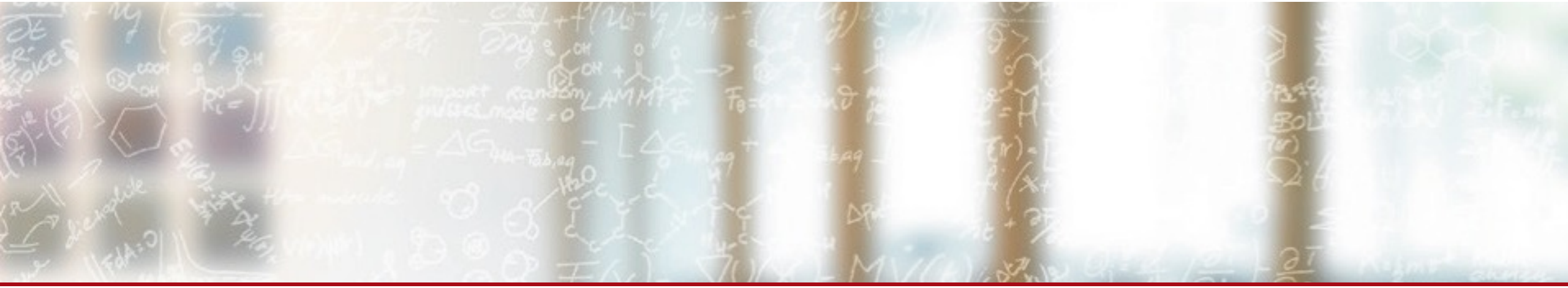
CSCS
Centro Svizzero di Calcolo Scientifico
Swiss National Supercomputing Centre

ETH zürich



ReFrame – Efficient System and Application Performance Testing

CSCS Webinar

Vasileios Karakasis, CSCS

August 29, 2022

Why should I bother?

- There is more to testing beyond unit testing
  - Integration testing: how does my application behave as a whole given different scenarios?
  - Validation testing: are the scientific results produced correct?
  - Regression testing: did my application experience a functionality or performance regression?
  - Performance testing: how does my application perform on various systems?
  - Scaling testing: how does my application's performance scale?
  - Performance exploration: what is the best configuration to run my application on system X?

The “good” old shell script solution!

- Several very good frameworks exist for unit testing based on the programming language of choice, but...
  - for all other types of testing the landscape is poor.

Two major, but non-ideal, solutions are usually employed:

1. Write customized shell scripts.
2. Extend the unit test framework’s functionality to perform tasks it was not designed for.

Non-portable solutions that don’t scale and have high maintenance costs.

Can we do better? – ReFrame

ReFrame is a powerful framework designed from ground up for writing portable validation, regression and performance tests running from laptops to Top500 supercomputers.

- Implements a Python eDSL allowing to write tests in a high-level declarative way
- Tests are composable by design and can be extended or reused across sites
- Multi-dimensional test parameterisation
- Efficient resource sharing and dependencies through test fixtures
- Auto-detection of processor topology and microarchitecture both locally and remotely
- Parallel execution of tests
- Support for native and containerised runs
- Performance logging through multiple channels
- Seamless integration with Gitlab CI/CD pipelines

ReFrame in numbers – A growing community

- Designed and developed by a small team in CSCS back in 2016 to replace a Bash-script based regression testing framework and test suite
- First public release in May 2017 on Github
  - https://github.com/reframe-hpc/reframe
  - 42 contributors, 78 forks, 156 stars
  - 70 releases
- Used by both academic institutions and industry around the world for testing and benchmarking their clusters
    - CSCS tests are publicly available at https://github.com/eth-cscs/cscs-reframe-tests
- 192 members in the project's Slack workspace:
  - https://reframe-slack.herokuapp.com/
  - Join for updates and support
- Community calls (check the `#confcalls` Slack channel)

A simple validation test

```python
import reframe as rfm
import reframe.utility.sanity as sn


@rfm.simple_test
class stream_test(rfm.RegressionTest):
    valid_systems = ['*']
    valid_prog_environs = ['*']
    build_system = 'SingleSource'
    sourcepath = 'stream.c'

@sanity_function
def validate_solution(self):
    return sn.assert_found(r'Solution Validates', self.stdout)
```

