SANDIA REPORT
SAND2021-12371
Unclassified Unlimited Release
Printed October 4, 2021

Sandia National Laboratories

A-SST Initial Specification

A. Rodrigues, S.D. Hammond, S. Hemmert, C. Hughes, J. Kenny, and G. Voskuilen
Sandia National Laboratories
Albuquerque, NM 87185
{afrodri, sdhammo, kshemme, chughes, jpkenny, grvosku}@sandia.gov

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185
Livermore, California 94550

Issued by Sandia National Laboratories, operated for the United States Department of Energy by National Technology & Engineering Solutions of Sandia, LLC.

Available to DOE and DOE contractors from

U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

Telephone: (865) 576-8401
Facsimile: (865) 576-5728
E-Mail: reports@osti.gov
Online ordering: http://www.osti.gov/scitech

Available to the public from

U.S. Department of Commerce
National Technical Information Service
5301 Shawnee Road
Alexandria, VA 22312

Telephone: (800) 553-6847
Facsimile: (703) 605-6900
E-Mail: orders@ntis.gov
Online order: https://classic.ntis.gov/help/order-methods

A-SST Initial Specification

A. Rodrigues, S.D. Hammond, S. Hemmert, C. Hughes, J. Kenny, and G. Voskuilen
Center for Computing Research, Sandia National Laboratories, Albuquerque, NM 87185

SAND2021-12371

ABSTRACT

The U.S. Army Research Office (ARO), in partnership with IARPA, are investigating innovative, efficient, and scalable computer architectures that are capable of executing next-generation large-scale data-analytic applications. These applications are increasingly sparse, unstructured, non-local, and heterogeneous.

Under the Advanced Graphic Intelligence Logical computing Environment (AGILE) program, Performer teams will be asked to design computer architectures to meet the future needs of the DoD and the Intelligence Community (IC). This design effort will require flexible, scalable, and detailed simulation to assess the performance, efficiency, and validity of their designs.

To support AGILE, Sandia National Labs will be providing the AGILE-enhanced Structural Simulation Toolkit (A-SST). This toolkit is a computer architecture simulation framework designed to support fast, parallel, and multi-scale simulation of novel architectures. This document describes the A-SST framework, some of its library of simulation models, and how it may be used by AGILE Performers.

CONTENTS

LIST OF FIGURES

Nomenclature

A-SST AGILE-enhanced Structural Simulation Toolkit

Clean Sheet A design methodology that starts without a preconceived design or other concepts. The effort is driven by the required application capabilities. This approach does not drive the effort to utilize existing design components.

Component An `SST::Component` object. It represents a model of a hardware component (e.g. cache, core, queue).

Event An `SST::Event` object. This structure contains a single discrete event that can be delivered from one component to another.

HDL Hardware Description Language

High-level Model A model of a portion of the computer system that provides functionally correct results and approximate timing.

Link An `SST::Link` object. This represents a connection between two SST Components over which SST Events can be transmitted.

Low-level Model A detailed model written in RTL used to validate correctness and cycle-accurate performance.

Mixed-Fidelity Simulation A simulation methodology in which less scalable, more detailed simulation models and faster, more abstract simulation models are run together in the same simulation.

Multi-Fidelity Simulation A simulation methodology in which less scalable, more detailed simulations are used to parameterize faster, more abstract simulations.

NoC Network-on-Chip

RTL Register Transfer Language

T&E Test and Evaluation

This page intentionally left blank.

1. A-SST

Performer teams in the IARPA AGILE program will be tested and evaluated using the AGILE-enhanced Structural Simulation Toolkit (A-SST), which is derived from Sandia National Labs' Structural Simulation Toolkit[1][18]. Under this program it is the Performer's responsibility to ensure interoperability of their models with the A-SST framework.

1.1. A-SST Overview

SST is an open-source computer architecture simulation toolkit developed to explore innovations in highly concurrent systems in the HPC and embedded space. Key features of SST:

- **Parallel**: SST is built from the ground up to be parallel. At its heart, SST is a component-based parallel discrete event simulator (PDES) with clocking that uses a conservative distance-based optimization over MPI and C++ Threads.
- **Multi-scale**: SST does not dictate a specific level of simulation. It can and has been used for detailed gate-level simulation, high-level runtime algorithm evaluation, abstract state machine-based simulations, and even non-computer simulations such as car washes. SST aims to provide a variety of simulation models "out of the box" which include both detailed and simple models for processors, network components, and memories.
- **Interoperable**: SST allows easy integration with a variety of simulators including RTL language tools, behavioral simulators, and commonly used simulators like GPGPU-Sim and Gem5.
- **Open**: SST is open source software released under a permissive license[2]. It allows redistribution and use in source and binary forms, with or without modification. To facilitate this on a software engineering level, the toolkit is designed to be modular and extendable without requiring pervasive changes. Individual component models can be added without any changes to the simulator core.

A-SST will be a development branch of SST with additional enhancements to support the AGILE program.

[1] SST's main website is `http://sst-simulator.org/` and the open Git repository is `https://github.com/sstsimulator/`.

[2] `https://github.com/sstsimulator/sst-elements/blob/master/LICENSE`

1.2. A-SST and AGILE

For the AGILE program, A-SST's goal is to provide Performers with a flexible simulation and testing framework. This will allow rapid design of system components and rapid modeling of the AGILE Applications. To support Performer productivity, the AGILE program will provide a number of example and baseline simulation configurations to demonstrate AGILE workflows with existing SST component models. However, the goal of the AGILE program is to enable "clean sheet" designs which are free from preconceived design concepts and are driven by application needs. As such, Performers are not required to use any existing SST component models in their designs.

As a direct branch of SST, A-SST will maintain these key features. In Phase 1 of AGILE it is expected that components will be represented by **high-level models** – models written in high-level languages that show functional correctness and approximate timing. In Phase 2 of AGILE, Performers will produce RTL designs for use in A-SST and on FPGA platforms. Use of FPGAs will allow more precise timing estimates and enable the T&E teams to perform validation and verification. Performers will need to prove that their designs work at a node-level and also at scale. Due to the limitations of simulation speed and available resources, at-scale simulation will require multi-fidelity or multi-resolution modeling where detailed, but non-scalable, simulation models and FPGA emulation runs are used to inform faster, more scalable abstracted models (See Section3.2).

1.3. A-SST Architecture

1.3.1. Core and Elements

A-SST is divided into two principal regions (Figure 1-1):

- The **Core**[3] includes the PDES capabilities that enable simulation. It also provides utilities and interfaces for simulation components (models) such as clock creation, event exchange and serialization, some common interfaces, statistics and parameter management, and parallelism support.

- The **Element**[4] libraries encapsulate the simulation models. Most importantly, element libraries contain **component** objects that perform the actual simulation (e.g. a processor core, cache, network router, etc...) and **event** objects that are the discrete event structures communicated between components (e.g. memory request, network packet, etc...).

