Sandia National Laboratories

Exceptional service in the national interest

The Structural Simulation Toolkit (SST)

IPDPS 2025

Presented by: Scott Hemmert, Clay Hughes, Joe Kenny, Gwen Voskuilen,
Sandia National Laboratories

Content by: SST Team

Milan, Italy

Tuesday, June 3

SAND2025-06737C

Sandia National Laboratories

Exceptional service in the national interest

SST Tutorial – IPDPS 2025

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and 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.

U.S. DEPARTMENT OF ENERGY

NNSA National Nuclear Security Administration

Instructors

Gwen Voskuilen	grvosku@sandia.gov
Scott Hemmert	kshemme@sandia.gov
Clay Hughes	chughes@sandia.gov
Joseph Kenny	jpkenny@sandia.gov
Dave Donofrio	ddonfrio@tactcomplabs.com
John Leidel	jleidel@tactcomplabs.com

Welcome!

Part 1: Introduction to SST

- SST overview – Why, What, How
- Basic simulation workflow
- A Tour of SST Elements

Break *15:30 – 16:00*

Part 2: Full System Modeling with SST

- Networks
- Applications

Learning Objectives – Part 1: Introduction to SST

This section of the tutorial will cover the following topics:

1. Why use SST?
2. What capabilities does SST provide?
3. How do I use SST?
    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
4. A summary of the many available components
5. 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 -> “ipdps2025_day1”
  - “Create codespace”
- Alternate instructions are available at https://github.com/sstsimulator/sst-tutorials/tree/master/ipdps2025
- 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

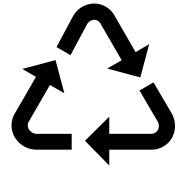
- Configuration File Format: http://sst-simulator.org/sst-docs/docs/guides/configuration/pythonConfigGuide
- Getting Started: http://sst-simulator.org/sst-docs/docs/guides/start
- SST-Core Doxygen: http://sst-simulator.org/SSTDoxygen/15.0.0_docs/html/
- Building SST
  - Quick start: http://sst-simulator.org/SSTPages/SSTBuildAndInstall_15dot0dot0_SeriesQuickStart/
  - Detailed: http://sst-simulator.org/SSTPages/SSTBuildAndInstall_15dot0dot0_SeriesDetailedBuildInstructions/

Part 1: Introduction to SST

Simulation appears in many contexts

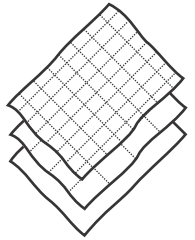
Research ***Prototyping*** ***Validation***

- When hardware or software doesn't exist yet
- When hardware exists but is difficult to access or measure



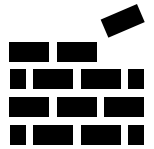
Reuse

- Different fidelities
- Different timescales
- Different system scales
- Related and sometimes intertwined problems



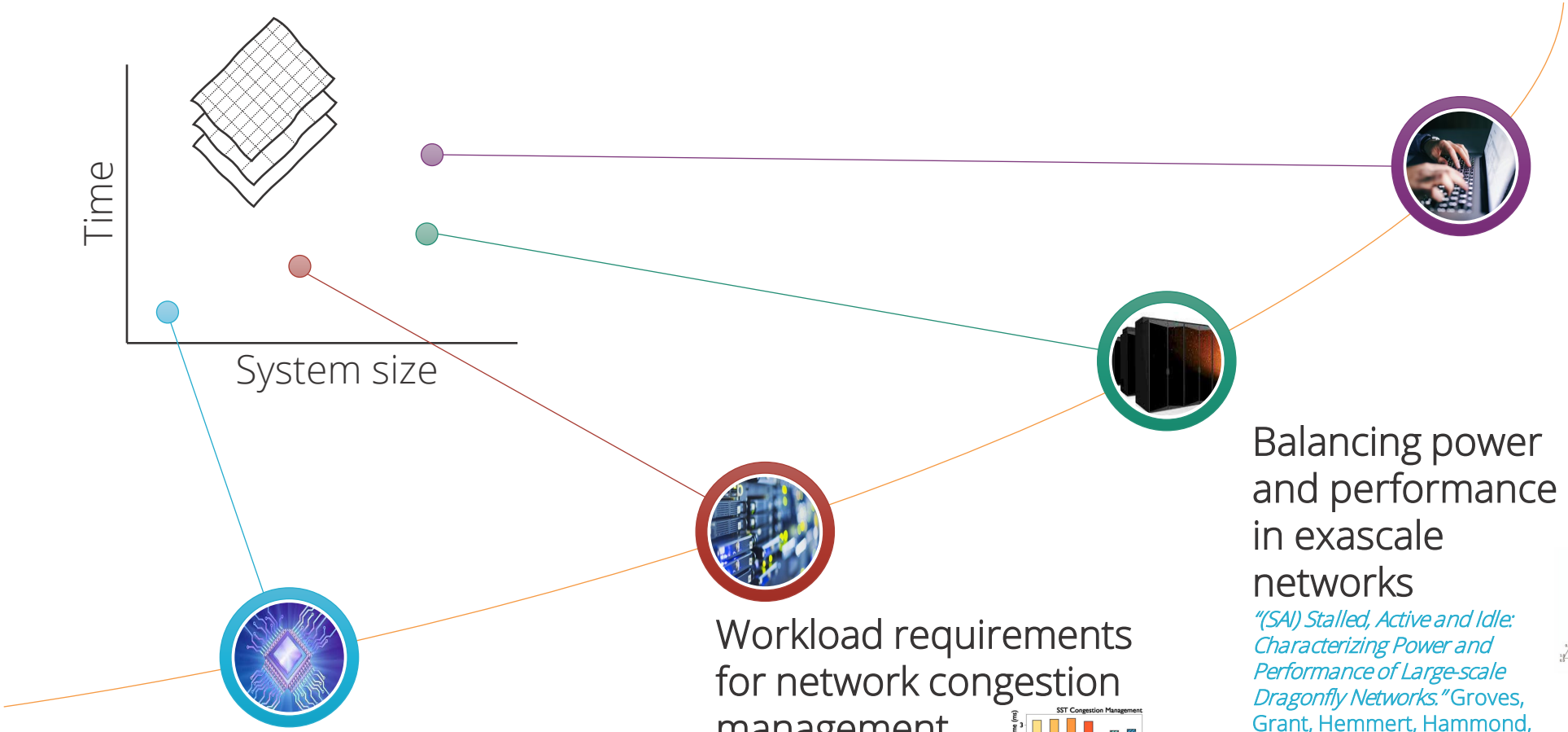
Multi-fidelity

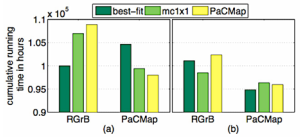
One ecosystem that can adapt to multiple needs and leverage community expertise



Modular

Simulation at many scales



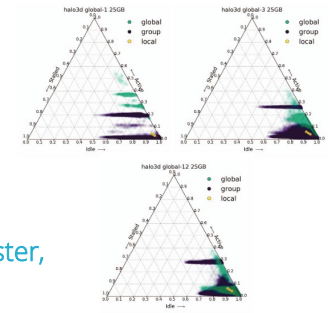


Job scheduling algorithms to improve runtime and throughput

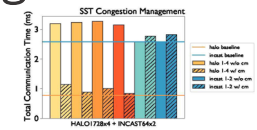
"PaCMap: Topology Mapping of Unstructured Communication Patterns onto Non-contiguous Allocations" Tuncer, Leung, Coskun. ICS, 2015.

Balancing power and performance in exascale networks

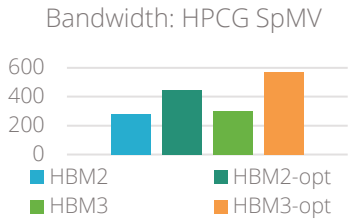
"(SAI) Stalled, Active and Idle: Characterizing Power and Performance of Large-scale Dragonfly Networks." Groves, Grant, Hemmert, Hammond, Levenhagen, Arnold. IEEE Cluster, 2016.



Workload requirements for network congestion management



"Exploration of Congestion Control Techniques on Dragonfly-class HPC Networks Through Simulation." McGlohon, Hemmert, Brown, Levenhagen, Chunduri, Ross, Carothers. PMBS Workshop. 2021.



On-chip network limits memory bandwidth

"HBM2/3 Evaluation on Many-Core CPU." Voskuilen, Gimenez, Peng, Moore, Gokhale. Report, Exascale Compute Project. 2018.

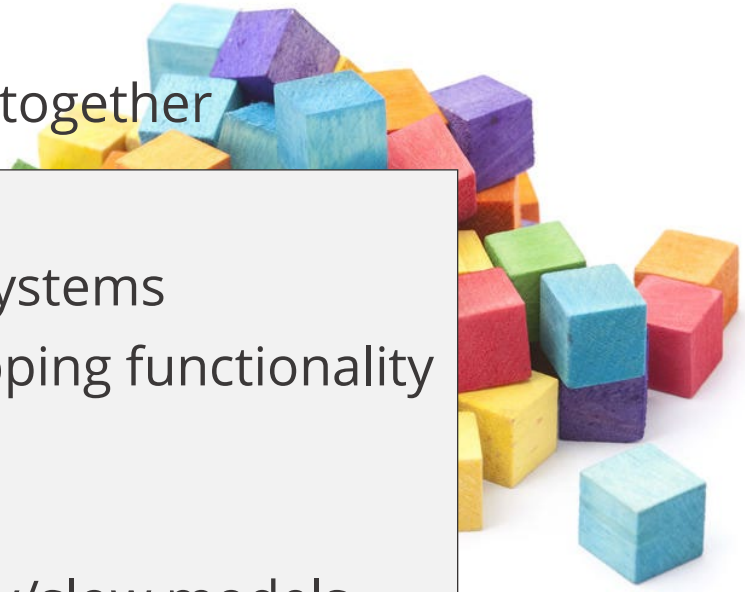
SST Overview

Why SST?

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 current and emerging 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 (15.0.0)
  - 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

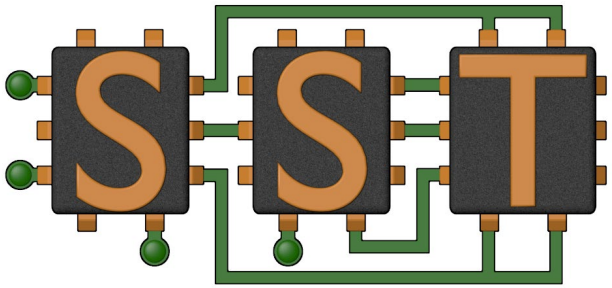
Open API

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

Time-scale independent core

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

C++, Python



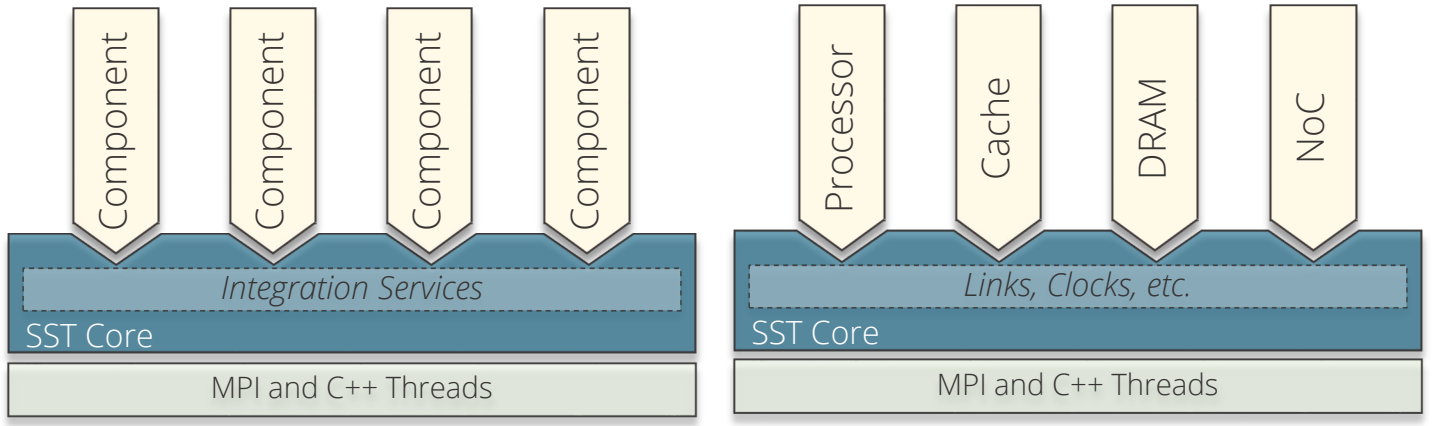
SST Architecture

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

- Models that perform the actual simulation
- Elements include processors, memory, network, etc.
  - Leverages many existing simulators: Spike, DRAMSim3, Ramulator2, etc.



Building Blocks: 101

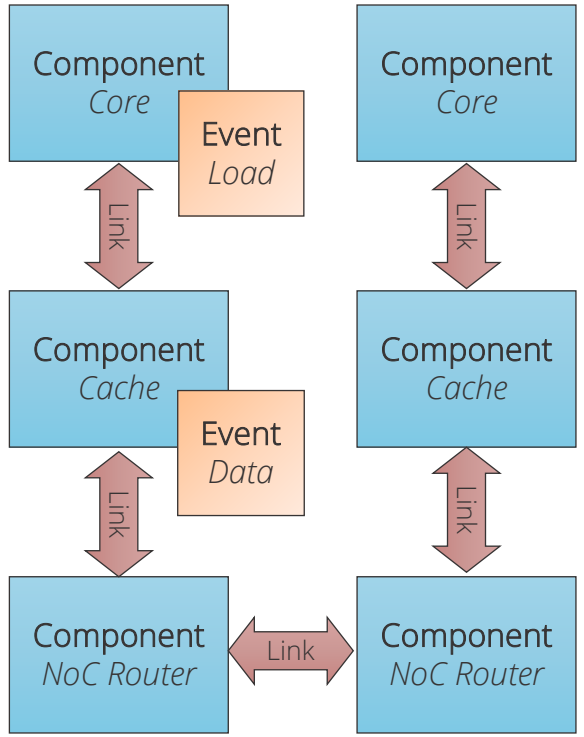
In SST, systems are described as a set of **components**

Components are connected by **links**

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

When creating models for SST, developers map their models to this structure.

When running models in SST, the user maps their desired system to this structure and tells SST which components to use and how they are linked.



Element Library

A collection of components (and other SST building blocks)

Building Blocks: Restrictions

Developers:

- Components may **only** communicate with other components via events

Users:

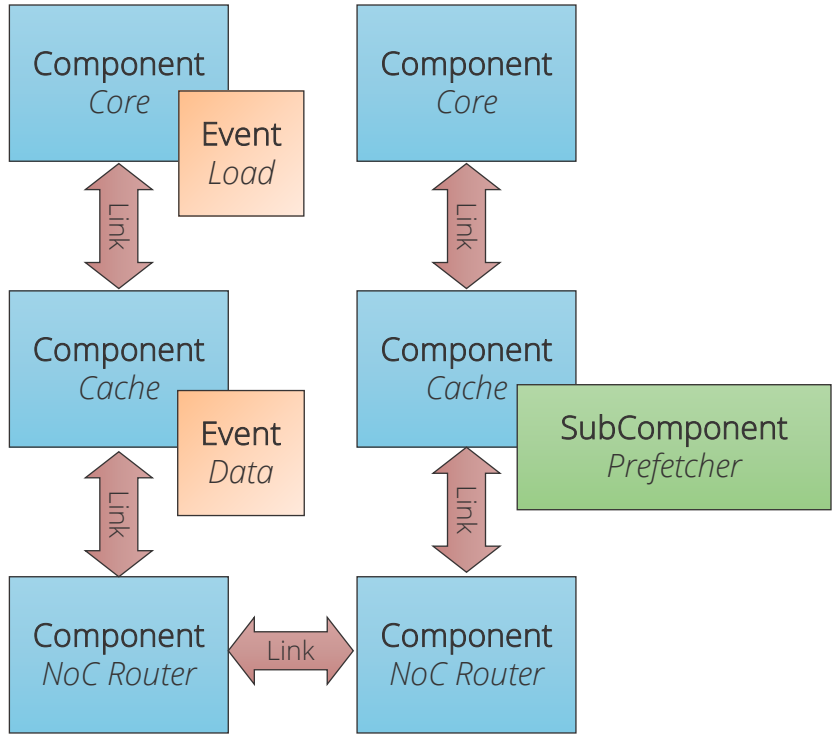
- Every event sent between components incurs a latency
- The latency cannot be zero and **the minimum latency is set by the user**

Building Blocks: 201

SubComponents and **Modules** allow users to swap in different functionality *within* a Component.

- SubComponents are larger and might participate in event exchange
  - E.g., a prefetcher
- Modules are smaller, with very limited access to simulation APIs
  - E.g., a hash function

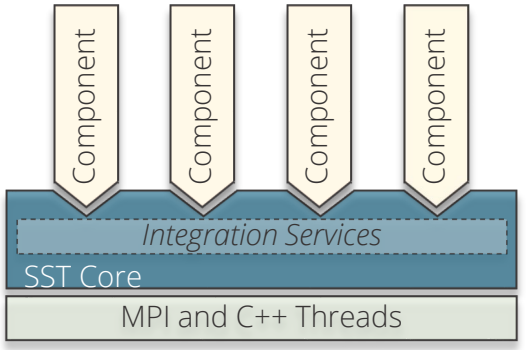
- **Standardized interfaces** for certain types of components makes swapping components simpler
  - Endpoint / Network: SimpleNetwork
  - Processor / Memory: StandardMem



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 basic configurations

Basic SST Simulation

Simulation in just 3 steps… ☺

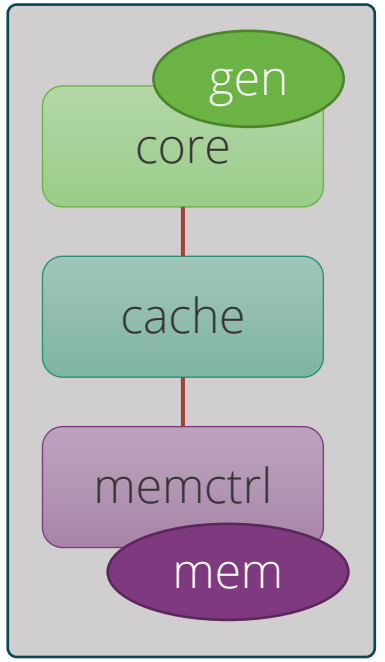
1. Define the system you want to simulate
2. Run SST
3. Analyze output

SST input files describe the *graph* of the system to be simulated

- The input file describes:
  - The **components** that make up the simulation (vertices) including their parameters
  - The **links** between the components and their latencies (edges)

- It also may:
  - Define any global options that control the simulation as a whole
  - Declare which statistics to collect and how to report them

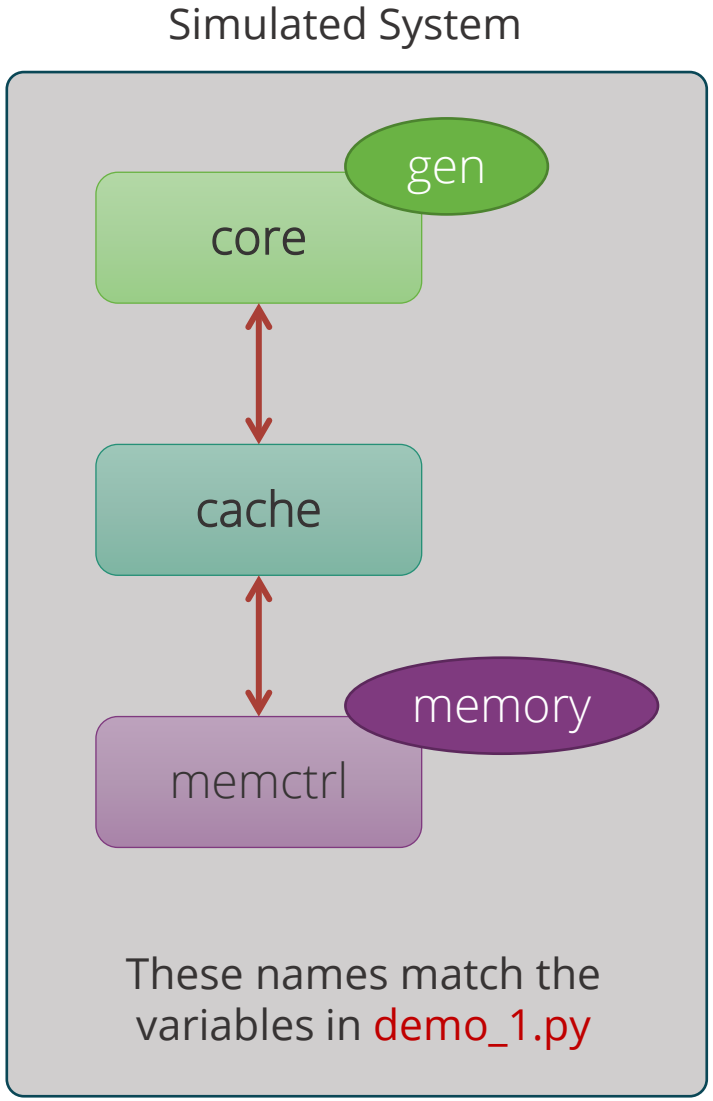
- Typically written in Python
  - JSON also supported



Our First Simulation – demo_1.py

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

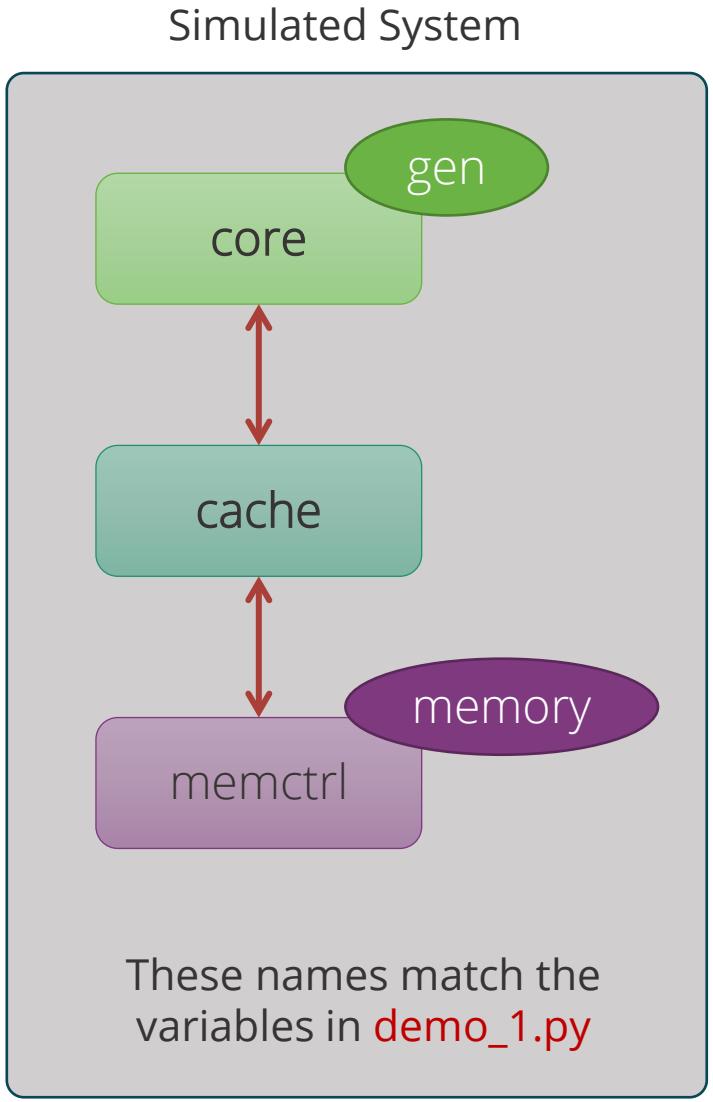
- *Location:* sst-tutorials → ipdps2025 → exercises → single-node



Our First Simulation – demo_1.py

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 can also 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

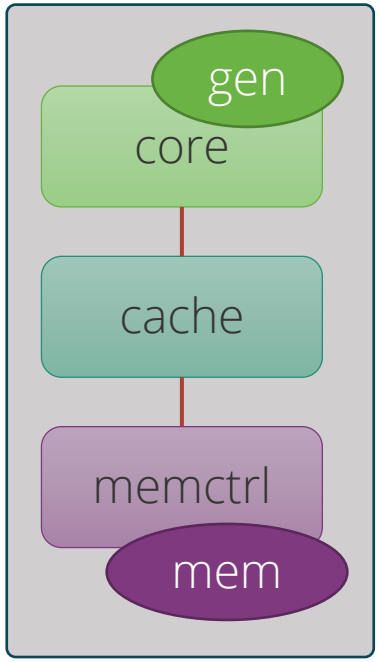
SST Python API
User-defined string
SST argument

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

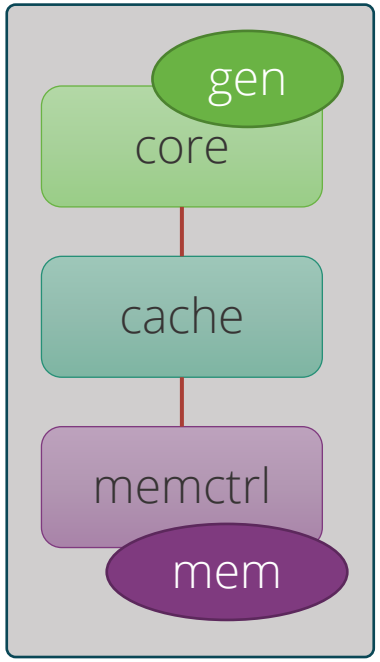


Configuration File: Configure the core

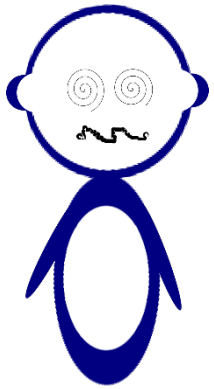
SST Python API
User-defined string
SST argument

