FireSim Documentation

Sagar Karandikar, Howard Mao,
Donggyu Kim, David Biancolin,
Alon Amid,
Berkeley Architecture Research

Feb 25, 2019

Getting Started:

1 FireSim Basics **3**
- 1.1 Two common use cases: . . . 3
  - 1.1.1 Single-Node Simulation, in Parallel . . . 3
  - 1.1.2 Datacenter/Cluster Simulation . . . 3
- 1.2 Other Use Cases . . . 4
- 1.3 Background/Terminology . . . 4
- 1.4 Using FireSim/The FireSim Workflow . . . 5

2 Initial Setup/Installation **7**
- 2.1 First-time AWS User Setup . . . 7
  - 2.1.1 Creating an AWS Account . . . 7
  - 2.1.2 AWS Credit at Berkeley . . . 7
  - 2.1.3 Requesting Limit Increases . . . 7
- 2.2 Configuring Required Infrastructure in Your AWS Account . . . 8
  - 2.2.1 Select a region . . . 8
  - 2.2.2 Key Setup . . . 8
  - 2.2.3 Check your EC2 Instance Limits . . . 8
  - 2.2.4 Start a t2.nano instance to run the remaining configuration commands . . . 9
  - 2.2.5 Run scripts from the t2.nano . . . 9
  - 2.2.6 Terminate the t2.nano . . . 10
  - 2.2.7 Subscribe to the AWS FPGA Developer AMI . . . 10
- 2.3 Setting up your Manager Instance . . . 10
  - 2.3.1 Launching a "Manager Instance" . . . 10
  - 2.3.2 Setting up the FireSim Repo . . . 12
  - 2.3.3 Completing Setup Using the Manager . . . 13

3 Running FireSim Simulations **15**
- 3.1 Running a Single Node Simulation . . . 15
  - 3.1.1 Building target software . . . 15
  - 3.1.2 Setting up the manager configuration . . . 16
  - 3.1.3 Launching a Simulation! . . . 18
- 3.2 Running a Cluster Simulation . . . 23
  - 3.2.1 Returning to a clean configuration . . . 23
  - 3.2.2 Building target software . . . 23
  - 3.2.3 Setting up the manager configuration . . . 24
  - 3.2.4 Launching a Simulation! . . . 26

4 Building Your Own Hardware Designs (FireSim FPGA Images) 35
4.1 Amazon S3 Setup . . . 35
4.2 Build Recipes . . . 35
4.3 Running a Build . . . 36

5 Manager Usage (the `firesim` command) 37
5.1 Overview . . . 37
5.1.1 "Inputs" to the Manager . . . 37
5.1.2 Logging . . . 37
5.2 Manager Command Line Arguments . . . 38
5.2.1 `--runtimeconfigfile FILENAME` . . . 39
5.2.2 `--buildconfigfile FILENAME` . . . 39
5.2.3 `--buildrecipesconfigfile FILENAME` . . . 39
5.2.4 `--hwdbconfigfile FILENAME` . . . 39
5.2.5 `--overrideconfigdata SECTION PARAMETER VALUE` . . . 39
5.2.6 `TASK` . . . 39
5.3 Manager Tasks . . . 39
5.3.1 `firesim managerinit` . . . 39
5.3.2 `firesim buildafi` . . . 40
5.3.3 `firesim shareagfi` . . . 40
5.3.4 `firesim launchrunfarm` . . . 41
5.3.5 `firesim terminaterunfarm` . . . 41
5.3.6 `firesim infrasetup` . . . 42
5.3.7 `firesim boot` . . . 42
5.3.8 `firesim kill` . . . 42
5.3.9 `firesim runworkload` . . . 43
5.3.10 `firesim runcheck` . . . 43
5.4 Manager Configuration Files . . . 43
5.4.1 `config_runtime.ini` . . . 43
5.4.2 `config_build.ini` . . . 47
5.4.3 `config_build_recipes.ini` . . . 49
5.4.4 `config_hwdb.ini` . . . 52
5.5 Manager Network Topology Definitions (`user_topology.py`) . . . 54
5.5.1 `user_topology.py` contents: . . . 54
5.6 AGFI Metadata/Tagging . . . 62

6 Workloads 63
6.1 Defining Custom Workloads . . . 63
6.1.1 Uniform Workload JSON . . . 64
6.1.2 Non-uniform Workload JSON (explicit job per simulated node) . . . 65
6.2 SPEC 2017 . . . 67
6.2.1 Intspeed . . . 68
6.2.2 Intrate . . . 68
6.3 Running Fedora on FireSim . . . 69
6.4 ISCA 2018 Experiments . . . 69
6.4.1 Prerequisites . . . 70
6.4.2 Building Benchmark Binaries/Rootfses . . . 70
6.4.3 Figure 5: Ping Latency vs. Configured Link Latency . . . 70
6.4.4 Figure 6: Network Bandwidth Saturation . . . 70
6.4.5 Figure 7: Memcached QoS / Thread Imbalance . . . 71
6.4.6 Figure 8: Simulation Rate vs. Scale . . . 71
6.4.7 Figure 9: Simulation Rate vs. Link Latency . . . 71
6.4.8 Running all experiments at once . . . 71
6.5 GAP Benchmark Suite . . . 72

7 FireMarshal (alpha) 73
7.1 Quick Start . . . 73
7.2 FireMarshal Commands . . . 74
7.2.1 Core Options . . . 74
7.2.2 build . . . 75
7.2.3 launch . . . 75
7.2.4 clean . . . 76
7.2.5 test . . . 76
7.2.6 install . . . 76
7.3 Workload Specification . . . 76
7.3.1 Example Configuration File . . . 76
7.3.2 Bare-Metal Workloads . . . 78
7.3.3 Configuration File Options . . . 78

8 Targets 83
8.1 Restrictions on Target-RTL . . . 83
8.2 Generating Different Target-RTL . . . 83
8.3 Rocket Chip Generator-based SoCs (firesim project) . . . 84
8.3.1 Rocket-based SoCs . . . 84
8.3.2 BOOM-based SoCs . . . 84
8.3.3 Generating A Different FASED Memory-Timing Model Instance . . . 85
8.4 Midas Examples (midasexamples project) . . . 85
8.4.1 Examples . . . 85
8.5 FASED Tests (fasedtests project) . . . 86
8.5.1 Examples . . . 86

9 Debugging 87
9.1 Debugging & Testing with RTL Simulation . . . 87
9.1.1 Target-Level Simulation . . . 88
9.1.2 MIDAS-Level Simulation . . . 88
9.1.3 FPGA-Level Simulation . . . 89
9.1.4 Scala Tests . . . 90
9.2 Debugging Using FPGA Integrated Logic Analyzers (ILA) . . . 90
9.2.1 Annotating Signals . . . 91
9.2.2 Using the ILA at Runtime . . . 91
9.3 Debugging Using TracerV . . . 91
9.3.1 Building a Design with TracerV . . . 92
9.3.2 Enabling Tracing at Runtime . . . 92
9.3.3 Interpreting the Trace Result . . . 93
9.4 Assertion Synthesis . . . 93
9.4.1 Enabling Assertion Synthesis . . . 93
9.4.2 Runtime Behavior . . . 93
9.4.3 Related Publications . . . 93
9.5 Printf Synthesis . . . 93
9.5.1 Enabling Printf Synthesis . . . 94
9.5.2 Runtime Arguments . . . 94
9.5.3 Related Publications . . . 94

10 Tutorial: Developing New Devices 95
10.1 Getting Started . . . 95
10.2 Memory-mapped Registers . . . 96
10.3 DMA and Interrupts . . . 97
10.3.1 TileLink Client Port . . . 97
10.3.2 TileLink Protocol and State Machine . . . 97

10.3.3 Interrupts . . . 99
10.4 Connecting Devices to Bus . . . 99
10.4.1 SoC Mixin Traits . . . 99
10.4.2 Top-Level Design and Configuration . . . 100
10.5 Running Test Software . . . 101
10.5.1 Debugging Verilog Simulation . . . 102
10.6 Creating Simulation Model . . . 103

11 Supernode - Multiple Simulated SoCs Per FPGA **107**
11.1 Introduction . . . 107
11.2 Building Supernode Designs . . . 107
11.3 Running Supernode Simulations . . . 108
11.4 Work in Progress! . . . 109

12 Miscellaneous Tips **111**
12.1 Add the `fsimcluster` column to your AWS management console . . . 111
12.2 FPGA Dev AMI Remote Desktop Setup . . . 111
12.3 Experimental Support for SSHing into simulated nodes and accessing the internet from within simulations . . . 112
12.4 Navigating the FireSim Codebase . . . 113

13 FireSim Asked Questions **115**
13.1 I just bumped the FireSim repository to a newer commit and simulations aren't running. What is going on? . . . 115
13.2 Is there a good way to keep track of what AGFI corresponds to what FireSim commit? . . . 115

14 Indices and tables **117**

New to FireSim? Jump to the *FireSim Basics* page for more info.

CHAPTER 1

FireSim Basics

FireSim is a cycle-accurate, FPGA-accelerated scale-out computer system simulation platform developed in the Berkeley Architecture Research Group in the EECS Department at the University of California, Berkeley.

FireSim is capable of simulating from **one to thousands of multi-core compute nodes**, derived from **silicon-proven** and **open** target-RTL, with an optional cycle-accurate network simulation tying them together. FireSim runs on FPGAs in **public cloud** environments like AWS EC2 F1, removing the high capex traditionally involved in large-scale FPGA-based simulation.

FireSim is useful both for datacenter architecture research as well as running many single-node architectural experiments in parallel on FPGAs. By harnessing a standardized host platform and providing a large amount of automation/tooling, FireSim drastically simplifies the process of building and deploying large-scale FPGA-based hardware simulations.

To learn more, see the FireSim website and the FireSim ISCA 2018 paper.

For a two-minute overview that describes how FireSim simulates a datacenter, see our ISCA 2018 lightning talk on YouTube.

1.1 Two common use cases:

1.1.1 Single-Node Simulation, in Parallel

In this mode, FireSim allows for simulation of individual Rocket Chip-based nodes without a network, which allows individual simulations to run at ~150 MHz. The FireSim manager has the ability to automatically distribute jobs to many parallel simulations, expediting the process of running large workloads like SPEC. For example, users can run all of SPECInt2017 on Rocket Chip in ~1 day by running the 10 separate workloads in parallel on 10 FPGAs.

1.1.2 Datacenter/Cluster Simulation

In this mode, FireSim also models a cycle-accurate network with parameterizeable bandwidth and link latency, as well as configurable topology, to accurately model current and future datacenter-scale systems. For example, FireSim has

been used to simulate 1024 quad-core Rocket Chip-based nodes, interconnected by a 200 Gbps, 2us network. To learn more about this use case, see our ISCA 2018 paper or two-minute lightning talk.

1.2 Other Use Cases

This release does not support a non-cycle-accurate network as our AWS Compute Blog Post/Demo used. This feature will be restored in a future release.

If you have other use-cases that we haven't covered, feel free to contact us!

1.3 Background/Terminology

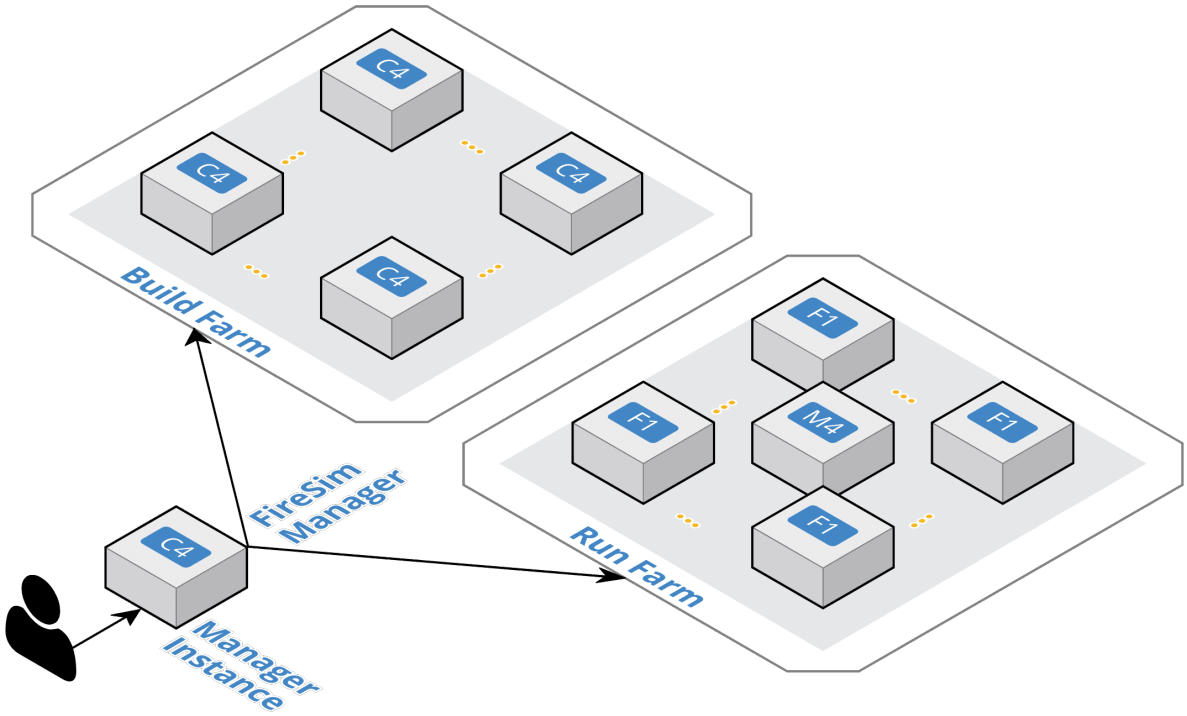

Fig. 1: FireSim Infrastructure Diagram

FireSim Manager (`firesim`) This program (available on your path as `firesim` once we source necessary scripts) automates the work required to launch FPGA builds and run simulations. Most users will only have to interact with the manager most of the time. If you're familiar with tools like Vagrant or Docker, the `firesim` command is just like the `vagrant` and `docker` commands, but for FPGA simulators instead of VMs/containers.

Manager Instance This is the AWS EC2 instance that you will SSH-into and do work on. This is where you'll clone your copy of FireSim and use the FireSim Manager to deploy builds/simulations from.

Build Farm These are instances that are elastically started/terminated by the FireSim manager when you run FPGA builds. The manager will automatically ship source for builds to these instances and run the Verilog -> FPGA Image process on them.

Run Farm These are a tagged collection of F1 (and M4) instances that the manager automatically launches and deploys simulations onto. You can launch multiple Run Farms in parallel, each with their own tag, to run multiple separate simulations in parallel.

To disambiguate between the computers being simulated and the computers doing the simulating, we also define:

Target The design and environment under simulation. Generally, a group of one or more multi-core RISC-V microprocessors with or without a network between them.

Host The computers executing the FireSim simulation – the **Run Farm** from above.

We frequently prefix words with these terms. For example, software can run on the simulated RISC-V system (*target*-software) or on a host x86 machine (*host*-software).

1.4 Using FireSim/The FireSim Workflow

The tutorials that follow this page will guide you through the complete flow for getting an example FireSim simulation up and running. At the end of this tutorial, you'll have a simulation that simulates a single quad-core Rocket Chip-based node with a 4 MB last level cache, 16 GB DDR3, and no NIC. After this, you can continue to a tutorial that shows you how to simulate a globally-cycle-accurate cluster-scale FireSim simulation. The final tutorial will show you how to build your own FPGA images with customized hardware. After you complete these tutorials, you can look at the Advanced documentation in the sidebar to the left.

Here's a high-level outline of what we'll be doing in our tutorials:

1. **Initial Setup/Installation**
   (a) First-time AWS User Setup: You can skip this if you already have an AWS account/payment method set up.
   (b) Configuring required AWS resources in your account: This sets up the appropriate VPCs/subnets/security groups required to run FireSim.
   (c) Setting up a "Manager Instance" from which you will coordinate building and deploying simulations.
2. **Single-node simulation tutorial**: This tutorial guides you through the process of running one simulation on a Run Farm consisting of a single `f1.2xlarge`, using our pre-built public FireSim AGFIs.
3. **Cluster simulation tutorial**: This tutorial guides you through the process of running an 8-node cluster simulation on a Run Farm consisting of one `f1.16xlarge`, using our pre-built public FireSim AGFIs and switch models.
4. **Building your own hardware designs tutorial (Chisel to FPGA Image)**: This tutorial guides you through the full process of taking Rocket Chip RTL and any custom RTL plugged into Rocket Chip and producing a FireSim AGFI to plug into your simulations. This automatically runs Chisel elaboration, FAME-1 Transformation, and the Vivado FPGA flow.

Generally speaking, you only need to follow step 4 if you're modifying Chisel RTL or changing non-runtime configurable hardware parameters.

Now, hit next to proceed with setup.

CHAPTER 2

Initial Setup/Installation

This section will guide you through initial setup of your AWS account to support FireSim, as well as cloning/installing FireSim on your manager instance.

2.1 First-time AWS User Setup

If you've never used AWS before and don't have an account, follow the instructions below to get started.

2.1.1 Creating an AWS Account

First, you'll need an AWS account. Create one by going to aws.amazon.com and clicking "Sign Up." You'll want to create a personal account. You will have to give it a credit card number.

2.1.2 AWS Credit at Berkeley

If you're an internal user at Berkeley and affiliated with UCB-BAR or the RISE Lab, see the RISE Lab Wiki for instructions on getting access to the AWS credit pool. Otherwise, continue with the following section.

2.1.3 Requesting Limit Increases

In our experience, new AWS accounts do not have access to EC2 F1 instances by default. In order to get access, you should file a limit increase request.

Follow these steps to do so:

https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/ec2-resource-limits.html

You'll probably want to start out with the following requests, depending on your existing limits:

Request 1:

```
Region:                 US East (Northern Virginia)
Primary Instance Type: f1.2xlarge
Limit:                  Instance Limit
New limit value:        1
```

Request 2:

```
Region:                 US East (Northern Virginia)
Primary Instance Type: f1.16xlarge
Limit:                  Instance Limit
New limit value:        1
```

This allows you to run one node on the `f1.2xlarge` or eight nodes on the `f1.16xlarge`.

For the "Use Case Description", you should describe your project and write something about hardware simulation and mention that information about the tool you're using can be found at: https://fires.im

This process has a human in the loop, so you should submit it ASAP. At this point, you should wait for the response to this request.

If you're at Berkeley/UCB-BAR, you also need to wait until your account has been added to the RISE billing pool, otherwise your personal CC will be charged for AWS usage.

Hit Next below to continue.

2.2 Configuring Required Infrastructure in Your AWS Account

Once we have an AWS Account setup, we need to perform some advance setup of resources on AWS. You will need to follow these steps even if you already had an AWS account as these are FireSim-specific.

2.2.1 Select a region

Head to the EC2 Management Console. In the top right corner, ensure that the correct region is selected. You should select one of: `us-east-1` (N. Virginia), `us-west-2` (Oregon), or `eu-west-1` (Ireland), since F1 instances are only available in those regions.

Once you select a region, it's useful to bookmark the link to the EC2 console, so that you're always sent to the console for the correct region.

2.2.2 Key Setup

In order to enable automation, you will need to create a key named `firesim`, which we will use to launch all instances (Manager Instance, Build Farm, Run Farm).

To do so, click "Key Pairs" under "Network & Security" in the left-sidebar. Follow the prompts, name the key `firesim`, and save the private key locally as `firesim.pem`. You can use this key to access all instances from your local machine. We will copy this file to our manager instance later, so that the manager can also use it.

2.2.3 Check your EC2 Instance Limits

AWS limits access to particular instance types for new/infrequently used accounts to protect their infrastructure. You should make sure that your account has access to `f1.2xlarge`, `f1.4xlarge`, `f1.16xlarge`, `m4.16xlarge`, and `c4.4xlarge` instances by looking at the "Limits" page in the EC2 panel, which you can access here. The values

listed on this page represent the maximum number of any of these instances that you can run at once, which will limit the size of simulations (# of nodes) that you can run. If you need to increase your limits, follow the instructions on the *Requesting Limit Increases* page. To follow this guide, you need to be able to run one `f1.2xlarge` instance and two `c4.4xlarge` instances.

2.2.4 Start a t2.nano instance to run the remaining configuration commands

To avoid having to deal with the messy process of installing packages on your local machine, we will spin up a very cheap `t2.nano` instance to run a series of one-time aws configuration commands to setup our AWS account for FireSim. At the end of these instructions, we'll terminate the `t2.nano` instance. If you happen to already have `boto3` and the AWS CLI installed on your local machine, you can do this locally.

Launch a `t2.nano` by following these instructions:

1. Go to the EC2 Management Console and click "Launch Instance"
2. On the AMI selection page, select "Amazon Linux AMI. . . ", which should be the top option.
3. On the Choose an Instance Type page, select `t2.nano`.
4. Click "Review and Launch" (we don't need to change any other settings)
5. On the review page, click "Launch"
6. Select the `firesim` key pair we created previously, then click Launch Instances.
7. Click on the instance name and note its public IP address.

2.2.5 Run scripts from the t2.nano

SSH into the `t2.nano` like so:

```
ssh -i firesim.pem ec2-user@INSTANCE_PUBLIC_IP
```

Which should present you with something like:

```
Last login: Mon Feb 12 21:11:27 2018 from 136.152.143.34

       __|  __|_  )
       _|  (     /   Amazon Linux AMI
      ___|\___|___|

https://aws.amazon.com/amazon-linux-ami/2017.09-release-notes/
4 package(s) needed for security, out of 5 available
Run "sudo yum update" to apply all updates.
[ec2-user@ip-172-30-2-66 ~]$
```

On this machine, run the following:

```
aws configure
[follow prompts]
```

See https://docs.aws.amazon.com/cli/latest/userguide/tutorial-ec2-ubuntu.html#configure-cli-launch-ec2 for more about aws configure. You should specify the same region that you chose above (one of `us-east-1`, `us-west-2`, `eu-west-1`) and set the default output format to `json`.

Again on the `t2.nano` instance, do the following:

```
sudo yum -y install python-pip
sudo pip install boto3
wget https://raw.githubusercontent.com/firesim/firesim/master/scripts/aws-setup.py
python aws-setup.py
```

This will create a VPC named `firesim` and a security group named `firesim` in your account.

2.2.6 Terminate the t2.nano

At this point, we are finished with the general account configuration. You should terminate the t2.nano instance you created, since we do not need it anymore (and it shouldn't contain any important data).

2.2.7 Subscribe to the AWS FPGA Developer AMI

Go to the AWS Marketplace page for the FPGA Developer AMI. Click the button to subscribe to the FPGA Dev AMI (it should be free) and follow the prompts to accept the EULA (but do not launch any instances).

Now, hit next to continue on to setting up our Manager Instance.

2.3 Setting up your Manager Instance

2.3.1 Launching a "Manager Instance"

Now, we need to launch a "Manager Instance" that acts as a "head" node that we will `ssh` or `mosh` into to work from. Since we will deploy the heavy lifting to separate `c4.4xlarge` and `f1` instances later, the Manager Instance can be a relatively cheap instance. In this guide, however, we will use a `c4.4xlarge`, running the AWS FPGA Developer AMI (be sure to subscribe if you have not done so. See *Subscribe to the AWS FPGA Developer AMI*).

Head to the EC2 Management Console. In the top right corner, ensure that the correct region is selected.

To launch a manager instance, follow these steps:

1. From the main page of the EC2 Management Console, click `Launch Instance`. We use an on-demand instance here, so that your data is preserved when you stop/start the instance, and your data is not lost when pricing spikes on the spot market.
2. When prompted to select an AMI, search in the `Community AMIs` tab for "FPGA" and select the option that starts with `FPGA Developer AMI - 1.5.0`. **DO NOT USE ANY OTHER VERSION.**
3. When prompted to choose an instance type, select the instance type of your choosing. A good choice is a `c4.4xlarge`.
4. On the "Configure Instance Details" page:
   (a) First make sure that the `firesim` VPC is selected in the drop-down box next to "Network". Any subnet within the `firesim` VPC is fine.
   (b) Additionally, check the box for "Protect against accidental termination." This adds a layer of protection to prevent your manager instance from being terminated by accident. You will need to disable this setting before being able to terminate the instance using usual methods.
   (c) Also on this page, expand "Advanced Details" and in the resulting text box, paste the following:

```
#!/bin/bash
echo "machine launch script started" > /home/centos/machine-launchstatus
sudo yum install -y mosh
sudo yum groupinstall -y "Development tools"
sudo yum install -y gmp-devel mpfr-devel libmpc-devel zlib-devel vim git java␣
↪java-devel
curl https://bintray.com/sbt/rpm/rpm | sudo tee /etc/yum.repos.d/bintray-sbt-
↪rpm.repo
sudo yum install -y sbt texinfo gengetopt
sudo yum install -y expat-devel libusb1-devel ncurses-devel cmake
↪"perl(ExtUtils::MakeMaker)"
# deps for poky
sudo yum install -y python34 patch diffstat texi2html texinfo subversion␣
↪chrpath git wget
# deps for qemu
sudo yum install -y gtk3-devel
# deps for firesim-software (note that rsync is installed but too old)
sudo yum install -y python34-pip python34-devel rsync
# install DTC. it's not available in repos in FPGA AMI
DTCversion=dtc-1.4.4
wget https://git.kernel.org/pub/scm/utils/dtc/dtc.git/snapshot/$DTCversion.
↪tar.gz
tar -xvf $DTCversion.tar.gz
cd $DTCversion
make -j16
make install
cd ..
rm -rf $DTCversion.tar.gz
rm -rf $DTCversion

# get a proper version of git
sudo yum -y remove git
sudo yum -y install epel-release
sudo yum -y install https://centos7.iuscommunity.org/ius-release.rpm
sudo yum -y install git2u

# install verilator
git clone http://git.veripool.org/git/verilator
cd verilator/
git checkout v4.002
autoconf && ./configure && make -j16 && sudo make install
cd ..

# bash completion for manager
sudo yum -y install bash-completion

# graphviz for manager
sudo yum -y install graphviz python-devel

# these need to match what's in deploy/requirements.txt
sudo pip2 install fabric==1.14.0
sudo pip2 install boto3==1.6.2
sudo pip2 install colorama==0.3.7
sudo pip2 install argcomplete==1.9.3
sudo pip2 install graphviz==0.8.3
# for some of our workload plotting scripts
sudo pip2 install --upgrade --ignore-installed pyparsing
```

(continues on next page)

(continued from previous page)

```
sudo pip2 install matplotlib==2.2.2
sudo pip2 install pandas==0.22.0
# this is explicitly installed to downgrade it to a version without deprec␣
↪warnings
sudo pip2 install cryptography==2.2.2

sudo activate-global-python-argcomplete

# get a regular prompt
echo "PS1='\u@\H:\w\\$ '" >> /home/centos/.bashrc
echo "machine launch script completed" >> /home/centos/machine-launchstatus
```

This will pre-install all of the dependencies needed to run FireSim on your instance.

5. On the next page ("Add Storage"), increase the size of the root EBS volume to ~300GB. The default of 150GB can quickly become tight as you accumulate large Vivado reports/outputs, large waveforms, XSim outputs, and large root filesystems for simulations. You can get rid of the small (5GB) secondary volume that is added by default.
6. You can skip the "Add Tags" page, unless you want tags.
7. On the "Configure Security Group" page, select the `firesim` security group that was automatically created for you earlier.
8. On the review page, click the button to launch your instance.

Make sure you select the `firesim` key pair that we setup earlier.

Access your instance

We **HIGHLY** recommend using mosh instead of `ssh` or using `ssh` with a screen/tmux session running on your manager instance to ensure that long-running jobs are not killed by a bad network connection to your manager instance. On this instance, the `mosh` server is installed as part of the setup script we pasted before, so we need to first ssh into the instance and make sure the setup is complete.

In either case, `ssh` into your instance (e.g. `ssh -i firesim.pem centos@YOUR_INSTANCE_IP`) and wait until the `~/machine-launchstatus` file contains all the following text:

```
centos@ip-172-30-2-140.us-west-2.compute.internal:~$ cat machine-launchstatus
machine launch script started
machine launch script completed!
```

Once this line appears, exit and re-`ssh` into the system. If you want to use `mosh`, `mosh` back into the system.

Key Setup, Part 2

Now that our manager instance is started, copy the private key that you downloaded from AWS earlier (`firesim.pem`) to `~/firesim.pem` on your manager instance. This step is required to give the manager access to the instances it launches for you.

2.3.2 Setting up the FireSim Repo

We're finally ready to fetch FireSim's sources. Run:

```
git clone https://github.com/firesim/firesim
cd firesim
./build-setup.sh fast
```

This will have initialized submodules and installed the RISC-V tools and other dependencies.