Tests are specially decorated Python classes

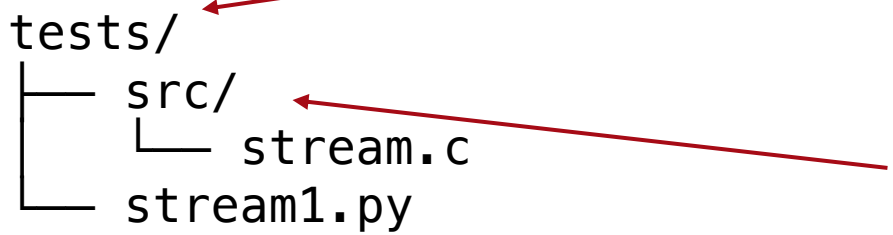
System and environment specs that this test is valid for

Build instructions

Test validation

- *Examples: https://github.com/vkarak/reframe/tree/doc/cscs-webinar-2022/tutorials/cscs-webinar-2022*
- *Demo run: https://asciinema.org/a/517693*

Test directory structure



Test top-level directory:

- Can contain multiple files

Default resource directory for tests at the same level

- Can be changed from inside the test

The test file

- Can define multiple tests

ReFrame is *NOT* a shell script generator

- It only generates **minimal** and **strictly necessary** shell scripts
  - They simply contain user code, job scheduler and environment setup as well as some ReFrame boilerplate, if applicable.
  - There is no test logic embedded in those scripts!

output/generic/default/builtin/stream_test/rfm_stream_test_build.sh

```
#!/bin/bash

_onerror()
{
    exitcode=$?
    echo "-reframe: command \`$BASH_COMMAND' failed (exit code: $exitcode)"
    exit $exitcode
}

trap _onerror ERR

cc stream.c -o ./stream_test
```

output/.../rfm_stream_test_job.sh

```
#!/bin/bash
./stream_test
```

Converting to a performance test

```
import reframe as rfm
import reframe.utility.sanity as sn


@rfm.simple_test
class stream_test(rfm.RegressionTest):
    valid_systems = ['*']
    valid_prog_environs = ['*']
    build_system = 'SingleSource'
    sourcepath = 'stream.c'

@sanity_function
def validate_solution(self):
    return sn.assert_found(r'Solution Validates', self.stdout)
```

```
@performance_function('MB/s')
def copy_bandwidth(self):
    return sn.extractsingle(r'Copy:\s+(\S+)\s+.*', self.stdout, 1, float)
```

Specially decorated function that extracts and converts a performance metric

- The name of the function is the name of the metric

ReFrame configuration

- ReFrame's configuration contains primarily a list of system and enviroment definitions
  - By default, a single generic system and environment are defined that allow ReFrame to run anywhere (locally).
  - Users extend this configuration with the systems and environments that need to be tested.
- Several other aspects of the framework's behaviour can be controlled through its configuration, through environment variables or through command-line options.

Adding systems and environments for testing

```
site_configuration = {
  'systems': [
    {
      'name': 'tresa',
      'descr': 'My laptop',
      'hostnames': ['tresa\.local'],
      'partitions': [
        {
          'name': 'default',
          'scheduler': 'local',
          'launcher': 'local',
          'environs': ['gnu', 'clang']
        }
      ]
    }
  ],
  ...
```

Name and description of the system

Hostname patterns to identify this system

System partitions

Job scheduler to use

Parallel launcher to use

Environments to test on this partition

Environment name

```
'environments': [
  {
    'name': 'gnu',
    'cc': 'gcc-9',
    'cxx': 'g++-9',
    'ftn': '',
    'features': ['openmp'],
    'extras': {'ompflag': '-fopenmp'}
  },
  {
    'name': 'clang',
    'cc': 'clang',
    'cxx': 'clang++',
    'ftn': ''
  },
  ...
]
```

Compilers

User defined features and properties

Customising further a test

```
@rfm.simple_test
class stream_test(rfm.RegressionTest):
    valid_systems = ['*']
    valid_prog_environs = ['+openmp']
    build_system = 'SingleSource'
    sourcepath = 'stream.c'
    reference = {
        'tresa': {
            'copy_bandwidth': (23000, -0.05, None, 'MB/s')
        }
    }
```

