FireSim Documentation

Release 1.13.2

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

Feb 28, 2022

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

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
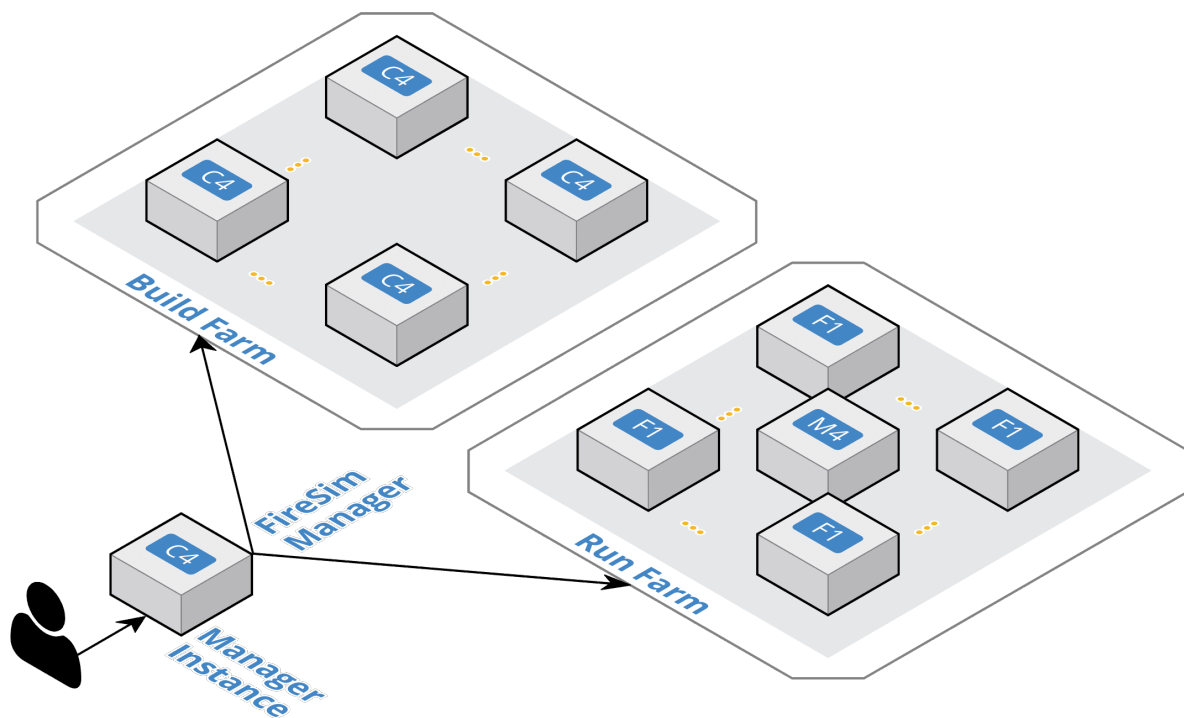
Fig. 1: FireSim Infrastructure Diagram

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).

Golden Gate (MIDAS II) The FIRRTL compiler used by FireSim to convert target RTL into a decoupled simulator. Formerly named MIDAS.

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.

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. You can learn more about EC2 instance limits here: https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/ec2-on-demand-instances.html#ec2-on-demand-instances-limits

To request a limit increase, follow these steps:

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

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

```
Limit Type:                EC2 Instances
Region:                    US East (Northern Virginia)
Primary Instance Type:     All F instances
Limit:                     Instance Limit
New limit value:           64
```

This limit of 64 vCPUs for F instances 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 `c5.4xlarge` 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 `c5.4xlarge` 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. Within the prompt, 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`. You will need to generate an AWS access key in the "Security Credentials" menu of your AWS settings (as instructed in https://docs.aws.amazon.com/general/latest/gr/aws-sec-cred-types.html#access-keys-and-secret-access-keys).

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

```
sudo yum install -y python36-pip
sudo pip3 install --upgrade pip
sudo python3 -m pip install boto3
sudo python3 -m pip install --upgrade awscli
```

(continues on next page)

```
wget https://raw.githubusercontent.com/firesim/firesim/master/deploy/awstools/aws_
↪setup.py
./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 `c5.4xlarge` and `f1` instances later, the Manager Instance can be a relatively cheap instance. In this guide, however, we will use a `c5.4xlarge`, running the AWS FPGA Developer AMI. (Be sure to subscribe to the AMI if you have not done so. See *Subscribe to the AWS FPGA Developer AMI*. Note that it might take a few minutes after subscribing to the AMI to be able to launch instances using it.)

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 Developer AMI - 1.11.1-40257ab5-6688-4c95-97d1-e251a40fd1fc` and select the AMI that appears (there should be only one). **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 `c5.4xlarge`.

4. On the "Configure Instance Details" page:

   1. 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.

   2. 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.

   3. Also on this page, expand "Advanced Details" and in the resulting text box, paste the following:

```bash
#!/bin/bash

set -ex
set -o pipefail

echo "machine launch script started" > /home/centos/machine-launchstatus
sudo chgrp centos /home/centos/machine-launchstatus
sudo chown centos /home/centos/machine-launchstatus

{
sudo yum install -y ca-certificates
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://www.scala-sbt.org/sbt-rpm.repo | sudo tee /etc/yum.repos.d/scala-
↪sbt-rpm.repo
sudo yum install -y sbt texinfo gengetopt libffi-devel
sudo yum install -y expat-devel libusb1-devel ncurses-devel cmake
↪"perl(ExtUtils::MakeMaker)"
# deps for poky
sudo yum install -y python36 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 python36-pip python36-devel rsync
# Install GNU make 4.x (needed to cross-compile glibc 2.28+)
sudo yum install -y centos-release-scl
sudo yum install -y devtoolset-8-make

# install DTC
sudo yum -y install dtc

# get a proper version of git
sudo yum -y remove git
sudo yum -y install epel-release
sudo yum -y install https://repo.ius.io/ius-release-el7.rpm
sudo yum -y install git224

# install verilator
git clone http://git.veripool.org/git/verilator
cd verilator/
git checkout v4.034
autoconf && ./configure && make -j4 && sudo make install
cd ..

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

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

# used for CI
sudo yum -y install expect

# upgrade pip
```

(continues on next page)

```
sudo pip3 install --upgrade pip==21.3.1
# install requirements
sudo python3 -m pip install fab-classic==1.19.1
sudo python3 -m pip install boto3==1.20.21
sudo python3 -m pip install colorama==0.4.3
sudo python3 -m pip install argcomplete==1.12.3
sudo python3 -m pip install graphviz==0.19
# for some of our workload plotting scripts
sudo python3 -m pip install pyparsing==3.0.6
sudo python3 -m pip install numpy==1.19.5
sudo python3 -m pip install kiwisolver==1.3.1
sudo python3 -m pip install matplotlib==3.3.4
sudo python3 -m pip install pandas==1.1.5
sudo python3 -m pip install awscli==1.22.21
sudo python3 -m pip install pytest==6.2.5
sudo python3 -m pip install moto==2.2.17
sudo python3 -m pip install sure==2.0.0
# needed for the awstools cmdline parsing
sudo python3 -m pip install pyyaml==5.4.1


# setup argcomplete
activate-global-python-argcomplete


} 2>&1 | tee /home/centos/machine-launchstatus.log


# 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 65GB can quickly become too small as you accumulate large Vivado reports/outputs, large waveforms, XSim outputs, and large root filesystems for simulations. You should 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
# checkout latest official firesim release
# note: this may not be the latest release if the documentation version !=
↪"stable"
git checkout 1.13.2
./build-setup.sh fast
```

The `build-setup.sh` script will validate that you are on a tagged branch, otherwise it will prompt for confirmation. 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.

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
./init-submodules.sh
./marshal -v build br-base.json
```

This process will take about 10 to 15 minutes on a c5.4xlarge instance. Once this is completed, you'll have the following files:

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

- `firesim/sw/firesim-software/images/br-base.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
always_expand_runfarm=yes

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

launch_instances_timeout_minutes=60

runinstancemarket=ondemand
spotinterruptionbehavior=terminate
spotmaxprice=ondemand

[targetconfig]
#Set topology=no_net_config to run without a network simulation
topology=example_8config
no_net_num_nodes=2
linklatency=6405
switchinglatency=10
netbandwidth=200
profileinterval=-1

# This references a section from config_build_recipes.ini
# In homogeneous configurations, use this to set the hardware config deployed
# for all simulators
defaulthwconfig=firesim-rocket-quadcore-nic-l2-llc4mb-ddr3

[tracing]
enable=no

# Trace output formats. Only enabled if "enable" is set to "yes" above
# 0 = human readable; 1 = binary (compressed raw data); 2 = flamegraph (stack
# unwinding -> Flame Graph)
output_format=0

# Trigger selector.
# 0 = no trigger; 1 = cycle count trigger; 2 = program counter trigger; 3 =
```

(continues on next page)

```
# instruction trigger
selector=1
start=0
end=-1

[autocounter]
readrate=0

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

[hostdebug]
# When enabled (=yes), Zeros-out FPGA-attached DRAM before simulations
# begin (takes 2-5 minutes).
# In general, this is not required to produce deterministic simulations on
# target machines running linux. Enable if you observe simulation non-determinism.
zerooutdram=no
# If disable_synth_asserts=no, simulation will print assertion message and
# terminate simulation if synthesized assertion fires.
# If disable_synth_asserts=yes, simulation ignores assertion firing and
# continues simulation.
disable_synth_asserts=no

[synthprint]
# Start and end cycles for outputing synthesized prints.
# They are given in terms of the base clock and will be converted
# for each clock domain.
start=0
end=-1
# When enabled (=yes), prefix print output with the target cycle at which the print␣
↪was triggered
cycleprefix=yes
```

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
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-rocket-quadcore-no-nic-l2-llc4mb-ddr3
```

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-rocket-quadcore-nic-l2-llc4mb-ddr3` to `firesim-rocket-quadcore-no-nic-l2-llc4mb-ddr3`. 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=0
m4_16xlarges=0
f1_2xlarges=1

runinstancemarket=ondemand
spotinterruptionbehavior=terminate
spotmaxprice=ondemand

[targetconfig]
topology=no_net_config
no_net_num_nodes=1
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-rocket-quadcore-no-nic-l2-llc4mb-ddr3
```

(continues on next page)

```
[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
> `firesim-boom-singlecore-no-nic-l2-llc4mb-ddr3`.

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 FireSim-FireSimQuadRocketConfig-F90MHz_BaseF1Config
[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] FireSim 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

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 br-base.json
```

This process will take about 10 to 15 minutes on a `c5.4xlarge` 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):

```ini
# 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
always_expand_runfarm=yes

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

launch_instances_timeout_minutes=60

runinstancemarket=ondemand
spotinterruptionbehavior=terminate
spotmaxprice=ondemand

[targetconfig]
#Set topology=no_net_config to run without a network simulation
topology=example_8config
no_net_num_nodes=2
linklatency=6405
switchinglatency=10
netbandwidth=200
profileinterval=-1

# This references a section from config_build_recipes.ini
# In homogeneous configurations, use this to set the hardware config deployed
# for all simulators
defaulthwconfig=firesim-rocket-quadcore-nic-l2-llc4mb-ddr3

[tracing]
enable=no

# Trace output formats. Only enabled if "enable" is set to "yes" above
# 0 = human readable; 1 = binary (compressed raw data); 2 = flamegraph (stack
# unwinding -> Flame Graph)
output_format=0

# Trigger selector.
# 0 = no trigger; 1 = cycle count trigger; 2 = program counter trigger; 3 =
# instruction trigger
selector=1
start=0
end=-1

[autocounter]
readrate=0
```

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

[hostdebug]
# When enabled (=yes), Zeros-out FPGA-attached DRAM before simulations
# begin (takes 2-5 minutes).
# In general, this is not required to produce deterministic simulations on
# target machines running linux. Enable if you observe simulation non-determinism.
zerooutdram=no
# If disable_synth_asserts=no, simulation will print assertion message and
# terminate simulation if synthesized assertion fires.
# If disable_synth_asserts=yes, simulation ignores assertion firing and
# continues simulation.
disable_synth_asserts=no

[synthprint]
# Start and end cycles for outputting synthesized prints.
# They are given in terms of the base clock and will be converted
# for each clock domain.
start=0
end=-1
# When enabled (=yes), prefix print output with the target cycle at which the print␣
→was triggered
cycleprefix=yes
```

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-rocket-quadcore-nic-l2-llc4mb-ddr3`: This tells the manager to use a quad-core Rocket Chip configuration with 512 KB of L2, 4 MB of L3 (LLC) 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
always_expand_runfarm=yes

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

launch_instances_timeout_minutes=60

runinstancemarket=ondemand
spotinterruptionbehavior=terminate
spotmaxprice=ondemand

[targetconfig]
#Set topology=no_net_config to run without a network simulation
topology=example_8config
no_net_num_nodes=2
linklatency=6405
switchinglatency=10
netbandwidth=200
profileinterval=-1

# This references a section from config_build_recipes.ini
# In homogeneous configurations, use this to set the hardware config deployed
# for all simulators
defaulthwconfig=firesim-rocket-quadcore-nic-l2-llc4mb-ddr3

[tracing]
enable=no

# Trace output formats. Only enabled if "enable" is set to "yes" above
# 0 = human readable; 1 = binary (compressed raw data); 2 = flamegraph (stack
# unwinding -> Flame Graph)
output_format=0

# Trigger selector.
# 0 = no trigger; 1 = cycle count trigger; 2 = program counter trigger; 3 =
# instruction trigger
selector=1
start=0
end=-1

[autocounter]
readrate=0

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

[hostdebug]
```

(continues on next page)

```
# When enabled (=yes), Zeros-out FPGA-attached DRAM before simulations
# begin (takes 2-5 minutes).
# In general, this is not required to produce deterministic simulations on
# target machines running linux. Enable if you observe simulation non-determinism.
zerooutdram=no
# If disable_synth_asserts=no, simulation will print assertion message and
# terminate simulation if synthesized assertion fires.
# If disable_synth_asserts=yes, simulation ignores assertion firing and
# continues simulation.
disable_synth_asserts=no

[synthprint]
# Start and end cycles for outputting synthesized prints.
# They are given in terms of the base clock and will be converted
# for each clock domain.
start=0
end=-1
# When enabled (=yes), prefix print output with the target cycle at which the print␣
→was triggered
cycleprefix=yes
```

> **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 `firesim-boom-singlecore-nic-l2-llc4mb-ddr3`.

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-FireSimQuadRocketConfig-F90MHz_BaseF1Config
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.
[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
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
```

(continues on next page)

```
[    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: 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
--------------------------------------------------------------------------------
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 `z1d.2xlarge`, so we default to that. If you do wish to use a different build instance type keep in mind that Vivado will consume in excess of 32 GiB for large designs.

To start out, let's build a simple design, `firesim-rocket-quadcore-no-nic-l2-llc4mb-ddr3`. This is a design that has four cores, no nic, and uses the 4MB LLC + DDR3 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-rocket-quadcore-no-nic-l2-llc4mb-ddr3
```

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:
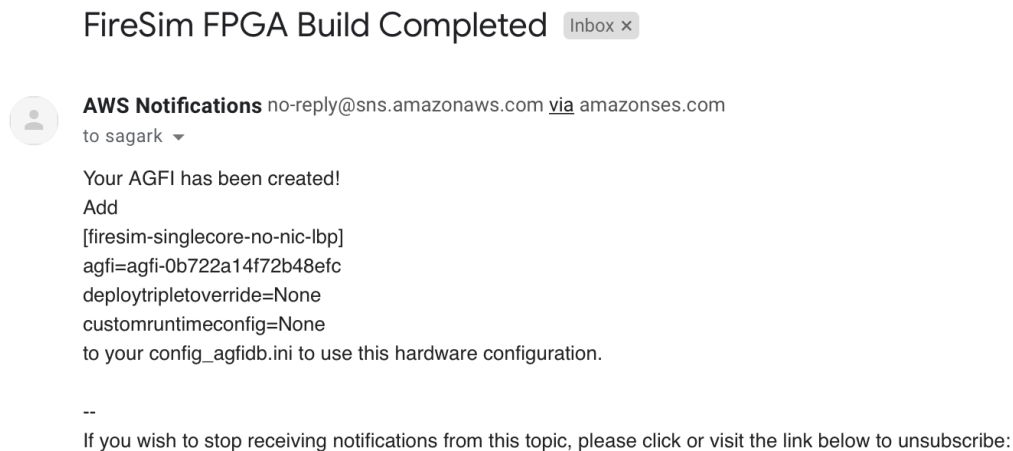
Fig. 1: Build Completion Email

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.

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
- Environment Variables
- 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 `--launchtime TIMESTAMP`

Specifies the "Y-m-d–H-M-S" timestamp to be used as the prefix in results-build directories. Useful when wanting to run tar2afi after an aborted buildafi was manually fixed.

5.2.7 `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.

- `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 tar2afi`

This command can be used to run only steps 9 & 10 from an aborted `firesim buildafi` that has been manually corrected. `firesim tar2afi` assumes that you have a `firesim/deploy/results-build/LAUNCHTIME-CONFIG_TRIPLET-BUILD_NAME/cl_firesim` directory tree that can be submitted to the AWS backend for conversion to an AFI.

When using this command, you need to also provide the `--launchtime LAUNCHTIME` cmdline argument, specifying an already existing LAUNCHTIME.

