LNT Documentation

Release 0.4.2dev

Daniel Dunbar

2017-09-27

Contents

CHAPTER 1

Introduction

LNT is an infrastructure for performance testing. The software itself consists of two main parts, a web application for accessing and visualizing performance data, and command line utilities to allow users to generate and submit test results to the server.

The package was originally written for use in testing LLVM compiler technologies, but is designed to be usable for the performance testing of any software.

If you are an LLVM developer who is mostly interested in just using LNT to run the test-suite against some compiler, then you should fast forward to the *Quickstart Guide* or to the information on *Test Producers*.

LNT uses a simple and extensible format for interchanging data between the test producers and the server; this allows the LNT server to receive and store data for a wide variety of applications.

Both the LNT client and server are written in Python, however the test data itself can be passed in one of several formats, including property lists and JSON. This makes it easy to produce test results from almost any language.

Installation

If you are only interested in using LNT to run tests locally, see the *Quickstart Guide*.

If you want to run an LNT server, you will need to perform the following additional steps:

2. Create a new LNT installation:

   ```
   lnt create path/to/install-dir
   ```

   This will create the LNT configuration file, the default database, and a .wsgi wrapper to create the application. You can execute the generated app directly to run with the builtin web server, or use:

   ```
   lnt runserver path/to/install-dir
   ```

   which provides additional command line options. Neither of these servers is recommended for production use.

3. Edit the generated 'lnt.cfg' file if necessary, for example to:

(a) Update the databases list.

(b) Update the public URL the server is visible at.

(c) Update the nt_emailer configuration.

4. Add the 'lnt.wsgi' app to your Apache configuration. You should set also configure the WSGIDaemonProcess and WSGIProcessGroup variables if not already done.

   If running in a virtualenv you will need to configure that as well; see the modwsgi wiki.

For production servers, you should consider using a full DBMS like PostgreSQL. To create an LNT instance with PostgreSQL backend, you need to do this instead:

1. Create an LNT database in PostgreSQL, also make sure the user has write permission to the database:

```
CREATE DATABASE "lnt.db"
```

2. Then create LNT installation:

```
lnt create path/to/install-dir --db-dir postgresql://user@host
```

3. Run server normally:

```
lnt runserver path/to/install-dir
```

Architecture

The LNT web app is currently implemented as a Flask WSGI web app, with Jinja2 for the templating engine. My hope is to eventually move to a more AJAXy web interface.

The database layer uses SQLAlchemy for its ORM, and is typically backed by SQLite, although I have tested on MySQL on the past, and supporting other databases should be trivial. My plan is to always support SQLite as this allows the possibility of developers easily running their own LNT installation for viewing nightly test results, and to run with whatever DB makes the most sense on the server.

Running a LNT Server Locally

LNT can accommodate many more users in the production config. In production: - Postgres or MySQL should be used as the database. - A proper wsgi server should be used, in front of a proxy like Nginx or Apache.

To install the extra packages for the server config:

```
virtualenv venv
. ./venv/bin/activate
pip install -r requirements.server.txt
python setup.py install --server
lnt create path/to/data_dir --db-dir postgresql://user@host  # data_dir path will be␣
↪where lnt data will go.
cd deployment
# Now edit app_wrapper.py to have your path/to/data_dir path and the log-file below.
gunicorn app_wrapper:app --bind 0.0.0.0:8000 --workers 8 --timeout 300 --name lnt_
↪server --log-file /var/log/lnt/lnt.log --access-logfile /var/log/lnt/gunicorn_
↪access.log --max-requests 250000
```

CHAPTER 2

Quickstart Guide

This quickstart guide is designed for LLVM developers who are primarily interested in using LNT to test compilers using the LLVM test-suite.

Installation

The first thing to do is to checkout install the LNT software itself. The following steps should suffice on any modern Unix variant:

1. Install `virtualenv`, if necessary:

   ```
   sudo easy_install virtualenv
   ```

   `virtualenv` is a standard Python tool for allowing the installation of Python applications into their own sandboxes, or virtual environments.