Test is valid only for environments that define the openmp feature

Performance references

System name

Performance metric

Baseline

Lower Threshold (*-5%*)

Upper Threshold (*none*)

Customising further a test (cont'd)

```python
@rfm.simple_test
class stream_test(rfm.RegressionTest):
    ...
    build_system = 'SingleSource'

    @run_before('compile')
    def setup_build(self):
        try:
            omp_flag = self.current_environ.extras['ompflag']
        except KeyError:
            envname = self.current_environ.name
            self.skip(f'"ompflag" not defined for environment {envname!r} ')

        self.build_system.cflags = [omp_flag, '-O3']
        self.build_system.cppflags = [f'-DSTREAM_ARRAY_SIZE={1 << 25}']

    @run_before('run')
    def setup_omp_env(self):
        procinfo = self.current_partition.processor
        self.num_cpus_per_task = procinfo.num_cores
        self.variables = {
            'OMP_NUM_THREADS': str(self.num_cpus_per_task),
            'OMP_PLACES': 'cores'
        }

    ...
```

Attach an arbitrary function to the pipeline

Get an environment property

Skip the test if property is not found

Customise the build

Processor topology information

- ReFrame automatically detects it even for remote partitions

Customise test's environment

The test pipeline and the framework's runtime

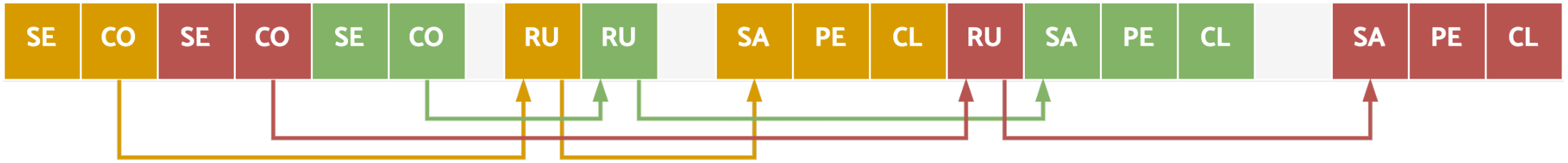
- Each test in ReFrame executes a series of well-defined stages that execute atomically and in order.
  - Tests can attach arbitrary functions for execution before or after each pipeline stage through the `@run_before` and `@run_after` builtin decorators.
- The runtime executes tests concurrently by interleaving the execution of the stages of multiple tests.
  - Concurrency limits can be set by the user.
- The runtime honors test dependencies and will not schedule a test for execution until all of its dependencies have finished successfully.

https://reframe-hpc.readthedocs.io/en/stable/pipeline.html

The test pipeline and the framework's runtime (cont'd)