Next, run:

```
source sourceme-f1-manager.sh
```

This will have initialized the AWS shell, added the RISC-V tools to your path, and started an `ssh-agent` that supplies `~/firesim.pem` automatically when you use `ssh` to access other nodes. Sourcing this the first time will take some time – however each time after that should be instantaneous. Also, if your `firesim.pem` key requires a passphrase, you will be asked for it here and `ssh-agent` should cache it.

Every time you login to your manager instance to use FireSim, you should ‘‘cd‘‘ into your firesim directory and source this file again.

2.3.3 Completing Setup Using the Manager

The FireSim manager contains a command that will interactively guide you through the rest of the FireSim setup process. To run it, do the following:

```
firesim managerinit
```

This will first prompt you to setup AWS credentials on the instance, which allows the manager to automatically manage build/simulation nodes. See https://docs.aws.amazon.com/cli/latest/userguide/tutorial-ec2-ubuntu.html#configure-cli-launch-ec2 for more about these credentials. When prompted, you should specify the same region that you chose above and set the default output format to `json`.

Next, it will create initial configuration files, which we will edit in later sections. Finally, it will prompt you for an email address, which is used to send email notifications upon FPGA build completion and optionally for workload completion. You can leave this blank if you do not wish to receive any notifications, but this is not recommended.

Now you’re ready to launch FireSim simulations! Hit Next to learn how to run single-node simulations.

CHAPTER 3

Running FireSim Simulations

These guides will walk you through running two kinds of simulations:

- First, we will simulate a single-node, non-networked target, using a pre-generated hardware image.
- Then, we will simulate an eight-node, networked cluster target, also using a pre-generated hardware image.

Hit next to get started!

3.1 Running a Single Node Simulation

Now that we've completed the setup of our manager instance, it's time to run a simulation! In this section, we will simulate **1 target node**, for which we will need a single `f1.2xlarge` (1 FPGA) instance.

Make sure you are `ssh` or `mosh`'d into your manager instance and have sourced `sourceme-f1-manager.sh` before running any of these commands.

3.1.1 Building target software

In these instructions, we'll assume that you want to boot Linux on your simulated node. To do so, we'll need to build our FireSim-compatible RISC-V Linux distro. For this tutorial, we will use a simple buildroot-based distribution. You can do this like so:

```
cd firesim/sw/firesim-software
./marshal -v build workloads/br-base.json
```

This process will take about 10 to 15 minutes on a `c4.4xlarge` instance. Once this is completed, you'll have the following files:

- `firesim/sw/firesim-software/images/br-disk-bin` - a bootloader + Linux kernel image for the nodes we will simulate.
- `firesim/sw/firesim-software/images/br-disk.img` - a disk image for each the nodes we will simulate

These files will be used to form base images to either build more complicated workloads (see the *Defining Custom Workloads* section) or to copy around for deploying.

3.1.2 Setting up the manager configuration

All runtime configuration options for the manager are set in a file called `firesim/deploy/config_runtime.ini`. In this guide, we will explain only the parts of this file necessary for our purposes. You can find full descriptions of all of the parameters in the *Manager Configuration Files* section.

If you open up this file, you will see the following default config (assuming you have not modified it):

```
# RUNTIME configuration for the FireSim Simulation Manager
# See docs/Advanced-Usage/Manager/Manager-Configuration-Files.rst for documentation
↪of all of these params.

[runfarm]
runfarmtag=mainrunfarm

f1_16xlarges=1
m4_16xlarges=0
f1_4xlarges=0
f1_2xlarges=0

runinstancemarket=ondemand
spotinterruptionbehavior=terminate
spotmaxprice=ondemand

[targetconfig]
topology=example_8config
no_net_num_nodes=2
linklatency=6405
switchinglatency=10
netbandwidth=200
profileinterval=-1

# This references a section from config_hwconfigs.ini
# In homogeneous configurations, use this to set the hardware config deployed
# for all simulators
defaulthwconfig=firesim-quadcore-nic-ddr3-llc4mb

[tracing]
enable=no
startcycle=0
endcycle=-1

[workload]
workloadname=linux-uniform.json
terminateoncompletion=no
```

We'll need to modify a couple of these lines.

First, let's tell the manager to use the correct numbers and types of instances. You'll notice that in the `[runfarm]` section, the manager is configured to launch a Run Farm named `mainrunfarm`, consisting of one `f1.16xlarge` and no `m4.16xlarges`, `f1.4xlarges`, or `f1.2xlarges`. The tag specified here allows the manager to differentiate amongst many parallel run farms (each running a workload) that you may be operating – but more on that later.

Since we only want to simulate a single node, let's switch to using one `f1.2xlarge` and no `f1.16xlarges`. To do so, change this section to:

```
[runfarm]
# per aws restrictions, this tag cannot be longer than 255 chars
runfarmtag=mainrunfarm
f1_16xlarges=0
f1_4xlarges=0
m4_16xlarges=0
f1_2xlarges=1
```

You'll see other parameters here, like `runinstancemarket`, `spotinterruptionbehavior`, and `spotmaxprice`. If you're an experienced AWS user, you can see what these do by looking at the *Manager Configuration Files* section. Otherwise, don't change them.

Now, let's change the `[targetconfig]` section to model the correct target design. By default, it is set to model an 8-node cluster with a cycle-accurate network. Instead, we want to model a single-node with no network. To do so, we will need to change a few items in this section:

```
[targetconfig]
topology=no_net_config
no_net_num_nodes=1
linklatency=6405
switchinglatency=10
netbandwidth=200

# This references a section from config_hwconfigs.ini
# In homogeneous configurations, use this to set the hardware config deployed
# for all simulators
defaulthwconfig=firesim-quadcore-no-nic-ddr3-llc4mb
```

Note that we changed three of the parameters here: `topology` is now set to `no_net_config`, indicating that we do not want a network. Then, `no_net_num_nodes` is set to `1`, indicating that we only want to simulate one node. Lastly, we changed `defaulthwconfig` from `firesim-quadcore-nic-ddr3-llc4mb` to `firesim-quadcore-no-nic-ddr3-llc4mb`. Notice the subtle difference in this last option? All we did is switch to a hardware configuration that does not have a NIC. This hardware configuration models a Quad-core Rocket Chip with 4 MB of L2 cache and 16 GB of DDR3, and **no** network interface card.

We will leave the last section (`[workload]`) unchanged here, since we do want to run the buildroot-based Linux on our simulated system. The `terminateoncompletion` feature is an advanced feature that you can learn more about in the *Manager Configuration Files* section.

As a final sanity check, your `config_runtime.ini` file should now look like this:

```
# RUNTIME configuration for the FireSim Simulation Manager
# See docs/Configuration-Details.rst for documentation of all of these params.

[runfarm]
runfarmtag=mainrunfarm

f1_16xlarges=0
f1_4xlarges=1
m4_16xlarges=0
f1_2xlarges=1

runinstancemarket=ondemand
spotinterruptionbehavior=terminate
spotmaxprice=ondemand
```

(continues on next page)

(continued from previous page)

```
[targetconfig]
topology=no_net_config
no_net_num_nodes=1
linklatency=6405
switchinglatency=10
netbandwidth=200

# This references a section from config_hwconfigs.ini
# In homogeneous configurations, use this to set the hardware config deployed
# for all simulators
defaulthwconfig=firesim-quadcore-no-nic-ddr3-llc4mb

[workload]
workloadname=linux-uniform.json
terminateoncompletion=no
```

Attention: [Advanced users] Simulating BOOM instead of Rocket Chip: If you would like to simulate a single-core BOOM as a target, set `defaulthwconfig` to `fireboom-singlecore-no-nic-ddr3-llc4mb`.

3.1.3 Launching a Simulation!

Now that we've told the manager everything it needs to know in order to run our single-node simulation, let's actually launch an instance and run it!

Starting the Run Farm

First, we will tell the manager to launch our Run Farm, as we specified above. When you do this, you will start getting charged for the running EC2 instances (in addition to your manager).

To do launch your run farm, run:

```
firesim launchrunfarm
```

You should expect output like the following:

```
centos@ip-172-30-2-111.us-west-2.compute.internal:~/firesim-new/deploy$ firesim␣
↪launchrunfarm
FireSim Manager. Docs: http://docs.fires.im
Running: launchrunfarm

Waiting for instance boots: f1.16xlarges
Waiting for instance boots: f1.4xlarges
Waiting for instance boots: m4.16xlarges
Waiting for instance boots: f1.2xlarges
i-0d6c29ac507139163 booted!
The full log of this run is:
/home/centos/firesim-new/deploy/logs/2018-05-19--00-19-43-launchrunfarm-
↪B4Q2ROAK0JN9EDE4.log
```

The output will rapidly progress to `Waiting for instance boots: f1.2xlarges` and then take a minute or two while your `f1.2xlarge` instance launches. Once the launches complete, you should see the instance id

printed and the instance will also be visible in your AWS EC2 Management console. The manager will tag the instances launched with this operation with the value you specified above as the `runfarmtag` parameter from the `config_runtime.ini` file, which we left set as `mainrunfarm`. This value allows the manager to tell multiple Run Farms apart – i.e., you can have multiple independent Run Farms running different workloads/hardware configurations in parallel. This is detailed in the *Manager Configuration Files* and the *firesim launchrunfarm* sections – you do not need to be familiar with it here.

Setting up the simulation infrastructure

The manager will also take care of building and deploying all software components necessary to run your simulation. The manager will also handle flashing FPGAs. To tell the manager to setup our simulation infrastructure, let's run:

```
firesim infrasetup
```

For a complete run, you should expect output like the following:

```
centos@ip-172-30-2-111.us-west-2.compute.internal:~/firesim-new/deploy$ firesim␣
↪infrasetup
FireSim Manager. Docs: http://docs.fires.im
Running: infrasetup

Building FPGA software driver for FireSimNoNIC-FireSimRocketChipQuadCoreConfig-
↪FireSimDDR3FRFCFSLLC4MBConfig
[172.30.2.174] Executing task 'instance_liveness'
[172.30.2.174] Checking if host instance is up...
[172.30.2.174] Executing task 'infrasetup_node_wrapper'
[172.30.2.174] Copying FPGA simulation infrastructure for slot: 0.
[172.30.2.174] Installing AWS FPGA SDK on remote nodes.
[172.30.2.174] Unloading XDMA/EDMA/XOCL Driver Kernel Module.
[172.30.2.174] Copying AWS FPGA XDMA driver to remote node.
[172.30.2.174] Loading XDMA Driver Kernel Module.
[172.30.2.174] Clearing FPGA Slot 0.
[172.30.2.174] Flashing FPGA Slot: 0 with agfi: agfi-0eaa90f6bb893c0f7.
[172.30.2.174] Unloading XDMA/EDMA/XOCL Driver Kernel Module.
[172.30.2.174] Loading XDMA Driver Kernel Module.
The full log of this run is:
/home/centos/firesim-new/deploy/logs/2018-05-19--00-32-02-infrasetup-9DJJCX29PF4GAIVL.
↪log
```

Many of these tasks will take several minutes, especially on a clean copy of the repo. The console output here contains the "user-friendly" version of the output. If you want to see detailed progress as it happens, `tail -f` the latest logfile in `firesim/deploy/logs/`.

At this point, the `f1.2xlarge` instance in our Run Farm has all the infrastructure necessary to run a simulation.

So, let's launch our simulation!

Running a simulation!

Finally, let's run our simulation! To do so, run:

```
firesim runworkload
```

This command boots up a simulation and prints out the live status of the simulated nodes every 10s. When you do this, you will initially see output like:

```
centos@ip-172-30-2-111.us-west-2.compute.internal:~/firesim-new/deploy$ firesim␣
↪runworkload
FireSim Manager. Docs: http://docs.fires.im
Running: runworkload

Creating the directory: /home/centos/firesim-new/deploy/results-workload/2018-05-19--
↪00-38-52-linux-uniform/
[172.30.2.174] Executing task 'instance_liveness'
[172.30.2.174] Checking if host instance is up...
[172.30.2.174] Executing task 'boot_simulation_wrapper'
[172.30.2.174] Starting FPGA simulation for slot: 0.
[172.30.2.174] Executing task 'monitor_jobs_wrapper'
```

If you don't look quickly, you might miss it, since it will get replaced with a live status page:

```
FireSim Simulation Status @ 2018-05-19 00:38:56.062737
--------------------------------------------------------------------------------
This workload's output is located in:
/home/centos/firesim-new/deploy/results-workload/2018-05-19--00-38-52-linux-uniform/
This run's log is located in:
/home/centos/firesim-new/deploy/logs/2018-05-19--00-38-52-runworkload-
↪JS5IGTV166X169DZ.log
This status will update every 10s.
--------------------------------------------------------------------------------
Instances
--------------------------------------------------------------------------------
Instance IP:   172.30.2.174 | Terminated: False
--------------------------------------------------------------------------------
Simulated Switches
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
Simulated Nodes/Jobs
--------------------------------------------------------------------------------
Instance IP:   172.30.2.174 | Job: linux-uniform0 | Sim running: True
--------------------------------------------------------------------------------
Summary
--------------------------------------------------------------------------------
1/1 instances are still running.
1/1 simulations are still running.
--------------------------------------------------------------------------------
```

This will only exit once all of the simulated nodes have shut down. So, let's let it run and open another ssh connection to the manager instance. From there, `cd` into your firesim directory again and `source sourceme-f1-manager.sh` again to get our ssh key setup. To access our simulated system, ssh into the IP address being printed by the status page, **from your manager instance**. In our case, from the above output, we see that our simulated system is running on the instance with IP `172.30.2.174`. So, run:

```
[RUN THIS ON YOUR MANAGER INSTANCE!]
ssh 172.30.2.174
```

This will log you into the instance running the simulation. Then, to attach to the console of the simulated system, run:

```
screen -r fsim0
```

Voila! You should now see Linux booting on the simulated system and then be prompted with a Linux login prompt, like so:

```
[truncated Linux boot output]
[    0.020000] VFS: Mounted root (ext2 filesystem) on device 254:0.
[    0.020000] devtmpfs: mounted
[    0.020000] Freeing unused kernel memory: 140K
[    0.020000] This architecture does not have kernel memory protection.
mount: mounting sysfs on /sys failed: No such device
Starting logging: OK
Starting mdev...
mdev: /sys/dev: No such file or directory
modprobe: can't change directory to '/lib/modules': No such file or directory
Initializing random number generator... done.
Starting network: ip: SIOCGIFFLAGS: No such device
ip: can't find device 'eth0'
FAIL
Starting dropbear sshd: OK

Welcome to Buildroot
buildroot login:
```

You can ignore the messages about the network – that is expected because we are simulating a design without a NIC.

Now, you can login to the system! The username is `root` and the password is `firesim`. At this point, you should be presented with a regular console, where you can type commands into the simulation and run programs. For example:

```
Welcome to Buildroot
buildroot login: root
Password:
# uname -a
Linux buildroot 4.15.0-rc6-31580-g9c3074b5c2cd #1 SMP Thu May 17 22:28:35 UTC 2018␣
↪riscv64 GNU/Linux
#
```

At this point, you can run workloads as you'd like. To finish off this tutorial, let's poweroff the simulated system and see what the manager does. To do so, in the console of the simulated system, run `poweroff -f`:

```
Welcome to Buildroot
buildroot login: root
Password:
# uname -a
Linux buildroot 4.15.0-rc6-31580-g9c3074b5c2cd #1 SMP Thu May 17 22:28:35 UTC 2018␣
↪riscv64 GNU/Linux
# poweroff -f
```

You should see output like the following from the simulation console:

```
# poweroff -f
[   12.456000] reboot: Power down
Power off
time elapsed: 468.8 s, simulation speed = 88.50 MHz
*** PASSED *** after 41492621244 cycles
Runs 41492621244 cycles
[PASS] FireSimNoNIC Test
SEED: 1526690334
Script done, file is uartlog

[screen is terminating]
```

You'll also notice that the manager polling loop exited! You'll see output like this from the manager:

```
FireSim Simulation Status @ 2018-05-19 00:46:50.075885
--------------------------------------------------------------------------------
This workload's output is located in:
/home/centos/firesim-new/deploy/results-workload/2018-05-19--00-38-52-linux-uniform/
This run's log is located in:
/home/centos/firesim-new/deploy/logs/2018-05-19--00-38-52-runworkload-
↪JS5IGTV166X169DZ.log
This status will update every 10s.
--------------------------------------------------------------------------------
Instances
--------------------------------------------------------------------------------
Instance IP:   172.30.2.174 | Terminated: False
--------------------------------------------------------------------------------
Simulated Switches
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
Simulated Nodes/Jobs
--------------------------------------------------------------------------------
Instance IP:   172.30.2.174 | Job: linux-uniform0 | Sim running: False
--------------------------------------------------------------------------------
Summary
--------------------------------------------------------------------------------
1/1 instances are still running.
0/1 simulations are still running.
--------------------------------------------------------------------------------
FireSim Simulation Exited Successfully. See results in:
/home/centos/firesim-new/deploy/results-workload/2018-05-19--00-38-52-linux-uniform/
The full log of this run is:
/home/centos/firesim-new/deploy/logs/2018-05-19--00-38-52-runworkload-
↪JS5IGTV166X169DZ.log
```

If you take a look at the workload output directory given in the manager output (in this case, `/home/centos/firesim-new/deploy/results-workload/2018-05-19--00-38-52-linux-uniform/`), you'll see the following:

```
centos@ip-172-30-2-111.us-west-2.compute.internal:~/firesim-new/deploy/results-
↪workload/2018-05-19--00-38-52-linux-uniform$ ls -la */*
-rw-rw-r-- 1 centos centos  797 May 19 00:46 linux-uniform0/memory_stats.csv
-rw-rw-r-- 1 centos centos  125 May 19 00:46 linux-uniform0/os-release
-rw-rw-r-- 1 centos centos 7316 May 19 00:46 linux-uniform0/uartlog
```

What are these files? They are specified to the manager in a configuration file (`firesim/deploy/workloads/linux-uniform.json`) as files that we want automatically copied back to our manager after we run a simulation, which is useful for running benchmarks automatically. The *Defining Custom Workloads* section describes this process in detail.

For now, let's wrap-up our tutorial by terminating the `f1.2xlarge` instance that we launched. To do so, run:

```
firesim terminaterunfarm
```

Which should present you with the following:

```
centos@ip-172-30-2-111.us-west-2.compute.internal:~/firesim-new/deploy$ firesim␣
↪terminaterunfarm
FireSim Manager. Docs: http://docs.fires.im
Running: terminaterunfarm
```

(continues on next page)

(continued from previous page)

```
IMPORTANT!: This will terminate the following instances:
f1.16xlarges
[]
f1.4xlarges
[]
m4.16xlarges
[]
f1.2xlarges
['i-0d6c29ac507139163']
Type yes, then press enter, to continue. Otherwise, the operation will be cancelled.
```

You must type `yes` then hit enter here to have your instances terminated. Once you do so, you will see:

```
[ truncated output from above ]
Type yes, then press enter, to continue. Otherwise, the operation will be cancelled.
yes
Instances terminated. Please confirm in your AWS Management Console.
The full log of this run is:
/home/centos/firesim-new/deploy/logs/2018-05-19--00-51-54-terminaterunfarm-
↪T9ZAED3LJUQQ3K0N.log
```

At this point, you should always confirm in your AWS management console that the instance is in the shutting-down or terminated states. You are ultimately responsible for ensuring that your instances are terminated appropriately.

Congratulations on running your first FireSim simulation! At this point, you can check-out some of the advanced features of FireSim in the sidebar to the left (for example, we expect that many people will be interested in the ability to automatically run the SPEC17 benchmarks: *SPEC 2017*), or you can continue on with the cluster simulation tutorial.

3.2 Running a Cluster Simulation

Now, let's move on to simulating a cluster of eight nodes, interconnected by a network with one 8-port Top-of-Rack (ToR) switch and 200 Gbps, 2μs links. This will require one `f1.16xlarge` (8 FPGA) instance.

Make sure you are `ssh` or `mosh`'d into your manager instance and have sourced `sourceme-f1-manager.sh` before running any of these commands.

3.2.1 Returning to a clean configuration

If you already ran the single-node tutorial, let's return to a clean FireSim manager configuration by doing the following:

```
cd firesim/deploy
cp sample-backup-configs/sample_config_runtime.ini config_runtime.ini
```

3.2.2 Building target software

If you already built target software during the single-node tutorial, you can skip to the next part (Setting up the manager configuration). If you haven't followed the single-node tutorial, continue with this section.

In these instructions, we'll assume that you want to boot the buildroot-based Linux distribution on each of the nodes in your simulated cluster. To do so, we'll need to build our FireSim-compatible RISC-V Linux distro. You can do this like so:

```
cd firesim/sw/firesim-software
./marshal -v build workloads/br-base.json
```

This process will take about 10 to 15 minutes on a `c4.4xlarge` instance. Once this is completed, you'll have the following files:

- `firesim/sw/firesim-software/images/br-disk-bin` - a bootloader + Linux kernel image for the nodes we will simulate.
- `firesim/sw/firesim-software/images/br-disk.img` - a disk image for each the nodes we will simulate

These files will be used to form base images to either build more complicated workloads (see the *Defining Custom Workloads* section) or to copy around for deploying.

3.2.3 Setting up the manager configuration

All runtime configuration options for the manager are set in a file called `firesim/deploy/config_runtime.ini`. In this guide, we will explain only the parts of this file necessary for our purposes. You can find full descriptions of all of the parameters in the *Manager Configuration Files* section.

If you open up this file, you will see the following default config (assuming you have not modified it):

```
# RUNTIME configuration for the FireSim Simulation Manager
# See docs/Advanced-Usage/Manager/Manager-Configuration-Files.rst for documentation
↪of all of these params.

[runfarm]
runfarmtag=mainrunfarm

f1_16xlarges=1
m4_16xlarges=0
f1_4xlarges=0
f1_2xlarges=0

runinstancemarket=ondemand
spotinterruptionbehavior=terminate
spotmaxprice=ondemand

[targetconfig]
topology=example_8config
no_net_num_nodes=2
linklatency=6405
switchinglatency=10
netbandwidth=200
profileinterval=-1

# This references a section from config_hwconfigs.ini
# In homogeneous configurations, use this to set the hardware config deployed
# for all simulators
defaulthwconfig=firesim-quadcore-nic-ddr3-llc4mb

[tracing]
enable=no
startcycle=0
endcycle=-1
```

(continues on next page)

(continued from previous page)

```
[workload]
workloadname=linux-uniform.json
terminateoncompletion=no
```

For the 8-node cluster simulation, the defaults in this file are exactly what we want. Let's outline the important parameters:

- `f1_16xlarges=1`: This tells the manager that we want to launch one `f1.16xlarge` when we call the `launchrunfarm` command.
- `topology=example_8config`: This tells the manager to use the topology named `example_8config` which is defined in `deploy/runtools/user_topology.py`. This topology simulates an 8-node cluster with one ToR switch.
- `linklatency=6405`: This models a network with 6405 cycles of link latency. Since we are modeling processors running at 3.2 Ghz, 1 cycle = 1/3.2 ns, so 6405 cycles is roughly 2 microseconds.
- `switchinglatency=10`: This models switches with a minimum port-to-port latency of 10 cycles.
- `netbandwidth=200`: This sets the bandwidth of the NICs to 200 Gbit/s. Currently you can set any integer value less than this without making hardware modifications.
- `defaulthwconfig=firesim-quadcore-nic-ddr3-llc4mb`: This tells the manager to use a quad-core Rocket Chip configuration with 4 MB of L2 and 16 GB of DDR3, with a NIC, for each of the simulated nodes in the topology.

You'll see other parameters here, like `runinstancemarket`, `spotinterruptionbehavior`, and `spotmaxprice`. If you're an experienced AWS user, you can see what these do by looking at the *Manager Configuration Files* section. Otherwise, don't change them.

As in the single-node tutorial, we will leave the last section (`[workload]`) unchanged here, since we do want to run the buildroot-based Linux on our simulated system. The `terminateoncompletion` feature is an advanced feature that you can learn more about in the *Manager Configuration Files* section.

As a final sanity check, your `config_runtime.ini` file should now look like this:

```
# RUNTIME configuration for the FireSim Simulation Manager
# See docs/Advanced-Usage/Manager/Manager-Configuration-Files.rst for documentation
↪of all of these params.

[runfarm]
runfarmtag=mainrunfarm

f1_16xlarges=1
m4_16xlarges=0
f1_4xlarges=0
f1_2xlarges=0

runinstancemarket=ondemand
spotinterruptionbehavior=terminate
spotmaxprice=ondemand

[targetconfig]
topology=example_8config
no_net_num_nodes=2
linklatency=6405
switchinglatency=10
netbandwidth=200
profileinterval=-1
```

(continues on next page)

(continued from previous page)

```
# This references a section from config_hwconfigs.ini
# In homogeneous configurations, use this to set the hardware config deployed
# for all simulators
defaulthwconfig=firesim-quadcore-nic-ddr3-llc4mb

[tracing]
enable=no
startcycle=0
endcycle=-1

[workload]
workloadname=linux-uniform.json
terminateoncompletion=no
```

Attention: [Advanced users] Simulating BOOM instead of Rocket Chip: If you would like to simulate a single-core BOOM as a target, set `defaulthwconfig` to `fireboom-singlecore-nic-ddr3-llc4mb`.

3.2.4 Launching a Simulation!

Now that we've told the manager everything it needs to know in order to run our single-node simulation, let's actually launch an instance and run it!

Starting the Run Farm

First, we will tell the manager to launch our Run Farm, as we specified above. When you do this, you will start getting charged for the running EC2 instances (in addition to your manager).

To do launch your run farm, run:

```
firesim launchrunfarm
```

You should expect output like the following:

```
centos@ip-172-30-2-111.us-west-2.compute.internal:~/firesim-new/deploy$ firesim␣
↪launchrunfarm
FireSim Manager. Docs: http://docs.fires.im
Running: launchrunfarm

Waiting for instance boots: f1.16xlarges
i-09e5491cce4d5f92d booted!
Waiting for instance boots: f1.4xlarges
Waiting for instance boots: m4.16xlarges
Waiting for instance boots: f1.2xlarges
The full log of this run is:
/home/centos/firesim-new/deploy/logs/2018-05-19--06-05-53-launchrunfarm-
↪ZGVP753DSU1Y9Q6R.log
```

The output will rapidly progress to `Waiting for instance boots: f1.16xlarges` and then take a minute or two while your `f1.16xlarge` instance launches. Once the launches complete, you should see the instance id printed and the instance will also be visible in your AWS EC2 Management console. The manager will tag the instances launched with this operation with the value you specified above as the `runfarmtag` parameter from the

`config_runtime.ini` file, which we left set as `mainrunfarm`. This value allows the manager to tell multiple Run Farms apart – i.e., you can have multiple independent Run Farms running different workloads/hardware configurations in parallel. This is detailed in the *Manager Configuration Files* and the *firesim launchrunfarm* sections – you do not need to be familiar with it here.

