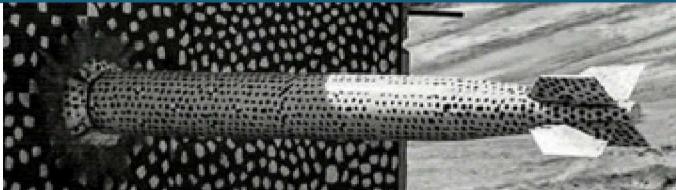


The Structural Simulation Toolkit



PRESENTED BY

Clay Hughes

SST Development Team, Sandia National Laboratories



SAND2019-15190PE



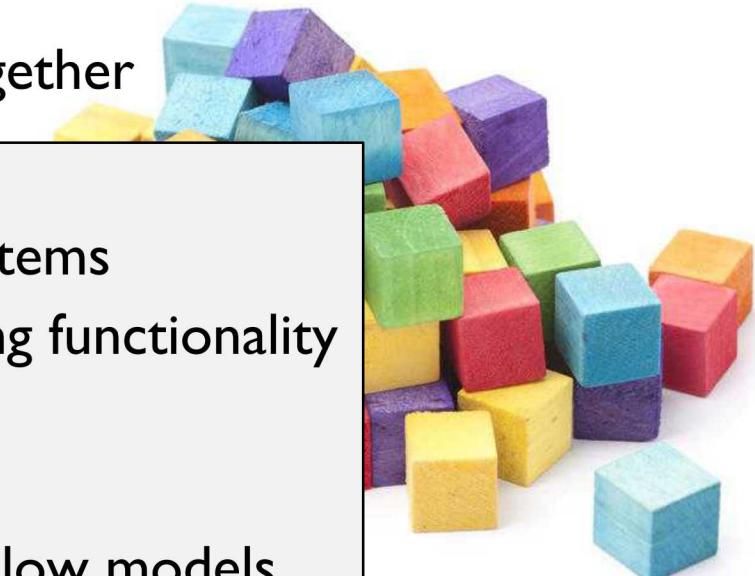
Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

References

- Websites
 - <http://www.sst-simulator.org/>
 - <https://github.com/sstsimulator>
- Configuration File Format:
 - <http://sst-simulator.org/SSTPages/SSTUserPythonFileFormat/>
- Doxygen Documentation:
 - http://sst-simulator.org/SSTDoxygen/9.1.0_docs/html/
- Developer FAQ:
 - <http://sst-simulator.org/SSTPages/SSTTopDocDeveloperInfo/>
- Building SST
 - <http://sst-simulator.org/SSTPages/SSTBuildAndInstall9dot1dot0SeriesQuickStart/>
 - <http://sst-simulator.org/SSTPages/SSTBuildAndInstall9dot1dot0SeriesDetailedBuildInstructions/>

So many simulators, so little interoperability

- Already 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 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



4 | 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

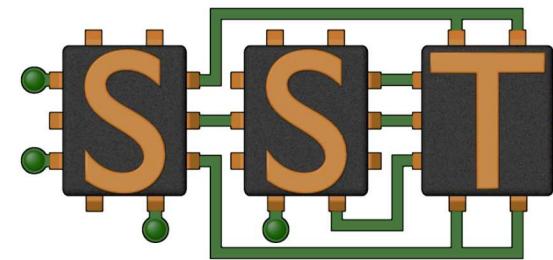
- Parallel framework (*SST Core*)
- Integrated libraries of components (*Elements*)
- Current Release (9.1.0)
 - <https://sst-simulator.org>
 - <https://github/sstsimulator>

Contributors



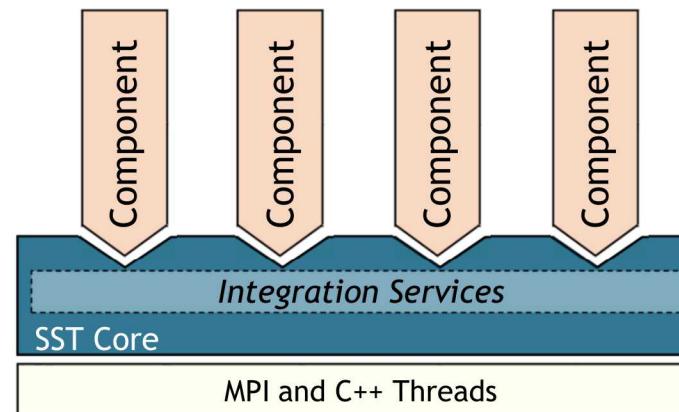
The SST Approach

- 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



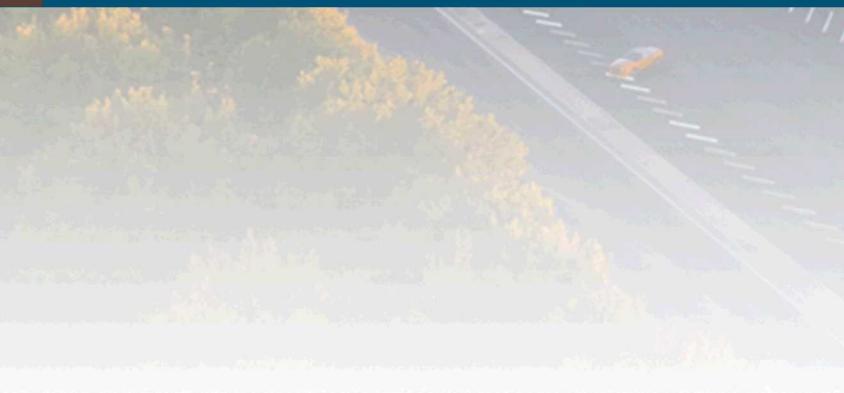
SST Architecture

- 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.



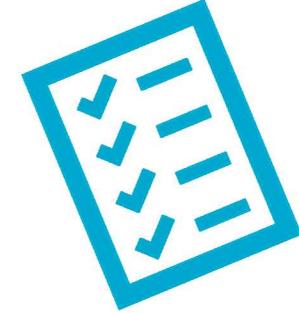


SST Elements: A Tour



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
 - See www.sst-simulator.org for more information
- You can also register your own elements