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]
         ...
```

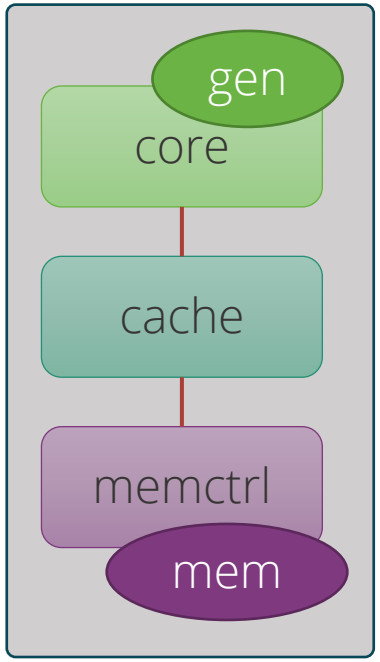
Configuration File: Add a subcomponent

SST Python API
User-defined string
SST argument

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

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

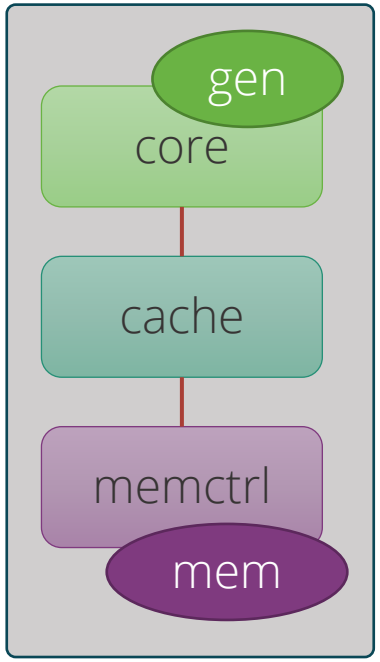
```
gen = core.setSubComponent("generator", "miranda.STREAMBenchGenerator")
gen.addParams({
    "n" : 1000, # Number of array elements
})
```



Configuration File: Configure the cache

SST Python API
User-defined string
SST argument

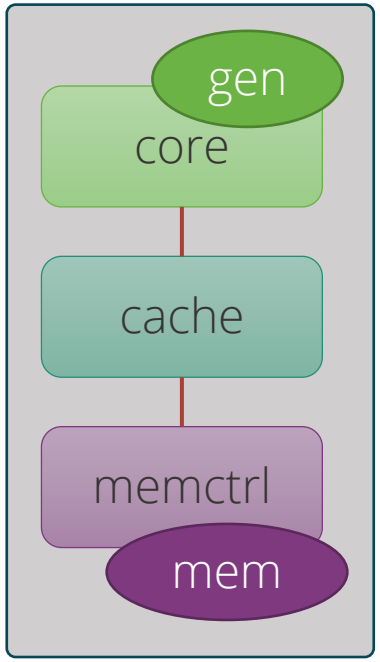
```
cache.addParams({
    "L1" : 1,
    "cache_frequency" : "2.4GHz",
    "access_latency_cycles" : 2,
    "cache_size" : "2KiB",
    "associativity" : 4,
    "replacement_policy" : "lru",
    "coherence_policy" : "MESI",
    "cache_line_size" : 64,
})
```



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",
})
```

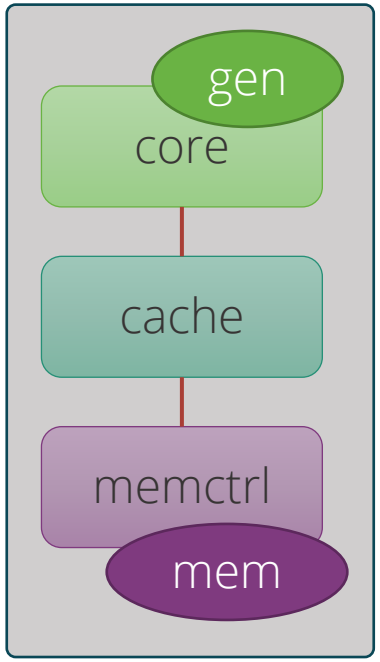


Configuration File: Declare and connect

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, "highlink", "100ps") )

cache_mem.connect( (cache, "lowlink", "100ps"),
                   (memctrl, ”highlink", "100ps") )
```

Link name



Configuration File: Connect links

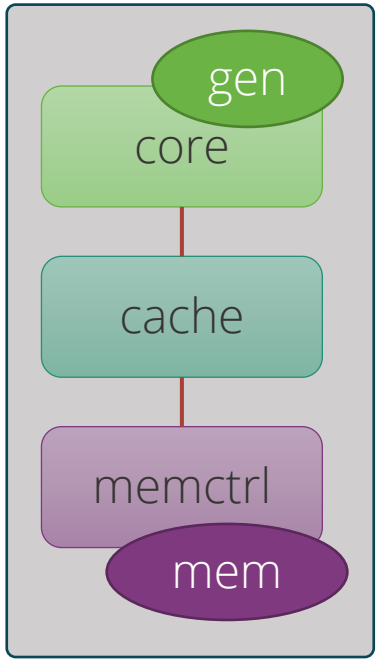
Connect components: `connect(endpoint1, endpoint2)`

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

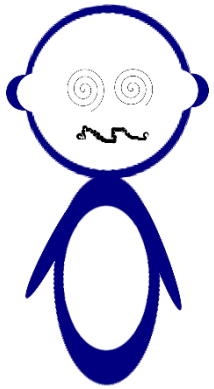
```
##########################################
## Connect components with the links
##########################################
core_cache.connect( (core, "cache_link", "100ps"),
                    (cache, ”highlink", "100ps") )

cache_mem.connect( (cache, "lowlink", "100ps"),
                   (memctrl, ”highlink", "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 ()
Components (15 total)
   Component 0: Cache
      Description: Cache controller
…
      Parameters (34 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 (12 total)
         highlink: Non-network upper/processor-side link (i.e., link towards the core/accelerator/etc.).
…
      Statistics (48 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 (48 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
```

The input file declare which statistics to report (enable)

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

…and how to report them

- 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

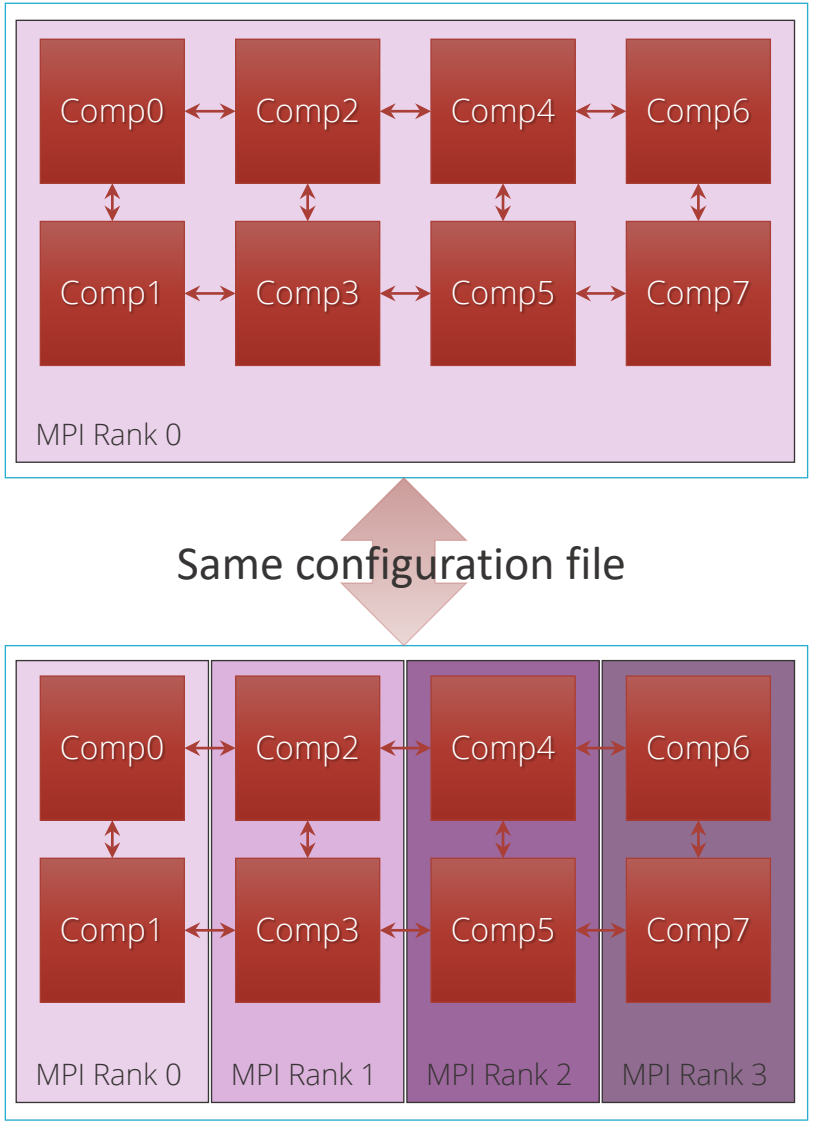
SST was designed from the ground up to enable scalable, parallel simulations

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 15.0 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
- **Mercury** – Application skeletons

Networks/NoCs

- **Merlin** – flexible network modeling
- **Kingsley** – mesh NoC

Processor Models

Elements:

Ariel Prospero

Miranda Vanadis

Ariel: PIN-based processor

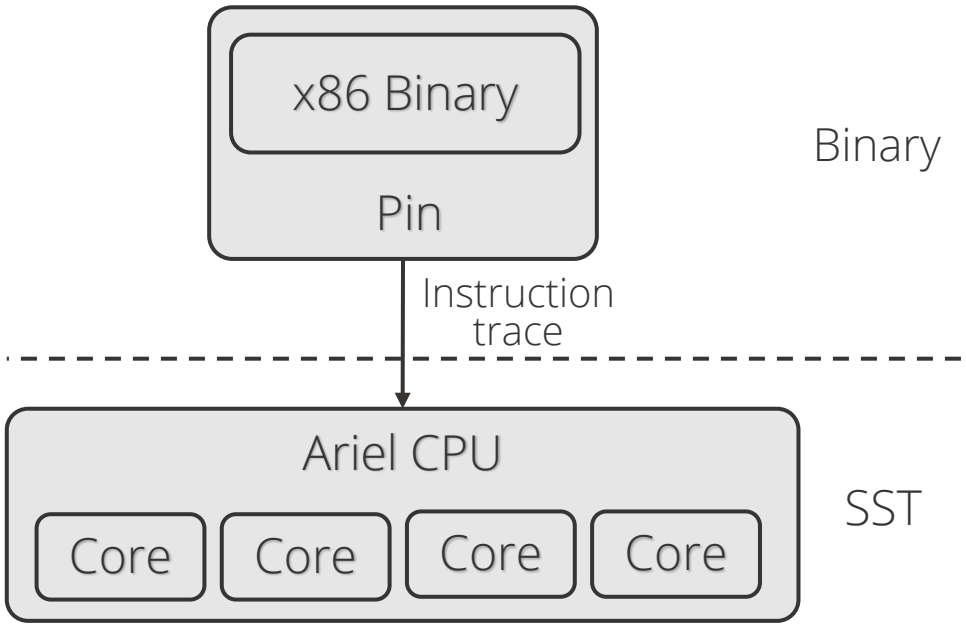
$ sst-info ariel.ariel

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



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

★ GPGPU-Sim Integration

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

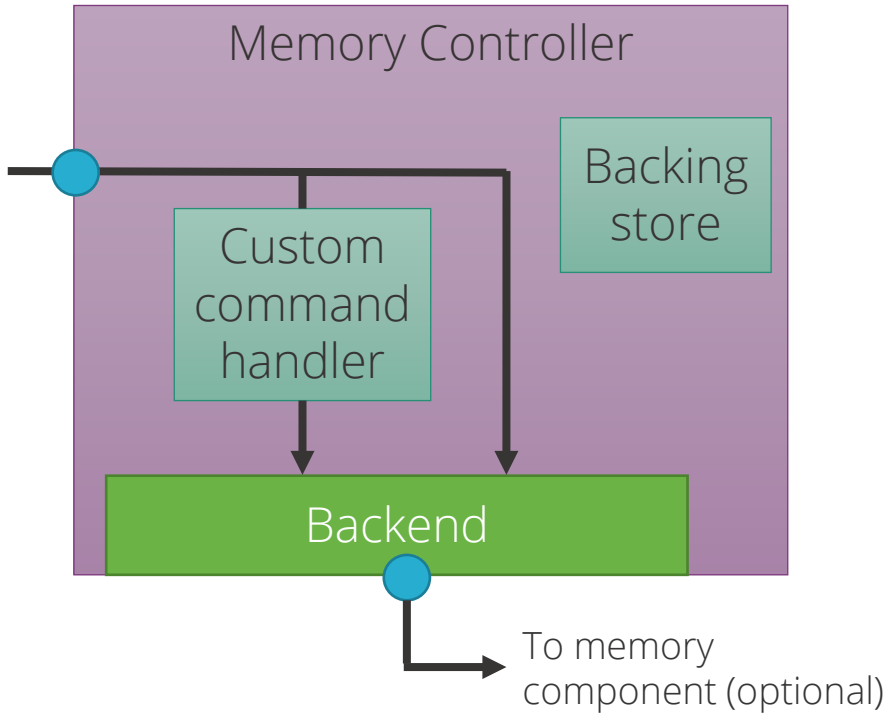
$ sst-info memHierarchy.MemController

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 15.0 backends

Memory (external)

- CramSim (DDR, HBM)
- DRAMSim3 DDR, LPDDR, GDDR HBM, HMC, STT-MRAM)
- HMCSim/GoblinHMC (HMC)
- Messier (NVRAM)
- Ramulator (DDR, HBM, HMC)
- Ramulator2 (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

<u>Elements:</u>

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

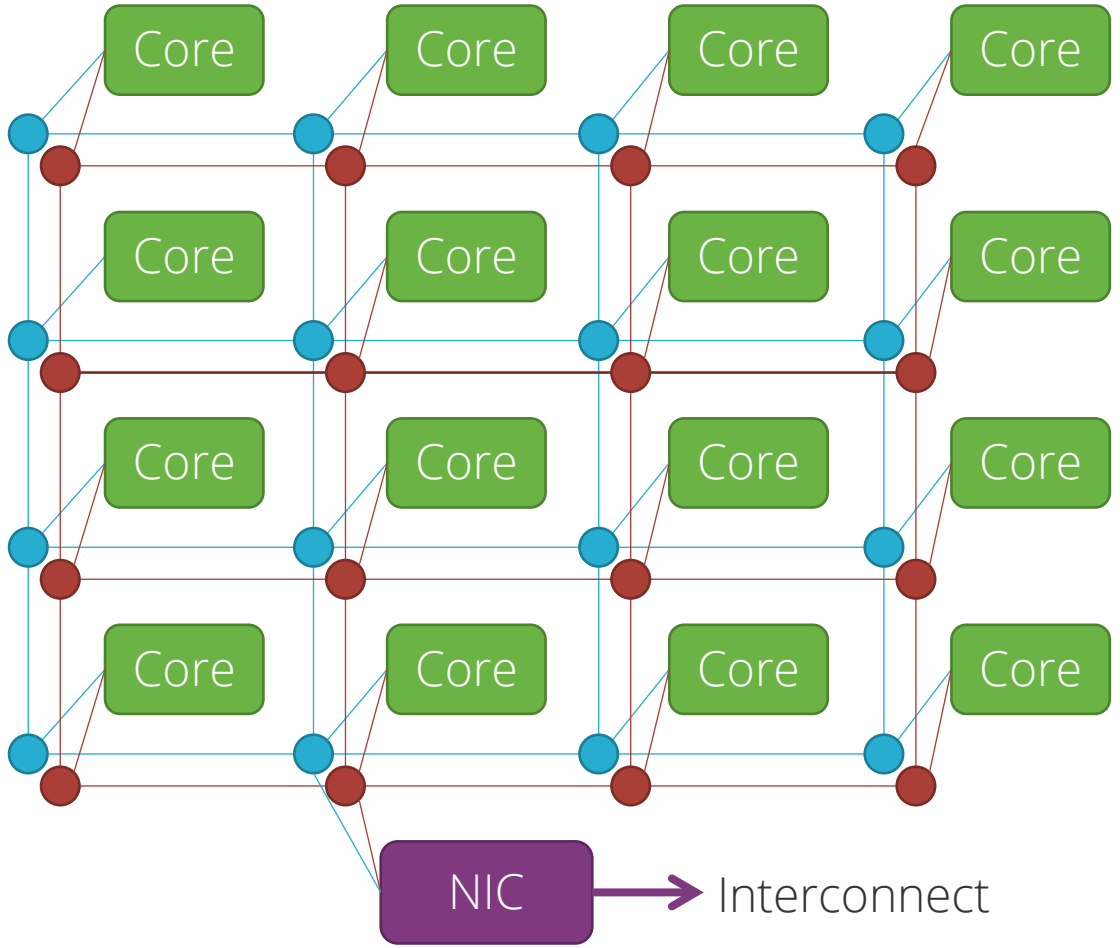
Kingsley: Mesh simulator

$ 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

Mercury

Iris

Deep dive on this topic will close the session

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

Mercury: Traffic Generator / Workload Modeling

$ sst-info hg

Light-weight endpoint modeling from SST/macro

- Virtual processes/tasks run as lightweight (user space) threads to manage overhead
- Executes relatively unmodified application code

An alternative driver for Merlin networks

(More) SST 15.0 Elements

Processors

- **Ariel** – PIN-based
- **Prospero** – trace execution
- **Miranda** – pattern generator
- **Vanadis** – MIPS32 and RV64 pipeline
- GeNSA – Spiking temporal processing unit
- Osseous – RTL simulation

Memory Subsystem

- **MemHierarchy** – caches, directory, memory
- CacheTracer – cache tracing
- Cassini – cache prefetchers
- CramSim – DDR, HBM
- Messier – NVM
- Samba – TLB
- VaultSim – vaulted stacked memory

Network drivers

- **Ember** – communication patterns
- **Firefly** – communication protocols
- **Hermes** – MPI-like driver interface
- Zodiac – trace based driver
- Thornhill – memory models for Ember

Networks/NoCs

- **Merlin** – flexible network modeling
- **Kingsley** – mesh NoC
- Shogun – crossbar NoC

Accelerators

- Balar – GPGPU-Sim interface
- Golem – Array-based in-situ computing
- Llyr – spatial compute

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_15dot0dot0_SeriesDetailedBuildInstructions/
- Many memory timing simulators are supported by memHierarchy

Viewing Configuration Graph

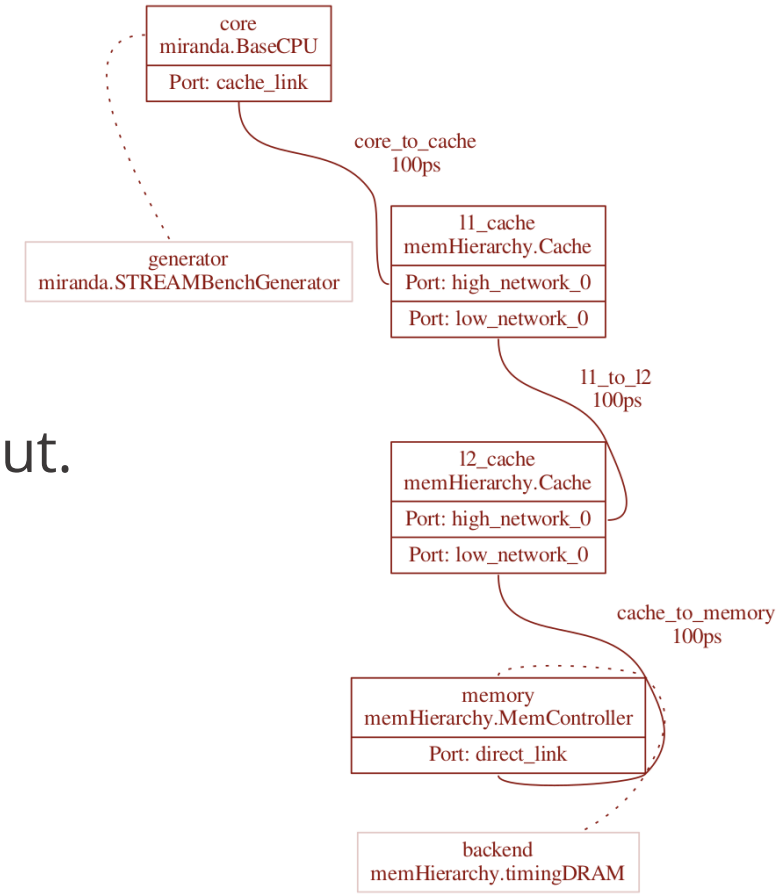
Let's take a look at how SST views our system

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

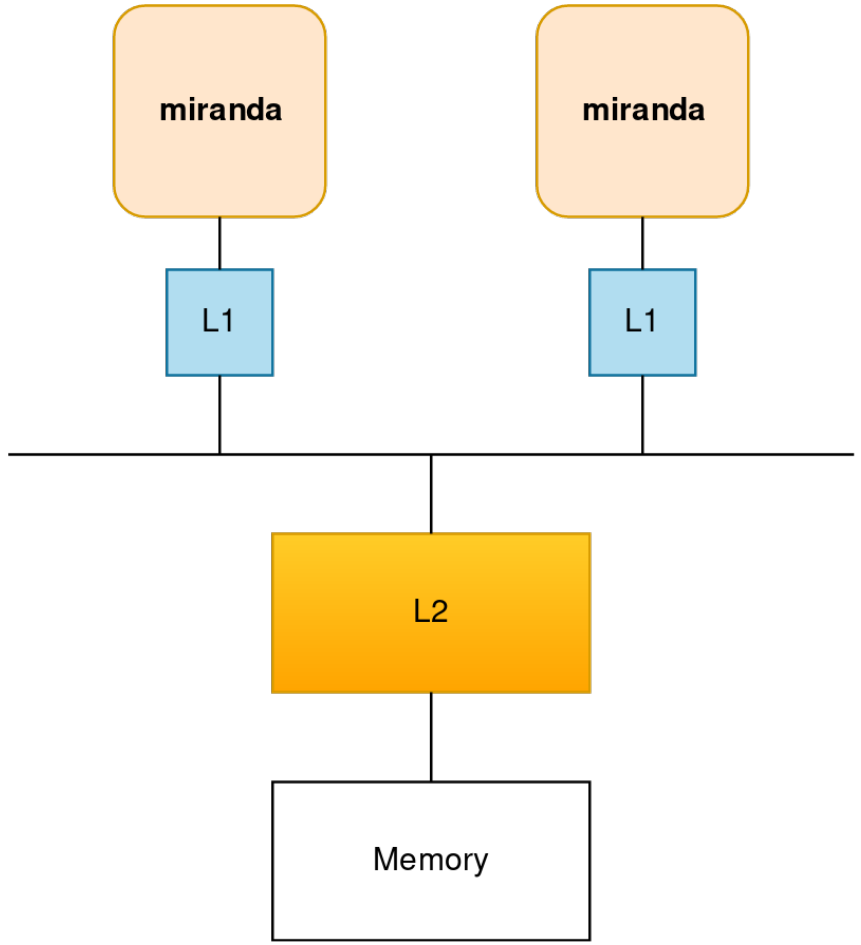
- What else might you need?

```
• 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

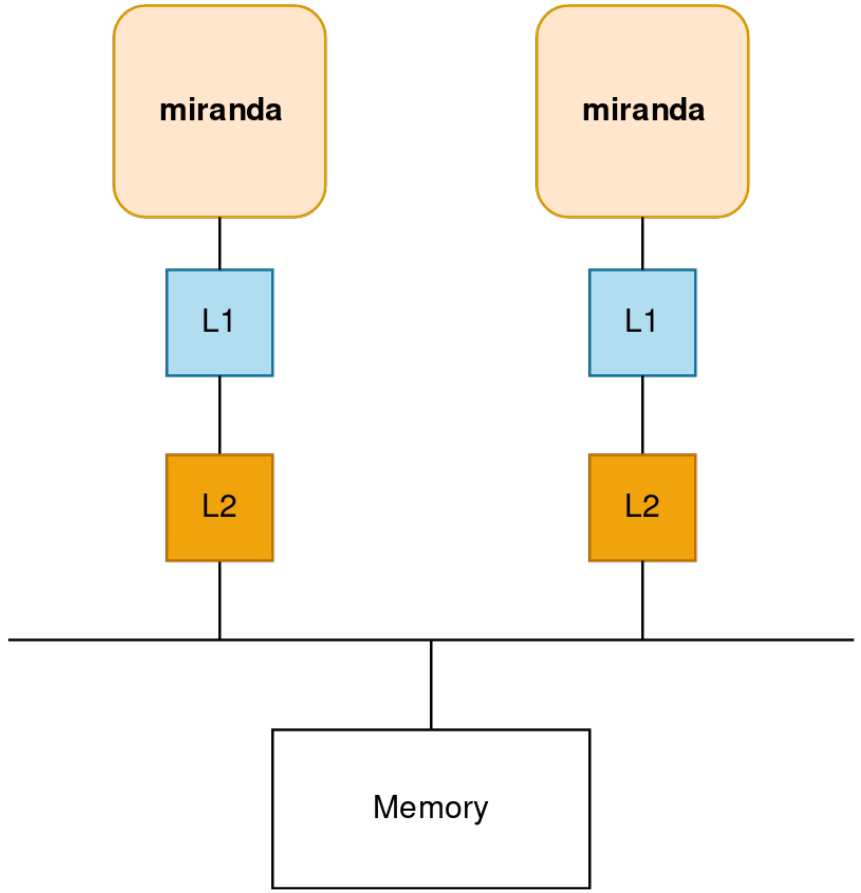
Copy configuration

```
$ 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
```

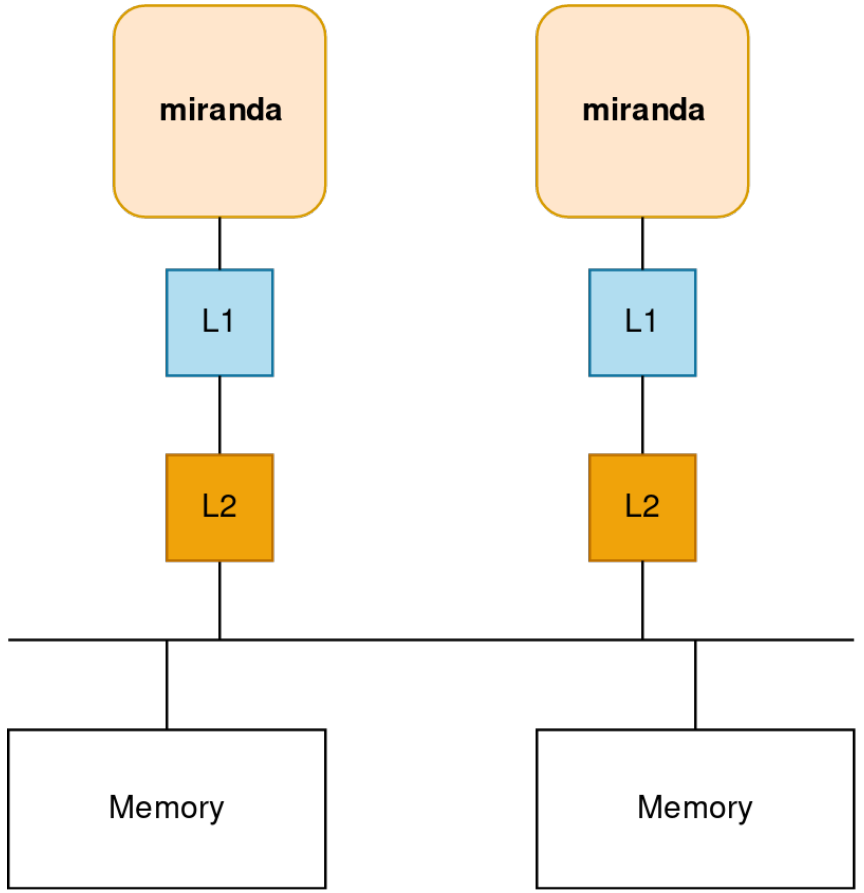
Add an a second L2 and memory controller