This command will run for the configurations specified in *config_build.ini* and *config_build_recipes.ini* as with *firesim buildafi*. It is likely that you may want to comment out BUILD_NAME that successfully completed *firesim buildafi* before running this command.

5.3.4 `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.

5.3.5 `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.

5.3.6 `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.

5.3.7 `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.

5.3.8 `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).

5.3.9 `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.

5.3.10 `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`.

5.3.11 `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]
runfarmtag=mainrunfarm
always_expand_runfarm=yes

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

launch_instances_timeout_minutes=60

runinstancemarket=ondemand
spotinterruptionbehavior=terminate
spotmaxprice=ondemand

[targetconfig]
#Set topology=no_net_config to run without a network simulation
topology=example_8config
no_net_num_nodes=2
linklatency=6405
switchinglatency=10
netbandwidth=200
profileinterval=-1

# This references a section from config_build_recipes.ini
# In homogeneous configurations, use this to set the hardware config deployed
# for all simulators
defaulthwconfig=firesim-rocket-quadcore-nic-l2-llc4mb-ddr3

[tracing]
enable=no

# Trace output formats. Only enabled if "enable" is set to "yes" above
# 0 = human readable; 1 = binary (compressed raw data); 2 = flamegraph (stack
# unwinding -> Flame Graph)
```

(continues on next page)

```
output_format=0

# Trigger selector.
# 0 = no trigger; 1 = cycle count trigger; 2 = program counter trigger; 3 =
# instruction trigger
selector=1
start=0
end=-1

[autocounter]
readrate=0

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

[hostdebug]
# When enabled (=yes), Zeros-out FPGA-attached DRAM before simulations
# begin (takes 2-5 minutes).
# In general, this is not required to produce deterministic simulations on
# target machines running linux. Enable if you observe simulation non-determinism.
zerooutdram=no
# If disable_synth_asserts=no, simulation will print assertion message and
# terminate simulation if synthesized assertion fires.
# If disable_synth_asserts=yes, simulation ignores assertion firing and
# continues simulation.
disable_synth_asserts=no

[synthprint]
# Start and end cycles for outputing synthesized prints.
# They are given in terms of the base clock and will be converted
# for each clock domain.
start=0
end=-1
# When enabled (=yes), prefix print output with the target cycle at which the print
→was triggered
cycleprefix=yes
```

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.

`always_expand_runfarm`

When `yes` (the default behavior when not given) the number of instances of each type (see `f1_16xlarges` etc. below) are launched every time you run `launchrunfarm`.

When `no`, `launchrunfarm` looks for already existing instances that match `runfarmtag` and treat `f1_16xlarges` (and other 'instance-type' values below) as a total count.

For example, if you have `f1_2xlarges` set to 100 and the first time you run `launchrunfarm` you have `launch_instances_timeout_minutes` set to 0 (i.e. giveup after receiving a `ClientError` for each AvailabilityZone) and AWS is only able to provide you 75 `f1_2xlarges` because of capacity issues, `always_expand_runfarm` changes the behavior of `launchrunfarm` in subsequent attempts. `yes` means `launchrunfarm` will try to launch 100 `f1_2xlarges` again. `no` means that `launchrunfarm` will only try to launch an additional 25 `f1_2xlarges` because it will see that there are already 75 that have been launched with the same `runfarmtag`.

`f1_16xlarges`, `m4_16xlarges`, `f1_4xlarges`, `f1_2xlarges`

Set these 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.

Also refer to `always_expand_runfarm` which determines whether `launchrunfarm` treats these counts as an incremental amount to be launched every time it is envoked or a total number of instances of that type and `runfarm` tag that should be made to exist. Note, `launchrunfarm` will never terminate instances.

`launch_instances_timeout_minutes`

Integer number of minutes that the `launchrunfarm` command will attempt to request new instances before giving up. This limit is used for each of the types of instances being requested. For example, if you set to 60, and you are requesting all four types of instances, `launchrunfarm` will try to launch each instance type for 60 minutes, possibly trying up to a total of four hours.

This limit starts to be applied from the first time `launchrunfarm` receives a `ClientError` response in all AvailabilityZones (AZs) for your region. In other words, if you request more instances than can possibly be requested in the given limit but AWS is able to satisfy all of the requests, the limit will not be enforced.

To experience the old (<= 1.12) behavior, set this limit to 0 and `launchrunfarm` will exit the first time it receives `ClientError` across all AZ's. The old behavior is also the default if `launch_instances_timeout_minutes` is not included in the `config_runtime.ini`.

`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.

profileinterval

The simulation driver periodically samples performance counters in FASED timing model instances and dumps the result to a file on the host. `profileinterval` defines the number of target cycles between samples; setting this field to -1 disables polling.

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...`).

[tracing]

This section manages TracerV-based tracing at simulation runtime. For more details, see the *Capturing RISC-V Instruction Traces with TracerV* page for more details.

enable

This turns tracing on, when set to `yes` and off when set to `no`. See the *Enabling Tracing at Runtime*.

output_format

This sets the output format for TracerV tracing. See the *Selecting a Trace Output Format* section.

selector, start, and end

These configure triggering for TracerV. See the *Setting a TracerV Trigger* section.

[autocounter]

This section configures AutoCounter. See the *AutoCounter: Profiling with Out-of-Band Performance Counter Collection* page for more details.

readrate

This sets the rate at which AutoCounters are read. See the *AutoCounter Runtime Parameters* section for more details.

[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. For a description of the JSON format, see *Defining Custom Workloads*.

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.

suffixtag

This allows you to append a string to a workload's output directory name, useful for differentiating between successive runs of the same workload, without renaming the entire workload. For example, specifying `suffixtag=test-v1` with a workload named `super-application` will result in a workload results directory named `results-workload/DATE--TIME-super-application-test-v1/`.

[hostdebug]

zerooutdram

Set this to `yes` to zero-out FPGA-attached DRAM before simulation begins. This process takes 2-5 minutes. In general, this is not required to produce deterministic simulations on target machines running linux, but should be enabled if you observe simulation non-determinism.

disable_synth_asserts

Set this to `yes` to make the simulation ignore synthesized assertions when they fire. Otherwise, simulation will print the assertion message and terminate when an assertion fires.

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

# Unnetworked designs use a three-domain configuration
# Tiles: 1600 MHz
#     <Rational Crossing>
# Uncore: 800 MHz
#     <Async Crossing>
# DRAM : 1000 MHz
firesim-rocket-quadcore-no-nic-l2-llc4mb-ddr3
firesim-boom-singlecore-no-nic-l2-llc4mb-ddr3

# All NIC-based designs use the legacy FireSim frequency selection, with the
# tiles and uncore running at 3.2 GHz to sustain 200Gb theoretical NIC BW
firesim-supernode-rocket-singlecore-nic-l2-lbp
firesim-rocket-quadcore-nic-l2-llc4mb-ddr3
firesim-boom-singlecore-nic-l2-llc4mb-ddr3

# SHA3 configs for tutorial
# firesim-singlecore-sha3-no-nic-l2-llc4mb-ddr3
# firesim-singlecore-sha3-print-no-nic-l2-llc4mb-ddr3

[agfistoshare]
firesim-rocket-quadcore-nic-l2-llc4mb-ddr3
firesim-rocket-quadcore-no-nic-l2-llc4mb-ddr3
firesim-boom-singlecore-no-nic-l2-llc4mb-ddr3
firesim-boom-singlecore-nic-l2-llc4mb-ddr3

firesim-supernode-rocket-singlecore-nic-l2-lbp

# SHA3 configs for tutorial
# firesim-singlecore-sha3-no-nic-l2-llc4mb-ddr3
# firesim-singlecore-sha3-print-no-nic-l2-llc4mb-ddr3

[sharewithaccounts]
# To share with a specific user:
somebodysname=123456789012
# To share publicly:
#public=public
```

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.

postbuildhook

(Optional) Provide an a script to run on the results copied back from a _single_ build instance. Upon completion of each design's build, the manager invokes this script and passing the absolute path to that instance's build-results directory as it's first argument.

[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-rocket-quadcore-nic-l2-llc4mb-ddr3]
# this is a comment that describes my favorite configuration!
agfi=agfi-0a6449b5894e96e53
deploytripletoverride=None
customruntimeconfig=None
```

you would use:

```
[agfistoshare]
firesim-rocket-quadcore-nic-l2-llc4mb-ddr3
```

[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

# Note: For large designs (ones that would fill a EC2.2xlarge/Xilinx VU9P)
# Vivado uses in excess of 32 GiB. Keep this in mind when selecting a
# non-default instancetype.


# Quad-core, Rocket-based recipes
[firesim-rocket-quadcore-nic-l2-llc4mb-ddr3]
DESIGN=FireSim
TARGET_CONFIG=WithNIC_DDR3FRFCFSLLC4MB_WithDefaultFireSimBridges_
↪WithFireSimHighPerfConfigTweaks_chipyard.QuadRocketConfig
PLATFORM_CONFIG=F90MHz_BaseF1Config
instancetype=z1d.2xlarge
deploytriplet=None

# NB: This has a faster host-clock frequency than the NIC-based design, because
# its uncore runs at half rate relative to the tile.
[firesim-rocket-quadcore-no-nic-l2-llc4mb-ddr3]
DESIGN=FireSim
```

(continues on next page)

```
TARGET_CONFIG=DDR3FRFCFSLLC4MB_WithDefaultFireSimBridges_
↪WithFireSimTestChipConfigTweaks_chipyard.QuadRocketConfig
PLATFORM_CONFIG=F140MHz_BaseF1Config
instancetype=z1d.2xlarge
deploytriplet=None


# Single-core, BOOM-based recipes
[firesim-boom-singlecore-nic-l2-llc4mb-ddr3]
DESIGN=FireSim
TARGET_CONFIG=WithNIC_DDR3FRFCFSLLC4MB_WithDefaultFireSimBridges_
↪WithFireSimHighPerfConfigTweaks_chipyard.LargeBoomConfig
PLATFORM_CONFIG=F65MHz_BaseF1Config
instancetype=z1d.2xlarge
deploytriplet=None

# NB: This has a faster host-clock frequency than the NIC-based design, because
# its uncore runs at half rate relative to the tile.
[firesim-boom-singlecore-no-nic-l2-llc4mb-ddr3]
DESIGN=FireSim
TARGET_CONFIG=DDR3FRFCFSLLC4MB_WithDefaultFireSimBridges_
↪WithFireSimTestChipConfigTweaks_chipyard.LargeBoomConfig
PLATFORM_CONFIG=F75MHz_BaseF1Config
instancetype=z1d.2xlarge
deploytriplet=None


# Single-core, CVA6-based recipes
[firesim-cva6-singlecore-no-nic-l2-llc4mb-ddr3]
DESIGN=FireSim
TARGET_CONFIG=DDR3FRFCFSLLC4MB_WithDefaultFireSimBridges_WithFireSimConfigTweaks_
↪chipyard.CVA6Config
PLATFORM_CONFIG=F90MHz_BaseF1Config
instancetype=z1d.2xlarge
deploytriplet=None


# Single-core, Rocket-based recipes with Gemmini
[firesim-rocket-singlecore-gemmini-no-nic-l2-llc4mb-ddr3]
DESIGN=FireSim
TARGET_CONFIG=DDR3FRFCFSLLC4MB_WithDefaultFireSimBridges_WithFireSimConfigTweaks_
↪chipyard.GemminiRocketConfig
PLATFORM_CONFIG=F30MHz_BaseF1Config
instancetype=z1d.2xlarge
deploytriplet=None


# RAM Optimizations enabled by adding _MCRams PLATFORM_CONFIG string
[firesim-boom-singlecore-no-nic-l2-llc4mb-ddr3-ramopts]
DESIGN=FireSim
TARGET_CONFIG=DDR3FRFCFSLLC4MB_WithDefaultFireSimBridges_
↪WithFireSimTestChipConfigTweaks_chipyard.LargeBoomConfig
PLATFORM_CONFIG=MCRams_F90MHz_BaseF1Config
instancetype=z1d.2xlarge
deploytriplet=None
```

```
# Supernode configurations -- multiple instances of an SoC in a single simulator
[firesim-supernode-rocket-singlecore-nic-l2-lbp]
DESIGN=FireSim
TARGET_CONFIG=WithNIC_SupernodeFireSimRocketConfig
PLATFORM_CONFIG=F85MHz_BaseF1Config
instancetype=z1d.2xlarge
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=z1d.2xlarge
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 be set to `FireSim`. We describe this in greater detail in *Generating Different Targets*).

`TARGET_CONFIG`

This specifies the hardware configuration of the target being simulated. Some examples include `FireSimRocketConfig` and `FireSimQuadRocketConfig`. We describe this in greater detail in *Generating Different Targets*).

`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`. We specify the host FPGA frequency in the `PLATFORM_CONFIG` by appending a frequency `Config` with an underscore (ex. BaseF1Config_F160MHz). We describe this in greater detail in *Generating Different Targets*).

`instancetype`

This defines the type of instance that the build will run on. Generally, running on a `z1d.2xlarge` 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.

[firesim-boom-singlecore-nic-l2-llc4mb-ddr3]
agfi=agfi-0989ed902efe56405
deploytripletoverride=None
customruntimeconfig=None


[firesim-boom-singlecore-no-nic-l2-llc4mb-ddr3]
agfi=agfi-0ece41b969f14b414
deploytripletoverride=None
customruntimeconfig=None


[firesim-rocket-quadcore-nic-l2-llc4mb-ddr3]
agfi=agfi-0878b36466b017ce0
deploytripletoverride=None
customruntimeconfig=None


[firesim-rocket-quadcore-no-nic-l2-llc4mb-ddr3]
agfi=agfi-0a8f04ece3a21b243
deploytripletoverride=None
customruntimeconfig=None


[firesim-supernode-rocket-singlecore-nic-l2-lbp]
```

```
agfi=agfi-05e9bda7b308d99ea
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-rocket-quadcore-nic-l2-llc4mb-ddr3]
# this is a comment that describes my favorite configuration!
agfi=agfi-0a6449b5894e96e53
deploytripletoverride=None
customruntimeconfig=None
```

[NAME_GOES_HERE]

In this example, `firesim-rocket-quadcore-nic-l2-llc4mb-ddr3` 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 Environment Variables

This page contains a centralized reference for the environment variables used by the manager.

5.5.1 `FIRESIM_RUNFARM_PREFIX`

This environment variable is used to prefix all Run Farm tags with some prefix. This is useful for separating run farms between multiple copies of FireSim.

This is set in `sourceme-f1-manager.sh`, so you can change it and commit it (e.g. if you're maintaining a branch for special runs). It can be unset or set to the empty string.

5.6 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`.

5.6.1 `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.
        """
```

(continues on next page)

```python
        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]])
```

```python
        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)
```

```python
    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()]
```

```python
        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)
```

```python
        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):
```

```python
        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)]
```

```python
        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)]

    # Spins up all of the precompiled, unnetworked targets
    def all_no_net_targets_config(self):
        hwdb_entries = [
            "firesim-boom-singlecore-no-nic-l2-llc4mb-ddr3",
            "firesim-rocket-quadcore-no-nic-l2-llc4mb-ddr3",
        ]
        assert len(hwdb_entries) == self.no_net_num_nodes
        self.roots = [FireSimServerNode(hwdb_entries[x]) for x in range(self.no_net_
→num_nodes)]


#    ######Used only for tutorial purposes####################
#    def example_sha3hetero_2config(self):
#        self.roots= [FireSimSwitchNode()]
#        servers = [FireSimServerNode(server_hardware_config=
#                    "fireboom-singlecore-nic-l2-llc4mb-ddr3"),
#              FireSimServerNode(server_hardware_config=
#                    "firesim-singlecore-sha3-nic-l2-llc4mb-ddr3")]
#        self.roots[0].add_downlinks(servers)
```

5.7 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.

Workloads

This section describes workload definitions in FireSim.

6.1 Defining Custom Workloads

This page documents the `JSON` input format that FireSim uses to understand your software workloads that run on the target design. Most of the time, you should not be writing these files from scratch. Instead, use *FireMarshal* to build a workload (including Linux kernel images and root filesystems) and use `firemarshal`'s `install` command to generate an initial `.json` file for FireSim. Once you generate a base `.json` with FireMarshal, you can add some of the options noted on this page to control additional files used as inputs/outputs to/from simulations.

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 uses these workload definitions to help the manager deploy your simulations. Historically, there was also a script to build workloads using these JSON files, but this has been replaced with a more powerful tool, *FireMarshal*. New workloads should always be built with *FireMarshal*.

In the following subsections, we will go through the two aforementioned example workload configurations, describing the how the manager uses each part of the JSON file inline.

The following examples use the default buildroot-based linux distribution (br-base). In order to customize Fedora, you should refer to the *Running Fedora on FireSim* page.

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-192-168-2-7.ec2.internal:~/firesim/deploy/workloads/linux-uniform$ ls -la
total 4
drwxrwxr-x  2 centos centos   69 Feb  8 00:07 .
drwxrwxr-x 19 centos centos 4096 Feb  8 00:39 ..
lrwxrwxrwx  1 centos centos   47 Feb  7 00:38 br-base-bin -> ../../../sw/firesim-
→software/images/br-base-bin
lrwxrwxrwx  1 centos centos   53 Feb  8 00:07 br-base-bin-dwarf -> ../../../sw/
→firesim-software/images/br-base-bin-dwarf
lrwxrwxrwx  1 centos centos   47 Feb  7 00:38 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. Additionally, you can use bash globbing for file names (ex: `file*name`).

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. Additionally, you can use bash globbing for file names (ex: `file*name`).

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.