2. Create a new virtual environment for the LNT application:

   ```
   virtualenv ~/mysandbox
   ```

   This will create a new virtual environment at `~/mysandbox`.

3. Checkout the LNT sources:

   ```
   svn co http://llvm.org/svn/llvm-project/lnt/trunk ~/lnt
   ```

4. Install LNT into the virtual environment:

   ```
   ~/mysandbox/bin/python ~/lnt/setup.py develop
   ```

   We recommend using `develop` instead of install for local use, so that any changes to the LNT sources are immediately propagated to your installation. If you are running a production install or care a lot about stability, you can use `install` which will copy in the sources and you will need to explicitly re-install when you wish to update the LNT application.

That's it!

Running Tests

To execute the LLVM test-suite using LNT you use the `lnt runtest` command. The information below should be enough to get you started, but see the *Test Producers* section for more complete documentation.

1. Checkout the LLVM test-suite, if you haven't already:

```
svn co http://llvm.org/svn/llvm-project/test-suite/trunk ~/llvm-test-suite
```

   You should always keep the test-suite directory itself clean (that is, never do a configure inside your test suite). Make sure not to check it out into the LLVM projects directory, as LLVM's configure/make build will then want to automatically configure it for you.

2. Execute the `lnt runtest nt` test producer, point it at the test suite and the compiler you want to test:

```
lnt runtest nt \
    --sandbox SANDBOX \
    --cc ~/llvm.obj/Release/bin/clang \
    --test-suite ~/llvm-test-suite
```

   The `SANDBOX` value is a path to where the test suite build products and results will be stored (inside a timestamped directory, by default).

3. On most systems, the execution time results will be a bit noisy. There are a range of things you can do to reduce noisiness (with LNT runtest nt command line options when available between brackets):

   - Only build the benchmarks in parallel, but do the actual running of the benchmark code at most one at a time. (`--threads 1 --build-threads 6`). Of course, when you're also interested in the measured compile time, you should also build sequentially. (`--threads 1 --build-threads 1`).
   - When running under linux: Make lnt use linux perf to get more accurate timing for short-running benchmarks (`--use-perf=1`)
   - Pin the running benchmark to a specific core, so the OS doesn't move the benchmark process from core to core. (Under linux: `--make-param="RUNUNDER=taskset -c 1"`)
   - Only run the programs that are marked as a benchmark; some of the tests in the test-suite are not intended to be used as a benchmark. (`--benchmarking-only`)
   - Make sure each program gets run multiple times, so that LNT has a higher chance of recognizing which programs are inherently noisy (`--multisample=5`)
   - Disable frequency scaling / turbo boost. In case of thermal throttling it can skew the results.
   - Disable as many processes or services as possible on the target system.

Viewing Results

By default, `lnt runtest nt` will show the passes and failures after doing a run, but if you are interested in viewing the result data in more detail you should install a local LNT instance to submit the results to.

You can create a local LNT instance with, e.g.:

```
lnt create ~/myperfdb
```

This will create an LNT instance at `~/myperfdb` which includes the configuration of the LNT application and a SQLite database for storing the results.

Once you have a local instance, you can either submit results directly with:

```
lnt import ~/myperfdb SANDBOX/test-<stamp>/report.json
```

or as part of a run with:

```
lnt runtest --submit ~/myperfdb nt ... arguments ...
```

Once you have submitted results into a database, you can run the LNT web UI with:

```
lnt runserver ~/myperfdb
```

which runs the server on `http://localhost:8000` by default.

In the future, LNT will grow a robust set of command line tools to allow investigation of performance results without having to use the web UI.

CHAPTER 3

The `lnt` Tool

The `lnt` command line utility provides the following commands for client-side use and server-side use. The following is a list of commands and the most importat options, use `lnt <toolname> --help` for more information on any particular tool.

Client-Side Tools

`lnt checkformat [<file>]` Checks the syntax of an LNT test report file. In addition to verifying that LNT can read the raw format (e.g., JSON or property list), this also creates a temporary in-memory database instance and ensures that the test report file can be imported correctly.