- 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

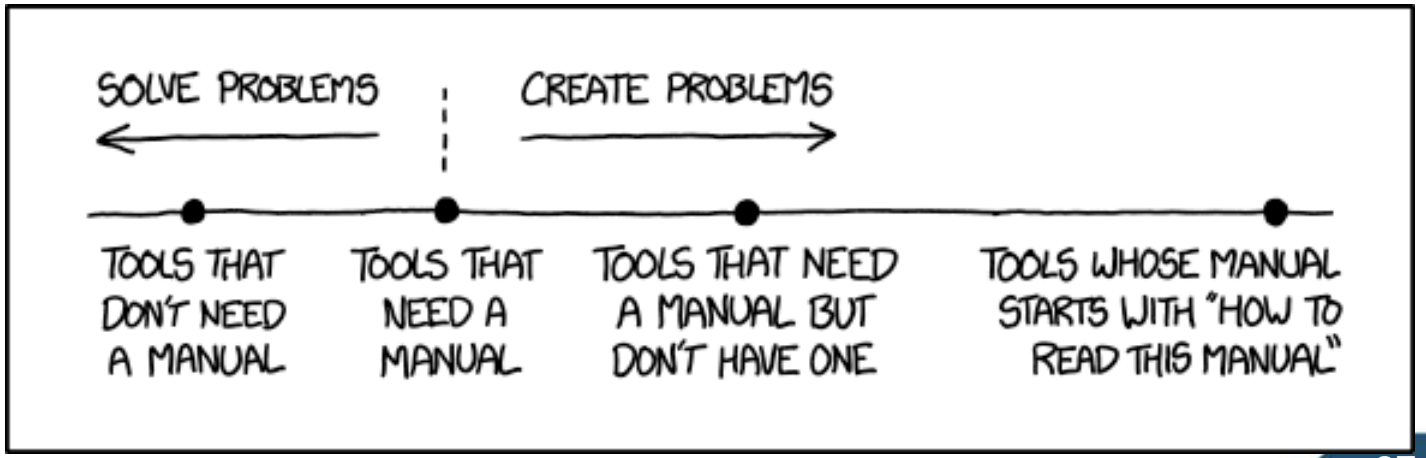
SST is a parallel, flexible simulation framework

- 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

- SST overview – Why, What, How
- Basic simulation workflow
- A Tour of SST Elements

Break *15:30 – 16:00*

Part 2: Full System Modeling with SST

- Networks
- Applications

BREAK

Return at 16:00

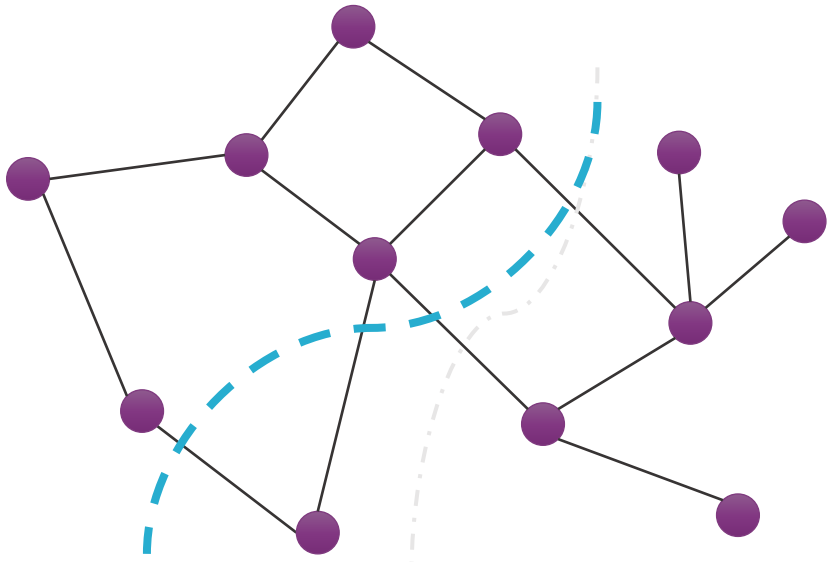
Introduction
Backup slides

SST uses *Parallel Discrete Event Simulation (PDES)*

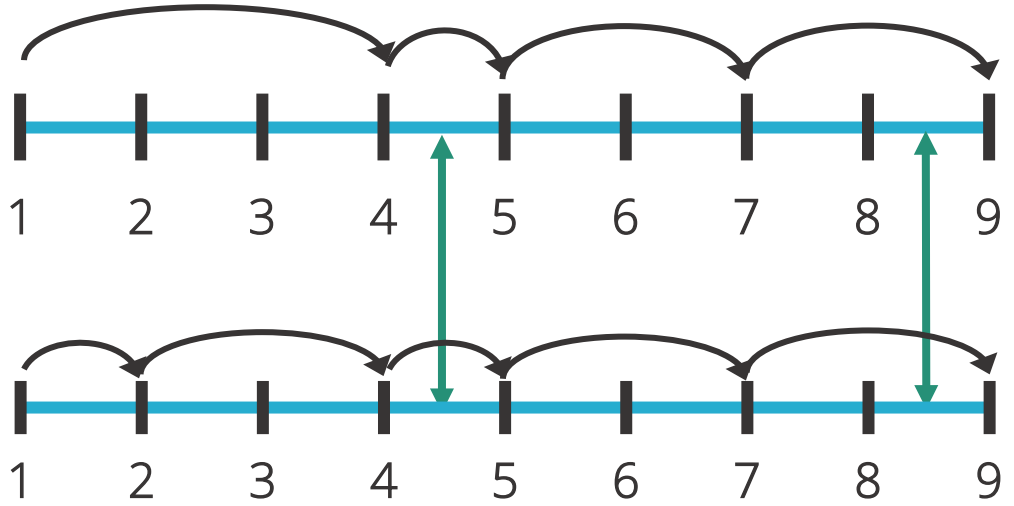
DES breaks system behavior into individual events that update system state

- Periodic (e.g., clock) or aperiodic (e.g., message)

Step 1 Treat simulation as a **graph** of *components* connected by *links*



Step 2 Exchange events and synchronize so that events *never "arrive in the past"* *(Conservative PDES)*



Merlin Models

Merlin: Network Simulator

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

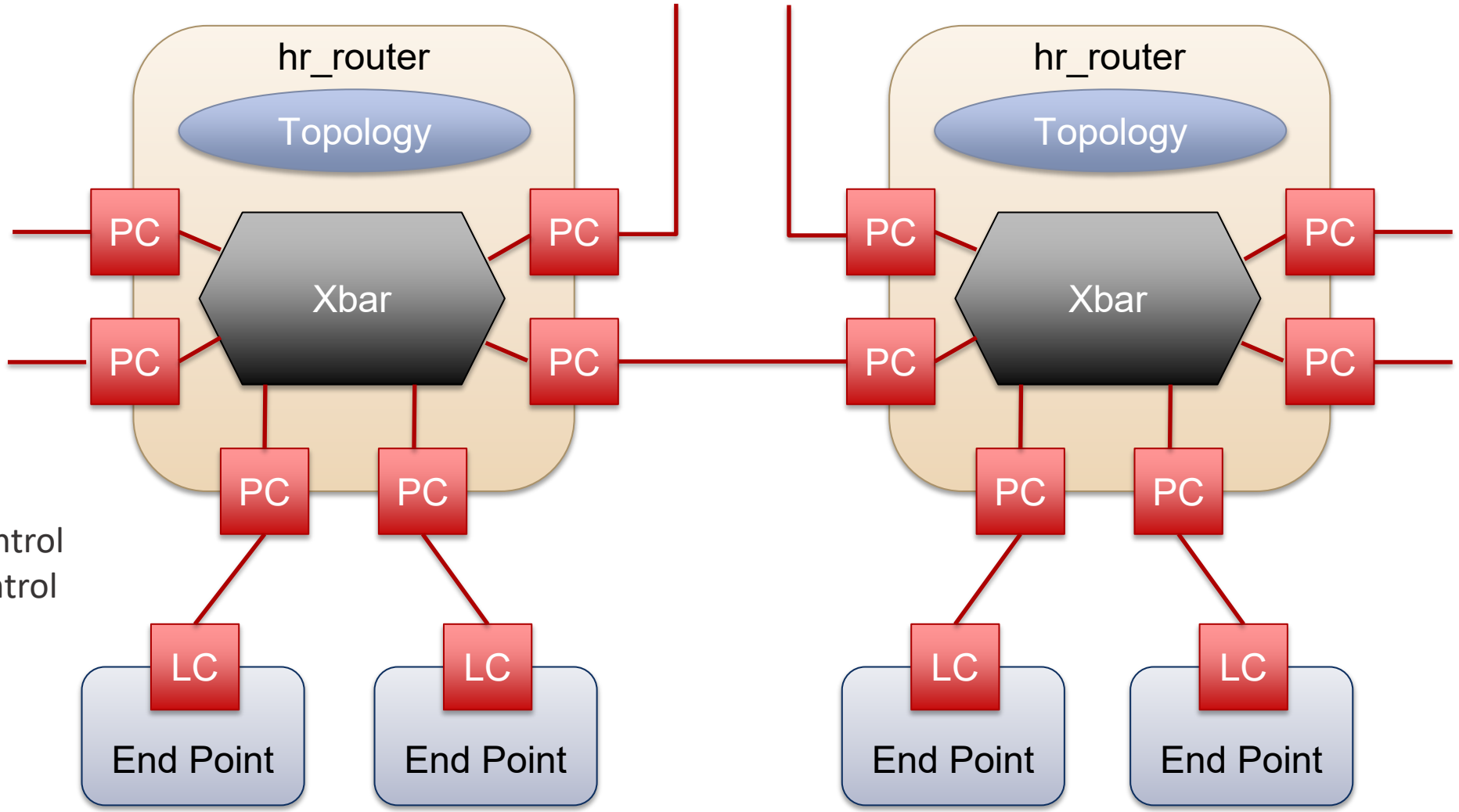
Capabilities

- 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*)
- Application skeletons (*Mercury*)
- 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 stream */

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

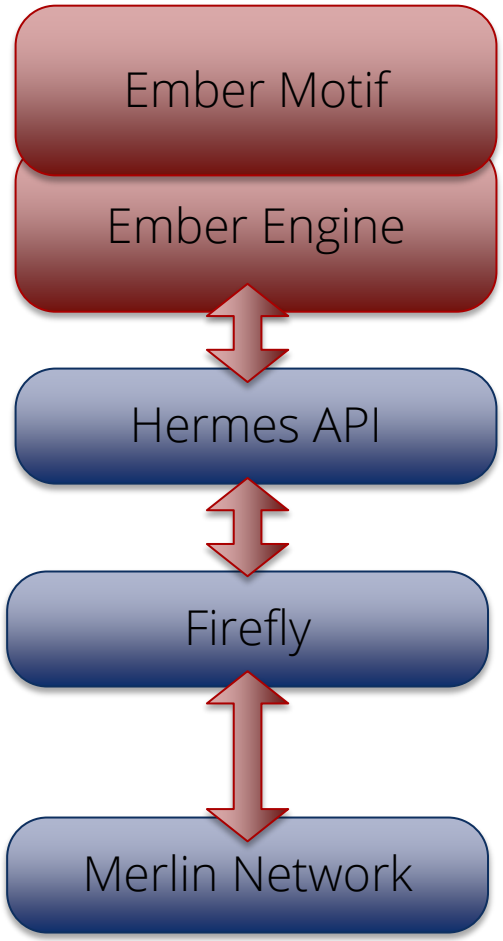
Merlin and Ember provide built-in Python modules to make configuring simulations easier

- 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

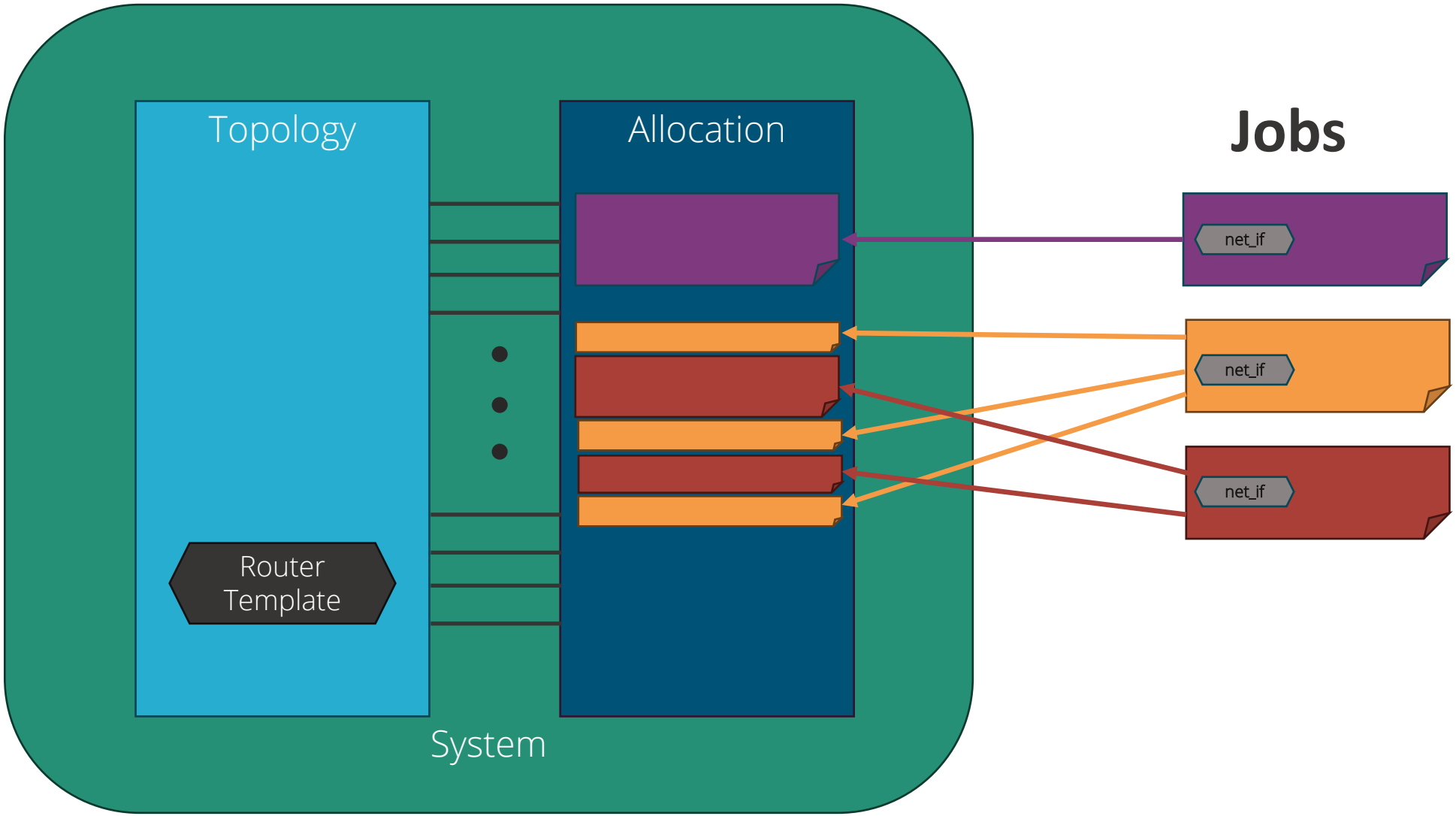
Event to Message Call, Motif Management
Handles the tracking of the motif

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)

Sandia National Laboratories

Exceptional service in the national interest

System-Scale Simulation using Ember, Mercury and Merlin

SST Tutorial – IPDPS 2025

Joseph Kenny, Sandia National Laboratories

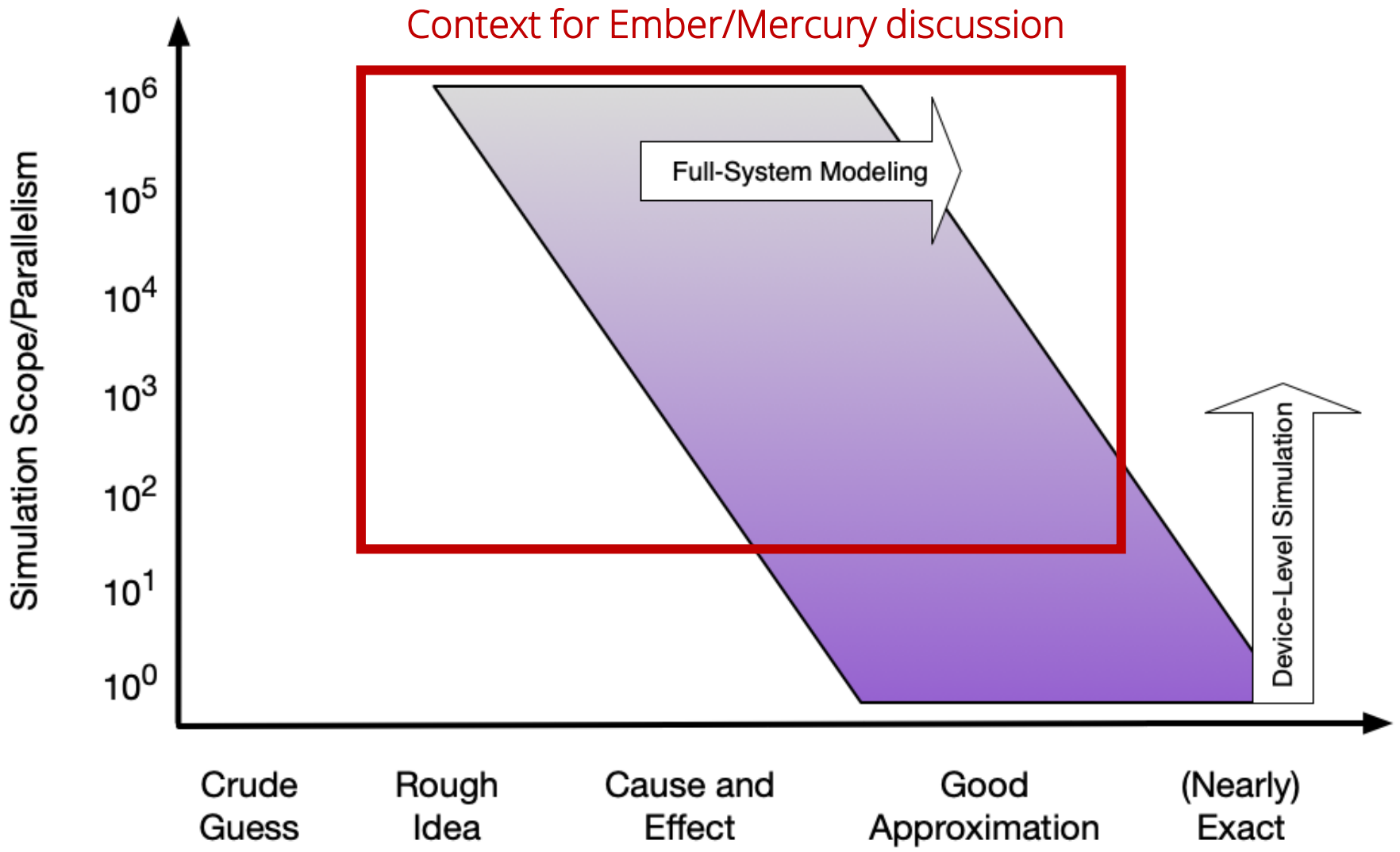
SST Development Team, Sandia National Laboratories

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and 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.

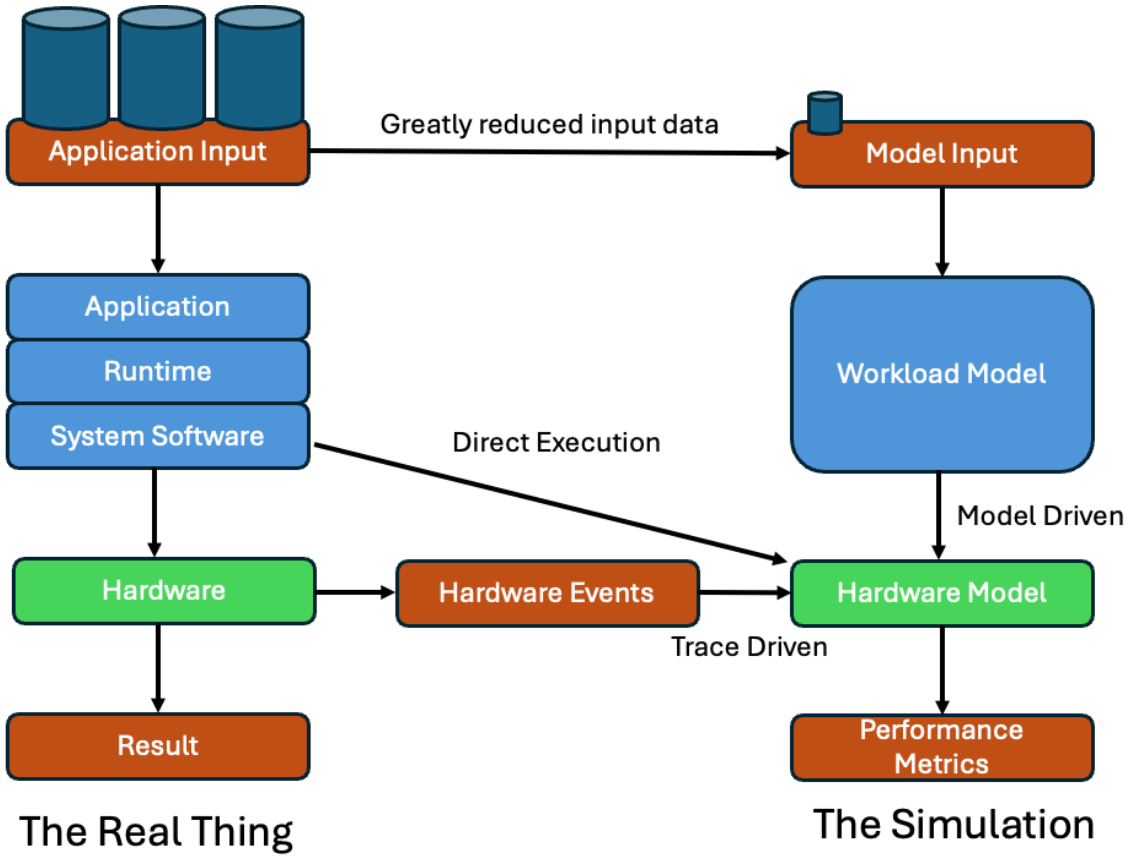
U.S. DEPARTMENT OF ENERGY

NNSA

System Scale Modeling



Simulation Approaches



- Direct execution is unachievable for full system scale
- Traces require full execution at target scale or some way to scale up traces accurately (difficult)
- Ideal workload models use parameterized representation to reproduce data movement and computation – **full results not computed**

Workload Modeling at Scale

- Ember/Mercury/Merlin allow accurate modeling of workload *data movement*
- There is no prescribed method to account for computation
  - Any computation in the model itself is performed on the host CPU
  - Simulation time does not advance due to host computation – explicit addition of computation modeling is required
- Full array of models available to simulation designer
  - Fixed timing estimates from small scale traces or simulations
  - Simple models (e.g. roofline)
  - Full device level simulation at small scale
  - Multifidelity: abstract model parameterized with results from direct execution (very active research area)

Sandia National Laboratories

Exceptional service in the national interest

System-Scale Simulation: Ember

SST Tutorial – IPDPS 2025

Joseph Kenny, Sandia National Laboratories

SST Development Team, Sandia National Laboratories

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and 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.

U.S. DEPARTMENT OF ENERGY

NNSA

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: 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

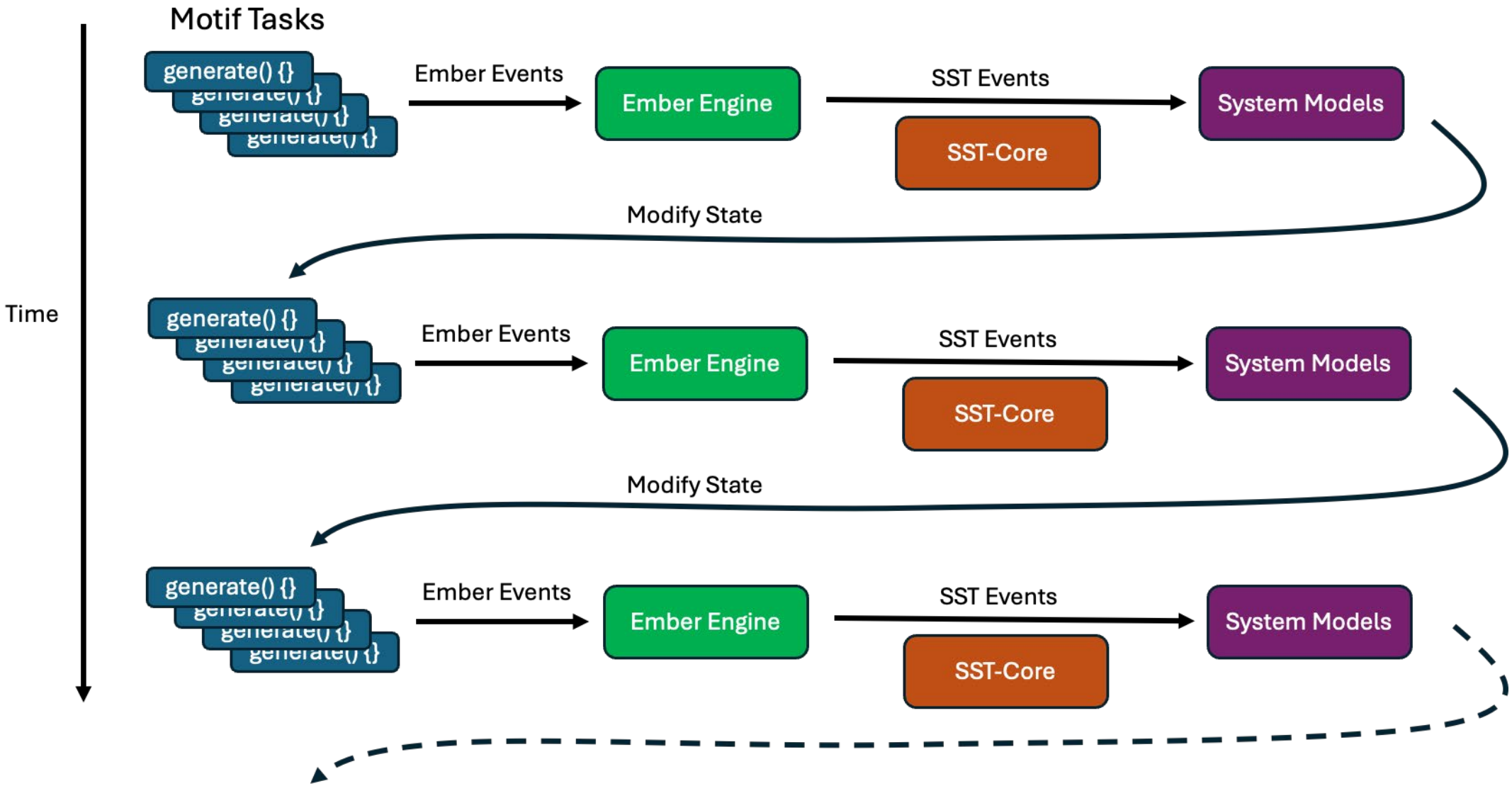
The EmberEngine creates and manages the motif

- 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.