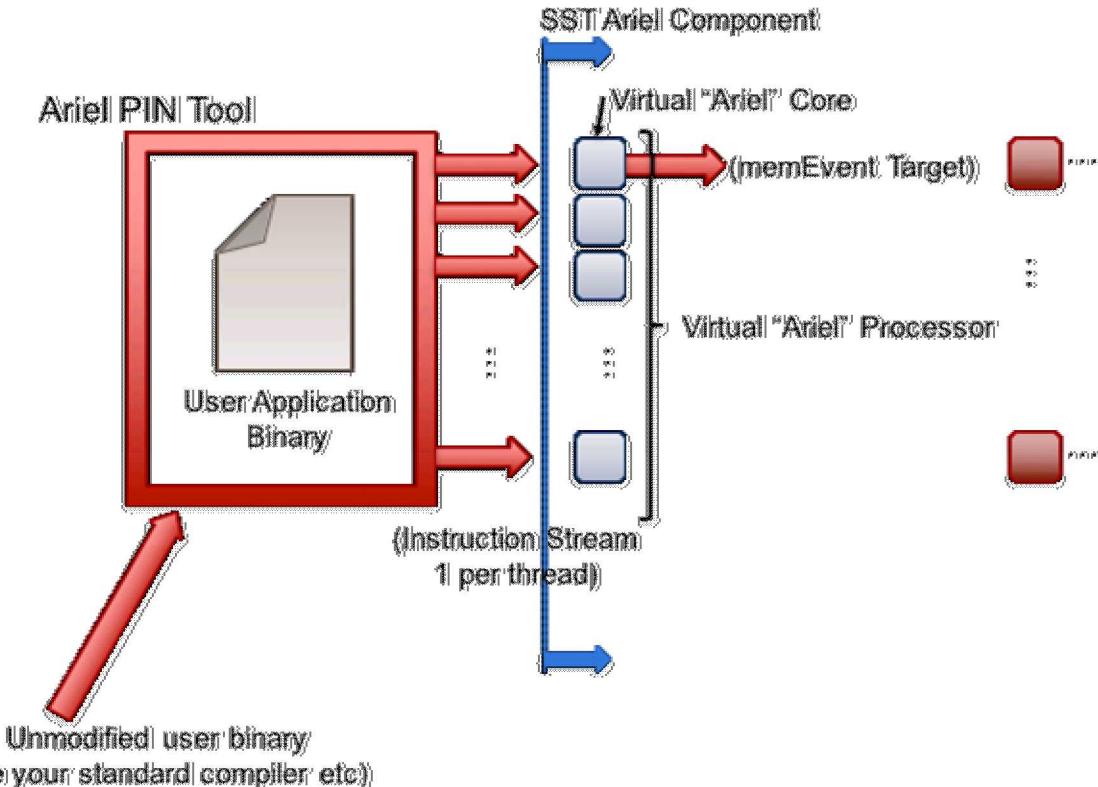


SST 9.1 Elements

- Processors
 - Ariel – PIN-based
 - Juno – simple ISA processor
 - Miranda – pattern generator
 - Prospero – trace execution
 - GeNSA – spiking temporal processing unit
- Memory Subsystem
 - cacheTracer – cache tracing
 - Cassini – cache prefetchers
 - CramSim – DDR, HBM
 - MemHierarchy – caches, directory, memory
 - Messier - NVM
 - Samba – TLB
 - VaultSimC – vaulted stacked memory
- Networks/NoCs
 - Merlin – flexible network modeling
 - Kingsley – mesh NoC
 - Shogun – crossbar NoC
- Network drivers
 - Ember – communication patterns
 - Firefly – communication protocols
 - Hermes – MPI-like driver interface
 - Zodiac – trace based driver
 - Thornhill – memory models for Ember sims
- Others
 - sst-macro – network drivers/network
 - scheduler – job scheduling
 - simpleSimulation – “car wash” example
 - simpleElementExample – many examples
 - sst-external-element – example element
 - balar – GPU model

Ariel: PIN-based processor

- 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

- 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 simpleMem interface
 - Generates SimpleMemRequests
 - Compatible with memHierarchy



Ariel: The Tradeoff

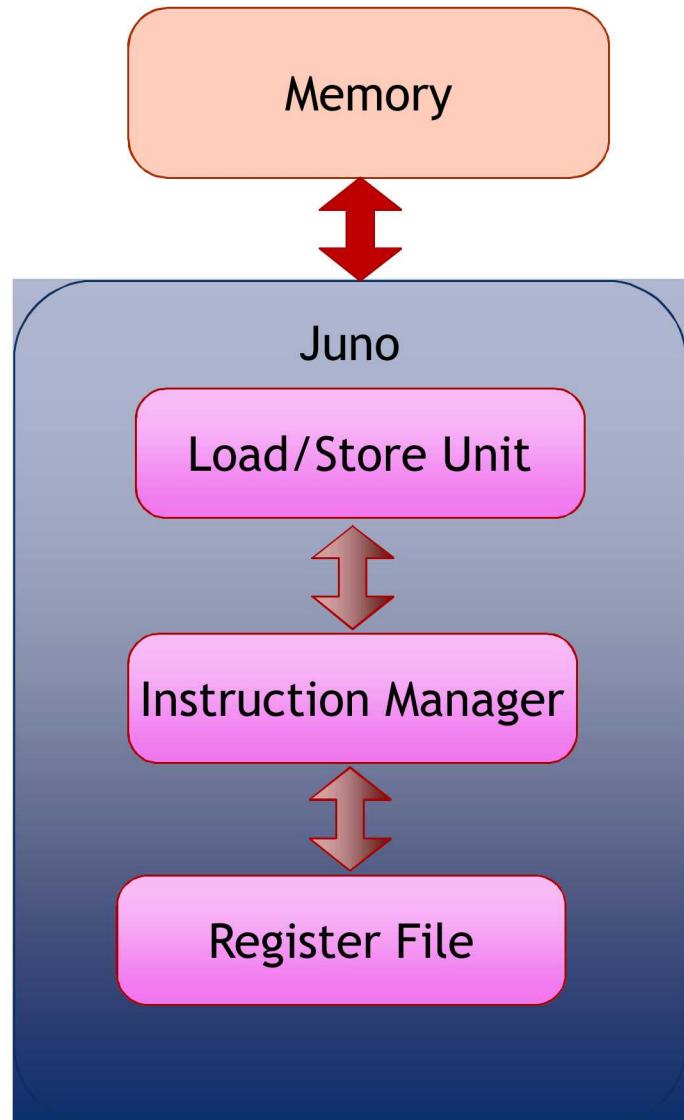


- Pros:
 - Faster than more complex/pipeline models
 - Reasonable approximation for studies on memory system performance
 - Especially for heavily memory-bound applications
 - Reasonable model of thread interactions
- Cons
 - Non-deterministic results
 - Interactions between pintool, threads, etc.
 - Variation is low ($O(1\%)$)
 - Not compatible with non-x86 binaries
 - Reliant on Pin 2.14
 - Currently working towards enabling Ariel to be used with other drivers



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 to 16 bits in either direction from current PC
- Up to 253 user registers
 - r0 = PC
 - r1 = data start register



Miranda: Pattern-based processor

- 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

Processor

Memory

Network/NoC

Network driver

Other

Miranda: The tradeoffs

- 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

Processor

Memory

Network/NoC

Network driver

Other

Prospero: Trace-based processor

- 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

Processor

Memory

Network/NoC

Network driver

Other

Prospero: The Tradeoff

- 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

Processor

Memory

Network/NoC

Network driver

Other

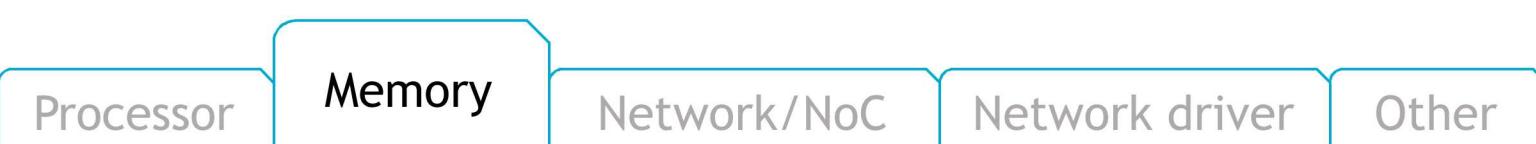
MemHierarchy: Memory system

- 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



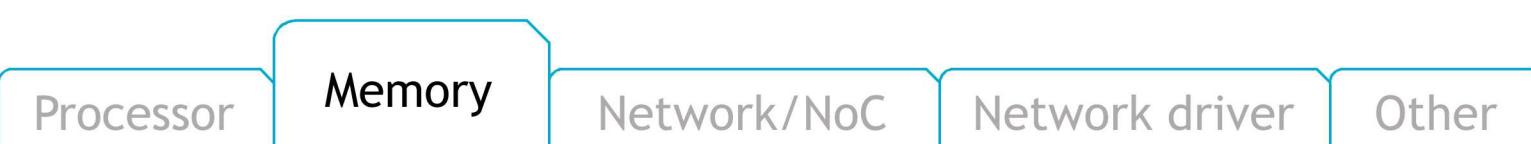
MemHierarchy: Cache modeling

- 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.