```json
{
  "benchmark_name" : "ping-latency",
  "common_bootbinary" : "bbl-vmlinux",
  "common_outputs" : [],
  "common_simulation_inputs" : [],
  "common_simulation_outputs" : ["uartlog"],
  "no_post_run_hook": "",
  "workloads" : [
    {
      "name": "pinger",
      "simulation_inputs": [],
      "simulation_outputs": [],
      "outputs": []
    },
    {
      "name": "pingee",
      "simulation_inputs": [],
      "simulation_outputs": [],
      "outputs": []
    },
    {
      "name": "idler-1",
      "simulation_inputs": [],
      "simulation_outputs": [],
      "outputs": []
    },
    {
      "name": "idler-2",
      "simulation_inputs": [],
      "simulation_outputs": [],
      "outputs": []
    },
    {
      "name": "idler-3",
      "simulation_inputs": [],
      "simulation_outputs": [],
      "outputs": []
    },
    {
      "name": "idler-4",
      "simulation_inputs": [],
      "simulation_outputs": [],
      "outputs": []
    },
    {
      "name": "idler-5",
      "simulation_inputs": [],
      "simulation_outputs": [],
      "outputs": []
```

(continues on next page)

```
    },
    {
      "name": "idler-6",
      "simulation_inputs": [],
      "simulation_outputs": [],
      "outputs": []
    }
  ]
}
```

Additionally, let's take a look at the state of the `ping-latency` directory AFTER the workload is built (assume that a tool like *FireMarshal* already created the rootfses and linux images):

```
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 you probably don't want to commit to the repo.

- `idler-[1-6].ext2`, `pingee.ext2`, `pinger.ext2`: These are rootfses that we want to run on different nodes in our simulation. They can be generated with a tool like *FireMarshal*.

Next, let's review some of the new fields present in this JSON file:

- `common_simulation_inputs`: This is an array of extra files that you would like to supply to the simulator as *input*. One example is supplying files containing DWARF debugging info for TracerV + Stack Unwinding. See the *Modifying your workload description* section of the *TracerV + Flame Graphs: Profiling Software with Out-of-Band Flame Graph Generation* page for an example.

- `no_post_run_hook`: This is a placeholder for running a script on your manager automatically once your workload completes. To use this option, rename it to `post_run_hook` and supply a command to be run. The manager will automatically suffix the command with the path of the workload's results directory.

- `workloads`: This time, you'll notice that we have this array, which is populated by objects that represent individual jobs (note the naming discrepancy here, from here on out, we will refer to the contents of this array as **jobs** rather than **workloads**). 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.

- – `simulation_inputs`: Just like `common_simulation_inputs`, but specific to this job.

- – `simulation_outputs`: Just like `common_simulation_outputs`, but specific to this job.

- – `outputs`: Just like `common_outputs`, but specific to this job.

Because each of these jobs do not supply a `rootfs` field, the manager instead assumes that that the rootfs for each job is named `name.ext2`. To explicitly supply a rootfs name that is distinct from the job name, add the `rootfs` field to a job and supply a path relative to the workload's directory.

Once you specify the `.json` for this workload (and assuming you have built the corresponding rootfses with *FireMarshal*, 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 FireMarshal

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 `sw/firesim-software/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.

For more information, see the official FireMarshal documentation, and its quickstart tutorial.

6.3 SPEC 2017

SPEC2017 support in FireSim is provided through FireMarshal, which cross-compiles spec using Speckle in Chipyard. Build SPEC2017 in `<chipyard-dir>/target-software/spec2017`, and then install to FireSim's workload directory using FireMarshal's *install* command. See https://github.com/ucb-bar/spec2017-workload for more detail on the SPEC2017 workload definition.

When using reference inputs, SPEC workloads tend to complete within one to two days, but this varies strongly as a function of the target microarchitecture, FPGA frequency, and FMR.

6.4 Running Fedora on FireSim

FireSim also supports running a fedora-based linux workload. To build this workload, you can follow FireMarshal's quickstart guide (replace all instances of `br-base.json` with `fedora-base.json`).

To boot Fedora on FireSim, we provide a pre-written FireSim workload JSON `firesim/deploy/workloads/fedora-uniform.json`, that points to the generated Fedora images. Simply change the `workloadname` option

in your `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*).

6.5 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 `4769e5d86acf6a9508d2b5a63141dc80a6ef20a6` (Oct. 2019) of FireSim. After this commit, the Linux version in FireSim has been bumped past Linux 4.15. To reproduce workloads that rely on OS behavior that has changed, like `memcached-thread-imbalance`, you must use the last tested Oct. 2019 commit.

6.5.1 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.

6.5.2 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
```

6.5.3 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
```

6.5.4 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
```

6.5.5 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
```

6.5.6 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`.

6.5.7 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

6.5.8 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
```

6.6 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.

6.7 [DEPRECATED] Defining Custom Workloads

> **Danger:** This version of the Defining Custom Workloads page is kept here to document some of the legacy workload configurations still present in `deploy/workloads/`. New workloads should NOT be generated using these instructions. New workloads should be written by following the current version of the *Defining Custom Workloads* page.

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 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-base). 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 in QEMU). Imporantly, Fedora currently does not support the "command" option for workloads.

6.7.1 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.

6.7.2 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": [],
```

(continues on next page)

```
      "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": []
    },
    {
      "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`: Just like in the `linux-uniform` case, this workload just uses the default Linux binary generated in `firesim-software`. Note that it's named differently here, but still symlinks to `br-base-bin` in `linux-uniform`.

- `.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 ]
./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.

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.

7.1 Restrictions on Target RTL

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

1. The top-level module must have no inputs or outputs. Input stimulus and output capture must be implemented using target RTL or target-to-host Bridges.

2. All target clocks must be generated by a single `RationalClockBridge`.

3. Black boxes must be "clock-gateable" by replacing its input clock with a gated equivalent which will be used to stall simulation time in that module.

    a. As a consequence, target clock-gating cannot be implemented using black-box primitives, and must instead be modeled by adding clock-enables to all state elements of the gated clock domain (i.e., by adding an enable or feedback mux on registers to conditionally block updates, and by gating write-enables on memories).

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.

7.1.1 Including Verilog IP

FireSim now supports target designs that incorporate Verilog IP using the standard `BlackBox` interface from Chisel. For an example of how to add Verilog IP to a target system based on Rocket Chip, see the Incorporating Verilog Blocks section of the Chipyard documentation.

1. For the transform to work, the Chisel Blackbox that wraps the Verilog IP must have input clocks that can safely be clock-gated.

2. The compiler that produces the decoupled simulator ("FAME Transform") automatically recognizes such black-boxes inside the target design.

3. The compiler automatically gates each clock of the Verilog IP to ensure that it deterministically advances in lockstep with the rest of the simulator.

4. This allows any Verilog module, subject to the constraint above, to be instantiated anywhere in the target design using the standard Chisel Blackbox interface.

7.1.2 Multiple Clock Domains

FireSim can support simulating targets that have multiple clock domains. As stated above, all clocks must be generated using a single `RationalClockBridge`. For most users the default FireSim test harness in Chipyard will suffice, if you need to define a custom test harness instantiate the `RationalClockBridge` like so:

```scala
// Here we request three target clocks (the base clock is implicit). All
// clocks beyond the base clock are specified using the RationalClock case
// class which gives the clock domain's name, and its clock multiplier and
// divisor relative to the base clock.
val clockBridge = RationalClockBridge(RationalClock("HalfRate", 1, 2),
                                      RationalClock("ThirdRate", 1, 3))

// The clock bridge has a single output: a Vec[Clock] of the requested clocks
// in the order they were specified, which we are now free to use through our
// Chisel design.  While not necessary, here we unassign the Vec to give them
// more informative references in our Chisel.
val Seq(fullRate, halfRate, thirdRate) = clockBridge.io.clocks.toSeq
```

Further documentation can be found in the source (`sim/midas/src/main/scala/midas/widgets/ClockBridge.scala`).

The Base Clock

By convention, target time is specified in cycles of the *base clock*, which is defined to be the clock of the `RationalClockBridge` whose clock ratio (multiplier / divisor) is one. While we suggest making the base clock the fastest clock in your system, as in any microprocessor-based system it will likely correspond to your core clock frequency, this is not a constraint.

Limitations:

- The number of target clocks FireSim can simulate is bounded by the number of BUFGCE resources available on the host FPGA, as these are used to independently clock-gate each target clock.

- As its name suggests, the `RationalClockBridge` can only generate target clocks that are rationally related. Specifically, all requested frequencies must be expressable in the form:

$$f_i = \frac{f_{lcm}}{k_i}$$

Where,

- f_i is the desired frequency of the ith clock
- f_{lcm}, is the least-common multiple of all requested frequencies
- k_i is a 16-bit unsigned integer

An arbitrary frequency can be modeled using a sufficiently precise rational multiple. Golden Gate will raise a compile-time error if it cannot support a desired frequency.

- Each bridge module must reside entirely within a single clock domain. The Bridge's target interface must contain a single input clock, and all inputs and outputs of the bridge module must be latched and launched, respectively, by registers in the same clock domain.

7.2 Provided Target Designs

7.2.1 Target Generator Organization

FireSim provides multiple *projects*, each for a different type of target. Each project has its own chisel generator that invokes Golden Gate, 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 Golden Gate example designs, a set of simple chisel circuits like GCD, that demonstrate how to use Golden Gate. These are useful test cases for bringing up new Golden Gate 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 Golden Gate
                    └─{other-scala-sources}.scala
```

7.2.2 Specifying A Target Instance

To generate a specific instance of a target, the build system leverages four Make variables:

1. `TARGET_PROJECT`: this points the Makefile (*sim/Makefile*) at the right target-specific Makefrag, which defines the generation and meta-simulation software recipes. The makefrag for the default target project is defined at `sim/src/main/makefrag/firesim`.

2. `DESIGN`: the name of the top-level Chisel module to generate (a Scala class name). These are defined in FireChip Chipyard generator.

3. `TARGET_CONFIG`: specifies a `Config` instance that is consumed by the target design's generator. For the default firesim target project, predefined configs are described in in the FireChip Chipyard generator.

4. `PLATFORM_CONFIG`: specifies a `Config` instance that is consumed by Golden Gate and specifies compiler-level and host-land parameters, such as whether to enable assertion synthesis, or multi-ported RAM optimizations. Common platform configs are described in `firesim-lib/sim/src/main/scala/configs/CompilerConfigs.scala`).

`TARGET_CONFIG` and `PLATFORM_CONFIG` are strings that are used to construct a `Config` instance (derives from RocketChip's parameterization system, `Config`, see [freechips.rocketchip.config](#)). These strings are of the form "{..._}{<Class Name>_}<Class Name>". Only the final, base class name is compulsory: class names that are prepended with "_" are used to create a compound Config instance.

```scala
// Specify by setting TARGET_CONFIG=Base
class Base extends Config((site, here, up) => {...})
class Override1 extends Config((site, here, up) => {...})
class Override2 extends Config((site, here, up) => {...})
// Specify by setting TARGET_CONFIG=Compound
class Compound extends Config(new Override2 ++ new Override1 ++ new Base)
// OR by setting TARGET_CONFIG=Override2_Override1_Base
// Can specify undefined classes this way. ex: TARGET_CONFIG=Override2_Base
```

With this scheme, you don't need to define a Config class for every instance you wish to generate. We use this scheme to specify FPGA frequencies (eg. "BaseF1Config_F90MHz") in manager build recipes, but it's also very useful for doing sweeping over a parameterization space.

Note that the precedence of Configs decreases from left to right in a string. Appending a config to an existing one will only have an effect if it sets a field not already set in higher precendence Configs. For example, "BaseF1Config_F90MHz" is equivalent to "BaseF1Config_F90MHz_F80MHz" as `DesiredHostFrequency` resolves to 90 MHz, but "F90MHz_BaseF1Config" is distinct from "F80MHz_F90MHz_BaseF1Config" in that `DesiredHostFrequency` resolves to 90 and 80 MHz respectively.

How a particular Config resolves it's `Field`s can be unintuitive for complex compound `Config`s. One precise way to check a config is doing what you expect is to open the scala REPL, instantiate an instance of the desired `Config`, and inspect its fields.

```scala
$ make sbt # Launch into SBT's shell with extra FireSim arguments

sbt:firechip> console # Launch the REPL

scala> val inst = (new firesim.firesim.FireSimRocketChipConfig).toInstance # Make an
↪instance

inst: freechips.rocketchip.config.Config = FireSimRocketChipConfig

scala> import freechips.rocketchip.subsystem._ # Get some important Fields

import freechips.rocketchip.subsystem.RocketTilesKey

scala> inst(RocketTilesKey).size # Query number of cores

res2: Int = 1

scala> inst(RocketTilesKey).head.dcache.get.nWays # Query L1 D$ associativity

res3: Int = 4
```

7.3 Rocket Chip Generator-based SoCs (firesim project)

Using the Make variables listed above, we give examples of generating different targets using the default Rocket Chip-based target project.

7.3.1 Rocket-based SoCs

Three design classes use Rocket scalar in-order pipelines.

Single core, Rocket pipeline (default)

```
make TARGET_CONFIG=FireSimRocketConfig
```

Single-core, Rocket pipeline, with network interface

```
make TARGET_CONFIG=WithNIC_FireSimRocketChipConfig
```

Quad-core, Rocket pipeline

```
make TARGET_CONFIG=FireSimQuadRocketConfig
```

7.3.2 BOOM-based SoCs

The BOOM (Berkeley Out-of-Order Machine) superscalar out-of-order pipelines can also be used with the same design classes that the Rocket pipelines use. Only the TARGET_CONFIG needs to be changed, as shown below:

Single-core BOOM

```
make TARGET_CONFIG=FireSimLargeBoomConfig
```

Single-core BOOM, with network interface

```
make TARGET_CONFIG=WithNIC_FireSimBoomConfig
```

7.3.3 Generating A Different FASED Memory-Timing Model Instance

Golden Gate'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 PLATFORM_CONFIG=BaseF1Config`.

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

:: make TARGET_CONFIG=DDR3FRFCFS_FireSimRocketConfig

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

```
make TARGET_CONFIG=DDR3FRFCFSLLC4MB_FireSimRocketConfig
```

FASED *timing-model* configurations are passed to the FASED Bridges in your Target's FIRRTL, and so must be prepended to `TARGET_CONFIG`.

7.4 Midas Examples (midasexamples project)

This project can generate a handful of toy target-designs (set with the make variable `DESIGN`). Each of these designs has their own chisel source file and serves to demostrate the features of Golden Gate.

Some notable examples are:

1. `GCD`: the "Hello World!" of hardware.

2. `WireInterconnect`: demonstrates how combinational paths can be modeled with Golden Gate.

3. `PrintfModule`: demonstrates synthesizable printfs

4. `AssertModule`: demonstrates synthesizable assertions

To generate a target, set the make variable `TARGET_PROJECT=midasexamples.` so that the right project makefrag is sourced.

7.4.1 Examples

To generate the GCD midasexample:

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

7.5 FASED Tests (fasedtests project)

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

7.5.1 Examples

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

```
make TARGET_PROJECT=fasedtests DESIGN=AXI4Fuzzer TARGET_CONFIG=FRFCFSConfig
```

As above, now configured to drive 10 million transactions through the instance.

```
make TARGET_PROJECT=fasedtests DESIGN=AXI4Fuzzer TARGET_CONFIG=NT10e7_FRFCFSConfig
```

CHAPTER 8

Debugging in Software

This section describes methods of debugging the target design and the simulation in FireSim, *before running on the FPGA*.

8.1 Debugging & Testing with Meta-Simulation

When we speak of RTL simulation in FireSim, we are generally referring to *meta-simulation*: simulating the FireSim simulator's RTL, typically using VCS or verilator. In contrast, we we'll refer to simulation of the target's RTL as target-level simulation. Target-level simulation in Chipyard is described at length here.

Meta-simulation is the most productive way to catch bugs before generating an AGFI, and a means for reproducing bugs seen on the FPGA. By default, meta-simulation uses an abstract but fast model of the host: the FPGA's DRAM controllers are modeled with DRAMSim2, the PCI-E subsystem is not simulated, instead the driver presents DMA and MMIO traffic directly via verilog DPI. Since FireSim simulations are robust against timing differences across hosts, target behavior observed in an FPGA-hosted simulation should be exactly reproducible in a meta-simulation.

Generally, meta-simulators are only slightly slower than target-level ones. This illustrated in the chart below.