Setting up the simulation infrastructure

The manager will also take care of building and deploying all software components necessary to run your simulation (including switches for the networked case). The manager will also handle flashing FPGAs. To tell the manager to setup our simulation infrastructure, let's run:

```
firesim infrasetup
```

For a complete run, you should expect output like the following:

```
centos@ip-172-30-2-111.us-west-2.compute.internal:~/firesim-new/deploy$ firesim␣
↪infrasetup
FireSim Manager. Docs: http://docs.fires.im
Running: infrasetup

Building FPGA software driver for FireSim-FireSimRocketChipQuadCoreConfig-
↪FireSimDDR3FRFCFSLLC4MBConfig
Building switch model binary for switch switch0
[172.30.2.178] Executing task 'instance_liveness'
[172.30.2.178] Checking if host instance is up...
[172.30.2.178] Executing task 'infrasetup_node_wrapper'
[172.30.2.178] Copying FPGA simulation infrastructure for slot: 0.
[172.30.2.178] Copying FPGA simulation infrastructure for slot: 1.
[172.30.2.178] Copying FPGA simulation infrastructure for slot: 2.
[172.30.2.178] Copying FPGA simulation infrastructure for slot: 3.
[172.30.2.178] Copying FPGA simulation infrastructure for slot: 4.
[172.30.2.178] Copying FPGA simulation infrastructure for slot: 5.
[172.30.2.178] Copying FPGA simulation infrastructure for slot: 6.
[172.30.2.178] Copying FPGA simulation infrastructure for slot: 7.
[172.30.2.178] Installing AWS FPGA SDK on remote nodes.
[172.30.2.178] Unloading XDMA/EDMA/XOCL Driver Kernel Module.
[172.30.2.178] Copying AWS FPGA XDMA driver to remote node.
[172.30.2.178] Loading XDMA Driver Kernel Module.
[172.30.2.178] Clearing FPGA Slot 0.
[172.30.2.178] Clearing FPGA Slot 1.
[172.30.2.178] Clearing FPGA Slot 2.
[172.30.2.178] Clearing FPGA Slot 3.
[172.30.2.178] Clearing FPGA Slot 4.
[172.30.2.178] Clearing FPGA Slot 5.
[172.30.2.178] Clearing FPGA Slot 6.
[172.30.2.178] Clearing FPGA Slot 7.
[172.30.2.178] Flashing FPGA Slot: 0 with agfi: agfi-09e85ffabe3543903.
[172.30.2.178] Flashing FPGA Slot: 1 with agfi: agfi-09e85ffabe3543903.
[172.30.2.178] Flashing FPGA Slot: 2 with agfi: agfi-09e85ffabe3543903.
[172.30.2.178] Flashing FPGA Slot: 3 with agfi: agfi-09e85ffabe3543903.
[172.30.2.178] Flashing FPGA Slot: 4 with agfi: agfi-09e85ffabe3543903.
[172.30.2.178] Flashing FPGA Slot: 5 with agfi: agfi-09e85ffabe3543903.
[172.30.2.178] Flashing FPGA Slot: 6 with agfi: agfi-09e85ffabe3543903.
[172.30.2.178] Flashing FPGA Slot: 7 with agfi: agfi-09e85ffabe3543903.
[172.30.2.178] Unloading XDMA/EDMA/XOCL Driver Kernel Module.
[172.30.2.178] Loading XDMA Driver Kernel Module.
```

(continues on next page)

(continued from previous page)

```
[172.30.2.178] Copying switch simulation infrastructure for switch slot: 0.
The full log of this run is:
/home/centos/firesim-new/deploy/logs/2018-05-19--06-07-33-infrasetup-2Z7EBCBIF2TSI66Q.
↪log
```

Many of these tasks will take several minutes, especially on a clean copy of the repo (in particular, `f1.16xlarges` usually take a couple of minutes to start, so don't be alarmed if you're stuck at `Checking if host instance is up...`). The console output here contains the "user-friendly" version of the output. If you want to see detailed progress as it happens, `tail -f` the latest logfile in `firesim/deploy/logs/`.

At this point, the `f1.16xlarge` instance in our Run Farm has all the infrastructure necessary to run everything in our simulation.

So, let's launch our simulation!

Running a simulation!

Finally, let's run our simulation! To do so, run:

```
firesim runworkload
```

This command boots up the 8-port switch simulation and then starts 8 Rocket Chip FPGA Simulations, then prints out the live status of the simulated nodes and switch every 10s. When you do this, you will initially see output like:

```
centos@ip-172-30-2-111.us-west-2.compute.internal:~/firesim-new/deploy$ firesim␣
↪runworkload
FireSim Manager. Docs: http://docs.fires.im
Running: runworkload

Creating the directory: /home/centos/firesim-new/deploy/results-workload/2018-05-19--
↪06-28-43-linux-uniform/
[172.30.2.178] Executing task 'instance_liveness'
[172.30.2.178] Checking if host instance is up...
[172.30.2.178] Executing task 'boot_switch_wrapper'
[172.30.2.178] Starting switch simulation for switch slot: 0.
[172.30.2.178] Executing task 'boot_simulation_wrapper'
[172.30.2.178] Starting FPGA simulation for slot: 0.
[172.30.2.178] Starting FPGA simulation for slot: 1.
[172.30.2.178] Starting FPGA simulation for slot: 2.
[172.30.2.178] Starting FPGA simulation for slot: 3.
[172.30.2.178] Starting FPGA simulation for slot: 4.
[172.30.2.178] Starting FPGA simulation for slot: 5.
[172.30.2.178] Starting FPGA simulation for slot: 6.
[172.30.2.178] Starting FPGA simulation for slot: 7.
[172.30.2.178] Executing task 'monitor_jobs_wrapper'
```

If you don't look quickly, you might miss it, because it will be replaced with a live status page once simulations are kicked-off:

```
FireSim Simulation Status @ 2018-05-19 06:28:56.087472
--------------------------------------------------------------------------------
This workload's output is located in:
/home/centos/firesim-new/deploy/results-workload/2018-05-19--06-28-43-linux-uniform/
This run's log is located in:
/home/centos/firesim-new/deploy/logs/2018-05-19--06-28-43-runworkload-
↪ZHZEJED9MDWNSCV7.log
```

(continues on next page)

(continued from previous page)

```
This status will update every 10s.
--------------------------------------------------------------------------------
Instances
--------------------------------------------------------------------------------
Instance IP:   172.30.2.178 | Terminated: False
--------------------------------------------------------------------------------
Simulated Switches
--------------------------------------------------------------------------------
Instance IP:   172.30.2.178 | Switch name: switch0 | Switch running: True
--------------------------------------------------------------------------------
Simulated Nodes/Jobs
--------------------------------------------------------------------------------
Instance IP:   172.30.2.178 | Job: linux-uniform1 | Sim running: True
Instance IP:   172.30.2.178 | Job: linux-uniform0 | Sim running: True
Instance IP:   172.30.2.178 | Job: linux-uniform3 | Sim running: True
Instance IP:   172.30.2.178 | Job: linux-uniform2 | Sim running: True
Instance IP:   172.30.2.178 | Job: linux-uniform5 | Sim running: True
Instance IP:   172.30.2.178 | Job: linux-uniform4 | Sim running: True
Instance IP:   172.30.2.178 | Job: linux-uniform7 | Sim running: True
Instance IP:   172.30.2.178 | Job: linux-uniform6 | Sim running: True
--------------------------------------------------------------------------------
Summary
--------------------------------------------------------------------------------
1/1 instances are still running.
8/8 simulations are still running.
--------------------------------------------------------------------------------
```

In cycle-accurate networked mode, this will only exit when any ONE of the simulated nodes shuts down. So, let's let it run and open another ssh connection to the manager instance. From there, `cd` into your firesim directory again and `source sourceme-f1-manager.sh` again to get our ssh key setup. To access our simulated system, ssh into the IP address being printed by the status page, **from your manager instance**. In our case, from the above output, we see that our simulated system is running on the instance with IP `172.30.2.178`. So, run:

```
[RUN THIS ON YOUR MANAGER INSTANCE!]
ssh 172.30.2.178
```

This will log you into the instance running the simulation. On this machine, run `screen -ls` to get the list of all running simulation components. Attaching to the screens `fsim0` to `fsim7` will let you attach to the consoles of any of the 8 simulated nodes. You'll also notice an additional screen for the switch, however by default there is no interesting output printed here for performance reasons.

For example, if we want to enter commands into node zero, we can attach to its console like so:

```
screen -r fsim0
```

Voila! You should now see Linux booting on the simulated node and then be prompted with a Linux login prompt, like so:

```
[truncated Linux boot output]
[    0.020000] Registered IceNet NIC 00:12:6d:00:00:02
[    0.020000] VFS: Mounted root (ext2 filesystem) on device 254:0.
[    0.020000] devtmpfs: mounted
[    0.020000] Freeing unused kernel memory: 140K
[    0.020000] This architecture does not have kernel memory protection.
mount: mounting sysfs on /sys failed: No such device
Starting logging: OK
```

(continues on next page)

(continued from previous page)

```
Starting mdev...
mdev: /sys/dev: No such file or directory
modprobe: can't change directory to '/lib/modules': No such file or directory
Initializing random number generator... done.
Starting network: OK
Starting dropbear sshd: OK

Welcome to Buildroot
buildroot login:
```

If you also ran the single-node no-nic simulation you'll notice a difference in this boot output – here, Linux sees the NIC and its assigned MAC address and automatically brings up the `eth0` interface at boot.

Now, you can login to the system! The username is `root` and the password is `firesim`. At this point, you should be presented with a regular console, where you can type commands into the simulation and run programs. For example:

```
Welcome to Buildroot
buildroot login: root
Password:
# uname -a
Linux buildroot 4.15.0-rc6-31580-g9c3074b5c2cd #1 SMP Thu May 17 22:28:35 UTC 2018␣
↪riscv64 GNU/Linux
#
```

At this point, you can run workloads as you'd like. To finish off this tutorial, let's poweroff the simulated system and see what the manager does. To do so, in the console of the simulated system, run `poweroff -f`:

```
Welcome to Buildroot
buildroot login: root
Password:
# uname -a
Linux buildroot 4.15.0-rc6-31580-g9c3074b5c2cd #1 SMP Thu May 17 22:28:35 UTC 2018␣
↪riscv64 GNU/Linux
# poweroff -f
```

You should see output like the following from the simulation console:

```
# poweroff -f
[    3.748000] reboot: Power down
Power off
time elapsed: 360.5 s, simulation speed = 37.82 MHz
*** PASSED *** after 13634406804 cycles
Runs 13634406804 cycles
[PASS] FireSim Test
SEED: 1526711978
Script done, file is uartlog

[screen is terminating]
```

You'll also notice that the manager polling loop exited! You'll see output like this from the manager:

```
--------------------------------------------------------------------------------
Instances
--------------------------------------------------------------------------------
Instance IP:   172.30.2.178 | Terminated: False
--------------------------------------------------------------------------------
Simulated Switches
```

(continues on next page)

(continued from previous page)

```
----------------------------------------------------------------------------------
Instance IP:   172.30.2.178 | Switch name: switch0 | Switch running: True
----------------------------------------------------------------------------------
Simulated Nodes/Jobs
----------------------------------------------------------------------------------
Instance IP:   172.30.2.178 | Job: linux-uniform1 | Sim running: True
Instance IP:   172.30.2.178 | Job: linux-uniform0 | Sim running: False
Instance IP:   172.30.2.178 | Job: linux-uniform3 | Sim running: True
Instance IP:   172.30.2.178 | Job: linux-uniform2 | Sim running: True
Instance IP:   172.30.2.178 | Job: linux-uniform5 | Sim running: True
Instance IP:   172.30.2.178 | Job: linux-uniform4 | Sim running: True
Instance IP:   172.30.2.178 | Job: linux-uniform7 | Sim running: True
Instance IP:   172.30.2.178 | Job: linux-uniform6 | Sim running: True
----------------------------------------------------------------------------------
Summary
----------------------------------------------------------------------------------
1/1 instances are still running.
7/8 simulations are still running.
----------------------------------------------------------------------------------
Teardown required, manually tearing down...
[172.30.2.178] Executing task 'kill_switch_wrapper'
[172.30.2.178] Killing switch simulation for switchslot: 0.
[172.30.2.178] Executing task 'kill_simulation_wrapper'
[172.30.2.178] Killing FPGA simulation for slot: 0.
[172.30.2.178] Killing FPGA simulation for slot: 1.
[172.30.2.178] Killing FPGA simulation for slot: 2.
[172.30.2.178] Killing FPGA simulation for slot: 3.
[172.30.2.178] Killing FPGA simulation for slot: 4.
[172.30.2.178] Killing FPGA simulation for slot: 5.
[172.30.2.178] Killing FPGA simulation for slot: 6.
[172.30.2.178] Killing FPGA simulation for slot: 7.
[172.30.2.178] Executing task 'screens'
Confirming exit...
[172.30.2.178] Executing task 'monitor_jobs_wrapper'
[172.30.2.178] Slot 0 completed! copying results.
[172.30.2.178] Slot 1 completed! copying results.
[172.30.2.178] Slot 2 completed! copying results.
[172.30.2.178] Slot 3 completed! copying results.
[172.30.2.178] Slot 4 completed! copying results.
[172.30.2.178] Slot 5 completed! copying results.
[172.30.2.178] Slot 6 completed! copying results.
[172.30.2.178] Slot 7 completed! copying results.
[172.30.2.178] Killing switch simulation for switchslot: 0.
FireSim Simulation Exited Successfully. See results in:
/home/centos/firesim-new/deploy/results-workload/2018-05-19--06-39-35-linux-uniform/
The full log of this run is:
/home/centos/firesim-new/deploy/logs/2018-05-19--06-39-35-runworkload-
↪4CDB78E3A4IA9IYQ.log
```

In the cluster case, you'll notice that shutting down ONE simulator causes the whole simulation to be torn down – this is because we currently do not implement any kind of "disconnect" mechanism to remove one node from a globally-cycle-accurate simulation.

If you take a look at the workload output directory given in the manager output (in this case, `/home/centos/firesim-new/deploy/results-workload/2018-05-19--06-39-35-linux-uniform/`), you'll see the following:

```
centos@ip-172-30-2-111.us-west-2.compute.internal:~/firesim-new/deploy/results-
↪workload/2018-05-19--06-39-35-linux-uniform$ ls -la */*
-rw-rw-r-- 1 centos centos  797 May 19 06:45 linux-uniform0/memory_stats.csv
-rw-rw-r-- 1 centos centos  125 May 19 06:45 linux-uniform0/os-release
-rw-rw-r-- 1 centos centos 7476 May 19 06:45 linux-uniform0/uartlog
-rw-rw-r-- 1 centos centos  797 May 19 06:45 linux-uniform1/memory_stats.csv
-rw-rw-r-- 1 centos centos  125 May 19 06:45 linux-uniform1/os-release
-rw-rw-r-- 1 centos centos 8157 May 19 06:45 linux-uniform1/uartlog
-rw-rw-r-- 1 centos centos  797 May 19 06:45 linux-uniform2/memory_stats.csv
-rw-rw-r-- 1 centos centos  125 May 19 06:45 linux-uniform2/os-release
-rw-rw-r-- 1 centos centos 8157 May 19 06:45 linux-uniform2/uartlog
-rw-rw-r-- 1 centos centos  797 May 19 06:45 linux-uniform3/memory_stats.csv
-rw-rw-r-- 1 centos centos  125 May 19 06:45 linux-uniform3/os-release
-rw-rw-r-- 1 centos centos 8157 May 19 06:45 linux-uniform3/uartlog
-rw-rw-r-- 1 centos centos  797 May 19 06:45 linux-uniform4/memory_stats.csv
-rw-rw-r-- 1 centos centos  125 May 19 06:45 linux-uniform4/os-release
-rw-rw-r-- 1 centos centos 8157 May 19 06:45 linux-uniform4/uartlog
-rw-rw-r-- 1 centos centos  797 May 19 06:45 linux-uniform5/memory_stats.csv
-rw-rw-r-- 1 centos centos  125 May 19 06:45 linux-uniform5/os-release
-rw-rw-r-- 1 centos centos 8157 May 19 06:45 linux-uniform5/uartlog
-rw-rw-r-- 1 centos centos  797 May 19 06:45 linux-uniform6/memory_stats.csv
-rw-rw-r-- 1 centos centos  125 May 19 06:45 linux-uniform6/os-release
-rw-rw-r-- 1 centos centos 8157 May 19 06:45 linux-uniform6/uartlog
-rw-rw-r-- 1 centos centos  797 May 19 06:45 linux-uniform7/memory_stats.csv
-rw-rw-r-- 1 centos centos  125 May 19 06:45 linux-uniform7/os-release
-rw-rw-r-- 1 centos centos 8157 May 19 06:45 linux-uniform7/uartlog
-rw-rw-r-- 1 centos centos  153 May 19 06:45 switch0/switchlog
```

What are these files? They are specified to the manager in a configuration file (`firesim/deploy/workloads/linux-uniform.json`) as files that we want automatically copied back to our manager after we run a simulation, which is useful for running benchmarks automatically. Note that there is a directory for each simulated node and each simulated switch in the cluster. The *Defining Custom Workloads* section describes this process in detail.

For now, let's wrap-up our tutorial by terminating the `f1.16xlarge` instance that we launched. To do so, run:

```
firesim terminaterunfarm
```

Which should present you with the following:

```
centos@ip-172-30-2-111.us-west-2.compute.internal:~/firesim-new/deploy$ firesim␣
↪terminaterunfarm
FireSim Manager. Docs: http://docs.fires.im
Running: terminaterunfarm

IMPORTANT!: This will terminate the following instances:
f1.16xlarges
['i-09e5491cce4d5f92d']
f1.4xlarges
[]
m4.16xlarges
[]
f1.2xlarges
[]
Type yes, then press enter, to continue. Otherwise, the operation will be cancelled.
```

You must type `yes` then hit enter here to have your instances terminated. Once you do so, you will see:

```
[ truncated output from above ]
Type yes, then press enter, to continue. Otherwise, the operation will be cancelled.
yes
Instances terminated. Please confirm in your AWS Management Console.
The full log of this run is:
/home/centos/firesim-new/deploy/logs/2018-05-19--06-50-37-terminaterunfarm-
↪3VF0Z2KCAKKDY0ZU.log
```

At this point, you should always confirm in your AWS management console that the instance is in the shutting-down or terminated states. You are ultimately responsible for ensuring that your instances are terminated appropriately.

Congratulations on running a cluster FireSim simulation! At this point, you can check-out some of the advanced features of FireSim in the sidebar to the left. Or, hit next to continue to a tutorial that shows you how to build your own custom FPGA images.

CHAPTER 4

Building Your Own Hardware Designs (FireSim FPGA Images)

This section will guide you through building an AFI image for a FireSim simulation.

4.1 Amazon S3 Setup

During the build process, the build system will need to upload a tar file to Amazon S3 in order to complete the build process using Amazon's backend scripts (which convert the Vivado-generated tar into an AFI). The manager will create this bucket for you automatically, you just need to specify a name.

So, choose a bucket name, e.g. `firesim-yourname`. Bucket names must be globally unique. If you choose one that's already taken, the manager will notice and complain when you tell it to build an AFI. To set your bucket name, open `deploy/config_build.ini` in your editor and under the `[afibuild]` header, replace

```
s3bucketname=firesim-yournamehere
```

with your own bucket name, e.g.:

```
s3bucketname=firesim-sagar
```

4.2 Build Recipes

In the `deploy/config_build.ini` file, you will notice that the `[builds]` section currently contains several lines, which indicates to the build system that you want to run all of these builds in parallel, with the parameters listed in the relevant section of the `deploy/config_build_recipes.ini` file. Here you can set parameters of the simulated system, and also select the type of instance on which the Vivado build will be deployed. From our experimentation, there are diminishing returns using anything above a `c4.4xlarge`, so we default to that.

To start out, let's build a simple design, `firesim-singlecore-no-nic-lbp`. This is a design that has one core, no nic, and uses the latency-bandwidth pipe memory model. To do so, comment out all of the other build entries in `deploy/config_build.ini`, besides the one we want.. So, you should end up with something like this (a line beginning with a # is a comment):

```
[builds]
# this section references builds defined in config_build_recipes.ini
# if you add a build here, it will be built when you run buildafi
#firesim-singlecore-nic-lbp
firesim-singlecore-no-nic-lbp
#firesim-quadcore-nic-lbp
#firesim-quadcore-no-nic-lbp
#firesim-quadcore-nic-ddr3-llc4mb
#firesim-quadcore-no-nic-ddr3-llc4mb
```

4.3 Running a Build

Now, we can run a build like so:

```
firesim buildafi
```

This will run through the entire build process, taking the Chisel RTL and producing an AFI/AGFI that runs on the FPGA. This whole process will usually take a few hours. When the build completes, you will see a directory in `deploy/results-build/`, named after your build parameter settings, that contains AGFI information (the `AGFI_INFO` file) and all of the outputs of the Vivado build process (in the `cl_firesim` subdirectory). Additionally, the manager will print out a path to a log file that describes everything that happened, in-detail, during this run (this is a good file to send us if you encounter problems). If you provided the manager with your email address, you will also receive an email upon build completion, that should look something like this:

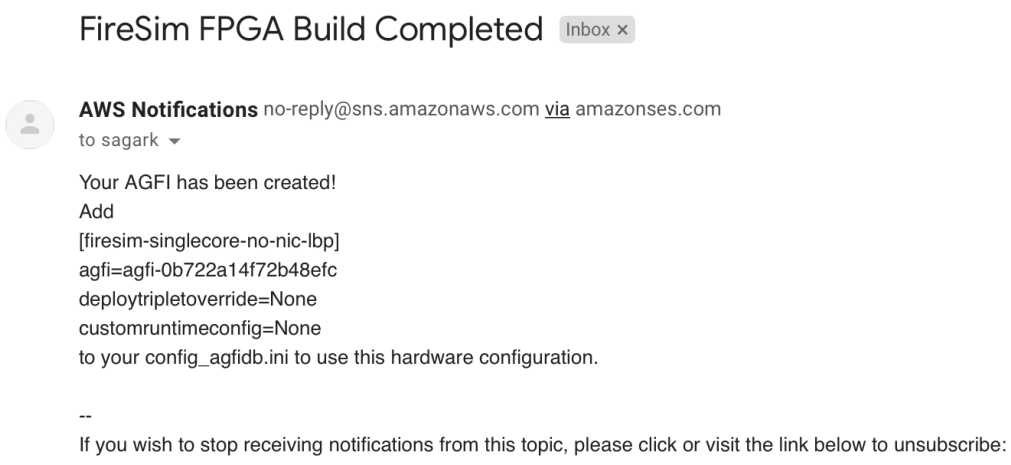

Fig. 1: Build Completion Email

Now that you know how to generate your own FPGA image, you can modify the target-design to add your own features, then build a FireSim-compatible FPGA image automatically! To learn more advanced FireSim features, you can choose a link under the "Advanced Docs" section to the left.

CHAPTER 5

Manager Usage (the `firesim` command)

5.1 Overview

When you source `sourceme-f1-manager.sh` in your copy of the firesim repo, you get access to a new command, `firesim`, which is the FireSim simulation manager. If you've used tools like Vagrant or Docker, the `firesim` program is to FireSim what `vagrant` and `docker` are to Vagrant and Docker respectively. In essence, `firesim` lets us manage the entire lifecycle of FPGA simulations, just like `vagrant` and `docker` do for VMs and containers respectively.

5.1.1 "Inputs" to the Manager

The manager gets configuration information from several places:

- Command Line Arguments, consisting of:
  - Paths to configuration files to use
  - A task to run
  - Arguments to the task
- Configuration Files
- Topology definitions for networked simulations (`user_topology.py`)

The following sections detail these inputs. Hit Next to continue.

5.1.2 Logging

The manager produces detailed logs when you run any command, which is useful to share with the FireSim developers for debugging purposes in case you encounter issues. The logs contain more detailed output than the manager sends to stdout/stderr during normal operation, so it's also useful if you want to take a peek at the detailed commands manager is running to facilitate builds and simulations. Logs are stored in `firesim/deploy/logs/`.

5.2 Manager Command Line Arguments

The manager provides built-in help output for the command line arguments it supports if you run `firesim --help`

```
usage: firesim [-h] [-c RUNTIMECONFIGFILE] [-b BUILDCONFIGFILE]
               [-r BUILDRECIPESCONFIGFILE] [-a HWDBCONFIGFILE]
               [-x OVERRIDECONFIGDATA] [-f TERMINATESOMEF116]
               [-g TERMINATESOMEF12] [-i TERMINATESOMEF14]
               [-m TERMINATESOMEM416] [-q]

               {managerinit,buildafi,launchrunfarm,infrasetup,boot,kill,
↪terminaterunfarm,runworkload,shareagfi,runcheck}

FireSim Simulation Manager.

positional arguments:
  {managerinit,buildafi,launchrunfarm,infrasetup,boot,kill,terminaterunfarm,
↪runworkload,shareagfi,runcheck}
                        Management task to run.

optional arguments:
  -h, --help            show this help message and exit
  -c RUNTIMECONFIGFILE, --runtimeconfigfile RUNTIMECONFIGFILE
                        Optional custom runtime/workload config file. Defaults
                        to config_runtime.ini.
  -b BUILDCONFIGFILE, --buildconfigfile BUILDCONFIGFILE
                        Optional custom build config file. Defaults to
                        config_build.ini.
  -r BUILDRECIPESCONFIGFILE, --buildrecipesconfigfile BUILDRECIPESCONFIGFILE
                        Optional custom build recipe config file. Defaults to
                        config_build_recipes.ini.
  -a HWDBCONFIGFILE, --hwdbconfigfile HWDBCONFIGFILE
                        Optional custom HW database config file. Defaults to
                        config_hwdb.ini.
  -x OVERRIDECONFIGDATA, --overrideconfigdata OVERRIDECONFIGDATA
                        Override a single value from one of the the RUNTIME
                        e.g.: --overrideconfigdata "targetconfig linklatency
                        6405".
  -f TERMINATESOMEF116, --terminatesomef116 TERMINATESOMEF116
                        Only used by terminatesome. Terminates this many of
                        the previously launched f1.16xlarges.
  -g TERMINATESOMEF12, --terminatesomef12 TERMINATESOMEF12
                        Only used by terminatesome. Terminates this many of
                        the previously launched f1.2xlarges.
  -i TERMINATESOMEF14, --terminatesomef14 TERMINATESOMEF14
                        Only used by terminatesome. Terminates this many of
                        the previously launched f1.4xlarges.
  -m TERMINATESOMEM416, --terminatesomem416 TERMINATESOMEM416
                        Only used by terminatesome. Terminates this many of
                        the previously launched m4.16xlarges.
  -q, --forceterminate  For terminaterunfarm, force termination without
                        prompting user for confirmation. Defaults to False
```

On this page, we will go through some of these options – others are more complicated, so we will give them their own section on the following pages.

5.2.1 `--runtimeconfigfile FILENAME`

This lets you specify a custom runtime config file. By default, `config_runtime.ini` is used. See *config_runtime.ini* for what this config file does.

5.2.2 `--buildconfigfile FILENAME`

This lets you specify a custom build config file. By default, `config_build.ini` is used. See *config_build.ini* for what this config file does.

5.2.3 `--buildrecipesconfigfile FILENAME`

This lets you specify a custom build **recipes** config file. By default, `config_build_recipes.ini` is used. See *config_build_recipes.ini* for what this config file does.

5.2.4 `--hwdbconfigfile FILENAME`

This lets you specify a custom hardware database config file. By default, `config_hwdb.ini` is used. See *config_hwdb.ini* for what this config file does.

5.2.5 `--overrideconfigdata SECTION PARAMETER VALUE`

This lets you override a single value from the runtime config file. For example, if you want to use a link latency of 3003 cycles for a particular run (and your `config_runtime.ini` file specifies differently), you can pass `--overrideconfigdata targetconfig linklatency 6405` to the manager. This can be used with any task that uses the runtime config.

5.2.6 `TASK`

This is the only required/positional command line argument to the manager. It tells the manager what it should be doing. See the next section for a list of tasks and what they do. Some tasks also take other command line arguments, which are specified with those tasks.

5.3 Manager Tasks

This page outlines all of the tasks that the FireSim manager supports.

5.3.1 `firesim managerinit`

This is a setup command that does the following:

- Run `aws configure`, prompt for credentials
- Replace the default config files (`config_runtime.ini`, `config_build.ini`, `config_build_recipes.ini`, and `config_hwdb.ini`) with clean example versions.
- Prompt the user for email address and subscribe them to notifications for their own builds.

You can re-run this whenever you want to get clean configuration files – you can just hit enter when prompted for aws configure credentials and your email address, and both will keep your previously specified values.

If you run this command by accident and didn't mean to overwrite your configuration files, you'll find backed-up versions in `firesim/deploy/sample-backup-configs/backup*`.

5.3.2 `firesim buildafi`

This command builds a FireSim AGFI (FPGA Image) from the Chisel RTL for the configurations that you specify. The process of defining configurations to build is explained in the documentation for *config_build.ini* and *config_build_recipes.ini*.

For each config, the build process entails:

1. [Locally] Run the elaboration process for your hardware configuration
2. [Locally] FAME-1 transform the design with MIDAS
3. [Locally] Attach simulation models (I/O widgets, memory model, etc.)
4. [Locally] Emit Verilog to run through the AWS FPGA Flow
5. Launch an FPGA Dev AMI build instance for each configuration you want built.
6. [Local/Remote] Prep build instances, copy generated verilog for hardware configuration to build instance.
7. [Remote] Run Vivado Synthesis and P&R for the configuration
8. [Local/Remote] Copy back all output generated by Vivado, including the final tar file
9. [Local/AWS Infra] Submit the tar file to the AWS backend for conversion to an AFI
10. [Local] Wait for the AFI to become available, then notify the user of completion by email.

This process happens in parallel for all of the builds you specify. The command will exit when all builds are completed (but you will get notified as INDIVIDUAL builds complete).

It is highly recommended that you either run this command in a ‘‘screen‘‘ or use ‘‘mosh‘‘ to access the build instance. Builds will not finish if the manager is killed due to disconnection to the instance.

When you run a build for a particular configuration, a directory named `LAUNCHTIME-CONFIG_TRIPLET-BUILD_NAME` is created in `firesim/deploy/results-build/`. This directory will contain:

- `AGFI_INFO`: Describes the state of the AFI being built, while the manager is running. Upon build completion, this contains the AGFI/AFI that was produced, along with its metadata.
- `cl_firesim:`: This directory is essentially the Vivado project that built the FPGA image, in the state it was in when the Vivado build process completed. This contains reports, stdout from the build, and the final tar file produced by Vivado.
- `cl_firesim_generated.sv`: This is a copy of the generated verilog used to produce this build. You can also find a copy inside `cl_firesim`.

