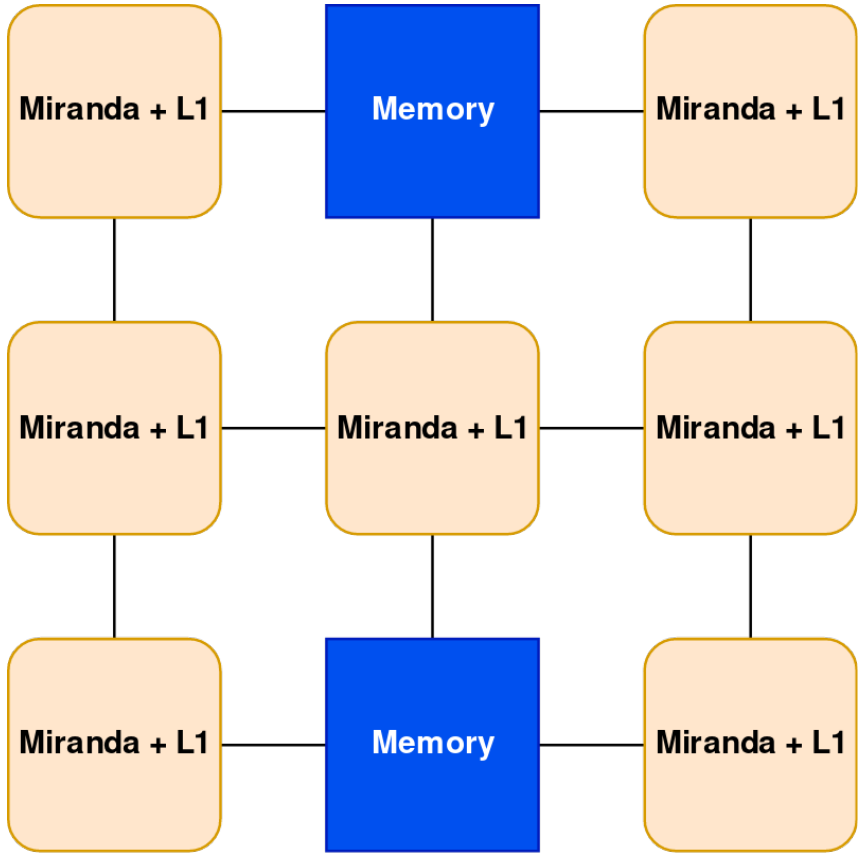
```
$ cp demo_6.py demo_7.py
```

Add the components to create a simple node

- Each PE has a Miranda generator and an L1
- The two memory nodes should use timingDRAM
- How should you connect everything?