[3] https://github.com/sstsimulator/sst-core
[4] https://github.com/sstsimulator/sst-elements

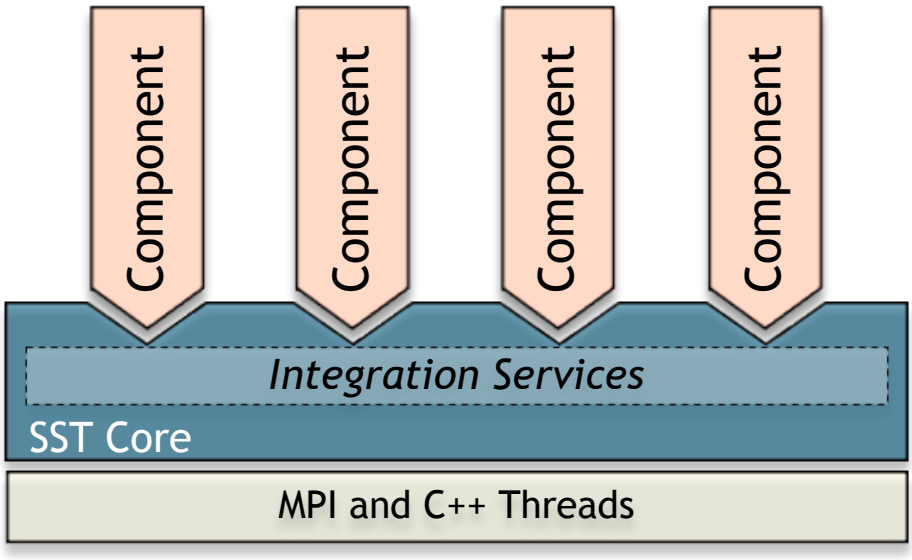


Figure 1-1. A-SST Core and Components

1.3.2. Key Objects

SST's Core and most elements are written in C++. The SST Core is limited to C++11 for portability, though elements are not restricted. The basic object structure of A-SST is simple (Figure 1-2). Writing a basic component requires only a handful of object types:

- **`SST::Component`** contains the actual simulation logic and state for a piece of hardware (e.g. cache, processor core, etc...).

- **`SST::Event`** a discrete event description that can be passed between components. User-specified events (which inherit from SST's base event class) can contain arbitrary data. If events need to be passed between components in a parallel simulation they will need to be serializable and provide a method for doing so.

- **`SST::Link`** represents a connection between two components over which `SST::Events` may be passed. Links are bidirectional and point-to-point. Each link is specified with a minimum latency to assist with parallel performance (See Section 1.3.3).

- (Optional) **Event Handler** is a function object within a component that is invoked when an event arrives (i.e. is "pushed" to the component) over an `SST::Link`. Components are not required to have event handlers, but if they do not they will need to "pull" events from the link, usually in a clock handler.

- (Optional) **Clock Handler** is a function object within a component that is invoked at a regular interval, equivalent to a clock edge. Within a clock handler, components can advance their state and send and recieve events.

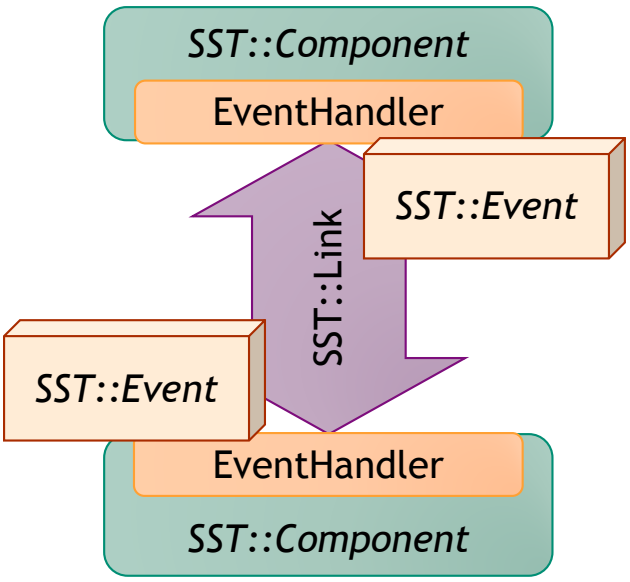


Figure 1-2. A-SST Key Objects

1.3.3. Parallelism

A-SST can use both threads and MPI to achieve parallelism. This is transparently handled by the SST Core, so components should never make an MPI call. For parallel simulation, events will need to be serializable. The core is conservative (i.e. no roll-back) and uses a distance-based optimization so sets of simulation components on each rank are only synchronized when needed to ensure correct message ordering.

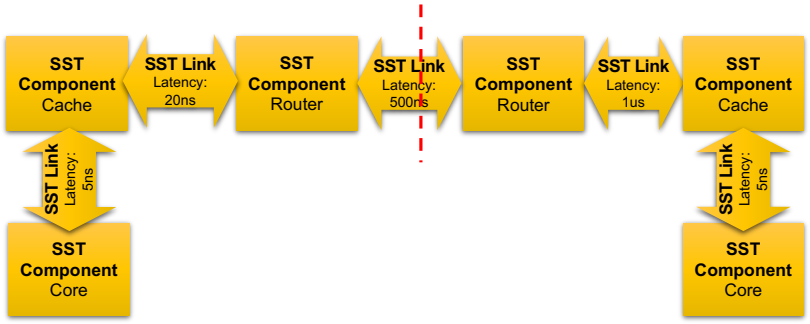


Figure 1-3. For best performance, components should be partitioned along high-latency links

Parallel execution over MPI requires that components be partitioned and assigned to MPI ranks (Figure 1-3). To maximize parallel performance, this partitioning should load balance (so roughly equal amounts of work occur on each rank) and partition on high-latency links (so synchronization is less frequent). A-SST users can specify a manual partitioning, write their own partitioning algorithm, or use one of A-SST's provided algorithms:

- **linear**: Partitions components by dividing Component ID space into roughly equal portions. Components with sequential IDs will be placed close together.

- **round-robin**: Partitions components using a simple round robin scheme based on ComponentID. Sequential IDs will be placed on different ranks.
- **simple**: Simple partitioning scheme which attempts to partition on high latency links while balancing the number of components per rank.
- **zoltan**: Uses the Zoltan[7] parallel partitioner (deprecated).
- **single**: Allocates all components to rank 0. Automatically selected for serial jobs.
- **self**: Used when partitioning is already manually specified by the user in the configuration file.

1.3.4. Simulation Lifecycle

Each SST simulation follows a standard "lifecycle":

1. **Configure** SST is configured with a Python script that specifies the components, their initialization parameters, links between the components, and other simulation configurations. The SST Core starts a simulation by reading the Python configuration file.
2. **Create Graph** SST will then create a connected graph of all the components in the simulation, and partition the graph (see Section 1.3.3) between MPI ranks and/or threads.
3. **Instantiate** Each MPI Rank in the SST simulation receives its potion of the partitioned graph and instantiates its components (i.e. calls their constructors).
4. **Connect** SST links are created and each component is connected as specified in the Python configuration script.
5. **Initialize** Once constructed, the components enter a multi-phase initialization period during which they may exchange data in an untimed fashion. This initialization period is generally used for network self-discovery, additional link parameter negotiation, and for loading executables in to memory. This initialization is contained in each component's `init()` function which is invoked repeatedly until all components' initialization is complete.
6. **Setup** Immediately before timed simulation begins, each component's `setup()` function is called, allowing any final configuration or initialization.
7. **Simulate** The actual simulation is a series of activities during which clock functions are invoked on components, events are sent and recieved between components, and time is advanced. Parallel synchronization and communication of events between components on different ranks or threads is handled by the SST Core. Simulation continues until a user-specified time or until all components indicate they are done.
8. **Complete** Similar to the initialization period, a multi-phase completion period occurs immediately after simulation (using each component's `complete()` function.
9. **Finalize** Similar to the Setup stage, each component's `finish()` function is called.

