CSCS
Centro Svizzero di Calcolo Scientifico
Swiss National Supercomputing Centre

ETH zürich

ReFrame Update

7th EasyBuild User Meeting, 2022

Vasileios Karakasis, CSCS

January 27, 2022

reframe@cscs.ch
https://reframe-hpc.readthedocs.io
https://github.com/eth-cscs/reframe
https://reframe-slack.herokuapp.com
@ReFrameHPC

ReFrame

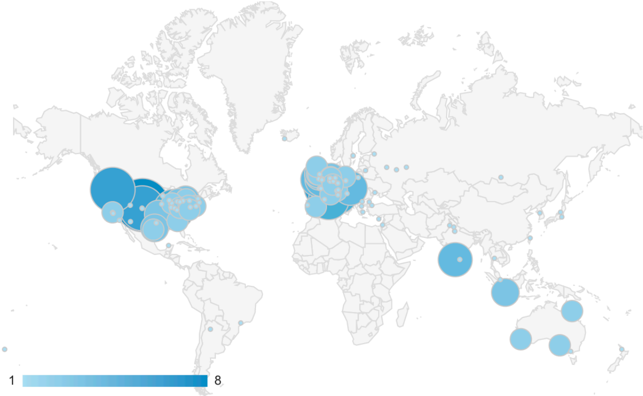
ReFrame is a powerful framework for writing system regression tests and benchmarks, specifically targeted to HPC systems.

- Allows writing high-level HPC tests in Python in a declarative way
- Composable tests allowing the creation of reusable site-agnostic test libraries
- Multi-dimensional test parameterization
- Efficient resource sharing and dependencies through test fixtures
- Efficient runtime that allows executing multiple tests in parallel
- Support for multiple scheduler backends, modules systems, build systems, container runtimes
- Performance logging through multiple channels
- No elevated privileges required

Growing community

- Documentation: https://reframe-hpc.readthedocs.io
- ReFrame Slack workspace (150+ members): https://reframe-slack.herokuapp.com/.
- Bi-weekly community call on Tuesdays @ 16:30 UTC (details on #confcalls Slack channel)
  - https://github.com/eth-cscs/reframe/wiki/Community-calls
  - Next call on Feb. 8
- Don’t hesitate to open bugs and feature requests on Github: https://github.com/eth-cscs/reframe

Growing community



Unique readers of last 30 days (200–300 unique readers monthly)

Progress since last year – Key enhancements

From ReFrame 3.4 (Jan. 26, 2021) to ReFrame 3.10.0 (Jan. 24, 2022)

- v3.5 New syntax for defining test variables and parameters
- v3.5 Generation of dynamic Gitlab pipelines
- v3.6 Generation of JUnit XML reports
- v3.6 Support for skipping tests dynamically
- v3.7 Automatic detection of processor topology
- v3.7 Support for using EasyBuild or Spack for building the test code
- v3.8 Support for setting test variables from the command line

Progress since last year – Key enhancements (cont'd)

v3.9 Support for test fixtures

v3.10 Asynchronous execution of the build phase

v3.10 New naming scheme for tests

- Introduction of test libraries
- New scheduler backends: SGE, OAR, LSF

New syntax elements

Old way of writing tests (still valid; no intention of deprecating it)

```
@simple_test
class my_test(RegressionTest):
    def __init__(self):
        self.valid_systems = ['*']
        self.valid_prog_environs = ['*']
        self.sourcepath = 'hello.c'
        self.sanity_patterns = sn.assert_found(r'Hello, World\!', self.stdout)
```

New way of writing tests

```
@simple_test
class my_test(RegressionTest):
    valid_systems = ['*']
    valid_prog_environs = ['*']
    sourcepath = 'hello.c'

    @sanity_function
    def validate(self):
        return sn.assert_found(r'Hello, World\!', self.stdout)
```