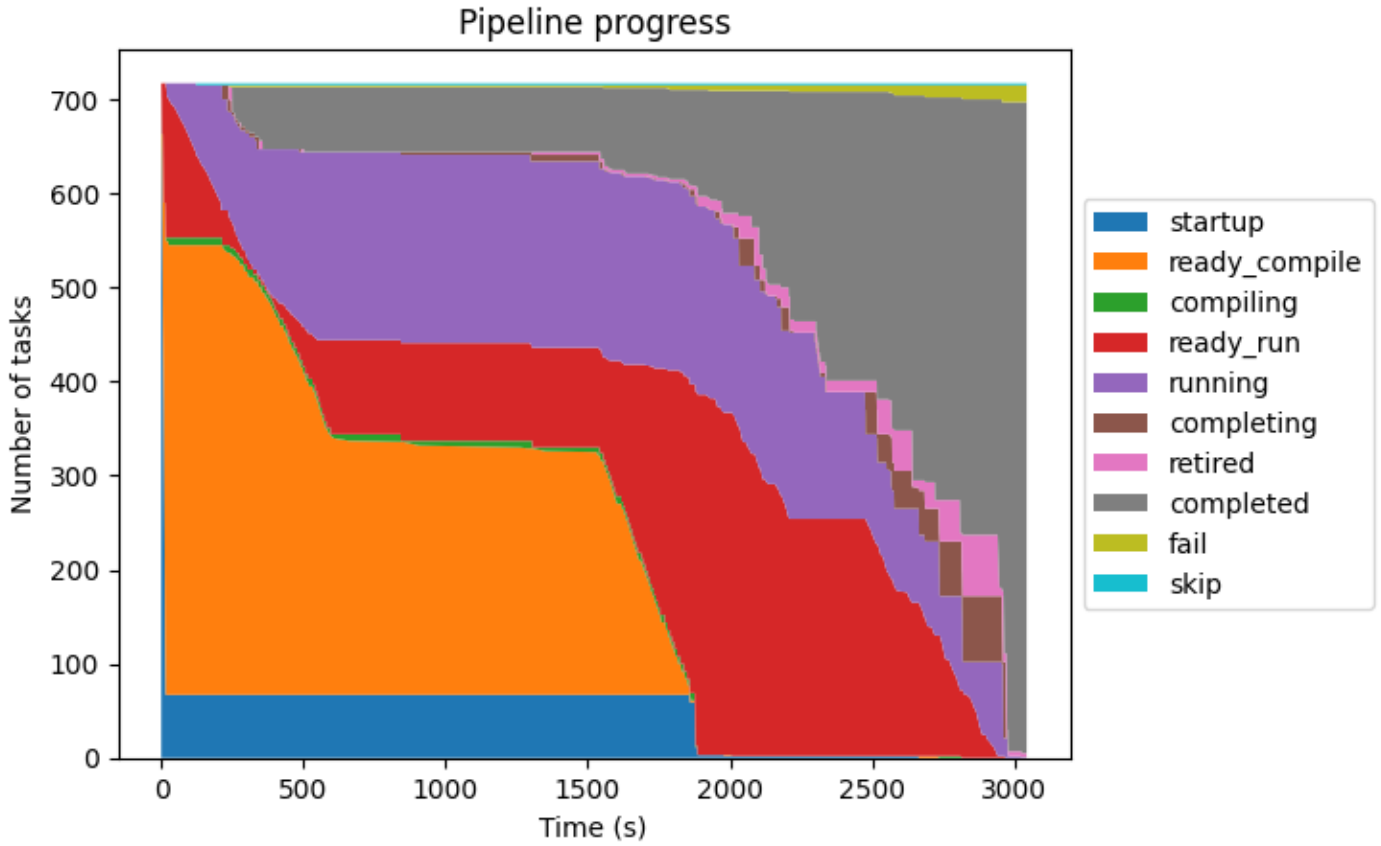
Pipeline stages

Serial Execution Policy (`--exec-policy=serial`)



Asynchronous Execution Policy (`--exec-policy=async`) (the default)

The test pipeline and the framework's runtime (cont'd)



- Progress of the full production test suite on Piz Daint
  - Two remote partitions with a limit of 100 concurrent jobs.
  - Up to 8 local builds.

Defining test variables

- Enforce type checking on assignments
- Can be set directly from the command-line
- All built-in `RegressionTest` fields are variables

```
@rfm.simple_test
class stream_test(rfm.RegressionTest):
    ...
    array_size = variable(int, value=(1 << 25))
    num_iters = variable(int, value=10)

    @run_before('compile')
    def setup_build(self):
        ...
        self.build_system.cflags = [omp_flag, '-O3']
        self.build_system.cppflags = [f'-DSTREAM_ARRAY_SIZE={self.array_size}',
                                      f'-DNTIMES={self.num_iters}']
```

Default value

Variable type

Variables are accessed as standard object attributes

Can be run as: **`reframe [...] -S stream_test.num_iters=100`**

Defining test parameters

- Generate different unique test variants for each parameter value
- Each parameter expands the parameterisation space with a new dimension
- Parameters can be inherited and then filtered or effectively suppressed

```
@rfm.simple_test
class stream_test(rfm.RegressionTest):
    ...
    elem_type = parameter(['double', 'float'])


    @run_before('compile')
    def setup_build(self):
        ...
        self.build_system.cflags = [omp_flag, '-O3']
        self.build_system.cppflags = [f'-DSTREAM_ARRAY_SIZE={self.array_size}',
                                      f'-DNTIMES={self.num_iters}',
                                      f'-DSTREAM_TYPE={self.elem_type}']
```

Parameter values

```
[List of matched checks]
- stream_test %elem_type=float
- stream_test %elem_type=double
Found 2 check(s)
```

The parameter value for this test variant

Using test fixtures

- A fixture in ReFrame is a dependent test that manages a resource of the test that uses it.
- Fixtures have scopes
  - **Session**: The fixture will be executed once for the whole run session on any valid system partition or environment.
  - **Partition**: The fixture will be executed once per system partition on any valid environment.
  - **Environment**: The fixture will be executed once per system partition and environment.
  - **Test**: The fixture is private to the test and it will be executed every time the test is executed.
- Fixtures can be parametrised and they will consequently parametrise the tests that use them.