Launch simulation

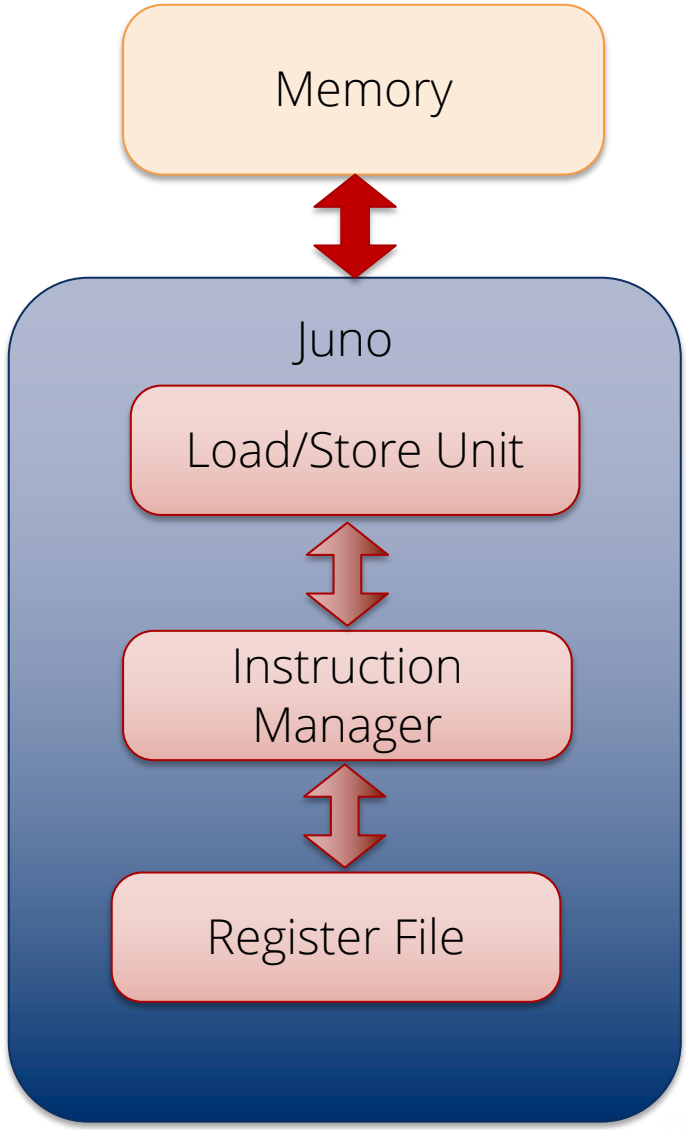
```
$ sst demo_7.py
```



Juno: Simple instruction processor

Executes a program written in simple “assembly”

- 32-bit wide instructions with 8 bit op codes
- 64-bit integer operations
  - ADD, SUB, DIV, MUL,
  - AND, OR, XOR, NOT
- Jump by register value (JGT-Zero, JLZ-Zero, J-Zero)
  - Jump up to16 bits in either direction from current PC
- Up to 253 user registers
  - r0 = PC
  - r1 = data start register



Running a Simulation – Add Components, Second Memory

Copy configuration

```
$ cp demo_6.py demo_7.py
```

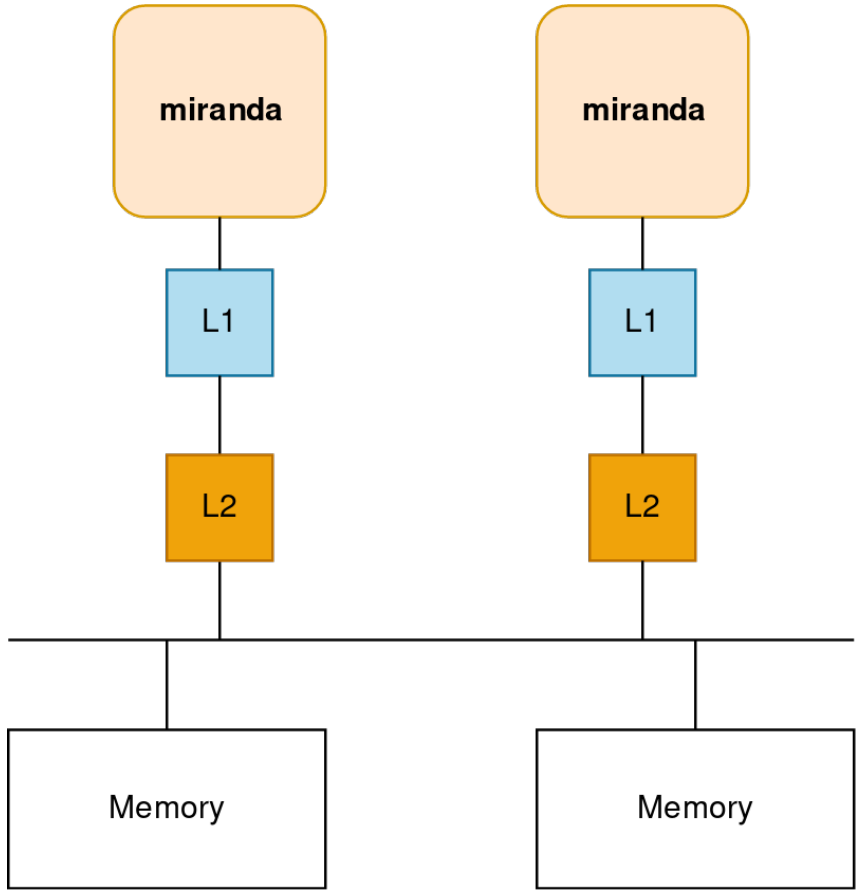
Add an a second L2 and memory controller