New syntax elements – Performance functions

Tests define performance variables in a more intuitive way and derived tests can easily extend those

```python
@simple_test
class StreamTest(RegressionTest):
    ...

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

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

    @performance_function('MB/s', perf_key='Scale')
    def extract_scale_perf(self):
        return sn.extractsingle(r'Scale:\s+(\S+)\s+.*', self.stdout, 1, float)

    @performance_function('MB/s', perf_key='Add')
    def extract_add_perf(self):
        return sn.extractsingle(r'Add:\s+(\S+)\s+.*', self.stdout, 1, float)

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

New syntax elements – Variables

Variables are type checked, can be inherited, made required or set in the test instance

- Variables are resolved lazily; if a required variable is not used anywhere, the test will execute without a problem
- All predefined test variables are defined with the `variable()` built-in

```
import reframe.utility.typecheck as typ

@simple_test
class base_test(...):
    x = variable(int)    # variable is required
    y = variable(str, value='spam')      # assign default value to y
    z = variable(typ.List[int], value=[1,2]) # Default value is deep copied into the variable

@simple_test
class derived_test(base_test):
    x = 3            # Define x
    z = required     # Make z required

@run_before('run')
def set_z(self):
    if self.current_system.name == 'S' and self.x == 3:
        self.z = 10      # TypeError: not a list of integers!
```

New syntax elements – Parameters

Old way of defining parameterized tests (DEPRECATED)

```
@parameterized_test(*([flags, lang] for link in ('static', 'dynamic') for lang in ('c', 'cpp', 'f90')))
class my_test(RegressionTest):
    def __init__(self, link, lang):
        ...
        self.sourcepath = f'hello.{lang}'
        self.variables = {'CRAY_PE_LINK_TYPE': link}
```

New way of defining parameterized tests

```
@simple_test
class my_test(RegressionTest):
    link = parameter(['static', 'dynamic'])
    lang = parameter(['c', 'cpp', 'f90'])
    ...

    @run_after('init')
    def prepare_compile(self):
        self.sourcepath = f'hello.{self.lang}'
        self.variables = {'CRAY_PE_LINK_TYPE': self.link}
```

New syntax elements – Parameters (cont'd)

Parameters are inherited and derived tests can extend or modify the parameter space.

```
class base_test(RegressionTest):
    p = parameter([1, 2, 3])


@simple_test
class derived_test(base_test):
    '''Generates 30 tests'''
    q = parameter(range(10))    # -> test is doubly parameterized on p, q


@simple_test
class derived2_test(derived_test):
    '''Generates 50 tests'''
    p = parameter(range(10))    # parameter p is overriden
    q = parameter(inherit_params=True,
                  filter_params=lambda values: [x for x in values if x < 5])  # q dimension is pruned
```

Setting test variables from the command line

- `variables` can be set from the command line using the `-S` or `--setvar` options
  - Variables set in hooks take precedence
- Type conversions are handled by the framework automatically (including aggregate and custom types)

```
# Set num_tasks to 10 for all selected tests
reframe -S num_tasks=10 -r

# Scope variable set to a specific test
# -> this will affect all parameterized variants though!
reframe -S my_test.num_tasks=10 -r

# Set environment modules and environment variables
reframe -S modules=spam,eggs -S variables=USE_HAM:y,NUM_EGGS:2 -r

# Override valid_systems and valid_prog_environs
reframe -S valid_systems='*' -S valid_prog_environs='*' -r
```

Test fixtures

Fixtures are normal tests that can be used by other tests in order to manage a resource:

- Fixtures are similar to test dependencies except that they are associated with a scope component.
- Fixtures inherit their `valid_systems` and `valid_prog_environs` from the test that defines them based on their scope.
- As with test dependencies, a test can access its fixture objects and retrieve their state but with a more intuitive syntax compared to test dependencies.

Test fixtures (cont'd)

```
class fetch_osu_benchmarks(RunOnlyRegressionTest):
    version = variable(str, value='5.6.2')
    executable = 'wget'
    executable_opts = [
        f'http://mvapich.cse.ohio-state.edu/download/mvapich/osu-micro-benchmarks-{version}.tar.gz'
    ]
    local = True

    @sanity_function
    def validate_download(self):
        return sn.assert_eq(self.job.exitcode, 0)

class build_osu_benchmarks(CompileOnlyRegressionTest):
    build_system = 'Autotools'
    build_prefix = variable(str)
  ⇒osu_benchmarks = fixture(fetch_osu_benchmarks, scope='session')