5.3.3 `firesim shareagfi`

This command allows you to share AGFIs that you have already built (that are listed in *config_hwdb.ini*) with other users. It will take the named hardware configurations that you list in the `[agfistoshare]` section of `config_build.ini`, grab the respective AGFIs for each from `config_hwdb.ini`, and share them across all F1 regions with the users listed in the `[sharewithaccounts]` section of `config_build.ini`. You can also specify `public=public` in `[sharewithaccounts]` to make the AGFIs public.

You must own the AGFIs in order to do this – this will NOT let you share AGFIs that someone else owns and gave you access to.

5.3.4 `firesim launchrunfarm`

This command launches a Run Farm on which you run simulations. Run Farms consist of `f1.16xlarge`, `f1.4xlarge`, `f1.2xlarge`, and `m4.16xlarge` instances. Before you run the command, you define the number of each that you want in `config_runtime.ini`.

A launched Run Farm is tagged with a `runfarmtag` from `config_runtime.ini`, which is used to disambiguate multiple parallel Run Farms; that is, you can have many Run Farms running, each running a different experiment at the same time, each with its own unique `runfarmtag`. One convenient feature to add to your AWS management panel is the column for `fsimcluster`, which contains the `runfarmtag` value. You can see how to do that in the *Add the fsimcluster column to your AWS management console* section.

The other options in the `[runfarm]` section, `runinstancemarket`, `spotinterruptionbehavior`, and `spotmaxprice` define *how* instances in the Run Farm are launched. See the documentation for `config_runtime.ini` for more details.

ERRATA: One current requirement is that you must define a target config in the `[targetconfig]` section of `config_runtime.ini` that does not require more resources than the Run Farm you are trying to launch. Thus, you should also setup your `[targetconfig]` parameters before trying to launch the corresponding Run Farm. This requirement will be removed in the future.

Once you setup your configuration and call `firesim launchrunfarm`, the command will launch the requested numbers and types of instances. If all succeeds, you will see the command print out instance IDs for the correct number/types of launched instances (you do not need to pay attention to these or record them). If an error occurs, it will be printed to console.

Once you run this command, your Run Farm will continue to run until you call ‘‘firesim terminaterunfarm‘‘. This means you will be charged for the running instances in your Run Farm until you call ‘‘terminaterunfarm‘‘. You are responsible for ensuring that instances are only running when you want them to be by checking the AWS EC2 Management Panel.

5.3.5 `firesim terminaterunfarm`

This command terminates some or all of the instances in the Run Farm defined in your `config_runtime.ini` file, depending on the command line arguments you supply. By default, running `firesim terminaterunfarm` will terminate ALL instances with the specified `runfarmtag`. When you run this command, it will prompt for confirmation that you want to terminate the listed instances. If you respond in the affirmative, it will move forward with the termination.

If you do not want to have to confirm the termination (e.g. you are using this command in a script), you can give the command the `--forceterminate` command line argument. For example, the following will TERMINATE ALL INSTANCES IN THE RUN FARM WITHOUT PROMPTING FOR CONFIRMATION:

```
firesim terminaterunfarm --forceterminate
```

There a few additional commandline arguments that let you terminate only some of the instances in a particular Run Farm: `--terminatesomef116 INT`, `--terminatesomef14 INT`, `--terminatesomef12 INT`, and `--terminatesomem416 INT`, which will terminate ONLY as many of each type of instance as you specify.

Here are some examples:

```
[ start with 2 f1.16xlarges, 2 f1.2xlarges, 2 m4.16xlarges ]

firesim terminaterunfarm --terminatesomef116 1 --forceterminate

[ now, we have: 1 f1.16xlarges, 2 f1.2xlarges, 2 m4.16xlarges ]
```

```
[ start with 2 f1.16xlarges, 2 f1.2xlarges, 2 m4.16xlarges ]

firesim terminaterunfarm --terminatesomef116 1 --terminatesomef12 2 --forceterminate

[ now, we have: 1 f1.16xlarges, 0 f1.2xlarges, 2 m4.16xlarges ]
```

Once you call ‘‘launchrunfarm‘‘, you will be charged for running instances in your Run Farm until you call ‘‘terminaterunfarm‘‘. You are responsible for ensuring that instances are only running when you want them to be by checking the AWS EC2 Management Panel.

5.3.6 `firesim infrasetup`

Once you have launched a Run Farm and setup all of your configuration options, the `infrasetup` command will build all components necessary to run the simulation and deploy those components to the machines in the Run Farm. Here is a rough outline of what the command does:

- Constructs the internal representation of your simulation. This is a tree of components in the simulation (simulated server blades, switches)
- For each type of server blade, query the AWS AFI API to get the build-triplet needed to construct the software simulation driver, then build each driver
- For each type of switch in the simulation, generate the switch model binary
- For each host instance in the Run Farm, collect information about all the resources necessary to run a simulation on that host instance, then copy files and flash FPGAs with the required AGFIs.

Details about setting up your simulation configuration can be found in *config_runtime.ini*.

Once you run a simulation, you should re-run ‘‘firesim infrasetup‘‘ before starting another one, even if it is the same exact simulation on the same Run Farm.

You can see detailed output from an example run of `infrasetup` in the *Running a Single Node Simulation* and *Running a Cluster Simulation* Tutorials.

5.3.7 `firesim boot`

Once you have run `firesim infrasetup`, this command will actually start simulations. It begins by launching all switches (if they exist in your simulation config), then launches all server blade simulations. This simply launches simulations and then exits – it does not perform any monitoring.

This command is useful if you want to launch a simulation, then plan to interact with the simulation by-hand (i.e. by directly interacting with the console).

5.3.8 `firesim kill`

Given a simulation configuration and simulations running on a Run Farm, this command force-terminates all components of the simulation. Importantly, this does not allow any outstanding changes to the filesystem in the simulated systems to be committed to the disk image.

5.3.9 `firesim runworkload`

This command is the standard tool that lets you launch simulations, monitor the progress of workloads running on them, and collect results automatically when the workloads complete. To call this command, you must have first called `firesim infrasetup` to setup all required simulation infrastructure on the remote nodes.

This command will first create a directory in `firesim/deploy/results-workload/` named as `LAUNCH_TIME-WORKLOADNAME`, where results will be completed as simulations complete. This command will then automatically call `firesim boot` to start simulations. Then, it polls all the instances in the Run Farm every 10 seconds to determine the state of the simulated system. If it notices that a simulation has shutdown (i.e. the simulation disappears from the output of `screen -ls`), it will automatically copy back all results from the simulation, as defined in the workload configuration (see the *Defining Custom Workloads* section).

For non-networked simulations, it will wait for ALL simulations to complete (copying back results as each workload completes), then exit.

For globally-cycle-accurate networked simulations, the global simulation will stop when any single node powers off. Thus, for these simulations, `runworkload` will copy back results from all nodes and force them to terminate by calling `kill` when ANY SINGLE ONE of them shuts down cleanly.

A simulation shuts down cleanly when the workload running on the simulator calls `poweroff`.

5.3.10 `firesim runcheck`

This command is provided to let you debug configuration options without launching instances. In addition to the output produced at command line/in the log, you will find a pdf diagram of the topology you specify, annotated with information about the workloads, hardware configurations, and abstract host mappings for each simulation (and optionally, switch) in your design. These diagrams are located in `firesim/deploy/generated-topology-diagrams/`, named after your topology.

Here is an example of such a diagram (click to expand/zoom):

Fig. 1: Example diagram for an 8-node cluster with one ToR switch

5.4 Manager Configuration Files

This page contains a centralized reference for all of the configuration options in `config_runtime.ini`, `config_build.ini`, `config_build_recipes.ini`, and `config_hwdb.ini`.

5.4.1 `config_runtime.ini`

Here is a sample of this configuration file:

```
# RUNTIME configuration for the FireSim Simulation Manager
# See docs/Advanced-Usage/Manager/Manager-Configuration-Files.rst for documentation␣
↪of all of these params.

[runfarm]
```

(continues on next page)

(continued from previous page)

```
runfarmtag=mainrunfarm

f1_16xlarges=1
m4_16xlarges=0
f1_4xlarges=0
f1_2xlarges=0

runinstancemarket=ondemand
spotinterruptionbehavior=terminate
spotmaxprice=ondemand

[targetconfig]
topology=example_8config
no_net_num_nodes=2
linklatency=6405
switchinglatency=10
netbandwidth=200
profileinterval=-1

# This references a section from config_hwconfigs.ini
# In homogeneous configurations, use this to set the hardware config deployed
# for all simulators
defaulthwconfig=firesim-quadcore-nic-ddr3-llc4mb

[tracing]
enable=no
startcycle=0
endcycle=-1

[workload]
workloadname=linux-uniform.json
terminateoncompletion=no
```

Below, we outline each section and parameter in detail.

[runfarm]

The `[runfarm]` options below allow you to specify the number, types, and other characteristics of instances in your FireSim Run Farm, so that the manager can automatically launch them, run workloads on them, and terminate them.

runfarmtag

Use `runfarmtag` to differentiate between different Run Farms in FireSim. Having multiple `config_runtime.ini` files with different `runfarmtag` values allows you to run many experiments at once from the same manager instance.

The instances launched by the `launchrunfarm` command will be tagged with this value. All later operations done by the manager rely on this tag, so you should not change it unless you are done with your current Run Farm.

Per AWS restrictions, this tag can be no longer than 255 characters.

f1_16xlarges, m4_16xlarges, f1_4xlarges, f1_2xlarges

Set these three values respectively based on the number and types of instances you need. While we could automate this setting, we choose not to, so that users are never surprised by how many instances they are running.

Note that these values are ONLY used to launch instances. After launch, the manager will query the AWS API to find the instances of each type that have the `runfarmtag` set above assigned to them.

runinstancemarket

You can specify either `spot` or `ondemand` here, to use one of those markets on AWS.

spotinterruptionbehavior

When `runinstancemarket=spot`, this value determines what happens to an instance if it receives the interruption signal from AWS. You can specify either `hibernate`, `stop`, or `terminate`.

spotmaxprice

When `runinstancemarket=spot`, this value determines the max price you are willing to pay per instance, in dollars. You can also set it to `ondemand` to set your max to the on-demand price for the instance.

[targetconfig]

The `[targetconfig]` options below allow you to specify the high-level configuration of the target you are simulating. You can change these parameters after launching a Run Farm (assuming you have the correct number of instances), but in many cases you will need to re-run the `infrasetup` command to make sure the correct simulation infrastructure is available on your instances.

topology

This field dictates the network topology of the simulated system. Some examples:

`no_net_config`: This runs N (see `no_net_num_nodes` below) independent simulations, without a network simulation. You can currently only use this option if you build one of the NoNIC hardware configs of FireSim.

`example_8config`: This requires a single `f1.16xlarge`, which will simulate 1 ToR switch attached to 8 simulated servers.

`example_16config`: This requires two `f1.16xlarge` instances and one `m4.16xlarge` instance, which will simulate 2 ToR switches, each attached to 8 simulated servers, with the two ToR switches connected by a root switch.

`example_64config`: This requires eight `f1.16xlarge` instances and one `m4.16xlarge` instance, which will simulate 8 ToR switches, each attached to 8 simulated servers (for a total of 64 nodes), with the eight ToR switches connected by a root switch.

Additional configurations are available in `deploy/runtools/user_topology.py` and more can be added there. See the *Manager Network Topology Definitions (user_topology.py)* section for more info.

no_net_num_nodes

This determines the number of simulated nodes when you are using `topology=no_net_config`.

linklatency

In a networked simulation, this allows you to specify the link latency of the simulated network in CYCLES. For example, 6405 cycles is roughly 2 microseconds at 3.2 GHz. A current limitation is that this value (in cycles) must be a multiple of 7. Furthermore, you must not exceed the buffer size specified in the NIC's simulation widget.

switchinglatency

In a networked simulation, this specifies the minimum port-to-port switching latency of the switch models, in CYCLES.

netbandwidth

In a networked simulation, this specifies the maximum output bandwidth that a NIC is allowed to produce as an integer in Gbit/s. Currently, this must be a number between 1 and 200, allowing you to model NICs between 1 and 200 Gbit/s.

defaulthwconfig

This sets the server configuration launched by default in the above topologies. Heterogeneous configurations can be achieved by manually specifying different names within the topology itself, but all the `example_Nconfig` configurations are homogeneous and use this value for all nodes.

You should set this to one of the hardware configurations you have defined already in `config_hwdb.ini`. You should set this to the NAME (section title) of the hardware configuration from `config_hwdb.ini`, NOT the actual agfi itself (NOT something like `agfi-XYZ...`).

[workload]

This section defines the software that will run on the simulated system.

workloadname

This selects a workload to run across the set of simulated nodes. A workload consists of a series of jobs that need to be run on simulated nodes (one job per node).

Workload definitions are located in `firesim/deploy/workloads/*.json`.

Some sample workloads:

`linux-uniform.json`: This runs the default FireSim Linux distro on as many nodes as you specify when setting the `[targetconfig]` parameters.

`spec17-intrate.json`: This runs SPECint 2017's rate benchmarks. In this type of workload, you should launch EXACTLY the correct number of nodes necessary to run the benchmark. If you specify fewer nodes, the manager will warn that not all jobs were assigned to a simulation. If you specify too many simulations and not enough jobs, the manager will not launch the jobs.

Others can be found in the aforementioned directory.

terminateoncompletion

Set this to `no` if you want your Run Farm to keep running once the workload has completed. Set this to `yes` if you want your Run Farm to be TERMINATED after the workload has completed and results have been copied off.

5.4.2 config_build.ini

Here is a sample of this configuration file:

```
# BUILDTIME/AGFI management configuration for the FireSim Simulation Manager
# See docs/Advanced-Usage/Manager/Manager-Configuration-Files.rst for documentation
↪of all of these params.

[afibuild]

s3bucketname=firesim-AWSUSERNAME
buildinstancemarket=ondemand
spotinterruptionbehavior=terminate
spotmaxprice=ondemand
postbuildhook=

[builds]
# this section references builds defined in config_build_recipes.ini
# if you add a build here, it will be built when you run buildafi
firesim-singlecore-no-nic-lbp
#firesim-singlecore-nic-lbp
#firesim-quadcore-no-nic-lbp
#firesim-quadcore-nic-lbp
firesim-quadcore-no-nic-ddr3-llc4mb
firesim-quadcore-nic-ddr3-llc4mb
#fireboom-singlecore-no-nic-lbp
fireboom-singlecore-no-nic-ddr3-llc4mb
#fireboom-singlecore-nic-lbp
fireboom-singlecore-nic-ddr3-llc4mb
#firesim-supernode-singlecore-nic-ddr3-llc4mb
#firesim-supernode-quadcore-nic-ddr3-llc4mb
firesim-supernode-singlecore-nic-lbp


[agfistoshare]
firesim-singlecore-no-nic-lbp
#firesim-singlecore-nic-lbp
#firesim-quadcore-no-nic-lbp
#firesim-quadcore-nic-lbp
firesim-quadcore-no-nic-ddr3-llc4mb
firesim-quadcore-nic-ddr3-llc4mb
#fireboom-singlecore-no-nic-lbp
fireboom-singlecore-no-nic-ddr3-llc4mb
#fireboom-singlecore-nic-lbp
fireboom-singlecore-nic-ddr3-llc4mb
#firesim-supernode-singlecore-nic-ddr3-llc4mb
#firesim-supernode-quadcore-nic-ddr3-llc4mb
firesim-supernode-singlecore-nic-lbp
```

(continues on next page)

(continued from previous page)

```
[sharewithaccounts]
somebodysname=123456789012
```

Below, we outline each section and parameter in detail.

[afibuild]

This exposes options for AWS resources used in the process of building FireSim AGFIs (FPGA Images).

s3bucketname

This is used behind the scenes in the AGFI creation process. You will only ever need to access this bucket manually if there is a failure in AGFI creation in Amazon's backend.

Naming rules: this must be all lowercase and you should stick to letters and numbers.

The first time you try to run a build, the FireSim manager will try to create the bucket you name here. If the name is unavailable, it will complain and you will need to change this name. Once you choose a working name, you should never need to change it.

In general, `firesim-yournamehere` is a good choice.

buildinstancemarket

You can specify either `spot` or `ondemand` here, to use one of those markets on AWS.

spotinterruptionbehavior

When `buildinstancemarket=spot`, this value determines what happens to an instance if it receives the interruption signal from AWS. You can specify either `hibernate`, `stop`, or `terminate`.

spotmaxprice

When `buildinstancemarket=spot`, this value determines the max price you are willing to pay per instance, in dollars. You can also set it to `ondemand` to set your max to the on-demand price for the instance.

[builds]

In this section, you can list as many build entries as you want to run for a particular call to the `buildafi` command (see `config_build_recipes.ini` below for how to define a build entry). For example, if we want to run the builds named `[awesome-firesim-config]` and `[quad-core-awesome-firesim-config]`, we would write:

```
[builds]
awesome-firesim-config
quad-core-awesome-firesim-config
```

[agfistoshare]

This is used by the `shareagfi` command to share the specified agfis with the users specified in the next (`[sharewithaccounts]`) section. In this section, you should specify the section title (i.e. the name you made up) for a hardware configuration in `config_hwdb.ini`. For example, to share the hardware config:

```
[firesim-quadcore-nic-ddr3-llc4mb]
# this is a comment that describes my favorite configuration!
agfi=agfi-0a6449b5894e96e53
deploytripletoverride=None
customruntimeconfig=None
```

you would use:

```
[agfistoshare]
firesim-quadcore-nic-ddr3-llc4mb
```

[sharewithaccounts]

A list of AWS account IDs that you want to share the AGFIs listed in `[agfistoshare]` with when calling the manager's `shareagfi` command. You should specify names in the form `usersname=AWSACCTID`. The left-hand-side is just for human readability, only the actual account IDs listed here matter. If you specify `public=public` here, the AGFIs are shared publicly, regardless of any other entires that are present.

5.4.3 config_build_recipes.ini

Here is a sample of this configuration file:

```
# Build-time design configuration for the FireSim Simulation Manager
# See docs/Advanced-Usage/Manager/Manager-Configuration-Files.rst for documentation
↪of all of these params.

# this file contains sections that describe hardware designs that /can/ be built.
# edit config_build.ini to actually "turn on" a config to be built when you run
# buildafi

#[firesim-singlecore-nic-lbp]
#DESIGN=FireSim
#TARGET_CONFIG=FireSimRocketChipSingleCoreConfig
#PLATFORM_CONFIG=FireSimConfig
#instancetype=c4.4xlarge
#deploytriplet=None

[firesim-singlecore-no-nic-lbp]
DESIGN=FireSimNoNIC
TARGET_CONFIG=FireSimRocketChipSingleCoreConfig
PLATFORM_CONFIG=FireSimConfig160MHz
instancetype=c4.4xlarge
deploytriplet=None

#[firesim-quadcore-nic-lbp]
#DESIGN=FireSim
#TARGET_CONFIG=FireSimRocketChipQuadCoreConfig
#PLATFORM_CONFIG=FireSimConfig
```

(continues on next page)

(continued from previous page)

```
#instancetype=c4.4xlarge
#deploytriplet=None
#
#[firesim-quadcore-no-nic-lbp]
#DESIGN=FireSimNoNIC
#TARGET_CONFIG=FireSimRocketChipQuadCoreConfig
#PLATFORM_CONFIG=FireSimConfig
#instancetype=c4.4xlarge
#deploytriplet=None

[firesim-quadcore-nic-ddr3-llc4mb]
DESIGN=FireSim
TARGET_CONFIG=FireSimRocketChipQuadCoreConfig
PLATFORM_CONFIG=FireSimDDR3FRFCFSLLC4MBConfig90MHz
instancetype=c4.4xlarge
deploytriplet=None

[firesim-quadcore-no-nic-ddr3-llc4mb]
DESIGN=FireSimNoNIC
TARGET_CONFIG=FireSimRocketChipQuadCoreConfig
PLATFORM_CONFIG=FireSimDDR3FRFCFSLLC4MBConfig90MHz
instancetype=c4.4xlarge
deploytriplet=None

[firesim-quadcore-no-nic-ddr3-llc4mb-3div]
DESIGN=FireSimNoNIC
TARGET_CONFIG=FireSimRocketChipQuadCoreConfig
PLATFORM_CONFIG=FireSimDDR3FRFCFSLLC4MB3ClockDivConfig
instancetype=c4.4xlarge
deploytriplet=None

# BOOM-based targets
#[fireboom-singlecore-no-nic-lbp]
#DESIGN=FireBoomNoNIC
#TARGET_CONFIG=FireSimBoomConfig
#PLATFORM_CONFIG=FireSimConfig
#instancetype=c4.4xlarge
#deploytriplet=None

[fireboom-singlecore-no-nic-ddr3-llc4mb]
DESIGN=FireBoomNoNIC
TARGET_CONFIG=FireSimBoomConfig
PLATFORM_CONFIG=FireSimDDR3FRFCFSLLC4MBConfig90MHz
instancetype=c4.4xlarge
deploytriplet=None

#[fireboom-singlecore-nic-lbp]
#DESIGN=FireBoom
#TARGET_CONFIG=FireSimBoomConfig
#PLATFORM_CONFIG=FireSimConfig
#instancetype=c4.4xlarge
#deploytriplet=None

[fireboom-singlecore-nic-ddr3-llc4mb]
DESIGN=FireBoom
TARGET_CONFIG=FireSimBoomConfig
PLATFORM_CONFIG=FireSimDDR3FRFCFSLLC4MBConfig90MHz
```

(continues on next page)

(continued from previous page)

```
instancetype=c4.4xlarge
deploytriplet=None

[firesim-supernode-singlecore-nic-lbp]
DESIGN=FireSimSupernode
TARGET_CONFIG=SupernodeFireSimRocketChipConfig
PLATFORM_CONFIG=FireSimConfig90MHz
instancetype=c4.4xlarge
deploytriplet=None

[firesim-supernode-quadcore-nic-lbp]
DESIGN=FireSimSupernode
TARGET_CONFIG=SupernodeFireSimRocketChipQuadCoreConfig
PLATFORM_CONFIG=FireSimConfig75MHz
instancetype=c4.4xlarge
deploytriplet=None

[firesim-supernode-singlecore-nic-ddr3-llc4mb]
DESIGN=FireSimSupernode
TARGET_CONFIG=SupernodeFireSimRocketChipConfig
PLATFORM_CONFIG=FireSimDDR3FRFCFSLLC4MBConfig90MHz
instancetype=c4.4xlarge
deploytriplet=None

[firesim-supernode-quadcore-nic-ddr3-llc4mb]
DESIGN=FireSimSupernode
TARGET_CONFIG=SupernodeFireSimRocketChipQuadCoreConfig
PLATFORM_CONFIG=FireSimDDR3FRFCFSLLC4MBConfig75MHz
instancetype=c4.4xlarge
deploytriplet=None

# MIDAS Examples -- BUILD SUPPORT ONLY; Can't launch driver correctly on runfarm
[midasexamples-gcd]
TARGET_PROJECT=midasexamples
DESIGN=GCD
TARGET_CONFIG=NoConfig
PLATFORM_CONFIG=DefaultF1Config
instancetype=c4.4xlarge
deploytriplet=None
```

Below, we outline each section and parameter in detail.

Build definition sections, e.g. `[awesome-firesim-config]`

In this file, you can specify as many build definition sections as you want, each with a header like `[awesome-firesim-config]` (i.e. a nice, short name you made up). Such a section must contain the following fields:

`DESIGN`

This specifies the basic target design that will be built. Unless you are defining a custom system, this should either be `FireSim`, for systems with a NIC, or `FireSimNoNIC`, for systems without a NIC. These are defined in `firesim/sim/src/main/scala/firesim/Targets.scala`.

TARGET_CONFIG

This specifies the hardware configuration of the target being simulation. Some examples include `FireSimRocketChipConfig` and `FireSimRocketChipQuadCoreConfig`. These are defined in `firesim/sim/src/main/scala/firesim/TargetConfigs.scala`.

PLATFORM_CONFIG

This specifies hardware parameters of the simulation environment - for example, selecting between a Latency-Bandwidth Pipe or DDR3 memory models. These are defined in `firesim/sim/src/main/scala/firesim/SimConfigs.scala`.

instancetype

This defines the type of instance that the build will run on. Generally, running on a `c4.4xlarge` is sufficient. In our experience, using more powerful instances than this provides little gain.

deploytriplet

This allows you to override the `deploytriplet` stored with the AGFI. Otherwise, the `DESIGN/TARGET_CONFIG/PLATFORM_CONFIG` you specify above will be used. See the AGFI Tagging section for more details. Most likely, you should leave this set to `None`. This is usually only used if you have proprietary RTL that you bake into an FPGA image, but don't want to share with users of the simulator.

TARGET_PROJECT *(Optional)*

This specifies the target project in which the target is defined (this is described in greater detail *here*). If `TARGET_PROJECT` is undefined the manager will default to `firesim`. Setting `TARGET_PROJECT` is required for building the MIDAS examples (`TARGET_PROJECT=midasexamples`) with the manager, or for building a user-provided target project.

5.4.4 config_hwdb.ini

Here is a sample of this configuration file:

```
# Hardware config database for FireSim Simulation Manager
# See docs/Advanced-Usage/Manager/Manager-Configuration-Files.rst for␣
↪documentation of all of these params.

# Hardware configs represent a combination of an agfi, a deploytriplet␣
↪override
# (if needed), and a custom runtime config (if needed)

# The AGFIs provided below are public and available to all users.
# Only AGFIs for the latest release of FireSim are guaranteed to be available.
# If you are using an older version of FireSim, you will need to generate your
# own images.

[fireboom-singlecore-nic-ddr3-llc4mb]
```

```
agfi=agfi-02abda61b5764288c
deploytripletoverride=None
customruntimeconfig=None

[fireboom-singlecore-no-nic-ddr3-llc4mb]
agfi=agfi-084aa1fcc32be2dcb
deploytripletoverride=None
customruntimeconfig=None

[firesim-quadcore-nic-ddr3-llc4mb]
agfi=agfi-0ca581654aeb2793d
deploytripletoverride=None
customruntimeconfig=None

[firesim-quadcore-no-nic-ddr3-llc4mb]
agfi=agfi-0fd2554e204e2b0e3
deploytripletoverride=None
customruntimeconfig=None

[firesim-singlecore-no-nic-lbp]
agfi=agfi-0594780e9a1888173
deploytripletoverride=None
customruntimeconfig=None

[firesim-supernode-singlecore-nic-lbp]
agfi=agfi-064190ff13a98df35
deploytripletoverride=None
customruntimeconfig=None
```

This file tracks hardware configurations that you can deploy as simulated nodes in FireSim. Each such configuration contains a name for easy reference in higher-level configurations, defined in the section header, an agfi, which represents the FPGA image, a custom runtime config, if one is needed, and a deploy triplet override if one is necessary.

When you build a new AGFI, you should put the default version of it in this file so that it can be referenced from your other configuration files.

The following is an example section from this file - you can add as many of these as necessary:

```
[firesim-quadcore-nic-ddr3-llc4mb]
# this is a comment that describes my favorite configuration!
agfi=agfi-0a6449b5894e96e53
deploytripletoverride=None
customruntimeconfig=None
```

[NAME_GOES_HERE]

In this example, `firesim-quadcore-nic-ddr3-llc4mb` is the name that will be used to reference this hardware design in other configuration locations. The following items describe this hardware configuration:

agfi

This represents the AGFI (FPGA Image) used by this hardware configuration.

deploytripletoverride

This is an advanced feature - under normal conditions, you should leave this set to `None`, so that the manager uses the configuration triplet that is automatically stored with the AGFI at build time. Advanced users can set this to a different value to build and use a different driver when deploying simulations. Since the driver depends on logic now hardwired into the FPGA bitstream, drivers cannot generally be changed without requiring FPGA recompilation.

customruntimeconfig

This is an advanced feature - under normal conditions, you can use the default parameters generated automatically by the simulator by setting this field to `None`. If you want to customize runtime parameters for certain parts of the simulation (e.g. the DRAM model's runtime parameters), you can place a custom config file in `sim/custom-runtime-configs/`. Then, set this field to the relative name of the config. For example, `sim/custom-runtime-configs/GREATCONFIG.conf` becomes `customruntimeconfig=GREATCONFIG.conf`.

Add more hardware config sections, like `[NAME_GOES_HERE_2]`

You can add as many of these entries to `config_hwdb.ini` as you want, following the format discussed above (i.e. you provide `agfi`, `deploytripletoverride`, or `customruntimeconfig`).

5.5 Manager Network Topology Definitions (`user_topology.py`)

Custom network topologies are specified as Python snippets that construct a tree. You can see examples of these in `firesim/deploy/runtools/user_topology.py`, shown below. Better documentation of this API will be available once it stabilizes.