Type	Waves	VCS	Verilator	Verilator -O1	Verilator -O2
Target	Off	4.8 kHz	3.9 kHz	6.6 kHz	N/A
Target	On	0.8 kHz	3.0 kHz	5.1 kHz	N/A
Meta	Off	3.8 kHz	2.4 kHz	4.5 kHz	5.3 KHz
Meta	On	2.9 kHz	1.5 kHz	2.7 kHz	3.4 KHz

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

Type	Waves	VCS	Verilator	Verilator -O1	Verilator -O2
Meta	Off	35s	48s	3m32s	4m35s
Meta	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 between meta-simulation and target-level simulation. Hence, these numbers are only intended to ball park simulation speeds, not provide a scientific comparison between simulators. VCS numbers collected on a local Berkeley machine, Verilator numbers collected on a c4.4xlarge. (meta-simulation verilator version: 4.002, target-level verilator version: 3.904)

8.1.1 Running Meta-Simulation

Meta-simulations are run out of the `firesim/sim` directory.

```
[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/sourceme-f1-manager.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.

Additionally, you can run a unique binary in the following way:

```
make SIM_BINARY=<PATH_TO_BINARY> run-<vcs|verilator>
make SIM_BINARY=<PATH_TO_BINARY> run-<vcs|verilator>-debug
```

Examples

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

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

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

```
make verilator-debug
make -j run-asm-tests-debug
make -j run-bmark-tests-debug
```

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

```
make
make $(pwd)/output/f1/FireSim-FireSimRocketConfig-BaseF1Config/rv64ui-p-simple.out
```

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

```
make vcs-debug
make EMUL=vcs $(pwd)/output/f1/FireSim-FireSimRocketConfig-BaseF1Config/rv64ui-p-
→simple.vpd
```

8.1.2 Understanding A Meta-Simulation Waveform

Module Hierarchy

To build out a simulator, Golden Gate adds multiple layers of module hierarchy to the target design and performs additional hierarchy mutations to implement bridges and resource optimizations. Meta-simulation uses the `FPGATop` module as the top-level module, which excludes the platform shim layer (`F1Shim`, for EC2 F1). The original top-level of the input design is nested three levels below FPGATop:
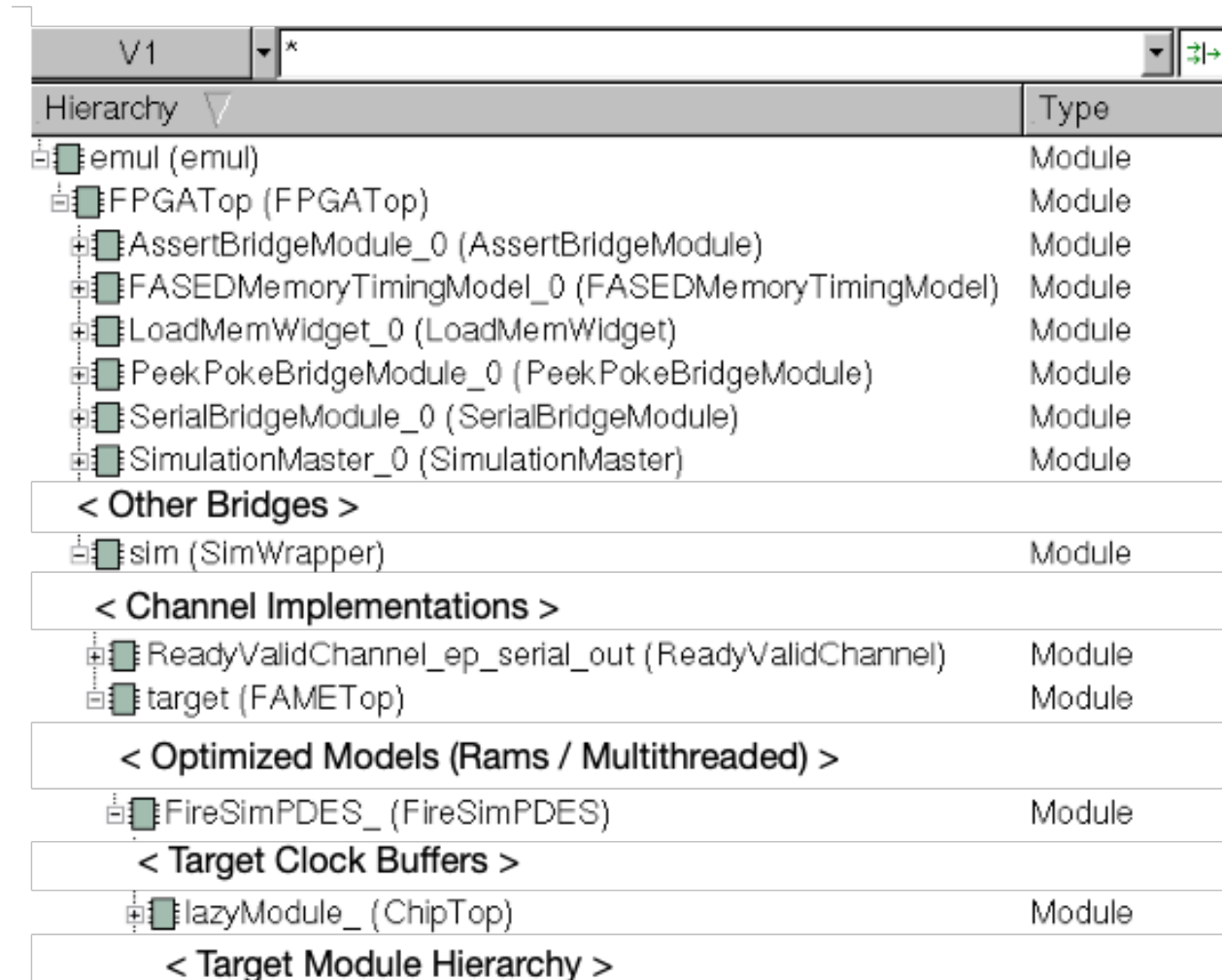

Fig. 1: The module hierarchy visible in a typical meta-simulation.

Note that many other bridges (under `FPGATop`), channel implementations (under `SimWrapper`), and optimized models (under `FAMETop`) may be present, and vary from target to target. Under the `FAMETop` module instance you will find the original top-level module (`FireSimPDES_`, in this case), however it has now been host-decoupled using the default LI-BDN FAME transformation and is referred to as the *hub model*. It will have ready-valid I/O interfaces for all of the channels bound to it, and internally containing additional channel enqueue and clock firing logic to control the advance of simulated time. Additionally, modules for bridges and optimized models will no longer be found contained in this submodule hierarchy. Instead, I/O for those extracted modules will now be as channel interfaces.

Clock Edges and Event Timing

Since FireSim derives target clocks by clock gating a single host clock, and since bridges and optimized models may introduce stalls of their own, timing of target clock edges in a meta-simulation will appear contorted relative to a conventional target-simulation. This is expected.

Finding The Source Of Simulation Stalls

In the best case, FireSim simulators will be able to launch new target clock pulses on every host clock cycle. In other words, for single-clock targets the simulation can run at FMR = 1. In the single clock case, delays are introduced by bridges (like FASED memory timing models) and optimized models. You can identify which bridges are responsible for additional delays between target clocks by filtering for input valid and output ready to the hub model. When input valid is deasserted, the corresponding bridge or model has not yet produced a token for the current timestep, effectively stalling the hub.

8.1.3 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 and FireSimNoNIC.

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 for a given project, with the sbt console open, do the familiar:

```
test
```

To run only tests on Rocket-Chip based targets, in the `firechip` SBT project run:

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

To run only the MIDAS examples, in the `firesim` SBT project:

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

Debugging and Profiling on the FPGA

A common issue with FPGA-prototyping is the difficulty involved in trying to debug and profile systems once they are running on the FPGA. FireSim addresses these issues with a variety of tools for introspecting on designs *once you have a FireSim simulation running on an FPGA*. This section describes these features.

9.1 Capturing RISC-V Instruction Traces with TracerV

FireSim can provide a cycle-by-cycle trace of a target CPU's architectural state over the course of execution, including fields like instruction address, raw instruction bits, privilege level, exception/interrupt status and cause, and a valid signal. This can be useful for profiling or debugging. **TracerV** is the FireSim bridge that provides this functionality. This feature was introduced in our FirePerf paper at ASPLOS 2020 .

This section details how to capture these traces in cycle-by-cycle formats, usually for debugging purposes.

For profiling purposes, FireSim also supports automatically producing stack traces from this data and producing Flame Graphs, which is documented in the *TracerV + Flame Graphs: Profiling Software with Out-of-Band Flame Graph Generation* section.

9.1.1 Building a Design with TracerV

In all FireChip designs, TracerV is included by default. Other targets can enable it by attaching a TracerV Bridge to the RISC-V trace port of each core they wish to trace (there should be one bridge per core). By default, only the cycle number, instruction address, and valid bit are collected.

9.1.2 Enabling Tracing at Runtime

To improve simulation preformance, FireSim does not collect and record data from the TracerV Bridge by default. To enable collection, modify the `enable` flag in the `[tracing]` section in your `config_runtime.ini` file to `yes` instead of `no`:

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

Now when you run a workload, a trace output file will be placed in the `sim_slot_<slot #>` directory on the F1 instance under the name `TRACEFILE-C0`. The `C0` represents core 0 in the simulated SoC. If you have multiple cores, each will have its own file (ending in `C1`, `C2`, etc). To copy all TracerV trace files back to your manager, you can add `TRACEFILE*` to your `common_simulation_outputs` or `simulation_outputs` in your workload `.json` file. See the *Defining Custom Workloads* section for more information about these options.

9.1.3 Selecting a Trace Output Format

FireSim supports three trace output formats, which can be set in your `config_runtime.ini` file with the `output_format` option in the `[tracing]` section:

```
[tracing]
enable=yes

# Trace output formats. Only enabled if "enable" is set to "yes" above
# 0 = human readable; 1 = binary (compressed raw data); 2 = flamegraph (stack
# unwinding -> Flame Graph)
output_format=0
```

See the "Interpreting the Trace Result" section below for a description of these formats.

9.1.4 Setting a TracerV Trigger

Tracing the entirety of a long-running job like a Linux-based workload can generate a large trace and you may only care about the state within a certain timeframe. Therefore, FireSim allows you to specify a trigger condition for starting and stopping trace data collection.

By default, TracerV does not use a trigger, so data collection starts at cycle 0 and ends at the last cycle of the simulation. To change this, modify the following under the `[tracing]` section of your `config_runtime.ini`. Use the `selector` field to choose the type of trigger (options are described below). The `start` and `end` fields are used to supply the start and end values for the trigger.

```
[tracing]
enable=yes

# Trace output formats. Only enabled if "enable" is set to "yes" above
# 0 = human readable; 1 = binary (compressed raw data); 2 = flamegraph (stack
# unwinding -> Flame Graph)
output_format=0

# Trigger selector.
# 0 = no trigger; 1 = cycle count trigger; 2 = program counter trigger; 3 =
# instruction trigger
selector=1
start=0
end=-1
```

The four triggering methods available in FireSim are as follows:

No trigger

Records the trace for the entire simulation.

This is option `0` in the `.ini` above.

The `start` and `end` fields are ignored.

Target cycle trigger

Trace recording begins when a specified start cycle is reached and ends when a specified end cycle is reached. Cycles are specified in base target-clock cycles (the zeroth output clock from the ClockBridge). For example, if the base clock drives the uncore, and the core clock frequency is 2x the uncore frequency, specifying start and end cycles of 100 and 200 result in instructions being collected between core-clock cycles 200 and 400.

This is option `1` in the `.ini` above.

The `start` and `end` fields are interpreted as decimal integers.

Program Counter (PC) value trigger

Trace recording begins when a specified program counter value is reached and ends when a specified program counter value is reached.

This is option `2` in the `.ini` above.

The `start` and `end` fields are interpreted as hexadecimal values.

Instruction value trigger

Trace recording begins when a specific instruction is seen in the instruction trace and ends when a specific instruction is seen in the instruction trace. This method is particularly valuable for setting the trigger from within the target software under evaluation, by inserting custom "NOP" instructions. Linux distributions included with FireSim include small trigger programs by default for this purpose; see the end of this subsection.

This is option `3` in the `.ini` above.

The `start` and `end` fields are interpreted as hexadecimal values. For each, the field is a 64-bit value, with the upper 32-bits representing a mask and the lower 32-bits representing a comparison value. That is, the start or stop condition will be satisfied when the following evaluates to true:

```
((inst value) & (upper 32 bits)) == (lower 32 bits)
```

That is, setting `start=ffffffff00008013` will cause recording to start when the instruction value is exactly `00008013` (the `addi x0, x1, 0` instruction in RISC-V).

This form of triggering is useful when recording traces only when a particular application is running within Linux. To simplify the use of this triggering mechanism, workloads derived from `br-base.json` in FireMarshal automatically include the commands `firesim-start-trigger` and `firesim-end-trigger`, which issue a `addi x0, x1, 0` and `addi x0, x2, 0` instruction respectively. In your `config_runtime.ini`, if you set the following trigger settings:

```
selector=3
start=ffffffff00008013
end=ffffffff00010013
```

And then run the following at the bash prompt on the simulated system:

```
$ firesim-start-trigger && ./my-interesting-benchmark && firesim-end-trigger
```

The trace will contain primarily only traces for the duration of `my-interesting-benchmark`. Note that there will be a small amount of extra trace information from `firesim-start-trigger` and `firesim-end-trigger`, as well as the OS switching between these and `my-interesting-benchmark`.

> **Attention:** While it is unlikely that a compiler will generate the aforementioned trigger instructions within normal application code, it is also a good idea to confirm that these instructions are not inadvertently present within the section of code you wish to profile. This will result in the trace recording inadvertently turning on and off in the middle of the workload.
>
> On the flip-side, a developer can deliberately insert the aforementioned `addi` instructions into the code they wish to profile, to enable more fine-grained control.

9.1.5 Interpreting the Trace Result

Human readable output

This is `output_format=0`.

The human readable trace output format looks like so:

```
# Clock Domain: baseClock, Relative Frequency: 1/1 of Base Clock
Cycle: 0000000000000079 I0: 0000000000010040
Cycle: 0000000000000105 I0: 000000000001004c
Cycle: 0000000000000123 I0: 0000000000010054
Cycle: 0000000000000135 I0: 0000000000010058
Cycle: 0000000000000271 I0: 000000000001005c
Cycle: 0000000000000307 I0: 0000000000010000
Cycle: 0000000000000327 I0: 0000000000010008
Cycle: 0000000000000337 I0: 0000000000010010
Cycle: 0000000000000337 I1: 0000000000010014
Cycle: 0000000000000337 I2: 0000000000010018
```

In this output, each line begins with the cycle (in decimal) in the core's clock domain that instruction was committed. For a given cycle, the instruction address (in hex) of each committed is prefixed `I<#>` according to their appearance in program order: `I0` is the oldest instruction committed, `I1` is the second oldest, and so forth. If no instructions were committed in a given cycle, that cycle will be skipped in the output file.

```
Cycle: 0000000000000337 I0: 0000000000010010
Cycle: 0000000000000337 I1: 0000000000010014
       |--------------|  ^        |--------|
              |          |              └ 40 bits of instruction address (hex)
              |          └ per-cycle commit-order
              └ 64-bit local-cycle count
```

Binary output

This is `output_format=1`.

This simply writes the 512 bits received from the FPGA each cycle to the output file in binary. Each 512-bit chunk is stored little-endian. The lowermost 64 bits stores the cycle, the second 64-bits stores the address and valid bits of

committed instruction 0 in little-endian, the next 64-bits stores the address and valid bits of committed instruction 1 in little-endian, and so on, up to a maximum of 7 instructions.

Flame Graph output

This is `output_format=2`. See the *TracerV + Flame Graphs: Profiling Software with Out-of-Band Flame Graph Generation* section.

9.1.6 Caveats

There are currently a few restrictions / manual tweaks that are required when using TracerV under certain conditions:

- TracerV by default outputs only instruction address and a valid bit and assumes that the combination of these fits within 64 bits. Changing this requires modifying `sim/firesim-lib/src/main/scala/bridges/TracerVBridge.scala`.
- The maximum IPC of the traced core cannot exceed 7.
- Please reach out on the FireSim mailing list if you need help addressing any of these restrictions: https://groups.google.com/forum/#!forum/firesim

9.2 Assertion Synthesis: Catching RTL Assertions on the FPGA

Golden Gate 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.

9.2.1 Enabling Assertion Synthesis

To enable assertion synthesis prepend `WithSynthAsserts` config to your PLATFORM_CONFIG. During compilation, Golden Gate will print the number of assertions it's synthesized. In the generated header, you will find the definitions of all synthesized assertions. The `synthesized_assertions_t` bridge driver will be automatically instantiated.

9.2.2 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: FireSimNoNIC.tile_1.core.issue_units_0.slots_3]
Assertion failed
    at issue_slot.scala:214 assert (!slot_p1_poisoned)
    at cycle: 2142042185
```

Just as in a software-hosted RTL simulation using verilator or VCS, the reported cycle is the number of target cycles that have elapsed in the clock domain in which the assertion was instantiated (in Chisel specifically this is the implicit clock at the time you called `assert`). If you rerun a FireSim simulation with identical inputs, the same assertion should fire deterministically at the same cycle.

9.2.3 Related Publications

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

9.3 Printf Synthesis: Capturing RTL printf Calls when Running on the FPGA

Golden Gate can synthesize printfs present in Chisel/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
```

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

9.3.1 Enabling Printf Synthesis

To synthesize a printf, you need to annotate the specific printfs you'd like to capture in your Chisel source code. 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, enable printf synthesis by prepending the `WithPrintfSynthesis` configuration mixin to your `PLATFORM_CONFIG` in `config_build_recipes.ini`. For example, if your previous `PLATFORM_CONFIG` was `PLATFORM_CONFIG=BaseF1Config_F120MHz`, then change it to `PLATFORM_CONFIG=WithPrintfSynthesis_BaseF1Config_F120MHz`. Notice that you must prepend the mixin (rather than appending). During compilation, Golden Gate will print the number of printfs it has synthesized. In the target's generated header (`FireSim-generated.const.h`), you'll find metadata for each of the printfs Golden Gate synthesized. This is passed as argument to the constructor of the `synthesized_prints_t` bridge driver, which will be automatically instantiated in FireSim driver.

9.3.2 Runtime Arguments

+print-file Specifies the file name prefix. Generated files will be of the form *<print-file><N>*, with one output file generated per clock domain. The associated clock domain's name and frequency relative to the base clock is

included in the header of the output file.

+print-start Specifies the target-cycle in cycles of the base clock 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 in cycles of the base clock 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.

You can set some of these options by changing the fields in the "synthprint" section of your config_runtime.ini.

