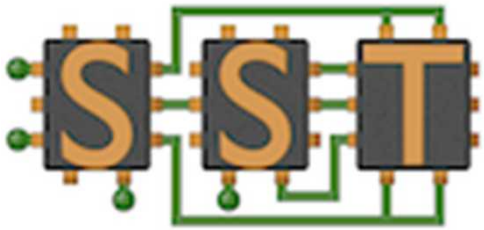
SAND2015-4701C

Sandia National Laboratories

Photos placed in horizontal position with even amount of white space between photos and header

Structural Simulation Toolkit (SST)



June 13, 2015

ISCA Tutorial

U.S. DEPARTMENT OF ENERGY

National Nuclear Security Administration

Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000. SAND NO. 2011-XXXXP

Welcome to the SST Tutorial

- We will be doing some demos using a provided VM

- Setup:
  - Copy the VirtualBox VM from the provided USB drives
  - Boot the VM
    - Pre-compiled SST Release 5.0.1
      - Path
    - Configuration files for demo
      - Path

Welcome!

Morning (9a-12:30p)

- Welcome and setup
- The SST Framework
- Demo: Running a simulation
- Tour of SST Elements

Afternoon (1:30p-5p)

- Tour of SST Elements (continued)
- Demo: Using the statistics API
- Use cases
- A user perspective
- Validation
- Future developments

Instructors

- Gwen Voskuilen
- Arun Rodrigues
- Si Hammond
- Branden Moore

Why SST?

- Problem: Simulation is slow
  - Tradeoff between accuracy and time to simulate
  - Many simulators are serial, unable to simulate very large systems
- Problem: Lack of simulator flexibility
  - Tightly-coupled simulations: faster but difficult to modify
  - Difficult to simulate at different levels of accuracy

The Structural Simulation Toolkit:

A parallel, discrete-event simulation framework

for scalability and flexibility

Overview

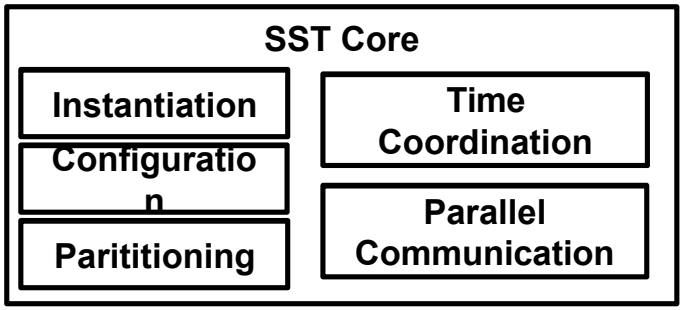
- **Parallel**
  - Built from the ground up to be scalable
  - Demonstrated scaling to 512+ processors
- **Flexible**
  - Enables “mix and match” of simulation components
  - Custom architectures
  - Custom tradeoff between accuracy and simulation time
    - E.g., cycle-accurate network with trace-driven endpoints

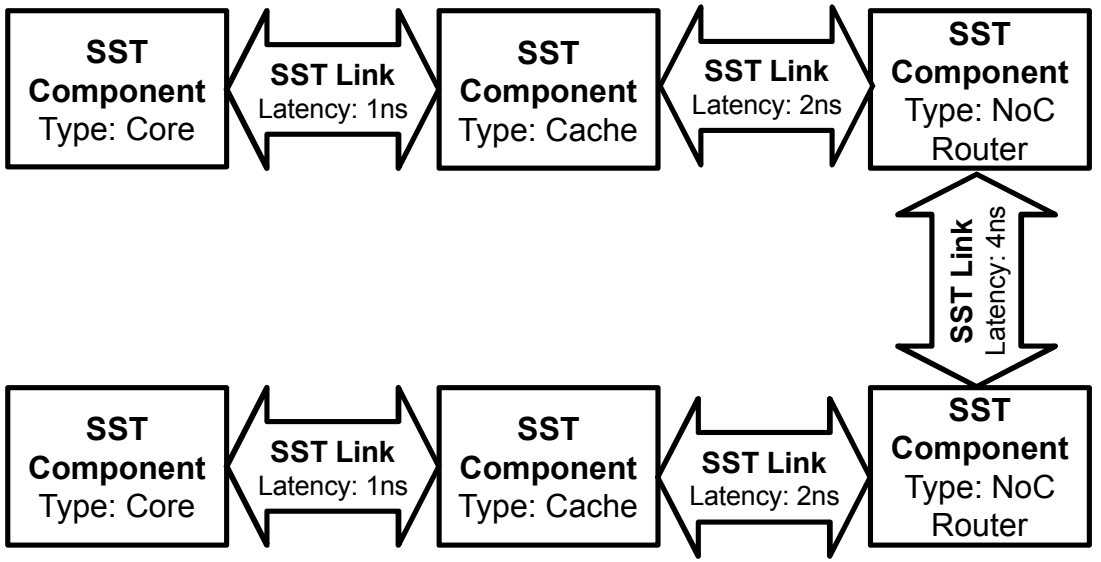
Capabilities

- SST Core (framework):
  - Time-scale independent: micro-, meso-, macro-scale simulations
  - Provides a number of interfaces and utilities for simulation models
- Components: SST's simulation models
  - Components perform the actual simulation
  - Many built-in models available: processors, memory, network
  - Compatible with external models: Gem5, DRAMSim2, many others
- Open API
  - Easily extensible with new models
  - Modular framework
  - Open-source core

SST: FRAMEWORK FOR PARALLEL SIMULATION

SST's discrete-event algorithm

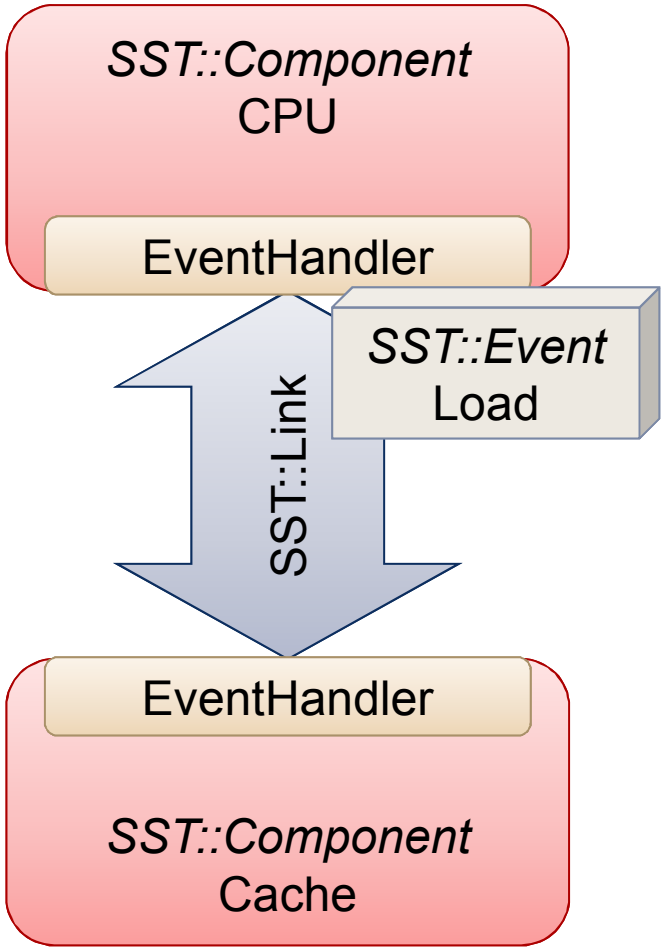




- Simulations are comprised of **components** connected by **links**
- **Components** interact by sending events over **links**
- Each **link** has a *minimum* latency
- **Components** can load **subComponents** and **modules** for additional functionality

Key objects

- SST::Component
  - Simulation model
- SST::Link
  - Communication path between two components
  - Has optional EventHandler
- SST::Event
  - A discrete event
- SST::Clock::Handler
  - Function to handle a clock tick
- SST::SubComponent
  - Add functionality to Components
- SST::Module
  - Add functionality to framework

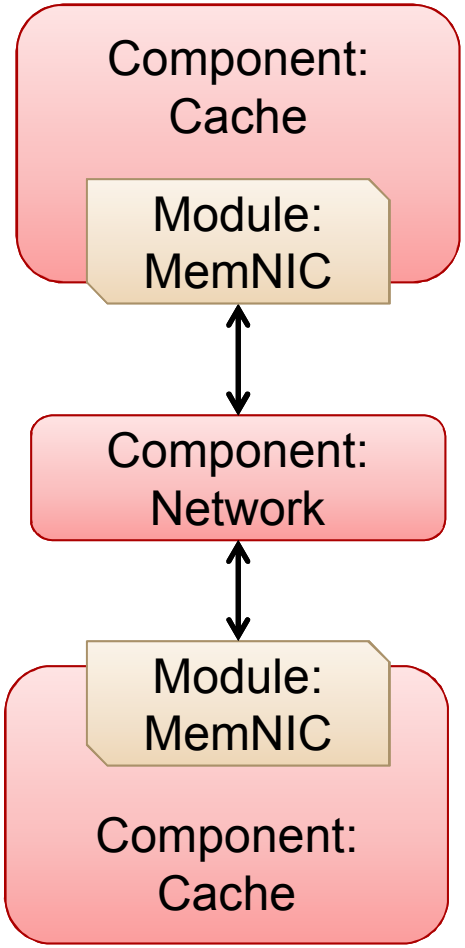


Component

- Basic building block of a simulation model
  - E.g., processor, cache, network router, etc.
- Performs the actual simulation
- Uses Links and Ports to communication with other components
  - Components define ports, links connect ports between components
  - Polled: Register a clock handler to poll the link
  - Interrupt: Register an event handler to be called when an event arrives
  - Both: Receive events on interrupt, send events on clock
- Uses SubComponents and Modules for additional functionality

SubComponents and Modules

- Add additional functionality to a Component
  - SubComponent (SC): *friend* class of component
  - Module (M): not a *friend* of component
- Provide modularity
  - Generic interface which can be used by multiple SC/M
  - Loaded by Components at runtime
- Tightly-coupled with a component
  - Components call SC/M as a C++ class instance
  - Do not need to communicate via Links
  - SC/M cannot exist by themselves
- Example
  - MemNIC: Implements the network interface (simpleNetwork), used by caches to communicate over a network



Link

- Connects two components
  - Connect a specific “Port” on component A to a “Port” on component B
- The ONLY mechanism by which components communicate
  - Necessary for parallel simulation
- Has a minimum, non-zero latency for communication
  - Except self-links
  - Except during initialization
- Transparently handles any MPI communication

Event

- Unit of communication between two components
  - Packet format is up to the communicating components
- Some standardized interfaces
  - Facilitate “mix and match” capability
  - sst/core/interfaces/
  - Memory (simpleMem)
    - Defines commands & event format for communication with memory
  - Network (simpleNetwork)
    - Defines a header for events sent through a network component