Fundamentally, you create a list of roots, which consists of switch or server nodes, then construct a tree by adding downlinks to these roots. Since links are bi-directional, adding a downlink from node A to node B implicitly adds an uplink from B to A.

You can add additional topology generation methods here, then use them in `config_runtime.ini`.

5.5.1 `user_topology.py` contents:

```
""" Define your additional topologies here. The FireSimTopology class inherits
from UserToplogies and thus can instantiate your topology. """

from runtools.firesim_topology_elements import *


class UserTopologies(object):
    """ A class that just separates out user-defined/configurable topologies
    from the rest of the boilerplate in FireSimTopology() """

    def clos_m_n_r(self, m, n, r):
        """ DO NOT USE THIS DIRECTLY, USE ONE OF THE INSTANTIATIONS BELOW. """
        """ Clos topol where:
        m = number of root switches
        n = number of links to nodes on leaf switches
```

```
        r = number of leaf switches

        and each leaf switch has a link to each root switch.

        With the default mapping specified below, you will need:
        m m4.16xlarges.
        n f1.16xlarges.

        TODO: improve this later to pack leaf switches with <= 4 downlinks␣
↪onto
        one 16x.large.
        """

        rootswitches = [FireSimSwitchNode() for x in range(m)]
        self.roots = rootswitches
        leafswitches = [FireSimSwitchNode() for x in range(r)]
        servers = [[FireSimServerNode() for x in range(n)] for y in range(r)]
        for rswitch in rootswitches:
            rswitch.add_downlinks(leafswitches)

        for leafswitch, servergroup in zip(leafswitches, servers):
            leafswitch.add_downlinks(servergroup)

        def custom_mapper(fsim_topol_with_passes):
            for i, rswitch in enumerate(rootswitches):
                fsim_topol_with_passes.run_farm.m4_16s[i].add_switch(rswitch)

            for j, lswitch in enumerate(leafswitches):
                fsim_topol_with_passes.run_farm.f1_16s[j].add_switch(lswitch)
                for sim in servers[j]:
                    fsim_topol_with_passes.run_farm.f1_16s[j].add_
↪simulation(sim)

        self.custom_mapper = custom_mapper

    def clos_2_8_2(self):
        """ clos topol with:
        2 roots
        8 nodes/leaf
        2 leaves. """
        self.clos_m_n_r(2, 8, 2)

    def clos_8_8_16(self):
        """ clos topol with:
        8 roots
        8 nodes/leaf
        16 leaves. = 128 nodes."""
        self.clos_m_n_r(8, 8, 16)

    def fat_tree_4ary(self):
        # 4-ary fat tree as described in
        # http://ccr.sigcomm.org/online/files/p63-alfares.pdf
        coreswitches = [FireSimSwitchNode() for x in range(4)]
        self.roots = coreswitches
```

```
        aggrswitches = [FireSimSwitchNode() for x in range(8)]
        edgeswitches = [FireSimSwitchNode() for x in range(8)]
        servers = [FireSimServerNode() for x in range(16)]
        for switchno in range(len(coreswitches)):
            core = coreswitches[switchno]
            base = 0 if switchno < 2 else 1
            dls = range(base, 8, 2)
            dls = map(lambda x: aggrswitches[x], dls)
            core.add_downlinks(dls)
        for switchbaseno in range(0, len(aggrswitches), 2):
            switchno = switchbaseno + 0
            aggr = aggrswitches[switchno]
            aggr.add_downlinks([edgeswitches[switchno], edgeswitches[switchno+1]])
            switchno = switchbaseno + 1
            aggr = aggrswitches[switchno]
            aggr.add_downlinks([edgeswitches[switchno-1],␣
↪edgeswitches[switchno]])
        for edgeno in range(len(edgeswitches)):
            edgeswitches[edgeno].add_downlinks([servers[edgeno*2],␣
↪servers[edgeno*2+1]])


        def custom_mapper(fsim_topol_with_passes):
            """ In a custom mapper, you have access to the firesim topology␣
↪with passes,
            where you can access the run_farm nodes:

            fsim_topol_with_passes.run_farm.{f1_16s, f1_2s, m4_16s}

            To map, call add_switch or add_simulation on run farm instance
            objs in the aforementioned arrays.

            Because of the scope of this fn, you also have access to whatever
            stuff you created in the topology itself, which we expect will be
            useful for performing the mapping."""

            # map the fat tree onto one m4.16xlarge (for core switches)
            # and two f1.16xlarges (two pods of aggr/edge/4sims per f1.
↪16xlarge)
            for core in coreswitches:
                fsim_topol_with_passes.run_farm.m4_16s[0].add_switch(core)

            for aggrsw in aggrswitches[:4]:
                fsim_topol_with_passes.run_farm.f1_16s[0].add_switch(aggrsw)
            for aggrsw in aggrswitches[4:]:
                fsim_topol_with_passes.run_farm.f1_16s[1].add_switch(aggrsw)

            for edgesw in edgeswitches[:4]:
                fsim_topol_with_passes.run_farm.f1_16s[0].add_switch(edgesw)
            for edgesw in edgeswitches[4:]:
                fsim_topol_with_passes.run_farm.f1_16s[1].add_switch(edgesw)

            for sim in servers[:8]:
                fsim_topol_with_passes.run_farm.f1_16s[0].add_simulation(sim)
```

```
            for sim in servers[8:]:
                fsim_topol_with_passes.run_farm.f1_16s[1].add_simulation(sim)

        self.custom_mapper = custom_mapper

    def example_multilink(self):
        self.roots = [FireSimSwitchNode()]
        midswitch = FireSimSwitchNode()
        lowerlayer = [midswitch for x in range(16)]
        self.roots[0].add_downlinks(lowerlayer)
        servers = [FireSimServerNode()]
        midswitch.add_downlinks(servers)

    def example_multilink_32(self):
        self.roots = [FireSimSwitchNode()]
        midswitch = FireSimSwitchNode()
        lowerlayer = [midswitch for x in range(32)]
        self.roots[0].add_downlinks(lowerlayer)
        servers = [FireSimServerNode()]
        midswitch.add_downlinks(servers)

    def example_multilink_64(self):
        self.roots = [FireSimSwitchNode()]
        midswitch = FireSimSwitchNode()
        lowerlayer = [midswitch for x in range(64)]
        self.roots[0].add_downlinks(lowerlayer)
        servers = [FireSimServerNode()]
        midswitch.add_downlinks(servers)

    def example_cross_links(self):
        self.roots = [FireSimSwitchNode() for x in range(2)]
        midswitches = [FireSimSwitchNode() for x in range(2)]
        self.roots[0].add_downlinks(midswitches)
        self.roots[1].add_downlinks(midswitches)
        servers = [FireSimServerNode() for x in range(2)]
        midswitches[0].add_downlinks([servers[0]])
        midswitches[1].add_downlinks([servers[1]])

    def small_hierarchy_8sims(self):
        self.custom_mapper = 'mapping_use_one_f1_16xlarge'
        self.roots = [FireSimSwitchNode()]
        midlevel = [FireSimSwitchNode() for x in range(4)]
        servers = [[FireSimServerNode() for x in range(2)] for x in range(4)]
        self.roots[0].add_downlinks(midlevel)
        for swno in range(len(midlevel)):
            midlevel[swno].add_downlinks(servers[swno])

    def small_hierarchy_2sims(self):
        self.custom_mapper = 'mapping_use_one_f1_16xlarge'
        self.roots = [FireSimSwitchNode()]
        midlevel = [FireSimSwitchNode() for x in range(1)]
        servers = [[FireSimServerNode() for x in range(2)] for x in range(1)]
        self.roots[0].add_downlinks(midlevel)
        for swno in range(len(midlevel)):
```

```
            midlevel[swno].add_downlinks(servers[swno])

    def example_1config(self):
        self.roots = [FireSimSwitchNode()]
        servers = [FireSimServerNode() for y in range(1)]
        self.roots[0].add_downlinks(servers)

    def example_2config(self):
        self.roots = [FireSimSwitchNode()]
        servers = [FireSimServerNode() for y in range(2)]
        self.roots[0].add_downlinks(servers)

    def example_4config(self):
        self.roots = [FireSimSwitchNode()]
        servers = [FireSimServerNode() for y in range(4)]
        self.roots[0].add_downlinks(servers)

    def example_8config(self):
        self.roots = [FireSimSwitchNode()]
        servers = [FireSimServerNode() for y in range(8)]
        self.roots[0].add_downlinks(servers)

    def example_16config(self):
        self.roots = [FireSimSwitchNode()]
        level2switches = [FireSimSwitchNode() for x in range(2)]
        servers = [[FireSimServerNode() for y in range(8)] for x in range(2)]

        for root in self.roots:
            root.add_downlinks(level2switches)

        for l2switchNo in range(len(level2switches)):
            level2switches[l2switchNo].add_downlinks(servers[l2switchNo])

    def example_32config(self):
        self.roots = [FireSimSwitchNode()]
        level2switches = [FireSimSwitchNode() for x in range(4)]
        servers = [[FireSimServerNode() for y in range(8)] for x in range(4)]

        for root in self.roots:
            root.add_downlinks(level2switches)

        for l2switchNo in range(len(level2switches)):
            level2switches[l2switchNo].add_downlinks(servers[l2switchNo])

    def example_64config(self):
        self.roots = [FireSimSwitchNode()]
        level2switches = [FireSimSwitchNode() for x in range(8)]
        servers = [[FireSimServerNode() for y in range(8)] for x in range(8)]

        for root in self.roots:
            root.add_downlinks(level2switches)

        for l2switchNo in range(len(level2switches)):
            level2switches[l2switchNo].add_downlinks(servers[l2switchNo])
```

```
    def example_128config(self):
        self.roots = [FireSimSwitchNode()]
        level1switches = [FireSimSwitchNode() for x in range(2)]
        level2switches = [[FireSimSwitchNode() for x in range(8)] for x in␣
↪range(2)]
        servers = [[[FireSimServerNode() for y in range(8)] for x in␣
↪range(8)] for x in range(2)]

        self.roots[0].add_downlinks(level1switches)

        for switchno in range(len(level1switches)):
            level1switches[switchno].add_downlinks(level2switches[switchno])

        for switchgroupno in range(len(level2switches)):
            for switchno in range(len(level2switches[switchgroupno])):
                level2switches[switchgroupno][switchno].add_
↪downlinks(servers[switchgroupno][switchno])

    def example_256config(self):
        self.roots = [FireSimSwitchNode()]
        level1switches = [FireSimSwitchNode() for x in range(4)]
        level2switches = [[FireSimSwitchNode() for x in range(8)] for x in␣
↪range(4)]
        servers = [[[FireSimServerNode() for y in range(8)] for x in␣
↪range(8)] for x in range(4)]

        self.roots[0].add_downlinks(level1switches)

        for switchno in range(len(level1switches)):
            level1switches[switchno].add_downlinks(level2switches[switchno])

        for switchgroupno in range(len(level2switches)):
            for switchno in range(len(level2switches[switchgroupno])):
                level2switches[switchgroupno][switchno].add_
↪downlinks(servers[switchgroupno][switchno])

    @staticmethod
    def supernode_flatten(arr):
        res = []
        for x in arr:
            res = res + x
        return res

    def supernode_example_6config(self):
        self.roots = [FireSimSwitchNode()]
        servers = [FireSimSuperNodeServerNode()] + [FireSimDummyServerNode()␣
↪for x in range(5)]
        self.roots[0].add_downlinks(servers)

    def supernode_example_4config(self):
        self.roots = [FireSimSwitchNode()]
        servers = [FireSimSuperNodeServerNode()] + [FireSimDummyServerNode()␣
↪for x in range(3)]
```

```
        self.roots[0].add_downlinks(servers)
    def supernode_example_8config(self):
        self.roots = [FireSimSwitchNode()]
        servers = UserTopologies.supernode_flatten([[FireSimSuperNodeServerNode(),
↪ FireSimDummyServerNode(), FireSimDummyServerNode(), FireSimDummyServerNode()]␣
↪for y in range(2)])
        self.roots[0].add_downlinks(servers)
    def supernode_example_16config(self):
        self.roots = [FireSimSwitchNode()]
        servers = UserTopologies.supernode_flatten([[FireSimSuperNodeServerNode(),
↪ FireSimDummyServerNode(), FireSimDummyServerNode(), FireSimDummyServerNode()]␣
↪for y in range(4)])
        self.roots[0].add_downlinks(servers)
    def supernode_example_32config(self):
        self.roots = [FireSimSwitchNode()]
        servers = UserTopologies.supernode_flatten([[FireSimSuperNodeServerNode(),
↪ FireSimDummyServerNode(), FireSimDummyServerNode(), FireSimDummyServerNode()]␣
↪for y in range(8)])
        self.roots[0].add_downlinks(servers)

    def supernode_example_64config(self):
        self.roots = [FireSimSwitchNode()]
        level2switches = [FireSimSwitchNode() for x in range(2)]
        servers = [UserTopologies.supernode_flatten([[FireSimSuperNodeServerNode(),
↪ FireSimDummyServerNode(), FireSimDummyServerNode(), FireSimDummyServerNode()]␣
↪for y in range(8)]) for x in range(2)]
        for root in self.roots:
            root.add_downlinks(level2switches)
        for l2switchNo in range(len(level2switches)):
            level2switches[l2switchNo].add_downlinks(servers[l2switchNo])

    def supernode_example_128config(self):
        self.roots = [FireSimSwitchNode()]
        level2switches = [FireSimSwitchNode() for x in range(4)]
        servers = [UserTopologies.supernode_flatten([[FireSimSuperNodeServerNode(),
↪ FireSimDummyServerNode(), FireSimDummyServerNode(), FireSimDummyServerNode()]␣
↪for y in range(8)]) for x in range(4)]
        for root in self.roots:
            root.add_downlinks(level2switches)
        for l2switchNo in range(len(level2switches)):
            level2switches[l2switchNo].add_downlinks(servers[l2switchNo])

    def supernode_example_256config(self):
        self.roots = [FireSimSwitchNode()]
        level2switches = [FireSimSwitchNode() for x in range(8)]
        servers = [UserTopologies.supernode_flatten([[FireSimSuperNodeServerNode(),
↪ FireSimDummyServerNode(), FireSimDummyServerNode(), FireSimDummyServerNode()]␣
↪for y in range(8)]) for x in range(8)]
        for root in self.roots:
            root.add_downlinks(level2switches)
        for l2switchNo in range(len(level2switches)):
            level2switches[l2switchNo].add_downlinks(servers[l2switchNo])

    def supernode_example_512config(self):
```

```
        self.roots = [FireSimSwitchNode()]
        level1switches = [FireSimSwitchNode() for x in range(2)]
        level2switches = [[FireSimSwitchNode() for x in range(8)] for x in␣
↪range(2)]
        servers = [[UserTopologies.supernode_flatten([[FireSimSuperNodeServerNode(),
↪ FireSimDummyServerNode(), FireSimDummyServerNode(), FireSimDummyServerNode()]␣
↪for y in range(8)]) for x in range(8)] for x in range(2)]
        self.roots[0].add_downlinks(level1switches)
        for switchno in range(len(level1switches)):
            level1switches[switchno].add_downlinks(level2switches[switchno])
        for switchgroupno in range(len(level2switches)):
            for switchno in range(len(level2switches[switchgroupno])):
                level2switches[switchgroupno][switchno].add_
↪downlinks(servers[switchgroupno][switchno])

    def supernode_example_1024config(self):
        self.roots = [FireSimSwitchNode()]
        level1switches = [FireSimSwitchNode() for x in range(4)]
        level2switches = [[FireSimSwitchNode() for x in range(8)] for x in␣
↪range(4)]
        servers = [[UserTopologies.supernode_flatten([[FireSimSuperNodeServerNode(),
↪ FireSimDummyServerNode(), FireSimDummyServerNode(), FireSimDummyServerNode()]␣
↪for y in range(8)]) for x in range(8)] for x in range(4)]
        self.roots[0].add_downlinks(level1switches)
        for switchno in range(len(level1switches)):
            level1switches[switchno].add_downlinks(level2switches[switchno])
        for switchgroupno in range(len(level2switches)):
            for switchno in range(len(level2switches[switchgroupno])):
                level2switches[switchgroupno][switchno].add_
↪downlinks(servers[switchgroupno][switchno])

    def supernode_example_deep64config(self):
        self.roots = [FireSimSwitchNode()]
        level1switches = [FireSimSwitchNode() for x in range(2)]
        level2switches = [[FireSimSwitchNode() for x in range(1)] for x in␣
↪range(2)]
        servers = [[UserTopologies.supernode_flatten([[FireSimSuperNodeServerNode(),
↪ FireSimDummyServerNode(), FireSimDummyServerNode(), FireSimDummyServerNode()]␣
↪for y in range(8)]) for x in range(1)] for x in range(2)]
        self.roots[0].add_downlinks(level1switches)
        for switchno in range(len(level1switches)):
            level1switches[switchno].add_downlinks(level2switches[switchno])
        for switchgroupno in range(len(level2switches)):
            for switchno in range(len(level2switches[switchgroupno])):
                level2switches[switchgroupno][switchno].add_
↪downlinks(servers[switchgroupno][switchno])

    def dual_example_8config(self):
        """ two separate 8-node clusters for experiments, e.g. memcached␣
↪mutilate. """
        self.roots = [FireSimSwitchNode(), FireSimSwitchNode()]
        servers = [FireSimServerNode() for y in range(8)]
        servers2 = [FireSimServerNode() for y in range(8)]
        self.roots[0].add_downlinks(servers)
```

```
        self.roots[1].add_downlinks(servers2)

    def triple_example_8config(self):
        """ three separate 8-node clusters for experiments, e.g. memcached␣
↪mutilate. """
        self.roots = [FireSimSwitchNode(), FireSimSwitchNode(),␣
↪FireSimSwitchNode()]
        servers = [FireSimServerNode() for y in range(8)]
        servers2 = [FireSimServerNode() for y in range(8)]
        servers3 = [FireSimServerNode() for y in range(8)]
        self.roots[0].add_downlinks(servers)
        self.roots[1].add_downlinks(servers2)
        self.roots[2].add_downlinks(servers3)

    def no_net_config(self):
        self.roots = [FireSimServerNode() for x in range(self.no_net_num_
↪nodes)]
```

5.6 AGFI Metadata/Tagging

When you build an AGFI in FireSim, the AGFI description stored by AWS is populated with metadata that helps the manager decide how to deploy a simulation. The important metadata is listed below, along with how each field is set and used:

- `firesim-buildtriplet`: This always reflects the triplet combination used to BUILD the AGFI.
- `firesim-deploytriplet`: This reflects the triplet combination that is used to DEPLOY the AGFI. By default, this is the same as `firesim-buildtriplet`. In certain cases however, your users may not have access to a particular configuration, but a simpler configuration may be sufficient for building a compatible software driver (e.g. if you have proprietary RTL in your FPGA image that doesn't interface with the outside system). In this case, you can specify a custom deploytriplet at build time. If you do not do so, the manager will automatically set this to be the same as `firesim-buildtriplet`.
- `firesim-commit`: This is the commit hash of the version of FireSim used to build this AGFI. If the AGFI was created from a dirty copy of the FireSim repo, "-dirty" will be appended to the commit hash.

CHAPTER 6

Workloads

Attention: FireSim is moving to a new workload-generation tool *FireMarshal (alpha)*. These instructions will be deprecated in future releases of FireSim.

This section describes workload definitions in FireSim.

6.1 Defining Custom Workloads

Workloads in FireSim consist of a series of **Jobs** that are assigned to be run on individual simulations. Currently, we require that a Workload defines either:

- A single type of job, that is run on as many simulations as specfied by the user. These workloads are usually suffixed with `-uniform`, which indicates that all nodes in the workload run the same job. An example of such a workload is `firesim/deploy/workloads/linux-uniform.json`.
- Several different jobs, in which case there must be exactly as many jobs as there are running simulated nodes. An example of such a workload is `firesim/deploy/workloads/ping-latency.json`.

FireSim supports can take these workload definitions and perform two functions:

- Building workloads using `firesim/deploy/workloads/gen-benchmark-rootfs.py`
- Deploying workloads using the manager

In the following subsections, we will go through the two aforementioned example workload configurations, describing how these two functions use each part of the json file inline.

ERRATA: You will notice in the following json files the field "workloads" this should really be named "jobs" – we will fix this in a future release.

ERRATA: The following instructions assume the default buildroot-based linux distribution (br-disk). In order to customize Fedora, you should build the basic Fedora image (as described in *Running Fedora on FireSim*) and modify the image directly (or use *FireMarshal* to generate the workload). Imporantly, Fedora currently does not support the "command" option for workloads.

6.1.1 Uniform Workload JSON

`firesim/deploy/workloads/linux-uniform.json` is an example of a "uniform" style workload, where each simulated node runs the same software configuration.

Let's take a look at this file:

```
{
  "benchmark_name" : "linux-uniform",
  "common_bootbinary" : "br-base-bin",
  "common_rootfs"     : "br-base.img",
  "common_outputs"    : ["/etc/os-release"],
  "common_simulation_outputs" : ["uartlog", "memory_stats.csv"]
}
```

There is also a corresponding directory named after this workload/file:

```
centos@ip-172-30-2-111.us-west-2.compute.internal:~/firesim-new/deploy/workloads/
↪linux-uniform$ ls -la
total 4
drwxrwxr-x  2 centos centos   42 May 17 21:58 .
drwxrwxr-x 13 centos centos 4096 May 18 17:14 ..
lrwxrwxrwx  1 centos centos   41 May 17 21:58 br-base-bin -> ../../../sw/firesim-
↪software/images/br-base-bin
lrwxrwxrwx  1 centos centos   41 May 17 21:58 br-base.img -> ../../../sw/firesim-
↪software/images/br-base.img
```

We will elaborate on this later.

Looking at the JSON file, you'll notice that this is a relatively simple workload definition.

In this "uniform" case, the manager will name simulations after the `benchmark_name` field, appending a number for each simulation using the workload (e.g. `linux-uniform0`, `linux-uniform1`, and so on). It is standard pratice to keep `benchmark_name`, the json filename, and the above directory name the same. In this case, we have set all of them to `linux-uniform`.

Next, the `common_bootbinary` field represents the binary that the simulations in this workload are expected to boot from. The manager will copy this binary for each of the nodes in the simulation (each gets its own copy). The `common_bootbinary` path is relative to the workload's directory, in this case `firesim/deploy/workloads/linux-uniform`. You'll notice in the above output from `ls -la` that this is actually just a symlink to `br-base-bin` that is built by the *FireMarshal* tool.

Similarly, the `common_rootfs` field represents the disk image that the simulations in this workload are expected to boot from. The manager will copy this root filesystem image for each of the nodes in the simulation (each gets its own copy). The `common_rootfs` path is relative to the workload's directory, in this case `firesim/deploy/workloads/linux-uniform`. You'll notice in the above output from `ls -la` that this is actually just a symlink to `br-base.img` that is built by the *FireMarshal* tool.

The `common_outputs` field is a list of outputs that the manager will copy out of the root filesystem image AFTER a simulation completes. In this simple example, when a workload running on a simulated cluster with `firesim runworkload` completes, `/etc/os-release` will be copied out from each rootfs and placed in the job's output directory within the workload's output directory (See the *firesim runworkload* section). You can add multiple paths here.

The `common_simulation_outputs` field is a list of outputs that the manager will copy off of the simulation host machine AFTER a simulation completes. In this example, when a workload running on a simulated cluster with `firesim runworkload` completes, the `uartlog` (an automatically generated file that contains the full console output of the simulated system) and `memory_stats.csv` files will be copied out of the simulation's base directory

on the host instance and placed in the job's output directory within the workload's output directory (see the *firesim runworkload* section). You can add multiple paths here.

ERRATA: "Uniform" style workloads currently do not support being automatically built – you can currently hack around this by building the rootfs as a single-node non-uniform workload, then deleting the `workloads` field of the JSON to make the manager treat it as a uniform workload. This will be fixed in a future release.

6.1.2 Non-uniform Workload JSON (explicit job per simulated node)

Now, we'll look at the `ping-latency` workload, which explicitly defines a job per simulated node.

```
{
  "common_bootbinary" : "bbl-vmlinux",
  "benchmark_name" : "ping-latency",
  "deliver_dir" : "/",
  "common_args" : [],
  "common_files" : ["bin/pinglatency.sh"],
  "common_outputs" : [],
  "common_simulation_outputs" : ["uartlog"],
  "no_post_run_hook": "",
  "workloads" : [
    {
      "name": "pinger",
      "files": [],
      "command": "pinglatency.sh && poweroff -f",
      "simulation_outputs": [],
      "outputs": []
    },
    {
      "name": "pingee",
      "files": [],
      "command": "while true; do sleep 1000; done",
      "simulation_outputs": [],
      "outputs": []
    },
    {
      "name": "idler-1",
      "files": [],
      "command": "while true; do sleep 1000; done",
      "simulation_outputs": [],
      "outputs": []
    },
    {
      "name": "idler-2",
      "files": [],
      "command": "while true; do sleep 1000; done",
      "simulation_outputs": [],
      "outputs": []
    },
    {
      "name": "idler-3",
      "files": [],
      "command": "while true; do sleep 1000; done",
      "simulation_outputs": [],
      "outputs": []
    },
    {
```

(continues on next page)

(continued from previous page)

```
      "name": "idler-4",
      "files": [],
      "command": "while true; do sleep 1000; done",
      "simulation_outputs": [],
      "outputs": []
    },
    {
      "name": "idler-5",
      "files": [],
      "command": "while true; do sleep 1000; done",
      "simulation_outputs": [],
      "outputs": []
    },
    {
      "name": "idler-6",
      "files": [],
      "command": "while true; do sleep 1000; done",
      "simulation_outputs": [],
      "outputs": []
    }
  ]
}
```

Additionally, let's take a look at the state of the `ping-latency` directory AFTER the workload is built:

```
centos@ip-172-30-2-111.us-west-2.compute.internal:~/firesim-new/deploy/workloads/ping-
↪latency$ ls -la
total 15203216
drwxrwxr-x  3 centos centos       4096 May 18 07:45 .
drwxrwxr-x 13 centos centos       4096 May 18 17:14 ..
lrwxrwxrwx  1 centos centos         41 May 17 21:58 bbl-vmlinux -> ../linux-uniform/
↪br-base-bin
-rw-rw-r--  1 centos centos          7 May 17 21:58 .gitignore
-rw-r--r--  1 centos centos 1946009600 May 18 07:45 idler-1.ext2
-rw-r--r--  1 centos centos 1946009600 May 18 07:45 idler-2.ext2
-rw-r--r--  1 centos centos 1946009600 May 18 07:45 idler-3.ext2
-rw-r--r--  1 centos centos 1946009600 May 18 07:45 idler-4.ext2
-rw-r--r--  1 centos centos 1946009600 May 18 07:45 idler-5.ext2
-rw-r--r--  1 centos centos 1946009600 May 18 07:46 idler-6.ext2
drwxrwxr-x  3 centos centos         16 May 17 21:58 overlay
-rw-r--r--  1 centos centos 1946009600 May 18 07:44 pingee.ext2
-rw-r--r--  1 centos centos 1946009600 May 18 07:44 pinger.ext2
-rw-rw-r--  1 centos centos       2236 May 17 21:58 ping-latency-graph.py
```

First, let's identify some of these files:

- `bbl-vmlinux`: This workload just uses the default linux binary generated for the `linux-uniform` workload.
- `.gitignore`: This just ignores the generated rootfses, which we'll learn about below.
- `idler-[1-6].ext2`, `pingee.ext2`, `pinger.ext2`: These are rootfses that are generated from the json script above. We'll learn how to do this shortly.

Additionally, let's look at the `overlay` subdirectory:

```
centos@ip-172-30-2-111.us-west-2.compute.internal:~/firesim-new/deploy/workloads/ping-
↪latency/overlay$ ls -la */*
-rwxrwxr-x 1 centos centos 249 May 17 21:58 bin/pinglatency.sh
```

This is a file that's actually committed to the repo, that runs the benchmark we want to run on one of our simulated systems. We'll see how this is used soon.

Now, let's take a look at how we got here. First, let's review some of the new fields present in this JSON file:

- `common_files`: This is an array of files that will be included in ALL of the job rootfses when they're built. This is relative to a path that we'll pass to the script that generates rootfses.
- `workloads`: This time, you'll notice that we have this array, which is populated by objects that represent individual jobs. Each job has some additional fields:
  - `name`: In this case, jobs are each assigned a name manually. These names MUST BE UNIQUE within a particular workload.
  - `files`: Just like `common_files`, but specific to this job.
  - `command`: This is the command that will be run automatically immediately when the simulation running this job boots up. This is usually the command that starts the workload we want.
  - `simulation_outputs`: Just like `common_simulation_outputs`, but specific to this job.
  - `outputs`: Just like `common_outputs`, but specific to this job.