    @run_before('compile')
    def prepare_build(self):
        tarball = f'osu-micro-benchmarks-{self.osu_benchmarks.version}.tar.gz'
        self.build_prefix = tarball[:-7]  # remove .tar.gz extension
      ⇒fullpath = os.path.join(self.osu_benchmarks.stagedir, tarball)
        self.prebuild_cmds = [
            f'cp {fullpath} {self.stagedir}',
            f'tar xzf {tarball}',
            f'cd {self.build_prefix}'
        ]
        self.build_system.max_concurrency = 8
```

Test fixtures (cont'd)

```
class osu_bandwidth_test(RunOnlyRegressionTest):
    valid_systems = ['daint:gpu']
    valid_prog_environs = ['gnu', 'pgi', 'intel']
    num_tasks = 2
    num_tasks_per_node = 1
  ⇒osu_binaries = fixture(build_osu_benchmarks, scope='environment')

    @run_before('run')
    def prepare_run(self):
      ⇒self.executable = os.path.join(
            self.osu_binaries.stagedir,
            self.osu_binaries.build_prefix,
            'mpi', 'pt2pt', 'osu_bw'
        )
        self.executable_opts = ['-x', '100', '-i', '1000']

    @sanity_function
    def validate_test(self):
        return sn.assert_found(r'^8', self.stdout)

    @performance_function('us')
    def latency(self):
        return sn.extractsingle(r'^4194304\s+(\S+)', self.stdout, 1, float)
```

Test fixtures (cont'd)

Fixtures can be concretized differently on different systems or different system/partition combinations based on their scope:

```
$ reframe -n osu_bandwidth_test -lC
<...omitted...>
- osu_bandwidth_test @daint:gpu+gnu
    ^build_osu_benchmarks ~daint:gpu+gnu @daint:gpu+gnu
      ^fetch_osu_benchmarks ~daint @daint:gpu+gnu
- osu_bandwidth_test @daint:gpu+intel
    ^build_osu_benchmarks ~daint:gpu+intel @daint:gpu+intel
      ^fetch_osu_benchmarks ~daint @daint:gpu+gnu
- osu_bandwidth_test @daint:gpu+pgi
    ^build_osu_benchmarks ~daint:gpu+pgi @daint:gpu+pgi
      ^fetch_osu_benchmarks ~daint @daint:gpu+gnu
Concretized 7 test case(s)
```

```
$ reframe -n osu_bandwidth_test -lC -p pgi
<...omitted...>
- osu_bandwidth_test @daint:gpu+pgi
    ^build_osu_benchmarks ~daint:gpu+pgi @daint:gpu+pgi
      ^fetch_osu_benchmarks ~daint @daint:gpu+pgi
Concretized 3 test case(s)
```

Dynamic generation of Gitlab child pipelines

New action option `--ci-generate` generates a Gitlab child pipeline file with one job per test

- Test dependencies are mapped to CI job dependencies
- Stage directories are cached between the jobs
- If `image` is defined, it is passed down to the child pipeline

CSCS

ETH zürich

Dynamic generation of Gitlab child pipelines (cont'd)

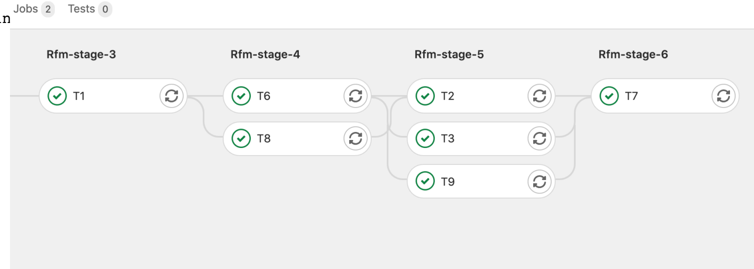
`.gitlab.yml`