- Think carefully about how addressing should will work…

```
Can you still use a bus?
Can the talk directly to the memory controller?
```

Launch simulation

```
$ sst demo_7.py
```



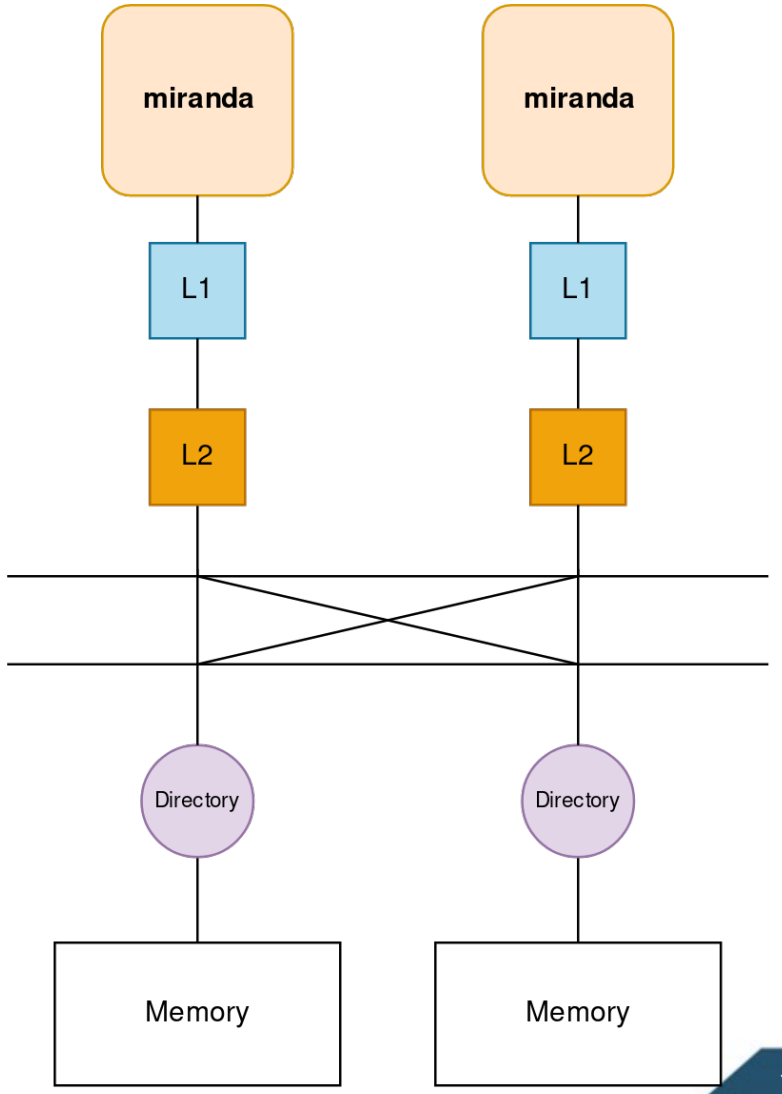
Running a Simulation – Add Components, Second Memory

Swap the bus component for the shogun component

```
shogun_xbar = sst.Component("shogunxbar",
"shogun.ShogunXBar")
shogun_xbar.addParams({
      "clock" : "1.0GHz",
      "port_count" : 4,
      "verbose" : 0
})
```

Add directory controllers before the memory

```
dirctrl = sst.Component("dirctrl_" + str(cache_id), "memHierarchy.DirectoryController")
 dirctrl.addParams({
      "coherence_protocol" : "MESI",
      "entry_cache_size" : "32768",
      "addr_range_end" : endAddr,
      "addr_range_start" : startAddr,
      "interleave_size" : "256B",
      "interleave_step" : str(numLLC * 256) + "B",
 })
 dc_cpulink = dirctrl.setSubComponent("cpulink", "memHierarchy.MemNIC")
 dc_memlink = dirctrl.setSubComponent("memlink", "memHierarchy.MemLink")
 dc_cpulink.addParams({
   "group" : 3,
 })
 dc_linkctrl = dc_cpulink.setSubComponent("linkcontrol", "shogun.ShogunNIC")
```



Slide Left Blank