10. **Output** Statistics are output[5].
11. **Cleanup** Component destructors are called and the SST Core shuts down.

[5] See `http://sst-simulator.org/SSTPages/SSTDeveloperSSTStatisticsAPI/` for more information on SST's statistics API

2. KEY A-SST ELEMENTS

A-SST's standard distribution contains a variety of element libraries and components. These components will be used to construct example architectures which demonstrate the AGILE workflows. They may be used, extended, and modified by Performers, but are not required to be used.

The following sections give brief overviews of several important components and element libraries in SST.

2.1. Memory Hierarchy Components

memHierarchy is a large set of components for modeling the memory hierarchy. It includes parametrizable caches, scratchpads, busses, connections to on-chip networks, directory controllers, and a variety of memory controllers. Through the Cassini element library, memHierarchy can access a a set of modular and composable prefetchers that use standard prefetching heuristics. MemHierarchy allows arbitrary-depth cache hierarchies and memory topologies, allowing complex systems to be modeled (Figure 2-1). Because the memory system plays such a central role in the node, memHierarchy is used to connect a variety of other SST components. Some of the main memory models it supports include:

- **SimpleMemory**: A simple fixed-timing model for main memory that is inaccurate but very fast.
- **SimpleDRAM**: A simple DRAM model that accounts for basic DDR parameters (tCAS, tRCD, and tRP) and bank contention.
- **TimingDRAM**: A moderately-complex DRAM model that accounts for basic DDR timing parameters and channel, rank, and bank contention.
- **dramsim3**: Connects to DRAMSim3[15] which contains very detailed models of several DRAM protocols such as DDR3, DDR4, LPDDR3, LPDDR4, GDDR5, GDDR6, HBM, HMC, and STT-MRAM.
- **goblinHMCSim**: Goblin HMC-Sim is a Hybrid Memory Cube functional simulator.
- **pagedMulti**: A memory backend that models a two-level main memory and a variety of policies to move data between the levels to improve performance[8]. Internally, it uses DRAMSim and fixed timing models.
- **cramsim**: Detailed modeling of DDR and HBM memory, based on CramSim[9].
- **Messier**: Parametrizable model of emerging non-volatile memories[2] including timing parameters and data placement policies.

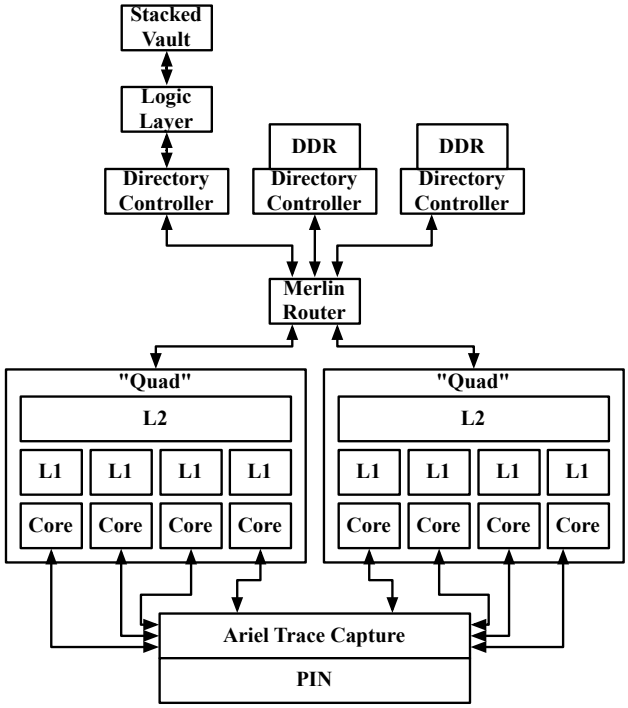


Figure 2-1. Example MemHierarchy simulation showing multiple levels of cache and multiple main memory types. Addresses can be interleaved or blocked between memory types.

2.2. Processor Components

A-SST contains a number of "processor" models ranging from simple memory traffic generators to full cycle-approximate simulators capable of booting an OS. These models interface with memory using the `simpleMem` or `standardMem` APIs. Some of the simulation models include:

- **Miranda** is a pattern-based processor model / traffic generator. Each Miranda core consists of a request generator and a core that issues requests, retires requests, and tracks request dependencies. It can easily be extended to produce a number of memory traffic patterns and comes with a set of existing pattern-generators:
  - **CopyGenerator**: Copies an array *A* to an array *B*
  - **RandomGenerator**: Generates random accesses
  - **GUPSGenerator**: Read followed by write to the same (random) address
  - **STREAMBenchGenerator**: Implements triad loop from the *Stream* benchmark.
  - **InOrderSTREAMBenchGenerator**: *Stream* benchmark with compiler optimization to block accesses (i.e. Read chunks of *B* and *C*, then write chunk of *A*).
  - **SingleStreamGenerator**: Stream of reads starting at an address and moving forwards
  - **ReverseSingleStreamGenerator**: Stream of reads starting at an address and moving backwards

- **SPMVGenerator**: Sparse Matrix-Vector access pattern
- **Stencil3DGenerator**: 3D 27-point stencil pattern

In addition, it is expected that the T&E team will produce pattern-generators for one or more of the AGILE workloads.

- **Ariel** is a lightweight processor core model that receives a stream of memory accesses from a dynamically instrumented binary. Currently, Ariel uses Intel's PIN[1] tool for dynamic instrumentation. It is expected that by the start of AGILE, a version based on DynamoRIO[5][2] will be available. Ariel can use existing binaries (including fixed-count multi-threaded binaries) which, through dynamic instrumentation, pass a stream of events to a simple core model in SST (Figure 2-2). These events are usually loads and stores, but can also contain other user-defined events such as memory allocations or system calls.

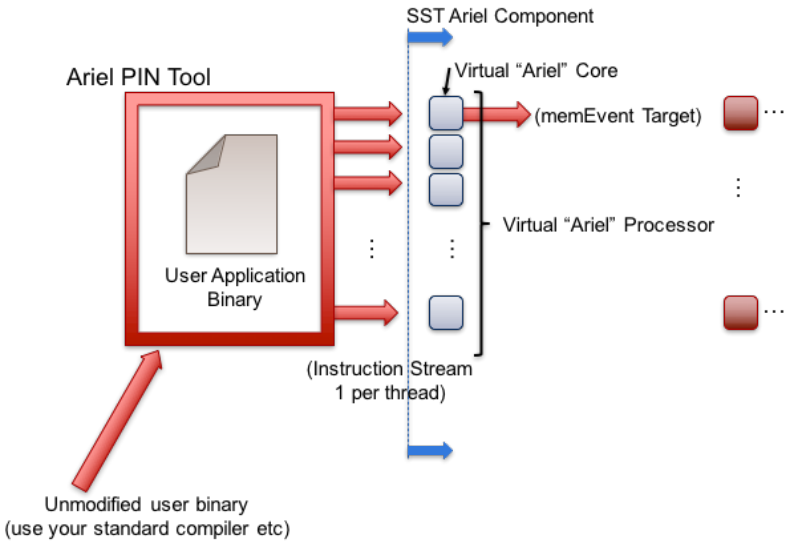


Figure 2-2. Ariel Processor Model

Ariel provides a simple mechanism for prototyping and is most useful when applications are memory- or communication-bound as the AGILE applications are expected to be.