```
stages:
  - generate
  - test

generate-pipeline:
  stage: generate
  script:
    - reframe --ci-generate=${CI_PROJECT_DIR}/pipeline.yml -c ${CI_PROJECT_DIR}/path/to/tests
  artifacts:
    paths:
      - ${CI_PROJECT_DIR}/pipeline.yml

test-jobs:
  stage: test
  trigger:
    include:
      - artifact: pipeline.yml
        job: generate-pipeline
    strategy: depend
```

Dynamic generation of Gitlab child pipelines (cont'd)

`.gitlab.yml`

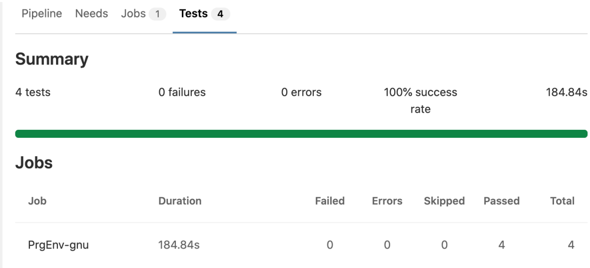
```
stages:
  - generate
  - test

generate-pipeline:
  stage: generate
  script:
    - reframe --ci-generate=${CI_PROJECT_DIR}/pipeline.yml -c ${CI_PROJECT_DIR}/path/to/tests
  artifacts:
    paths:
      - ${CI_PROJECT_DIR}/pipelin

test-jobs:
  stage: test
  trigger:
    include:
      - artifact: pipeline.yml
        job: generate-pipeline
    strategy: depend
```



JUnit XML reports

Use the `--report-junit=report.xml` option to generate a report

- Integrates well with Gitlab and Jenkins for reporting test results
- Does not take into account multiple retries of the same test



Skipping tests dynamically

- `skip(message)`: skip test unconditionally
- `skip_if(cond, message)`: skip test on condition

```
@simple_test
class my_test(RunOnlyRegressionTest):
    x = variable(int, value=10)

    @run_after('init')
    def check_x(self):
        '''Test will not be listed or run if filtered here'''
        self.skip_if(self.x < 3, 'x is lower than 3')

    @run_before('run')
    def check_x2(self):
        '''Test will be skipped during runtime'''
        self.skip_if(self.x < 5, 'x lower than 5')
```

```
[----------] start processing checks
[ RUN      ] my_test @generic:default+builtin
[     SKIP ] (1/1) x is lower than 5
[----------] all spawned checks have finished
```

Automatic detection of processor topology

ReFrame can detect the processor topology both of local or remote partitions and make it available in the tests

- Very useful for benchmarks
- Topology files are cached under
  `$HOME/.reframe/topology/<system>-<partition>/processor.json`
- Set `RFM_REMOTE_DETECT` to detect remote partitions
- Use `skip_if_no_procinfo()` to skip the test if no topology information is available

```
@run_before('run')
def setup_run(self):
    self.skip_if_no_procinfo()
    proc = self.current_partition.processor
    self.num_tasks = self.num_nodes * self.num_tasks_per_node
    self.num_cpus_per_task = proc.num_cores
    self.variables = {
        'OMP_NUM_THREADS': str(self.num_cpus_per_task)
    }
```

Using EasyBuild or Spack to build the test code

This feature was primarily designed for CI/CD workflows, but you can leverage it for integrating s/w stack installation and testing:

- Build system backends for EasyBuild and Spack
- Test code is installed in the test's stage directory
- No side-effects if something fails
- `eb` and `spack` commands are expected to be available on the system

Using EasyBuild or Spack to build the test code (cont'd)

```
@simple_test
class BZip2EBCheck(RegressionTest):
    descr = 'Demo test using EasyBuild to build the test code'
    valid_systems = ['*']
    valid_prog_environs = ['builtin']
    executable = 'bzip2'
    executable_opts = ['--help']
    build_system = 'EasyBuild'

    @run_after('init')
    def setup_build_system(self):
        self.build_system.easyconfigs = ['bzip2-1.0.6.eb']
        self.build_system.options = ['-f']
        # self.build_system.prefix = '/path/to/global/prefix'
        # self.build_system.options = ['--any-options-to-eb-command']

    @run_before('run')
    def prepare_run(self):
        self.modules = self.build_system.generated_modules

    @sanity_function
    def assert_version(self):
        return sn.assert_found(r'Version 1.0.6', self.stderr)
```

Using EasyBuild or Spack to build the test code (cont'd)

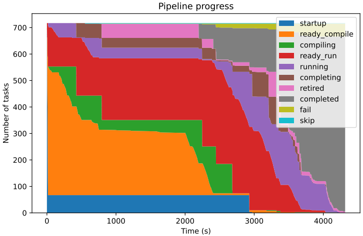
- Spack backend leverages environments to do the local installation
- You can direct the backend to install in a custom path