In this example, we specify one node that boots up and runs the `pinglatency.sh` benchmark, then powers off cleanly and 7 nodes that just idle waiting to be pinged.

Given this JSON description, our existing `pinglatency.sh` script in the overlay directory, and the base rootfses generated in `firesim-software`, the following command will automatically generate all of the rootfses that you see in the `ping-latency` directory.

```
[ from the workloads/ directory ]
python gen-benchmark-rootfs.py -w ping-latency.json -r -b ../../sw/firesim-software/
↪images/br-base.img -s ping-latency/overlay
```

Notice that we tell this script where the json file lives, where the base rootfs image is, and where we expect to find files that we want to include in the generated disk images. This script will take care of the rest and we'll end up with `idler-[1-6].ext2`, `pingee.ext2`, and `pinger.ext2`!

You'll notice a Makefile in the `workloads/` directory – it contains many similar commands for all of the workloads included with FireSim.

Once you generate the rootfses for this workload, you can run it with the manager by setting `workload=ping-latency.json` in `config_runtime.ini`. The manager will automatically look for the generated rootfses (based on workload and job names that it reads from the json) and distribute work appropriately.

Just like in the uniform case, it will copy back the results that we specify in the json file. We'll end up with a directory in `firesim/deploy/results-workload/` named after the workload name, with a subdirectory named after each job in the workload, which will contain the output files we want.

6.2 SPEC 2017

SPEC2017 is supported using the `firesim-2017` branch of Speckle, which provides the tooling required to cross-compile SPEC for RISCV. These instructions presuppose you've have a license for, and have installed SPEC on your machine either EC2 or locally. Additionally, your SPEC environment must be setup; `SPEC_DIR` must be set. If

you are building binaries on a different machine, you should be able to trivially copy Speckle's generated overlay directories to EC2.

Some notes:

- Benchmarks use reference inputs by default. `train` or `test` inputs can be used by specifying an argument in make: `make spec-int{rate,speed} input={test,train,ref}`
- You may need to increase the size of the RootFS in buildroot in `firesim/sw/firesim-software/images`.
- No support for fp{rate, speed} benchmarks yet.

Attention: Regarding BOOM. Users wishing to run SPEC on BOOM must change the default architecture from `-march=rv64imafdc` to `-march=rv64imafd` in Speckle's target SPEC configuration (`riscv.cfg`), as BOOM does not support compressed instructions

6.2.1 Intspeed

The intspeed workload definition splits the xz benchmark into two jobs (these are two independent inputs) to achieve better load balance across the simulations (9T dynamic instructions becomes 4T and 5T.)

To Build Binaries And RootFSes:

```
cd firesim/deploy/workloads/
make spec17-intspeed
```

Run Resource requirements:

```
f1_16xlarges=0
m4_16xlarges=0
f1_2xlarges=11
```

To Run:

```
./run-workload.sh workloads/spec17-intspeed.ini --withlaunch
```

On a single-core rocket-based SoC with a DDR3 + 256 KiB LLC model, with a 160 MHz host clock, the longest benchmarks (xz, mcf) complete in about 1 day. All other benchmarks finish in under 15 hours.

6.2.2 Intrate

By default, the intrate workload definition spins up **four copies** of each benchmark, which may be entirely inappropriate for your target machine. This can be changed by modifying the json.

To Build Binaries and RootFSes:

```
cd firesim/deploy/workloads/
make spec17-intrate
```

Run Resource Requirements:

```
f1_16xlarges=0
m4_16xlarges=0
f1_2xlarges=10
```

To Run:

```
./run-workload.sh workloads/spec17-intrate.ini --withlaunch
```

Simulation times are host and target dependent. For reference, on a four-core rocket-based SoC with a DDR3 + 1 MiB LLC model, with a 160 MHz host clock, the longest benchmarks complete in about 30 hours when running four copies.

6.3 Running Fedora on FireSim

All workload-generation related commands and code are in `firesim/sw/firesim-software`.

FireMarshal comes with a Fedora-based workload that you can use right out of the box in `workloads/fedora-base.json`. We begin by building the workload (filesystem and boot-binary):

```
./marshal build workloads/fedora-base.json
```

The first time you build a workload may take a long time (we need to download and decompress a pre-built fedora image), but subsequent builds of the same base will use cached results. Once the command completes, you should see two new files in `images/`: `fedora-base-bin` and `fedora-base.img`. These are the boot-binary (linux + boot loader) and root filesystem (respectively). We can now launch this workload in qemu:

```
./marshal launch workloads/fedora-base.json
```

You should now see linux booting and be presented with a login prompt. Sign in as 'root' with password 'firesim'. From here you can download files, use the package manager (e.g. 'dnf install'), and generally use the image as if it had booted on real hardware with an internet connection. Any changes you make here will be persistent between reboots. Once you are done exploring, simply shutdown the workload:

```
$ poweroff
```

It is typically not a good idea to modify the *-base workloads directly since many other workloads might inherit those changes. To make sure that we've cleaned out any changes, let's clean and rebuild the workload:

```
./marshal clean workloads/fedora-base.json
./marshal build workloads/fedora-base.json
```

Note that this build took significantly less time than the first; FireMarshal caches intermediate build steps whenever possible. The final step is to run this workload on the real firesim RTL with full timing accuracy. For the basic fedora distribution, we will use the pre-made firesim config at `firesim/deploy/workloads/fedora-uniform.json`. Simply change the `workloadname` option in `firesim/deploy/config_runtime.ini` to "fedora-uniform.json" and then follow the standard FireSim procedure for booting a workload (e.g. *Running a Single Node Simulation* or *Running a Cluster Simulation*).

Attention: For the standard distributions we provide pre-built firesim workloads. In general, FireMarshal can derive a FireSim workload from the FireMarshal configuration using the `install` command (see *FireMarshal Commands*)

6.4 ISCA 2018 Experiments

This page contains descriptions of the experiments in our ISCA 2018 paper and instructions for reproducing them on your own simulations.

One important difference between the configuration used in the ISCA 2018 paper and the open-source release of FireSim is that the ISCA paper used a proprietary L2 cache design that is not open-source. Instead, the open-source FireSim uses an LLC model that models the behavior of having an L2 cache as part of the memory model. Even with the LLC model, you should be able to see the same trends in these experiments, but exact numbers may vary.

Each section below describes the resources necessary to run the experiment. Some of these experiments require a large number of instances – you should make sure you understand the resource requirements before you run one of the scripts.

Compatiblity: These were last tested with commit `bba9dea4811a2445f22809ef226cf00971674758` of FireSim.

6.4.1 Prerequisites

These guides assume that you have previously followed the single-node/cluster-scale experiment guides in the FireSim documentation. Note that these are **advanced** experiments, not introductory tutorials.

6.4.2 Building Benchmark Binaries/Rootfses

We include scripts to automatically build all of the benchmark rootfs images that will be used below. To build them, make sure you have already run `./marshal build workloads/br-base.json` in `firesim/sw/firesim-software`, then run:

```
cd firesim/deploy/workloads/
make allpaper
```

6.4.3 Figure 5: Ping Latency vs. Configured Link Latency

Resource requirements:

```
f1_16xlarges=1
m4_16xlarges=0
f1_2xlarges=0
```

To Run:

```
cd firesim/deploy/workloads/
./run-ping-latency.sh withlaunch
```

6.4.4 Figure 6: Network Bandwidth Saturation

Resource requirements:

```
f1_16xlarges=2
m4_16xlarges=1
f1_2xlarges=0
```

To Run:

```
cd firesim/deploy/workloads/
./run-bw-test.sh withlaunch
```

6.4.5 Figure 7: Memcached QoS / Thread Imbalance

Resource requirements:

```
f1_16xlarges=3
m4_16xlarges=0
f1_2xlarges=0
```

To Run:

```
cd firesim/deploy/workloads/
./run-memcached-thread-imbalance.sh withlaunch
```

6.4.6 Figure 8: Simulation Rate vs. Scale

Resource requirements:

```
f1_16xlarges=32
m4_16xlarges=5
f1_2xlarges=0
```

To Run:

```
cd firesim/deploy/workloads/
./run-simperf-test-scale.sh withlaunch
```

A similar benchmark is also provided for supernode mode, see `run-simperf-test-scale-supernode.sh`.

6.4.7 Figure 9: Simulation Rate vs. Link Latency

Resource requirements:

```
f1_16xlarges=1
m4_16xlarges=0
f1_2xlarges=0
```

To Run:

```
cd firesim/deploy/workloads/
./run-simperf-test-latency.sh withlaunch
```

A similar benchmark for supernode mode will be provided soon. See https://github.com/firesim/firesim/issues/244

6.4.8 Running all experiments at once

This script simply executes all of the above scripts in parallel. One caveat is that the bw-test script currently cannot run in parallel with the others, since it requires patching the switches. This will be resolved in a future release.

```
cd firesim/deploy/workloads/
./run-all.sh
```

6.5 GAP Benchmark Suite

You can run the reference implementation of the GAP (Graph Algorithm Performance) Benchmark Suite. We provide scripts that cross-compile the graph kernels for RISCV.

For more information about the benchmark itself, please refer to the site: http://gap.cs.berkeley.edu/benchmark.html

Some notes:

- Only the Kron input graph is currently supported.
- Benchmark uses `graph500` input graph size of 2^20 vertices by default. `test` input size has 2^10 vertices and can be used by specifying an argument into make: `make gapbs input=test`
- The reference input size with 2^27 verticies is not currently supported.

By default, the gapbs workload definition runs the benchmark multithreaded with number of threads equal to the number of cores. To change the number of threads, you need to edit `firesim/deploy/workloads/runscripts/gapbs-scripts/gapbs.sh`. Additionally, the workload does not verify the output of the benchmark by default. To change this, add a `--verify` parameter to the json.

To Build Binaries and RootFSes:

```
cd firesim/deploy/workloads/
make gapbs
```

Run Resource Requirements:

```
f1_16xlarges=0
m4_16xlarges=0
f1_2xlarges=6
```

To Run:

```
./run-workload.sh workloads/gapbs.ini --withlaunch
```

Simulation times are host and target dependent. For reference, on a four-core rocket-based SoC with a DDR3 + 1 MiB LLC model, with a 90 MHz host clock, `test` and `graph500` input sizes finish in a few minutes.

CHAPTER 7

FireMarshal (alpha)

Attention: FireMarshal is still in alpha. You are encouraged to try it out and use it for new workloads. The old-style workload generation is still supported (see *Defining Custom Workloads* for details).

Workload generation in FireSim is handled by a tool called **FireMarshal** in `firesim/sw/firesim-software/`.

Workloads in FireMarshal consist of a series of **Jobs** that are assigned to logical nodes in the target system. If no jobs are specified, then the workload is considered `uniform` and only a single image will be produced for all nodes in the system. Workloads are described by a json file and a corresponding workload directory and can inherit their definitions from existing workloads. Typically, workload configurations are kept in `workloads` although you can use any directory you like. We provide a few basic workloads to start with including buildroot or Fedora-based linux distributions and bare-metal.

Once you define a workload, the `marshal` command will produce a corresponding boot-binary and rootfs for each job in the workload. This binary and rootfs can then be launched on qemu or spike (for functional simulation), or installed to firesim for running on real RTL.

7.1 Quick Start

Attention: FireMarshal is still in alpha. You are encouraged to try it out and use it for new workloads. The old-style workload generation is still supported (see *Defining Custom Workloads* for details).

All workload-generation related commands and code are in `firesim/sw/firesim-software`.

FireMarshal comes with a few basic workloads that you can build right out of the box (in `workloads/`). In this example, we will build and test the buildroot-based linux distribution (called *br-base*). We begin by building the workload:

```
./marshal build workloads/br-base.json
```

The first time you build a workload may take a long time (buildroot must download and cross-compile a large number of packages), but subsequent builds of the same base will use cached results. Once the command completes, you should see two new files in `images/`: `br-base-bin` and `br-base.img`. These are the boot-binary (linux + boot loader) and root filesystem (respectively). We can now launch this workload in qemu:

```
./marshal launch workloads/br-base.json
```

You should now see linux booting and be presented with a login prompt. Sign in as 'root' with password 'firesim'. From here you can manipulate files, run commands, and generally use the image as if it had booted on real hardware. Any changes you make here will be persistent between reboots. Once you are done exploring, simply shutdown the workload:

```
$ poweroff
```

It is typically not a good idea to modify the *-base workloads directly since many other workloads might inherit those changes. To make sure that we've cleaned out any changes, let's clean and rebuild the workload:

```
./marshal clean workloads/br-base.json
./marshal build workloads/br-base.json
```

Note that this build took significantly less time than the first; FireMarshal caches intermediate build steps whenever possible. The final step is to run this workload on the real firesim RTL with full timing accuracy. To do that we must first install the workload:

```
./marshal install workloads/br-base.json
```

This command creates a firesim workload file at `firesim/deploy/workloads/br-base.json`. You can now run this workload using the standard FireSim commands (e.g. *Running a Single Node Simulation*, just change the `workloadname` option to "br-base.json" from "linux-uniform.json").

Attention: While the FireMarshal `install` command is the recommended way to create firesim configurations, you can still hand-create firesim workloads if needed. For example, the linux-uniform workload described in *Running a Single Node Simulation* is a manually created workload that uses the br-base-bin and br-base.img files directly.

7.2 FireMarshal Commands

Attention: FireMarshal is still in alpha. You are encouraged to try it out and use it for new workloads. The old-style workload generation is still supported (see *Defining Custom Workloads* for details).

7.2.1 Core Options

The base `marshal` command provides a number of options that apply to most sub-commands. You can also run `marshal -h` for the most up-to-date documentation.

`--workdir`

By default, FireMarshal will search the same directory as the provided configuration file for `base` references and the workload source directory. This option instructs FireMarshal to look elsewhere for these references.

-i --initramfs

By default, FireMarshal assumes that your workload includes both a rootfs and a boot-binary. However, it may be necessary (e.g. when using spike) to build the rootfs into the boot-binary and load it into RAM during boot. This is only supported on linux-based workloads. This option instructs FireMarshal too use the *-initramfs boot-binary instead of the disk-based outputs.

-v --verbose

FireMarshal will redirect much of it's output to a log file in order to keep standard out clean. This option instructs FireMarshal to print much more output to standard out (in addition to logging it).

7.2.2 build

The build command is used to generate the rootfs's and boot-binaries from the workload configuration file. The output will be `images/NAME-JOBNAME-bin` and `images/NAME-JOBNAME.img` files for each job in the workload. If you passed the –initramfs option to FireMarshal, a `images/NAME-JOBNAME-bin-initramfs` file will also be created.

```
./marshal build [-B] [-I] config [config]
```

You may provide multiple config files to build at once.

-I -B

These options allow you to build only the image (rootfs) or boot-binary (respectively). This is occasionally useful if you have incomplete changes in the image or binary definitions but would still like to test the other.

7.2.3 launch

The launch command will run the workload in either Qemu (a high-performance functional simulator) or spike (the official RISC-V ISA simulator). Qemu will be used by default and is the best choice in most circumstances.

```
./marshal launch [-s] [-j [JOB]] config
```

-j --job

FireMarshal currently only supports launching one node at a time. By default, only the main workload will be run, you can specify jobs (using the job 'name') to run using the –job option.

-s --spike

In some cases, you may need to boot your workload in spike (typically due to a custom ISA extension or hardware model). In that case, you may use the -s option. Note that spike currently does not support network or block devices. You must pass the –initramfs option to FireMarshal when using spike.

7.2.4 clean

Deletes all outputs for the provided configuration (rootfs and bootbinary). Running the build command multiple times will re-run guest-init scripts and re-apply any files, but will not re-produce the base image. If you need to inherit changes from an updated base config, or generate a clean image (e.g. if the filesystem was corrupted), you must clean first.

7.2.5 test

The test command will build and run the workload, and compare its output against the `testing` specification provided in its configuration. See *Workload Specification* for details of the testing specification. If jobs are specified, all jobs will be run independently and their outputs will be included in the output directory.

`-s --spike`

Test using spike instead of qemu (requires the –initramfs option to the `marshal` command).

`-m testDir --manual testDir`

Do not build and launch the workload, simply compare it's `testing` specification against a pre-existing output. This allows you to check the output of firesim runs against a workload. It is also useful when developing a workload test.

7.2.6 install

Creates a firesim workload definition file in `firesim/deploy/workloads` with all appropriate links to the generated workload. This allows you to launch the workload in firesim using standard commands (see *Running FireSim Simulations*).

7.3 Workload Specification

Attention: FireMarshal is still in alpha. You are encouraged to try it out and use it for new workloads. The old-style workload generation is still supported (see *Defining Custom Workloads* for details).

Workloads are defined by a configuration file and corresponding workload source directory, both typically in the `firesim/sw/firesim-software/workloads/` directory. Most paths in the configuration file are assumed to be relative to the workload source directory.

7.3.1 Example Configuration File

FireMarshal supports many configuration options (detailed below), many of which are not commonly used. We will now walk through an example that uses most of the common options: `workloads/example-fed.json`. In this example, we produce a 2-node workload that runs two benchmarks: quicksort and spam-filtering. This will require installing a number of packages on Fedora, as well as cross-compiling some code. The configuration is as follows:

```
{
  "name" : "example-fed",
  "base" : "fedora-base.json",
  "overlay" : "overlay",
  "guest-init" : "guest-init.sh",
  "host-init" : "host-init.sh",
  "jobs" : [
    {
      "name" : "qsort",
      "run" : "runQsort.sh"
    },
    {
      "name" : "spamBench",
      "run" : "runSpam.sh"
    }
  ]
}
```

The `name` field is required and (by convention) should match the name of the configuration file. Next is the `base` (fedora-base.json). This option specifies an existing workload to base off of. FireMarshal will first build `fedora-base.json`, and use a copy of its rootfs for example-fed before applying the remaining options. Additionally, if fedora-base.json specifies any configuration options that we do not include, we will inherit those (e.g. we will use the `linux-config` option specified by fedora-base). Notice that we do not specify a workload source directory. FireMarshal will look in `workloads/example-fed/` for any sources specified in the remaining options.

Next come a few options that specify common setup options used by all jobs in this workload. The `overlay` option specifies a filesystem overlay to copy into our rootfs. In this case, it includes the source code for our benchmarks (see `workloads/example-fed/overlay`). Next is a `host-init` option, this is a script that should be run on the host before building. In our case, it cross-compiles the quicksort benchmark (cross-compilation is much faster than natively compiling).

```
#!/bin/bash

echo "Building qsort benchmark"
cd overlay/root/qsort

make
```

Next is `guest-init`, this script should run exactly once natively within our workload. For example-fed, this script installs a number of packages that are required by our benchmarks. Note that guest-init scripts are run during the build process; this can take a long time, especially with fedora. You will see linux boot messages and may even see a login prompt. There is no need to login or interact at all, the guest-init script will run in the background. Note that guest-init.sh ends with a `poweroff` command, all guest-init scripts should include this (leave it off to debug the build process).

```
#!/bin/bash

echo "Installing the real time tool (not the shell builtin)"
dnf install -y time

echo "Installing the spambayes python module for the spam benchmark"
pip install spambayes

poweroff
```

Finally, we specify the two jobs that will run on each simulated node. Job descriptions have the same format and options as normal workloads. However, notice that the job descriptions are much shorter than the basic descriptions.

Jobs implicitly inherit from the root configuration. In this case, both qsort and spamBench will have the overlay and host/guest-init scripts already set up for them. If needed, you could override these options with a different `base` option in the job description. In our case, we need only provide a custom `run` option to each workload. The run option specifies a script that should run natively in each job every time the job is launched. In our case, we run each benchmark, collecting some statistics along the way, and then shutdown. Finishing a run script with `poweroff` is a common pattern that allows workloads to run automatically (no need to log-in or interact at all).

```
#!/bin/bash
set -x

cd root/qsort
/usr/bin/time -f "%S,%M,%F" ./qsort 10000 2> ../run_result.csv
poweroff
```

We can now build and launch this workload:

```
./marshal build workloads/example-fed.json
./marshal launch -j qsort workloads/example-fed.json
./marshal launch -j spamBench workloads/example-fed.json
```

For more examples, see the `test/` directory that contains many workloads used for testing FireMarshal.

7.3.2 Bare-Metal Workloads

FireMarshal was primarily designed to support linux-based workloads. However, it provides basic support for bare-metal workloads. Take `test/bare.json` as an example:

```
{
  "name" : "bare",
  "base" : "bare",
  "host-init" : "build.sh",
  "bin" : "hello",
  "testing" : {
    "refDir" : "refOutput"
  }
}
```

This workload creates a simple "Hello World" bare-metal workload. This workload simply inherits from the "bare" distro in its `base` option. This tells FireMarshal to not attempt to build any linux binaries or rootfs's for this workload. It then includes a simple host-init script that simply calls the makefile to build the bare-metal boot-binary. Finally, it hard-codes a path to the generated boot-binary. Note that we can still use all the standard FireMarshal commands with bare-metal workloads. In this case, we provide a testing specification that simply compares the serial port output against the known good output of "Hello World!".

A complete discussion of generating bare-metal boot-binaries is out of scope for this documentation.

7.3.3 Configuration File Options

Below is a complete list of configuration options available to FireMarshal.

name

Name to use for the workload. Derived objects (rootfs/bootbin) will be named according to this option.

Non-heritable

base

Configuration file to inherit from. FireMarshal will look in the same directory as the workload config file for the base configuration (or the workdir if `--workdir` was passed to the marshal command). A copy of the rootfs from `base` will be used when building this workload. Additionally, most configuration options will be inherited if not explicitly provided (options that cannot be inherited will be marked as 'non-heritable' in this documentation).

In addition to normal configuration files, you may inherit from several hard-coded "distros" including: fedora, br (buildroot), and bare. This is not recommended for the linux-based distros because the fedora-base.json and br-base.json configurations include useful additions to get things like serial ports or the network to work. However, basing on the 'bare' distro is the recommended way to generate bare-metal workloads.

Non-heritable

spike

Path to binary for spike (riscv-isa-sim) to use when running this workload in spike. Useful for custom forks of spike to support custom instructions or hardware models. Defaults to the version of spike on your PATH (typically the one include with riscv-tools).

linux-src

Path to riscv-linux source directory to use when building the boot-binary for this workload. Defaults to the riscv-linux source submoduled at `firesim/sw/firesim-software/riscv-linux`.

linux-config

Linux configuration file to use when building linux. Take care when using a custom configuration, FireSim may require certain boot arguments and device drivers to work properly.

host-init

A script to run natively on your host (i.e., your manager instance where you invoked FireMarshal) from the workload source directory each time you explicitly build this workload.

Non-heritable

guest-init ^^^^^^^^^^^^^^^ A script to run natively on the guest (i.e., your workload running in qemu) exactly once while building. The guest init script will be run from the root directory with root privileges. This script should end with a call to `poweroff` to make the build process fully automated. Otherwise, the user will need to log in and shut down manually on each build.

post_run_hook

A script or command to run on the output of your run. At least the serial port output of each run is captured, along with any file outputs specified in the `outputs` option. The script will be called like so:

```
cd workload-dir
post_run_hook /path/to/output
```

The output directory will follow roughly the following format:

```
runOutput/name-DATETIME-RAND/
  name-job/
    uartlog
    OUTPUT_FILE1
    ...
    OUTPUT_FILEN
```

When running as part of the `test` command, there will be a folder for each job in the workload.

overlay

Filesystem overlay to apply to the workload rootfs. An overlay should match the rootfs directory structure, with the overlay directory corresponding to the root directory. This is especially useful for overriding system configuration files (e.g. /etc/fstab). The owner of all copied files will be changed to root in the workload rootfs after copying.

files

A list of files to copy into the rootfs. The file list has the following format:

```
[ ["src1", "dst1"], ["src2", "dst2"], ... ]
```

The source paths are relative to the workload source directory, the destination paths are absolute with respect to the workload rootfs (e.g. ["file1", "/root/"]). The ownership of each file will be changed to 'root' after copying.

outputs

A list of files to copy out of the workload rootfs after running. Each path should be absolute with respect to the workload rootfs. Files will be placed together in the output directory. You cannot specify the directory structure of the output.

run

A script to run automatically every time this workload runs. The script will run after all other initialization finishes, but does not require the user to log in (run scripts run concurrently with any user interaction). Run scripts typically end with a call to `poweroff` to make the workload fully automated, but this can be omitted if you would like to interact with the workload after its run script has finished.

Note: Unlike FireSim workloads, the FireMarshal launch command uses the same rootfs for each run (not a copy), so you should avoid using `poweroff -f` to prevent filesystem corruption.

Non-heritable

command

A command to run every time this workload runs. The command will be run from the root directory and will automatically call `poweroff` when complete (the user does not need to include this).

Non-heritable

workdir

Directory to use as the workload source directory. Defaults to a directory with the same name as the configuration file.

Non-heritable

launch

Enable/Disable launching of a job when running the 'test' command. This is occasionally needed for special 'dummy' workloads or other special-purpose jobs that only make sense when running on FireSim. Defaults to 'yes'.

jobs

A list of configurations describing individual jobs that make up this workload. This list is ordered (FireSim places these jobs in-order in simulation slots). Job descriptions have the same syntax and options as normal workloads. The one exception is that jobs implicitly inherit from the parent workload unless a `base` option is explicitly provided. The job name will be appended to the workload name when creating boot-binaries and rootfs's. For example, a workload called "foo" with two jobs named 'bar' and 'baz' would create 3 rootfs's: foo.img, foo-bar.img, and foo-baz.img.

Non-heritable: You cannot use jobs as a `base`, only base workloads.

bin

Explicit path to the boot-binary to use. This will override any generated binaries created during the build process. This is particularly useful for bare-metal workloads that generate their own raw boot code.

Non-heritable

img

Explicit path to the rootfs to use. This will override any generated rootfs created during the build process. This is mostly used for debugging.

Non-heritable

testing

Provide details of how to test this workload. The `test` command will ignore any workload that does not have a `testing` field. This option is a map with the following options (only `refDir` is required):

Non-heritable

refDir

Path to a directory containing reference outputs for this workload. Directory structures are compared directly (same folders, same file names). Regular files are compared exactly. Serial outputs (uartlog) need only match a subset of outputs; the entire reference uartlog contents must exist somewhere (contiguously) in the test uartlog.

buildTimeout

Maximum time (in seconds) that the workload should take to build. The test will fail if building takes longer than this. Defaults to infinite.

Note: workloads with many jobs and guest-init scripts, could take a very long time to build.

runTimeout

Maximum time (in seconds) that any particular job should take to run and exit. The test will fail if a job runs for longer than this before exiting. Defaults to infinite.

strip

Attempt to clean up the uartlog output before comparing against the reference. This will remove all lines not generated by a run script or command, as well as stripping out any extra characters that might be added by the run-system (e.g. the systemd timestamps on Fedora). This option is highly recommended on Fedora due to it's non-deterministic output.

CHAPTER 8

Targets

FireSim generates SoC models by transforming RTL emitted by a Chisel generator, such as the Rocket SoC generator. Subject to conditions outlined in *Restrictions on Target-RTL*, if it can be generated by Chisel, it can be simulated in FireSim.

8.1 Restrictions on Target-RTL

Current limitations in MIDAS place the following restrictions on the (FIR)RTL that can be transformed and thus used in FireSim:

1. The RTL must not contain multiple clock domains.
2. The RTL must not contain multi-cycle paths.
3. The RTL must not contain black boxes, with the exception of Rocket Chip's async reset register.
4. Asynchronous reset must only be implemented using Rocket Chip's black box async reset. These are replaced with synchronously reset registers using a FIRRTL transformation.

8.2 Generating Different Target-RTL

FireSim provides multiple *projects*, each for a different type of target. Each project has it's own chisel generator that invokes MIDAS, its own driver sources, and a makefrag that plugs into the Make-based build system that resides in `sim/`.

These projects are:

1. **firesim** (Default): rocket chip-based targets. These include targets with either BOOM or rocket pipelines, and should be your starting point if you're building an SoC with the Rocket Chip generator.
2. **midasexamples**: the MIDAS example designs, a set of simple chisel circuits like GCD, that demonstrate how to use MIDAS. These are useful test cases for bringing up new MIDAS features.
3. **fasedtests**: designs to do integration testing of FASED memory-system timing models.