Component/link interface

- Components use these calls to manage links and events

- SST::Component::configureLink()
  - Registers a link and (optionally) a handler
- SST::Link::recv()
  - Pull an event from a link
- SST::Link::send()
  - Push an event down a link
- SST::Component::registerClock()
  - Register a clock frequency and a handler to be called on each clock tick

Simulation lifecycle

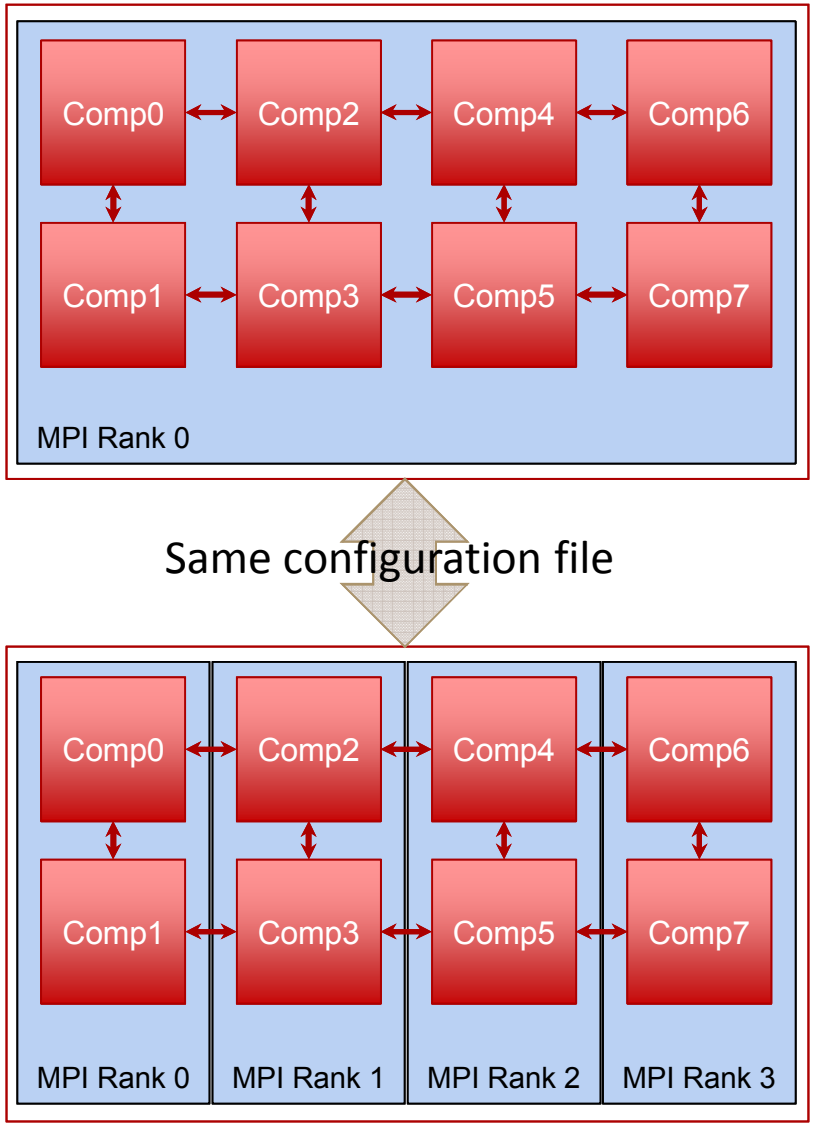
- Birth
  - **Create graph** of components using Python configuration file
  - **Partition** graph and assign components to MPI ranks
  - **Instantiate** components
  - **Connect** components via links
  - **Initialize** components using their init() functions
  - **Setup** components using their setup() functions
- Life
  - **Send events**
  - **Manage clock and event handlers**
- Death
  - **Finalize** components using their finish() functions
  - **Output** statistics
  - **Cleanup** simulation, delete components

Component lifecycle

- *Pre-construction: define and partition components*
- *Construction: call component constructors and parse parameters*
- Initialization – init()
  - Components send "init" events to each other over links
    - Discover neighbors, negotiate parameters, initialize data structures, etc.
  - Multiple rounds of communication until no more "init" events are sent
- Setup – setup()
  - Each component does its final setup and schedules initial events
- Run
  - Actual simulation
  - Continues until a set time, or all components agree to finish
- Finalize – finish()
  - Simulation complete
  - Write statistics, free memory, etc.

SST in parallel

- SST was designed from the ground up to enable scalable, parallel simulations
- Components are distributed among MPI ranks
- Links allow parallelism
  - Hence, components should communicate via links only
  - Transparently handle any MPI communication
  - Specified link-latency determines MPI synchronization rate



Agenda

Morning

- Welcome and setup
- The SST Framework
- **Demo: Running a simulation**
- Tour of SST Element Libraries

Afternoon

- Tour of SST Elements (continued)
- Demo: Using the statistics API
- Use cases
- Validation
- Future developments
- Wrap-up

Getting and installing SST

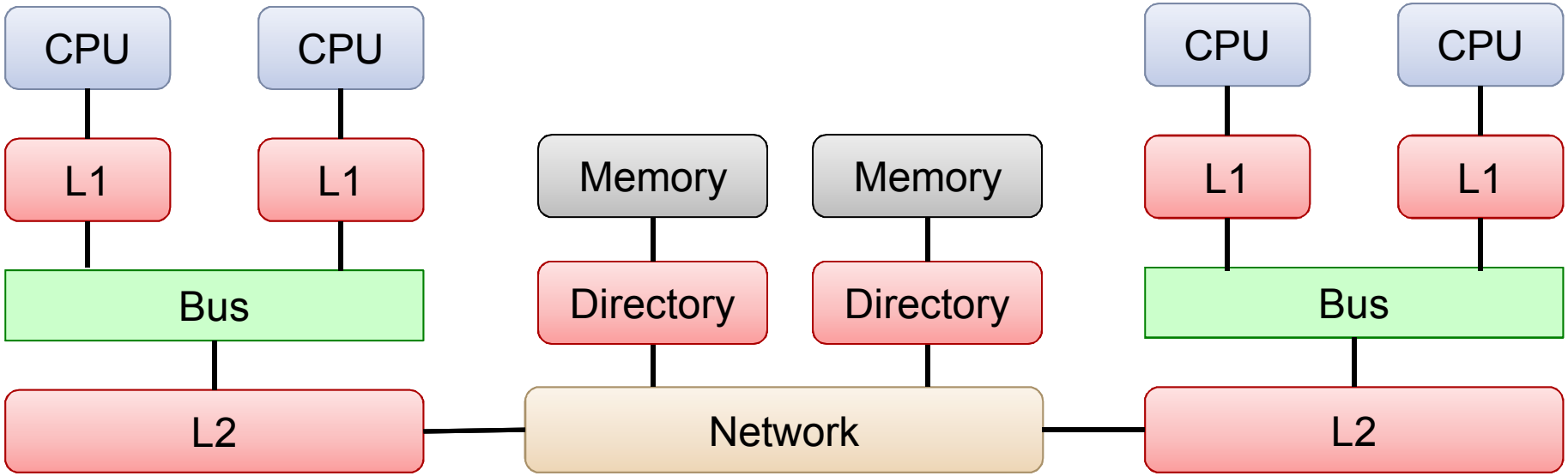
- Already installed in the VM, but for future reference…
- www.sst-simulator.org
  - Current release (5.0.1) source download
  - Directions for SVN checkout
    - Caveat: SST is under active development, code in the trunk may not have passed testing yet
  - Detailed build instructions including dependencies for Linux & Mac
  - Links to mailing lists for updates and support

Navigating the source code

- *~/examples*
  - Example configuration files
- *~/sst/core*
  - Source for the core framework, APIs, etc.
- *~/sst/elements*
  - Source for the Element Libraries
  - Element: Collection of components (e.g., *Merlin* – networks)
- *~/tools*
  - Standalone and component-specific tools
    - SSTWorkbench: GUI for generating configuration files

Configuring a simulation

- SST uses a Python configuration file
  - Defines global parameters for the simulation
  - Defines and configures components
  - Specifies links and link latencies between components
- Open ‘demo.py’



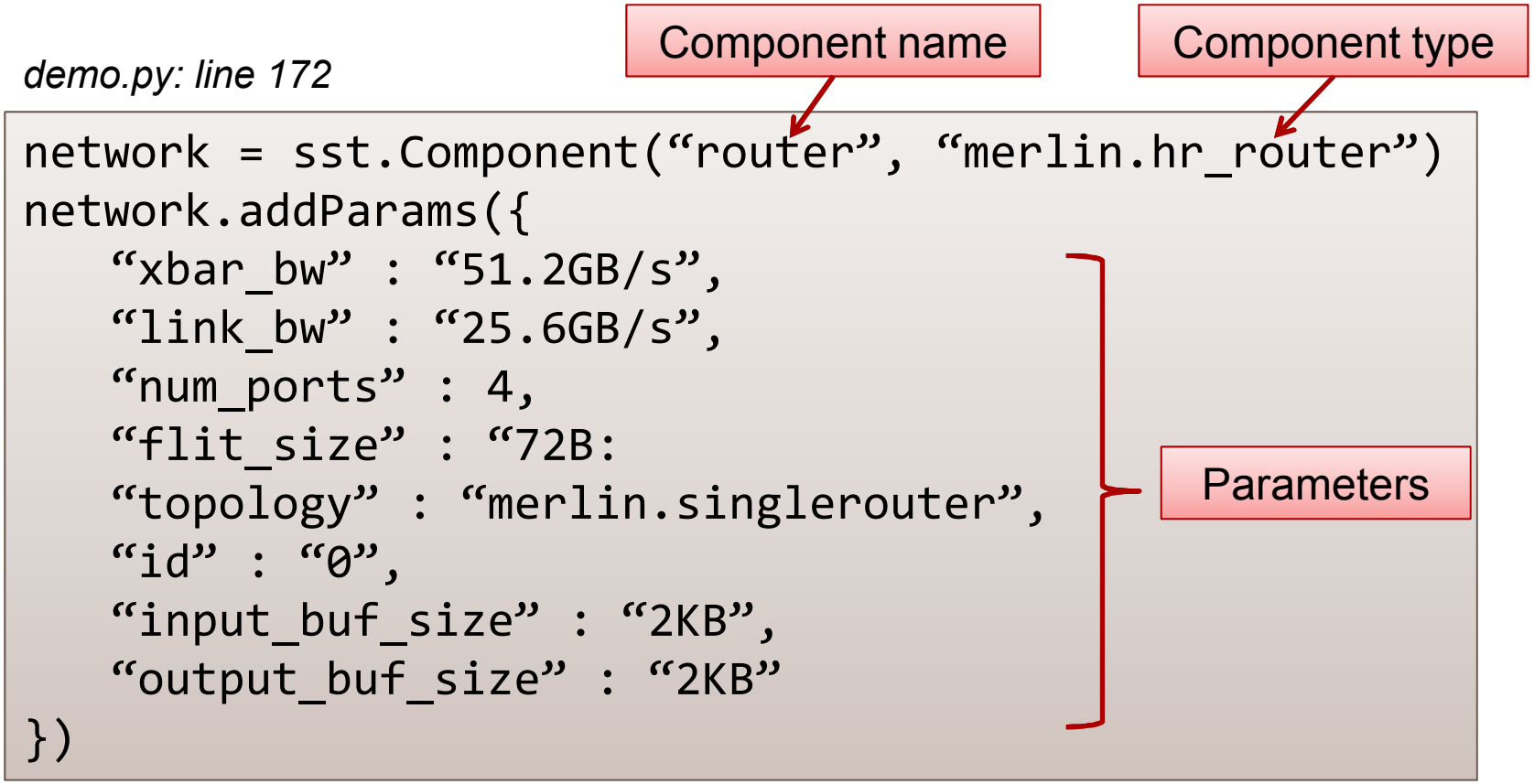
Part 1: Configure SST

- Global simulation parameters
- `sst.setProgramOption("stopAtCycle", "100ms")`
  - Kill simulation (nicely!) if it runs to 100ms
- `sst.setProgramOption("timebase", "1ns")`
  - Tell SST that we're simulating at a granularity around 1ns
  - Used by SST core when time units are not specified by a component
  - Not a lower limit! (clocks can be > 1 GHz)