Ember Execution Loop



Sandia National Laboratories

Exceptional service in the national interest

System-Scale Simulation: Mercury

SST Tutorial – IPDPS 2025

Joseph Kenny, Sandia National Laboratories

SST Development Team, Sandia National Laboratories

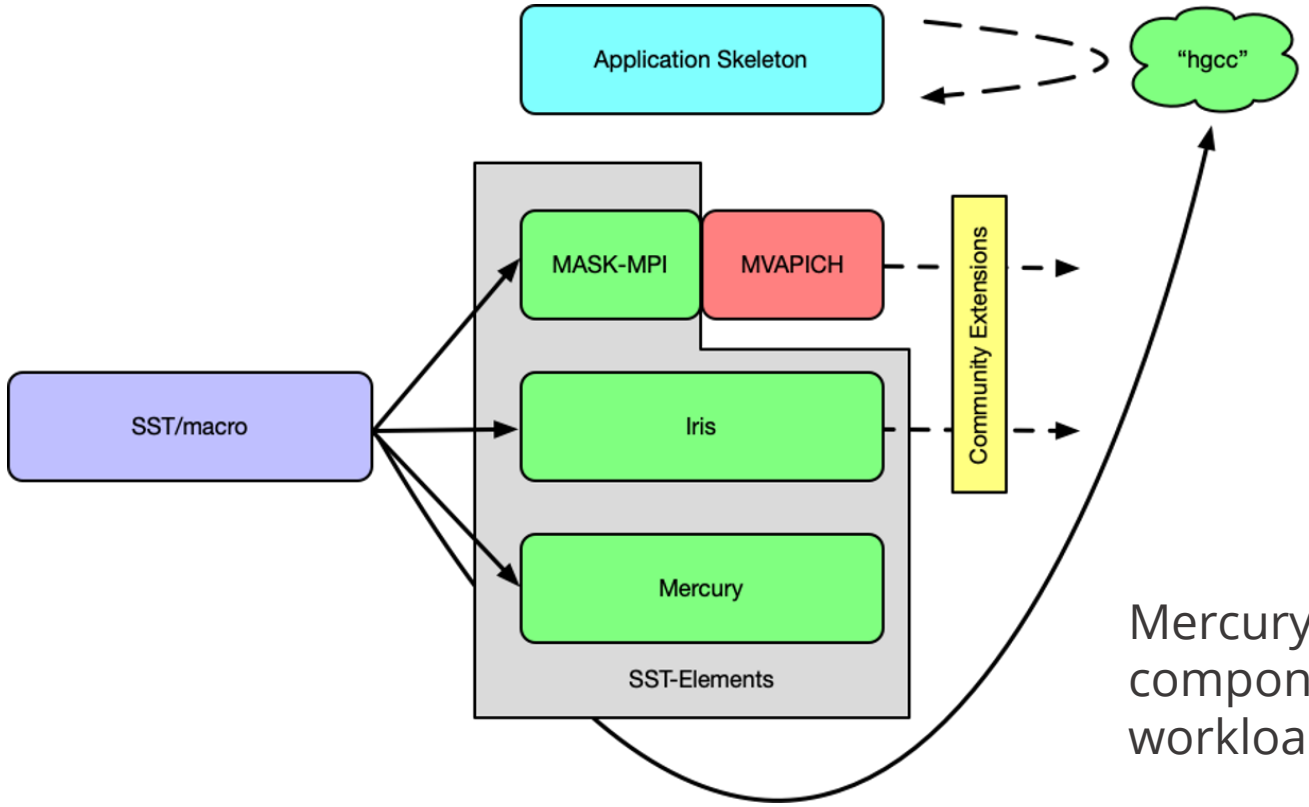
Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia LLC, a wholly owned subsidiary of Honeywell International Inc. for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

U.S. DEPARTMENT OF ENERGY

NNSA

Introducing the Mercury Environment

Mercury is a recent effort to bring workload modeling technology developed by the SST/macro project into a fully integrated environment within SST-Elements



Mercury provides foundational components for efficiently building flexible workload simulations

Automation tick-tock (philosophical digression)

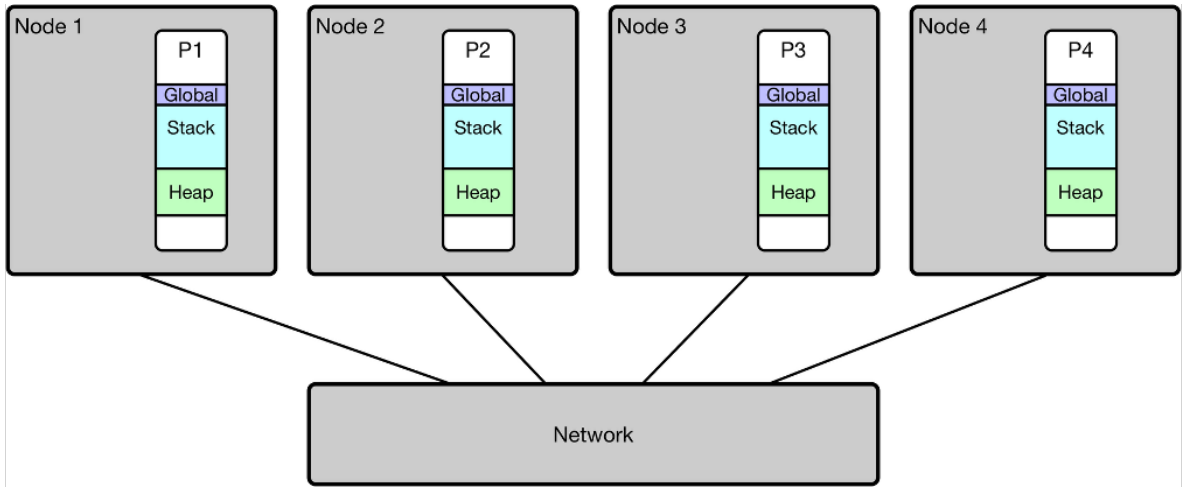
Oversimplified but instructive way to differentiate SST-Elements and SST/macro:

- SST/macro *is a simulator* (drink the Kool-Aid) -- often more end-user friendly, not as easy to mix-and-match/extend.
- SST-Elements is a toolkit, aimed at simulation developers, that *assists building simulators* – more challenging to configure but functionality is well-encapsulated by design.

SST/macro strove towards automation and hiding complexity – great when it worked, hard to understand and work around with it didn't work; it is also fragile and challenging to maintain.

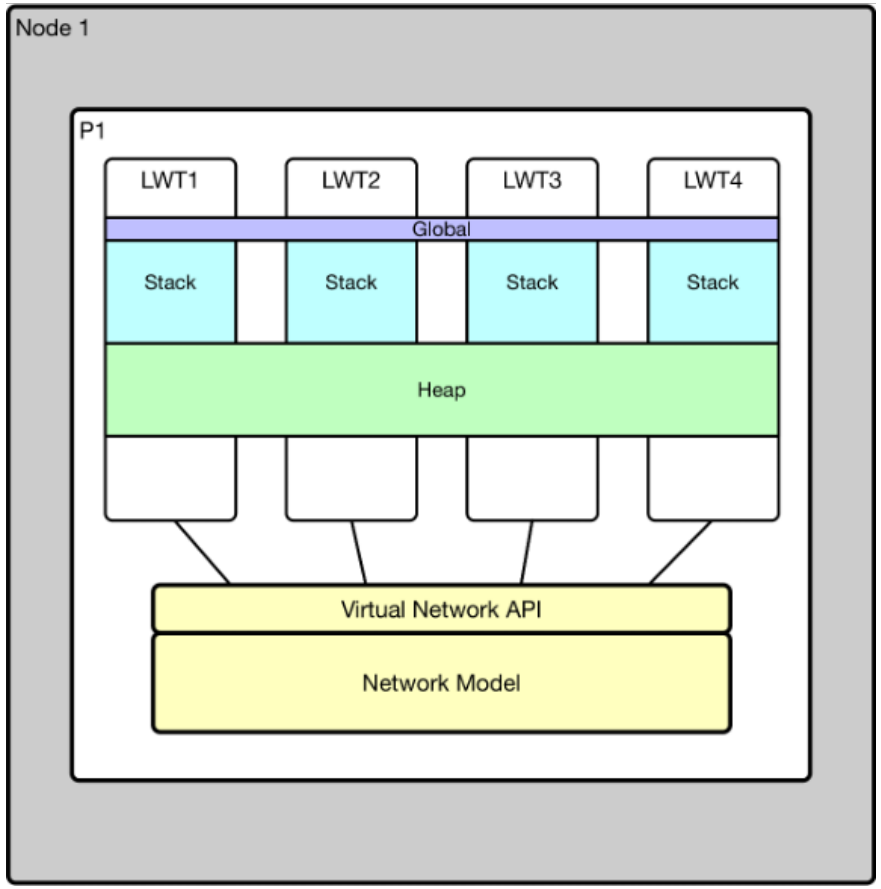
The design philosophy for Mercury is to support and encourage straightforward solutions to workload modeling challenges. Some automation from SST/macro (e.g. source-to-source compiler) is being pulled over, but reducing complexity is a high-level goal.

Mercury Concepts: Pedagogical Target System



A simplified "supercomputer"

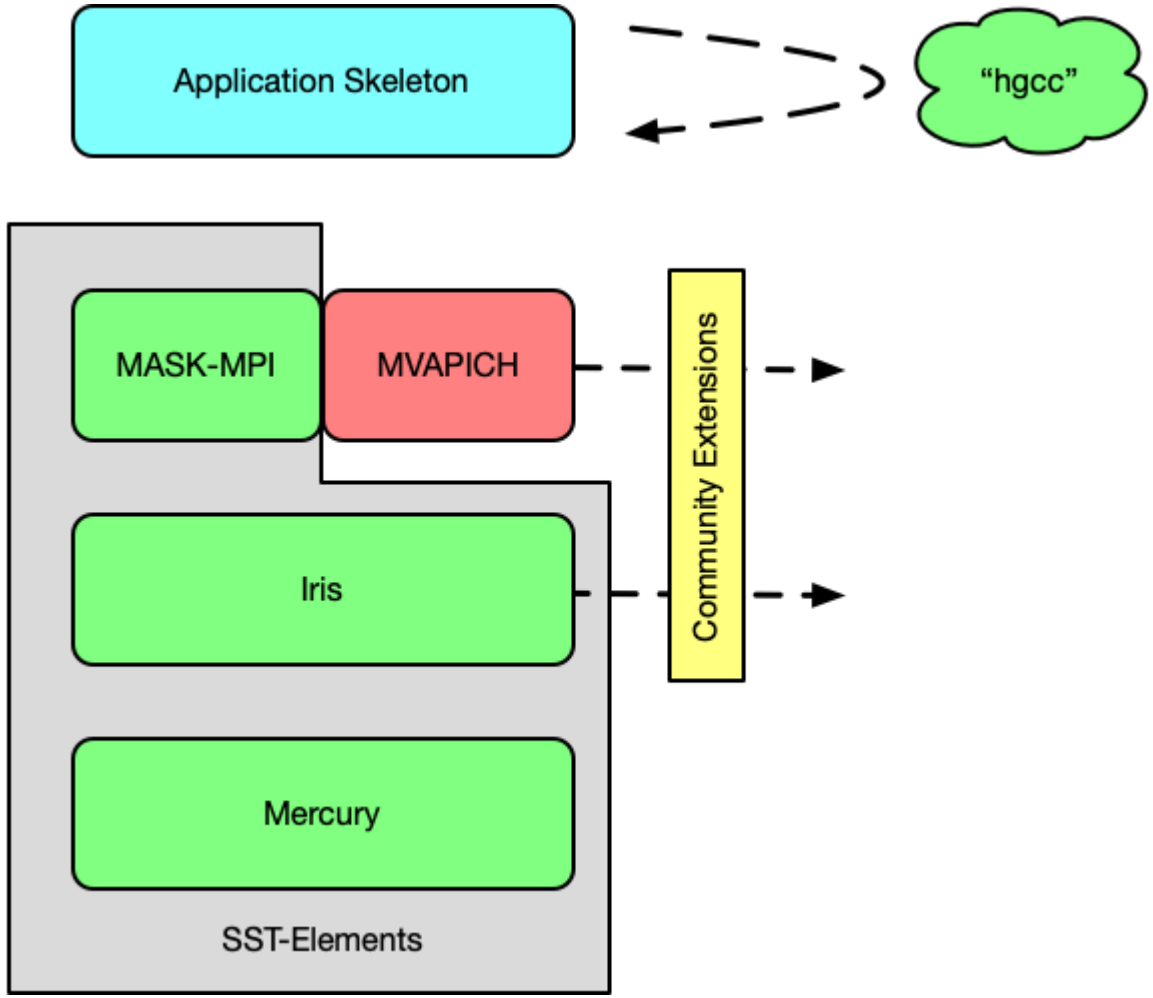
Mercury Concepts: Lightweight Threads



- Simulation: run virtual processes/tasks as lightweight (user space) threads to manage overhead
- Executes relatively unmodified application code, but on host system not simulated devices
- Challenges
  - Single address space means global data is now shared between virtual processes (shared heap is ok)
  - Compute/memory blow up if you try to emulate large system
  - Some host features need to be replaced with implementations appropriate for simulation environment (e.g. threading)
- Compilers can help with some of these problems

This is the fundamental capability that Mercury provides

Mercury Environment: Modular Building Blocks for Workload Models



- Breaks monolithic nature of SST/macro
- **Iris:** communication middleware
  - **SUMI** – low-level messaging interface
  - **Libfabric** provider targeting SUMI
- **MASK-MPI:** port of SST/macro simulator-specific MPI implementation (targets SUMI)
  - Lightweight
  - Broad suppport of MPI features
- **sst-hgcc:** compiler support
  - Privatize shared variables
  - Semi-automated skeletonization
- Capability Demonstration – run full MVAPICH implementation over SUMI using hgcc skeletonization

Mercury Example – Basic Skeletonization

```
$cd exercises/mercury/x2
$cat x2.cc
#define ssthg_app_name x2
#include <mask_mpi.h>
#include <iostream>
#include <mercury/common/skeleton.h>

int main(int argc, char** argv) {

  int rank, size, intervals = 1e2;
  double sum = 0.0;
  double result = 0.0;

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);

  for (int i=0; i <= intervals / 2; ++i) {
    double x = rank/2.0 + i * 1/(1.0*intervals) + 1/(2.0 * intervals);
    sum += x * x * 1/(1.0 * intervals);
  }

  MPI_Reduce(&sum,&result,1,MPI_DOUBLE,MPI_SUM,0,MPI_COMM_WORLD);
  if(rank==0) std::cerr << "Riemann sum is " << result << std::endl;

  MPI_Finalize();
  return 0;
}
```

- Example – compute the Riemann sum for X*X over [0,1]
- Straight MPI code aside from header modifications

How do we configure a simulation?...

Mercury Example – Basic Skeletonization

```
$ cat platform_file_hg_test.py
import sst
from sst.merlin.base import *

platdef = PlatformDefinition("platform_hg_test")
PlatformDefinition.registerPlatformDefinition(platdef)

platdef.addParamSet("node",{
    "verbose"                 : "0",
    "name"                    : "hg.NodeCL",
    "negligible_compute_bytes" : "64B",
    "parallelism"             : "1.0",
    "frequency"               : "2.1GHz",
    "flow_mtu"                : "512",
    "channel_bandwidth"       : "11.2 GB/s",
    "num_channels"            : "4",
})

platdef.addParamSet("nic",{
    "verbose" : "0",
    "mtu" : "4096",
})

platdef.addParamSet("operating_system",{
    "verbose" : "0",
    "name"    : "hg.OperatingSystemCL"
})

platdef.addParamSet("topology",{
    "link_latency" : "20ns",
})

platdef.addClassType("network_interface","sst.merlin.interface.ReorderLinkControl")

platdef.addParamSet("network_interface",{
    "link_bw" : "12 GB/s",
    "input_buf_size" : "64 kB",
    "output_buf_size" : "64 kB"
})

platdef.addParamSet("router",{
    "link_bw" : "12 GB/s",
    "flit_size" : "8B",
    "xbar_bw" : "50 GB/s",
    "input_latency" : "20ns",
    "output_latency" : "20ns",
    "input_buf_size" : "64 kB",
    "output_buf_size" : "64 kB",
    "num_vns" : 1,
    "xbar_arb" : "merlin.xbar_arb_lru",
})
```

```
platdef.addParamSet("operating_system", {
    "ncores" : "24",
    "nsockets" : "4",
    "app1.verbose" : "0",
    "app1.post_rdma_delay" : "88us",
    "app1.post_header_delay" : "0.36us",
    "app1.poll_delay" : "0us",
    "app1.rdma_pin_latency" : "5.43us",
    "app1.rdma_page_delay" : "50.50ns",
    "app1.rdma_page_size" : "4096",
    "app1.max_vshort_msg_size" : "512 B",
    "app1.max_eager_msg_size" : "65536 B",
    "app1.use_put_window" : "true",
    "app1.compute_library_access_width" : "512",
    "app1.compute_library_loop_overhead" : "1.0",
    "app1.smp_optimize" : "false",
})

platdef.addClassType("router","sst.merlin.base.hr_router")
```

- Hg::OperatingSystem does the heavy lifting, loads/runs software stacks

- Typical parameters for Merlin simulation
- "CL" versions of Mercury components support simple ComputeLibrary delay modeling

Mercury Example – Basic Skeletonization

```
$ cat run_x2.py
#!/usr/bin/env python
import sst
from sst.merlin.base import *
from sst.merlin.endpoint import *
from sst.merlin.interface import *
from sst.merlin.topology import *
from sst.hg import *

if __name__ == "__main__":

    PlatformDefinition.loadPlatformFile("platform_file_hg_test")
    PlatformDefinition.setCurrentPlatform("platform_hg_test")
    platform = PlatformDefinition.getCurrentPlatform()

    platform.addParamSet("operating_system", {
        "verbose" : 0,
        "app1.name" : "x2",
        "app1.exe_library_name"  : "x2",
        "app1.verbose" : 0,
        "app1.dependencies" : ["sumi",],
        "app1.libraries"    : ["systemlibrary:SystemLibrary",
                               "computelibrary:ComputeLibrary",
                               "mask_mpi:MpiApi",]

    })

    topo = topoFatTree()
    topo.shape = "4,4:4,4:8"

    ep = HgJob(0,2)

    system = System()
    system.setTopology(topo)
    system.allocateNodes(ep,"linear")

    system.build()
```

```
$ ./build.sh
$ time sst run_x2.py
Riemann sum is 0.345976
Simulation is complete, simulated time: 2.20723 us

real    0m0.891s
user    0m0.409s
sys     0m0.084s
```

What's different from an Ember input?

How to find application code

Libraries that aren't directly called by app

Libraries that are directly called by app

HgJob instead of EmberJob

```
$ grep -m 1 intervals x2.cc
  int rank, size, intervals = 1e8;
$ ./build.sh
$ time sst run_x2.py
Riemann sum is 0.333333
Simulation is complete, simulated time: 2.20723 us

real    0m1.217s
user    0m0.788s
sys     0m0.081s
```

Mercury Example – Basic Skeletonization

```
$ ./build.sh
$ time sst run_x2.py
Riemann sum is 0.345976
Simulation is complete, simulated time: 2.20723 us

real    0m0.891s
user    0m0.409s
sys     0m0.084s
```

Not such an impressive result for 100 intervals

```
$ grep -m 1 intervals x2.cc
  int rank, size, intervals = 1e8;
$ ./build.sh
$ time sst run_x2.py
Riemann sum is 0.333333
Simulation is complete, simulated time: 2.20723 us

real    0m1.217s
user    0m0.788s
sys     0m0.081s
```

More intervals yields better result

Wall clock increase ~50%

Simulated time is unaffected

Exercise: Skeletonize x2.cc

- Skeletonize – reduce computation and memory footprint while maintaining control flow and parameters for network operations
- Important to know:
  - MPI send/recv buffers should be set to ssthg_nullptr;
  - The following header will allow you to call Mercury's sleep(int);

```
$ cd x2-skeleton/
$ head -n 5 x2.cc
#define ssthg_app_name x2
#include <libraries/system/replacements/unistd.h>
#include <mask_mpi.h>
#include <iostream>
#include <mercury/common/skeleton.h>
```

Mercury Example – Skeletonized x2.cc

```
$ cat x2.cc
#define ssthg_app_name x2
#include <libraries/system/replacements/unistd.h>
#include <mask_mpi.h>
#include <iostream>
#include <mercury/common/skeleton.h>

int main(int argc, char** argv) {

  int rank, size, intervals = 1e8;
  //double sum = 0.0;
  //double result = 0.0;
  void *sum = sst_hg_nullptr;
  void *result = sst_hg_nullptr;

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);

//  for (int i=0; i <= intervals / 2; ++i) {
//    double x = rank/2.0 + i * 1/(1.0*intervals) + 1/(2.0
//    sum += x * x * 1/(1.0 * intervals);
//  }

  sleep(1);

  MPI_Reduce(&sum,&result,1,MPI_DOUBLE,MPI_SUM,0,MPI_COMM_
  if(rank==0) std::cerr << "Rieman sum is " << "not comput

  MPI_Finalize();
  return 0;
}
~
```

```
$ grep -m 1 intervals x2.cc
  int rank, size, intervals = 1e8;
$ ./build.sh
$ time sst run_x2.py
Rieman sum is not computed
Simulation is complete, simulated time: 1 s

real    0m0.859s
user    0m0.413s
sys     0m0.083s
```

Now simulated time has changed (we've replaced sleep() with a version that can advance simulation time).

Note, however that wall clock time is comparable to the version with minimal computation.

```
$ cd ../x2
$ grep -m 1 intervals x2.cc
  int rank, size, intervals = 1e2;
$ ./build.sh
$ time sst run_x2.py
Riemann sum is 0.345976
Simulation is complete, simulated time: 2.20723 us

real    0m0.875s
user    0m0.412s
sys     0m0.083s
```

A More Complex Skeleton – halo3d-26

A slightly modified (fixed input parameters) halo3d-26 proxy application is included and compiled in mask-mpi (mask-mpi/skeletons).

```
$ cd ../halo3d-26/
$ sst run_halo.py
halo3d-26 executed successfully
Simulation is complete, simulated time: 158.429 ms
```

Halo3d is a good entry point to explore more complicated skeleton applications -- try to run it via the provided input deck.

Advanced Topic – Compiling with sst-hgcc

sst-hgcc is a separate repo that provides experimental compiler support for building Mercury-compatible skeletons

Two modes:

- Wrapper mode: alleviates tedious and error prone determination of compiler options (functional prototype)
- LLVM-based source-to-source for shared variable privatization (functional prototype) and pragma based skeletonization assistance (upcoming)

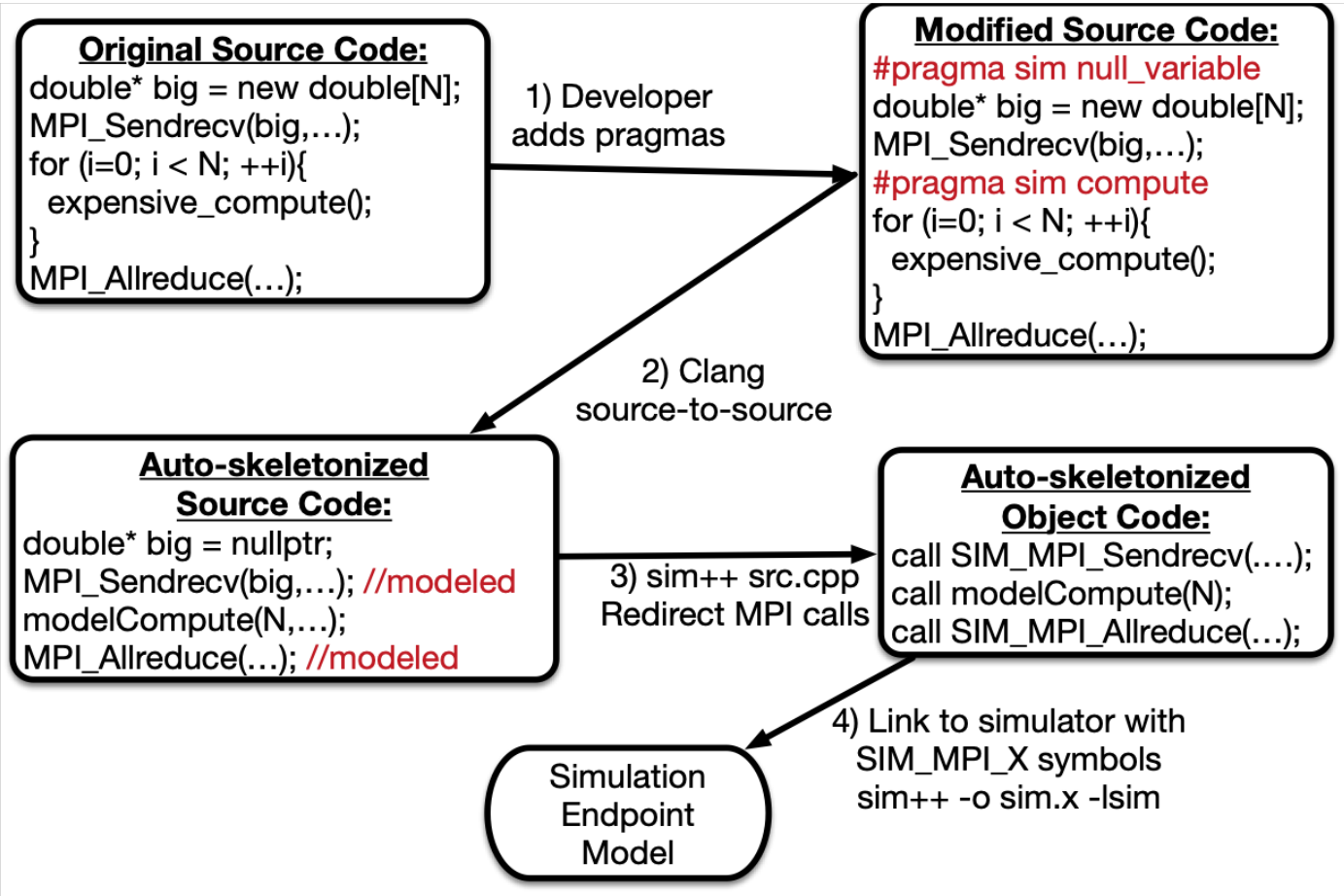
Current implementation is fragile and difficult to maintain and build

```
$ cd ../x2-hgcc
$ cat build.sh
#!/bin/bash
hg++ -c x2.cc
hg++ x2.o -o libx2.so
$ ./build.sh
$
```