Projects have the following directory structure:

```
sim/
├--Makefile # Top-level makefile for projects where FireSim is the top-level repo
├--Makefrag # Target-agnostic makefrag, with recipes to generate drivers and RTL␣
↪simulators
├--src/main/scala/{target-project}/
│                 └─Makefrag # Defines target-specific make variables and recipes.
├--src/main/cc/{target-project}/
│              ├─{driver-csrcs}.cc # The target's simulation driver, and sofware-model␣
↪sources
│              └─{driver-headers}.h
└--src/main/makefrag/{target-project}/
                     ├─Generator.scala # Contains the main class that generates␣
↪target RTL and calls MIDAS
                     └─{other-scala-sources}.scala
```

8.3 Rocket Chip Generator-based SoCs (firesim project)

This project can generate four different target-design classes (set with the make variable `DESIGN`, see `sim/src/main/scala/firesim/Targets.scala`). Like any other Rocket Chip-derived generator, a different parameterization is selected with a Chisel configuration (make variable `TARGET_CONFIG`, see `sim/src/main/scala/firesim/TargetConfigs.scala`). Since I/O and DRAM models are not generated by Rocket Chip but by MIDAS, their parameterization is captured in a second Chisel configuration (make variable `PLATFORM_CONFIG`, see `sim/src/main/scala/firesim/SimConfigs.scala`).

We give examples of generating different targets using these three variables in the sections that follow.

8.3.1 Rocket-based SoCs

Three design classes use Rocket scalar in-order pipelines.

Single core, Rocket pipeline (default)

```
make DESIGN=FireSim TARGET_CONFIG=FireSimRocketChipConfig
```

Single-core, Rocket pipeline, no network interface

```
make DESIGN=FireSimNoNIC TARGET_CONFIG=FireSimRocketChipConfig
```

Quad-core, Rocket pipeline

```
make DESIGN=FireSim TARGET_CONFIG=FireSimRocketChipQuadCoreConfig
```

8.3.2 BOOM-based SoCs

Two design classes use BOOM (Berkeley Out-of-Order Machine) superscalar out-of-order pipelines.

Single-core BOOM

```
make DESIGN=FireBoom TARGET_CONFIG=FireSimBoomConfig
```

Single-core BOOM, no network interface

```
make DESIGN=FireBoomNoNIC TARGET_CONFIG=FireSimBoomConfig
```

8.3.3 Generating A Different FASED Memory-Timing Model Instance

MIDAS's memory-timing model generator, FASED, can elaborate a space of different DRAM model instances: we give some typical ones here. These targets use the Makefile-defined defaults of `DESIGN=FireSim` `TARGET_CONFIG=FireSimRocketChipConfig`.

Quad-rank DDR3 first-come first-served memory access scheduler

```
make PLATFORM_CONFIG=FireSimDDR3FCFSConfig
```

Quad-rank DDR3 first-ready, first-come first-served memory access scheduler

```
make PLATFORM_CONFIG=FireSimDDR3FRFCFSConfig
```

As above, but with a 4 MiB (maximum simulatable capacity) last-level-cache model

```
make PLATFORM_CONFIG=FireSimDDR3FRFCFSLLC4MBConfig
```

8.4 Midas Examples (midasexamples project)

This project can generate nine different target-designs (set with the make variable `DESIGN`), each of these designs has their own chisel source file. They include:

1. EnableShiftRegister
2. GCD
3. Parity
4. PointerChaser
5. ResetShiftRegister
6. Risc
7. RiscSRAM
8. ShiftRegister
9. Stack

To generate MIDAS example targets, set the make variable `TARGET_PROJECT=midasexamples`. so that the right project makefrag is sourced.

8.4.1 Examples

Generate the GCD midas-example

```
make DESIGN=GCD TARGET_PROJECT=midasexamples
```

8.5 FASED Tests (fasedtests project)

This project generates target designs capable of driving considerably more bandwidth to an AXI4-memory slave than current FireSim-targets. Used used to do integration and stress testing of FASED instances.

8.5.1 Examples

Generate a synthesizable AXI4Fuzzer (based off of Rocket Chip's TL fuzzer), driving a DDR3 FR-FCFS-based FASED instance.

```
make TARGET_PROJECT=midasexamples DESIGN=AXI4Fuzzer PLATFORM_CONFIG=FRFCFSConfig
```

As above, but with a fuzzer configue to drive 10 million transactions through the instance.

```
make TARGET_PROJECT=midasexamples DESIGN=AXI4Fuzzer PLATFORM_CONFIG=NT10e7_
↪FRFCFSConfig
```

CHAPTER 9

Debugging

This section describes methods of debugging the target design and the simulation in FireSim.

9.1 Debugging & Testing with RTL Simulation

Simulation of a single FireSim node using software RTL simulators like Verilator, Synopsys VCS, or XSIM, is the most productive way to catch bugs before generating an AGFI.

FireSim provides flows to do RTL simulation at three different levels of the design/abstraction hierarchy. Ordered from least to most detailed, they are:

- **Target-Level**: This simulates just the RTL of the target-design (Rocket Chip). There are no host-level features being simulated. Supported simulators: VCS, Verilator.
- **MIDAS-Level**: This simulates the target-design after it's been transformed by MIDAS. The target- and host-clock are decoupled. FPGA-hosted simulation models are present. Abstract models for host-FPGA provided services, like DRAM, memory-mapped IO, and PCIS are used here. Supported simulators: VCS, Verilator.
- **FPGA-Level**: This is a complete simulation of the design that will passed to the FPGA tools, including clock-domain crossings, width adapters, PLLS, FPGA-periphery blocks like DRAM and PCI-E controllers. This leverages the simulation flow provided by AWS. Supported simulators: VCS, Vivado XSIM.

Generally, MIDAS-level simulations are only slightly slower than target-level ones. Moving to FPGA-Level is very expensive. This illustrated in the chart below.

Level	Waves	VCS	Verilator	Verilator -O1	Verilator -O2	XSIM
Target	Off	4.8 kHz	3.9 kHz	6.6 kHz	N/A	N/A
Target	On	0.8 kHz	3.0 kHz	5.1 kHz	N/A	N/A
MIDAS	Off	3.8 kHz	2.4 kHz	4.5 kHz	5.3 KHz	N/A
MIDAS	On	2.9 kHz	1.5 kHz	2.7 kHz	3.4 KHz	N/A
FPGA	On	2.3 Hz	N/A	N/A	N/A	0.56 Hz

Note that using more agressive optimization levels when compiling the Verilated-design dramatically lengths compile time:

Level	Waves	VCS	Verilator	Verilator -O1	Verilator -O2
MIDAS	Off	35s	48s	3m32s	4m35s
MIDAS	On	35s	49s	5m27s	6m33s

Notes: Default configurations of a single-core, Rocket-based instance running rv64ui-v-add. Frequencies are given in target-Hz. Presently, the default compiler flags passed to Verilator and VCS differ from level to level. Hence, these numbers are only intended to ball park simulation speeds, not provide a scientific comparison between simulators. VCS numbers collected on Millenium, Verilator numbers collected on a c4.4xlarge. (ML verilator version: 4.002, TL verilator version: 3.904)

9.1.1 Target-Level Simulation

This is described in *Debugging Verilog Simulation*, as part of the *Developing New Devices* tutorial.

9.1.2 MIDAS-Level Simulation

MIDAS-level simulations are run out of the `firesim/sim` directory. Currently, FireSim lacks support for MIDAS-level simulation of the NIC since DMA_PCIS is not yet supported. So here we'll be setting `DESIGN=FireSimNoNIC`. To compile a simulator, type:

```
[in firesim/sim]
make <verilator|vcs>
```

To compile a simulator with full-visibility waveforms, type:

```
make <verilator|vcs>-debug
```

As part of target-generation, Rocket Chip emits a make fragment with recipes for running suites of assembly tests. MIDAS puts this in `firesim/sim/generated-src/f1/<DESIGN>-<TARGET_CONFIG>-<PLATFORM_CONFIG>/firesim.d`. Make sure your `$RISCV` environment variable is set by sourcing `firesim/source-me*.sh` or `firesim/env.sh`, and type:

```
make run-<asm|bmark>-tests EMUL=<vcs|verilator>
```

To run only a single test, the make target is the full path to the output. Specifically:

```
make EMUL=<vcs|verilator> $PWD/output/f1/<DESIGN>-<TARGET_CONFIG>-<PLATFORM_CONFIG>/
↪<RISCV-TEST-NAME>.<vpd|out>
```

A `.vpd` target will use (and, if required, build) a simulator with waveform dumping enabled, whereas a `.out` target will use the faster waveform-less simulator.

Examples

Run all RISCV-tools assembly and benchmark tests on a verilated simulator.

```
[in firesim/sim]
make DESIGN=FireSimNoNIC
make DESIGN=FireSimNoNIC -j run-asm-tests
make DESIGN=FireSimNoNIC -j run-bmark-tests
```

Run all RISCV-tools assembly and benchmark tests on a verilated simulator with waveform dumping.

```
make DESIGN=FireSimNoNIC verilator-debug
make DESIGN=FireSimNoNIC -j run-asm-tests-debug
make DESIGN=FireSimNoNIC -j run-bmark-tests-debug
```

Run rv64ui-p-simple (a single assembly test) on a verilated simulator.

```
make DESIGN=FireSimNoNIC
make DESIGN=FireSimNoNIC $(pwd)/output/f1/FireSimNoNIC-FireSimRocketChipConfig-
↪FireSimConfig/rv64ui-p-simple.out
```

Run rv64ui-p-simple (a single assembly test) on a VCS simulator with waveform dumping.

```
make DESIGN=FireSimNoNIC vcs-debug
make DESIGN=FireSimNoNIC EMUL=vcs $(pwd)/output/f1/FireSimNoNIC-
↪FireSimRocketChipConfig-FireSimConfig/rv64ui-p-simple.vpd
```

9.1.3 FPGA-Level Simulation

Like MIDAS-level simulation, there is currently no support for DMA_PCIS, so we'll restrict ourselves to instances without a NIC by setting *DESIGN=FireSimNoNIC*. As with MIDAS-level simulations, FPGA-level simulations run out of `firesim/sim`.

Since FPGA-level simulation is up to 1000x slower than MIDAS-level simulation, FPGA-level simulation should only be used in two cases:

1. MIDAS-level simulation of the simulation is working, but running the simulator on the FPGA is not.
2. You've made changes to the AWS Shell/IP/cl_firesim.sv in aws-fpga and want to test them.

FPGA-level simulation consists of two components:

1. A FireSim-f1 driver that talks to a simulated DUT instead of the FPGA
2. The DUT, a simulator compiled with either XSIM or VCS, that receives commands from the aforementioned FireSim-f1 driver

Usage

To run a simulation you need to make both the DUT and driver targets by typing:

```
make xsim
make xsim-dut <VCS=1> & # Launch the DUT
make run-xsim SIM_BINARY=<PATH/TO/BINARY/FOR/TARGET/TO/RUN> # Launch the driver
```

When following this process, you should wait until `make xsim-dut` prints `opening driver to xsim` before running `make run-xsim` (getting these prints from `make xsim-dut` will take a while). Additionally, you will want to use `DESIGN=FireSimNoNIC`, since the XSim scripts included with `aws-fpga` do not support DMA PCIS.

Once both processes are running, you should see:

```
opening driver to xsim
opening xsim to driver
```

This indicates that the DUT and driver are successfully communicating. Eventually, the DUT will print a commit trace Rocket Chip. There will be a long pause (minutes, possibly an hour, depending on the size of the binary) after the first 100 instructions, as the program is being loaded into FPGA DRAM.

XSIM is used by default, and will work on EC2 instances with the FPGA developer AMI. If you have a license, setting `VCS=1` will use VCS to compile the DUT (4x faster than XSIM). Berkeley users running on the Millennium machines should be able to source `firesim/scripts/setup-vcsmx-env.sh` to setup their environment for VCS-based FPGA-level simulation.

The waveforms are dumped in the FPGA build directories(`firesim/platforms/f1/aws-fpga/hdk/cl/developer_designs/cl_<DESIGN>-<TARGET_CONFIG>-<PLATFORM_CONFIG>`).

For XSIM:

```
<BUILD_DIR>/verif/sim/vivado/test_firesim_c/tb.wdb
```

And for VCS:

```
<BUILD_DIR>/verif/sim/vcs/test_firesim_c/test_null.vpd
```

When finished, be sure to kill any lingering processes if you interrupted simulation prematurely.

9.1.4 Scala Tests

To make it easier to do RTL-simulation-based regression testing, the scala tests wrap calls to Makefiles, and run a limited set of tests on a set of selected designs, including all of the MIDAS examples, FireSimNoNIC and FireBoom-NoNIC.

The selected tests, target configurations, as well as the type of RTL simulator to compile can be modified by changing the scala tests that reside at `firesim/sim/src/test/scala/<target-project>/`.

To run all tests, with the sbt console open, do the familiar:

```
test
```

To run only tests on Rocket-Chip based targets:

```
testOnly firesim.firesim.*
```

To run only the MIDAS examples:

```
testOnly firesim.midasexamples.*
```

9.2 Debugging Using FPGA Integrated Logic Analyzers (ILA)

Sometimes it takes too long to simulate FireSim on RTL simulators, and in some occasions we would also like to debug the simulation infrastructure itself. For these purposes, we can use the Xilinx Integrated Logic Analyzer resources on the FPGA.

ILAs allows real time sampling of pre-selected signals during FPGA runtime, and provided and interface for setting trigger and viewing samples waveforms from the FPGA. For more information about ILAs, please refer to the Xilinx guide on the topic.

MIDAS, in its `targetutils` package, provides annotations for labeling signals directly in the Chisel source. These will be consumed by a downstream FIRRTL pass which wires out the annotated signals, and binds them to an appropriately sized ILA instance.

9.2.1 Annotating Signals

In order to annotate a signal, we must import the `midas.targetutils.FpgaDebug` annotator. FpgaDebug's apply method accepts a vararg of chisel3.Data. Invoke it as follows:

```
import midas.targetutils.FpgaDebug

class SomeModuleIO(implicit p: Parameters) extends SomeIO()(p){
   val out1 = Output(Bool())
   val in1 = Input(Bool())
   FpgaDebug(out1, in1)
}
```

You can annotate signals throughout FireSim, including in MIDAS and Rocket-Chip Chisel sources, with the only exception being the Chisel3 sources themselves (eg. in Chisel3.util.Queue).

Note: In case the module with the annotated signal is instantiated multiple times, all instatiations of the annotated signal will be wired to the ILA.

9.2.2 Using the ILA at Runtime

Prerequisite: Make sure that ports 3121 and 10201 are enabled in the firesim AWS security group.

In order to use the ILA, we must enable the GUI interface on our manager instance. This can be done by running the command:

```
/home/centos/src/scripts/setup_gui.sh
```

When the command will finish running, a temporary password will be printed out. This password will be used to access the GUI interface of the master instance. We will connect to the GUI interface of the manager instance using an RDP client. Use the public IP address of the manager instances in order to connect using the RDP client. The username is *centos*, and the password is the temporary password that was printed out at the end of the previous command. An additional login screen with the username Cloud-User and the same password may appear in some occasion. More information about the AWS GUI interface can be found in the `~/src/GUI_README` on the manager instance.

After access the GUI interface, open a terminal, and open `vivado`. Follow the instructions in the AWS-FPGA guide for connecting xilinx hardware manager on vivado (running on a remote machine) to the debug target .

where `<hostname or IP address>` is the internal IP of the simulation instance (not the manager instance. i.e. The IP starting with 192.168.X.X). The probes file can be found in the manager instance under the path `firesim/deploy/results-build/<build_identifier>/cl_firesim/build/checkpoints/<probes_file.ltx>`

Select the ILA with the description of *WRAPPER_INST/CL/CL_FIRESIM_DEBUG_WIRING_TRANSFORM*, and you may now use the ILA just as if it was on a local FPGA.

9.3 Debugging Using TracerV

FireSim can provide a cycle-by-cycle trace of the CPU's architectural state over the course of execution. This can be useful for profiling or debugging. The tracing functionality is provided by the TracerV widget.

9.3.1 Building a Design with TracerV

To use TracerV in your design, you must build and use one of the target configurations that contain the tracer. For instance, if you are using the FireSimRocketChipQuadCoreConfig, switch to the FireSimRocketCoreQuadCoreTracedConfig.

In TargetConfigs.scala:

```
class FireSimRocketChipQuadCoreTracedConfig extends Config(
  new WithTraceRocket ++ new FireSimRocketChipQuadCoreConfig)
```

In config_build_recipes.ini:

```
[firesim-quadcore-traced-nic-ddr3-llc4mb]
DESIGN=FireSim
TARGET_CONFIG=FireSimRocketChipQuadCoreTracedConfig
PLATFORM_CONFIG=FireSimDDR3FRFCFSLLC4MBConfig
instancetype=c4.4xlarge
deploytriplet=None
```

In config_build.ini:

```
[builds]
firesim-quadcore-traced-nic-ddr3-llc4mb
```

Then run "firesim buildafi" to build an FPGA image. Add the resulting AGFI as a new entry in config_hwdb.ini.

```
[firesim-quadcore-traced-nic-ddr3-llc4mb]
agfi=agfi-XXXXX
deploytripletoverride=None
customruntimeconfig=None
```

Finally, use this image as the defaulthwconfig in config_runtime.ini.

9.3.2 Enabling Tracing at Runtime

By default, FireSim will not collect data from the TracerV widget, even if it is included. To enable collection, add a new section to your config_runtime.ini.

```
[tracing]
enable=yes
```

Now when you run a workload, a trace output file will be placed in the sim_slot_X directory on the F1 instance under the name TRACEFILE. Tracing the entirety of a long-running job like a Linux-based workload can generate a pretty large image, and you may only care about the state within a certain timeframe. Therefore, FireSim allows you to specify a start cycle and end cycle for collecting data. By default, it starts at cycle 0 and ends at the last cycle of the simulation. To change this, add the following under the "tracing" section.

```
startcycle=XXXX
endcycle=YYYY
```

9.3.3 Interpreting the Trace Result

9.4 Assertion Synthesis

MIDAS can synthesize assertions present in FIRRTL (implemented as `stop` statements) that would otherwise be lost in the FPGA synthesis flow. Rocket and BOOM include hundreds of such assertions which, when synthesized, can provide great insight into why the target may be failing.

9.4.1 Enabling Assertion Synthesis

To enable assertion synthesis add the `WithSynthAsserts` Config to your PLATFORM_CONFIG in SimConfigs.scala. During compilation, MIDAS will print the number of assertions it's synthesized. In the target's `generated-src/` directory, you'll find a `*.asserts` file with the definitions of all synthesized assertions. If assertion synthesis has been enabled, the `synthesized_assertions_t` endpoint driver will be automatically instantiated the driver.

9.4.2 Runtime Behavior

If an assertion is caught during simulation, the driver will print the assertion cause, the path to module instance in which it fired, a source locator, and the cycle on which the assertion fired. Simulation will then terminate.

An example of an assertion caught in a dual-core instance of BOOM is given below:

```
id: 1190, module: IssueSlot_4, path: FireBoomNoNIC.tile_1.core.issue_units_0.slots_3]
Assertion failed
    at issue_slot.scala:214 assert (!slot_p1_poisoned)
    at cycle: 2142042185
```

9.4.3 Related Publications

Assertion synthesis was first presented in our FPL2018 paper, DESSERT.

9.5 Printf Synthesis

MIDAS can synthesize printfs present in FIRRTL (implemented as `printf` statements) that would otherwise be lost in the FPGA synthesis flow. Rocket and BOOM have printfs of their commit logs and other useful transaction streams.

```
C0:         409 [1] pc=[008000004c] W[r10=0000000000000000][1] R[r 0=0000000000000000]␣
↪R[r20=0000000000000003] inst=[f1402573] csrr    a0, mhartid
C0:         410 [0] pc=[008000004c] W[r 0=0000000000000000][0] R[r 0=0000000000000000]␣
↪R[r20=0000000000000003] inst=[f1402573] csrr    a0, mhartid
C0:         411 [0] pc=[008000004c] W[r 0=0000000000000000][0] R[r 0=0000000000000000]␣
↪R[r20=0000000000000003] inst=[f1402573] csrr    a0, mhartid
C0:         412 [1] pc=[0080000050] W[r 0=0000000000000000][0] R[r10=0000000000000000]␣
↪R[r 0=0000000000000000] inst=[00051063] bnez    a0, pc + 0
C0:         413 [1] pc=[0080000054] W[r 5=0000000080000054][1] R[r 0=0000000000000000]␣
↪R[r 0=0000000000000000] inst=[00000297] auipc   t0, 0x0
C0:         414 [1] pc=[0080000058] W[r 5=0000000080000064][1] R[r 5=0000000080000054]␣
↪R[r16=0000000000000003] inst=[01028293] addi    t0, t0, 16
C0:         415 [1] pc=[008000005c] W[r 0=0000000000010000][1] R[r 5=0000000080000064]␣
↪R[r 5=0000000080000064] inst=[30529073] csrw    mtvec, t0
```

(continues on next page)

(continued from previous page)

Synthesizing these printfs lets you capture the same logs on a running FireSim instance.

9.5.1 Enabling Printf Synthesis

To synthesize a printf, in your Chisel source you need to annotate the specific printfs you'd like to capture. Presently, due to a limitation in Chisel and FIRRTL's annotation system, you need to annotate the arguments to the printf, not the printf itself, like so:

```
printf(midas.targetutils.SynthesizePrintf("x%d p%d 0x%x\n", rf_waddr, rf_waddr, rf_
↪wdata))
```

Be judicious, as synthesizing many, frequently active printfs, will slow down your simulator.

Once your printfs have been annotated, to enable printf synthesis add the `WithPrintfSynthesis` Config to your PLATFORM_CONFIG in SimConfigs.scala. During compilation, MIDAS will print the number of printfs it's synthesized. In the target's generated header (`<DESIGN>-const.h`), you'll find metadata for each of the printfs MIDAS synthesized. This is passed as argument to the constructor of the `synthesized_prints_t` endpoint driver, which will be automatically instantiated in FireSim driver.

9.5.2 Runtime Arguments

+print-file Specifies the file into which the synthesized printf log should written.

+print-start Specifies the target-cycle at which the printf trace should be captured in the simulator. Since capturing high-bandwidth printf traces will slow down simulation, this allows the user to reach the region-of-interest at full simulation speed.

+print-end Specifies the target cycle at which to stop pulling the synthesized print trace from the simulator.

+print-binary By default, a captured printf trace will be written to file formatted as it would be emitted by a software RTL simulator. Setting this dumps the raw binary coming off the FPGA instead, improving simulation rate.

+print-no-cycle-prefix (Formatted output only) This removes the cycle prefix from each printf to save bandwidth in cases where the printf already includes a cycle field. In binary-output mode, since the target cycle is implicit in the token stream, this flag has no effect.

9.5.3 Related Publications

Printf synthesis was first presented in our FPL2018 paper, DESSERT.

CHAPTER 10

Tutorial: Developing New Devices

10.1 Getting Started

In this tutorial, we will show you how to design a new memory-mapped IO device, test it in simulation, and then build and run it on FireSim.

To start with, you will need to clone a copy of FireChip, the repository that aggregates all the target RTL for FireSim. FireSim already contains FireChip as a submodule under `target-design/firechip`, but it makes patches to the codebase so that it will work with the FPGA tools. Therefore, you will need to clone a clean copy if you want to use FireChip standalone.

Go to https://github.com/firesim/firechip and click the "Fork" button to fork the repository to your own account. Now clone the new repo to your local machine and initialize the submodules.

```
$ git clone https://github.com/yourusername/firechip.git
$ cd firechip
$ git submodule update --init
$ cd rocket-chip
$ git submodule update --init
$ cd ..
```

You will not need to install the riscv-tools again because you'll just be reusing the one in firesim. So make sure to go into firesim and source `sourceme-f1-full.sh` before you run the rest of the commands in this tutorial.

Now that everything is checked out, you can build the VCS simulator and run the regression tests to make sure everything is working.

```
$ cd vsim # or "cd verisim" for verilator
$ make # builds the DefaultExampleConfig
$ make run-regression-tests
```

If everything is set up correctly, you should see a bunch of `*.out` files in the `output/` directory. If you open these up, they should all say "Completed after XXXXX cycles" at the end and not have any error messages.

10.2 Memory-mapped Registers

In this tutorial, we will create a device which pulls in data from an externally-connected input stream and writes the data to memory. We'll create our device in the file `src/main/scala/example/InputStream.scala`. The first thing we need to do is set up some memory-mapped control registers that the CPU can use to communicate with the device. The easiest way to do this is by creating a `TLRegisterNode`, which provides a `regmap` method that can be used to generate the hardware for reading and writing to RTL registers.

```
class InputStream(
    address: BigInt,
    val beatBytes: Int = 8)
    (implicit p: Parameters) extends LazyModule {

  val device = new SimpleDevice("input-stream", Seq("example,input-stream"))
  val regnode = TLRegisterNode(
    address = Seq(AddressSet(address, 0x3f)),
    device = device,
    beatBytes = beatBytes)

  lazy val module = new InputStreamModuleImp(this)
}
```

We want to specify or override three arguments in the `TLRegisterNode` constructor. The first is the address of the device in the memory map. The address is specified as an `AddressSet` containing two values, a base address and a mask. The system bus will route all addresses that match the base address on the bits not set in the mask. In this case, we set the mask to `0x3f`, which sets the lower six bits. This means that a 64 byte region starting from the base address will be routed to this device.

The second argument to `TLRegisterNode` is a `SimpleDevice` object, which provides the name and compatibility of the device table entry that will be created for the peripheral. We won't show how this is used in this tutorial, but it will be important if you want to create a Linux kernel driver for the device.

The third argument to `TLRegisterNode` is `beatBytes`, which specifies the width of the TileLink interface. We will just pass this through from a class argument.

We want the device to be able to write a specified amount of bytes to a specified location in memory, so we'll provide `addr` and `len` registers. We will also want a `running` register for the CPU to signal that the device should start operation and a `complete` register for the device to signal to the CPU that it has completed.

```
class InputStreamModuleImp(outer: InputStream) extends LazyModuleImp(outer) {
    val addrBits = 64
    val w = 64
    val io = IO(new Bundle {
        // Not used yet
        val in = Flipped(Decoupled(UInt(w.W)))
    }
    val addr = Reg(UInt(addrBits.W))
    val len = Reg(UInt(addrBits.W))
    val running = RegInit(false.B)
    val complete = RegInit(false.B)

    outer.regnode.regmap(
        0x00 -> Seq(RegField(addrBits, addr)),
        0x08 -> Seq(RegField(addrBits, len)),
        0x10 -> Seq(RegField(1, running)),
        0x18 -> Seq(RegField(1, complete)))
}
```

The arguments to `regmap` should be a series of mappings from address offsets to sequences of `RegField` objects. The `RegField` constructor takes two arguments, the width of the register field and the RTL register itself.

10.3 DMA and Interrupts

10.3.1 TileLink Client Port

In order to move data from the external input stream to memory, we need to perform direct memory access (DMA). We can achieve this by giving the device a TLClientNode. Once we add it, the `LazyModule` will now look like this:

```
class InputStream(
    address: BigInt,
    val beatBytes: Int = 8,
    val maxInflight: Int = 4)
    (implicit p: Parameters) extends LazyModule {

  val device = new SimpleDevice("input-stream", Seq("example,input-stream"))
  val regnode = TLRegisterNode(
    address = Seq(AddressSet(address, 0x3f)),
    device = device,
    beatBytes = beatBytes)
  val dmanode = TLClientNode(Seq(TLClientPortParameters(
    Seq(TLClientParameters(
      name = "input-stream",
      sourceId = IdRange(0, maxInflight))))))

  lazy val module = new InputStreamModuleImp(this)
}
```

For our `TLClientNode`, we only need a single port, so we specify a single set of `TLClientPortParameters` and `TLClientParameters`. We override two arguments in the `TLClientParameters` constructor. The `name` is the name of the port and `sourceId` indicates the range of transaction IDs that can be used in memory requests. The lower bound is inclusive, and the upper bound is exclusive, so this device can use source IDs from 0 to `maxInflight - 1`.