Part 2: Define components

- **Define:** `sst.Component("name", "type")`
- **Configure:** `addParams ({ "parameter" : value, … })`

demo.py: line 172



SSTInfo: Getting component info

- Prints parameters, port names, and statistics

```
$ sstinfo memHierarchy.Cache
PROCESSED 25 .so (SST ELEMENT) FILES FOUND IN DIRECTORY /home/sst/build/lib/sst
Filtering output on Element.Component = "memHierarchy.Cache"
================================================================================
ELEMENT 18 = memHierarchy (Cache Hierarchy)
COMPONENT 0 = Cache [MEMORY COMPONENT] (Cache Component)
NUM PARAMETERS = 32
    PARAMETER 0 = cache_frequency (Clock frequency with units. For L1s, this is
                  usually the same as the CPU's frequency.) [REQUIRED]
    …
    PARAMETER 21 = network_bw (Network link bandwidth.) [1GB/s]
    …
NUM PORTS = 4
    …
    PORT 3 [1 Valid Events] = directory (Network link port to directory)
        VALID EVENT 0 = MemHierarchy.MemRtrEvent
    …
NUM STATISTICS = 32
```

Optionally filter for a specific component

"REQUIRED" or default value

Parameter

Definition

Port name

Definition

Type of event(s) used on the link

Part 3: Defining links

- Example: Connect socket 0's L2 cache (l2cache0) to network
  - **Create a link:** sst.Link("name")
  - **Define link endpoints:** connect(endpoint1, endpoint2)
    - Endpoint is defined as: (Component, Port, Latency)
    - Note: Latencies of the two endpoints can differ

demo.py: line 220

```
…
l2cache0_network_link = sst.Link("l2cache0_network_link")
…
l2cache0_network_link.connect(
    (l2cache0, "directory", "50ps"),
    (network, "port0", "50ps") )
…
```

Link name

Endpoints

Running SST

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

`-v | --verbose`	Print verbose information during runtime
`--debug-file <filename>`	Send debugging output to specified file (default: sst_output)
`--add-lib-path <dirname>`	Add <dirname> to search path for element libraries
`--heartbeat-period <period>`	Every <period> time, print a heartbeat message
`--paritioner <zoltan | self | simple | rrobin | linear | lib.partitioner.name>`	Specify the partitioning mechanism for parallel runs
`--model-options "<args>"`	Command line arguments to send to the Python configuration file
`--output-partition <filename>`	Write partitioning information to <filename>
`--output-dot <filename>`	Output a graph representing the configuration in "Dot" format to <filename>

Demo: Running the simulation

- Launch simulation

```
$ sst demo.py
```

- Output

```
Inserting stop event at cycle 100ms, 100000000000
ARIEL-SST PIN tool activating with 4 threads
ARIEL: Default memory pool set to 0
ARIEL: Tool is configured to begin with profiling immediately.
ARIEL: Starting program.
Performing iteration 0
Performing iteration 0
Performing iteration 0
Performing iteration 0
…
…
Simulation is complete, simulated time: 125.209 us
```

Agenda

Morning

- Welcome and setup
- The SST Framework
- Demo: Running a simulation
- **Tour of SST Element Libraries**

Afternoon

- Tour of SST Elements (continued)
- Demo: Using the statistics API
- Use cases
- Validation
- Future developments
- Wrap-up

Element libraries

- Libraries which contain a set of related components, subComponents, and modules
- SST comes with many built-in libraries
  - Processors, memory, network, etc.
  - Tested for inter-library compatibility
- Also compatible with many external “libraries”
  - DRAMSim2, Gem5, many others
  - See **www.sst-simulator.org** for more information

SST5.0 element libraries

- Processors
  - Ariel – PIN-based
  - Prospero – Trace-based
  - Miranda – Pattern-based
- Memory
  - MemHierarchy – Caches, memory
  - VaultSimC - Stacked memory
  - Cassini – Cache prefetchers
- Network driver
  - Ember – Pattern-based
  - Firefly – communication protocols
  - Hermes - MPI-like driver interface
  - Zodiac – trace-based
- Network models
  - Merlin – Network simulator
- Other
  - Scheduler
  - simpleElementExample

SST5.0 external components

- Processors
  - Gem5* - Cycle-accurate processor model
  - Qsim - Processor model
  - MacSim - GPU model
- Memory
  - DRAMSim2 - DRAM
  - NVDIMMSim - Non-volatile (NV) memory
  - HybridSim - NV + DRAM

**Gem5 support through v4 for an older branch of Gem5; starting with v5.0, support for the Gem5 stable release version is being provided within Gem5*

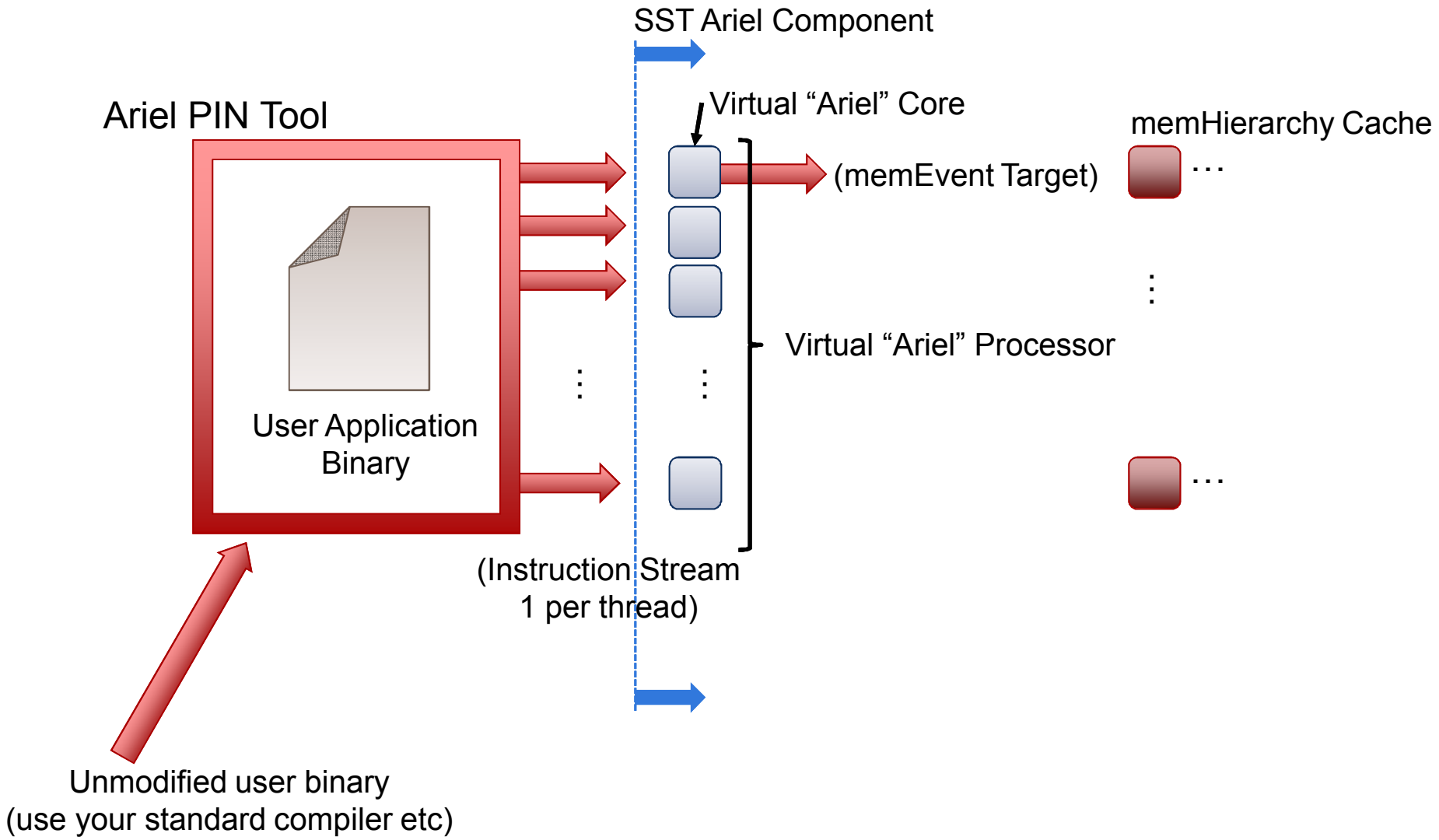
Ariel: PIN-based processor

- Lightweight processor core model
- Uses Intel's PIN tools and XED decoders to analyze binaries
  - Runs x86, x86-64, SSE/AVX, etc. compiled binaries
  - Supports fixed thread count parallelism (OpenMP, Qthreads, etc.)
- Passes information to virtual core in SST
- Implements SST's memory interface to interact with a memory model

Ariel: The tradeoff

- Pros:
  - Faster than cycle-accurate processor models (e.g., *Gem5*)
  - Reasonable approximation for studies on memory system performance
    - Especially for heavily memory-bound applications
  - Reasonable model of thread interactions
- Cons
  - Slower than trace/pattern-based processor models
  - Does not give cycle-reproducible results
    - Use of threads can disturb reproducibility
  - Non-deterministic results
  - Not compatible with non-x86 binaries

Ariel: Architecture



Ariel: Details

- Ariel's virtual cores
  - Instruction information currently limited to memory ops or instructions with no memory operands
  - Clocked: Reads instruction stream in chunks but processes on clock
  - Back pressure from FIFO halts real binary execution
  - Does not maintain dependence order or register locations (yet)!
  - Performs a TLB mapping of virtual-to-physical addresses
- Key user knobs
  - Memory ops issued/cycle
  - Load/store queue size
- Memory interface
  - Generates memEvents which can be sent to a cache model
  - Tracks basic statistics (request counts, split-cache line loads, etc.)