If run without arguments, this expects to read the input file from `stdin`.

`lnt convert <input path> [<output path>]` Convert between LNT test report formats. By default, this will convert to the property list format. You can use – for either the input (to read from `stdin`) or the output (to write to ``stdout`).

`lnt importreport <input path> [<output path>]` Convert text based key value pairs into a LNT json report file.

`lnt submit <server url> <file>+` Submits one or more files to the given server. The `<server url>` should be the url to the actual `submitRun` page on the server; the database being submitted to is effectively a part of this URL.

`lnt showtests` List available built-in tests. See the *Test Producers* documentation for more details on this tool.

`lnt runtest [<run options>] <test name> ... test arguments ...` Run a built-in test. See the *Test Producers* documentation for more details on this tool.

`lnt check-no-errors <file>+` Check that the report file contains *"no_errors": true* in their run section otherwise end with returncode 1. This is useful for continuous integration scripts which want to report an error if any of the benchmarks didn't compile or run correctly.

Server Administration

The `lnt admin` tool allows connecting to a server through LNTs REST API and perform data queries and modifications. Data modification is only possible with an authentication mechanism specified in the *lntadmin.cfg* file. See *Write Operations* for details.

`lnt admin create-config` Create a *lntadmin.cfg* configuration file in the current directory. The file describes the URL, authentication settings and default database and test-suite settings for an LNT server. The other admin commands read this file if it exists.

`lnt admin list-machines` List machines and their id numbers

`lnt admin machine-info <machine>` Display information about the specified machine.

`lnt admin get-machine <machine>` Download machine information and save data in a json file.

`lnt admin rm-machine <machine>` Removes the specified machine and related runs and samples.

`lnt admin rename-machine <machine> <new-name>` Renames the specified machine.

`lnt admin merge-machine-into <machine> <merge-into-machine>` Move all runs from one machine to another machine and delete the machine.

`lnt admin list-runs <machine>` List all runs for the specified machine.

`lnt admin get-run <run>+` Download the specified runs.

`lnt admin post-run <filename>+` Post the specified report files as a new runs to the server.

`lnt admin rm-run <run>+` Remove the specified runs and related samples.

Server-Side Tools

The following tools are used to interact with an LNT server:

`lnt create <path>` Creates a new LNT server instance. This command has a number of parameters to tweak the generated server, but they can all be modified after the fact in the LNT configuration file.

The default server will have a sqlite3 database named *default*. You can specify to use PostgreSQL using `--db-dir postgresql://user@hostname`.

`lnt import <instance path> <file>+` Import an LNT data file into a database. You can use `--database` to select the database to write to. Note that by default this will also generate report emails if enabled in the configuration, you can use `--no-email` to disable this.

`lnt runserver <instance path>` Start the LNT server using a development WSGI server. Additional options can be used to control the server host and port, as well as useful development features such as automatic reloading.

`lnt updatedb --database <NAME> --testsuite <NAME> <instance path>` Modify the given database and testsuite.

Currently the only supported commands are `--delete-machine` and `--delete-run`.

All commands which take an instance path support passing in either the path to the `lnt.cfg` file, the path to the instance directory, or the path to a (compressed) tarball. The tarball will be automatically unpacked into a temporary directory and removed on exit. This is useful for testing and for passing database instances back and forth, for example when others only need to be able to view the results.

CHAPTER 4

Test Producers

On the client-side, LNT comes with a number of built-in test data producers. This section focuses on the LLVM test-suite (aka nightly test) generator, since it is the primary test run using the LNT infrastructure, but note that LNT also includes tests for other interesting pieces of data, for example Clang compile-time performance.

LNT also makes it easy to add new test data producers and includes examples of custom data importers (e.g., to import buildbot build information into) and dynamic test data generators (e.g., abusing the infrastructure to plot graphs, for example).