```
@simple_test
class BZip2SpackCheck(RegressionTest):
    descr = 'Demo test using Spack to build the test code'
    valid_systems = ['*']
    valid_prog_environs = ['builtin']
    executable = 'bzip2'
    executable_opts = ['--help']
    build_system = 'Spack'

    @run_after('init')
    def setup_build_system(self):
        self.build_system.specs = ['bzip2@1.0.6']
        # self.build_system.install_tree = '/path/to/global/spack/install'

    @sanity_function
    def assert_version(self):
        return sn.assert_found(r'Version 1.0.6', self.stderr)
```

Asynchronous execution of the build stage

Asynchronous execution policy executes also the build stage in parallel:

- By default up to 8 local build jobs (configurable)
  - 30-40% higher throughout for workloads with many compilations.
- Passing `-S build_locally=false` from the command line will execute all the build operations on the target partition.



- 2× remote partitions with 100 jobs limit
- 1× local partition with 4 jobs limit

New test naming scheme

- Parameters are no more encoded in the test name
- Parameterized tests and fixtures get a unique compact name which is used in paths
- Tests are displayed in a human readable form encoding parameters and fixture data (scopes, etc.)
- The `-n` option matches the human-readable name, as well as the unique name if needed.
- Enable with `RFM_COMPACT_TEST_NAMES=y` (will become the default in 4.0)

New test naming scheme (cont'd)

```
@simple_test
class amber_nve_check(RunOnlyRegressionTest, pin_prefix=True):
    '''The library test'''
    benchmark_info = parameter([
        ('Cellulose_production_NVE', -443246.0, 5.0E-05),
        ('FactorIX_production_NVE', -234188.0, 1.0E-04),
        ('JAC_production_NVE_4fs', -44810.0, 1.0E-03),
        ('JAC_production_NVE', -58138.0, 5.0E-04)
    ], fmt=lambda x: x[0])
    variant = parameter(['mpi', 'cuda'])
    ...
```

```
@simple_test
class cscs_amber_check(amber_nve_check):
    '''CSCS specialization of the test'''
    num_nodes = parameter([1, 4, 6, 8, 16])
    ...
```

```
$ reframe -n cscs_amber_check -l
...
- cscs_amber_check %benchmark_info=JAC_production_NVE %variant=cuda %num_nodes=4
- cscs_amber_check %benchmark_info=JAC_production_NVE %variant=cuda %num_nodes=1
- cscs_amber_check %benchmark_info=JAC_production_NVE %variant=mpi %num_nodes=16
- cscs_amber_check %benchmark_info=JAC_production_NVE %variant=mpi %num_nodes=8
...
```

Test libraries

The high composability of the ReFrame tests, the processor auto-detection feature and the powerful `-S` option pave the path towards site-agnostic tests

- Library tests do not define `valid_systems` or `valid_prog_environs` or `modules` or `references` and anything that can be site-specific.
- They might define required variables and parameters
- Users can either run directly the library test specifying all the required variables with `-S` from the command line or extend it to write their site-specific version.
- Files under `hpctestlib/` (docs in https://reframe-hpc.readthedocs.io/en/stable/hpctestlib.html)

```
$ reframe -S valid_systems=daint:gpu -S valid_prog_environs=builtin \
  -S modules=Amber -S num_tasks=1 -n 'amber_nve_check.*%variant=cuda' -r
```

Test libraries (cont'd)

Work in test libraries is still in progress

- We need to figure out best practices in writing such library tests
- We probably need generalizations for programming environments for library tests that need to be compiled
- We probably need feature checks to filter out tests without using `valid_systems` or `valid_prog_environs`

Future Outlook

- Continue work on libraries and test generalization, as well as expand the current set of tests
- Support different test run scenarios
  - Fill in a partition with multiple single-node jobs
  - Run repeatedly a set of tests for a certain amount of time
- Make tests container runtime agnostic
- ReFrame 4.0 ?

Thank you for your attention

reframe@cscs.ch

https://reframe-hpc.readthedocs.io

https://github.com/eth-cscs/reframe

https://reframe-slack.herokuapp.com

@ReFrameHPC