Prospero: Trace-based processor

- Trace-based processor model
  - 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-style 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

- Pros
  - Faster than Ariel and Gem5
    - Provided you can get a trace
  - Good for heavily memory-bound applications
  - Reasonable approximation to memory system performance
- 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

- Extremely light-weight processor model
  - Generates specific memory address patterns
- Current 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)

Miranda: The tradeoffs

- Pros
  - Very lightweight – no binary, no trace
  - Good for applications whose address patterns are predictable
    - E.g., not much pointer-chasing
- Cons
  - Need a generator for the memory pattern of interest
    - Requires a good understanding of the pattern

MemHierarchy: Memory system

- Cycle-accurate cache and memory simulation
  - Inter- and intra-socket coherence
  - Multiple main memory models
- Highly configurable
  - Can model any number of caches (L10s!)
  - Arbitrary topologies, multiple memories
  - Single- and multi-socket configurations
- Capable of modeling modern memory hierarchies
  - Intel core i7, Xeon Phi
  - Arm Cortex A8, A7, A15, A53, A57
  - SPARC T6

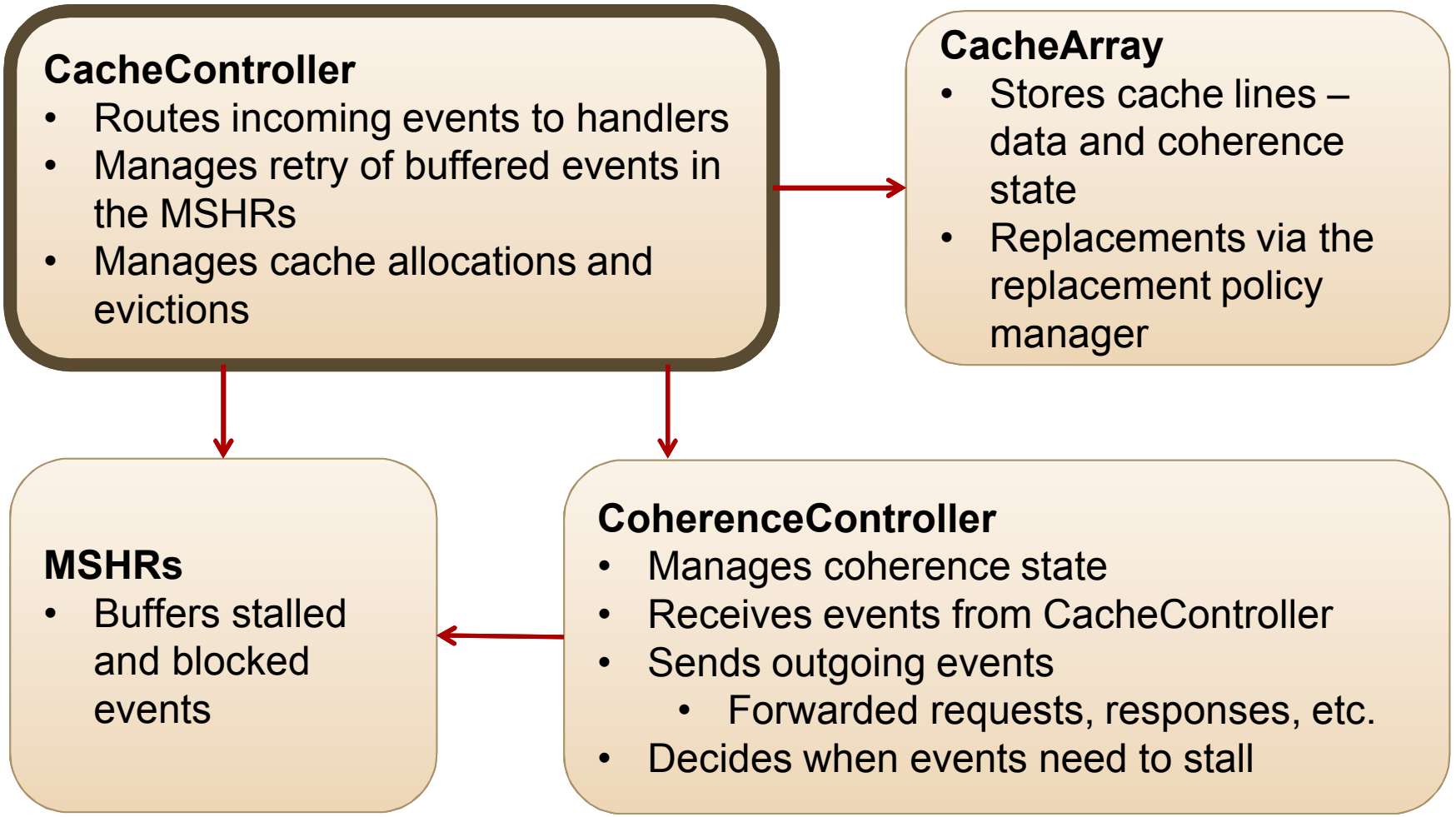
MemHierarchy: Components

- Cache
  - Includes coherence protocols (MSI, MESI, etc.)
- Bus
- Directory controller
  - Inter-socket coherence
- Memory controller
  - Backs up simulated memory, interfaces with memory backends
- Memory backends
  - Main memory simulators for DRAM, stacked DRAM, NVRAM, etc.
- TrivialCPU & StreamCPU
  - Very simple memory request generators for testing

MemHierarchy: Caches

- Store actual data
- Set associative, configurable replacement policies
  - LRU, LFU, Random, MRU, NMRU (not MRU)
- Use MSHRs to buffer outstanding requests
- Can communicate via a direct link or over a bus or network
  - Implements simpleNetwork interface via the “MemNIC” module
- Can model a single shared cache or multiple cache slices
- Handles atomics, LLSC, non-cacheable requests, etc.
- Prefetch capability by using the *Cassini* element library

MemHierarchy: Cache structure



MemHierarchy: Main memory

- MemoryController
  - Contains a 'backing store' for simulated data
  - Can communicate over a network or via a direct link with a cache or directory
  - Interfaces with multiple memory backends
- Available backends
  - SimpleMem – basic read/write with associated latencies
  - DRAMSim2 – DRAM (external)
  - NVDIMMSim – Non-volatile memory (e.g., Flash) (external)
  - HybridSim – non-volatile memory with a DRAM cache (external)
  - VaultSimC – stacked DRAM

Welcome back!

Afternoon

- **Tour of SST Elements (continued)**
- Demo: Using the statistics API
- Use cases
- A user perspective
- Validation
- Future developments

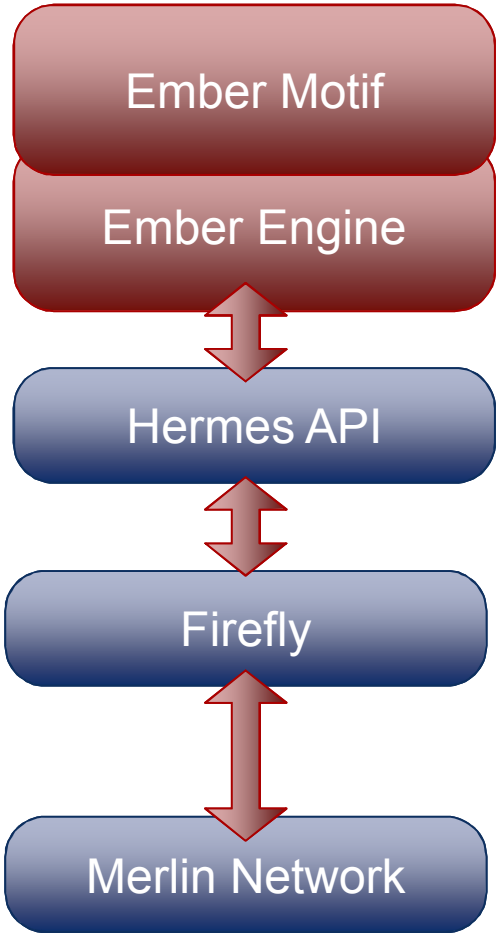
Merlin: Network simulator

- 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
- 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

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)

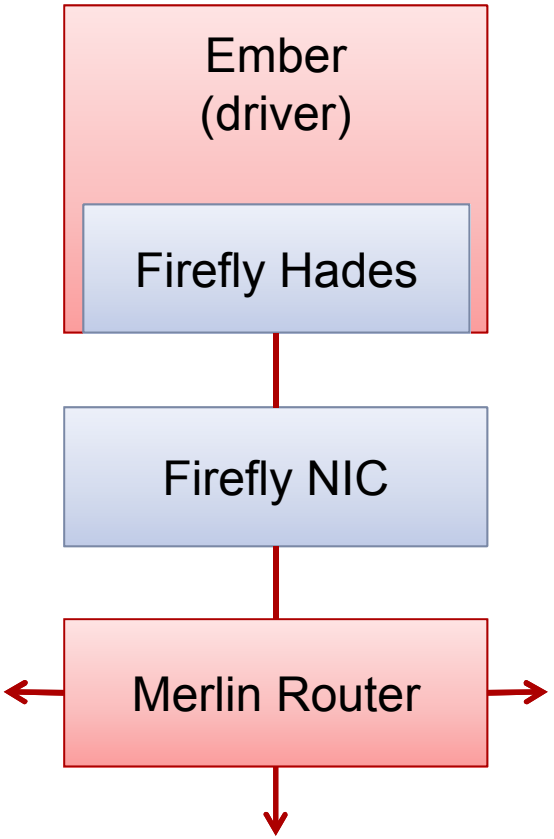
- The EmberEngine creates and manages the motif
  - Creates an event queue which the motif adds events to when probed
  - The Engine executes the queued events in order, converting them to message semantic calls as needed
  - When the queue is empty, the motif is probed again for events
- Events correspond 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 the routers to form the network
  - Fat tree, torus, etc.
- Firefly is the interface between the driver and the router
  - Message passing library → Firefly Hades
  - NIC → Firefly NIC



Scheduler

- 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 is currently a stand-alone element library
  - The schedComponent and nodeComponent must be used together
  - The faultInjectionComponent is optional

Other Libraries

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

Extending SST

- SST is designed for extensibility
  - Creating new Element Libraries and components
  - Wrapping existing simulators to interact with other SST components
  - We recommend that libraries be built outside the source tree
- Helpful information
  - Example element library
    - Components demonstrating links, ports, clocks, event handling, etc.
    - sst/elements/simpleElementExample
  - Wiki
    - *Getting Started Extending SST (a little out of date)*
    - *Building Element Libraries outside SST source tree*
  - Mailing lists – sst-developer and sst-user

Agenda

Afternoon

- Tour of SST Elements (continued)
- **Demo: Using the statistics API**
- Use cases
- A user perspective
- Validation
- Future developments
- Wrap-up

Demo: Using the Statistics API

- Goals of this demo
  - Learn to configure a simulation to give statistics via the Statistics API
  - Add a statistic to memHierarchy for some hands-on experience extending and compiling SST

Statistics API: Overview

- New in v5.0!
- Unified interface for Components to collect and return stats
- Provides users with flexibility in getting statistics

- Using the API
  - **Components** declare and register statistics
    - Name, “load level”, data type (int, double, etc.)
  - **Components** update statistics during simulation
  - **User** enables some or all statistics
  - **User** determines how often and in what format (e.g., sum or histogram) to return statistics
  - **User** determines the output format (console, text, CSV, etc.)