- **Prospero** is a trace-based processor model. Like Ariel, it is memory instruction oriented. Propsero reads memory operations from a file and passes them to the simulated memory system. It supports arbitrary length reads to account for variable vector widths. SST comes packaged with the Prospero Trace Tool to generate traces.
- **Vanadis** provides flexible processor core models in multiple ISAs. It has been designed to integrate closely with SST and support a variety of use cases (Figure 2-3). Currently, it supports the MIPS ISA with RiscV support expected by the start of the AGILE program. OS calls are emulated.

[1] https://software.intel.com/content/www/us/en/develop/articles/pin-a-dynamic-binary-instrumentation-tool.html

[2] https://dynamorio.org/

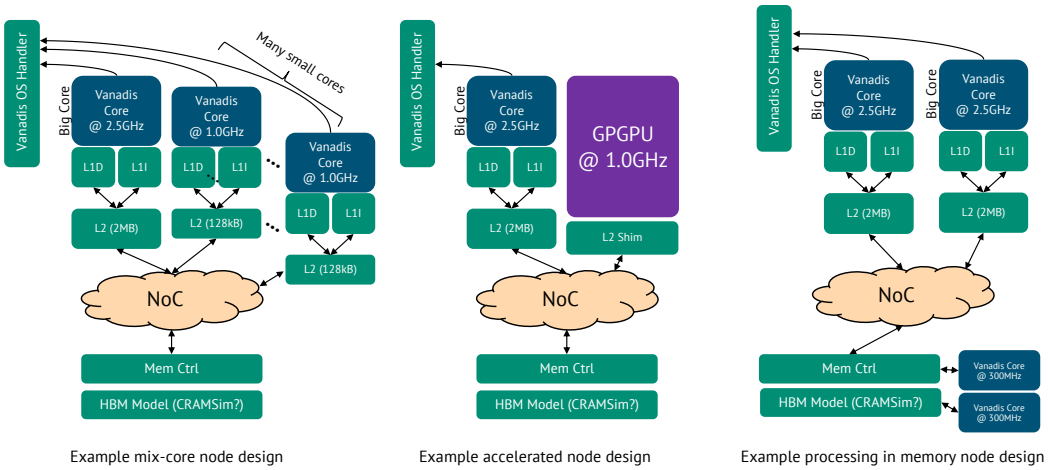


Figure 2-3. Vanadis Use Concepts

- **Gem5**[16] is a flexible architecture research platform for system and microarchitecture exploration. It is in wide use in academia and industry. It is capable of booting full operating systems and supports a number of ISAs. Gem5/SST integration work is ongoing, but currently is capable of connecting Gem5 cores and certain peripherals to SST's memHierarchy components.
- **GPGPU-Sim**[13] is a cycle-level simulator that models GPUs running CUDA. It has been integrated with SST under the Balar[10][3] element library. Balar provides additional modularity and scheduling flexibility to GPGPU-sim.
- **Spike** is a functional simulator of the RISC-V ISA[4] built on the Miranda core model.

2.3. Network Components

A-SST contains a number of ways to represent networks and network endpoints. Endpoints interface with network models using the `simpleNetwork` API.

2.3.1. Network Endpoints

- The **Ember / Hermes / Firefly** stack (Figure 2-4) is a set of related components that can be used to create abstracted network endpoints that are very light-weight and highly scalable. Network simulations with one million endpoints have been performed using the Ember family of components

  Ember, Hermes, and Firefly represent a nested series of state machine-like components that encapsulate an application's communication pattern (or *motif*) and associated messaging runtime and use it to generate a series of network events and transactions.

[3] `https://github.com/sstsimulator/balar`
[4] `https://github.com/tactcomplabs/SSTStake`

Figure 2-4. Ember/Hermes/Firefly Stack

At the top of this stack is **Ember**[5] which packages communication patterns in to state-machine like components. This state machine can encode a high level of complexity in the patterns. Ember also includes generic methods for users to add new communication patterns. Some example motifs include 1D, 2D, and 3D Halo exchange and communication sweep patterns (e.g. Sweep3D). It is anticipated that the T&E teams will provide additional motifs representing AGILE workloads or kernels.

Ember sends a series of events to the **Hermes** component that correspond to communication primitives such as send, receive, or allreduce. Hermes transforms these events into network traffic and handles message passing semantics (e.g., collectives, matching, etc...). Hermes can currently model MPI-like and SHMEM-like communication runtimes.

At the bottom of the stack is **Firefly**[6], which, like a NIC, handles low-level data movement such as packetization and communication with the network. Firefly connects to Merlin network models.

- **SHMEMNIC** is a SHMEM-based NIC that can perform load and store operations over a disaggregated memory network. Currently, it interoperates with the Vanadis core model.
- **Macro**[14] is a collection of high-level network components and endpoint models to support coarse-grained simulation of full-scale HPC systems. A key feature of the Macro model is the use of "skeletonization" to produce endpoint models. Skeletonization is a semi-automatic process that recompiles code, stripping out memory allocations and compute and intercepting communication or other runtime calls (Figure 2-5). The resulting code can be integrated into a simulation to emulate the concurrent execution of many virtual application processes in a single simulator process.

  Because the application's state and compute are greatly reduced simulations can be performed much more quickly while still retaining the application's messaging pattern and

[5] http://sst-simulator.org/SSTPages/SSTElementEmber/
[6] http://sst-simulator.org/SSTPages/SSTElementFirefly/

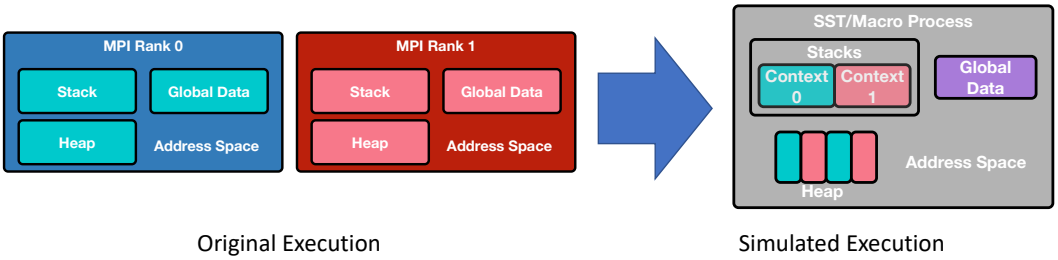


Figure 2-5. Compiler-based skeletonization of an application for simulation

internal logic. Additionally, by intercepting runtime calls it may be possible to use skeletonization as an efficient mechanism for prototyping runtime or hardware-based runtime acceleration (See Section 3.3).

2.3.2. Network Models

- **Macro**, in addition to providing network endpoint models, also provides coarse-grained network models. These can model multiple topologies as packet-based networks.
- **Merlin**[6][12] is an element library for modeling High-speed system-level or NoC networks. Merlin uses a High-radix router model (`hr_router`) that is based on common HPC routers. It contains buffers, an internal cross-bar and logic for routing, flow control, and network initialization (Figure 2-6). Merlin can handle a number of topologies including meshes, *n*-dimensional tori, fat-tree, and dragonfly. There are several options for adaptive routing. Additionally, there is a defined interface for adding new topologies to the router. Merlin can be used as a system-level interconnect (often combined with the Ember/Firefly stack) or as an on-chip network (usually combined with memHierarchy components).