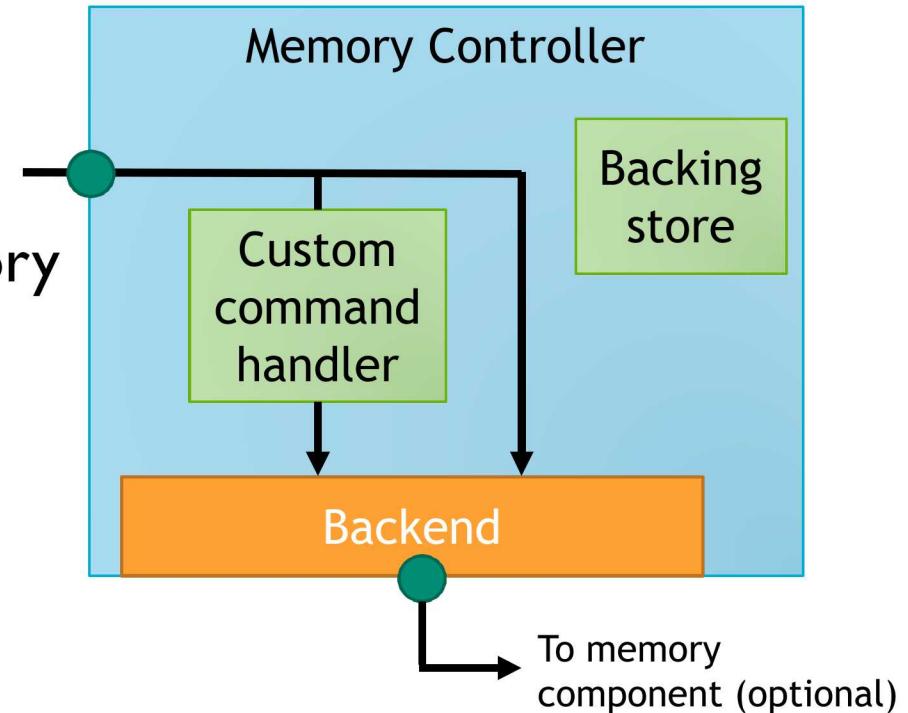
MemHierarchy: Memory modeling

- Interface to memory is the *MemController*
- MemControllers implement *backends*
 - Timing model for memory controller with a link to memory
 - Timing model for memory controller and memory
 - Interface to another component(s) that does the memory controller/memory timing
 - In this case just translates request formats
 - Wrapper for an external/non-native-SST component
 - Ramulator, DRAMSim2, etc.
- Support *custom memory instructions*
 - Including ability to do cache shootdowns for coherence maintenance



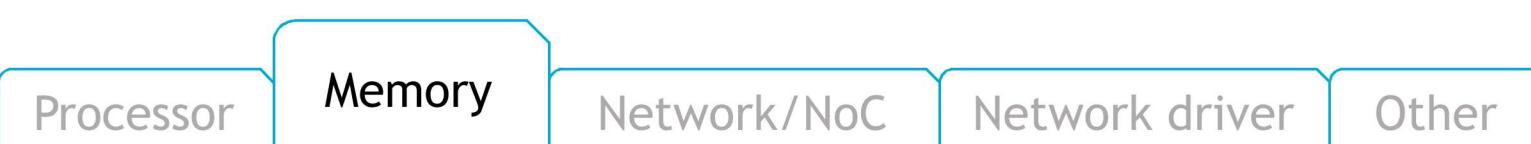
MemHierarchy: Memory modeling

- Memory controller
 - Manages data values if needed (backing store)
 - Facilitates custom memory commands
 - Including cache shutdowns 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 9.1 backends

- Memory (external)
 - CramSim (DDR, HBM)
 - DRAMSim2 (DDR)
 - PagedMulti – 2-level memory variant
 - FlashDIMMSim (FLASH)
 - HMCSim/GoblinHMC (HMC)
 - HBMDRAMSim2 (HBM)
 - HBMPagedMulti – 2-level memory variant
- Plus a few that can be used with other backends to reorder requests, add latency, etc.
 - Messier (NVRAM)
 - Ramulator (DDR, HBM, HMC)
 - SimpleDRAM (DDR)
 - SimpleMem (constant latency)
 - TimingDRAM (DDR)
 - VaultSimC (HMC-like)