Enabling statistics

- First, the easy case → enable all statistics for all components
- Step 1: Set *load level*
  - Enables statistics that have a load level ≤ set level
  - Max level is 7, default is 0

```
sst.setStatisticLoadLevel(7)
```

- Step 2: Enable all statistics for all components

```
sst.enableAllStatisticsForAllComponents()
```

- Step 3: Send statistics output to the console

```
sst.setStatisticOutput("sst.statOutputConsole")
```

Try it!

- Open “demoStatistic.py”
  - Uncomment lines 237-239 near the end of the file (under “Demo #1”)
- Run the simulation and view the output

```
$ sst demoStatistic.py
…
 cpu.read_requests.0 : Accumulator : Sum.u64 = 11006; SumSQ.u64 = 11006;
Count.u64 = 11006;

 cpu.write_requests.0 : Accumulator : Sum.u64 = 5355; SumSQ.u64 = 5535;
Count.u64 = 5535;
…
```

component.statistic

collection mode

sum

sum squared

data type

Number of times data was added to the statistic

- **Tip:** Use *sstinfo* to view information about a component’s statistics

Output format options

- Print statistics to a file instead of the console
  - Regular file: CSV (statOutputCSV) or text (statOutputTxt)
  - Compressed file: CSV (statOutputCSVGz) or text (statOutputTxtGz)

```
sst.setStatisticOutput("sst.statOutputCSV")
```

- Output options for file outputs

```
sst.setStatisticOutputOption("filepath" : "myStats.csv")
```

- Specific options depend on output type
  - Use "help" option to see all options and defaults

```
sst.setStatisticOutput("sst.statOutputTxt")
sst.setStatisticOutputOption("help" : "1")
```

Enabling statistics individually

- Enable statistics for all components of a particular type

```
sst.enableAllStatisticsForComponentType("merlin.hr_router")
```

- Enable statistics for a specific component

```
sst.enableAllStatisticsForComponentName("l1cache_0")
```

- Enable a single statistic for all components of a particular type

```
sst.enableStatisticForComponentType(
    "memHierarchy.Cache", "CacheHits")
```

- Enable a single statistic for a single component

```
sst.enableStatisticForComponentName("l1cache_0", "CacheHits")
```

- *Next: customizing statistic output*

Customizing a statistic

- Option 1: Specify collection type
  - Accumulator (default): sums the data added to the statistic
  - Histogram: bins the data added to the statistic

- Option 2: Specify output frequency
  - Dump statistics at the end of simulation (default)
  - Dump statistics at a regular interval during simulation

Customizing a statistic

- Accumulator example: print the sum of cache hits every 50 us

```
sst.enableStatisticForComponentType(
“memHierarchy.Cache”, “CacheHits”,
{   “type” : “sst.AccumulatorStatistic”,
    “rate” : “50 us”
})
```

- Histogram example: print packet latency as a histogram

```
sst.enableStatisticForComponentType (
“memHierarchy.DirectoryController”, “packet_latency”,
{   “type”                  : “sst.HistogramStatistic”,
    “minvalue”              : “0”,
    “binwidth”              : “2”,
    “numbins”               : “50”,
    “dumpbinsonoutput”      : “1”,
    “includeoutofbounds”    : “1”
})
```

Exercise: Create a new statistic

- Now let's add a new statistic
  - Count the number of writes that arrive at the memory controller
- Steps
  - Define the statistic in ElementInfoStatistic
  - Create a variable in the Component for counting the statistic
  - Register the statistic
  - Call addData() for the statistic
  - Recompile
  - Update configuration file to print out our new statistic

Adding a statistic

- Define the statistic in ElementInfoStatistic
  - Per-component structure that defines the component's statistics
  - <name, definition, load level>
- Navigate to the memHierarchy element
  - sst/elements/memHierarchy/
  - Open libmemhierarchy.cc
    - Component definitions for memHierarchy
  - Find ElementInfo* for the memory controller (line XXX)
    - No ElementInfoStatistic yet → Add it!

```
static const ElementInfoStatistic memctrl_statistics[] = {
    {"WriteCount",    "Number of writes received.", "count", 1}
};
```

Load level

Statistic name

Definition

String defining the "unit" of the statistic

Adding a statistic (continued)

- Navigate to the memHierarchy element
  - sst/elements/memHierarchy
  - Open memoryController.h and memoryController.cc
- Define a variable for the statistic
  - Add the following to the "private" section of the .h file (line 162)

```
Statistic<uint64_t>* statWriteCount;
```

- Register the variable to correspond to a particular statistic
  - Add the following line to the constructor in the .cc file (after line 164)

```
statWriteCount = registerStatistic<uint64_t>("WriteCount");
```

variable

Statistic type

Statistic name

Adding a statistic (continued)

- Count the statistic
  - Open memoryController.cc
  - Find the "handleEvent" method (line 169)
    - Called each time a new event arrives at the memory controller
    - Add: Increment our new statistic if the event's command (cmd) is "GetX"

```
190: if (cmd == GetX) statWriteCount->addData(1);
```

- Now we're ready to compile
  - Navigate back up to the sst root directory
  - Run "make all install"

```
$: make all install
```

- While that's compiling…
  - Questions?

Adding a statistic (continued)

- Finally, edit demoStatistics.py to print the new statistic
  - Re-comment the lines under “Demo #1”
  - Uncomment lines under “Demo #2” (lines 244-245)
    - Note: Another syntax for enabling statistics per component!

```
memory_0.enableAllStatistics()
memory_1.enableAllStatistics()
```

- And run again:

```
$ sst demoStatistics.py
…
…
memory_0.WriteCount : Accumulator : …
memory_1.WriteCount : Accumulator : …
```

Final notes on the Statistics API

- The API is new → more components will use statistics in newer SST releases
  - Also likely to see more configuration options
- The API can be used by Components and SubComponents
  - But Modules cannot use statistics directly (due to inheritance structure)

Agenda

Afternoon

- Tour of SST Elements (continued)
- Demo: Using the statistics API
- **Use cases**
- A user perspective
- Validation
- Future developments
- Wrap-up

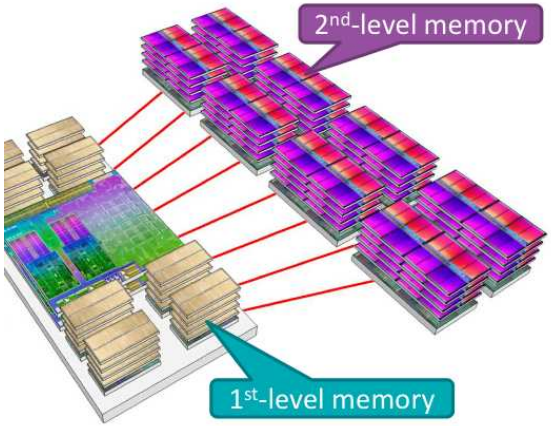
SST: Use cases

- SST is capable of simulating a wide variety of systems
  - Full-system simulation of a multicore with multiple types of memory
    - DRAM + NVRAM + stacked DRAM
  - Large networks of nodes
  - Job scheduling across thousands of nodes

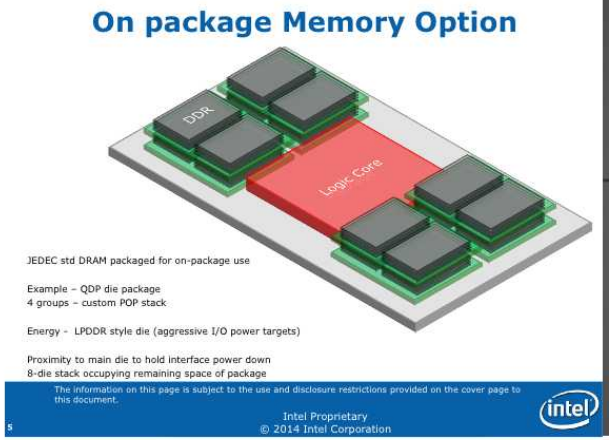
- Next: Some studies using SST

Case #1: Multi-level memory

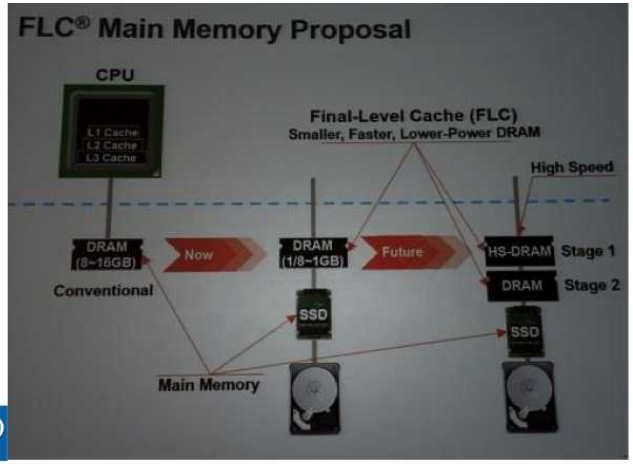
- Future memory systems will be Multi-Level Memory
- MLM can potentially offer more "usable" bandwidth, less cost
- Challenges:
  - substantial software and hardware (co-)design
  - no "one size fits all"
- SST can explore HW & SW organization