Running a Local Server

It is useful to set up a local LNT server to view the results of tests, either for personal use or to preview results before submitting them to a public server. To set up a one-off server for testing:

```
# Create a new installation in /tmp/FOO.
$ lnt create /tmp/FOO
created LNT configuration in '/tmp/FOO'
...

# Run a local LNT server.
$ lnt runserver /tmp/FOO &> /tmp/FOO/runserver.log &
[2] 69694

# Watch the server log.
$ tail -f /tmp/FOO/runserver.log
* Running on http://localhost:8000/
...
```

Running Tests

The built-in tests are designed to be run via the `lnt` tool. The following tools for working with built-in tests are available:

`lnt showtests` List the available tests. Tests are defined with an extensible architecture. FIXME: Point at docs on how to add a new test.

`lnt runtest [<run options>] <test name> ... test arguments ...` Run the named test. The run tool itself accepts a number of options which are common to all tests. The most common option is `--submit=<url>` which specifies the server to submit the results to after testing is complete. See `lnt runtest --help` for more information on the available options.

The remainder of the options are passed to the test tool itself. The options are specific to the test, but well behaved tests should respond to `lnt runtest <test name> --help`. The following section provides specific documentation on the built-in tests.

Built-in Tests

LLVM CMake test-suite

The llvm test-suite can be run with the `test-suite` built-in test.

Running the test-suite via CMake and lit uses a different LNT test:

```
$ rm -rf /tmp/BAR
$ lnt runtest test-suite \
     --sandbox /tmp/BAR \
     --cc ~/llvm.obj.64/Release+Asserts/bin/clang \
     --cxx ~/llvm.obj.64/Release+Asserts/bin/clang++ \
     --use-cmake=/usr/local/bin/cmake \
     --use-lit=~/llvm/utils/lit/lit.py \
     --test-suite ~/llvm-test-suite \
     --cmake-cache Release
```

Since the CMake test-suite uses lit to run the tests and compare their output, LNT needs to know the path to your LLVM lit installation. The test-suite Holds some common common configurations in CMake caches. The `--cmake-cache` flag and the `--cmake-define` flag allow you to change how LNT configures cmake for the test-suite run.

LLVM Makefile test-suite (aka LLVM Nightly Test)

Note: The Makefile test-suite is deprecated. Consider using the cmake based `lnt runtest test-suite` mode instead. It is actively maintained, collects additional metrics such as code size and has extra features such as producing and using PGO data.

The `nt` built-in test runs the LLVM test-suite execution and performance tests, in the "nightly test" configuration. This test allows running many different applications and benchmarks (e.g., SPEC), with various compile options, and in several different configurations (for example, using an LLVM compiler like `clang` or `llvm-gcc`, running under the LLVM JIT compiler using the LLVM `lli` bit-code interpreter, or testing new code generator passes).

The `nt` test requires that the LLVM test-suite repository, a working LLVM compiler, and a LLVM source and build tree are available. Currently, the LLVM build tree is expected to have been built-in the Release+Asserts configuration.

Unlike the prior `NewNightlyTest.pl`, the `nt` tool does not checkout or build any thing, it is expected that users manage their own LLVM source and build trees. Ideally, each of the components should be based on the same LLVM revision (except perhaps the LLVM test-suite), but this is not required.

The test runs the LLVM test-suite builds and execution inside a user specificed sandbox directory. By default, each test run will be done in a timestamped directory inside the sandbox, and the results left around for post-mortem analysis. Currently, the user is responsible for cleaning up these directories to manage disk space.

The tests are always expected to be run using out-of-tree builds – this is a more robust model and allow sharing the same source trees across many test runs. One current limitation is that the LLVM test-suite repository will not function correctly if an in-tree build is done, followed by an out-of-tree build. It is very important that the LLVM test-suite repository be left pristine.