Merlin: Network simulator

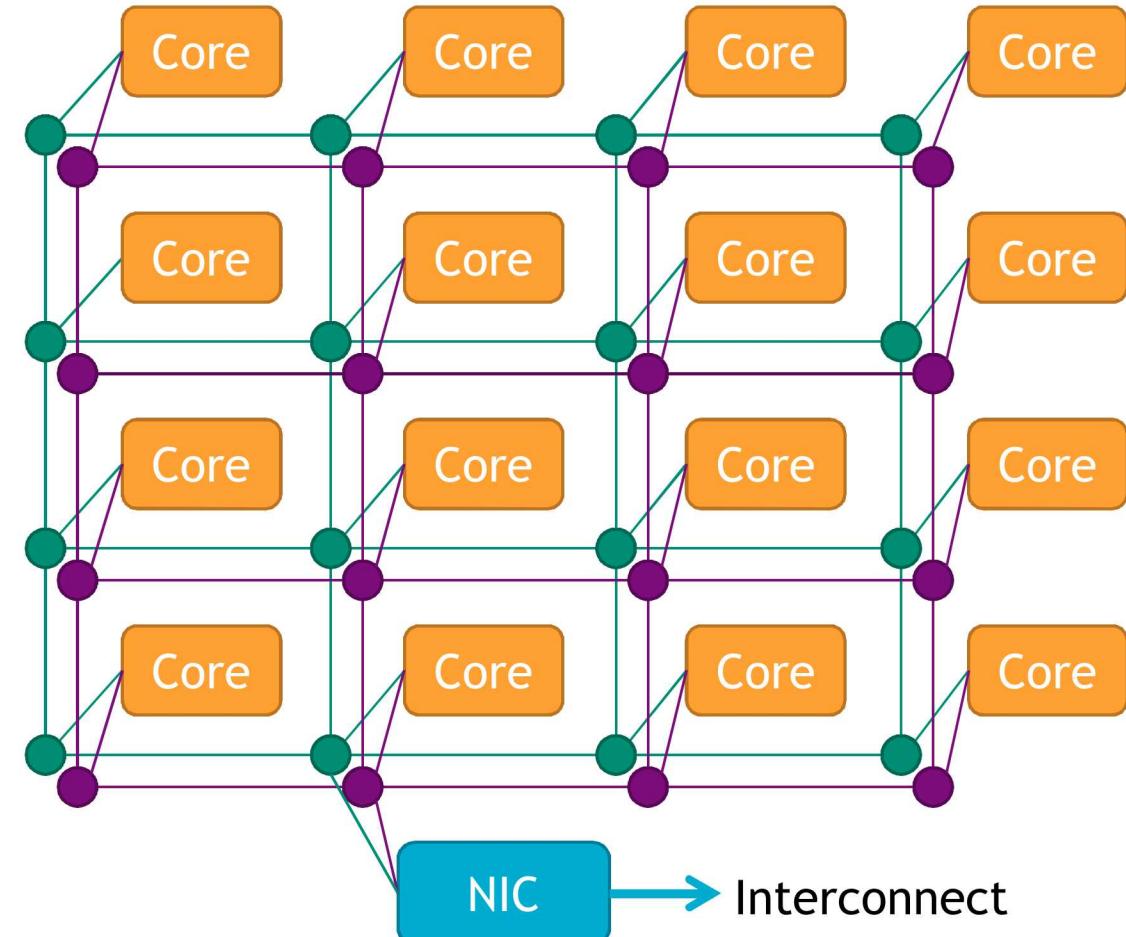


- Low-level networking components that can be used to simulate high-speed networks (machine level) or on-chip networks
- Capabilities
 - High radix router model (`hr_router`)
 - Topologies – mesh, n-dim tori, fat-tree, dragonfly
- Many ways to drive a network
 - Simple traffic generation models
 - Nearest neighbor, uniform, uniform w/ hotspot, normal, binomial
 - *MemHierarchy*
 - Lightweight network endpoint models (*Ember* – coming up next)
 - Or, make your own



Kingsley: Mesh simulator

- 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

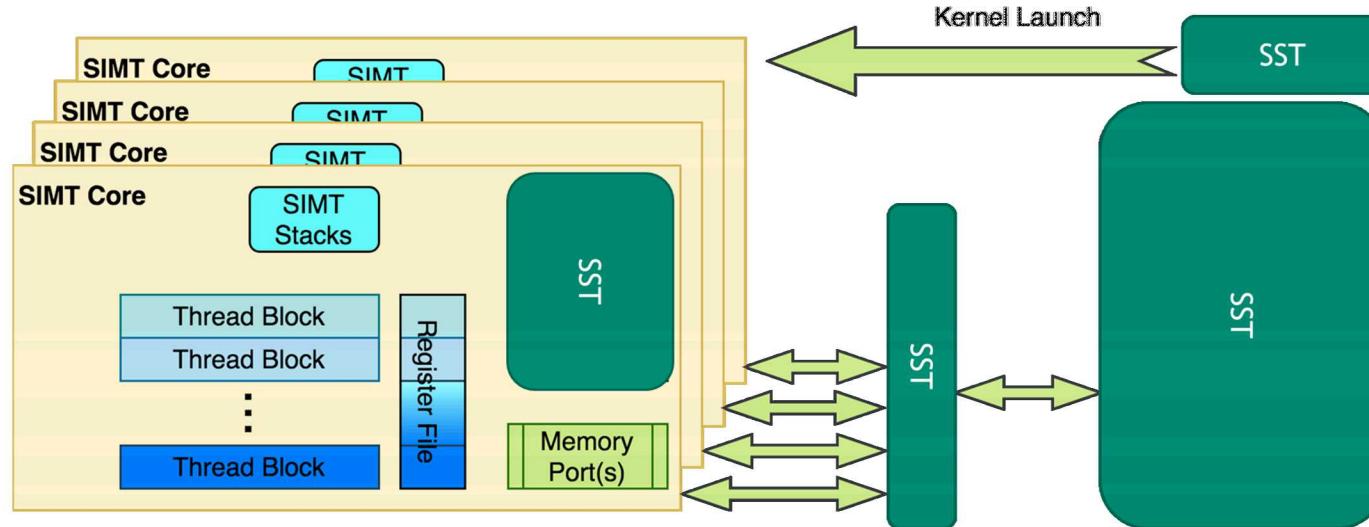


Scheduler: Job scheduling

- Models HPC system-wide job scheduling
- Three components
 - **Sched**: schedules and allocates resources for a stream of jobs
 - **Node**: runs scheduled jobs on their allocated resources
 - **FaultInjection**: injects failures onto the resources
- The scheduler can be a stand-alone element library
 - The schedComponent and nodeComponent must be used together
 - The faultInjectionComponent is optional
- Can be used with Ember/Firefly/Merlin stack
 - Examine topology aware scheduling and allocation

balar: GPGPU Model

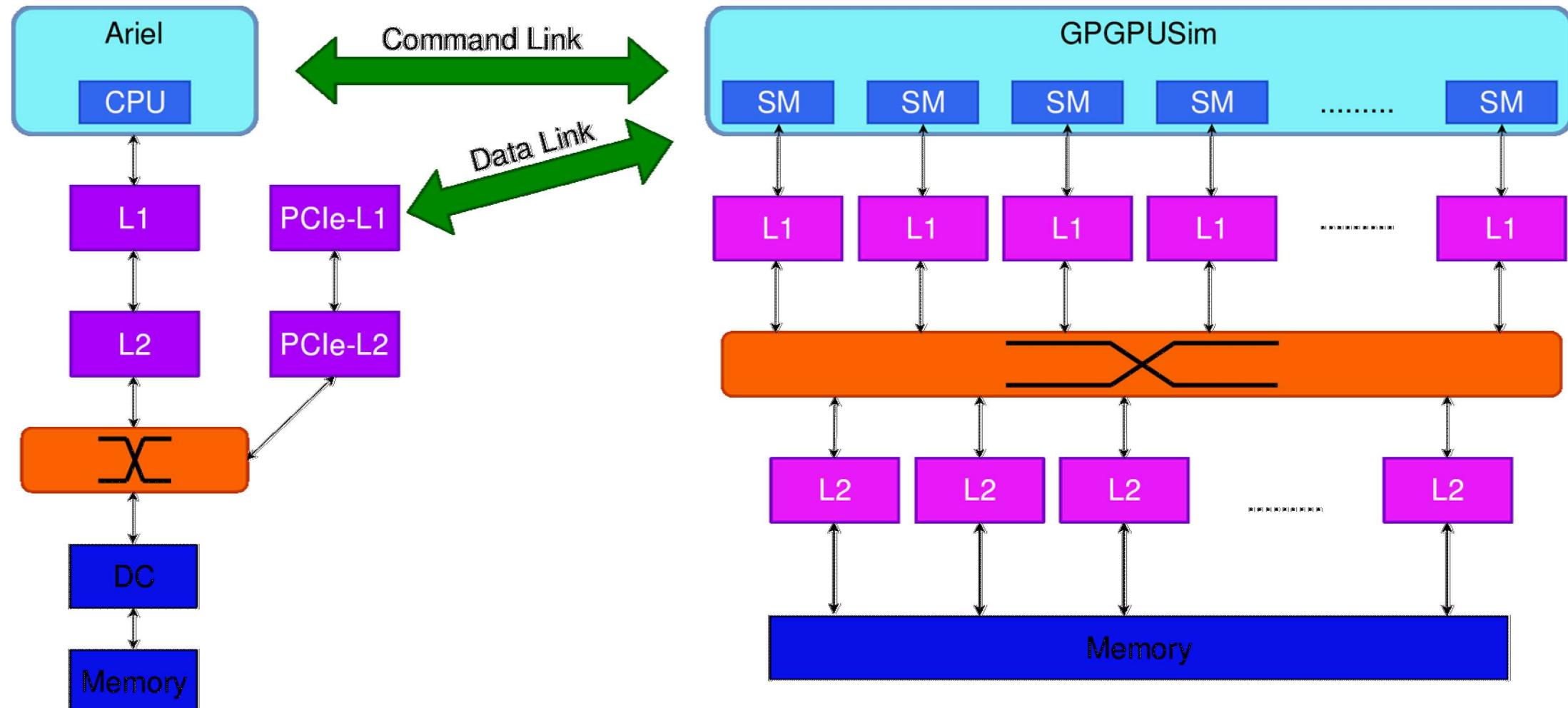
- Derived from GPGPU-Sim (https://github.com/gpgpu-sim/gpgpu-sim_distribution)