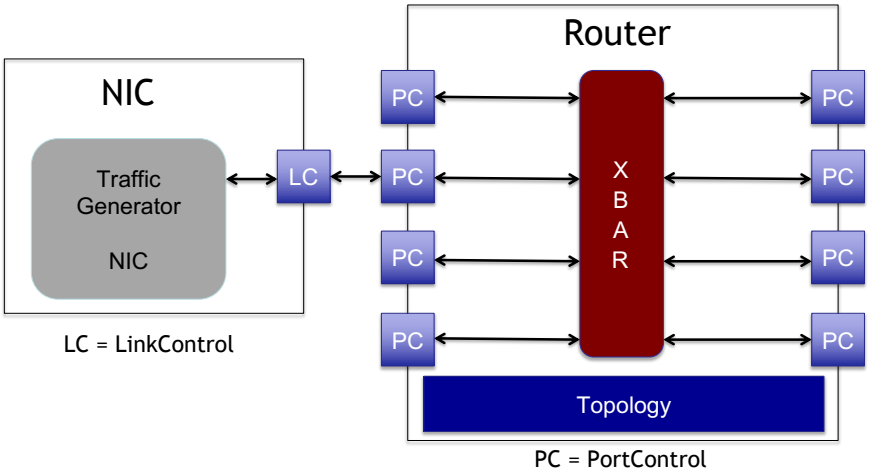


Figure 2-6. Merlin `hr_router` Internals

- **Kingsley** is similar to Merlin, but focused on NoC networks. As such, it uses input-only router buffering policies, unlike Merlin. Kingsley models a mesh topology.

- **Shogun** provides a crossbar NoC model.

2.4. Low-Level Models

Phase 1 of AGILE will focus on flexible high-level development of architectural components. Phase 2 will transition to **low-level models** written in RTL to allow verification and validation of functional correctness and to refine timing estimates.

The T&E team will provide ERAS[17], a framework for converting RTL models to be integrated with SST, so Performers can use their Phase 2 designs within SST (Figure 2-7). ERAS uses the Essential Signal Simulation Enabled by Netlist Transforms (ESSENT)[4] software package to convert FIRRTL[11] intermediate representation code into a C++ simulator. This C++ simulator is then parsed and 'wrapped' in automatically generated and user-supplied code to create an SST component. FIRRTL code can be produced from Chisel[3] sources or from Verilog[1] via the Yosys[20] synthesis suite.

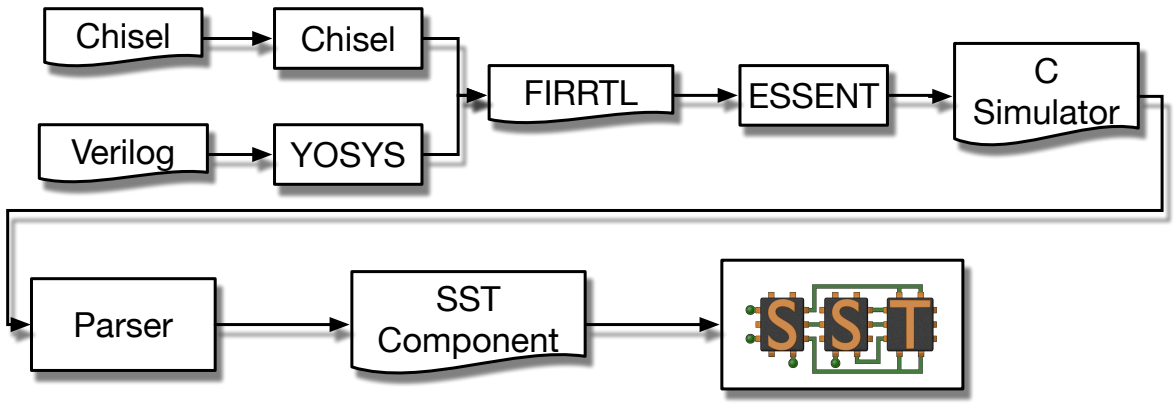


Figure 2-7. RTL-level Simulation Workflow with ESSENT and SST

The ERAS parser combines several inputs to create an SST component (Figure 2-8). The ESSENT toolchain produces a C++ header file that contains the model of the components to be simulated. The ERAS framework's parser combines this header file with an internal template and user-supplied code. Predefined templates which connect common interfaces like AXI and UART to SST's memHierarchy components are part of ERAS. The user specifies additional code for component creation and interfacing with SST's Core (e.g., statistics gathering, `init()`, `finish()`, etc...).