```
[synthprint]
start=0
end=-1
cycleprefix=yes
```

The "start" field corresponds to "print-start", "end" to "print-end", and "cycleprefix" to "print-no-cycle-prefix".

9.3.3 Related Publications

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

9.4 AutoILA: Simple Integrated Logic Analyzer (ILA) Insertion

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.

The `midas.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.

9.4.1 Enabling AutoILA

To enable AutoILA, mixin *WithAutoILA* must be prepended to the *PLATFORM_CONFIG*. Prior to version 1.13, this was done by default.

9.4.2 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 Golden Gate 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.

9.4.3 Setting a ILA Depth

The ILA depth parameter specifies the duration in cycles to capture annotated signals around a trigger. Increasing this parameter may ease debugging, but will also increase FPGA resource utilization. The default depth is 1024 cycles. The desired depth can be configured much like the desired HostFrequency by appending a mixin to the *PLATFORM_CONFIG*. See *Provided Target Designs* for details on *PLATFORM_CONFIG*.

Below is an example *PLATFORM_CONFIG* that can be used in the *build_recipes* config file.

```
PLATFORM_CONFIG=ILADepth8192_BaseF1Config
```

9.4.4 Using the ILA at Runtime

Prerequisite: Make sure that ports 8443, 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. In the past, AWS had a custom `setup_gui.sh` script. However, this was recently deprecated due to compatibility issues with various packages. Therefore, AWS currently recommends using NICE DCV as a GUI client. You should download a DCV client, and then run the following commands on your FireSim manager instance:

```
sudo yum -y groupinstall "GNOME Desktop"
sudo yum -y install glx-utils
sudo rpm --import https://s3-eu-west-1.amazonaws.com/nice-dcv-publish/NICE-GPG-KEY
wget https://d1uj6qtbmh3dt5.cloudfront.net/2019.0/Servers/nice-dcv-2019.0-7318-el7.tgz
tar xvf nice-dcv-2019.0-7318-el7.tgz
cd nice-dcv-2019.0-7318-el7
sudo yum -y install nice-dcv-server-2019.0.7318-1.el7.x86_64.rpm
sudo yum -y install nice-xdcv-2019.0.224-1.el7.x86_64.rpm
sudo systemctl enable dcvserver
sudo systemctl start dcvserver
sudo passwd centos
sudo systemctl stop firewalld
dcv create-session --type virtual --user centos centos
```

These commands will setup Linux desktop pre-requisites, install the NICE DCV server, ask you to setup the password to the `centos` user, disable firewalld, and finally create a DCV session. You can now connect to this session through the DCV client.

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.

9.5 AutoCounter: Profiling with Out-of-Band Performance Counter Collection

FireSim can provide visibility into a simulated CPU's architectural and microarchitectural state over the course of execution through the use of counters. These are similar to performance counters provided by processor vendors, and more general counters provided by architectural simulators.

This functionality is provided by the AutoCounter feature (introduced in our FirePerf paper at ASPLOS 2020), and can be used for profiling and debugging. Since AutoCounter injects counters only in simulation (unlike target-level performance counters), these counters do not affect the behavior of the simulated machine, no matter how often they are sampled.

9.5.1 Chisel Interface

AutoCounter enables the addition of ad-hoc counters using the `PerfCounter` object in the *midas.targetutils* package. PerfCounters counters can be added in one of two modes:

Accumulate, using the standard **`PerfCounter.apply`** method. Here the annotated UInt (1 or more bits) is added to a 64b accumulation register: the target is treated as representing an N-bit UInt and will increment the counter by a value between [0, 2^n - 1] per cycle.

Identity, using the **`PerfCounter.identity`** method. Here the annotated UInt is sampled directly. This can be used to annotate a sample with values are not accumulator-like (e.g., a PC), and permits the user to define more complex instrumentation logic in the target itself.

We give examples of using PerfCounter below:

```
// A standard boolean event. Increments by 1 or 0 every local clock cycle.
midas.targetutils.PerfCounter(en_clock, "gate_clock", "Core clock gated")

// A multibit example. If the core can retire three isntructions per cycle,
// encode this as a two-bit unit. Extra-width is OK but the encoding to the UInt
// (e.g., doing a pop count), must be done by the user.
midas.targetutils.PerfCounter(insns_ret, "iret", "Instructions retired")

// An identity value. Note: the pc here must be <= 64b wide.
midas.targetutils.PerfCounter.identity(pc, "pc", "The value of the program counter at␣
↪the time of a sample")
```

See the PerfCounter Scala API docs for more detail about the Chisel-side interface.

9.5.2 Enabling AutoCounter in Golden Gate

By default, annotated events are not synthesized into AutoCounters. To enable AutoCounter when compiling a design, prepend the `WithAutoCounter` config to your `PLATFORM_CONFIG`. During compilation, Golden Gate will print the signals it is generating counters for.

9.5.3 Rocket Chip Cover Functions

The cover function is applied to various signals in the Rocket Chip generator repository to mark points of interest (i.e., interesting signals) in the RTL. Tools are free to provide their own implementation of this function to process these signals as they wish. In FireSim, these functions can be used as a hook for automatic generation of counters.

Since cover functions are embedded throughout the code of Rocket Chip (and possibly other code repositories), Auto-Counter provides a filtering mechanism based on module granularity. As such, only cover functions that appear within selected modules will generate counters.

The filtered modules can be indicated using one of two methods:

1. A module selection annotation within the top-level configuration implementation (when using Chipyard, this would usually be `DigitalTop`, but can also be any other module). To use this method, add the `AutoCounterCoverModuleAnnotation` annotation with the name of the module for which you want the cover functions to be turned into AutoCounters. The following example will generate counters from cover functions within the `StreamWriter` module:

```
class DigitalTop(implicit p: Parameters) extends ChipyardSystem
{
  override lazy val module = new DigitalTopModule(this)

  chisel3.experimental.annotate(AutoCounterCoverModuleAnnotation("StreamWriter"))
}
```

2. An input file with a list of module names. This input file is named `autocounter-covermodules.txt`, and includes a list of module names separated by new lines (no commas).

9.5.4 AutoCounter Runtime Parameters

AutoCounter currently takes a single runtime configurable parameter, defined under the `[autocounter]` section in the `config_runtime.ini` file. The `readrate` parameter defines the rate at which the counters should be read, and is measured in target-cycles of the base target-clock (clock 0 produced by the ClockBridge). Hence, if the read-rate is defined to be 100 and the tile frequency is 2x the base clock (ex., which may drive the uncore), the simulator will read and print the values of the counters every 200 core-clock cycles. If the core-domain clock is the base clock, it would do so every 100 cycles. By default, the read-rate is set to 0 cycles, which disables AutoCounter.

```
[autocounter]
# read counters every 100 cycles
readrate=100
```

Note: AutoCounter is designed as a coarse-grained observability mechanism, as sampling each counter requires two (blocking) MMIO reads (each read takes O(100) ns on EC2 F1). As a result sampling at intervals less than O(10000) cycles may adversely affect simulation performance for large numbers of counters. If you intend on reading counters at a finer granularity, consider using synthesizable printfs.

9.5.5 AutoCounter CSV Output Format

AutoCounter output files are CSVs generated in the working directory where the simulator was invoked (this applies to metasimulators too), with the default names `AUTOCOUNTERFILE<i>.csv`, one per clock domain. The CSV output format is depicted below, assuming a sampling period of `N` base clock cycles.

Table 1: AutoCounter CSV Format

version	*version number*				
clock info	domain name	multiplier	M	divisor	N
labels	local_clock	label0	label1	...	labelN
description	local clock cycle	desc0	desc1	...	descN
event width	1	width0	width1	...	widthN
acculator width	64	64	64	...	64
type	Increment	type0	type1	...	typeN
N	cycle @ time N	value0 @ tN	value1 @ tN	...	value @ tN
...	...	...	...	...	
kN	cycle @ time kN	value0 @ tkN	value1 @ tkN	...	valueN @ tkN

Column Notes:

1. Each column beyond the first two corresponds to a PerfCounter instance in the clock domain.

2. Column 0 past the header corresponds to the base clock cycle of the sample.

3. The local_cycle counter (column 1) is implemented as an always enabled single-bit event, and increments even when the target is under reset.

Row Notes:

1. Header row 0: autocounter csv format version, an integer.

2. Header row 1: clock domain information.

3. Header row 2: the label parameter provided to PerfCounter suffixed with the instance path.

4. Header row 3: the description parameter provided to PerfCounter. Quoted.

5. Header row 4: the width of the field annotated in the target.

6. Header row 5: the width of the accumulation register. Not configurable, but makes it clear when to expect rollover.

7. Header row 6: indicates the accumulation scheme. Can be "Identity" or "Accumulate".

8. Sample row 0: sampled values at the bitwidth of the accumulation register.

9. Sample row k: ditto above, k * N base cycles later

9.5.6 Using TracerV Trigger with AutoCounter

In order to collect AutoCounter results from only from a particular region of interest in the simulation, AutoCounter has been integrated with TracerV triggers. See the *Setting a TracerV Trigger* section for more information.

9.5.7 AutoCounter using Synthesizable Printfs

The AutoCounter transformation in Golden Gate includes an event-driven mode that uses Synthesizable Printfs (see *Printf Synthesis: Capturing RTL printf Calls when Running on the FPGA*) to export counter results *as they are updated* rather than sampling them periodically with a dedicated Bridge. This mode can be enabled by prepending the `WithAutoCounterCoverPrintf` config to your `PLATFORM_CONFIG` instead of `WithAutoCounterCover`. Based on the selected event mode the printfs will have the following runtime behavior:

- *Accumulate*: On a non-zero increment, the local cycle count and the new counter value are printed. This produces a series of prints with monotonically increasingly values.

- *Identity*: On a transition of the annotated target, the local cycle count and the new value are printed. Thus a target that transitions every cycle will produce printf traffic every cycle.

This mode may be useful for temporally fine-grained observation of counters. The counter values will be printed to the same output stream as other synthesizable printfs. This mode uses considerably more FPGA resources per counter, and may consume considerable amounts of DMA bandwidth (since it prints every cycle a counter increments), which may adversely affect simulation performance (increased FMR).

9.5.8 Reset & Timing Considerations

- Events and identity values provided while under local reset, or while the `GlobalResetCondition` asserted, are zero-ed out. Similarly, printfs that might otherwise be active under a reset are masked out.

- The sampling period in slower clock domains is currently calculated using a truncating division of the period in the base clock domain. Thus, when the base clock period can not be cleanly divided, samples in the slower clock domain will gradually fall out of phase with samples in the base clock domain. In all cases, the "local_cycle" column is most accurate measure of sample time.

9.6 TracerV + Flame Graphs: Profiling Software with Out-of-Band Flame Graph Generation

FireSim supports generating Flame Graphs out-of-band, to visualize the performance of software running on simulated processors. This feature was introduced in our FirePerf paper at ASPLOS 2020 .

Before proceeding, make sure you understand the *Capturing RISC-V Instruction Traces with TracerV* section.

9.6.1 What are Flame Graphs?

Fig. 1: Example Flame Graph (from http://www.brendangregg.com/FlameGraphs/)

Flame Graphs are a type of histogram that shows where software is spending its time, broken down by components of the stack trace (e.g., function calls). The x-axis represents the portion of total runtime spent in a part of the stack trace, while the y-axis represents the stack depth at that point in time. Entries in the flame graph are labeled with and sorted by function name (not time).

Given this visualization, time-consuming routines can easily be identified: they are leaves (top-most horizontal bars) of the stacks in the flame graph and consume a significant proportion of overall runtime, represented by the width of the horizontal bars.

Traditionally, data to produce Flame Graphs is collected using tools like `perf`, which sample stack traces on running systems in software. However, these tools are limited by the fact that they are ultimately running additional software on the system being profiled, which can change the behavior of the software that needs to be profiled. Furthermore, as sampling frequency is increased, this effect becomes worse.

In FireSim, we use the out-of-band trace collection provided by TracerV to collect these traces *cycle-exactly* and *without perturbing running software*. On the host-software side, TracerV unwinds the stack based on DWARF information about the running binary that you supply. This stack trace is then fed to the open-source FlameGraph stack trace visualizer to produce Flame Graphs.

9.6.2 Prerequisites

1. Make sure you understand the *Capturing RISC-V Instruction Traces with TracerV* section.

2. You must have a design that integrates the TracerV bridge. See the *Building a Design with TracerV* section.

9.6.3 Enabling Flame Graph generation in `config_runtime.ini`

To enable Flame Graph generation for a simulation, you must set `enable=yes` and `output_format=2` in the `[tracing]` section of your `config_runtime.ini` file, for example:

```
[tracing]
enable=yes

# Trace output formats. Only enabled if "enable" is set to "yes" above
# 0 = human readable; 1 = binary (compressed raw data); 2 = flamegraph (stack
# unwinding -> Flame Graph)
output_format=2

# Trigger selector.
# 0 = no trigger; 1 = cycle count trigger; 2 = program counter trigger; 3 =
# instruction trigger
selector=1
start=0
end=-1
```

The trigger selector settings can be set as described in the *Setting a TracerV Trigger* section. In particular, when profiling the OS only when a desired application is running (e.g., `iperf3` in our ASPLOS 2020 paper), instruction value triggering is extremely useful. See the *Instruction value trigger* section for more.

9.6.4 Producing DWARF information to supply to the TracerV driver

When running in FirePerf mode, the TracerV software driver expects a binary containing DWARF debugging information, which it will use to obtain labels for stack unwinding.

TracerV expects this file to be named exactly as your `bootbinary`, but suffixed with `-dwarf`. For example (and as we will see in the following section), if your `bootbinary` is named `br-base-bin`, TracerV will require you to provide a file named `br-base-bin-dwarf`.

If you are generating a Linux distribution with FireMarshal, this file containing debug information for the generated Linux kernel will automatically be provided (and named correctly) in the directory containing your images. For example, building the `br-base.json` workload will automatically produce `br-base-bin`, `br-base-bin-dwarf` (for TracerV flame graph generation), and `br-base.img`.

9.6.5 Modifying your workload description

Finally, we must make three modifications to the workload description to complete the flame graph flow. For general documentation on workload descriptions, see the *Defining Custom Workloads* section.

1. We must add the file containing our DWARF information as one of the `simulation_inputs`, so that it is automatically copied to the remote F1 instance running the simulation.

2. We must modify `simulation_outputs` to copy back the generated trace file.

3. We must set the `post_run_hook` to `gen-all-flamegraphs-fireperf.sh` (which FireSim puts on your path by default), which will produce flame graphs from the trace files.

To concretize this, let us consider the default `linux-uniform.json` workload, which does not support Flame Graph generation:

```
{
  "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"]
}
```

Below is the modified version of this workload, `linux-uniform-flamegraph.json`, which makes the afore-mentioned three changes:

```
{
  "benchmark_name"            : "linux-uniform",
  "common_bootbinary"         : "br-base-bin",
  "common_rootfs"             : "br-base.img",
  "common_simulation_inputs"  : ["br-base-bin-dwarf"],
  "common_outputs"            : ["/etc/os-release"],
  "common_simulation_outputs" : ["uartlog", "memory_stats*.csv", "TRACEFILE*"],
  "post_run_hook"             : "gen-all-flamegraphs-fireperf.sh"
}
```

Note that we are adding `TRACEFILE*` to `common_simulation_outputs`, which will copy back all generated trace files to your workload results directory. The `gen-all-flamegraphs-fireperf.sh` script will automatically produce a flame graph for each generated trace.

Lastly, if you have created a new workload definition, make sure you update your `config_runtime.ini` to use this new workload definition.

9.6.6 Running a simulation

At this point, you can follow the standard FireSim flow to run a workload. Once your workload completes, you will find trace files with stack traces (as opposed to instruction traces) and generated flame graph SVGs in your workload's output directory.

9.6.7 Caveats

The current stack trace construction code does not distinguish between different userspace programs, instead consolidating them into one entry. Expanded support for userspace programs will be available in a future release.

9.7 Dromajo Co-simulation with BOOM designs

Instead of using TracerV to provide a cycle-by-cycle trace of a target CPU's architectural state, you can use the Dromajo co-simulator to verify the functionality of a BOOM design.

Note: This work is highly experimental. We hope to integrate this into FireSim in a cleaner fashion at a later point.

Note: This work currently only works for single core BOOM designs.

9.7.1 Building a Design with Dromajo

In all FireChip designs, TracerV is included by default. To enable Dromajo, you just need to add the Dromajo bridge (`WithDromajoBridge`) to your BOOM target design config (default configs. located in `$CHIPYARD/generators/firechip/src/main/scala/TargetConfigs.scala`). An example configuration with Dromajo is shown below:

```scala
class FireSimLargeBoomConfig extends Config(
  new WithDromajoBridge ++ // add Dromajo bridge to simulation
  new WithDefaultFireSimBridges ++
  new WithDefaultMemModel ++
  new WithFireSimConfigTweaks ++
  new chipyard.LargeBoomConfig)
```

At this point, you should run the `firesim buildafi` command for the BOOM config wanted.

9.7.2 Running a FireSim Simulation

To run a simulation with Dromajo, you must modify the workload `json` to support Dromajo. The following is an example using the base Linux workload generated from FireMarshal and modifying it for Dromajo. Here is the modified workload json (renamed to `br-base-dromajo` from `br-base`):

```json
{
  "benchmark_name": "br-base-dromajo",
  "common_simulation_outputs": [
    "uartlog",
    "dromajo_snap.re_regs"
  ],
  "common_bootbinary": "../../../../../software/firemarshal/images/br-base-bin",
  "common_rootfs": "../../../../../software/firemarshal/images/br-base.img",
  "common_simulation_inputs": [
    "br-base-bin.rom",
    "br-base-bin.dtb"
  ]
}
```

You will notice there are two extra simulation inputs needed compared to the "base" unmodified `br-base` workload: a bootrom (`rom`) and a device tree blob (`dtb`). Both files are found in your generated sources and should be moved into the workload directory (i.e. `workloads/br-base-dromajo`).