Pro tip: user SST_HG_VERBOSE with hg++ to get suggested compiler options

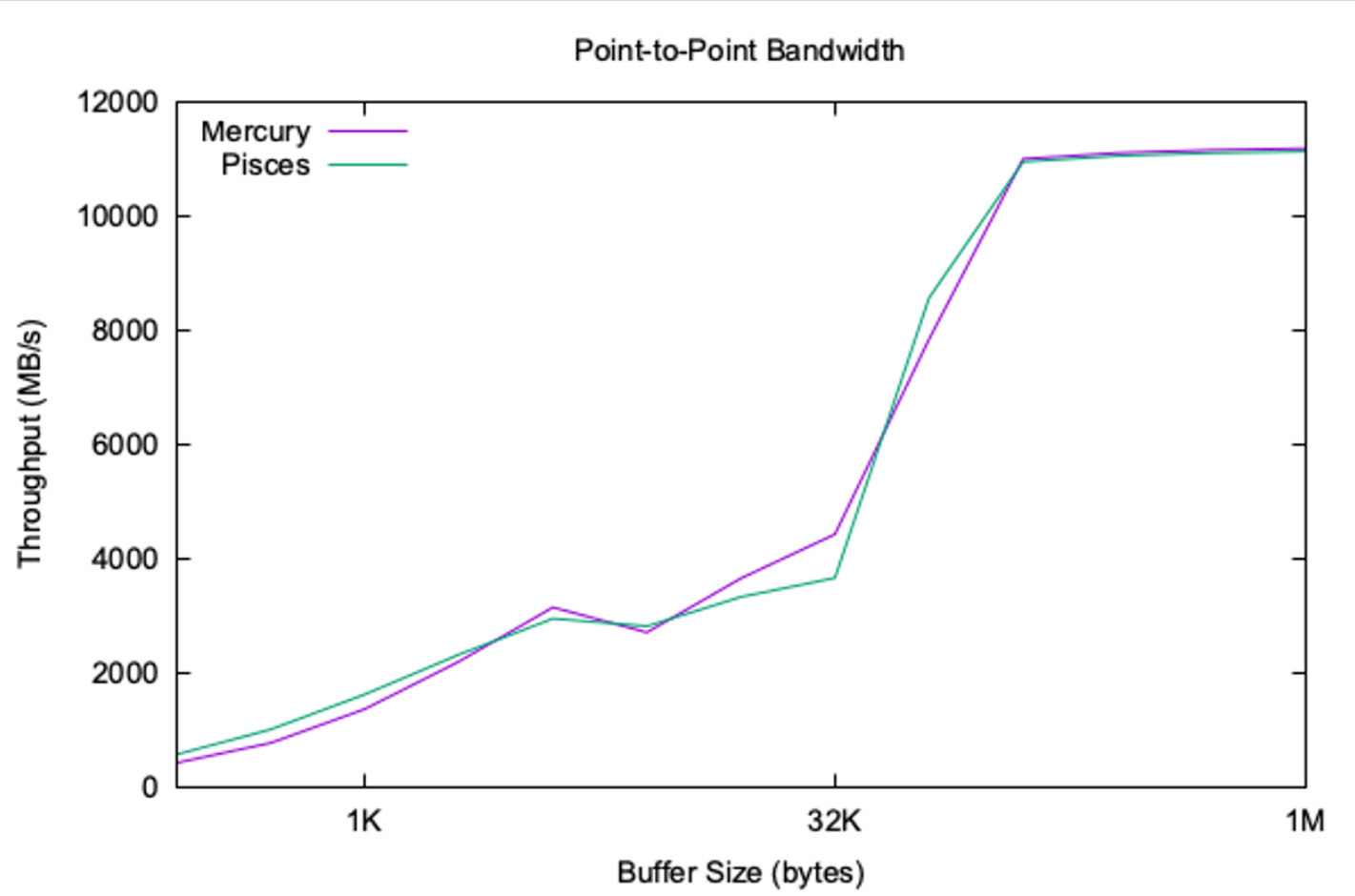
```
$ cat build-verbose.sh
#!/bin/bash
SST_HG_VERBOSE=1 hg++ -c x2.cc
SST_HG_VERBOSE=1 hg++ x2.o -o libx2.so
$ ./build-verbose.sh
==============================
/usr/bin/g++ -include cstdint -include /Users/jpkenny/install/sst-elements-hg-fixpartition2-mpi/include/sst/elements/mercury/common/skeleton.h -DSST_HG_EXTERNAL_SKELETON
-I/Users/jpkenny/install/sst-elements-hg-fixpartition2-mpi/include/sst/elements/mercury/libraries/replacements/mpi -I/Users/jpkenny/install/sst-elements-hg-fixpartition2-
mpi/include/sst/elements/mercury/libraries/replacements -I/Users/jpkenny/install/sst-elements-hg-fixpartition2-mpi/include/sst/elements/sumi -I/Users/jpkenny/install/sst-
core-devel-mpi/include -I/Users/jpkenny/install/sst-elements-hg-fixpartition2-mpi/include/sst/elements -I/Users/jpkenny/install/sst-elements-hg-fixpartition2-mpi/include/
sst/elements/mercury -I/Users/jpkenny/install/sst-elements-hg-fixpartition2-mpi/include/sst/elements/mercury/common -I/Users/jpkenny/install/sst-elements-hg-fixpartition2
-mpi/include/sst/elements/mercury/software/libraries -I/Users/jpkenny/install/sst-elements-hg-fixpartition2-mpi/include/sst/elements/sumi -I/Users/jpkenny/install/sst-ele
ments-hg-fixpartition2-mpi/include/sst/elements/mask-mpi -fPIC -ffp-contract=off -std=c++17 -c x2.cc -o x2.o
==============================
/usr/bin/g++ -Wl,-rpath,/Users/jpkenny/install/sst-hgcc-noclang/lib -L/Users/jpkenny/install/sst-elements-hg-fixpartition2-mpi/lib/sst-elements-library -bundle -undefined
 dynamic_lookup -fPIC -ffp-contract=off -std=c++17 x2.o -o libx2.so
```

Advanced Topic – Source-to-source Compiler



See: Wilke, Jeremiah J., et al. "Compiler-assisted source-to-source skeletonization of application models for system simulation." *High Performance Computing: 33rd International Conference, ISC High Performance 2018, Frankfurt, Germany, June 24-28, 2018, Proceedings 33*. Springer International Publishing, 2018.

Mercury Results – Early Comparison of SST/macro and Mercury/Merlin



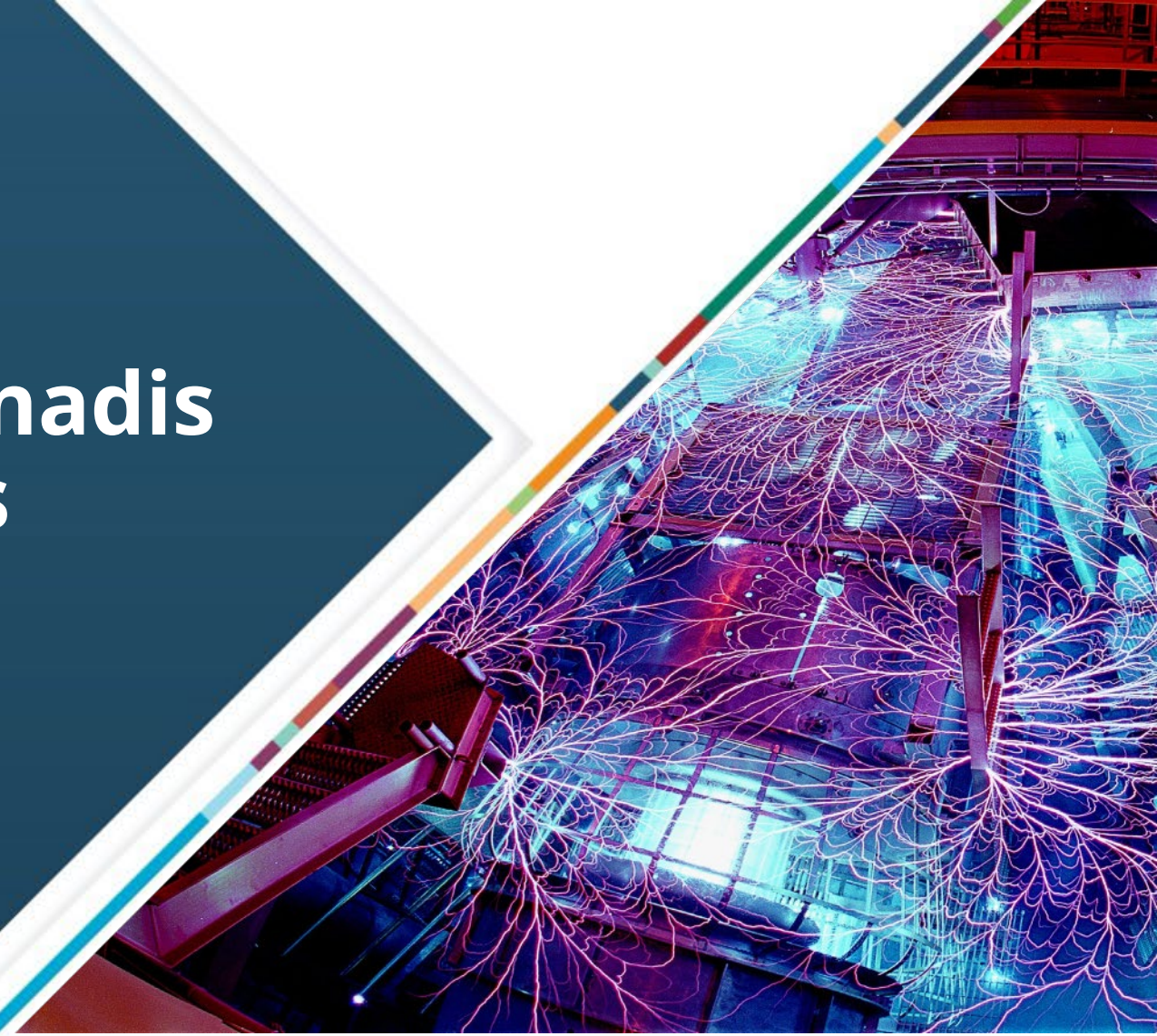
- Pisces is the highest fidelity SST/macro network model
- Very good agreement between completely independent network models

Sandia National Laboratories

Exceptional service in the national interest

Introduction to the vanadis and balar Components

SST Tutorial – IPDPS 2025



Clay Hughes, Sandia National Laboratories

SST Development Team, Sandia National Laboratories

Architecture Accelerator Lab, Purdue University

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and 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.

U.S. DEPARTMENT OF ENERGY

NNSA

Welcome!

Part 1: Introdcution to SST	*2:00 – 2:15*
Part 2: Introduction to vanadis	*2:15 - 2:45*
Part 3: Introduction to balar	*2:45 – 3:15*
Part 3: Introduction to rev	*3:15 - 4:00*
Break / Finish	*4:00 - 4:30*

Introduction to *vanadis*

Processor Models and SST

Ariel – PIN instruction trap and forward for memory requests

- Utilize full range of X86-64 instructions (including complex vectorization)
- Easy to use with binaries from the host, including large/complex binaries, dynamically linked
- Forwards requests to the memory subsystem and maintains back-pressure in load/store queues to preserve approximate timing

Juno – simple in-order instruction using a (very) small but extensible ISA

- Proof of concept for external processor components
- Very fast to execute, but custom/simple ISA is assembly-based and cannot be used with high-level language/compiler

Miranda – state machine based core model for memory subsystem

- Allows complex memory patters to be analyzed with very lightweight core models

Spike – external RISC-V model developed by Tactical Computing

GEM5 – previously integrated; not tested and does not support SST parallelization

Requirements for a Flexible Core Model

Want a processor core component that can utilize full range of SST capabilities

- Ability to parallelize simulation of designs which may have multi/many-core processors with >1,000 cores and any hardware threads (using SST's MPI and thread based PDES engine)
- Ability to run for long simulation periods (billions of instructions)
- Works natively with SST memory subsystem and accelerator models
- Including time/periodic and event based output of statistics
- Produces native SST statistics for feedback modeling (e.g. energy, DVFS, resilience etc)
- Ability to extend/replace ISA using SST modules/sub-components

Not strong requirements but research interest in:

- RISC-like core pipelines (use for modeling full server-class cores through to small accelerator like processors)
- RISC-V and Arm ISA support
- Possibility to replace sections of the processor model using SST sub-components to make models more extensible
- Custom instructions and functional units

vanadis Pipeline Model

Initial research concept for a ISA-agnostic RISC pipeline model for SST

- Provides operating system emulation mode (does not need to boot a kernel *etc.*)
- Decoders are full SST sub-components so can be extended and swapped out
- On a per-hardware thread or per-core basis
- Does not require ISA to be compiled in (user can provide these later using sub-components)

Pipeline is out-of-order and parameterized to permit hardware exploration

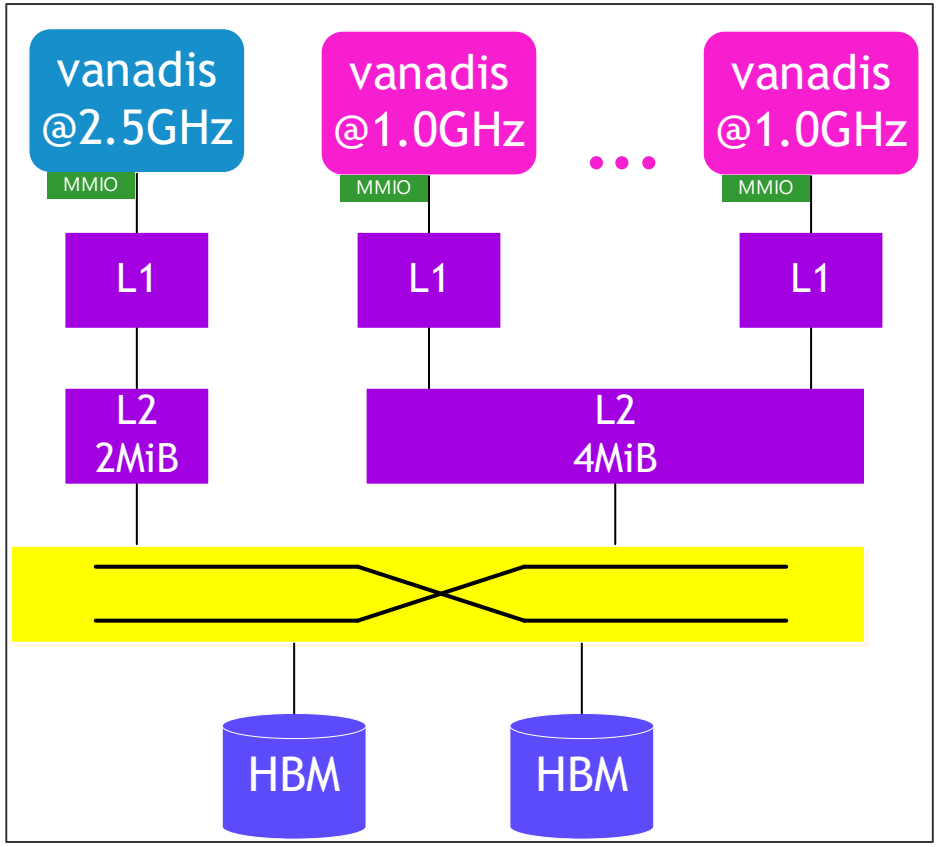
- Execute-at-execute pipeline model
- Pipeline instructions are generic and agnostic to the specific ISA being used (so easy to add ISAs and use existing instructions that are regularly tested)
- Functional units are fully extensible/swappable and individually parameterizable
- Ability to add (custom) instructions into the pipeline and out-of-order engines comply with providing register dependencies and correct retirement

Example Configurations

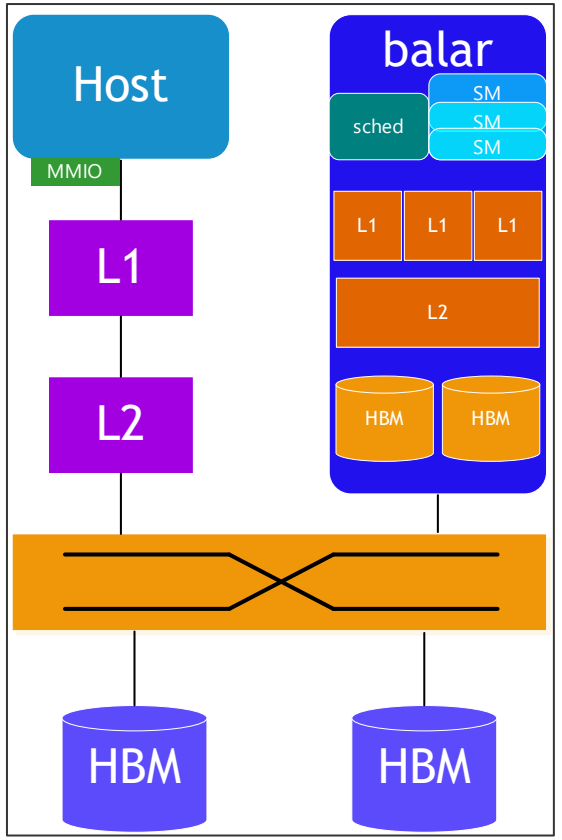
Flexibility and parameterization of vanadis allows it to be used in many different design scenarios

Integration with SST allows significant degree of flexibility

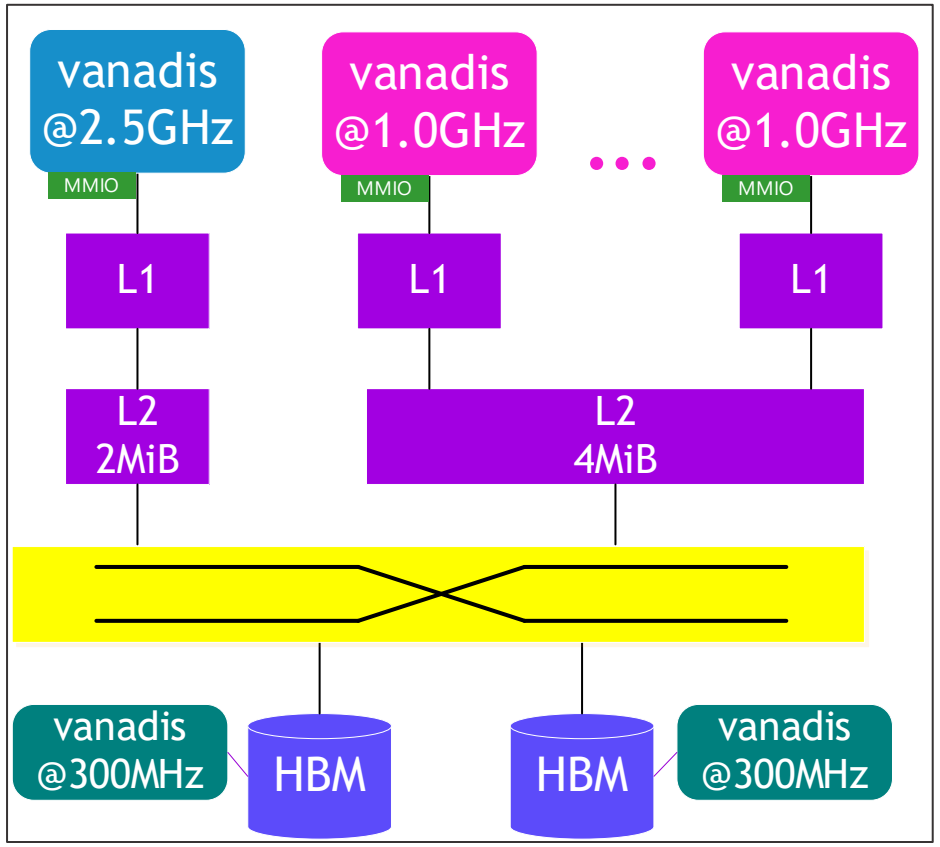
Potential for different cores to have different ISAs (e.g. PIM use case



Mix-core Node



Accelerated Node



PIM Node

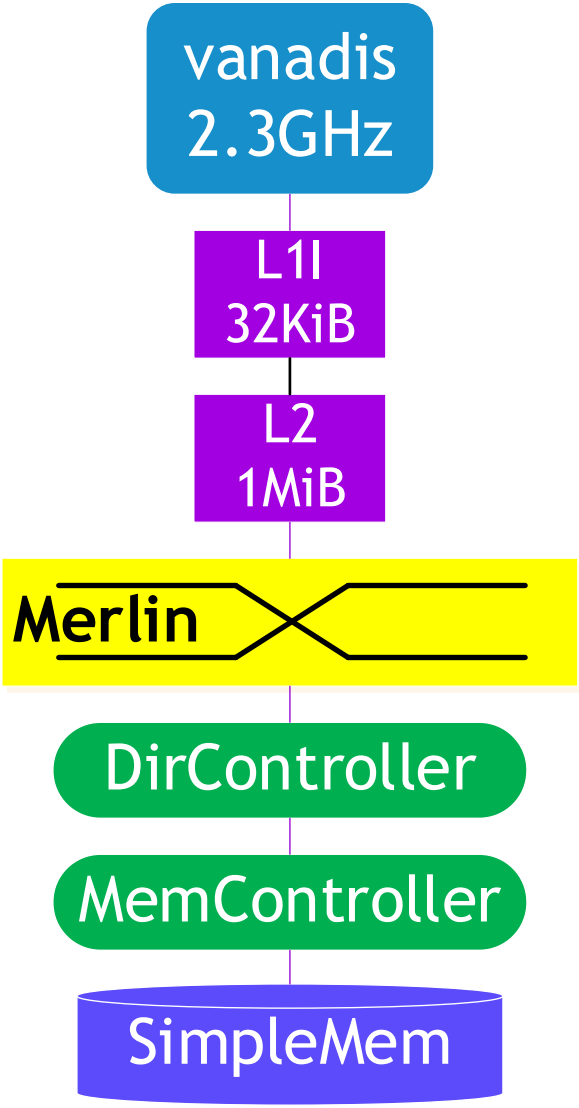
Single Core Example

Required elements

- Vanadis
- memHierarchy
- Merlin

Example located in:
../vanadis/tests/basic_vanadis.py

Test harness can instantiate multiple CPUs but this example assumes a single CPU



Global/System Parameters

```
# Tell SST what statistics handling we want
sst.setStatisticLoadLevel(4)
sst.setStatisticOutput("sst.statOutputConsole")

full_exe_name = os.getenv("VANADIS_EXE", "./small/" + testDir + "/" + exe +  "/" + isa + "/" + exe )
exe_name= full_exe_name.split("/")[-1]

verbosity = int(os.getenv("VANADIS_VERBOSE", 0))
os_verbosity = os.getenv("VANADIS_OS_VERBOSE", verbosity)
pipe_trace_file = os.getenv("VANADIS_PIPE_TRACE", "")
lsq_ld_entries = os.getenv("VANADIS_LSQ_LD_ENTRIES", 16)
lsq_st_entries = os.getenv("VANADIS_LSQ_ST_ENTRIES", 8)

rob_slots = os.getenv("VANADIS_ROB_SLOTS", 64)
retires_per_cycle = os.getenv("VANADIS_RETIRES_PER_CYCLE", 4)
issues_per_cycle = os.getenv("VANADIS_ISSUES_PER_CYCLE", 4)
decodes_per_cycle = os.getenv("VANADIS_DECODES_PER_CYCLE", 4)

integer_arith_cycles = int(os.getenv("VANADIS_INTEGER_ARITH_CYCLES", 2))
integer_arith_units = int(os.getenv("VANADIS_INTEGER_ARITH_UNITS", 2))
fp_arith_cycles = int(os.getenv("VANADIS_FP_ARITH_CYCLES", 8))
fp_arith_units = int(os.getenv("VANADIS_FP_ARITH_UNITS", 2))
branch_arith_cycles = int(os.getenv("VANADIS_BRANCH_ARITH_CYCLES", 2))

cpu_clock = os.getenv("VANADIS_CPU_CLOCK", "2.3GHz")

numCpus = int(os.getenv("VANADIS_NUM_CORES", 1))
numThreads = int(os.getenv("VANADIS_NUM_HW_THREADS", 1))

vanadis_cpu_type = "vanadis."
vanadis_cpu_type += os.getenv("VANADIS_CPU_ELEMENT_NAME","dbg_VanadisCPU")
```

Core Parameters

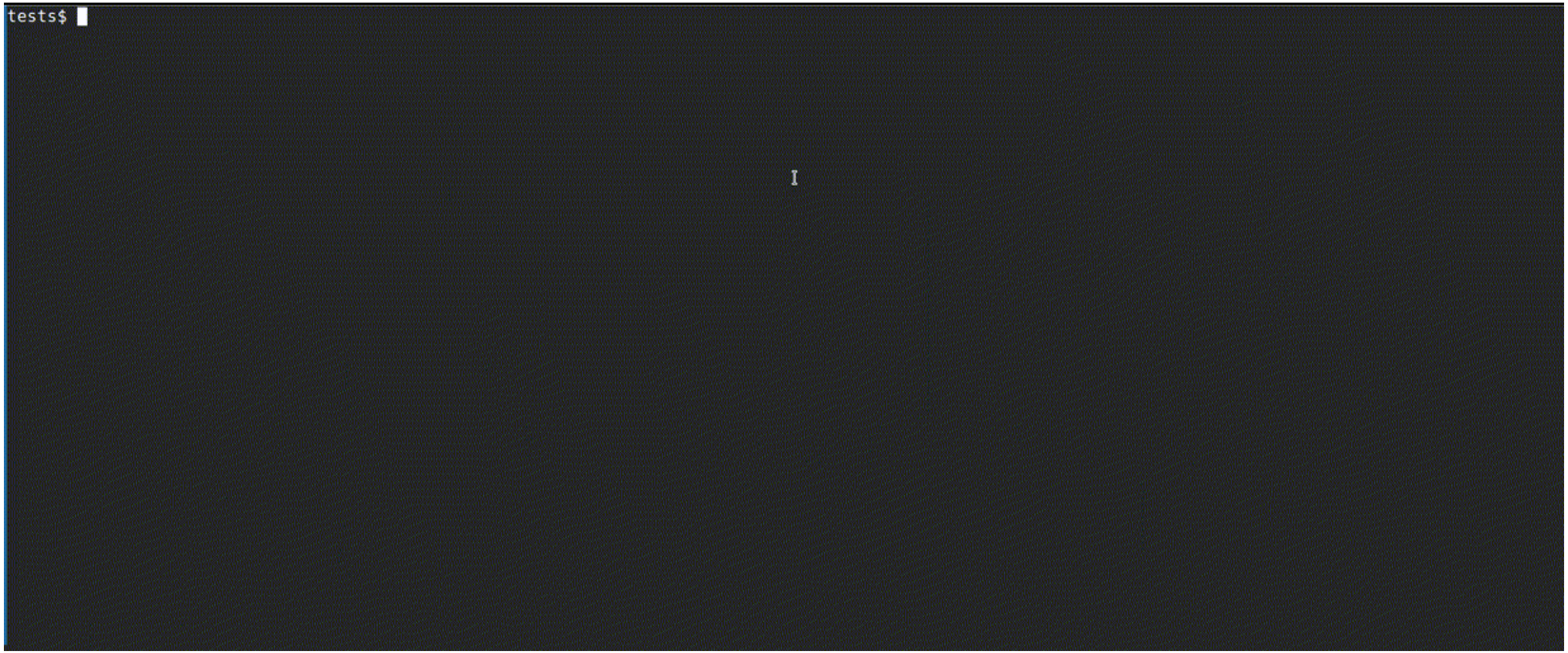
```
cpuParams = {
    "clock" : cpu_clock,
    "verbose" : verbosity,
    "hardware_threads": numThreads,
    "physical_fp_registers" : 168 * numThreads,
    "physical_integer_registers" : 180 * numThreads,
    "integer_arith_cycles" : integer_arith_cycles,
    "integer_arith_units" : integer_arith_units,
    "fp_arith_cycles" : fp_arith_cycles,
    "fp_arith_units" : fp_arith_units,
    "branch_unit_cycles" : branch_arith_cycles,
    "print_int_reg" : False,
    "print_fp_reg" : False,
    "pipeline_trace_file" : pipe_trace_file,
    "reorder_slots" : rob_slots,
    "decodes_per_cycle" : decodes_per_cycle,
    "issues_per_cycle" :  issues_per_cycle,
    "retires_per_cycle" : retires_per_cycle,
    "pause_when_retire_address" : os.getenv("VANADIS_HALT_AT_ADDRESS", 0),
    "start_verbose_when_issue_address": dbgAddr,
    "stop_verbose_when_retire_address": stopDbg,
    "print_rob" : False,
    "checkpointDir" : checkpointDir,
    "checkpoint" : checkpoint
}
```