Using fixtures

```
class stream_build(rfm.CompileOnlyRegressionTest):
    build_system = 'SingleSource'
    sourcepath = 'stream.c'
    array_size = variable(int, value=(1 << 25))
    num_iters = variable(int, value=10)
    elem_type = parameter(['double', 'float'])
    executable = 'stream'

    @run_before('compile')
    def setup_build(self):
        ...

@rfm.simple_test
class stream_test(rfm.RunOnlyRegressionTest):
    stream_binaries = fixture(stream_build, scope='environm
    valid_systems = ['*']
    valid_prog_environs = ['+openmp']
    ...

    @run_before('run')
    def setup_omp_env(self):
        self.executable = os.path.join(self.stream_binaries.stagedir, 'stream')
```

Test is split in two parts:

- A compile-only test that produces the binaries
- A run-only that runs the benchmark and uses the first one as a fixture

Useful approach to avoid redundant compilations when parameterisation of the run phase is needed

```
[List of matched checks]
- stream_test %stream_binaries.elem_type=float
    ^stream_build %elem_type=float ~tresa:default+gnu
- stream_test %stream_binaries.elem_type=double
    ^stream_build %elem_type=double ~tresa:default+gnu
Found 2 check(s)
```

The class na

The final test instance is bound to the fixture variable.

Writing a scaling test

We inherit from the previous test and we add a new parameter

```
@rfm.simple_test
class stream_scale_test(stream_test):
    num_threads = parameter([1, 2, 4, 8, 16, 32])
    reference = {}

    @run_before('run')
    def set_cpus_per_task(self):
        self.num_cpus_per_task = self.num_threads
        self.variables['OMP_NUM_THREADS'] = str(self.num_cpus_per_task)

    @run_after('setup')
    def skip_if_too_large(self):
        procinfo = self.current_partition.processor
        self.skip_if(self.num_threads > procinfo.num_cores, 'not enough cores')
```

Set up the run based on the parameter value

Skip the test if the requested number of threads exceeds the current processor's cores

The downside of this approach is that we hardcode the `num_threads` values...

Writing a scaling test (in a flexible way)

```
import reframe.core.runtime as rt
...
def threads_per_part():
    for p in rt.runtime().system.partitions:
        nthr = 1
        while nthr < p.processor.num_cores:
            yield (p.fullname, nthr)
            nthr <<= 1

    yield (p.fullname, p.processor.num_cores)


@rfm.simple_test
class stream_scale_test(stream_test):
    threading = parameter(threads_per_part(), fmt=lambda x: x[1])
    reference = {}

    @run_after('init')
    def setup_thread_config(self):
        self.valid_systems = [self.threading[0]]
        self.num_threads = self.threading[1]

    @run_before('run')
    def set_cpus_per_task(self):
        self.num_cpus_per_task = self.num_threads
        self.variables['OMP_NUM_THREADS'] = str(self.num_cpus_per_task)
```

We can access all the current system's partitions by accessing the framework's runtime info.

We generate a list of partition and number of threads combinations up to the max. number of cores of each partition

We can format how parameter values are displayed

`valid_systems` and `valid_prog_environs` can only be set up until the test initialisation phase

Unpack the parameter pack

Porting the tests to another system

- ReFrame allows writing tests in a portable way, so that adding new systems and environments in our config will allow them to run out-of-the-box on a new system.
- But we can always do system-specific tweaking by checking/using the `current_system`, `current_partition` and `current_environ` test attributes.
  - For example, defining reference values