- keep the SIMT units and replace everything else
 - CPU → SST execution component (Ariel, Juno, etc.)
 - Interconnect → SST networking component (Shogun, Merlin, Kingsley, etc.)
 - Memory Partition & Caches → SST memHierarchy component for caches and various other backends for the backing store (DRAMSim, SimpleMem, Cramsim, etc.)

balar: GPGPU Model

- GPGPU-Sim Integration

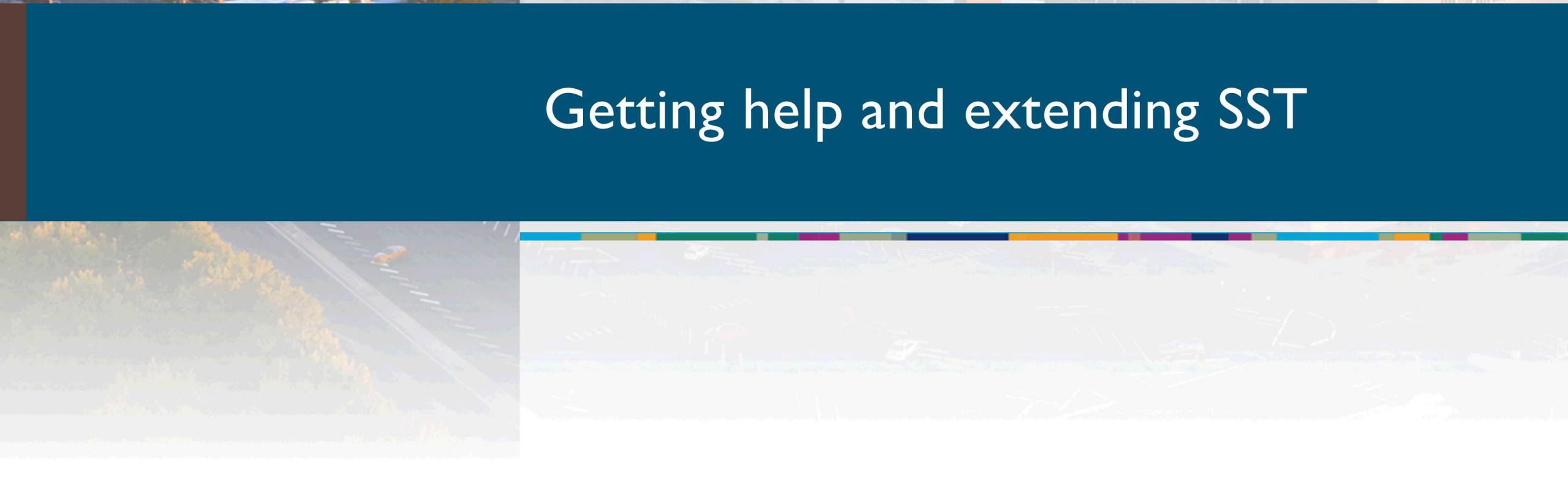


Other Libraries

- More information on these and other element libraries and external components is available on the wiki
 - www.sst-simulator.org



Getting help and 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>
- Website
 - *Getting Started Extending SST (a little out of date)*
 - *Building Element Libraries outside SST source tree*
 - Past tutorial material (under Downloads)
 - <sst-simulator.org/SST-website> API documentation

Finally: Getting help

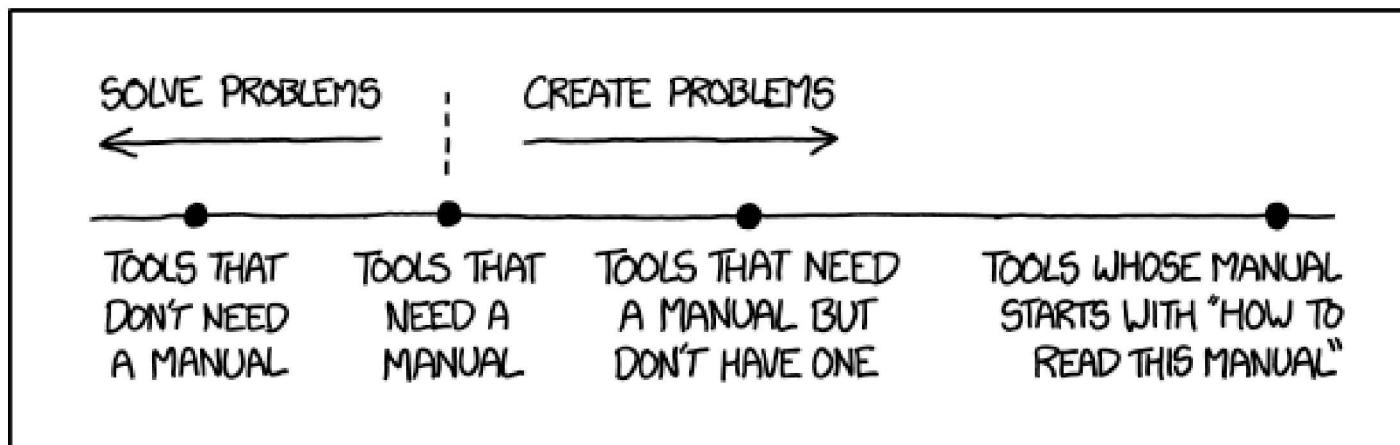
- SST wiki contains lots of information (www.sst-simulator.org)
 - Downloading, installing, and running SST
 - Element libraries and external components
 - Guides for extending SST
 - Information on APIs
 - Information about current development efforts
 - Past tutorial slides and exercises
- SST Github
 - Current development
 - Issues track user questions as well as development plans, bugs, etc.

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?





Running a simulation and code orientation

High-Level View

Getting and Installing SST

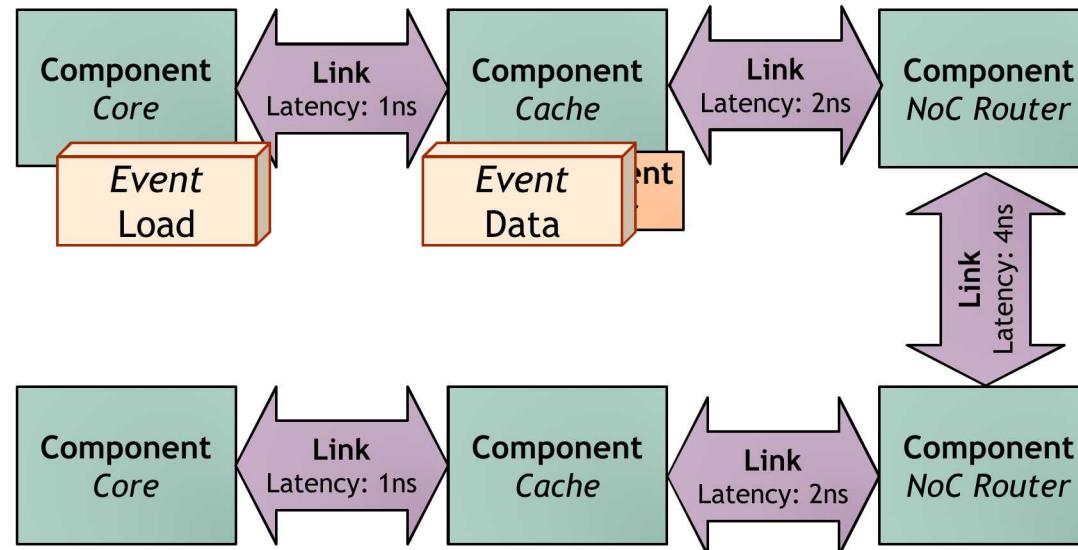
- <http://www.sst-simulator.org>
 - Current release source download
 - Detailed build instructions including dependencies for Linux & Mac
 - Archived tutorial materials
- <https://github.com/sstsimulator>
 - Source code checkout
 - Master branch – has passed testing
 - Devel branch – has passed *basic* testing