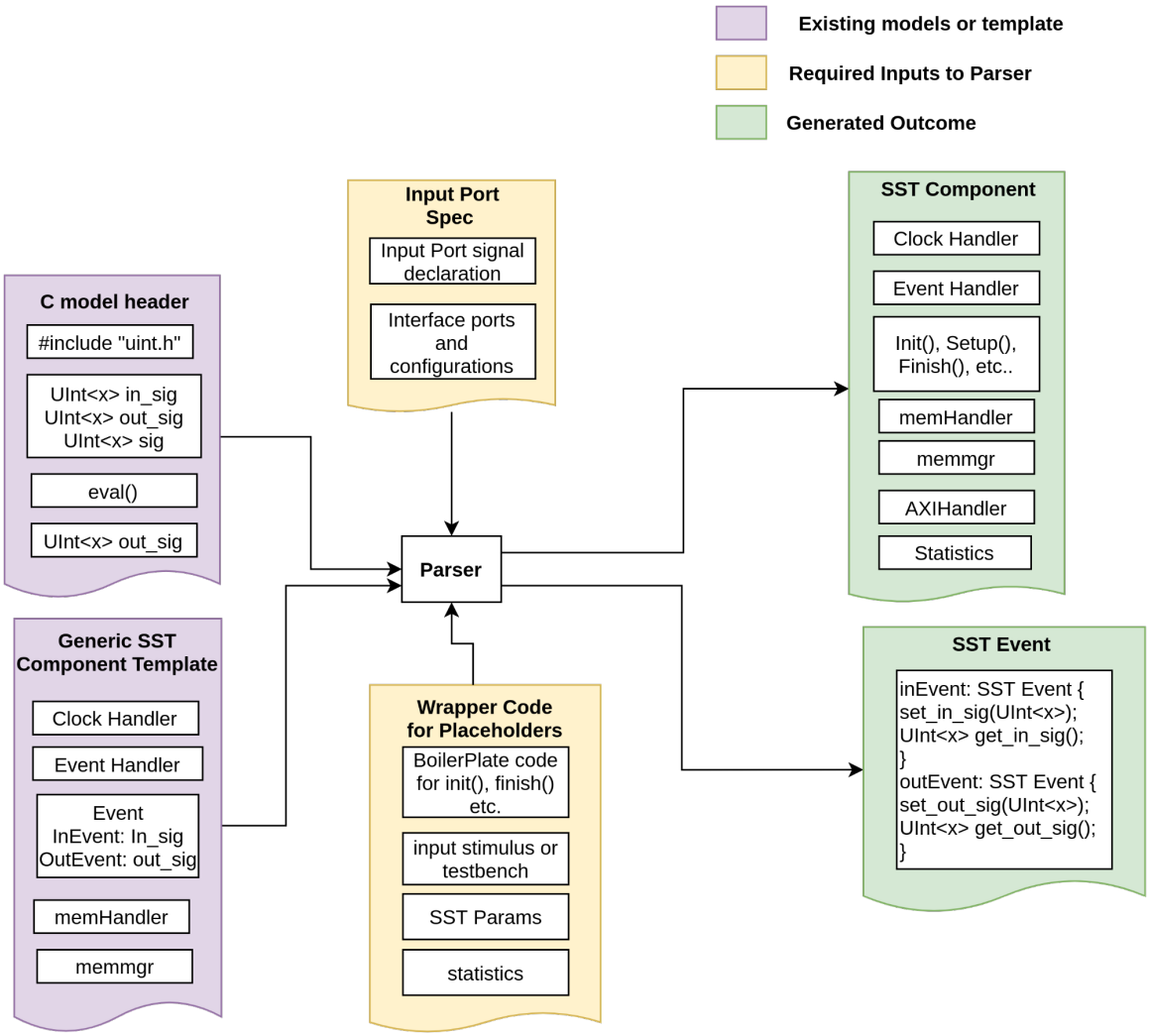


Figure 2-8. ESSENT Parser Inputs and Outputs

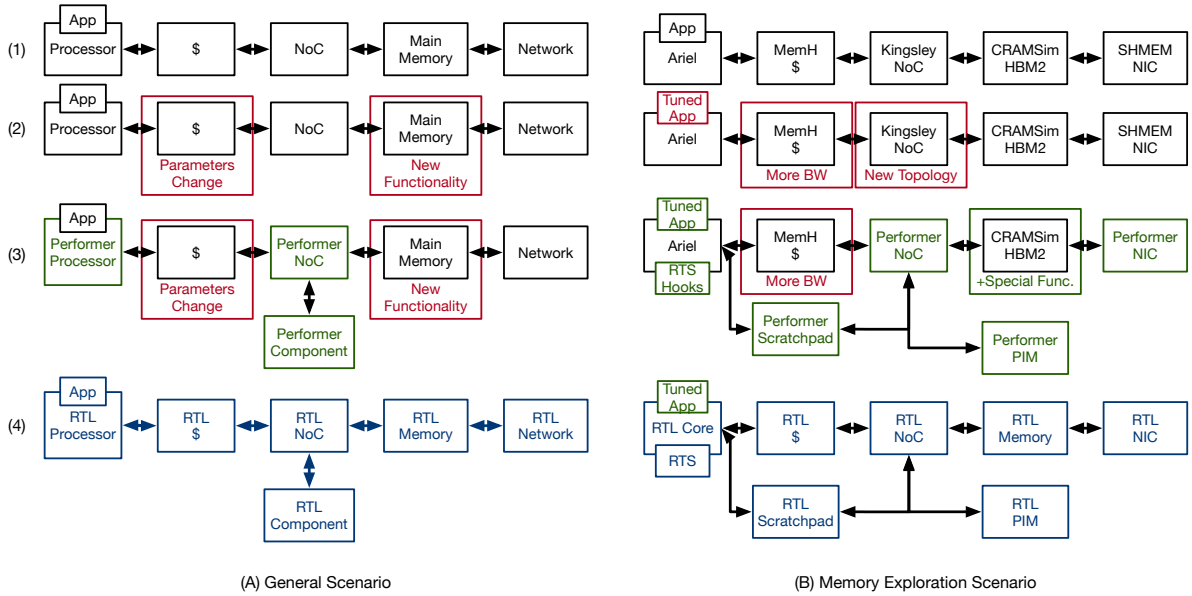
3. USAGE SCENARIOS

The goal of A-SST in the AGILE program is to provide Performers with a flexible toolkit for exploring and developing their designs. Use of the A-SST Core is required, but the exact way in which Performers will interface with A-SST will vary depending on Performer and AGILE program requirements.

3.1. General Usage Scenarios

To enhance Performer productivity, the T&E team will provide several element libraries (Section 2) of common components and A-SST configurations for executing some of the AGILE workflows and kernels on select hardware configurations. The objective is to provide Performers with an "out of the box" set of components and designs to quickly begin exploring the AGILE applications and to supply a foundation for further design. However, AGILE anticipates the need for a "Clean Sheet" design methodology. So, Performers are not required and should not feel compelled to use any of the elements libraries or configurations for their final design.

Based on previous collaborations using SST, some possible use scenarios are shown in Figure 3-1.



(A) General Scenario

(B) Memory Exploration Scenario

Figure 3-1. (A) General A-SST Usage Scenario for Performers and (B) Hypothetical Memory Exploration Example

One general model that Performers may follow:

1. **"Out of the Box"**: Starting with existing A-SST components and T&E-supplied configurations, Performers can immediately began examining the performance and execution of AGILE workflows.

2. **Customization**: Performers begin to tune and extend existing components. To discover bottlenecks and gain early insight, Performers may change parameters or add new functionality to existing components.

3. **Replacement**: As designs become more detailed, Performers replace key T&E-supplied components in the configuration with their own components and begin to add new components. Existing A-SST components are used as a 'bridge' to fundamentally new designs.

4. **RTL**: In Phase 2, Performers will replace high-level A-SST components with Performer-written low-level RTL A-SST components. T&E-supplied or Performer-written high-level components may be used with low-level RTL components as a 'bridge' during development.

5. **Multi-Fidelity & Mixed-Fidelity**: High-level and low-level simulation is critical for design and proving performance, but will be limited by simulation speed. To explore the scaling properties of Performer designs, it is anticipated that multi-fidelity and/or mixed-fidelity modeling will be required (See Section 3.2). This modeling may be performed concurrently with steps 3 (Replacement) and 4 (RTL).

To give a more concrete hypothetical example (Figure 3-1(B)):

1. **"Out of the Box"**: The Performer starts with a configuration provided by the T&E team for a given AGILE app, gathering a variety of statistics about its execution.

2. **Customization**: Based on the statistics gathered, the Performer identifies several bottlenecks. To alleviate these, they examine the effect of increased bandwidth in the memHierarchy cache and modify the Kingsley NoC with a new topology. The application, running in Ariel, is tuned for the new architecture.

3. **Replacement**: The Performer begins to elaborate their design. The tuned application is replaced with one making calls to the Performer's runtime. These calls are intercepted by a modified Ariel component. A user-controlled scratchpad is added. The Kingsley NoC is replaced with a custom NoC. The CRAMSim HBM2 model is augmented with in-memory operations that are triggered by the runtime calls. A near-memory PIM processor is added and the SHMEM NIC is replaced.

4. **RTL**: In Phase 2, the Performer begins replacing the high-level components with RTL components.

3.2. Multi-Fidelity Simulation

Ensuring adequate performance of AGILE designs at scale will require simulation. However, due to time and resource constraints it will be impossible to simulate an entire system at full scale and full detail. Instead, it will be necessary to use **Multi-Fidelity** and **Mixed-Fidelity** simulation methodologies.