Porting the tests to another system (cont'd)

```
{
  'name': 'daint',
  'descr': 'Piz Daint supercomputer',
  'hostnames': ['daint', 'dom'],
  'modules_system': 'tmod32',
  'partitions': [
    {
      'name': 'login',
      'scheduler': 'local',
      'launcher': 'local',
      'environs': ['gnu', 'cray', 'intel', 'nvidia']
    },
    {
      'name': 'hybrid',
      'scheduler': 'slurm',
      'launcher': 'srun',
      'access': ['-Cgpu', '-Acsstaff'],
      'environs': ['gnu', 'cray', 'intel', 'nvidia']
    },
    {
      'name': 'multicore',
      'scheduler': 'slurm',
      'launcher': 'srun',
      'access': ['-Cmc', '-Acsstaff'],
      'environs': ['gnu', 'cray', 'intel', 'nvidia']
    }
  ]
},
```

This system uses a modules system

How access to this partition is granted

```
{
  'name': 'gnu',
  'modules': ['PrgEnv-gnu'],
  'cc': 'gcc',
  'cxx': 'g++',
  'ftn': 'gfortran',
  'features': ['openmp'],
  'extras': {'ompflag': '-fopenmp'},
  'target_systems': ['daint']
},
{
  'name': 'intel',
  'modules': ['PrgEnv-intel'],
  'cc': 'icc',
  'cxx': 'icpc',
  'ftn': 'ifort',
  'features': ['openmp'],
  'extras': {'ompflag': '-qopenmp'},
  'target_systems': ['daint']
},
{
  'name': 'nvidia',
  'modules': ['PrgEnv-nvidia'],
  'cc': 'nvc',
  'cxx': 'nvc++',
  'ftn': 'nvfortran',
  'features': ['openmp'],
  'extras': {'ompflag': '-mp'},
  'target_systems': ['daint']
},
...
```

Modules that load this environment

Scope this definition to a specific system

Porting the tests to another system (cont'd)

```
karakasv@dom101$ reframe -C config/mysettings.py -c tests/stream9.py -l
```

```
[List of matched checks]
- stream_scale_test %threading=36 %stream_binaries.elem_type=float
    ^stream_build %elem_type=float ~daint:multicore+gnu
    ^stream_build %elem_type=float ~daint:multicore+cray
    ^stream_build %elem_type=float ~daint:multicore+intel
    ^stream_build %elem_type=float ~daint:multicore+nvidia
- stream_scale_test %threading=32 %stream_binaries.elem_type=float
    ^stream_build %elem_type=float ~daint:multicore+gnu
    ^stream_build %elem_type=float ~daint:multicore+cray
    ^stream_build %elem_type=float ~daint:multicore+intel
    ^stream_build %elem_type=float ~daint:multicore+nvidia
[...]
- stream_scale_test %threading=20 %stream_binaries.elem_type=double
    ^stream_build %elem_type=double ~daint:login+gnu
    ^stream_build %elem_type=double ~daint:login+cray
    ^stream_build %elem_type=double ~daint:login+intel
    ^stream_build %elem_type=double ~daint:login+nvidia
- stream_scale_test %threading=16 %stream_binaries.elem_type=double
    ^stream_build %elem_type=double ~daint:login+gnu
    ^stream_build %elem_type=double ~daint:login+cray
    ^stream_build %elem_type=double ~daint:login+intel
    ^stream_build %elem_type=double ~daint:login+nvidia
[...]
Found 38 check(s)
```

ReFrame automatically generated variants for the threading configuration of each partition.

ReFrame generates an environment fixture for each valid environment of every partition.

Test concretisation

- For each test ReFrame generates multiple **test cases**, one for each valid partition and environment: this process is called **test concretisation**.
  - The valid partitions and environs can vary based on the configuration, the constraints set in the test or in the command line (`--system` and `-p` options)
  - Fixtures may be concretised differently depending on their scope.
- ReFrame schedules for execution the generated test cases, **not** the tests.
- Use `-lC` or `--list=C` to view the list of the actual test cases that will be executed.