Cache Parameters

```
l1dcacheParams = {
    "access_latency_cycles" : "2",
    "cache_frequency" : cpu_clock,
    "replacement_policy" : "lru",
    "coherence_protocol" : protocol,
    "associativity" : "8",
    "cache_line_size" : "64",
    "cache_size" : "32 KB",
    "L1" : "1",
    "debug" : mh_debug,
    "debug_level" : mh_debug_level,
}

l1icacheParams = {
    "access_latency_cycles" : "2",
    "cache_frequency" : cpu_clock,
    "replacement_policy" : "lru",
    "coherence_protocol" : protocol,
    "associativity" : "8",
    "cache_line_size" : "64",
    "cache_size" : "32 KB",
    "prefetcher" : "cassini.NextBlockPrefetcher",
    "prefetcher.reach" : 1,
    "L1" : "1",
    "debug" : mh_debug,
    "debug_level" : mh_debug_level,
}
```

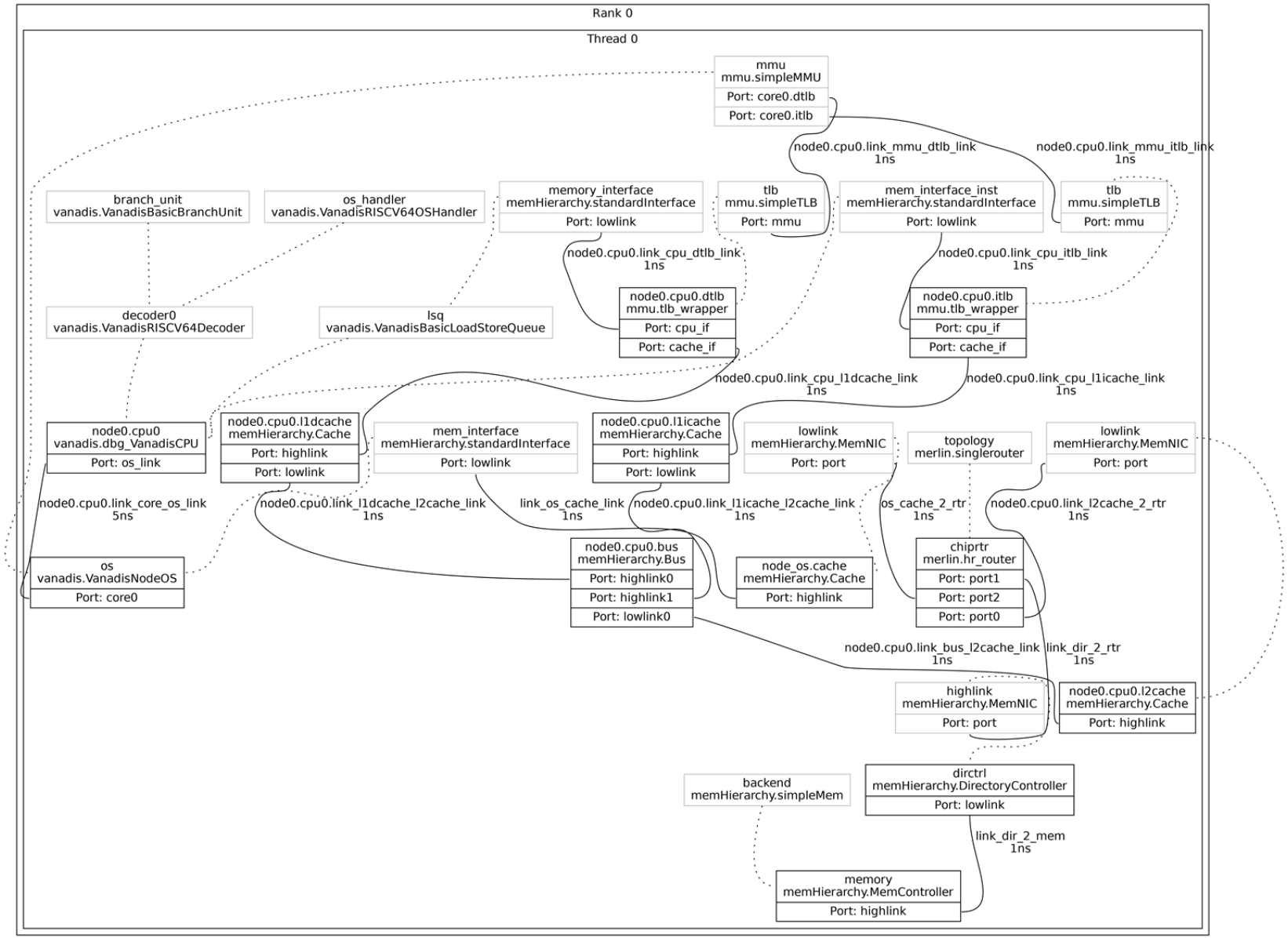
```
l2cacheParams = {
    "access_latency_cycles" : "14",
    "cache_frequency" : cpu_clock,
    "replacement_policy" : "lru",
    "coherence_protocol" : protocol,
    "associativity" : "16",
    "cache_line_size" : "64",
    "cache_size" : "1MB",
    "mshr_latency_cycles": 3,
    "debug" : mh_debug,
    "debug_level" : mh_debug_level,
}
```

Example – *basic_vanadis.py*



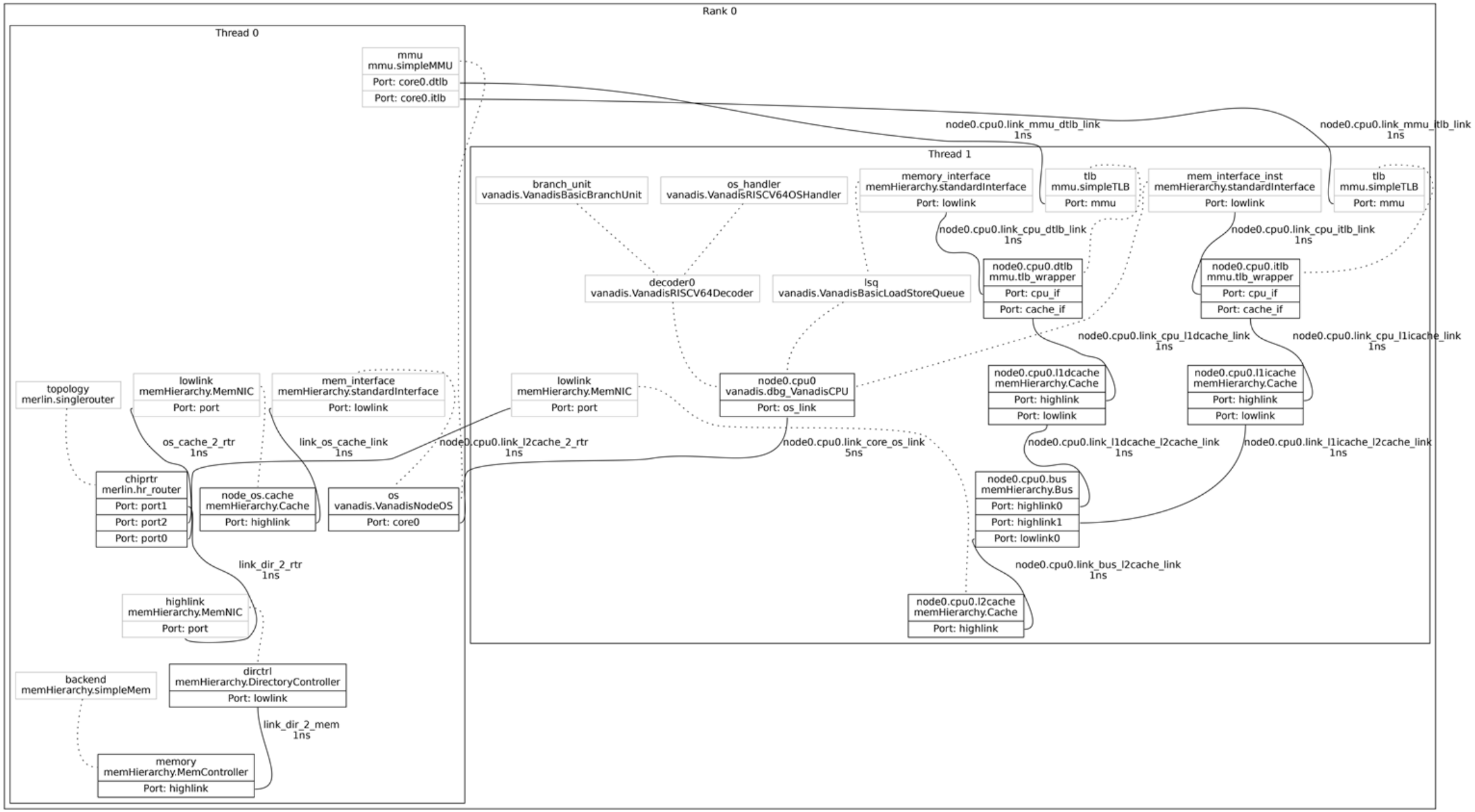
SST Configuration Graph – 1R1T

```
sst --output-dot=<> --dot-verbosity=10
```



SST Configuration Graph – 1R2T

```
sst --output-dot=<> --dot-verbosity=10
```



Accelerators and RoCC

Enable support for tightly-coupled accelerators

Leverages the Rocket Custom Coproeccoer (RoCC) interface

- Takes advantage of the custom instruction encoding space within the RISC-V ISA while standardizing the control signals used by the accelerator
- Enables early-stage software development targeting the accelerator
  - An essential capability since software support is often a major blocker in the deployment of systems with custom accelerators

Currently being tested with integration of analog Matrix Vector Multiplication (MVM) components (golem)

- Modified LLVM compiler, initial software stack and application development has begun
- https://github.com/sstsimulator/sst-elements/tree/master/src/sst/elements/golem

Introduction to *balar*

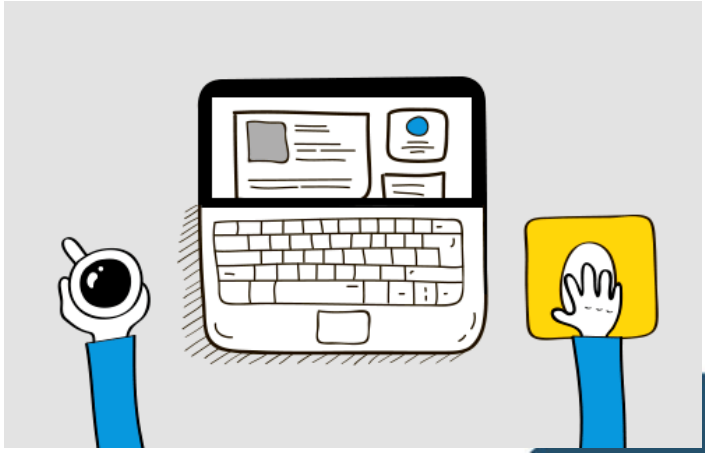
SST GPGPU Modeling Capability

Long history of GPGPU simulators, mostly from academia

- GPGPUSim [Bakhoda, 2009]
- Gem5+GPU [Power, 2014]
- MacSim/Ocelot [Hyesoon, 2012], [Kerr, 2009]
- Multi2Sim v5.0 [Gong, 2017]
- NVArchSim

After a literature review and calls with the developers of GPGPUSim and Multi2Sim, the SST team chose GPGPUSim as the integration target

- More amendable to SST integration
  - Software designed around well-defined objects
- More tractable for SST workflows



GPGPUSim Integration

GPGPUSim

- Runs *host* code directly
- Runs *device* code on a functional simulator
- Interfaces with programs through a custom library, re
- Simulation split into functional and timing

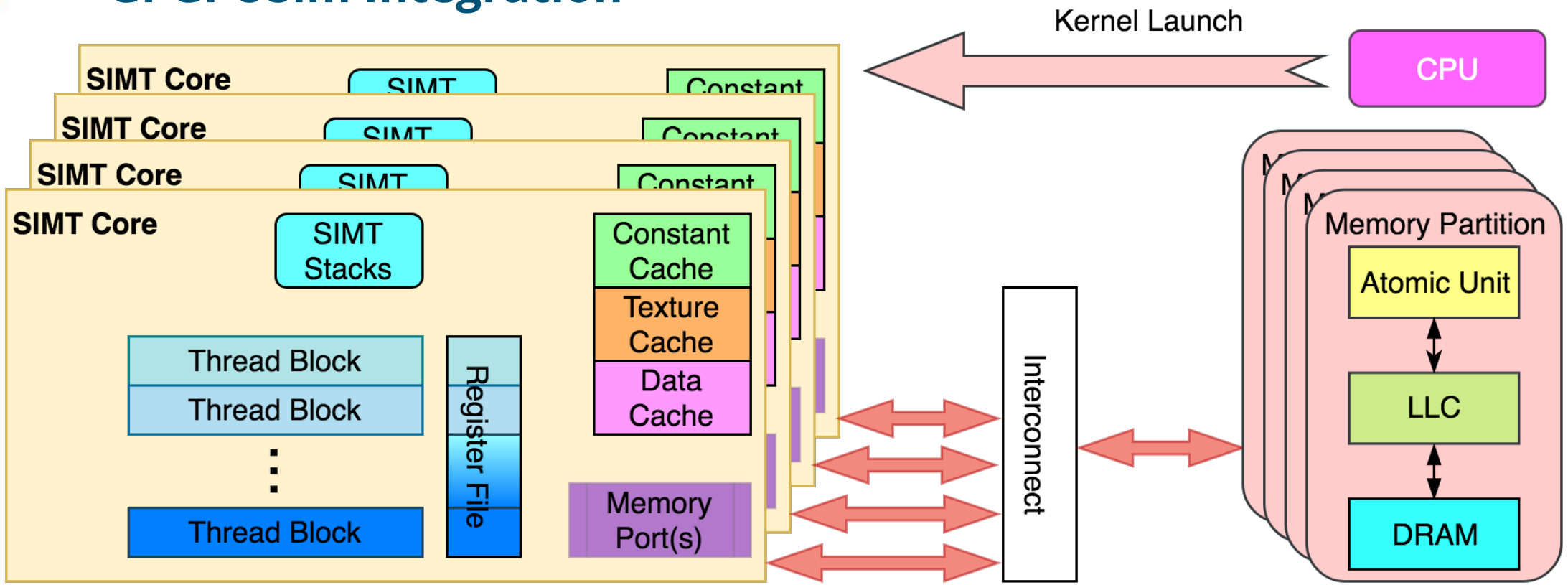
Functional

- CTA
  - Pool of threads with access to shared memory
- Thread
  - PC + registers + allocated memory
- Instruction
  - 1:1 mapping with opcode and implementation function; simulated at issue

Timing

- SIMT Core
  - Roughly equivalent to Fermi's SM
- Cache
  - L1 → Per-core, write-through/no-allocate
  - L2 → Device, write-back/no-allocate
- Interconnect
  - Intersim (Booksim)
- Backing store
  - GDDR3/GDDR5

GPGPUSim Integration



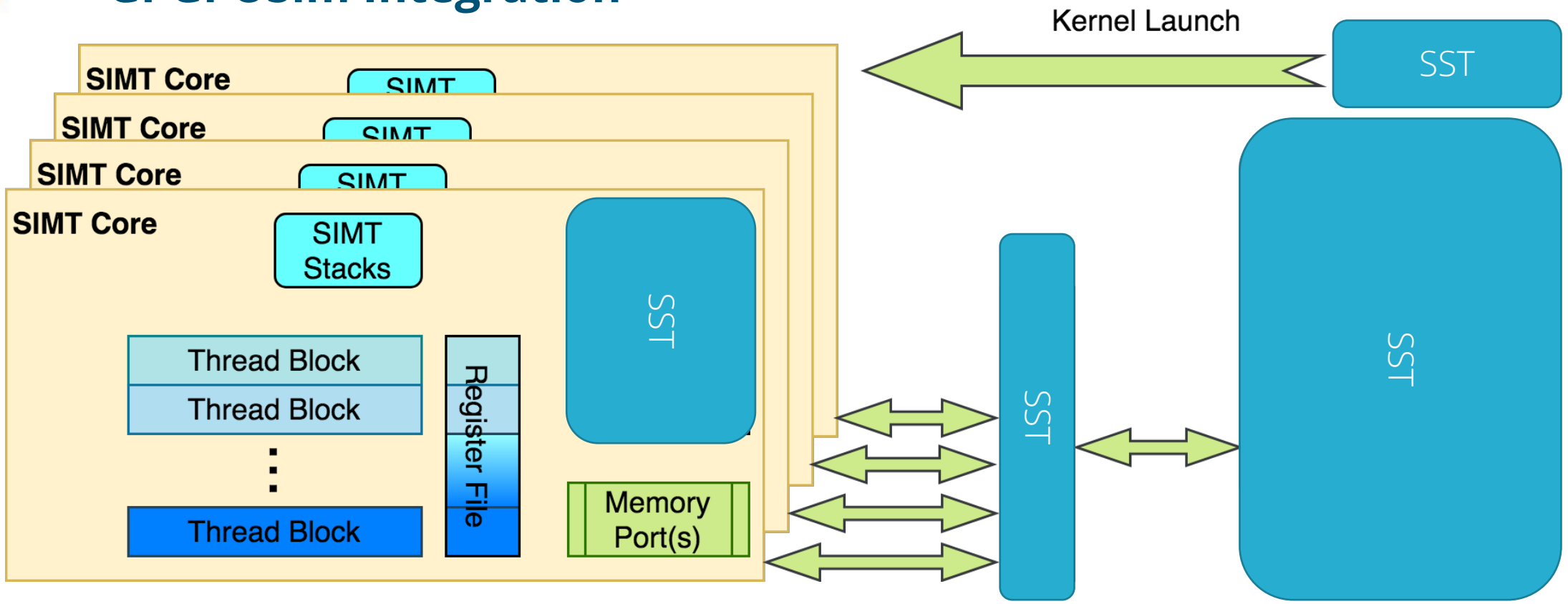
Functional

- SIMT units track default functional model from GPGPUSim

Timing

- SIMT units maintain execution timing; Booksim is used for the interconnect; and simple timing models are used for the memory partition

GPGPUSim Integration



Plan to keep the SIMT units and replace everything else

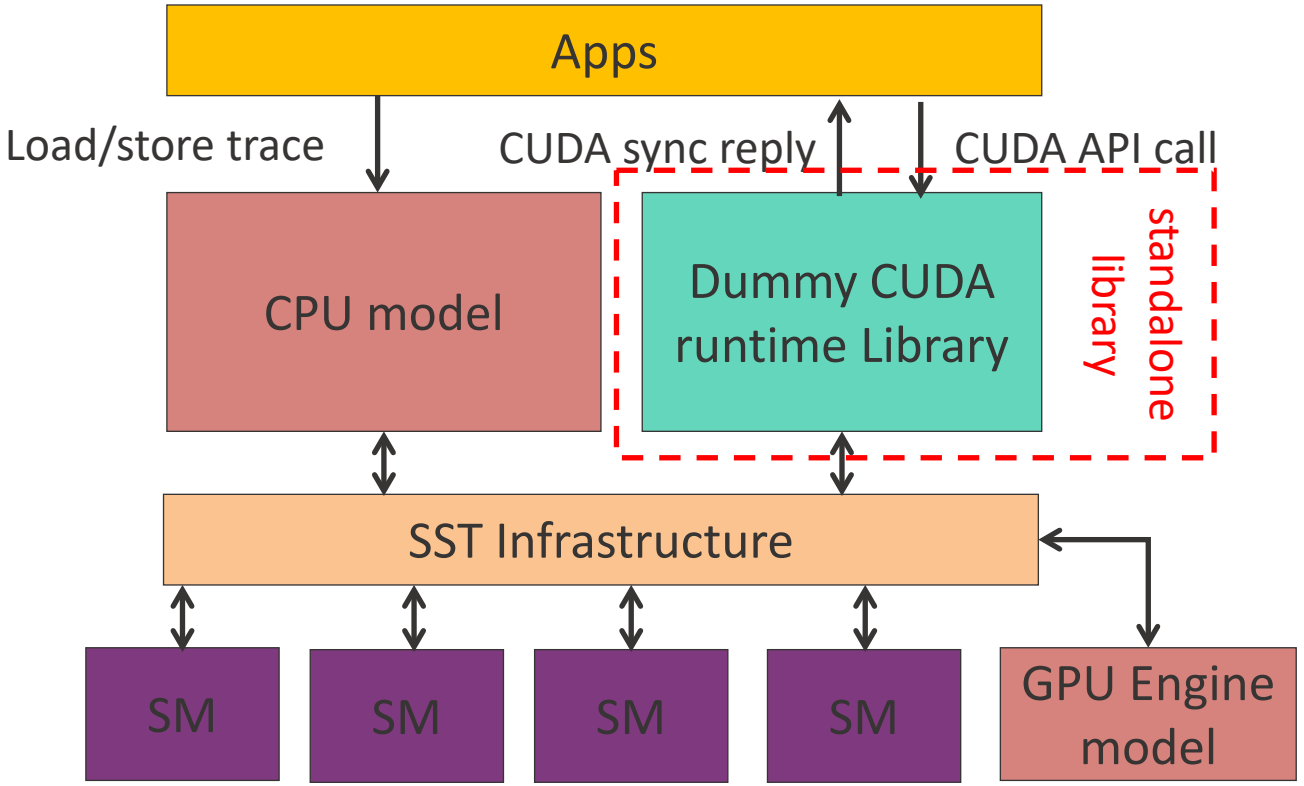
- CPU → SST execution component (Vanadis, Ariel, etc.)
- Interconnect → SST networking component (Shogun, Merlin, Kingsley, etc.)
- Memory Partition & Caches→ SST memHierarchy component for caches and various other backends for the backing store (DRAMSim, SimpleMem, Cramsim, etc.)