AMD



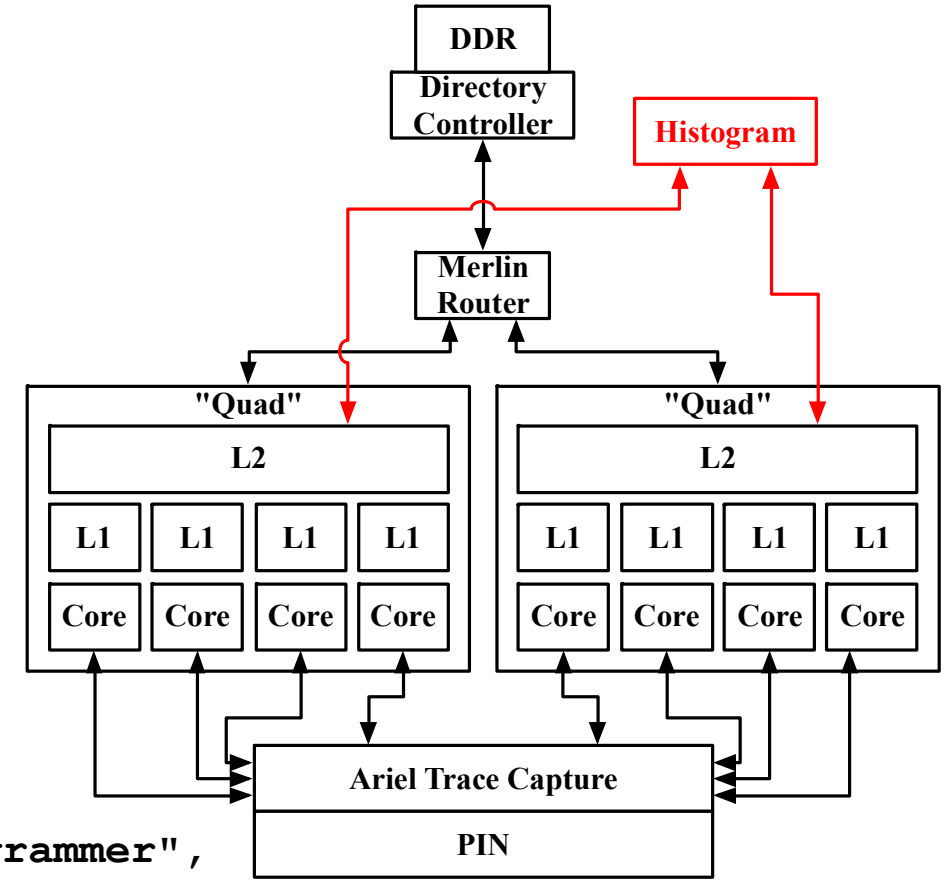
Intel



Marvell

Analyzing Memory Accesses

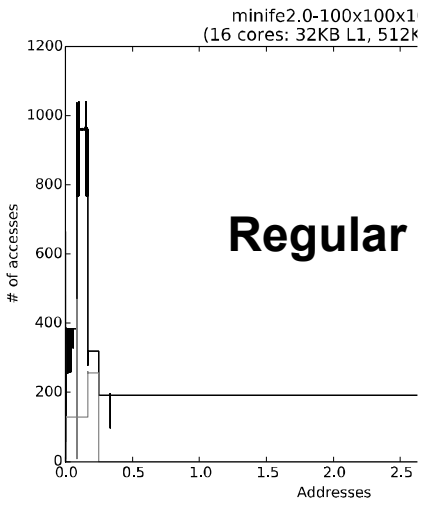
- Capture post-cache accesses
- Setup:
  - "Quads" of 4 cores
  - Histogram generator implemented as a prefetcher



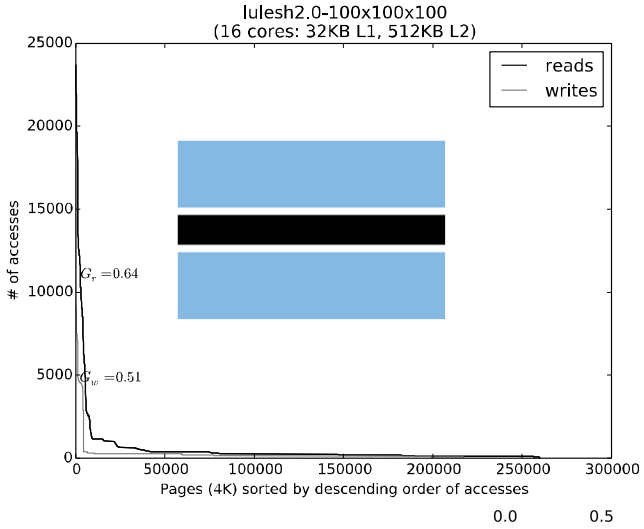

```
l2SnoopParams = {
  "prefetcher": "cassini.AddrHistogrammer",
  "prefetcher.histo_bin_width": 4096,
  "prefetcher.heap_begin": "1 GiB",
  "prefetcher.heap_end": "9 GiB"
}
```

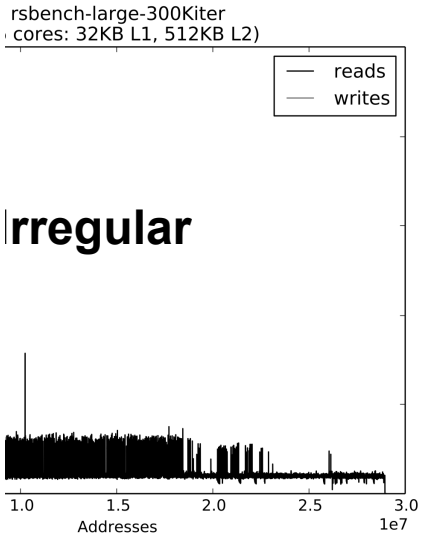
Analysis: Diverse Patterns

Physical address histograms

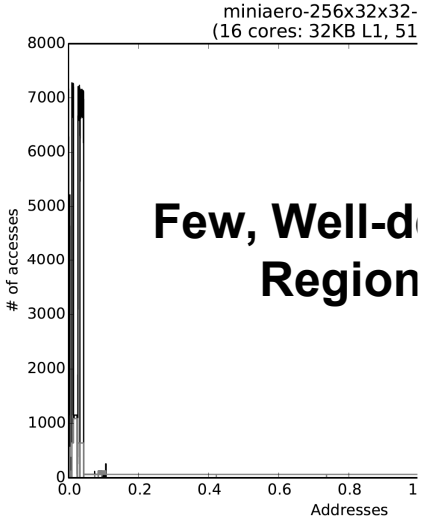


Regular

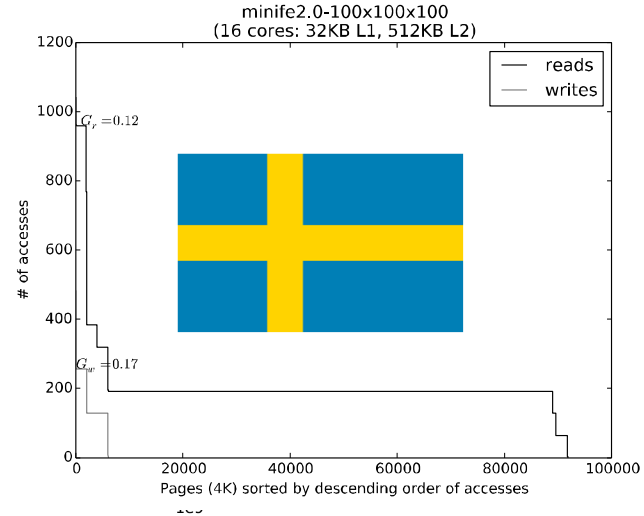


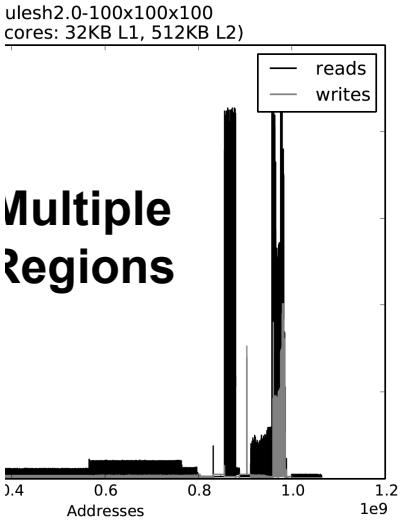


rregular



Few, Well-d
Region

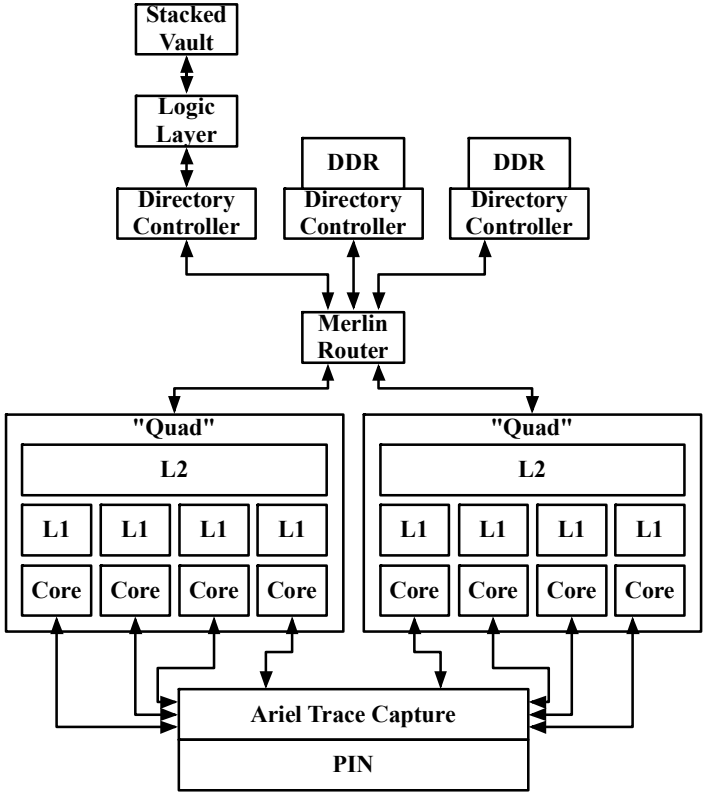




Multiple
Regions

Multi-Level Memory Simulation

- Multiple memory types:
  - DDR DRAM (DramSim)
  - HMC-like Stacked Memory (VaultSim)
  - NVRAM (NVDIMMSim)
- Addresses can be interleaved, or blocked between memory types




```
dc.addParams({
  "addr_range_start": start_pos,
  "addr_range_end":  end_pos,
  "interleave_size": interleave_size/1024,
  "interleave_step": interleave_step,
  "entry_cache_size": 128*1024,
  "clock": memclock,
  "network_address": netPort
})
```

MLM Explorations

- Analysis of application memory use distribution
- Quick exploration of “Naïve” address assignment, capacity ratios on performance
- Not shown: Feedback results from histograms to determine address assignment