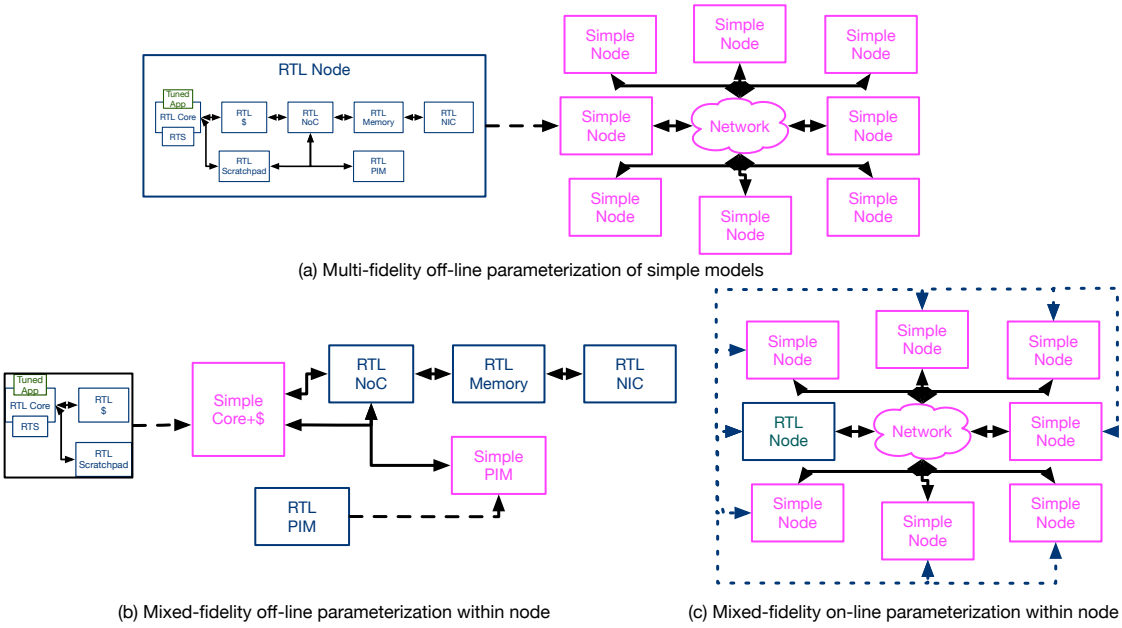


Figure 3-2. Multi-Fidelity and Multi-Resolution Usage Scenarios

Both of these methodologies combine less scalable, more detailed simulation models and faster, more abstract models to enable scalable, yet still accurate, simulation. More detailed simulations may include FPGA-based emulation.

There are a number of potential paths for multi-fidelity and mixed-fidelity simulation (Figure 3-2). The exact methodology will depend on the Performer's design. For example:

- Detailed node-level RTL models of the design are run (possibly on an FPGA) to determine performance characteristics. These characteristics are used to parameterize simpler models that are run at scale. (e.g. Figure 3-2(a))
- Detailed models of a core and cache and a near-memory processor are run (possibly on an FPGA) to determine performance characteristics. These characteristics are used to construct simple models that are run in conjunction with detailed RTL models of the NoC, memory, and NIC at larger scale. (e.g. Figure 3-2(b))
- A small number of detailed node models are run with a large number of simple node models. The detailed model is instrumented to provide a stream of updates to the simple models changing their performance characteristics to reflect different phases of the application. (e.g. Figure 3-2(c))

How these simple models will be constructed will depend on the specifics of the Performer's design. A-SST will offer a number of options. Pervious work has used the Ember / Firefly stack of components, coupled with more detailed memory models, to perform mixed-fidelity simulation.

3.3. Runtime Modeling

Modeling and simulation of the Performer's runtime system is a key area of concern. A-SST's goal is to offer a range of options to Performers to allow flexible investigation and development of AGILE runtime systems.

Runtime modeling can take advantage of several existing components:

- State Machines such as the Ember/Firefly stack can be used to generate streams of calls to a communication runtime. On-node, generators like Miranda can be used to produce a stream of memory accesses and also calls to a runtime. Because they are a simple state machine it is easier to modify these components than to build a complete execution-driven runtime.
- **Dynamic binary instrumentation** components such as Ariel or DynamoRIO allow rapid prototyping of modified applications because they can intercept runtime calls and communicate them to the simulator.
- Execution-based **processor models** such as Vanadis or Gem5 can be used for detailed performance measurements of runtime systems.

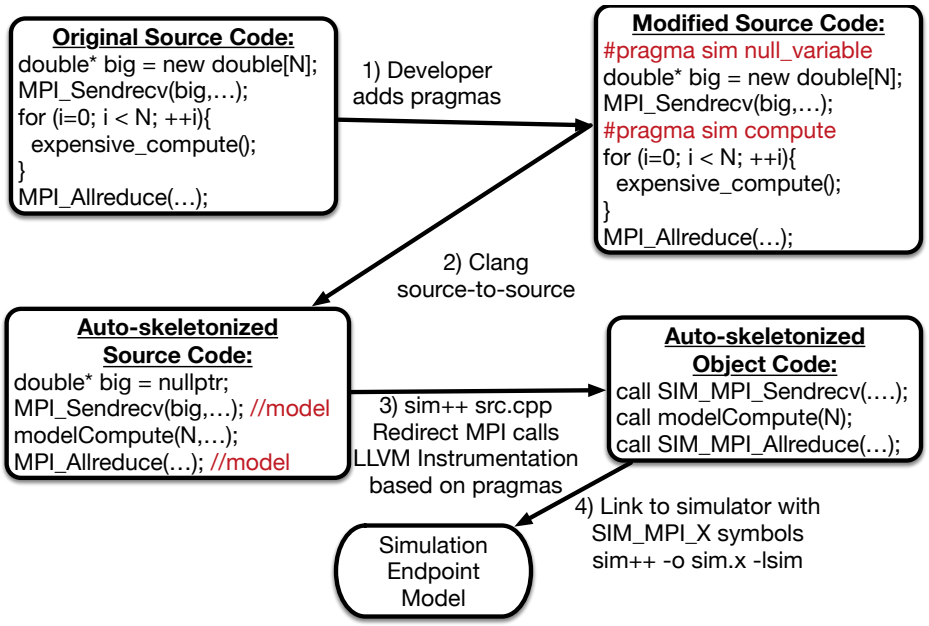


Figure 3-3. Skeletonizer

- The Macro **"Skeletonizer"** tool[19] can be used to assist runtime development (Figure 3-3). This tool provides three key features:

- Encapsulation – converting processes of the target codes into lightweight simulation threads. This provides a foundation for early runtime experimentation within the simulator before the entire toolchain or processor model is complete.
- Interception – intercepting calls and converting them to simulator actions allows key runtime calls to directly invoke simulation model primitives.
- Skeletonization – expensive computations are replaced with delay models and large allocations are removed allowing efficient, scalable simulation of key portions of a large application or workflow.

4. ADDITIONAL INFORMATION

Additional A-SST resources:

- General information about SST and A-SST and announcements can be found on the project webpage: `http://sst-simulator.org/`.
- The main SST GitHub page is `https://github.com/sstsimulator` and will contain public A-SST information when it is released. SST/A-SST issues can be reported at `http://sst-simulator.org/SSTPages/SSTMainSupport/`.
- SST Tutorials can be found at `http://sst-simulator.org/SSTPages/SSTTopDocTutorial/`. A-SST specific tutorials will be released at the project kickoff workshop.
- SST documentation is located at `http://sst-simulator.org/SSTPages/SSTMainDocumentation/` and developer-specific documentation (including SST Core's Doxygen documentation) is at `http://sst-simulator.org/SSTPages/SSTTopDocDeveloperInfo/`.
- Performers can send questions about A-SST to the SST help email reflector `wg-sst@sandia.gov` **except** during the BAA response period. While the BAA is in effect, Performer's questions should be sent to `dni-iarpa-AGILE-BAA-2021@iarpa.gov`.

REFERENCES

[1] Ieee standard for verilog hardware description language. *IEEE Std 1364-2005 (Revision of IEEE Std 1364-2001)*, pages 1–590, 2006.