CUDA API interception model

Standalone dummy CUDA lib which is independent of CPU model

Applications link dynamically with lib (shared object)

Dummy lib interacts with GPU engine for kernel launch and CUDA memory copies

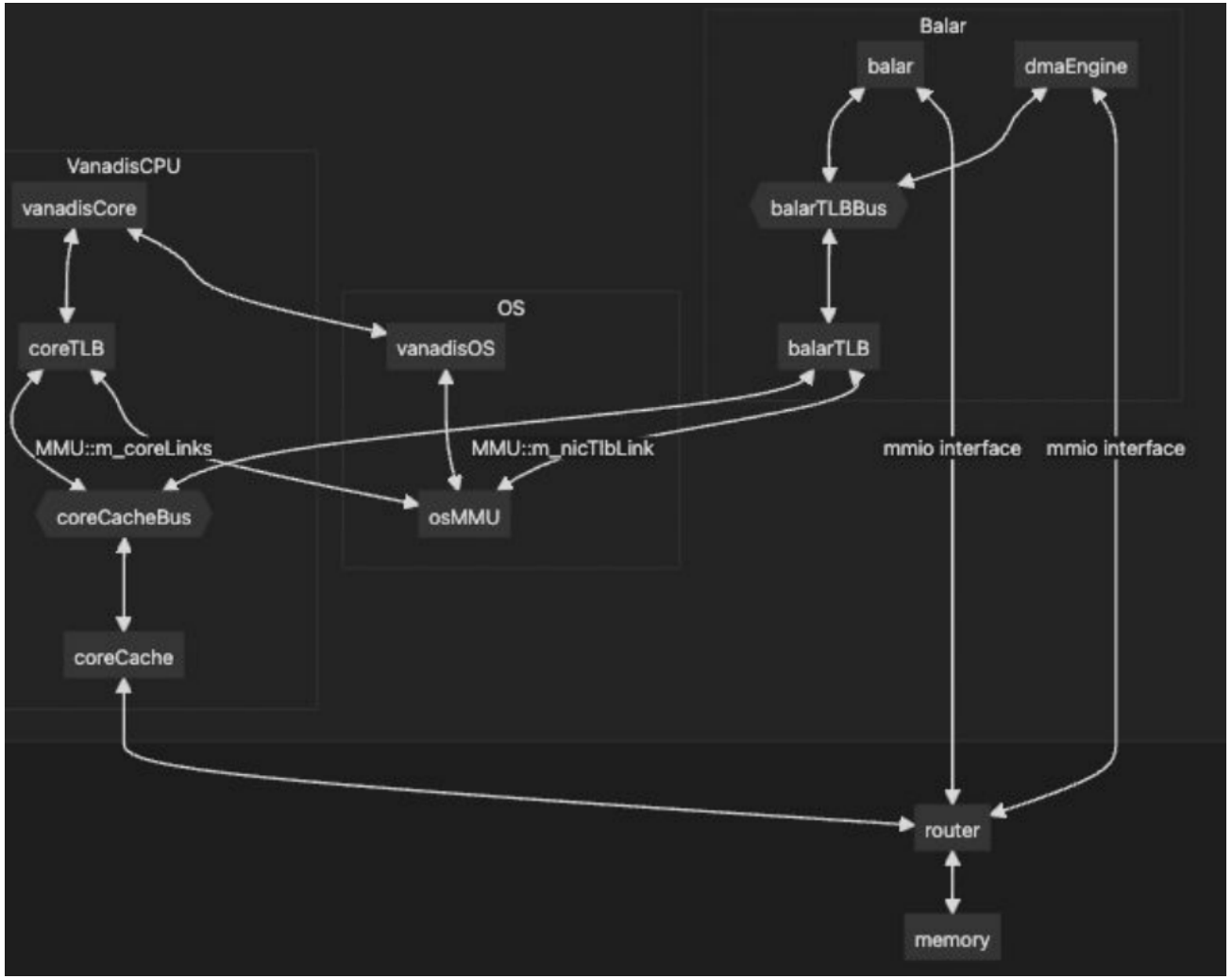


API Calls Forwarded to GPU Model	
`__cudaRegisterFatBinary`	`cudaFree`
`__cudaRegisterFunction`	`cudaLaunch`
`cudaMalloc`	`cudaGetLastError`
`cudaMemcpy`	`cudaRegisterVar`
`cudaConfigureCall`	`cudaSetupArgument`
`cudaOccupancyMaxActiveBlocksPerMultiprocessorWithFlags`	

balar Integrated With *vanadis* via MMAP

When coupled with vanadis, RVA23+CUDA binaries can be run

- Add balar as a device in VanadisOS with file descriptor being -2000 at 0x8010000
- balar and the dmaEngine have both mem_iface (for data packets) and mmio_iface (for command packets)



Building SST with GPU Support

Requires LLVM and RISC-V gcc toolchain

- https://github.com/llvm/llvm-project.git
- https://github.com/riscv-collab/riscv-gnu-toolchain.git

Requires GPGPU-Sim

- https://github.com/accel-sim/gpgpu-sim_distribution.git

Requires CUDA (tested with 10 and 11)

Build instructions in ../sst/elements/balar/README.md

Running an Example – Hello World

```
# ============================================================
# Begin configuring Balar and Vanadis
# ============================================================
# Overall configuration graph
# Balar side:
#   Balar memory interface
#       BalarMMIO <--mem_iface--> BalarTLBBus <--mem_iface--> dmaEngine
#                                       |
#       BalarTLB <--------------------|
#   Balar mmio interface
#       BalarMMIO <--mmio_iface--> memory
#       dmaEngine <--mmio_iface--> memory
#
# Vanadis side:
#   VanadisOS side:
#       VanadisCore <----> VanadisOS <----> mmu
#       iTLB <----> mmu
#       dTLB <----> mmu
#   Vanadis Cache side:
#       iTLB <----> L1Icache
#       dTLB <----> coreCacheBus <----> BalarTLB
#                           |
#       L1Dcache <-------|
#       L1caches <----> L2cache <----> memory
```

Python script shows how components are connected

Running an Example – Hello World

```
#For V100, we have 4 HBM stack, each with 8 channels, so total we have 32 memory parts
[GPU]
clock: 1447MHz
gpu_cores: 80
gpu_l2_parts: 32
gpu_l2_capacity: 192KiB
#we assume PCIE with 16GB/Sec. Each cycle, we transfer a 4KB page, thus latency = 4/(16*1024*1024) = 23840ps
gpu_cpu_latency: 23840ps
#this is not used for now
gpu_cpu_bandwidth: 16GB/s

[GPUMemory]
clock: 877MHz
#network_bw per mem_part. this should be equal to the actual mem_bw. So, total mem BW = gpu_mem_parts*network_bw = 1 TB/S (in Volta V100, it is 900 GB/Sec, so very close)
network_bw: 32GB/s
#this is the total capacity of all gpu_mem_parts. This should be in MiB.
capacity: 16384MiB
memControllers: 2
hbmStacks: 4
hbmChan: 4
hbmRows: 16384

[GPUNetwork]
#1200MHz time period plus slack for xbar latency
frequency: 1200MHz
buffer_depth: 128
input_ports: 3
output_ports: 3

latency: 750ps
#total BW
bandwidth: 4800GB/s
#BW per xbar link
linkbandwidth: 37.5GB/s
flit_size: 40B
```

GPU configuration passed in as an option to the script

Running an Example – Hello World

```
../balar/tests/testBalar-vanadis.py
VANADIS_EXE=./vanadisLLVMRISCV/vecadd VANADIS_ISA=RISCV64
BALAR_CUDA_EXE_PATH=./vanadisLLVMRISCV/vecadd sst testBalar-vanadis.py --
model-options='-c gpu-v100-mem.cfg'
```

- This runs the vector add kernel from the NVIDIA examples
- The source is compiled as riscv64 target; LLVM needed for PTX but uses libraries from RISC-V gcc toolchain
- Custom libcudart used to intercept CUDA APIs

Running an Example – Hello World

Too long to run in real time (~6min)

Stats file contains 22362 unique statistics

```
balar:mmio_iface:memlink.packet_latency : Accumulator : SimTime = 289825490; Rank = 0; Sum.u64 = 0; SumSQ.u64 = 0; Count.u64 = 0; Min.u64 = 0; Max.u64 = 0;
balar:mmio_iface:memlink.send_bit_count : Accumulator : SimTime = 289825490; Rank = 0; Sum.u64 = 0; SumSQ.u64 = 0; Count.u64 = 0; Min.u64 = 0; Max.u64 = 0;
balar:mmio_iface:memlink.output_port_stalls : Accumulator : SimTime = 289825490; Rank = 0; Sum.u64 = 0; SumSQ.u64 = 0; Count.u64 = 0; Min.u64 = 0; Max.u64 = 0;
balar:mmio_iface:memlink.idle_time : Accumulator : SimTime = 289825490; Rank = 0; Sum.u64 = 0; SumSQ.u64 = 0; Count.u64 = 0; Min.u64 = 0; Max.u64 = 0;
gpu_xbar.output_packet_count.port0 : Accumulator : SimTime = 289825490; Rank = 0; Sum.u64 = 0; SumSQ.u64 = 0; Count.u64 = 0; Min.u64 = 0; Max.u64 = 0;
gpu_xbar.input_packet_count.port0 : Accumulator : SimTime = 289825490; Rank = 0; Sum.u64 = 0; SumSQ.u64 = 0; Count.u64 = 0; Min.u64 = 0; Max.u64 = 0;
gpu_xbar.output_packet_count.port1 : Accumulator : SimTime = 289825490; Rank = 0; Sum.u64 = 0; SumSQ.u64 = 0; Count.u64 = 0; Min.u64 = 0; Max.u64 = 0;
gpu_xbar.input_packet_count.port1 : Accumulator : SimTime = 289825490; Rank = 0; Sum.u64 = 0; SumSQ.u64 = 0; Count.u64 = 0; Min.u64 = 0; Max.u64 = 0;
l1gcache_79.AckPut_recv : Accumulator : SimTime = 289825490; Rank = 0; Sum.u64 = 0; SumSQ.u64 = 0; Count.u64 = 0; Min.u64 = 0; Max.u64 = 0;
l1gcache_79.MSHR_occupancy : Accumulator : SimTime = 289825490; Rank = 0; Sum.u64 = 0; SumSQ.u64 = 0; Count.u64 = 419429; Min.u64 = 0; Max.u64 = 0;
l1gcache_79.Bank_conflicts : Accumulator : SimTime = 289825490; Rank = 0; Sum.u64 = 0; SumSQ.u64 = 0; Count.u64 = 0; Min.u64 = 0; Max.u64 = 0;
Simplehbm_0.requests_received_GetS : Accumulator : SimTime = 289825490; Rank = 0; Sum.u64 = 0; SumSQ.u64 = 0; Count.u64 = 0; Min.u64 = 0; Max.u64 = 0;
Simplehbm_0.requests_received_GetSX : Accumulator : SimTime = 289825490; Rank = 0; Sum.u64 = 0; SumSQ.u64 = 0; Count.u64 = 0; Min.u64 = 0; Max.u64 = 0;
Simplehbm_0.requests_received_GetX : Accumulator : SimTime = 289825490; Rank = 0; Sum.u64 = 0; SumSQ.u64 = 0; Count.u64 = 0; Min.u64 = 0; Max.u64 = 0;
Simplehbm_0.requests_received_Write : Accumulator : SimTime = 289825490; Rank = 0; Sum.u64 = 0; SumSQ.u64 = 0; Count.u64 = 0; Min.u64 = 0; Max.u64 = 0;
Simplehbm_0.requests_received_PutM : Accumulator : SimTime = 289825490; Rank = 0; Sum.u64 = 0; SumSQ.u64 = 0; Count.u64 = 0; Min.u64 = 0; Max.u64 = 0;
node0.dirctrl.eventSent_UnblockFlush : Accumulator : SimTime = 289825490; Rank = 0; Sum.u64 = 0; SumSQ.u64 = 0; Count.u64 = 0; Min.u64 = 0; Max.u64 = 0;
node0.dirctrl.eventSent_read_directory_entry : Accumulator : SimTime = 289825490; Rank = 0; Sum.u64 = 0; SumSQ.u64 = 0; Count.u64 = 0; Min.u64 = 0; Max.u64 = 0;
node0.dirctrl.eventSent_write_directory_entry : Accumulator : SimTime = 289825490; Rank = 0; Sum.u64 = 0; SumSQ.u64 = 0; Count.u64 = 0; Min.u64 = 0; Max.u64 = 0;
node0.dirctrl.MSHR_occupancy : Accumulator : SimTime = 289825490; Rank = 0; Sum.u64 = 44752; SumSQ.u64 = 53398; Count.u64 = 289802; Min.u64 = 0; Max.u64 = 5;
node0.memory.requests_received_GetS : Accumulator : SimTime = 289825490; Rank = 0; Sum.u64 = 177; SumSQ.u64 = 177; Count.u64 = 177; Min.u64 = 1; Max.u64 = 1;
node0.memory.requests_received_GetSX : Accumulator : SimTime = 289825490; Rank = 0; Sum.u64 = 1; SumSQ.u64 = 1; Count.u64 = 1; Min.u64 = 1; Max.u64 = 1;
node0.memory.requests_received_GetX : Accumulator : SimTime = 289825490; Rank = 0; Sum.u64 = 4165; SumSQ.u64 = 4165; Count.u64 = 4165; Min.u64 = 1; Max.u64 = 1;
node0.memory.requests_received_Write : Accumulator : SimTime = 289825490; Rank = 0; Sum.u64 = 0; SumSQ.u64 = 0; Count.u64 = 0; Min.u64 = 0; Max.u64 = 0;
node0.memory.requests_received_PutM : Accumulator : SimTime = 289825490; Rank = 0; Sum.u64 = 3679; SumSQ.u64 = 3679; Count.u64 = 3679; Min.u64 = 1; Max.u64 = 1;
node0.memory.outstanding_requests : Accumulator : SimTime = 289825490; Rank = 0; Sum.u64 = 24066; SumSQ.u64 = 24072; Count.u64 = 666265; Min.u64 = 0; Max.u64 = 2;
node0.memory.latency_GetS : Accumulator : SimTime = 289825490; Rank = 0; Sum.u64 = 531; SumSQ.u64 = 1593; Count.u64 = 177; Min.u64 = 3; Max.u64 = 3;
node0.memory.latency_GetSX : Accumulator : SimTime = 289825490; Rank = 0; Sum.u64 = 3; SumSQ.u64 = 9; Count.u64 = 1; Min.u64 = 3; Max.u64 = 3;
```


QUESTIONS

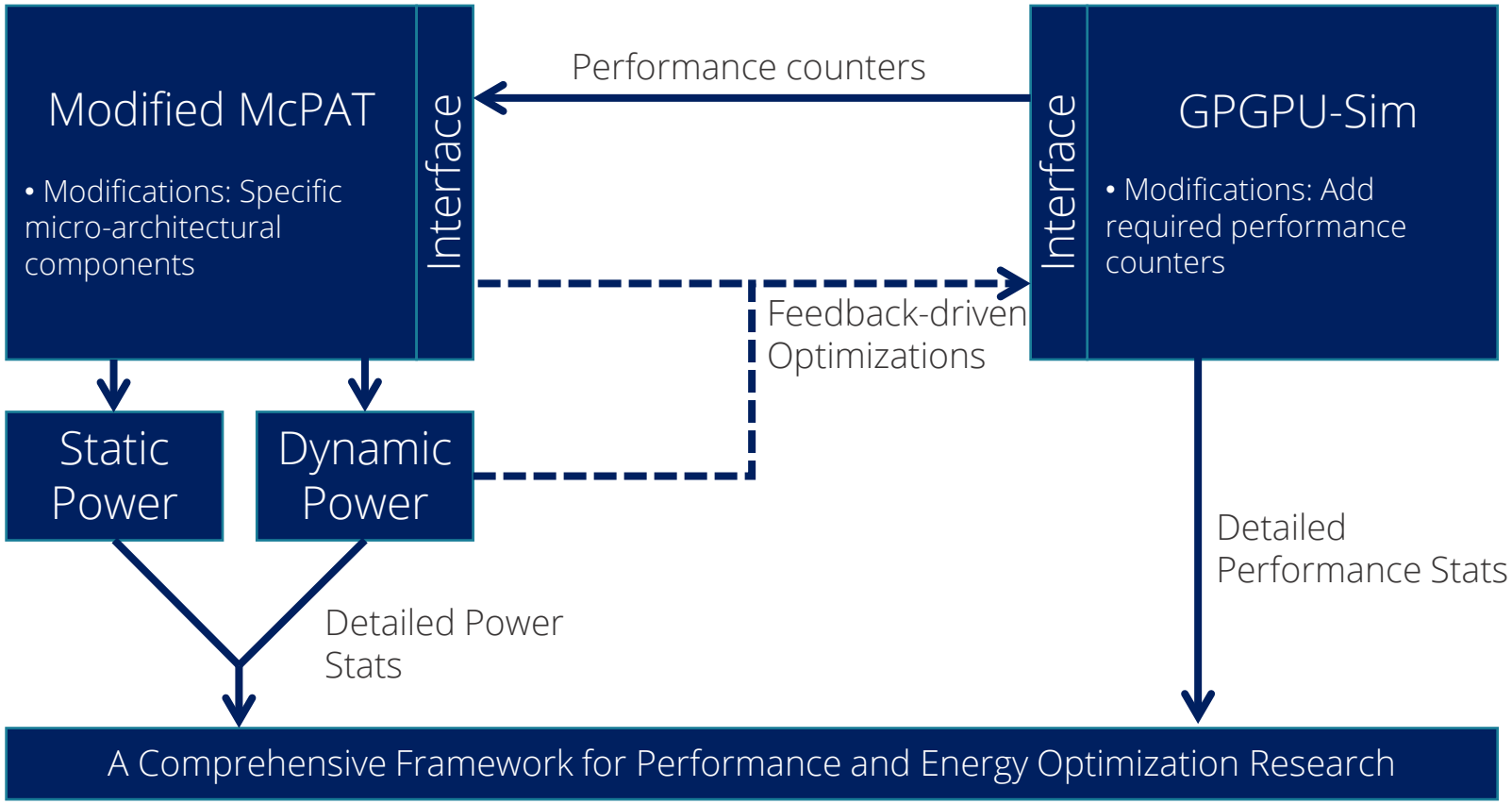
Backup Slides

GPUWattch: Power model

Estimate power consumed by the GPU according to the timing behavior

Ideal for evaluating fine-grained power management mechanisms

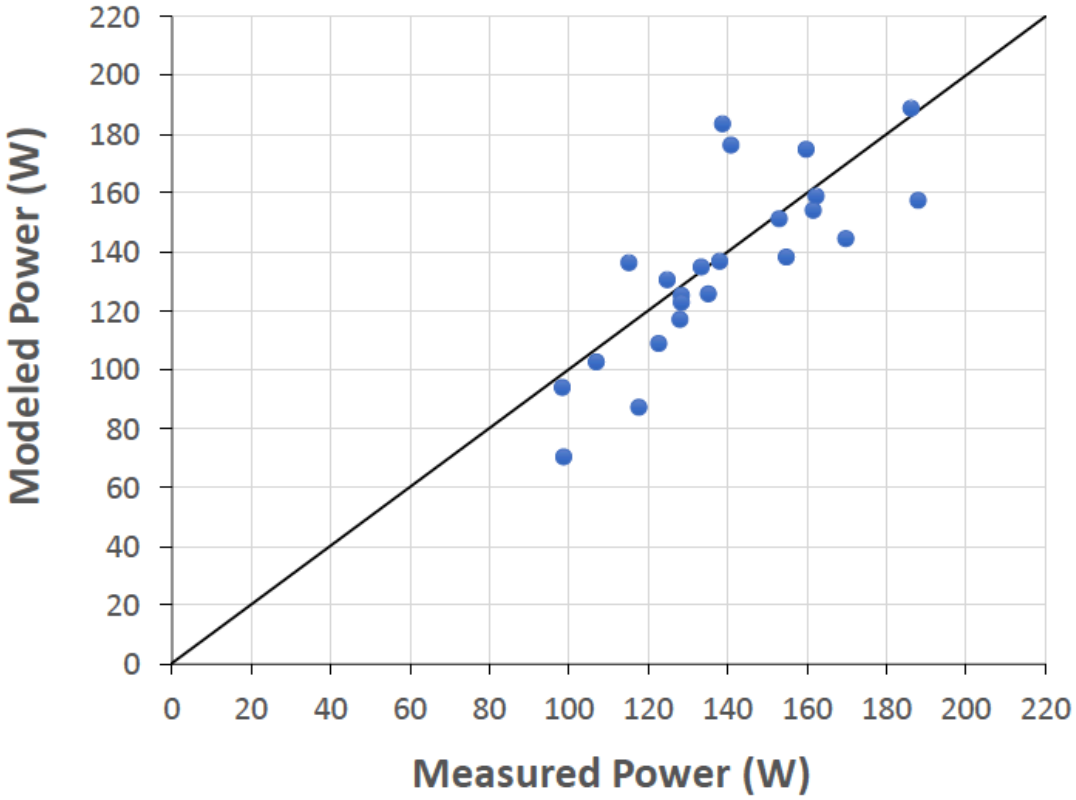
Validated with power measurements from GV100

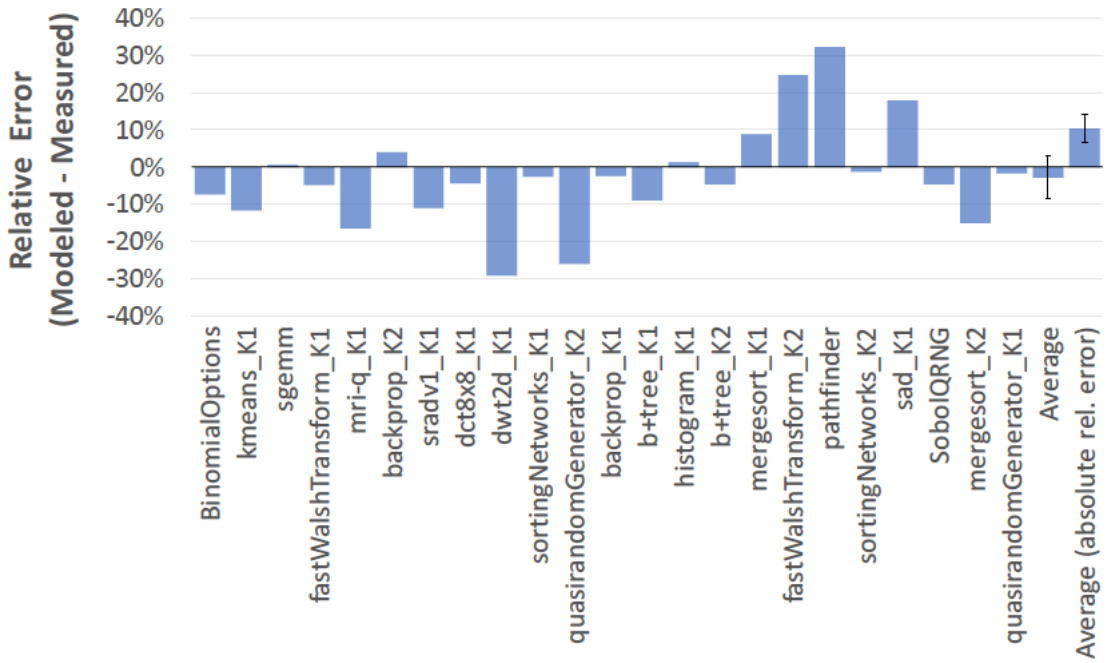


GPUWattch: Power model

Correlation for GV100

- Coefficient of 0.8
- Average relative error of $-2.67\% \pm 5.66\%$ (95% confidence interval)





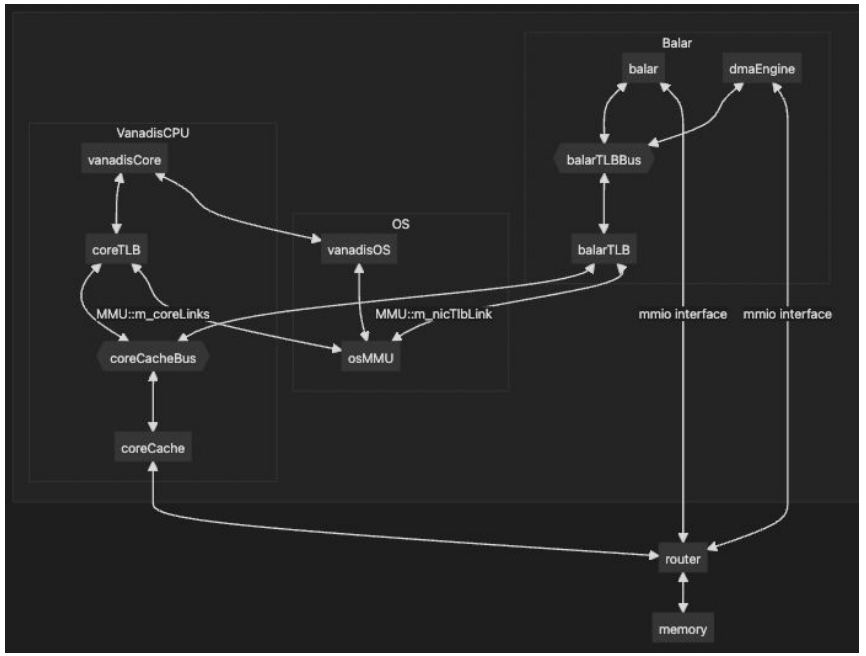
Outline

GPGPU-Sim Introduction

- GPU and programming model
- Functional model
- Performance model
- GPUWattch: power model

New Features in GPGPU-Sim

- Volta model
- Run closed source libraries
- Tensor Core
- Run CUTLASS library



New Feature: Volta Model Motivation

ISA cycle correlation[1] before new Volta model[2]

- For Pascal Titan X

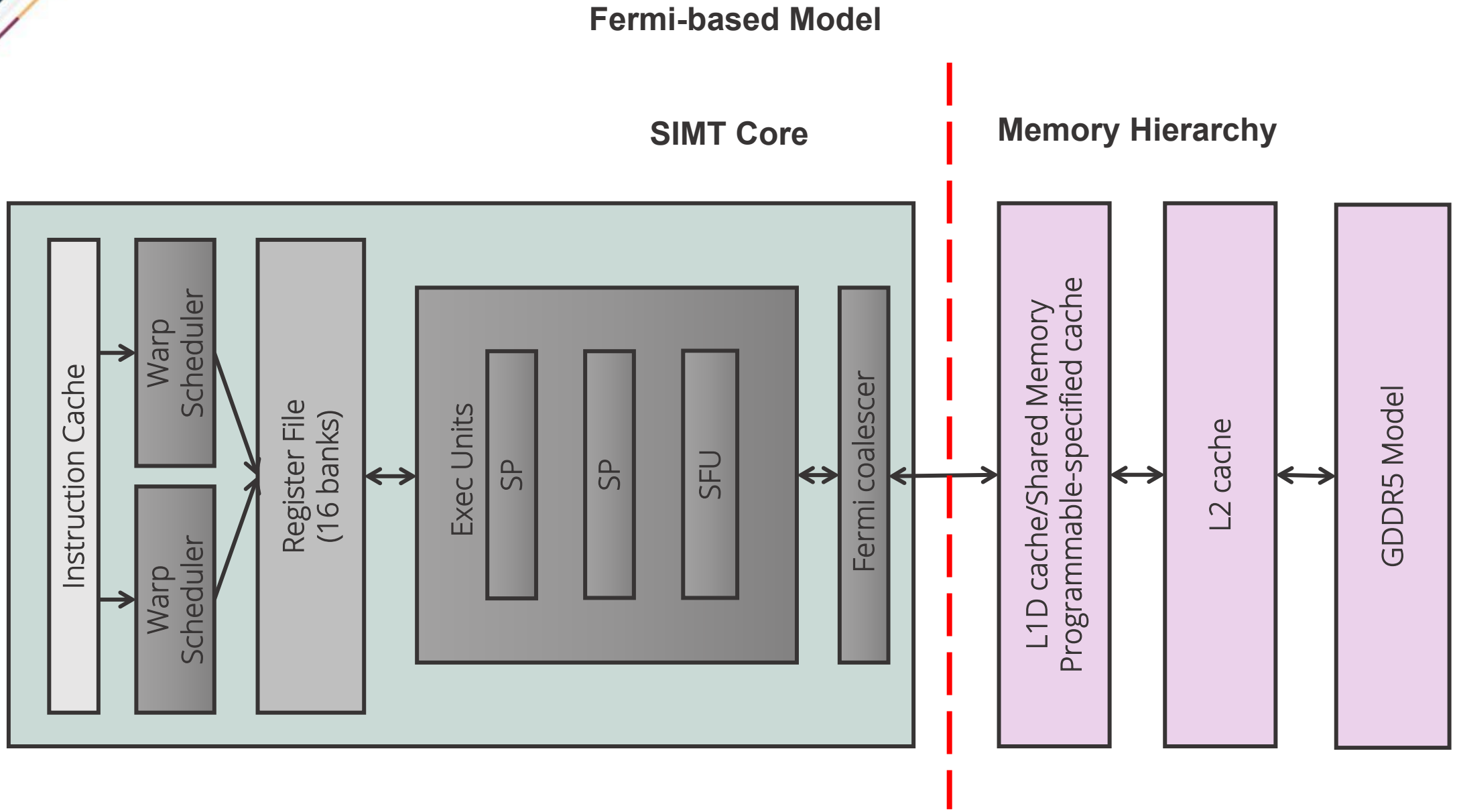
Benchmark	Means Abs Error		Correlation	
	vISA	mISA	vISA	mISA
Compute Intensive	43.3%	21.9%	91.1%	99.0%
Cache Sensitive	104.8%	100.4%	81.2%	82.0%
Memory Sensitive	31.6%	29.8%	96.0%	95.1%
Compute Balanced	58.5%	70.7%	96.1%	93.5%

Result

- Compute intensive: mISA > vISA
- Cache sensitive: both show inaccurate cache model
- Memory sensitive(streaming): not related to cache model
- Compute balanced: vISA > mISA