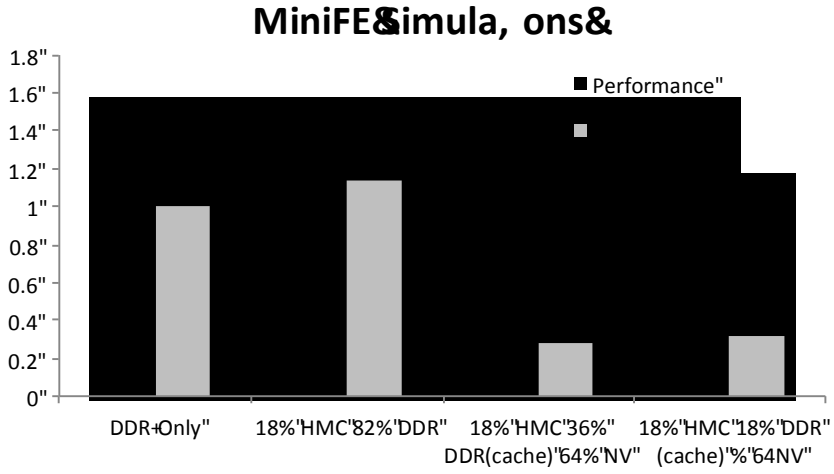


Figure 5: MiniFE Simulation results

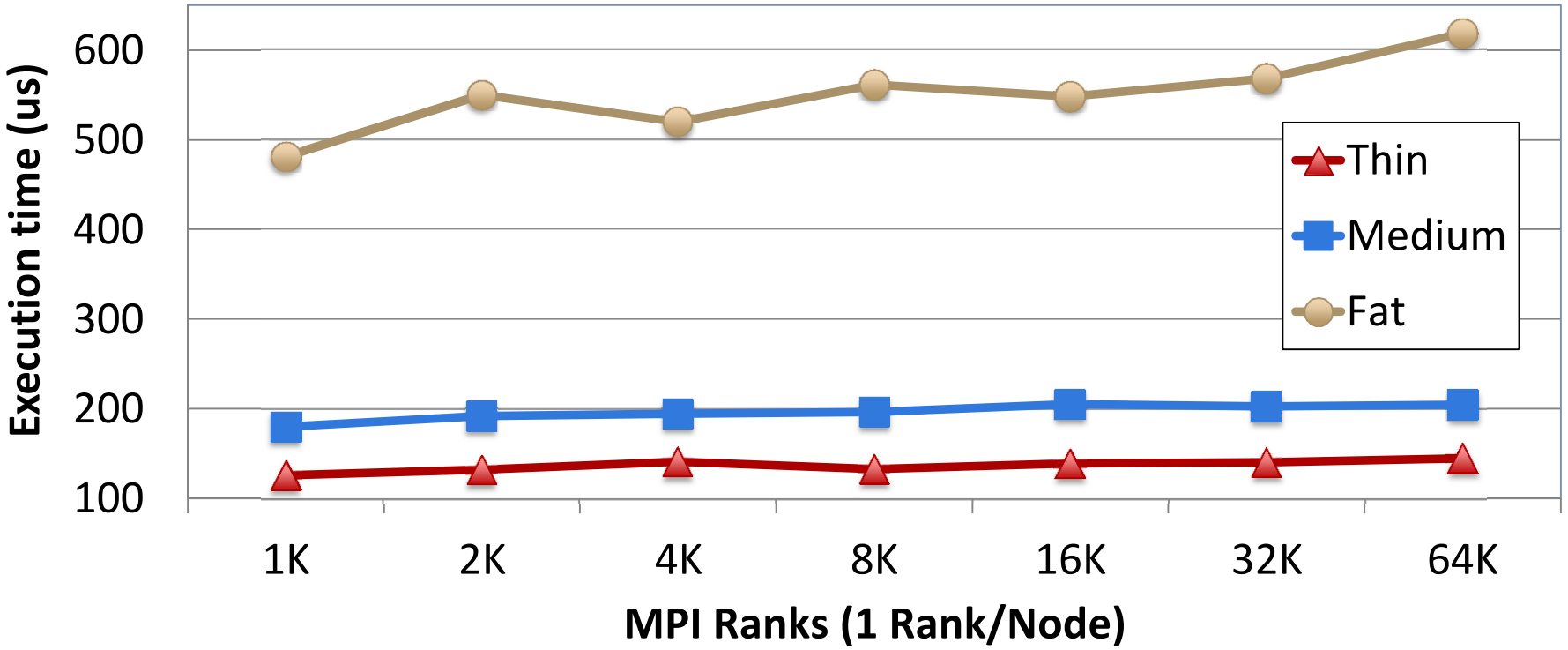
Case #2: Network

- What is the network latency achieved by different platforms during a 3D halo exchange?
  - *Halo exchange:* Exchange boundary data with neighbors
  - Platform 1: "Fat" nodes – Eight 20TF/s cores per node
  - Platform 2: "Medium" nodes – Two 20TF/s cores per node
  - Platform 3: "Thin" nodes – One 10TF/s core per node
- Evaluate for 1K to 64K participating nodes
- Evaluate at three different link bandwidths
  - 12.5GB/s, 50GB/s, 125GB/s

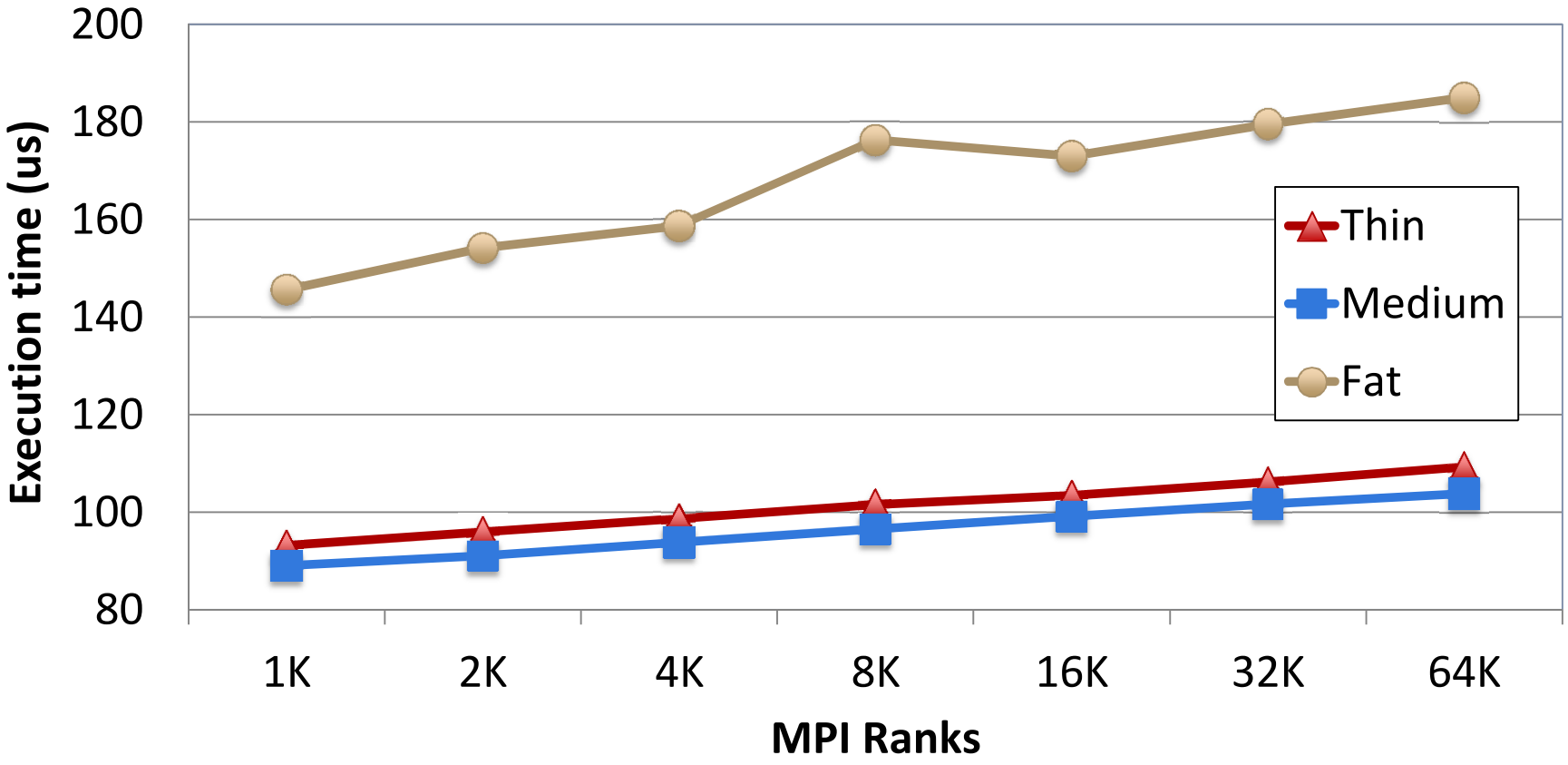
Network: Simulation setup

- Use SST *Ember* to model nodes
  - Lightweight model focused on communication pattern
    - Estimates compute time using the node's FLOPS
    - Detailed model of communication
  - Enables scaling the simulated system to a larger number of nodes
    - Compared to a detailed processor model + memory model
- Use SST *Firefly* to model the NIC
- Use SST *Merlin* to model the network
  - Detailed, cycle-accurate models for network (routers, links, etc.)