[2] A. Awad, S.D. Hammond, G.R. Voskuilen, C. Hughes, A.F. Rodrigues, and R.J. Hoekstra. Messier: A detailed nvm-based dimm model for the sst simulation framework. SAND Report SAND2017-1830, Sandia National Labs, Albuquerque, New Mexico 87185, Feb 2017.

[3] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. Chisel: Constructing hardware in a scala embedded language. In *DAC Design Automation Conference 2012*, pages 1212–1221, 2012.

[4] Scott Beamer and David Donofrio. Efficiently exploiting low activity factors to accelerate rtl simulation. In *Proceedings of the 57th ACM/EDAC/IEEE Design Automation Conference*, DAC '20. IEEE Press, 2020.

[5] Derek Bruening, Timothy Garnett, and Saman Amarasinghe. An infrastructure for adaptive dynamic optimization. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*, CGO '03, page 265–275, USA, 2003. IEEE Computer Society.

[6] Tiffany A. Connors, Taylor Groves, Tony Quan, and Scott Hemmert. Simulation framework for studying optical cable failures in dragonfly topologies. In *2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 859–864, 2019.

[7] Karen Devine, Erik Boman, Lee Riesen, Umit Catalyurek, and Cédric Chevalier. Getting started with zoltan: A short tutorial. In *Dagstuhl Seminar Combinatorial Scientific Computing*, 01 2009.

[8] Simon D. Hammond, Arun F. Rodrigues, and Gwendolyn R. Voskuilen. Multi-level memory policies: What you add is more important than what you take out. In *Proceedings of the Second International Symposium on Memory Systems*, MEMSYS '16, page 88–93, New York, NY, USA, 2016. Association for Computing Machinery.

[9] Michael B. Healy and Seokin Hong. Cramsim: Controller and memory simulator. In *Proceedings of the International Symposium on Memory Systems*, MEMSYS '17, page 83–85, New York, NY, USA, 2017. Association for Computing Machinery.

[10] C. Hughes, S.D. Hammond, R.J. Hoekstra, M. Zhang, M. Khairy, and T. Rogers. Balar: A sst gpu component for performance modeling and profiling. SAND Report SAND2019-10389, Sandia National Labs, Albuquerque, New Mexico 87185, Sept 2019.

[11] A. Izraelevitz, J. Koenig, P. Li, R. Lin, A. Wang, A. Magyar, D. Kim, C. Schmidt, C. Markley, J. Lawson, and J. Bachrach. Reusability is firrtl ground: Hardware construction languages, compiler frameworks, and transformations. In *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 209–216, Nov 2017.

[12] Fulya Kaplan, Ozan Tuncer, Vitus J. Leung, Scott K. Hemmert, and Ayse K. Coskun. Unveiling the interplay between global link arrangements and network management algorithms on dragonfly networks. In *2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, pages 325–334, 2017.

[13] Mahmoud Khairy, Zhesheng Shen, Tor M. Aamodt, and Timothy G Rogers. Accel-sim: An extensible simulation framework for validated gpu modeling. In *47th IEEE/ACM International Symposium on Computer Architecture (ISCA)*, May 2020.

[14] Samuel Knight, Joseph P. Kenny, and Jeremiah J. Wilke. Supercomputer in a laptop: Distributed application and runtime development via architecture simulation. In *International Conference on High Performance Computing*. Springer, 2018.

[15] S. Li, Z. Yang, D. Reddy, A. Srivastava, and B. Jacob. Dramsim3: a cycle-accurate, thermal-capable dram simulator. In *IEEE Computer Architecture Letters*. IEEE, Feb 2020.

[16] Jason Lowe-Power, Abdul Mutaal Ahmad, Ayaz Akram, Mohammad Alian, Rico Amslinger, Matteo Andreozzi, Adrià Armejach, Nils Asmussen, Srikant Bharadwaj, Gabe Black, Gedare Bloom, Bobby R. Bruce, Daniel Rodrigues Carvalho, Jerónimo Castrillón, Lizhong Chen, Nicolas Derumigny, Stephan Diestelhorst, Wendy Elsasser, Marjan Fariborz, Amin Farmahini Farahani, Pouya Fotouhi, Ryan Gambord, Jayneel Gandhi, Dibakar Gope, Thomas Grass, Bagus Hanindhito, Andreas Hansson, Swapnil Haria, Austin Harris, Timothy Hayes, Adrian Herrera, Matthew Horsnell, Syed Ali Raza Jafri, Radhika Jagtap, Hanhwi Jang, Reiley Jeyapaul, Timothy M. Jones, Matthias Jung, Subash Kannoth, Hamidreza Khaleghzadeh, Yuetsu Kodama, Tushar Krishna, Tommaso Marinelli, Christian Menard, Andrea Mondelli, Tiago Mück, Omar Naji, Krishnendra Nathella, Hoa Nguyen, Nikos Nikoleris, Lena E. Olson, Marc S. Orr, Binh Pham, Pablo Prieto, Trivikram Reddy, Alec Roelke, Mahyar Samani, Andreas Sandberg, Javier Setoain, Boris Shingarov, Matthew D. Sinclair, Tuan Ta, Rahul Thakur, Giacomo Travaglini, Michael Upton, Nilay Vaish, Ilias Vougioukas, Zhengrong Wang, Norbert Wehn, Christian Weis, David A. Wood, Hongil Yoon, and Éder F. Zulian. The gem5 simulator: Version 20.0+. *CoRR*, abs/2007.03152, 2020.

[17] S. Nema, R. Razdan, A. Rodrigues, K. S. Hemmert, G. Voskuilen, D. Adak, S. Hammond, A. Awad, and C. Hughes. Eras: Enabling the integration of real-world intellectual properties (ips) in architectural simulators. SAND Report SAND2021-12065, Sandia National Laboratories, Albuquerque, New Mexico 87185, Sept 2021.

[18] Arun F Rodrigues, K Scott Hemmert, Brian W Barrett, Chad Kersey, Ron Oldfield, Marlo Weston, Rolf Risen, Jeanine Cook, Paul Rosenfeld, Elliot Cooper-Balis, et al. The structural simulation toolkit. *ACM SIGMETRICS Performance Evaluation Review*, 38(4):37–42, 2011.

[19] Jeremiah J. Wilke and et al. Compiler-assisted source-to-source skeletonization of application models for system simulation. In *International Conference on High Performance Computing*. Springer, 2018.

[20] Clifford Wolf. Design and implementation of the yosys open synthesis suite. Technical report, 2013.

DISTRIBUTION

Email—External (encrypt for OUO)

Name	Company Email Address	Company Name
William Harrod	william.harrod@iarpa.gov	IARPA
Sonia McCarthy	sonia.mccarthy@iarpa.gov	IARPA (Contractor)

Email—Internal (encrypt for OUO)

Name	Org.	Sandia Email Address
Simon D. Hammond	01422	sdhammo@sandia.gov
Arun Rodrigues	01422	afrodri@sandia.gov
Scott Hemmert	01422	kshemme@sandia.gov
Clayton Hughes	01422	chughes@sandia.gov
Joseph Kenny	08753	jpkenny@sandia.gov
Gwendolyn Voskeilen	01422	grvosku@sandia.gov
Robert Hoekstra	01420	rjhoeks@sandia.gov
Technical Library	1911	sanddocs@sandia.gov

Sandia National Laboratories

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