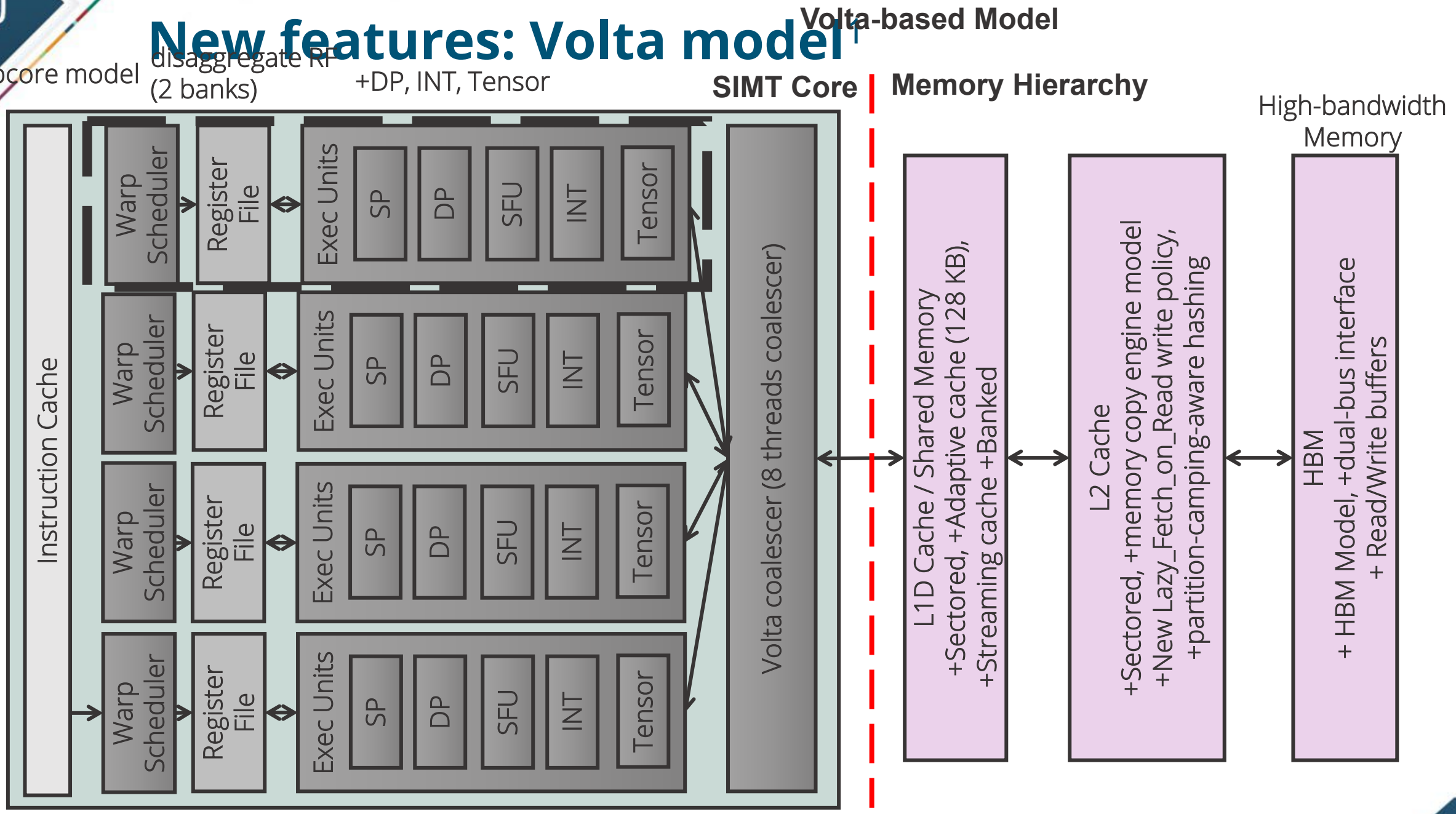
[1] Akshay Jain, Mahmoud Khairy, Timothy G. Rogers, A Quantitative Evaluation of Contemporary GPU Simulation Methodology. SIGMETRICS 2018
[2] Mahmoud Khairy, Jain Akshay, Tor Aamodt, Timothy G Rogers, Exploring Modern GPU Memory System Design Challenges through Accurate Modeling, arXiv:1810.07269

New features: Volta model[1]



[1] Mahmoud Khairy, Jain Akshay, Tor Aamodt, Timothy G Rogers, Exploring Modern GPU Memory System Design Challenges through Accurate Modeling,

New features: Volta model[1]



[1] Mahmoud Khairy, Jain Akshay, Tor Aamodt, Timothy G Rogers, Exploring Modern GPU Memory System Design Challenges through Accurate Modeling, arXiv:1810.07269

Hardware correlation for Volta model[1]

Statistic	Means Abs Error			
	Old Model	New Model	Old	
Execution Cycles	68%	27%		
L1 Reqs	48%	0.5%	92%	100%
L1 Hit Ratio	41%	18%	89%	93%
L2 Reads	66%	1%	49%	94%
L2 Writes	56%	1%		
L2 Read Hits	80%	15%	68%	81%
DRAM Reads	89%	11%	60%	95%

2.5X error reduction in exec time

0.5% error in L1 reqs (96x reqs error reduction)

SIMT Core

Memory Hierarchy

1% error in L2 behavior (66x read error reduction)

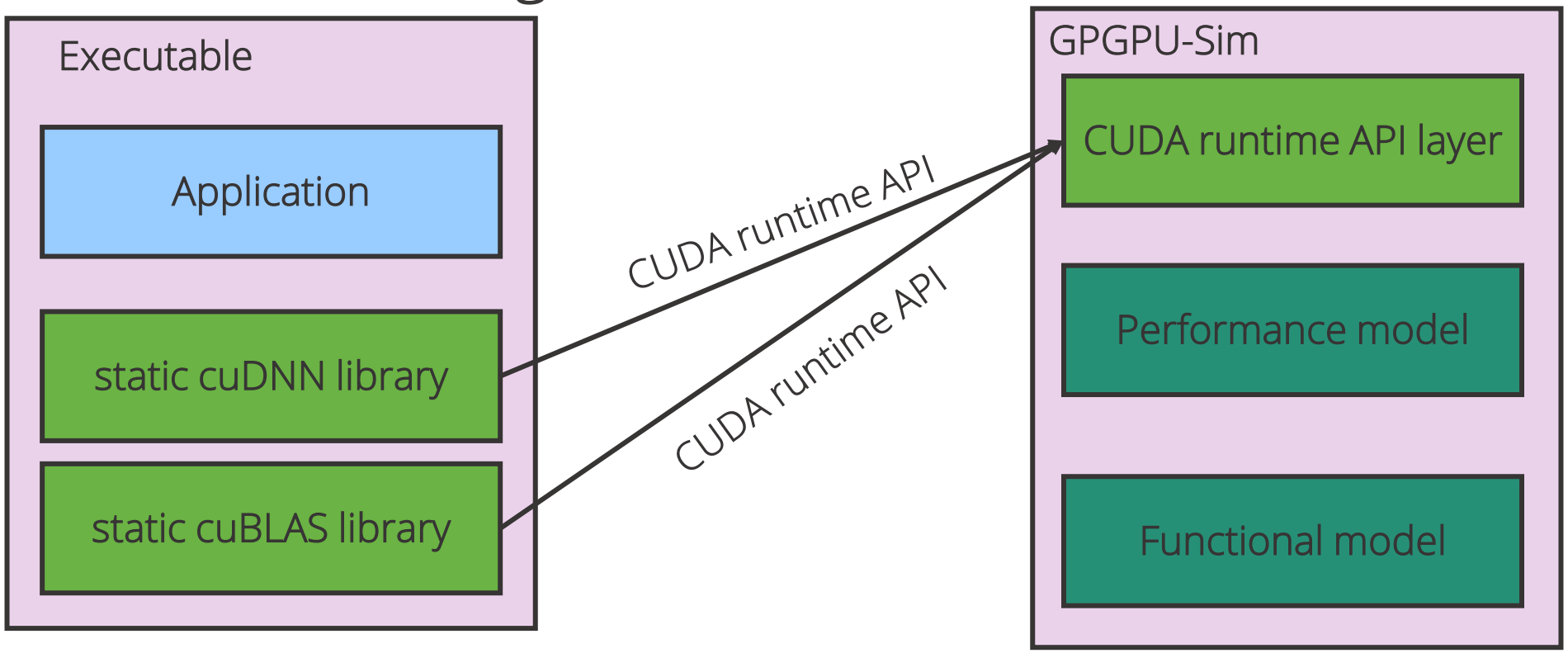
7X error reduction in DRAM reads

[1] Mahmoud Khairy, Jain Akshay, Tor Aamodt, Timothy G Rogers, Exploring Modern GPU Memory System Design Challenges through Accurate Modeling, arXiv:1810.07269

Run closed source model[1]

Run applications with cuDNN/cuBLAS

- Static linking closed source libraries
- LeNet for MNIST using cuDNN/cuBLAS



[1] Jonathan Lew, Deval Shah, Suchita Pati, Shaylin Cattell, Mengchi Zhang, Amruth Sandhupatla, Christopher Ng, Negar Goli, Matthew D. Sinclair, Timothy G. Rogers, Tor M. Aamodt Analyzing Machine Learning Workloads Using a Detailed GPU Simulator, arXiv:1811.08933

Tensor Core in GV100

Accelerate FP operations

8 Tensor Core/SM

Each perform 64 FP FMA/clock

512 FMA/clock/SM

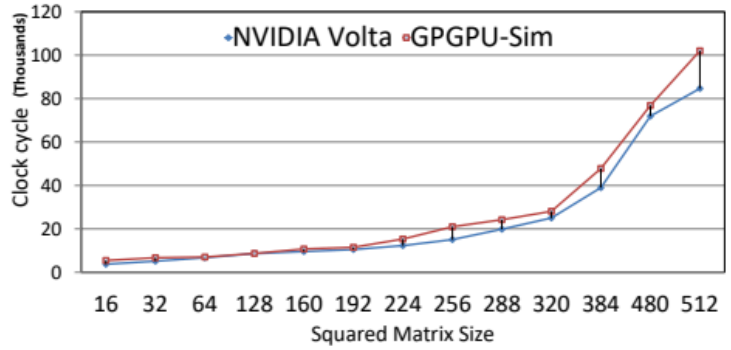
Or 1024 FP ops/clock/SM

Tensor Core in Tesla Titan V[1]

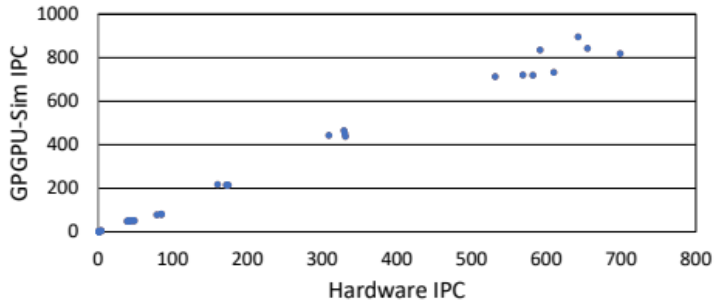
Standard deviation of less than 5%

99.60% IPC correlation

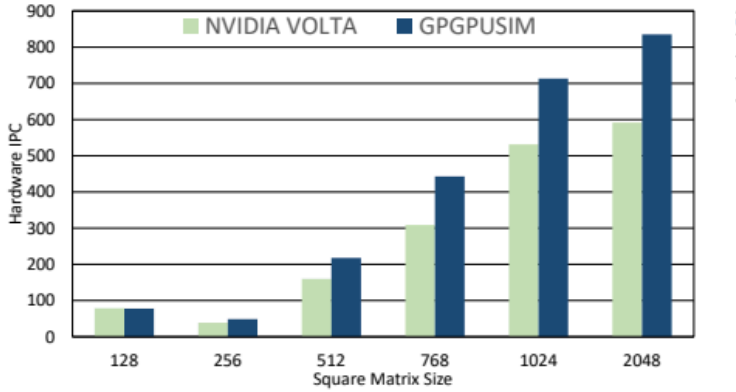
GPGPU-Sim shows higher perf than HW as matrix size increase



(a) WMMA-based GEMM kernel cycle count as matrix size varies.



(b) Instructions per cycle (IPC) correlation of CUTLASS GEMM kernel on GPGPU-Sim vs Titan V.



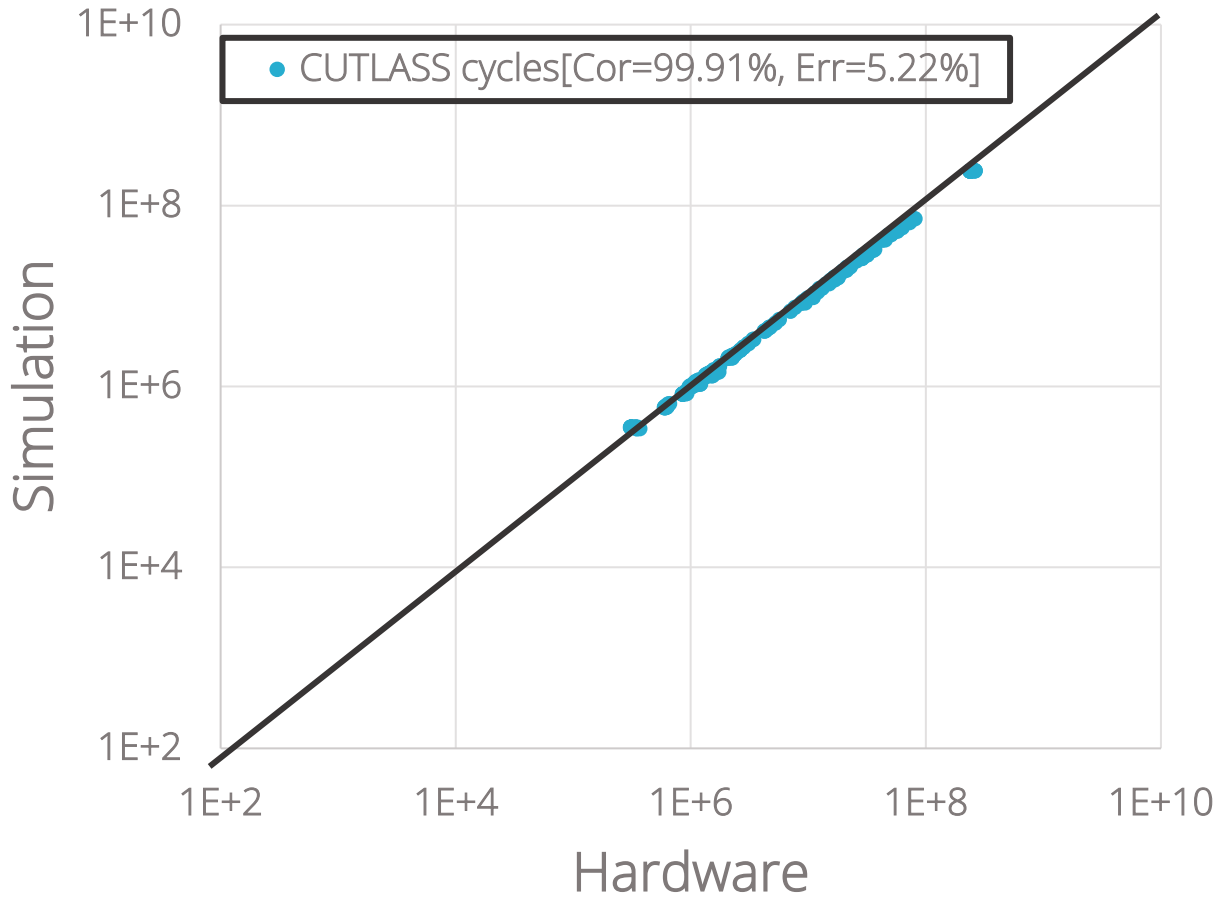
(c) CUTLASS-based GEMM kernel cycle count as matrix size varies.

[1] Md Aamir Raihan, Negar Goli, Tor Aamodt, Modeling Deep Learning Accelerator Enabled GPUs, ISPASS 2019

Run CUTLASS library

Combine CUTLASS, tensor core[1], and volta model[2]

Correlation: using Deepbench training/inference test from real scenario

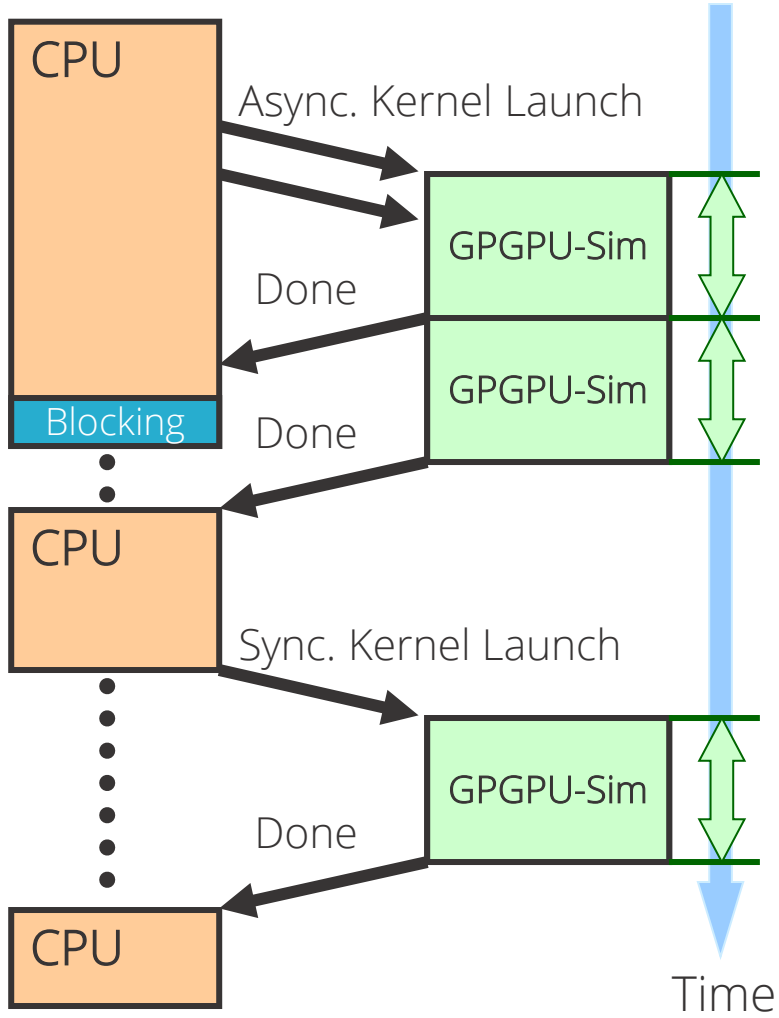


[1] Md Aamir Raihan, Negar Goli, Tor Aamodt, Modeling Deep Learning Accelerator Enabled GPUs, ISPASS 2019
[2] Mahmoud Khairy, Jain Akshay, Tor Aamodt, Timothy G Rogers, Exploring Modern GPU Memory System Design Challenges through Accurate Modeling, arXiv:1810.07269

GPGPU-Sim Introduction

GPGPU-Sim simulates kernel

- Transfer data to GPU memory
- GPU kernels runs on GPGPU-Sim:
  - Reports statistics for the kernels
- Transfer data back to CPU memory



Functional model

Single Instruction Multiple Thread(SIMT):

- SIMD + multithreading
- Grid, Block, Warp, Thread

Virtual ISA vs. Machine ISA

- vISA: PTX = Parallel Thread eXecution: virtual ISA defined by Nvidia
- mISA: SASS = Native ISA for Nvidia GPUs
- GPGPU-Sim use PTXPlus to represent SASS
  - 1:1 mapping from SASS to PTXPlus

GPGPU-Sim supports:

- PTX for new architectures(CUDA 10): new, inaccurate, well documented
- SASS for architecture before Fermi(SM_1.X): old, accurate, less documented

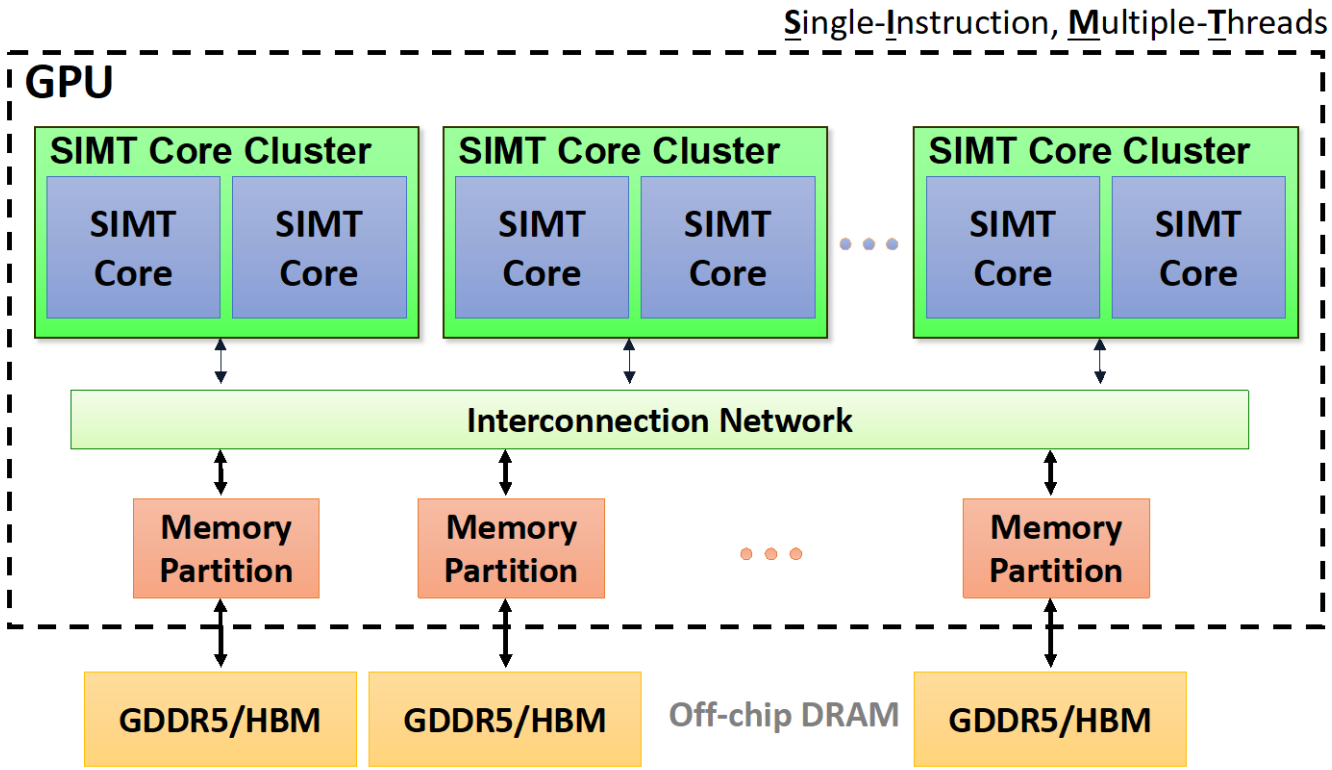
Performance model

GPGPU-Sim models timing

- SIMT Core
- Caches and texture/constant/shared memory
- Interconnection network(Books`
- DRAM(GDDR5/HBM)

DO NOT model

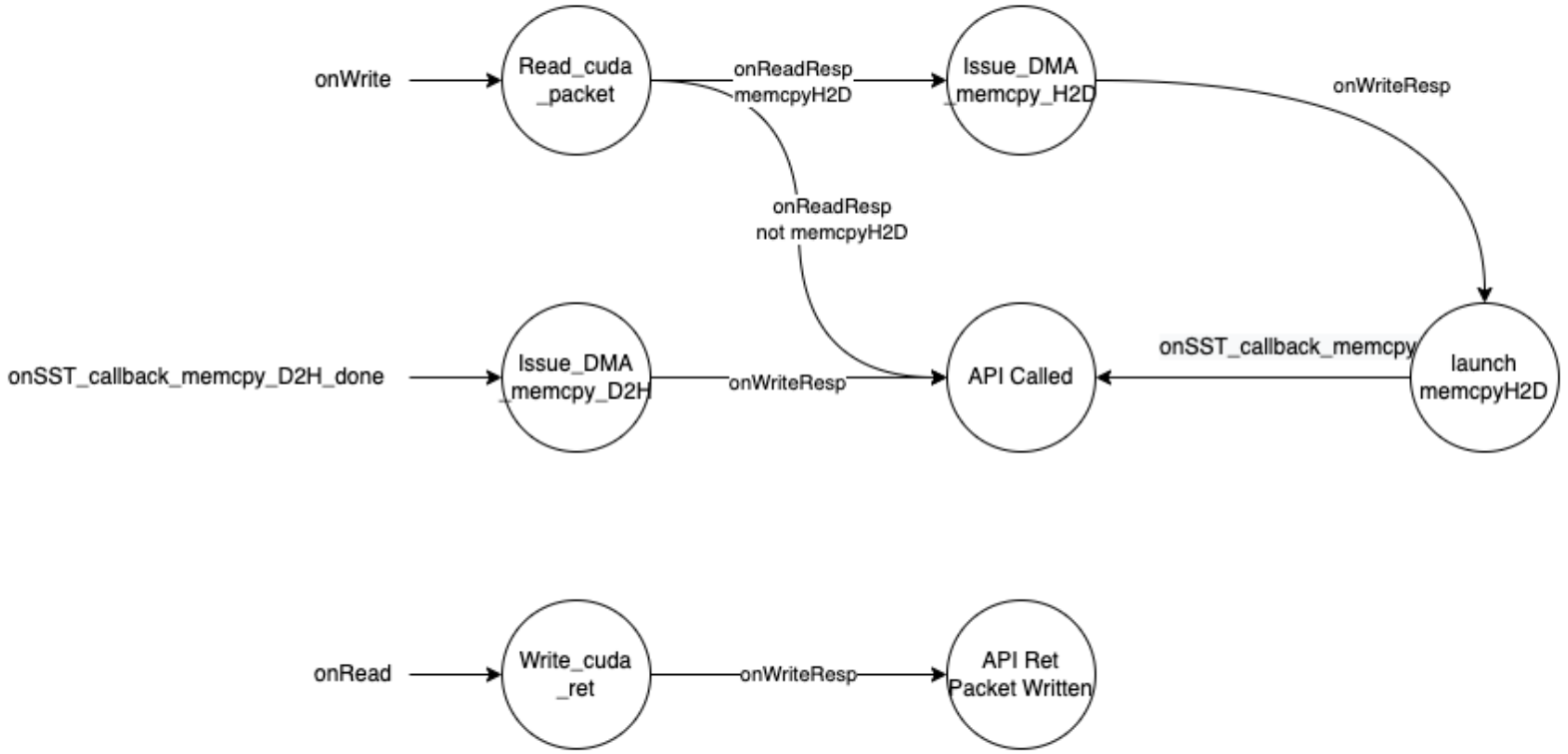
- Graphic Specific Hardware



BalarMMIO

Responsible for relaying CUDA api requests from SST to GPGPU-Sim.

Currently it supports running with CUDA traces without a real CPU model (with BalarTestCPU) or with Vanadis core



Slide Left Blank