The screenshot shows the SST Simulator website at sst-simulator.org. The header features the SST logo and the text "The Structural Simulation Toolkit" with the subtitle "Using the supercomputers of today to build the supercomputers of tomorrow". A navigation bar includes links for Home, Downloads, Documentation, and Support. The main content area is titled "Introduction to SST" and contains the subtext "Exploring novel computer system designs". On the left, there's a sidebar with links for Home, Downloads, and Documentation.

The screenshot shows the SST GitHub repository page. The header includes the GitHub logo and the text "SST Simulator". The main content area displays the repository details for "Structural Simulation Toolkit". It shows 20 repositories, 37 people, 9 teams, and 0 projects. The "Pinned repositories" section lists three repositories: "sst-elements", "sst-core", and "sst-sqe". Each pinned repository card includes a brief description, programming language (C++), stars (21 or 4), and forks (26 or 15).

Building Blocks

- SST simulations are comprised of **components** 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

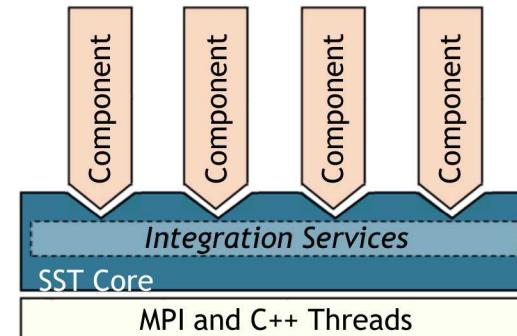


Element Library
A collection of components, subcomponents, and/or modules

SST Code Structure

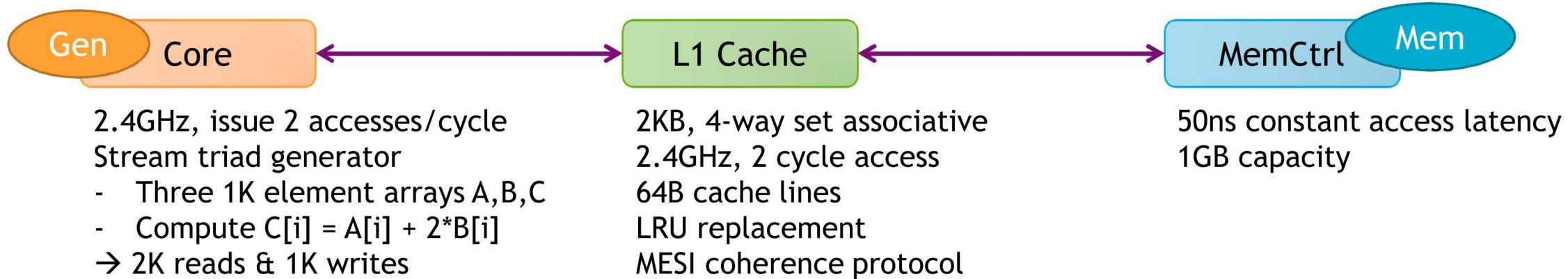


- **SST Core** and **SST Elements** are compiled separately
 - Element libraries *register* with the core
 - External elements (not part of SST Elements) can also be registered with the core
 - Example at github.com/sstsimulator/sst-external-element
 - Core maintains a database of registered libraries
 - 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
 - Often a good starting point for example configurations



Simulating with SST

- We'll walk through how to configure a simulation and then run it
 - Available at: <https://github.com/sstsimulator/sst-tutorials/tree/master/gt2019>
 - (github → sstsimulator → sst-tutorials → gt2019)
- Element libraries in our example simulation
 - **Miranda** - Simple core model that runs generated instruction streams
 - Generators produce memory access patterns (SubComponents)
 - **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.setProgramOption("stopAtCycle", "100ms")`
 - End simulation at 100ms if it hasn't ended already
- Other options
 - Most are also available as command line arguments to SST

SST Python API

User-defined string

SST argument

Option	Definition
debug-file	File to print debug output to
heartbeat-period	If set, SST will print a heartbeat message at the specified period
timebase	Units of simulation. Default is picoseconds which enables 2/3 of a year.
partitioner	Partitioner to use for parallel execution
output-partition	File to print partition to

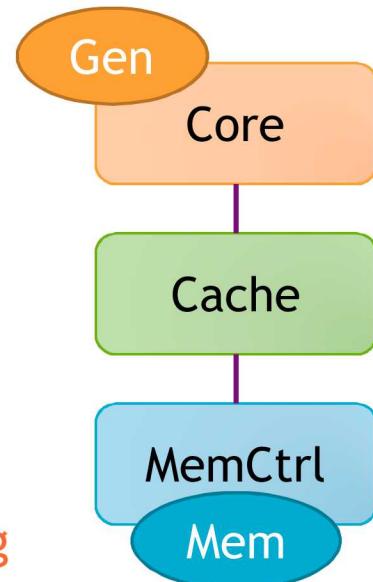
Configuration File: Declare components and links

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

```
core = sst.Component("core", "miranda.BaseCPU")
cache = sst.Component("L1", "memHierarchy.Cache")
mctrl = sst.Component("memctrl", "memHierarchy.MemController")
```

Component name

Component library.type



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

```
link0 = sst.Link("core_to_cache")
link1 = sst.Link("cache_to_memory")
```

Link name

SST Python API
User-defined string
SST argument

Configuration File: Configure the components

- **Parameters:** addParams({ “parameter” : “value”, ... })

```
core.addParams({ “clock” : “2.4GHz” })
```

- **SubComponents:** setSubComponent(“slotname”, “type”)

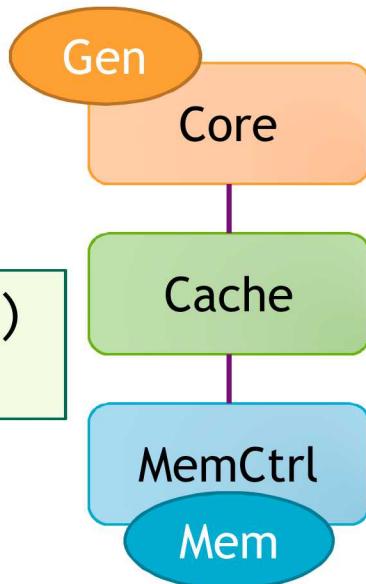
- Recall: SubComponent is a *swappable piece of functionality*

```
gen = core.setSubComponent(“generator”, “miranda.STREAMBenchGenerator”)
memory = mctrl.setSubComponent(“backend”, “memHierarchy.simpleMem”)
```

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



SST Python API
User-defined string
SST argument



SSTInfo: Getting component info

- `sst-info`: utility to query element libraries

```
$ sst-info miranda.BaseCPU
Optionally filter for a specific library and/or component
PROCESSED 1 .so (SST ELEMENT) FILES FOUND IN DIRECTORY(s) /home/sst/build/sst-elements/lib/...
Filtering output on Element = "miranda.BaseCPU"
=====
ELEMENT 0 = miranda ()
Num Components = 1
  Component 0: BaseCPU
  CATEGORY: PROCESSOR COMPONENT
    NUM STATISTICS = 17
    ...
    NUM PORTS = 2
      PORT 0 = cache_link (Link to Memory Controller)
      ...
      NUM SUBCOMPONENT SLOTS = 2
        SUB COMPONENT SLOT 0 = generator (What address generator to load) [SST::Miranda::RequestGenerator]
        ...
        NUM PARAMETERS = 12
          PARAMETER 0 = clock (Clock for the base CPU) [2GHz]
          ...
          Parameter
          Definition
          Definition
          Slot name
          Definition
          SubComponent API
          "REQUIRED" or
          default value
```

Configuration File: Connecting the components

- Declared links and components a couple slides ago...