10.3.2 TileLink Protocol and State Machine

In the module implementation, we can now implement a state machine that sends write requests to memory. We first call *outer.dmanode.out* to get a sequence of output port tuples. Since we only have one port, we can just pull out the first element of this sequence. For each port, we get a pair of objects. The first is the physical TileLink port, which we can connect to RTL. The second is a `TLEdge` object, which we can use to get extra metadata about the tilelink port (like the number of address and data bits).

```
class InputStreamModuleImp(outer: InputStream) extends LazyModuleImp(outer) {
  val (tl, edge) = outer.dmanode.out(0)
  val addrBits = edge.bundle.addressBits
  val w = edge.bundle.dataBits
  val beatBytes = (w / 8)

  val io = IO(new Bundle {
    val in = Flipped(Decoupled(UInt(w.W)))
  })
```

(continues on next page)

(continued from previous page)

```
  val addr = Reg(UInt(addrBits.W))
  val len = Reg(UInt(addrBits.W))
  val running = RegInit(false.B)
  val complete = RegInit(false.B)

  val s_idle :: s_issue :: s_wait :: Nil = Enum(3)
  val state = RegInit(s_idle)

  val nXacts = outer.maxInflight
  val xactBusy = RegInit(0.U(nXacts.W))
  val xactOnehot = PriorityEncoderOH(~xactBusy)
  val canIssue = (state === s_issue) && !xactBusy.andR

  io.in.ready := canIssue && tl.a.ready
  tl.a.valid  := canIssue && io.in.valid
  tl.a.bits   := edge.Put(
    fromSource = OHToUInt(xactOnehot),
    toAddress = addr,
    lgSize = log2Ceil(beatBytes).U,
    data = io.in.bits)._2
  tl.d.ready := running && xactBusy.orR

  xactBusy := (xactBusy |
    Mux(tl.a.fire(), xactOnehot, 0.U(nXacts.W))) &
    ~Mux(tl.d.fire(), UIntToOH(tl.d.bits.source), 0.U)

  when (state === s_idle && running) {
    assert(addr(log2Ceil(beatBytes)-1,0) === 0.U,
      s"InputStream base address not aligned to ${beatBytes} bytes")
    assert(len(log2Ceil(beatBytes)-1,0) === 0.U,
      s"InputStream length not aligned to ${beatBytes} bytes")
    state := s_issue
  }

  when (io.in.fire()) {
    addr := addr + beatBytes.U
    len := len - beatBytes.U
    when (len === beatBytes.U) { state := s_wait }
  }

  when (state === s_wait && !xactBusy.orR) {
    running := false.B
    complete := true.B
    state := s_idle
  }

  outer.regnode.regmap(
    0x00 -> Seq(RegField(addrBits, addr)),
    0x08 -> Seq(RegField(addrBits, len)),
    0x10 -> Seq(RegField(1, running)),
    0x18 -> Seq(RegField(1, complete)))
}
```

The state machine starts in the `s_idle` state. In this state, the CPU should set the `addr` and `len` registers and then set the `running` register to 1. The state machine then moves into the `s_issue` state, in which it forwards data from the `in` decoupled interface to memory through the TileLink *A* channel.

We construct the *A* channel requests using the `Put` method in the `TLEdge` object we extracted earlier. The `Put` method takes a unique source ID in `fromSource`, the address to write to in `toAddress`, the base-2 logarithm of the size in bytes in `lgSize`, and the data to be written in `data`.

The source field must observe some constraints. There can only be one transaction with each distinct source ID in flight at a given time. Once you send a request on the *A* channel with a specific source ID, you cannot send another until after you've received the response for it on the *D* channel.

Once all requests have been sent on the *A* channel, the state machine transitions to the `s_wait` state to wait for the remaining responses on the *D* channel. Once the responses have all returned, the state machine sets `running` to false and `completed` to true. The CPU can poll the `completed` register to check if the operation has finished.

10.3.3 Interrupts

For long-running operations, we would like to have the device notify the CPU through an interrupt. To add an interrupt to the device, we need to create an `IntSourceNode` in the lazy module.

```
val intnode = IntSourceNode(IntSourcePortSimple(resources = device.int))
```

Then, in the module implementation, we can connect the `complete` register to the interrupt line. That way, the CPU will get interrupted once the state machine completes. It can clear the interrupt by writing a 0 to the `complete` register.

```
val (interrupt, _) = outer.intnode.out(0)

interrupt(0) := complete
```

10.4 Connecting Devices to Bus

10.4.1 SoC Mixin Traits

Now that we have finished designing our peripheral device, we need to hook it up into the SoC. To do this, we first need to create two traits: one for the lazy module and one for the module implementation. The lazy module trait is the following.

```
trait HasPeripheryInputStream { this: BaseSubsystem =>
  private val portName = "input-stream"
  val streamWidth = pbus.beatBytes * 8
  val inputstream = LazyModule(new InputStream(0x10017000, pbus.beatBytes))
  pbus.toVariableWidthSlave(Some(portName)) { inputstream.regnode }
  sbus.fromPort(Some(portName))() := inputstream.dmanode
  ibus.fromSync := inputstream.intnode
}
```

We add the line `this: BaseSubsystem =>` to indicate that this trait will eventually be mixed into a class that extends `BaseSubsystem`, which contains the definition of the system bus `sbus`, peripheral bus `pbus`, and interrupt bus `ibus`. We instantiate the `InputStream` lazy module and give it the base address `0x10017000`. We connect the `pbus` into the register node, DMA node to the `sbus`, and interrupt node to the `ibus`.

The module implementation trait is as follows:

```
class FixedInputStream(data: Seq[BigInt], w: Int) extends Module {
    val io = IO(new Bundle {
```

(continues on next page)

(continued from previous page)

```
        val out = Decoupled(UInt(w.W))
    })

    val dataVec = VecInit(data.map(_.U(w.W)))
    val (dataIdx, dataDone) = Counter(io.out.fire(), data.length)
    val sending = RegInit(true.B)

    io.out.valid := sending
    io.out.bits := dataVec(dataIdx)

    when (dataDone) { sending := false.B }
}

trait HasPeripheryInputStreamModuleImp extends LazyModuleImp {
  val outer: HasPeripheryInputStream

  val stream_in = IO(Flipped(Decoupled(UInt(outer.streamWidth.W))))
  outer.inputstream.module.io.in <> stream_in

  def connectFixedInput(data: Seq[BigInt]) {
    val fixed = Module(new FixedInputStream(data, outer.streamWidth))
    stream_in <> fixed.io.out
  }
}
```

Since the interrupts and memory ports have already been connected in the lazy module trait, the module implementation trait only needs to create the external decoupled interface and connect that to the `InputStream` module implementation.

The `connectFixedInput` method will be used by the test harness to connect an input stream model that just sends a pre-specified stream of data.

10.4.2 Top-Level Design and Configuration

We can now mix these traits into the SoC design. Open up `src/main/scala/example/Top.scala` and add the following:

```
class ExampleTopWithInputStream(implicit p: Parameters) extends ExampleTop
    with HasPeripheryInputStream {
  override lazy val module = new ExampleTopWithInputStreamModule(this)
}

class ExampleTopWithInputStreamModule(outer: ExampleTopWithInputStream)
  extends ExampleTopModuleImp(outer)
  with HasPeripheryInputStreamModuleImp
```

We can then build a simulation using our new SoC by adding a configuration to `src/main/scala/example/Configs.scala`. This configuration will cause the test harness to instantiate an SoC with the `InputStream` device and then connect a fixed input stream model to it.

```
class WithFixedInputStream extends Config((site, here, up) => {
  case BuildTop => (clock: Clock, reset: Bool, p: Parameters) => {
    val top = Module(LazyModule(new ExampleTopWithInputStream()(p)).module)
    top.connectFixedInput(Seq(
      BigInt("1002abcd", 16),
```

(continues on next page)

(continued from previous page)

```
      BigInt("34510204", 16),
      BigInt("10329999", 16),
      BigInt("92101222", 16)))
    top
  }
})

class FixedInputStreamConfig extends Config(
  new WithFixedInputStream ++ new BaseExampleConfig)
```

We can now compile the simulation using VCS.

```
cd vsim
make CONFIG=FixedInputStreamConfig
```

This will produce a `simv-example-FixedInputStreamConfig` executable that can be used to run tests. We will discuss how to write and run those tests in the next section.

If you don't have VCS installed and want to use verilator instead, the commands are similar.

```
cd verisim
make CONFIG=FixedInputStreamConfig
```

This creates an executable called `simulator-example-FixedInputStreamConfig`.

10.5 Running Test Software

To test our input stream device, we want to write an application that uses the device to write data into memory, then reads the data and prints it out.

In project-template, test software is placed in the `tests/` directory, which includes a Makefile and library code for developing a baremetal program. We'll create a new file at `tests/input-stream.c` with the following code:

```
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>

#include "mmio.h"

#define N 4
#define INPUTSTREAM_BASE 0x10017000L
#define INPUTSTREAM_ADDR     (INPUTSTREAM_BASE + 0x00)
#define INPUTSTREAM_LEN      (INPUTSTREAM_BASE + 0x08)
#define INPUTSTREAM_RUNNING  (INPUTSTREAM_BASE + 0x10)
#define INPUTSTREAM_COMPLETE (INPUTSTREAM_BASE + 0x18)

uint64_t values[N];

int main(void)
{
        reg_write64(INPUTSTREAM_ADDR, (uint64_t) values);
        reg_write64(INPUTSTREAM_LEN, N * sizeof(uint64_t));
        asm volatile ("fence");
        reg_write64(INPUTSTREAM_RUNNING, 1);
```

(continues on next page)

(continued from previous page)

```
        while (reg_read64(INPUTSTREAM_COMPLETE) == 0) {}
        reg_write64(INPUTSTREAM_COMPLETE, 0);

        for (int i = 0; i < N; i++)
                printf("%016lx\n", values[i]);

        return 0;
}
```

This program statically allocates an array for the data to be written to. It then sets the `addr` and `len` registers, executes a `fence` instruction to make sure they are committed, and then sets the `running` register. It then continuously polls the `complete` register until it sees a non-zero value, at which point it knows the data has been written to memory and is safe to read back.

To compile this program, add "input-stream" to the `PROGRAMS` list in `tests/Makefile` and run `make` from the tests directory.

To run the program, return to the `vsim/` directory and run the simulator executable, passing the newly compiled `input-stream.riscv` executable as an argument.

```
$ cd vsim
$ ./simv-example-FixedInputStreamConfig ../tests/input-stream.riscv
```

The program should print out

```
000000001002abcd
0000000034510204
0000000010329999
0000000092101222
```

For verilator, the command is the following:

```
$ cd verisim
$ ./simulator-example-FixedInputStreamConfig ../tests/input-stream.riscv
```

10.5.1 Debugging Verilog Simulation

If there is a bug in your hardware, one way to diagnose the issue is to generate a waveform from the simulation so that you can introspect into the design and see what values signals take over time.

In VCS, you can accomplish this with the `+vcdplusfile` flag, which will generate a VPD file that can be viewed in DVE. To use this flag, you will need to build the debug version of the simulator executable.

```
$ cd vsim
$ make CONFIG=FixedInputStreamConfig debug
$ ./simv-example-FixedInputStreamConfig-debug +max-cycles=50000 +vcdplusfile=input-
↪stream.vpd ../tests/input-stream.riscv
$ dve -full64 -vpd input-stream.vpd
```

The `+max-cycles` flag is used to set a timeout for the simulation. This is useful in the case the program hangs without completing.

If you are using verilator, you can generate a VCD file that can be viewed in an open source waveform viewer like GTKwave.

```
$ cd verisim
$ make CONFIG=FixedInputStreamConfig debug
$ ./simulator-example-FixedInputStreamConfig-debug +max-cycles=50000 -vinput-stream.
↪vcd ../tests/input-stream.riscv
$ gtkwave -o input-stream.vcd
```

10.6 Creating Simulation Model

So far, we've been using a fixed input stream model to test our device. But, ideally, we'd like an input stream that is defined by a software model and configurable at runtime. We'd like to put the input data in a file and pass it in as a command-line argument. We can't do that in Chisel. We'll have to create the model in Verilog and call out to C++ using the Verilog DPI-C API.

First, how do we include Verilog code in a Chisel codebase? We can do this using the Chisel BlackBox class. BlackBox modules can be used like regular Chisel modules and have defined IO ports, but the internal implementation is left to Verilog.

```
class SimInputStream(w: Int) extends BlackBox(Map("DATA_BITS" -> IntParam(w))) {
  val io = IO(new Bundle {
    val clock = Input(Clock())
    val reset = Input(Bool())
    val out = Decoupled(UInt(w.W))
  })
}
```

One key difference in the IO bundle definition is that the implicit `clock` and `reset` signals must be explicitly defined in a BlackBox. The BlackBox class also takes a map that defines parameters that will be passed to the verilog implementation. To connect the BlackBox in the test harness, we should create a `connectSimInput` method in the `HasPeripheryInputStreamModuleImp` trait.

```
def connectSimInput(clock: Clock, reset: Bool) {
  val sim = Module(new SimInputStream(outer.streamWidth))
  sim.io.clock := clock
  sim.io.reset := reset
  stream_in <> sim.io.out
}
```

We then add a new configuration class in `src/main/scala/example/Configs.scala` that calls the `connectSimInput` method.

```
class WithSimInputStream extends Config((site, here, up) => {
  case BuildTop => (clock: Clock, reset: Bool, p: Parameters) => {
    val top = Module(LazyModule(new ExampleTopWithInputStream()(p)).module)
    top.connectSimInput(clock, reset)
    top
  }
})

class SimInputStreamConfig extends Config(
  new WithSimInputStream ++ new BaseExampleConfig)
```

Now we need to create the verilog implementation of the `SimInputStream` module. Make a new directory `src/main/resources` and add `vsrc` and `csrc` subdirectories under it.

```
$ mkdir -p src/main/resources/{vsrc,csrc}
```

In the `vsrc` directory, create a file called `SimInputStream.v` and add the following code.

```
import "DPI-C" function void input_stream_init
(
    input string filename,
    input int    data_bits
);

import "DPI-C" function void input_stream_tick
(
    output bit     out_valid,
    input  bit     out_ready,
    output longint out_bits
);

module SimInputStream #(DATA_BITS=64) (
    input                   clock,
    input                   reset,
    output                  out_valid,
    input                   out_ready,
    output [DATA_BITS-1:0] out_bits
);

    bit __out_valid;
    longint __out_bits;
    string filename;
    int data_bits;

    reg                  __out_valid_reg;
    reg [DATA_BITS-1:0] __out_bits_reg;

    initial begin
        data_bits = DATA_BITS;
        if ($value$plusargs("instream=%s", filename)) begin
            input_stream_init(filename, data_bits);
        end
    end

    always @(posedge clock) begin
        if (reset) begin
            __out_valid = 0;
            __out_bits = 0;

            __out_valid_reg <= 0;
            __out_bits_reg <= 0;
        end else begin
            input_stream_tick(
                __out_valid,
                out_ready,
                __out_bits);
            __out_valid_reg <= __out_valid;
            __out_bits_reg  <= __out_bits;
        end
    end
```

(continues on next page)

(continued from previous page)

```
    assign out_valid = __out_valid_reg;
    assign out_bits  = __out_bits_reg;

endmodule
```

The verilog defines its inputs and outputs to match the definition in the Chisel BlackBox. But most of the implementation is left to C++ through the DPI functions `input_stream_init` and `input_stream_tick`. We define these functions in a `SimInputStream.cc` file in the `csrc` directory.

```
#include <stdio.h>
#include <stdint.h>
#include <stdlib.h>

class InputStream {
  public:
    InputStream(const char *filename, int nbytes);
    ~InputStream(void);

    bool out_valid() { return !complete; }
    uint64_t out_bits() { return data; }
    void tick(bool out_ready);

  private:
    void read_next(void);
    bool complete;
    FILE *file;
    int nbytes;
    uint64_t data;
};

InputStream::InputStream(const char *filename, int nbytes)
{
    this->nbytes = nbytes;
    this->file = fopen(filename, "r");
    if (this->file == NULL) {
        fprintf(stderr, "Could not open %s\n", filename);
        abort();
    }

    read_next();
}

InputStream::~InputStream(void)
{
    fclose(this->file);
}

void InputStream::read_next(void)
{
    int res;

    this->data = 0;

    res = fread(&this->data, this->nbytes, 1, this->file);
    if (res < 0) {
        perror("fread");
```

(continues on next page)

(continued from previous page)

```
        abort();
    }

    this->complete = (res == 0);
}

void InputStream::tick(bool out_ready)
{
    int res;

    if (out_valid() && out_ready)
        read_next();
}

InputStream *stream = NULL;

extern "C" void input_stream_init(const char *filename, int data_bits)
{
    stream = new InputStream(filename, data_bits/8);
}

extern "C" void input_stream_tick(
        unsigned char *out_valid,
        unsigned char out_ready,
        long long     *out_bits)
{
    stream->tick(out_ready);
    *out_valid = stream->out_valid();
    *out_bits  = stream->out_bits();
}
```

In the C++ file, we implement an `InputStream` class that takes a file name as its argument. It opens the file and reads `nbytes` from it for every ready-valid handshake. The `input_stream_init` function constructs an `InputStream` class and assigns it to a global pointer. The `input_stream_tick` function updates the state by calling the `tick` method, passing in the inputs from verilog. It then assigns values to the verilog outputs.

You can now build this new configuration in VCS.

```
$ cd vsim
$ make CONFIG=SimInputStreamConfig
```

Now create a file that can be used as the input stream data. Just getting random bytes from `/dev/urandom` would work. Pass this to your simulation through the `+instream=` flag, and you should see the data get printed out in the `input-stream.riscv` test.

```
$ dd if=/dev/urandom of=instream.img bs=32 count=1
$ hexdump instream.img
0000000 189b f12a 1cc1 9eb5 b65d bbef 96b6 4949
0000010 f8c8 636c 76fe 15f3 0665 0ef9 8c5d 3011
0000020
$ ./simv-example-SimInputStreamConfig +instream=instream.img ../tests/input-stream.
↪riscv
9eb51cc1f12a189b
494996b6bbefb65d
15f376fe636cf8c8
30118c5d0ef90665
```

CHAPTER 11

Supernode - Multiple Simulated SoCs Per FPGA

Supernode allows users to run multiple simulated SoCs per-FPGA in order to improve FPGA resource utilization and reduce cost. For example, in the case of using FireSim to simulate a datacenter scale system, supernode mode allows realistic rack topology simulation (32 simulated nodes) using a single `f1.16xlarge` instance (8 FPGAs).

Below, we outline the build and runtime configuration changes needed to utilize supernode designs. Supernode is currently only enabled for RocketChip designs with NICs. More details about supernode can be found in the FireSim ISCA 2018 Paper.

11.1 Introduction

By default, supernode packs 4 identical designs into a single FPGA, and utilizes all 4 DDR channels available on each FPGA on AWS F1 instances. It currently does so by generating a wrapper top level target which encapsualtes the four simulated target nodes. The packed nodes are treated as 4 separate nodes, are assigned their own individual MAC addresses, and can perform any action a single node could: run different programs, interact with each other over the network, utilize different block device images, etc. In the networked case, 4 separate network links are presented to the switch-side.

11.2 Building Supernode Designs

Here, we outline some of the changes between supernode and regular simulations that are required to build supernode designs.

The Supernode target configuration wrapper can be found in `firesim/sim/src/main/scala/firesim/TargetConfigs.scala`. An example wrapper configuration is:

```
class SupernodeFireSimRocketChipConfig extends Config(new WithNumNodes(4)
++ new FireSimRocketChipConfig)
```

In this example, `SupernodeFireSimRocketChipConfig` is the wrapper, while `FireSimRocketChipConfig` is the target node configuration. To simulate a different target configuration,

we will generate a new supernode wrapper, with the new target configuration. For example, to simulate 4 quad-core nodes on one FPGA, you can use:

```
class SupernodeFireSimRocketChipQuadCoreConfig extends Config(new
WithNumNodes(4) ++ new FireSimRocketChipQuadCoreConfig)
```

Next, when defining the build recipe, we must remmber to use the supernode configuration: The `DESIGN` parameter should always be set to `FireSimSupernode`, while the `TARGET_CONFIG` parameter should be set to the wrapper configuration that was defined in `firesim/sim/src/main/scala/firesim/TargetConfigs.scala`. The `PLATFORM_CONFIG` can be selected the same as in regular FireSim configurations. For example:

```
DESIGN=FireSimSupernode
TARGET_CONFIG=SupernodeFireSimRocketChipQuadCoreConfig
PLATFORM_CONFIG=FireSimDDR3FRFCFSLLC4MBConfig90MHz
instancetype=c4.4xlarge
deploytriplet=None
```

We currently provide a single pre-built AGFI for supernode of 4 quad-core RocketChips with DDR3 memory models. You can build your own AGFI, using the supplied samples in `config_build_recipes.ini`. Importantly, in order to meet FPGA timing contraints, Supernode target may require lower host clock frequencies. host clock frequencies can be configured as parts of the PLATFORM_CONFIG in `config_build_recipes.ini`.

11.3 Running Supernode Simulations

Running FireSim in supernode mode follows the same process as in "regular" mode. Currently, the only difference is that the main simulation screen remains with the name `fsim0`, while the three other simulation screens can be accessed by attaching `screen` to `uartpty1`, `uartpty2`, `uartpty3` respectively. All simulation screens will generate uart logs (`uartlog1`, `uartlog2`, `uartlog3`). Notice that you must use `sudo` in order to attach to the uartpty or view the uart logs. The additional uart logs will not be copied back to the manager instance by default (as in a "regular" FireSim simulation). It is neccessary to specify the copying of the additional uartlogs (uartlog1, uartlog2, uartlog3) in the workload definition.

Supernode topologies utilize a `FireSimSuperNodeServerNode` class in order to represent one of the 4 simulated target nodes which also represents a single FPGA mapping, while using a `FireSimDummyServerNode` class which represent the other three simulated target nodes which do not represent an FPGA mapping. In supernode mode, topologies should always add nodes in pairs of 4, as one `FireSimSuperNodeServerNode` and three `FireSimDummyServerNode` s.

Various example Supernode topologies are provided, ranging from 4 simulated target nodes to 1024 simulated target nodes.

Below are a couple of useful examples as templates for writing custom Supernode topologies.

A sample Supernode topology of 4 simulated target nodes which can fit on a single `f1.2xlarge` is:

```
def supernode_example_4config(self):
    self.roots = [FireSimSwitchNode()]
    servers = [FireSimSuperNodeServerNode()] + [FireSimDummyServerNode() for x in␣
↪range(3)]
    self.roots[0].add_downlinks(servers)
```

A sample Supernode topology of 32 simulated target nodes which can fit on a single `f1.16xlarge` is:

```
def supernode_example_32config(self):
    self.roots = [FireSimSwitchNode()]
```

(continues on next page)

(continued from previous page)

```
    servers = UserTopologies.supernode_flatten([[FireSimSuperNodeServerNode(),␣
↪FireSimDummyServerNode(), FireSimDummyServerNode(), FireSimDummyServerNode()] for y␣
↪in range(8)])
    self.roots[0].add_downlinks(servers)
```

Supernode `config_runtime.ini` requires selecting a supernode agfi in conjunction with a defined supernode topology.

11.4 Work in Progress!

We are currently working on restructuring supernode to support a wider-variety of use cases (including non-networked cases, and increased packing of nodes). More documentation will follow. Not all FireSim features are currently available on Supernode. As a rule-of-thumb, target-related features have a higher likelihood of being supported "out-of-the-box", while features which involve external interfaces (such as TracerV) has a lesser likelihood of being supported "out-of-the-box"

CHAPTER 12

Miscellaneous Tips

12.1 Add the `fsimcluster` column to your AWS management console

Once you've deployed a simulation once with the manager, the AWS management console will allow you to add a custom column that will allow you to see at-a-glance which FireSim run farm an instance belongs to.

To do so, click the gear in the top right of the AWS management console. From there, you should see a checkbox for `fsimcluster`. Enable it to see the column.

12.2 FPGA Dev AMI Remote Desktop Setup

To Remote Desktop into your manager instance, you must do the following:

```
curl https://s3.amazonaws.com/aws-fpga-developer-ami/1.5.0/Scripts/setup_gui.sh -o /
↪home/centos/src/scripts/setup_gui.sh
sudo sed -i 's/enabled=0/enabled=1/g' /etc/yum.repos.d/CentOS-CR.repo
/home/centos/src/scripts/setup_gui.sh
# keep manager paramiko compatibility
sudo pip2 uninstall gssapi
```

See

https://forums.aws.amazon.com/message.jspa?messageID=848073#848073

and

https://forums.aws.amazon.com/ann.jspa?annID=5710

12.3 Experimental Support for SSHing into simulated nodes and accessing the internet from within simulations

This is assuming that you are simulating a 1-node networked cluster. These instructions will let you both ssh into the simulated node and access the outside internet from within the simulated node:

1. Set your config files to simulate a 1-node networked cluster (`example_1config`)
2. Run `firesim launchrunfarm && firesim infrasetup` and wait for them to complete
3. cd to `firesim/target-design/switch/`
4. Go into the newest directory that is prefixed with `switch0-`
5. Edit the `switchconfig.h` file so that it looks like this:

```
// THIS FILE IS MACHINE GENERATED. SEE deploy/buildtools/switchmodelconfig.py

#ifdef NUMCLIENTSCONFIG
#define NUMPORTS 2
#endif
#ifdef PORTSETUPCONFIG
ports[0] = new ShmemPort(0);
ports[1] = new SSHPort(1);
#endif

#ifdef MACPORTSCONFIG
uint16_t mac2port[3]  {1, 2, 0};
#endif
```

6. Run `make` then `cp switch switch0`
7. Run `scp switch0 YOUR_RUN_FARM_INSTANCE_IP:switch_slot_0/switch0`
8. On the RUN FARM INSTANCE, run:

```
sudo ip tuntap add mode tap dev tap0 user $USER
sudo ip link set tap0 up
sudo ip addr add 172.16.0.1/16 dev tap0
sudo ifconfig tap0 hw ether 8e:6b:35:04:00:00
sudo sysctl -w net.ipv6.conf.tap0.disable_ipv6=1
```

9. Run `firesim runworkload`. Confirm that the node has booted to the login prompt in the fsim0 screen.
10. To ssh into the simulated machine, you will need to first ssh onto the Run Farm instance, then ssh into the IP address of the simulated node (172.16.0.2), username root, password firesim. You should also prefix with TERM=linux to get backspace to work correctly: So:

```
ssh YOUR_RUN_FARM_INSTANCE_IP
# from within the run farm instance:
TERM=linux ssh root@172.16.0.2
```

11. To also be able to access the internet from within the simulation, run the following on the RUN FARM INSTANCE:

```
sudo sysctl -w net.ipv4.ip_forward=1
export EXT_IF_TO_USE=$(ifconfig -a | sed 's/[ \t].*//;/^\(lo:\|\)$/d' | sed 's/[ \t].
↪*//;/^\(tap0:\|\)$/d' | sed 's/://g')
sudo iptables -A FORWARD -i $EXT_IF_TO_USE -o tap0 -m state --state RELATED,
↪ESTABLISHED -j ACCEPT
```

(continues on next page)

(continued from previous page)

```
sudo iptables -A FORWARD -i tap0 -o $EXT_IF_TO_USE -j ACCEPT
sudo iptables -t nat -A POSTROUTING -o $EXT_IF_TO_USE -j MASQUERADE
```

12. Then run the following in the simulation:

```
route add default gw 172.16.0.1 eth0
echo "nameserver 8.8.8.8" >> /etc/resolv.conf
echo "nameserver 8.8.4.4" >> /etc/resolv.conf
```

At this point, you will be able to access the outside internet, e.g. `ping google.com` or `wget google.com`.

12.4 Navigating the FireSim Codebase

This is a large codebase with tons of dependencies, so navigating it can be difficult. By default, a `tags` file is generated when you run `./build-setup.sh` which aids in jumping around the codebase. This file is generated by Exuberant Ctags and many editors support using this file to jump around the codebase. You can also regenerate the `tags` file if you make code changes by running `./gen-tags.sh` in your FireSim repo.

For example, to use these tags to jump around the codebase in `vim`, add the following to your `.vimrc`:

```
set tags=tags;/
```

Then, you can move the cursor over something you want to jump to and hit `ctrl-]` to jump to the definition and `ctrl-t` to jump back out. E.g. in top-level configurations in FireSim, you can jump all the way down through the Rocket Chip codebase and even down to Chisel.

CHAPTER 13

FireSim Asked Questions

13.1 I just bumped the FireSim repository to a newer commit and simulations aren't running. What is going on?

Anytime there is an AGFI bump, FireSim simulations will break/hang due to outdated AFGI. To get the new default AGFI's you must run the manager initialization again by doing the following:

```
cd firesim
source sourceme-f1-manager.sh
firesim managerinit
```

13.2 Is there a good way to keep track of what AGFI corresponds to what FireSim commit?

When building an AGFI during `firesim buildafi`, FireSim keeps track of what FireSim repository commit was used to build the AGFI. To view a list of AGFI's that you have built and what you have access to, you can run the following command:

```
cd firesim
source sourceme-f1-manager.sh
aws ec2 describe-fpga-images --fpga-image-ids # List all AGFI images
```

You can also view a specific AGFI image by giving the AGFI ID (found in `deploy/config_hwdb.ini`) through the following command:

```
cd firesim
source sourceme-f1-manager.sh
aws ec2 describe-fpga-images --filter Name=fpga-image-global-id,Values=agfi-<Your ID␣
↪Here> # List particular AGFI image
```

After querying an AGFI, you can find the commit hash of the FireSim repository used to build the AGFI within the "Description" field.

For more information, you can reference the AWS documentation at https://docs.aws.amazon.com/cli/latest/reference/ec2/describe-fpga-images.html.

CHAPTER 14

Indices and tables

- genindex
- modindex
- search