Test concretisation (cont'd)

```
karakasv@dom101$ reframe -C config/mysettings.py -c tests/stream9.py -lC
```

```
[List of matched checks]
- stream_scale_test %threading=36 %stream_binaries.elem_type=float @daint:multicore+nvidia
    ^stream_build %elem_type=float ~daint:multicore+nvidia @daint:multicore+nvidia
- stream_scale_test %threading=32 %stream_binaries.elem_type=float @daint:multicore+nvidia
    ^stream_build %elem_type=float ~daint:multicore+nvidia @daint:multicore+nvidia
- stream_scale_test %threading=16 %stream_binaries.elem_type=float @daint:multicore+nvidia
    ^stream_build %elem_type=float ~daint:multicore+nvidia @daint:multicore+nvidia
- stream_scale_test %threading=8 %stream_binaries.elem_type=float @daint:multicore+nvidia
    ^stream_build %elem_type=float ~daint:multicore+nvidia @daint:multicore+nvidia
...
- stream_scale_test %threading=12 %stream_binaries.elem_type=double @daint:hybrid+intel
    ^stream_build %elem_type=double ~daint:hybrid+intel @daint:hybrid+intel
- stream_scale_test %threading=12 %stream_binaries.elem_type=double @daint:hybrid+nvidia
    ^stream_build %elem_type=double ~daint:hybrid+nvidia @daint:hybrid+nvidia
- stream_scale_test %threading=8 %stream_binaries.elem_type=double @daint:hybrid+gnu
    ^stream_build %elem_type=double ~daint:hybrid+gnu @daint:hybrid+gnu
...
Concretized 192 test case(s)
```

The only two test classes generated 192 test cases for execution without a single test code change!

Other cool features

- Testing containerised applications
  - By setting `self.container_platform.image` the test will run inside the supplied image.
  - ReFrame emits the right commands for pulling the image and launching the container.
  - The container runtime is configured in the configuration file.
  - The same test can be used to test both the native and containerised version of an application
  - More info: https://reframe-hpc.readthedocs.io/en/stable/tutorial_advanced.html#testing-containerized-applications
- Integration with Gitlab CI
  - The `--ci-generate` option will generate a Gitlab child pipeline where each test will run as a separate CI job; the CI jobs will be properly linked in case of inter-dependent tests.
  - More info: https://reframe-hpc.readthedocs.io/en/stable/tutorial_tips_tricks.html#integrating-into-a-ci-pipeline
- Repeat a set of tests multiple times with `--repeat=N`
- Distribute a set of tests to every single node with `--distribute`

Summary

- ReFrame is a poweful framework for integration, regression and performance testing designed from scratch with the goal of test portability.
- Defines a high-level DSL embedded in Python for writing tests.
- Comes with a runtime for efficient test scheduling and execution.

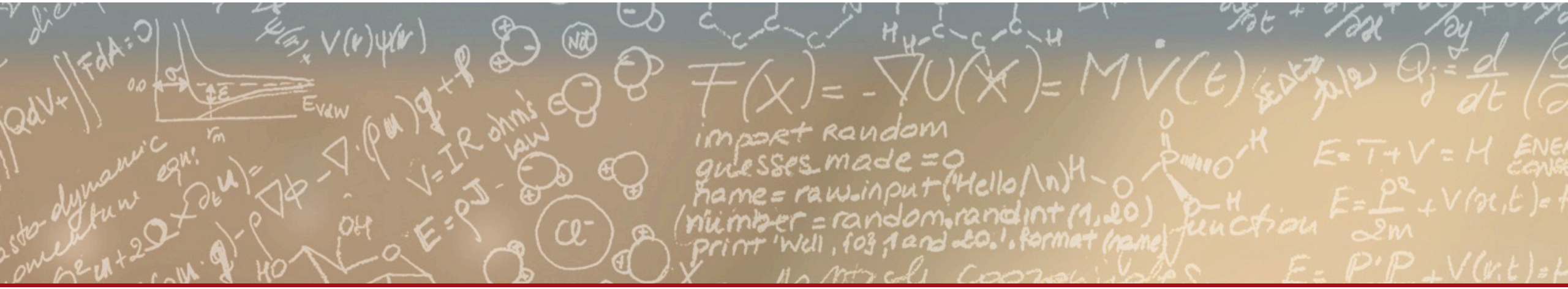
"Continuous validation of software performance at CSCS: How can you contribute?" **@CSCS User Lab Day 2022**


https://reframe-hpc.readthedocs.io


https://github.com/reframe-hpc/reframe


https://reframe-slack.herokuapp.com/


@ReFrameHPC



Thank you for your attention.