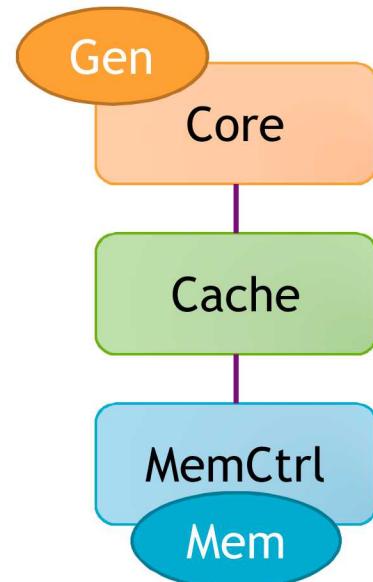
```
link0 = sst.Link("core_to_cache")
link1 = sst.Link("cache_to_memory")
```

```
core = ...
cache = ...
mctrl = ...
```

- Connect components:** connect(`endpoint1`, `endpoint2`)
- Where endpoint is: (component, port, latency)

```
link0.connect(
    (core, "cache_link", "100ps"),
    (cache, "high_network_0", "100ps")
)
link1.connect(
    (cache, "low_network_0", "100ps"),
    (mctrl, "direct_link", "100ps")
)
```

The diagram illustrates the connections defined in the configuration file. It shows two endpoints, Endpoint 1 and Endpoint 2, each connected to multiple components via different links. Endpoint 1 is connected to the Core component via the 'cache_link' port and to the Cache component via the 'high_network_0' port. Endpoint 2 is connected to the Cache component via the 'low_network_0' port and to the MemCtrl component via the 'direct_link' port.



Running SST

- **Usage:** `sst [options] configFile.py`
- **Common options:**

<code>-v --verbose</code>	Print information about core runtime
<code>--debug-file <filename></code>	Send debugging output to specified file (default: <code>sst_output</code>)
<code>--partitioner <zoltan self simple rrobin linear lib.partitionner.name></code>	Specify the partitioning mechanism for parallel runs
<code>-n --num_threads <num></code>	Specify number of threads per rank
<code>--model-options "<args>"</code>	Command line arguments to send to the Python configuration file
<code>--output-partition <filename></code>	Write partitioning information to <filename>
<code>--output-dot <filename></code> <code>--output-xml <filename></code> <code>--output-json <filename></code>	Output the configuration graph in various formats to <filename>

Running a simulation

- Launch simulation

```
$ sst demo_1.py
```

- Output

```
Simulation is complete, simulated time: 6.66491 us
```

- We probably want more information about what happened though
 - Enable statistics!

Enabling statistics

- Most Components and SubComponents define statistics

```
$ sst-info memHierarchy.Cache  
...  
NUM STATISTICS = 51  
    STATISTIC 0 = CacheHits [Total number of cache hits] (count) Enable Level = 1  
    STATISTIC 1 = latency_GetS_hit [Latency for read hits] (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.statoutputX where X=
 - console
 - txt
 - json
 - 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?

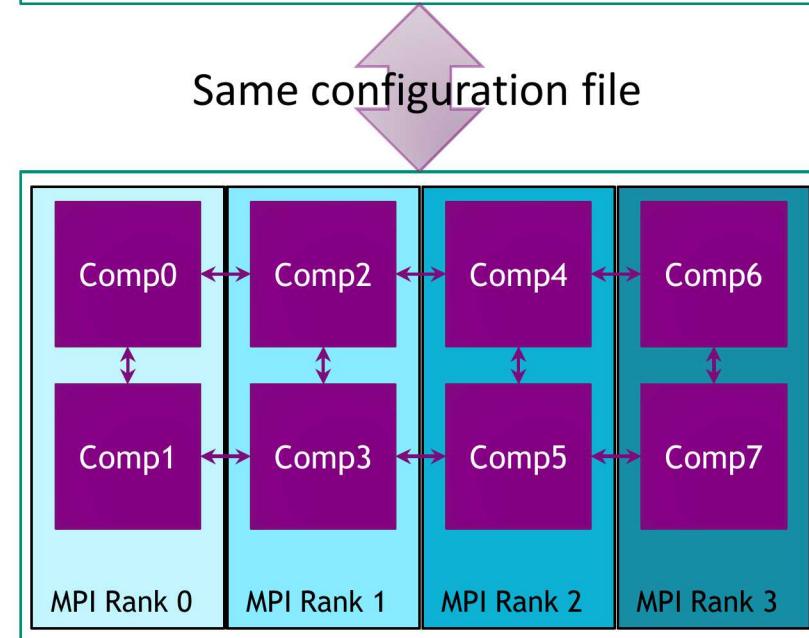
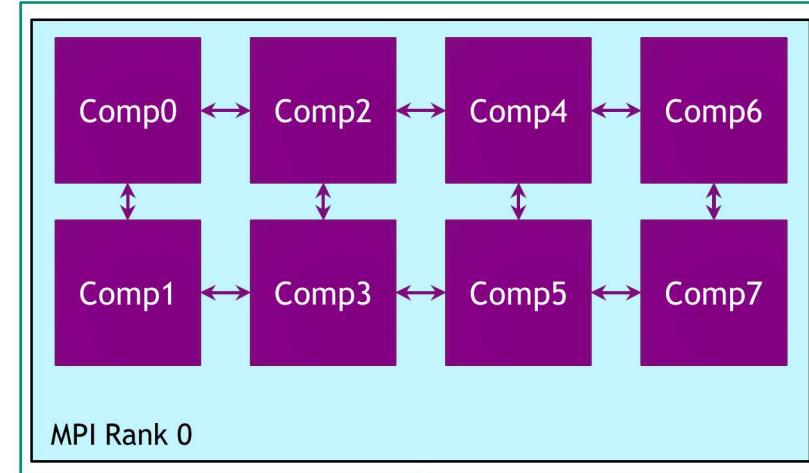
SST in parallel

- SST was designed from the ground up to enable scalable, parallel simulations
- Components distributed among MPI ranks/threads
 - Link latency controls synchronization rate