```
cd $CHIPYARD

# copy/rename the rom file and put in the proper folder
cp sim/generated-src/f1/<LONG_NAME>/<LONG_NAME>.rom $FIRESIM/deploy/workloads/br-base-
→dromajo/br-base-bin.rom

# copy/rename the dtb file and put in the proper folder
cp sim/generated-src/f1/<LONG_NAME>/<LONG_NAME>.dtb $FIRESIM/deploy/workloads/br-base-
→dromajo/br-base-bin.dtb
```

After this process, you should see the following `workloads/br-base-dromajo` folder layout:

```
br-base-dromajo/
    br-base-bin.rom
    br-base-bin.dtb
    README
```

Note: The name of the rom and dtb files must match the name of the workload binary i.e. common_bootbinary.

At this point you are ready to run the simulation with Dromajo. The commit log trace will by default print to the uartlog. However, you can avoid printing it out by changing verbose == false in the dromajo_cosim. cpp file located in $CHIPYARD/tools/dromajo/dromajo-src/src/ folder.

9.7.3 Troubleshooting Dromajo Simulations with Meta-Simulations

If FPGA simulation fails with Dromajo, you can use meta-simulation to determine if your Dromajo setup is correct. First refer to *Debugging & Testing with Meta-Simulation* for more information on meta-simulation. The main difference between those instructions and simulations with Dromajo is that you need to manually point to the dtb, rom, and binary files when invoking the simulator. Here is an example of a make command that can be run to check for a correct setup.

```
# enter simulation directory
cd $FIRESIM/sim/

# make command to run a binary
# <BIN> - absolute path to binary
# <DTB> - absolute path to dtb file
# <BOOTROM> - absolute path to rom file
# <YourBoomConfig> - Single-core BOOM configuration to test
make TARGET_CONFIG=<YourBoomConfig> SIM_BINARY=<BIN> EXTRA_SIM_ARGS="+drj_dtb=<DTB>
↪+drj_rom=<BOOTROM> +drj_bin=<BIN>" run-vcs
```

It is important to have the +drj_* arguments, otherwise Dromajo will not match the simulation running on the DUT.

Note: Sometimes simulations in VCS will diverge unless a +define+RANDOM=0 is added to the VCS flags in sim/midas/src/main/cc/rtlsim/Makefrag-vcs.

Warning: Dromajo currently only works in VCS and FireSim simulations.

9.8 Debugging a Hanging Simulator

A common symptom of a failing simulation is that appears to hang. Debugging this is especially daunting in FireSim because it's not immediately obvious if it's a bug in the target, or somewhere in the host. To make it easier to identify the problem, the simulation driver includes a polling watchdog that tracks for simulation progress, and periodically updates an output file, heartbeat.csv, with a target cycle count and a timestamp. When debugging these issues, we always encourage the use of meta-simulation to try reproducing the failure if possible. We outline three common cases in the section below.

9.8.1 Case 1: Target hang.

Symptoms: There is no output from the target (i.e., the uartlog might cease), but simulated time continues to advance (heartbeat.csv will be periodically updated). Simulator instrumentation (TracerV, printf) may continue to produce new output.

Causes: Typically, a bug in the target RTL. However, bridge bugs leading to erroneous token values will also produce this behavior.

Next steps: You can deploy the full suite of FireSim's debugging tools for failures of this nature, since assertion synthesis, printf synthesis, and other target-side features still function. Assume there is a bug in the target RTL and trace back the failure to a bridge if applicable.

9.8.2 Case 2: Simulator hang due to FPGA-side token starvation.

Symptoms: The driver's main loop spins freely, as no bridge gets new work to do. As a result, the polling interval quickly elapses and the simulation is torn down due to a lack of forward progress.

Causes: Generally, a bug in a bridge implementation (ex. the BridgeModule has accidentally dequeued a token without producing a new output token; the BridgeModule is waiting on a driver interaction that never occurs).

Next steps: These are the trickiest to solve. Try to identify the bridge that's responsible by removing unnecessary ones, using an AutoILA, and adding printfs to BridgeDriver sources. Target-side debugging utilities may be used to identify problematic target behavior, but tend not to be useful for identifying the root cause.

9.8.3 Case 3: Simulator hang due to driver-side deadlock.

Symptoms: The loss of all output, notably, `heartbeat.csv` ceases to be further updated.

Causes: Generally, a bridge driver bug. For example, the driver may be busy waiting on some output from the FPGA, but the FPGA-hosted part of the simulator has stalled waiting for tokens.

Next Steps: Identify the buggy driver using printfs or attaching to the running simulator using GDB.

9.8.4 Simulator Heartbeat PlusArgs

`+heartbeat-polling-interval=<int>`: Specifies the number of round trips through the simulator main loop before polling the FPGA's target cycle counter. Disable the heartbeat by setting this to -1.

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.

10.1 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.

10.2 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 Chipyard in `chipyard/generators/firechip/src/main/scala/TargetConfigs.scala`. An example wrapper configuration is:

```
class SupernodeFireSimRocketConfig extends Config(
  new WithNumNodes(4) ++
  new freechips.rocketchip.subsystem.WithExtMemSize((1 << 30) * 8L) ++ // 8 GB
  new FireSimRocketConfig)
```

In this example, `SupernodeFireSimRocketConfig` is the wrapper, while `FireSimRocketConfig` 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 SupernodeFireSimQuadRocketConfig extends Config(
   new WithNumNodes(4) ++
   new freechips.rocketchip.subsystem.WithExtMemSize((1 << 30) * 8L) ++ // 8 GB
   new FireSimQuadRocketConfig)
```

Next, when defining the build recipe, we must remmber to use the supernode configuration: The `DESIGN` parameter should always be set to `FireSim`, while the `TARGET_CONFIG` parameter should be set to the wrapper configuration that was defined in `chipyard/generators/firechip/src/main/scala/TargetConfigs.scala`. The `PLATFORM_CONFIG` can be selected the same as in regular FireSim configurations. For example:

```
DESIGN=FireSim
TARGET_CONFIG=SupernodeFireSimQuadRocketConfig
PLATFORM_CONFIG=BaseF1Config
instancetype=c5.4xlarge
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`.

10.3 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()]
    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.

10.4 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"

Miscellaneous Tips

11.1 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.

11.2 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

11.3 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)

```
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`.

11.4 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.

11.5 Using FireSim CI

For more information on how to deal with the FireSim CI and how to run FPGA simulations in the CI, refer to the the `CI_README.md` under the `.github/` directory.

FireSim Asked Questions

12.1 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
```

12.2 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.

12.3 Help, My Simulation Hangs!

Oof. It can be difficult to pin this one down, read through *Debugging a Hanging Simulator* for some tips to get you started.

12.4 Should My Simulator Produce Different Results Across Runs?

No.

Unless you've intentionally introduced a side-channel (e.g., you're running an interactive simulation, or you've connected the NIC to the internet), this is likely a bug in one of your custom bridge implementations or in FireSim. In fact, for a given target-design, enabling printf synthesis, assertion synthesis, autocounter, or Auto ILA, should not change the simulated behavior of the machine.

12.5 Is there a way to compress workload results when copying back to the manager instance?

FireSim doesn't support compressing workload results before copying them back to the manager instance. Instead we recommend that you use a modern filesystem (like ZFS) to provide compression for you. For example, if you want to use ZFS to transparently compress data:

1. Attach a new volume to your EC2 instance (either at runtime or during launch). This is where data will be stored in a compressed format.

2. Make sure that the volume is attached (using something like `lsblk -f`). This new volume should not have a filesystem type and should be unmounted (volume name example: `nvme1n1`).

3. Install ZFS according to the ZFS documentation. Check `/etc/redhat-release` to verify the CentOS version of the manager instance.

4. Mount the volume and setup the ZFS filesystem with compression.

> **Warning:** Creating the zpool will destroy all pre-existing data on that partition. Double-check that the device node is correct before running any commands.

```
# replace /dev/nvme1n1 with the proper device node
zpool create -o ashift=12 -O compression=on <POOL_NAME> /dev/nvme1n1
zpool list
zfs list
```

5. At this point, you can use `/<POOL_NAME>` as a normal directory to store data into where it will be compressed. To see the compression ratio, use `zfs get compressratio`.

Overview & Philosophy

Underpinning FireSim is Golden Gate (MIDAS II), a FIRRTL-based compiler and C++ library, which is used to transform Chisel-generated RTL into a deterministic FPGA-accelerated simulator.

13.1 Golden Gate vs FPGA Prototyping

Key to understanding the design of Golden Gate, is understanding that Golden Gate-generated simulators are not FPGA prototypes. Unlike in a prototype, Golden Gate-generated simulators decouple the target-design clocks from all FPGA-host clocks (we say it is *host-decoupled*): one cycle in the target machine is simulated over a dynamically variable number FPGA clock cycles. In constrast, a conventional FPGA-prototype "emulates" the SoC by implementing the target directly in FPGA logic, with each FPGA-clock edge executing a clock edge of the SoC.

13.2 Why Use Golden Gate & FireSim

The host decoupling by Golden Gate-generated simulators enables:

1. **Deterministic simulation** Golden Gate creates a closed simulation environment such that bugs in the target can be reproduced despite timing-differences (eg. DRAM refresh, PCI-E transport latency) in the underlying host. The simulators for the same target can be generated for different host-FPGAs but will maintain the same target behavior.

2. **FPGA-host optimizations** Structures in ASIC RTL that map poorly to FPGA logic can be replaced with models that preserve the target RTL's behavior, but take more host cycles to save resources. eg. A 5R, 3W-ported register file with a dual-ported BRAM over 4 cycles.

3. **Distributed simulation & software co-simulation** Since models are decoupled from host time, it becomes much easier to host components of the simulator on multiple FPGAs, and on a host-CPU, while still preserving simulation determinism. This feature serves as the basis for building cycle-accurate scale-out systems with FireSim.

4. **FPGA-hosted, timing-faithful models of I/O devices** Most simple FPGA-prototypes use FPGA-attached DRAM to model the target's DRAM memory system. If the available memory system does not match that

of the target, the target's simulated performance will be artificially fast or slow. Host-decoupling permits writing detailed timing models that provide host-independent, deterministic timing of the target's memory system, while still use FPGA-host resources like DRAM as a functional store.

13.3 Why Not Golden Gate

Ultimately, Golden Gate-generated simulators introduce overheads not present in an FPGA-prototype that *may* increase FPGA resource use, decrease fmax, and decrease overall simulation throughput[1]. Those looking to develop soft-cores or develop a complete FPGA-based platform with their own boards and I/O devices would be best served by implementing their design directly on an FPGA. For those looking to building a system around Rocket-Chip, we'd suggest looking at SiFive's Freedom platform to start.

13.4 How is Host-Decoupling Implemented?

Host-decoupling in Golden Gate-generated simulators is implemented by decomposing the target machine into a dataflow graph of latency-insensitive models. As a user of FireSim, understanding this dataflow abstraction is essential for debugging your system and for developing your own software models and bridges. We describe it in the next section.

---

[1] These overheads varying depending on the features implemented and optimizations applied. Certain optimizations, currently in development, may increase fmax or decrease resource utilization over the equivalent prototype.

Target Abstraction & Host Decoupling

Golden Gate-generated simulators are deterministic, cycle-exact representations of the source RTL fed to the compiler. To achieve this, Golden Gate consumes input RTL (as FIRRTL) and transforms it into a latency-insensitive bounded dataflow network (LI-BDN) representation of the same RTL.

14.1 The Target as a Dataflow Graph

Dataflow graphs in Golden Gate consist of models, tokens, and channels:

1) Models – the nodes of the graph, these capture the behavior of the target machine by consuming and producing tokens.

2) Tokens – the messages of dataflow graph, these represent a hardware value as they would appear on a wire after they have converged for a given cycle.

3) Channels – the edges of the graph, these connect the output port of one model to the input of another.

In this system, time advances locally in each model. A model advances once cycle in simulated time when it consumes one token from each of its input ports and enqueues one token into each of its output ports. Models are *latency-insensitive*: they can tolerate variable input token latency as well as backpressure on output channels. Give a sequence of input tokens for each input port, a correctly implemented model will produce the same sequence of tokens on each of its outputs, regardless of when those input tokens arrive.

We give an example below of a dataflow graph representation of a 32-bit adder, simulating two cycles of execution.

14.2 Model Implementations

In Golden Gate, there are two dimensions of model implementation:

1) CPU- or FPGA-hosted: simply, where the model is going to execute. CPU-hosted models, being software, are more flexible and easy to debug but slow. Conversely, FPGA-hosted models are fast, but more difficult to write and debug.

2) Cycle-Exact or Abstract: cycle-exact models faithfully implement a chunk of the SoC's RTL~(this formalized later), where as abstract models are handwritten and trade fidelity for reduced complexity, better simulation performance, improved resource utilization, etc. . .

Hybrid, CPU-FPGA-hosted models are common. Here, a common pattern is write an RTL timing-model and a software functional model.

14.3 Expressing the Target Graph

The target graph is captured in the FIRRTL for your target. The bulk of the RTL for your system will be transformed by Golden Gate into one or more cycle-exact, FPGA-hosted models. You introduce abstract, FPGA-hosted models and CPU-hosted models into the graph by using Target-to-Host Bridges. During compilation, Golden Gate extracts the target-side of the bridge, and instantiates your custom RTL, called an BridgeModule, which together with a CPU-hosted Bridge Driver, gives you the means to model arbitrary target-behavior. We expand on this in the Bridge section.

14.4 Latency-Insensitive Bounded Dataflow Networks

In order for the resulting simulator to be a faithful representation of the target RTL, models must adhere to three properties. We refer the reader to the LI-BDN paper for the formal definitions of these properties. English language equivalents follow.

Partial Implementation: The model output token behavior matches the cycle-by-cyle output of the reference RTL, given the same input provided to both the reference RTL and the model (as a arbitrarily delayed token stream). Cycle exact models must implement PI, whereas abstract models do not.

The remaining two properties ensure the graph does not deadlock, and must be implemented by both cycle-exact and abstract models.

Self-Cleaning: A model that has enqueued N tokens into each of it's output ports *must* eventually dequeue N tokens from each of it's input ports.

No Extranenous Dependencies: If a given output channel of an LI-BDN simulation model has received a number of tokens no greater than any other channel, and if the model receives all input tokens required to compute the next output token for that channel, the model must eventually enqueue that output token, regardless of future external activity. Here, a model enqueueing an output token is synonymous with the corresponding output channel "receiving" the token.

Target-to-Host Bridges

A custom model in a FireSim Simulation, either CPU-hosted or FPGA-hosted, is deployed by using a *Target-to-Host Bridge*, or Bridge for short. Bridges provide the means to inject hardware and software models that produce and consume token streams.

Bridges enable:

1. **Deterministic, host-agnostic I/O models.** This is the most common use case. Here you instantiate bridges at the I/O boundary of your chip, to provide a simulation models of the environment your design is executing in. For an FPGA-hosted model, see FASED memory timing models. For co-simulated models see the UARTBridge, BlockDeviceBridge, and SerialBridge.

2. **Verification against a software golden model.** Attach an bridge (anywhere in your target RTL) to an interface you'd like to monitor, (e.g., a processor trace port). In the host, you can pipe the token stream coming off this interface to a software model running on a CPU (e.g, a functional ISA simulator). See TracerV.

3. **Distributed simulation.** The original FireSim application. You can stitch together networks of simulated machines by instantiating bridges at your SoC boundary. Then write software models and bridge drivers that move tokens between each FPGA. See the SimpleNICBridge.

4. **Resource optimizations.** Resource-intensive components of the target can be replaced with models that use fewer FPGA resources or run entirely in software.

The use of Bridges in a FireSim simulation has many analogs to doing mixed-language (Verilog-C++) simulation of the same system in software. Where possible, we'll draw analogies. After reading this page we encourage you to read the *Bridge Walkthrough*, which concretely explains the implementation of the UARTBridge.

15.1 Terminology

Bridges have a *target side*, consisting of a specially annotated Module, and *host side*, which consists of an FPGA-hosted *bridge module* (deriving from `BridgeModule`) and an optional CPU-hosted *bridge driver* (deriving from `bridge_driver_t`).

In a mixed-language software simulation, a verilog procedural interface (VPI) is analogous to the target side of a bridge, with the C++ backing that interface being the host side.

15.2 Target Side

In your target side, you will mix-in `midas.widgets.Bridge` into a Chisel `BaseModule` (this can be a black or white-box Chisel module) and implement its abstract members. This trait indicates that the associated module will be replaced with a connection to the host-side of the bridge that sources and sinks token streams. During compilation, the target-side module will be extracted by Golden Gate and its interface will be driven by your bridge's host-side implementation.

This trait has two type parameters and two abstract members you'll need define for your Bridge. Since you must mix `Bridge` into a Chisel `BaseModule`, the IO you define for that module constitutes the target-side interface of your bridge.

15.2.1 Type Parameters:

1. **Host Interface Type** `HPType <:  TokenizedRecord`: The Chisel type of your Bridge's host-land interface. This describes how the target interface has been divided into separate token channels. One example, `HostPortIO[T]`, divides a Chisel Bundle into a single bi-directional token stream and is sufficient for defining bridges that do not model combinational paths between token streams. We suggest starting with `HostPortIO[T]` when defining a Bridge for modeling IO devices, as it is the simplest to reasonable about and can run at FMR = 1. For other port types, see Bridge Host Interaces.

2. **BridgeModule Type** `WidgetType <:  BridgeModule`: The type of the host-land BridgeModule you want Golden Gate to connect in-place of your target-side module. Golden Gate will use its class name to invoke its constructor.

15.2.2 Abstract Members:

1. **Host Interface Mock** `bridgeIO:  HPType`: Here you'll instantiate a mock instance of your host-side interface. **This does not add IO to your target-side module**. Instead used to emit annotations that tell Golden Gate how the target-land IO of the target-side module is being divided into channels.

2. **Bridge Module Constructor Arg** `constructorArg:  Option[AnyRef]`: A optional Scala case class you'd like to pass to your host-land BridgeModule's constructor. This will be serialized into an annotation and consumed later by Golden Gate. If provided, your case class should capture all target-land configuration information you'll need in your Module's generator.

Finally at the bottom of your Bridge's class definition **you'll need to call generateAnnotations()**. This is necessary to have Golden Gate properly detect your bridge.

You can freely instantiate your Bridge anywhere in your Target RTL: at the I/O boundary of your chip or deep in its module hierarchy. Since all of the Golden Gate-specific metadata is captured in FIRRTL annotations, you can generate your target design and simulate it a target-level RTL simulation or even pass it off to ASIC CAD tools – Golden Gate's annotations will simply be unused.

15.3 What Happens Next?

If you pass your design to Golden Gate, it will find your target-side module, remove it, and wire its dangling target-interface to the top-level of the design. During host-decoupling transforms, Golden Gate aggregates fields of your bridge's target interface based on channel annotations emitted by the target-side of your bridge, and wraps them up into decoupled interfaces that match your host interface definition. Finally, once Golden Gate is done performing compiler transformations, it generates the bridge modules (by looking up their constructors and passing them their serialized constructor argument) and connects them to the tokenized interfaces presented by the now host-decoupled simulator.

15.4 Host Side

The host side of a bridge has two components:

1. A FPGA-hosted bridge module (`BridgeModule`).

2. An optional, CPU-hosted, bridge driver (`bridge_driver_t`).

In general, bridges have both: in FASED memory timing models, the BridgeModule contains a timing model that exposes timing parameters as memory-mapped registers that the driver configures at the start of simulation. In the Block Device model, the driver periodically polls queues in the bridge module checking for new functional requests to be served. In the NIC model, the driver moves tokens in bulk between the software switch model and the bridge module, which simply queues up tokens as they arrive.

Communication between a bridge module and driver is implemented with two types of transport:

1. **MMIO**: In the module, this is implemented over a 32-bit AXI4-lite bus. Reads and writes to this bus are made by drivers using `simif_t::read()` and `simif_t::write()`. Bridge modules register memory mapped registers using methods defined in `midas.widgets.Widget`, addresses for these registers are passed to the drivers in a generated C++ header.

2. **DMA**: In the module this is implemented with a wide (e.g., 512-bit) AXI4 bus, that is mastered by the CPU. Bridge drivers initiate bulk transactions by passing buffers to `simif_t::push()` and `simif_t::pull()` (DMA from the FPGA). DMA is typically used to stream tokens into and out of out of large FIFOs in the BridgeModule.

15.5 Compile-Time (Parameterization) vs Runtime Configuration

As when compiling a software RTL simulator, the simulated design is configured over two phases:

1. **Compile Time**, by parameterizing the target RTL and BridgeModule generators, and by enabling Golden Gate optimization and debug transformations. This changes the simulator's RTL and thus requires a FPGA-recompilation. This is equivalent to, but considerably slower than, invoking VCS to compile a new simulator.

2. **Runtime**, by specifying plus args (e.g., +latency=1) that are passed to the BridgeDrivers. This is equivalent to passing plus args to a software RTL simulator, and in many cases the plus args passed to an RTL simulator and a FireSim simulator can be the same.

15.6 Target-Side vs Host-Side Parameterization

Unlike in a software RTL simulation, FireSim simulations have an additional phase of RTL elaboration, during which bridge modules are generated (they are themselves Chisel generators).

The parameterization of your bridge module can be captured in two places.

1. **Target side.** here parameterization information is provided both as free parameters to the target's generator, and extracted from the context in which the bridge is instantiated. The latter might include things like widths of specific interfaces or bounds on the behavior the target might expose to the bridge (e.g., a maximum number of inflight requests). All of this information must be captured in a _single_ serializable constructor argument, generally a case class (see `Bridge.constructorArg`).

2. **Host side.** This is parameterization information captured in Golden Gate's `Parameters` object. This should be used to provide host-land implementation hints (that ideally don't change the simulated behavior of the system), or to provide arguments that cannot be serialized to the annotation file.

In general, if you can capture target-behavior-changing parameterization information from the target-side you should. This makes it easier to prevent divergence between a software RTL simulation and FireSim simulation of the same FIRRTL. It's also easier to configure multiple instances of the same type of bridge from the target side.

Bridge Walkthrough

In this section, we'll walkthrough a simple Target-to-Host bridge, the UARTBridge, provided with FireSim to demonstrate how to integrate your own. The UARTBridge uses host-MMIO to model a UART device.

Reading the Bridges section is a prerequisite to reading these sections.

16.1 UART Bridge (Host-MMIO)

Source code for the UART Bridge lives in the following directories:

```
sim/
├─firesim-lib/src/main/
│                  ├─scala/bridges/UARTBridge.scala # Target-Side Bridge and␣
↪BridgeModule Definitions
│                  ├─cc/brides/uart.cc # Bridge Driver source
│                  └─cc/brides/uart.h  # Bridge Driver header
├─src/main/cc/firesim/firesim_top.cc  # Driver instantiation in the main simulation␣
↪driver
└─src/main/makefrag/firesim/Makefrag  # Build system modifications to compile Bridge␣
↪Driver code
```

16.1.1 Target Side

The first order of business when designing a new bridge is to implement its target side. In the case of UART we've defined a Chisel BlackBox[1] extending Bridge. We'll instantiate this BlackBox and connect it to UART IO in the top-level of our chip. We first define a class that captures the target-side interface of the Bridge:

---

[1] You can also extend a non-BlackBox Chisel Module, but any Chisel source contained within will be removed by Golden Gate. You may wish to do this to enclose a synthesizable model of the Bridge for other simulation backends, or simply to wrap a larger chunk RTL you wish to model in the host-side of the Bridge.

```scala
class UARTBridgeTargetIO(val uParams: UARTParams) extends Bundle {
  val clock = Input(Clock())
  val uart = Flipped(new UARTPortIO(uParams))
  // Note this reset is optional and used only to reset target-state modelled
  // in the bridge This reset just like any other Bool included in your target
  // interface, simply appears as another Bool in the input token.
  val reset = Input(Bool())
}
```

Here, we define a case class that carries additional metadata to the host-side BridgeModule. For UART, this is simply the clock-division required to produce the baudrate:

```scala
// Out bridge module constructor argument. This captures all of the extra
// metadata we'd like to pass to the host-side BridgeModule. Note, we need to
// use a single case class to do so, even if it is simply to wrap a primitive
// type, as is the case for UART (int)
case class UARTKey(uParams: UARTParams, div: Int)
```

Finally, we define the actual target-side module (specifically, a BlackBox):

```scala
class UARTBridge(uParams: UARTParams)(implicit p: Parameters) extends BlackBox
    with Bridge[HostPortIO[UARTBridgeTargetIO], UARTBridgeModule] {
  // Since we're extending BlackBox this is the port will connect to in our target's
  ↪RTL
  val io = IO(new UARTBridgeTargetIO(uParams))
  // Implement the bridgeIO member of Bridge using HostPort. This indicates that
  // we want to divide io, into a bidirectional token stream with the input
  // token corresponding to all of the inputs of this BlackBox, and the output token
  ↪consisting of
  // all of the outputs from the BlackBox
  val bridgeIO = HostPort(io)

  // Do some intermediate work to compute our host-side BridgeModule's constructor
  ↪argument
  val frequency = p(PeripheryBusKey).dtsFrequency.get
  val baudrate = uParams.initBaudRate
  val div = (frequency / baudrate).toInt

  // And then implement the constructorArg member
  val constructorArg = Some(UARTKey(uParams, div))

  // Finally, and this is critical, emit the Bridge Annotations -- without
  // this, this BlackBox would appear like any other BlackBox to Golden Gate
  generateAnnotations()
}
```

To make it easier to instantiate our target-side module, we've also defined an optional companion object:

```scala
object UARTBridge {
  def apply(clock: Clock, uart: UARTPortIO)(implicit p: Parameters): UARTBridge = {
    val ep = Module(new UARTBridge(uart.c))
    ep.io.uart <> uart
    ep.io.clock := clock
    ep
  }
}
```

That completes the target-side definition.

16.1.2 Host-Side BridgeModule

The remainder of the file is dedicated to the host-side BridgeModule definition. Here we have to process tokens generated by the target, and expose a memory-mapped interface to the bridge driver.

Inspecting the top of the class:

```
// Our UARTBridgeModule definition, note:
// 1) it takes one parameter, key, of type UARTKey --> the same case class we
↪captured from the target-side
// 2) It accepts one implicit parameter of type Parameters
// 3) It extends BridgeModule passing the type of the HostInterface
//
// While the Scala type system will check if you parameterized BridgeModule
// correctly, the types of the constructor arugument (in this case UARTKey),
// don't match, you'll only find out later when Golden Gate attempts to generate your
↪module.
class UARTBridgeModule(key: UARTKey)(implicit p: Parameters) extends
↪BridgeModule[HostPortIO[UARTBridgeTargetIO]]()(p) {
  lazy val module = new BridgeModuleImp(this) {
    val div = key.div
    // This creates the interfaces for all of the host-side transport
    // AXI4-lite for the simulation control bus, =
    // AXI4 for DMA
    val io = IO(new WidgetIO())

    // This creates the host-side interface of your TargetIO
    val hPort = IO(HostPort(new UARTBridgeTargetIO(key.uParams)))

    // Generate some FIFOs to capture tokens...
    val txfifo = Module(new Queue(UInt(8.W), 128))
    val rxfifo = Module(new Queue(UInt(8.W), 128))

    val target = hPort.hBits.uart
    // In general, your BridgeModule will not need to do work every host-cycle. In
↪simple Bridges,
    // we can do everything in a single host-cycle -- fire captures all of the
    // conditions under which we can consume and input token and produce a new
    // output token
    val fire = hPort.toHost.hValid && // We have a valid input token: toHost ~=
↪leaving the transformed RTL
              hPort.fromHost.hReady && // We have space to enqueue a new output token
              txfifo.io.enq.ready      // We have space to capture new TX data
    val targetReset = fire & hPort.hBits.reset
    rxfifo.reset := reset.asBool || targetReset
    txfifo.reset := reset.asBool || targetReset

    hPort.toHost.hReady := fire
    hPort.fromHost.hValid := fire
```

Most of what follows is responsible for modeling the timing of the UART. As a bridge designer, you're free to take as many host-cycles as you need to process tokens. In simpler models, like this one, it's often easiest to write logic that operates in a single cycle but gate state-updates using a "fire" signal that is asserted when the required tokens are available.

Now, we'll skip to the end to see how to add registers to the simulator's memory map that can be accessed using MMIO from bridge driver.

```
    // Exposed the head of the queue and the valid bit as a read-only registers
    // with name "out_bits" and out_valid respectively
    genROReg(txfifo.io.deq.bits, "out_bits")
    genROReg(txfifo.io.deq.valid, "out_valid")

    // Generate a writeable register, "out_ready", that when written to dequeues
    // a single element in the tx_fifo. Pulsify derives the register back to false
    // after pulseLength cycles to prevent multiple dequeues
    Pulsify(genWORegInit(txfifo.io.deq.ready, "out_ready", false.B), pulseLength = 1)

    // Generate regisers for the rx-side of the UART; this is eseentially the reverse␣
↪of the above
    genWOReg(rxfifo.io.enq.bits, "in_bits")
    Pulsify(genWORegInit(rxfifo.io.enq.valid, "in_valid", false.B), pulseLength = 1)
    genROReg(rxfifo.io.enq.ready, "in_ready")

    // This method invocation is required to wire up all of the MMIO registers to
    // the simulation control bus (AXI4-lite)
    genCRFile()
```

16.1.3 Host-Side Driver

To complete our host-side definition, we need to define a CPU-hosted bridge driver. Bridge Drivers extend the
`bridge_driver_t` interface, which declares 5 virtual methods a concrete bridge driver must implement:

```
// Bridge Drivers are the CPU-hosted component of a Target-to-Host Bridge. A
// Bridge Driver interacts with their accompanying FPGA-hosted BridgeModule
// using MMIO (via read() and write() methods) or CPU-mastered DMA (via pull()
// and push()).

class bridge_driver_t
{
public:
  bridge_driver_t(simif_t* s): sim(s) { }
  virtual ~bridge_driver_t() {};
  // Initialize BridgeModule state -- this can't be done in the constructor currently
  virtual void init() = 0;
  // Does work that allows the Bridge to advance in simulation time (one or more␣
↪cycles)
  // The standard FireSim driver calls the tick methods of all registered bridge␣
↪drivers.
  // Bridges whose BridgeModule is free-running need not implement this method
  virtual void tick() = 0;
  // Indicates the simulation should terminate.
  // Tie off to false if the brige will never call for the simulation to teriminate.
  virtual bool terminate() = 0;
  // If the bridge driver calls for termination, encode a cause here. 0 = PASS All␣
↪other
  // codes are bridge-implementation defined
  virtual int exit_code() = 0;
  // The analog of init(), this provides a final opportunity to interact with
  // the FPGA before destructors are called at the end of simulation. Useful
  // for doing end-of-simulation clean up that requires calling {read,write,push,pull}
↪.
  virtual void finish() = 0;
```

The declaration of the Uart bridge driver lives at `sim/firesim-lib/src/main/cc/bridges/uart.h`. It is inlined below:

```
//See LICENSE for license details
#ifndef __UART_H
#define __UART_H

#include "serial.h"
#include <signal.h>

// The definition of the primary constructor argument for a bridge is generated
// by Golden Gate at compile time _iff_ the bridge is instantiated in the
// target. As a result, all bridge driver definitions conditionally remove
// their sources if the constructor class has been defined (the
// <cname>_struct_guard macros are generated along side the class definition.)
//
// The name of this class and its guards are always BridgeModule class name, in
// all-caps, suffixed with "_struct" and "_struct_guard" respectively.