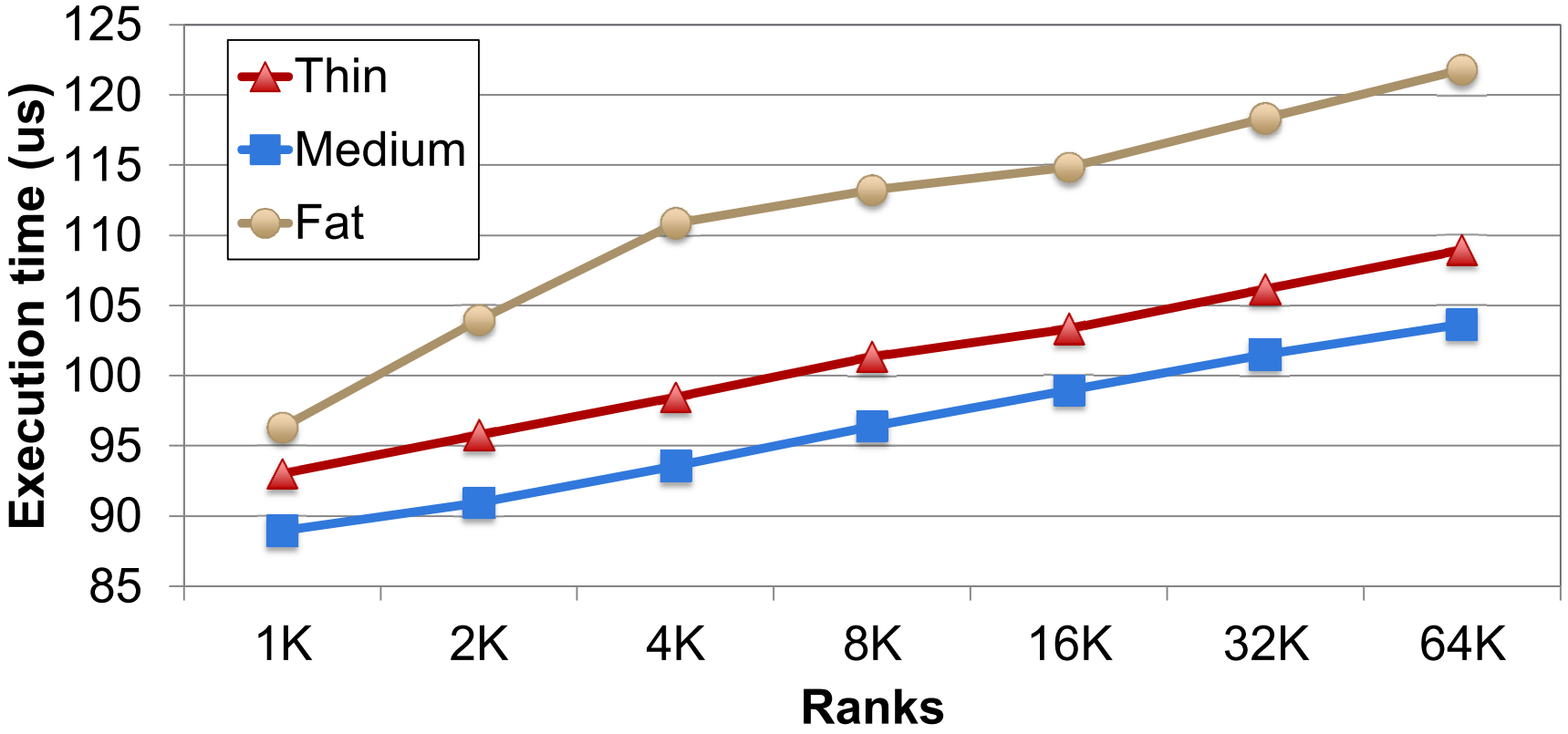
Link bandwidth = 12.5GB/s



Link bandwidth = 50GB/s



Link bandwidth = 125GB/s



Case #3: Scheduling

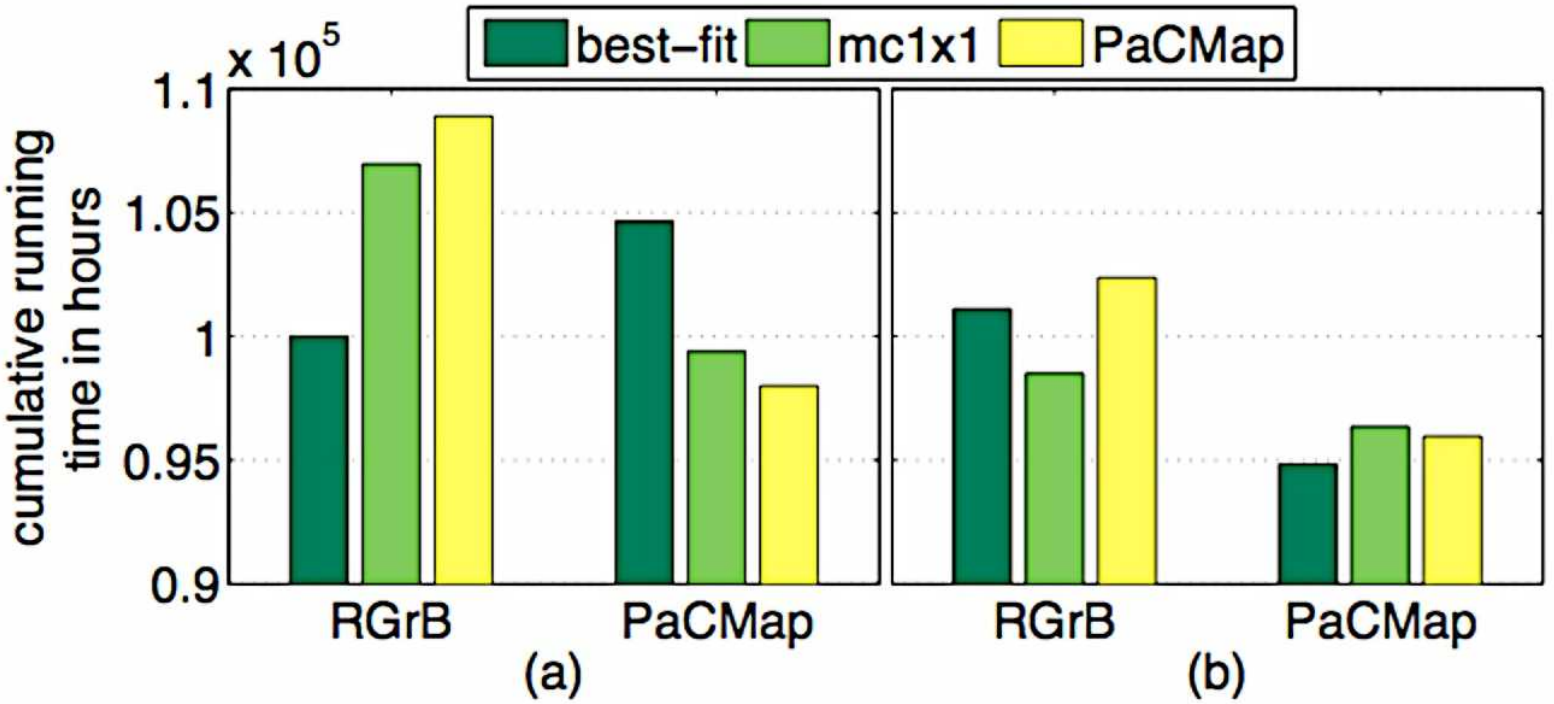
- *PaCMap: Topology mapping of unstructured communication patterns onto non-contiguous allocations (ICS 2015)*
  - *Tuncer, Leung, and Coskun*
- Problem: Want to map a job's tasks to nodes in a way that reduces communication overhead
  - Two optimizations: (1) allocate nodes to a job and (2) map a job's tasks to its allocated nodes
  - Traditionally: communication pattern-unaware allocation followed by communication pattern-aware mapping
  - But, overhead affected by both allocation and mapping
  - Challenges: Non-contiguous allocation and irregular communication
- ***PaCMap***: Joint, communication-aware, allocation and mapping

PaCMap: Simulation challenge

- Big challenge for simulation
  - System has 30K-90K+ cores across thousands of nodes
  - Workloads run 1K-3K jobs, each with up to 115K tasks
  - Interested in system performance over a period of two weeks
    - Decision for current job affects future job's performance
- Experimental methodology
  - Workloads: Trace-based
  - Create macro-level performance estimate
    - Coarse-grained performance estimate calibrated using data from real hardware
    - Tradeoff fidelity for simulation speed
  - Uses *scheduler*, METIS, LibTopoMap for partitioning and scheduling

PaCMap: Results

Cumulative running time for jobs in (a) LLNL-Atlas and (b) CEA-Curie for different allocator-mapper pairs



- PaCMap reduces cumulative running time by 2-3% (3000 node hours) over two weeks
- For jobs with 1K+ tasks, PaCMap reduces network traffic volume by up to 30% (not shown)

A user perspective

- Sebastien Rumley, Columbia University

Agenda

Afternoon

- Tour of SST Elements (continued)
- Demo: Using the statistics API
- Use cases
- A user perspective
- **Validation**
- Future developments
- Wrap-up

Validation

- On-going effort to validate simulation against real hardware
  - Compare performance, bandwidth, etc. using kernels & mini-Apps
    - Mini-Apps: DOE benchmarks based on production codes
      - Mantevo Suite: miniFE, miniAMR, miniGhost, etc.
    - Micro-benchamrks: STREAM, LMBench, GUPS
  - Quantize accuracy of high vs. low detail processing models
- Initial results
  - Memory studies focus on bandwidth and latency
    - Have resulted in some bug fixes to memHierarchy
  - Discovered bug where the Intel PIN tool for *Ariel* was not able to follow a forked child despite setting the correct PIN parameters

Validation study #1

- Validate that observed and configured cache latencies match
  - Validate latencies independently for L1 and L2
  - Evaluate using *BlackjackBench*
  - *Ariel* processor + *memHierarchy* L1, L2, and memory
- Results
  - The number of memory accesses reported by Ariel matches the count estimated from the application source
  - As expected, load latencies change at cache size boundaries
  - When data fits in L1, average load latency matches configured latency
  - Latency for L2 is less than expected
    - MemHierarchy computes latency correctly
    - Caused by complex interaction between Ariel and application
      - Especially when using *gettimeofday* system call

Validation study #2

- Validate the bandwidth and latency reported by VaultSim and DRAMSim
  - Evaluate using the *Mantevo* miniApp suite
  - *Ariel* processors + *memHierarchy* caches and memory + *merlin* network
- Results
  - Coarse grain: VaultSim performs better than DDR3 (as expected)
  - Fine grain: Working to establish that latencies between caches and memory are correct (in progress)
  - Led to modification of the memNIC to support larger buffers
  - Led to improved latency statistics for caches
  - Takeaway: Care must be taken in setting bandwidth, clock rates, buffer sizes, etc. across many components and links to achieve specified bandwidth

Agenda

Afternoon

- Tour of SST Elements (continued)
- Demo: Using the statistics API
- Use cases
- Validation
- A user perspective
- **Future developments**

Current development efforts

- Re-integrating SST & Gem5
  - Previous integration was with a branch of Gem5, emulation mode only
  - New integration is with the main Gem5 stable release
    - Ability to run full-system
  - Testing of the new integration is underway
  - Integration is owned by Gem5
- Parallel simulation via threads
  - SST core relies on MPI for parallel simulation
  - Recently began an effort to integrate threading into the SST core
  - Enable parallel simulation via threads or MPI + threads

Current development efforts

- SST Macro integration
  - SST Macro: effort to enable flexible, full-system simulations
    - Coarse grained architecture and OS models
    - Direct compilation of “skeleton” application source code
  - Combined focus on hardware *and* software design
    - Co-design of app, runtime, middleware, and hardware
  - SST Macro also does parallel discrete-event (PDES)
  - Effort to integrate macroscale PDES algorithms (MPI + PThread) into SST core

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
- SST maintains mailing lists for additional support
  - ***sst-user*:** For questions on building, compiling, extending, and using SST
  - ***sst-developer*:** For questions on developing SST components
  - *sst-announce*: Release announcements
  - *sst-commit*: Notification of commits to the SVN repository
  - Subscribe via the wiki

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

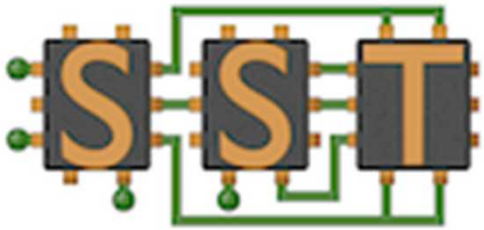
- Thank you for attending!

Exceptional service in the national interest

Sandia National Laboratories

Photos placed in horizontal position with even amount of white space between photos and header

Structural Simulation Toolkit (SST)



June 13, 2015

ISCA Tutorial

U.S. DEPARTMENT OF ENERGY

National Nuclear Security Administration

Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000. SAND NO. 2011-XXXXP