```
# Two ranks
$ mpirun -np 2 sst demo1.py

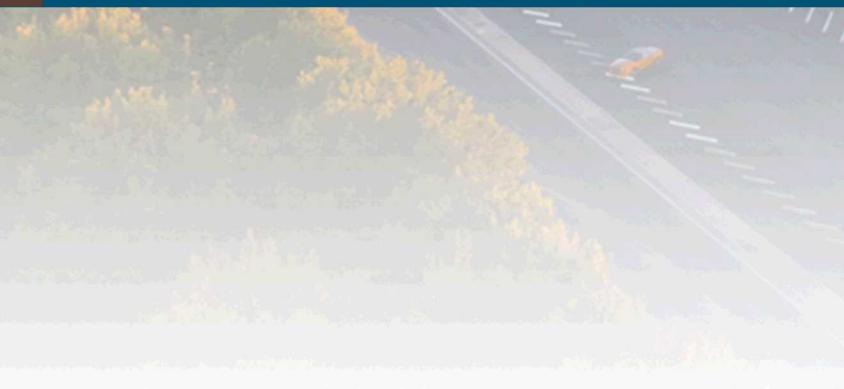
# Two threads
$ sst -n 2 demo1.py

# 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 demo1.py
```

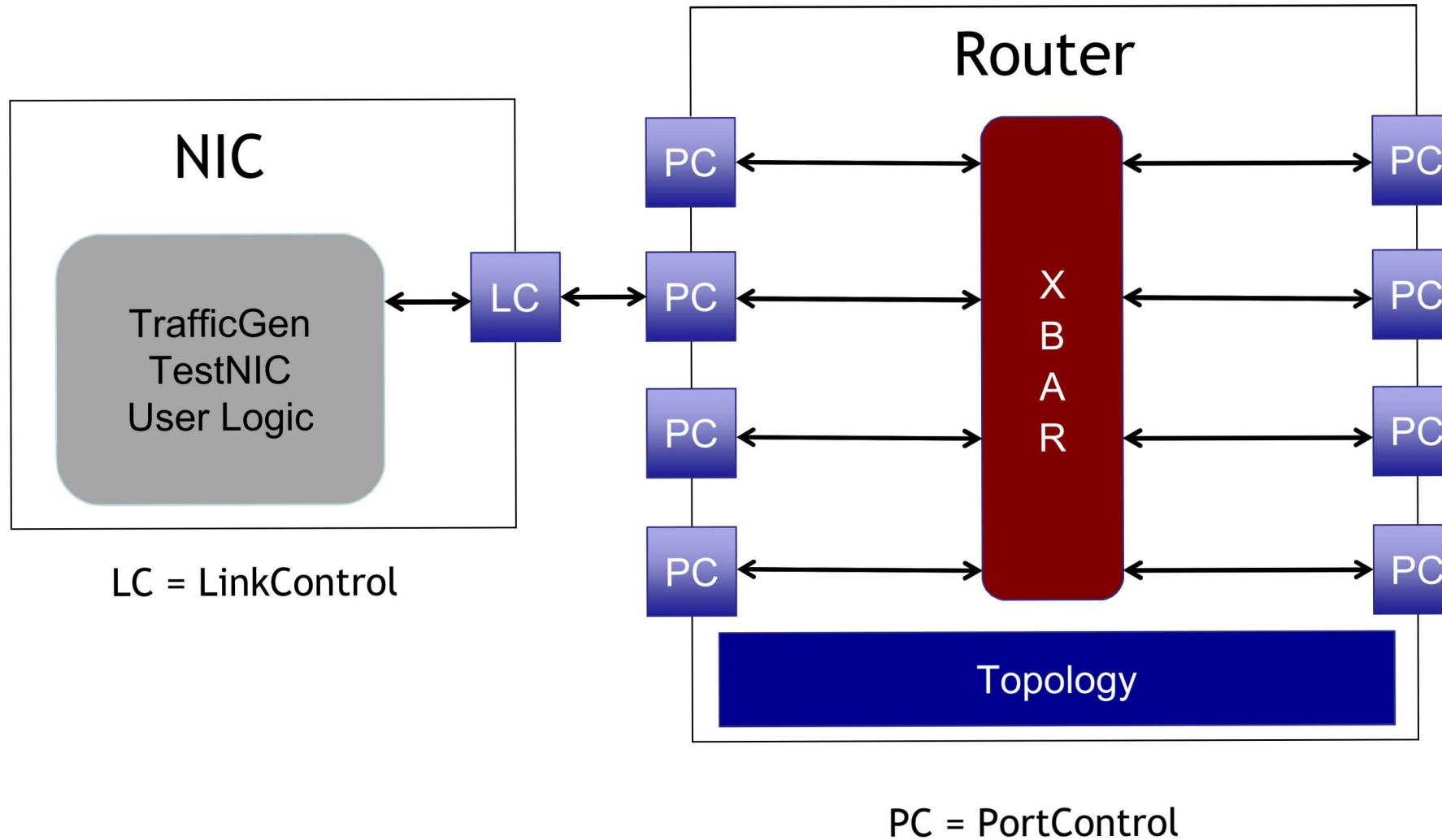




Backup



Merlin: Organization

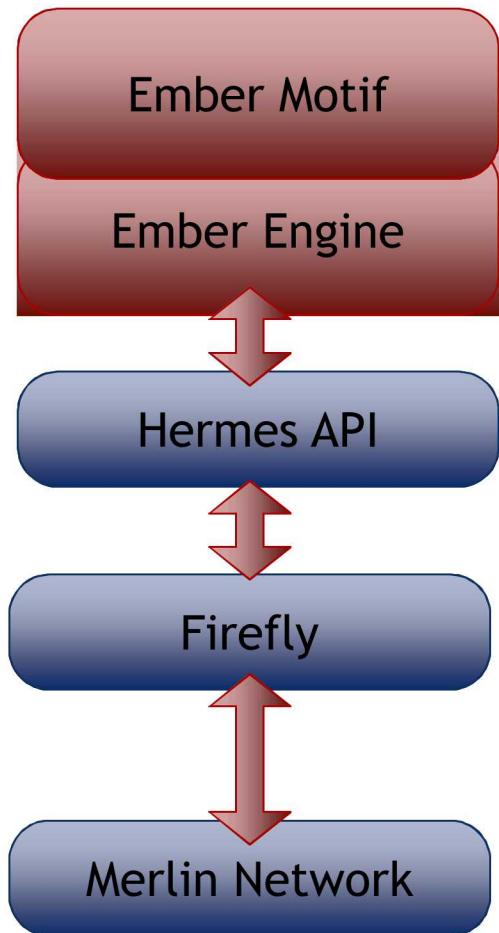


Ember: Network Traffic Generator

- 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



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

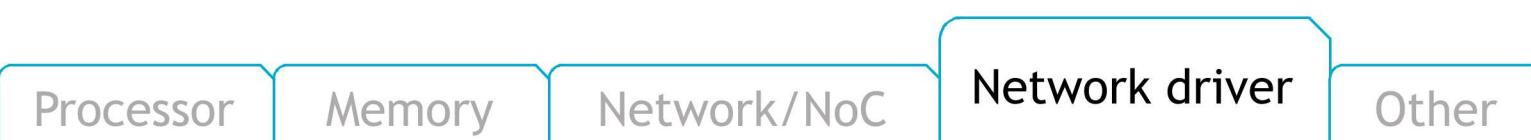
- Motifs are lightweight patterns of communication
 - Tend to have very small state
 - Extracted from parent applications
 - Models as an MPI program (serial flow of control)
 - Many motifs acting in the simulation create the parallel behavior
- Example motifs
 - Halo exchanges (1, 2, and 3D)
 - MPI collections – reductions, all-reduce, gather, barrier
 - Communication sweeping (Sweep3D, LU, etc.)

Ember: Motifs (continued)

- 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

- **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

- 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