#ifdef UARTBRIDGEMODULE_struct_guard
class uart_t: public bridge_driver_t
{
    public:
        uart_t(simif_t* sim, UARTBRIDGEMODULE_struct * mmio_addrs, int uartno);
        ~uart_t();
        virtual void tick();
        // Our UART bridge's initialzation and teardown procedures don't
        // require interaction with the FPGA (i.e., MMIO), and so we don't need
        // to define init and finish methods (we can do everything in the
        // ctor/dtor)
        virtual void init() {};
        virtual void finish() {};
        // Our UART bridge never calls for the simulation to terminate
        virtual bool terminate() { return false; }
        // ... and thus, never returns a non-zero exit code
        virtual int exit_code() { return 0; }

    private:
        UARTBRIDGEMODULE_struct * mmio_addrs;
        serial_data_t<char> data;
        int inputfd;
        int outputfd;
        int loggingfd;
        void send();
        void recv();
};
#endif // UARTBRIDGEMODULE_struct_guard

#endif // __UART_H
```

The bulk of the driver's work is done in its `tick()` method. Here, the driver polls the BridgeModule and then does some work. Note: the name, `tick` is vestigial: one invocation of tick() may do work corresponding to an arbitrary number of target cycles. It's critical that tick be non-blocking, as waiting for work from the BridgeModule may deadlock the simulator.

16.1.4 Registering the Driver

With the Bridge Driver implemented, we now have to register it in the main simulator simulator class defined in `sim/src/main/cc/firesim/firesim_top.cc`. Here, we rely on the C preprocessor macros to instantiate the bridge driver only when the corresponding BridgeModule is present:

```
   // Here we instantiate our driver once for each bridge in the target
   // Golden Gate emits a <BridgeModuleClassName>_<id>_PRESENT macro for each
→instance
   // which you may use to conditionally instantiate your driver
   #ifdef UARTBRIDGEMODULE_0_PRESENT
   // Create an instance of the constructor argument (this has all of
   // addresses of the BridgeModule's memory mapped registers)
   UARTBRIDGEMODULE_0_substruct_create;
   // Instantiate the driver; register it in the main simulation class
   add_bridge_driver(new uart_t(this, UARTBRIDGEMODULE_0_substruct, 0));
   #endif

   // Repeat the code above with modified indices as many times as necessary
   // to support the maximum expected number of bridge instances
   #ifdef UARTBRIDGEMODULE_1_PRESENT
   UARTBRIDGEMODULE_1_substruct_create;
   add_bridge_driver(new uart_t(this, UARTBRIDGEMODULE_1_substruct, 1));
   #endif
```

16.1.5 Build-System Modifications

The final consideration in adding your bridge concerns the build system. You should be able to host the Scala sources for your bridge with rest of your target RTL: SBT will make sure those classes are available on the runtime classpath. If you're hosting your bridge driver sources outside of the existing directories, you'll need to modify your target-project Makefrag to include them. The default Chipyard/Rocket Chip-based one lives here: `sim/src/main/makefrag/firesim/Makefrag`

Here the main order of business is to add header and source files to `DRIVER_H` and `DRIVER_CC` respectively, by modifying the lines below:

```
##########################
# Driver Sources & Flags #
##########################

# dromajo modifications
DROMAJO_DIR = $(chipyard_dir)/tools/dromajo/dromajo-src/src

DROMAJO_LIB_NAME = dromajo_cosim
DROMAJO_LIB = $(DROMAJO_DIR)/lib$(DROMAJO_LIB_NAME).a

DROMAJO_H = $(GENERATED_DIR)/dromajo_params.h
DROMAJO_LONG_H = $(GENERATED_DIR)/$(long_name).dromajo_params.h

TESTCHIPIP_CSRC_DIR = $(chipyard_dir)/generators/testchipip/src/main/resources/
→testchipip/csrc

CHIPYARD_ROM = $(chipyard_dir)/generators/testchipip/bootrom/bootrom.rv64.img
DROMAJO_ROM = $(GENERATED_DIR)/$(long_name).rom

DTS_FILE = $(GENERATED_DIR)/$(long_name).dts
```

(continues on next page)

```makefile
DROMAJO_DTB = $(GENERATED_DIR)/$(long_name).dtb


$(DROMAJO_LIB):
        $(MAKE) -C $(DROMAJO_DIR)


$(DROMAJO_LONG_H) $(DTS_FILE): $(simulator_verilog)


$(DROMAJO_H): $(DROMAJO_LONG_H)
        rm -rf $(DROMAJO_H)
        ln -s $(DROMAJO_LONG_H) $(DROMAJO_H)


$(DROMAJO_DTB): $(DTS_FILE)
        dtc -I dts -O dtb -o $(DROMAJO_DTB) $(DTS_FILE)


$(DROMAJO_ROM): $(CHIPYARD_ROM)
        rm -rf $(DROMAJO_ROM)
        ln -s $(CHIPYARD_ROM) $(DROMAJO_ROM)


DROMAJO_REQS = $(DROMAJO_H) $(DROMAJO_ROM) $(DROMAJO_DTB)


firesim_lib_dir = $(firesim_base_dir)/firesim-lib/src/main/cc
driver_dir = $(firesim_base_dir)/src/main/cc
DRIVER_H = $(shell find $(driver_dir) -name "*.h") \
        $(shell find $(firesim_lib_dir) -name "*.h") \
        $(DROMAJO_REQS) \
        $(TESTCHIPIP_CSRC_DIR)/testchip_tsi.h


DRIVER_CC = $(wildcard $(addprefix $(driver_dir)/, $(addsuffix .cc, firesim/*))) \
            $(wildcard $(addprefix $(firesim_lib_dir)/, $(addsuffix .cc, bridges/*␣
→fesvr/* bridges/tracerv/*)))  \
            $(RISCV)/lib/libfesvr.a \
            $(DROMAJO_LIB) \
            $(TESTCHIPIP_CSRC_DIR)/testchip_tsi.cc


# Per-repository dependencies (e.g., libdwarf) are installed under this sysroot
firesim_local_sysroot = $(firesim_base_dir)/lib-install

TARGET_CXX_FLAGS += -g -I$(TESTCHIPIP_CSRC_DIR) -I$(firesim_lib_dir) -I$(firesim_
→local_sysroot)/include -I$(driver_dir)/firesim -I$(RISCV)/include -I$(DROMAJO_DIR) -
→I$(GENERATED_DIR)
TARGET_LD_FLAGS += -L$(RISCV)/lib -L$(firesim_local_sysroot)/lib -l:libdwarf.so -
→l:libelf.so -lz -L$(DROMAJO_DIR) -l$(DROMAJO_LIB_NAME)
```

That's it! At this point you should be able to both test your bridge in software simulation using meta-simulation, or deploy it to an FPGA.

16.1. UART Bridge (Host-MMIO)

Simulation Triggers

It is often useful to globally coordinate debug and instrumentation features using specific target-events that may be distributed across the target design. For instance, you may wish to enable collection of synthesized prints and sampling of AutoCounters simulataenously when a specific instruction is committed on any core, or alternatively if the memory system sees a write to a particular memory address. Golden Gate's trigger system enables this by aggregating annotated `TriggerSources` distributed throughout the design using a centralized credit-based system which then drives a single-bit level-sensitive enable to all `TriggerSinks` distributed throughout the design. This enable signal is asserted while the design remains in the region-of-interest (ROI). Sources signal the start of the ROI by granting a credit and signal the end of the ROI by asserting a debit. Since there can be multiple sources, each of which might grant credits, the trigger is only disabled when the system has been debited as exactly as many times as it has been credited (it has a balance of 0).

17.1 Quick-Start Guide

17.1.1 Level-Sensitive Trigger Source

Assert the trigger while some boolean `enable` is true.

```
import midas.targetutils.TriggerSource
TriggerSource.levelSensitiveEnable(enable)
```

Caveats:

- The trigger visible at the sink is delayed. See *Trigger Timing*.

- Assumes this is the only source; the trigger is only cleared if no additional credit has been granted.

17.1.2 Distributed, Edge-Sensitive Trigger Source

Assert trigger enable when some boolean *start* undergoes a positive transition, and clear the trigger when a second signal *stop* undergoes a positive transition.

```
// Some arbitarily logic to drive the credit source and sink. Replace with your own!
val start = lfsr(1)
val stop = ShiftRegister(lfsr(0), 5)

// Now annotate the signals.
import midas.targetutils.TriggerSource
TriggerSource.credit(start)
TriggerSource.debit(stop)
// Note one could alternatively write: TriggerSource(start, stop)
```

Caveats:

- The trigger visible at the sink is delayed. See *Trigger Timing*.

- Assumes these are the only sources; the trigger is only cleared if no additional credit has been granted.

17.2 Chisel API

Trigger sources and sinks are Boolean signals, synchronous to a particular clock domain, that have been annotated as such. The `midas.targetutils` package provides chisel-facing utilities for annotating these signals in your design. We describe these utilities below, the source for which can be found in `sim/midas/targetutils/src/main/scala/annotations.scala`.

17.2.1 Trigger Sources

In order to permit distributing trigger sources across the whole design, you must annotate distinct boolean signals as credits and debits using methods provided by the `TriggerSource` object. We provide an example below (the distributed example from the quick-start guide).

```
// Some arbitarily logic to drive the credit source and sink. Replace with your own!
val start = lfsr(1)
val stop = ShiftRegister(lfsr(0), 5)

// Now annotate the signals.
import midas.targetutils.TriggerSource
TriggerSource.credit(start)
TriggerSource.debit(stop)
// Note one could alternatively write: TriggerSource(start, stop)
```

Using the methods above, credits and debits issued while the design is under reset are not counted (the reset used is implicit reset of the Chisel Module in which you invoked the method). If the module provides no implicit reset or if you wish to credit or debit the trigger system while the local module's implicit reset is asserted, use `TriggerSource.{creditEvenUnderReset, debitEvenUnderReset}` instead.

17.2.2 Trigger Sinks

Like sources, trigger sinks are boolean signals that have been annotated alongside their associated clock. These signals will be driven by a Boolean value created by the trigger system. If trigger sources exist in your design, the generated trigger will **override all assignments made in the chisel to the same signal**, otherwise, it will take on a default value provided by the user. We provide an example of annotating a sink using the the `TriggerSink` object below.

```
// Note: this can be any reference you wish to have driven by the trigger.
val sinkBool = WireDefault(true.B)

import midas.targetutils.TriggerSink
// Drives true.B if no TriggerSource credits exist in the design.
// Note: noSourceDefault defaults to true.B if unset, and can be omitted for brevity
TriggerSink(sinkBool, noSourceDefault = true.B)
```

Alternatively, if you wish to use a trigger sink as a predicate for a Chisel `when` block, you may use `TriggerSink.whenEnabled` instead

```
/** A simpler means for predicating stateful updates, printfs, and assertions.
 *   Sugar for:
 *    val sinkEnable = Wire(Bool())
 *    TriggerSink(sinkEnable, false.B)
 *    when (sinkEnable) { <...> }
 */
TriggerSink.whenEnabled(noSourceDefault = false.B) {
  printf(SynthesizePrintf(s"${printfPrefix}CYCLE: %d\n", cycle))
}
```

17.3 Trigger Timing

Golden Gate implements the trigger system by generating a target circuit that synchronizes all credit and debits into the base clock domain using a *single* register stage, before doing a global accounting. If the total number of credits exceeds debits the trigger is asserted. This trigger is then synchronized in each sink domain using a single register stage before driving the annotated sink. The circuit that implements this functionality is depicted below:

Fig. 1: Trigger generation circuit. Not shown: a sub-circuit analagous to that which totalCredit' is replicated to count debits. Similarly, the sub-circuit feeding the add-reduction is generated for each clock domain that contains at least one source annotation.

Given the present implementation, an enabled trigger becomes visible in a sink domain no sooner than one base-clock edge and one local-clock edge have elapsed, in that order, after the credit was asserted. This is depicted in the waveform below.

Fig. 2: Trigger timing diagram.

Note that trigger sources and sinks that reside in the base clock domain still have the additional synchronization registers even though they are uneeded. Thus, a credit issued by a source in the base clock domain will be visible to a sink also in the base clock domain exactly 2 cycles after it was issued.

Bridges that use the default `HostPort` interface add an additional cycle of latency in the bridge's local domain since their token channels model a single register stage to improve simulation FMR. Thus, without using a different `HostPort` implementation, trigger sources generated by a Bridge and trigger sinks that feed into a Bridge will each see one additional bridge-local cycle of latency. In constrast, synthesized printfs and assertions, and AutoCounters all use wire channels (since they are unidirectional interfaces, the extra register stage is not required to improve FMR) and will see no additional sink latency.

17.4 Limitations & Pitfalls

- The system is limited to no more than one trigger signal. Presently, there is no means to generate unique triggers for distinct sets of sinks.

- Take care to never issue more debits than credits, as this may falsely enable the trigger under the current implementation.

CHAPTER 18

Optimizing FPGA Resource Utilization

One advantage of a host-decoupled simulator is the ability to spread expensive operations out over multiple FPGA cycles while maintaining perfect cycle accuracy. When employing this strategy, a simulator can rely on a resource-efficient implementation that takes multiple cycles to complete the underlying computation to determine the next state of the target design. In the abstract, this corresponds with the simulator having *less* parallelism in its host implementation than the target design. While this strategy is intrinsic to the design of the compilers that map RTL circuits to software simulators executing on sequential, general-purpose hardware, it is less prevalent in FPGA simulation. These *space-time* tradeoffs are mostly restricted to hand-written, architecture-specific academic simulators or to implementing highly specific host features like I/O cuts in a partitioned, multi-FPGA environment.

With the Golden Gate compiler, we provide a framework for automating these optimization, as discussed in the 2019 ICCAD paper on the design of Golden gate. Furthermore, current versions of FireSim include two optional optimizations that can radically reduce resource utilization (and therefore simulate much large SoCs). The first optimization reduces the overhead of memories that are extremely to implement via direct RTL translation on an FPGA host, including multi-ported register files, while the second applies to repeated instances of large blocks in the target design by *threading* the work associated with simulating multiple instances across a single underlying host implementation.

18.1 Multi-Ported Memory Optimization

ASIC multi-ported RAMs are a classic culprit of poor resource utilization in FPGA prototypes, as they cannot be trivially implemented with Block RAMs (BRAMs) and are instead decomposed into lookup tables (LUTs), multiplexers and registers. While using double-pumping, BRAM duplication, or FPGA-optimized microarchitectures can help, Golden Gate can automatically extract such memories and replace them with a decoupled model that simulates the RAM via serialized accesses to an underlying implementation that is amenable mapping to an efficiency Block RAM (BRAM). While this serialization comes at the cost of reduced emulation speed, the resulting simulator can fit larger SoCs onto existing FPGAs. Furthermore, the decoupling framework of Golden Gate ensures that the simulator will still produce bit-identical, cycle-accurate results.

While the details of this optimization are discussed at length in the ICCAD paper, it is relatively simple to deploy. First, the desired memories must be annotated via Chisel annotations to indicate that they should be optimized; for Rocket- and BOOM-based systems, these annotations are already provided for the cores' register files, which are the most FPGA-hostile memories in the designs. Next, with these annotations in place, enabling the optimization requires mixing in the `MCRams` class to the platform configuration, as shown in the following example build recipe:

```
[firesim-boom-mem-opt]
DESIGN=FireSim
TARGET_CONFIG=WithNIC_DDR3FRFCFSLLC4MB_FireSimLargeBoomConfig
PLATFORM_CONFIG=MCRams_BaseF1Config
instancetype=z1d.2xlarge
deploytriplet=None
```

18.2 Multi-Threading of Repeated Instances

While optimizing FPGA-hostile memories can allow up to 50% higher core counts on the AWS-hosted VU9P FPGAs, significantly larger gains can be had by threading repeated instances in the target system. The *model multi-threading* optimization extracts these repeated instances and simulates each instance with a separate thread of execution on a shared underlying physical implementation.

As with the memory optimization, this requires the desired set of instances to be annotated in the target design. However, since the largest effective FPGA capacity increases for typical Rocket Chip targets are realized by threading the tiles that each contain a core complex, these instances are pre-annotated for both Rocket- and BOOM-based systems. To enable this tile multi-threading, it is necessary to mix in the `MTModels` class to the platform configuration, as shown in the following example build recipe:

```
[firesim-threaded-cores-opt]
DESIGN=FireSim
TARGET_CONFIG=WithNIC_DDR3FRFCFSLLC4MB_FireSimQuadRocketConfig
PLATFORM_CONFIG=MTModels_BaseF1Config
instancetype=z1d.2xlarge
deploytriplet=None
```

This simulator configuration will rely on a single threaded model to simulate the four Rocket tiles. However, it will still produce bit- and cycle-identical results to any other platform configuration simulating the same target system.

In practice, the largest benefits will be realized by applying both the `MCRams` and `MCModels` optimizations to large, multi-core BOOM-based systems. While these simulator platforms will have reduced throughput relative to unoptimized FireSim simulators, very large SoCs that would otherwise never fit on a single FPGA can be simulated without the cost and performance drawbacks of partitioning.

```
[firesim-optimized-big-soc]
DESIGN=FireSim
TARGET_CONFIG=MyMultiCoreBoomConfig
PLATFORM_CONFIG=MTModels_MCRams_BaseF1Config
instancetype=z1d.2xlarge
deploytriplet=None
```

Output Files

Golden Gate generates many output files, we describe them here. Note, the GG CML-argument `--output-filename-base=<BASE>` defines defines a common prefix for all output files.

19.1 Core Files

These are used in nearly all flows.

- **<BASE>.sv**: The verilog implementation of the simulator which will be synthesized onto the FPGA. The top-level is the Shim module specified in the `PLATFORM_CONFIG`.

- **<BASE>.const.h**: A target-specific header containing all necessary metadata to instantiate bridge drivers. This is linked into the simulator driver and meta-simulators (FPGA-level / MIDAS-level). Often referred to as "the header".

- **<BASE>.runtime.conf**: Default plus args for generated FASED memory timing models. Most other bridges have their defaults baked into the driver.

19.2 FPGA Synthesis Files

These are additional files passed to the FPGA build directory.

- **<BASE>.defines.vh**: Verilog macro definitions for FPGA synthesis.

- **<BASE>.env.tcl**: Used a means to inject arbitrary TCL into the start of the build flow. Controls synthesis and implementation strategies, and sets the host_clock frequency before the clock generator (MCMM) is synthesized.

- **<BASE>.ila_insert_vivado.tcl**: Synthesizes an ILA for the design. See *AutoILA: Simple Integrated Logic Analyzer (ILA) Insertion* for more details about using ILAs in FireSim.

- **<BASE>.ila_insert_{inst, ports, wires}.v**: Instantiated in the FPGA project via `` `include `` directives to instantiate the generated ILA.

19.3 Meta-simulation Files

These are additional sources used only in MIDAS-level simulators

- **<BASE>.const.vh**: Verilog macros to define variable width fields.

CHAPTER 20

Indices and tables

- genindex
- modindex
- search