The following command shows an example of running the `nt` test suite on a local build:

```
$ rm -rf /tmp/BAR
$ lnt runtest nt \
     --sandbox /tmp/BAR \
     --cc ~/llvm.obj.64/Release+Asserts/bin/clang \
     --cxx ~/llvm.obj.64/Release+Asserts/bin/clang++ \
     --llvm-src ~/llvm \
     --llvm-obj ~/llvm.obj.64 \
     --test-suite ~/llvm-test-suite \
     TESTER_NAME \
      -j 16
2010-04-17 23:46:40: using nickname: 'TESTER_NAME__clang_DEV__i386'
2010-04-17 23:46:40: creating sandbox: '/tmp/BAR'
2010-04-17 23:46:40: starting test in '/private/tmp/BAR/test-2010-04-17_23-46-40'
2010-04-17 23:46:40: configuring...
2010-04-17 23:46:50: testing...
2010-04-17 23:51:04: loading test data...
2010-04-17 23:51:05: generating report: '/private/tmp/BAR/test-2010-04-17_23-46-40/
↪report.json'
```

The first seven arguments are all required – they specify the sandbox path, the compilers to test, and the paths to the required sources and builds. The `TESTER_NAME` argument is used to derive the name for this tester (in conjunction which some inferred information about the compiler under test). This name is used as a short identifier for the test machine; generally it should be the hostname of the machine or the name of the person who is responsible for the tester. The `-j 16` argument is optional, in this case it specifies that tests should be run in parallel using up to 16 processes.

In this case, we can see from the output that the test created a new sandbox directory, then ran the test in a subdirectory in that sandbox. The test outputs a limited about of summary information as testing is in progress. The full information can be found in .log files within the test build directory (e.g., `configure.log` and `test.log`).

The final test step was to generate a test report inside the test directory. This report can now be submitted directly to an LNT server. For example, if we have a local server running as described earlier, we can run:

```
$ lnt submit http://localhost:8000/submitRun \
    /tmp/BAR/test-2010-04-17_23-46-40/report.json
STATUS: 0

OUTPUT:
IMPORT: /tmp/FOO/lnt_tmp/data-2010-04-17_16-54-35ytpQm_.plist
  LOAD TIME: 0.34s
  IMPORT TIME: 5.23s
ADDED: 1 machines
ADDED: 1 runs
ADDED: 1990 tests
```

```
COMMITTING RESULT: DONE
TOTAL IMPORT TIME: 5.57s
```

and view the results on our local server.

LNT-based NT test modules

In order to support more complicated tests, or tests which are not easily integrated into the more strict SingleSource or MultiSource layout of the LLVM test-suite module, the `nt` built-in test provides a mechanism for LLVM test-suite tests that just define an extension test module. These tests are passed the user configuration parameters for a test run and expected to return back the test results in the LNT native format.

Test modules are defined by providing a `TestModule` file in a subdirectory of the `LNTBased` root directory inside the LLVM test-suite repository. The `TestModule` file is expected to be a well-formed Python module that provides a `test_class` global variable which should be a subclass of the `lnt.tests.nt.TestModule` abstract base class.

The test class should override the `execute_test` method which is passed an options dictionary containg the NT user parameters which apply to test execution, and the test should return the test results as a list of `lnt.testing.TestSamples` objects.

The `execute_test` method is passed the following options describing information about the module itself:

- `MODULENAME` - The name of the module (primarily intended for use in producing well structured test names).
- `SRCROOT` - The path to the modules source directory.
- `OBJROOT` - The path to a directory the module should use for temporary output (build products). The directory is guaranteed to exist but is not guaranteed to be clean.

The method is passed the following options which apply to how tests should be executed:

- `THREADS` - The number of parallel processes to run during testing.
- `BUILD_THREADS` - The number of parallel processes to use while building tests (if applicable).

The method is passed the following options which specify how and whether tests should be executed remotely. If any of these parameters are present then all are guaranteed to be present.

- `REMOTE_HOST` - The host name of the remote machine to execute tests on.
- `REMOTE_USER` - The user to log in to the remote machine as.
- `REMOTE_PORT` - The port to connect to the remote machine on.
- `REMOTE_CLIENT` - The `rsh` compatible client to use to connect to the remote machine with.

The method is passed the following options which specify how to build the tests:

- `CC` - The C compiler command to use.
- `CXX` - The C++ compiler command to use.
- `CFLAGS` - The compiler flags to use for building C code.
- `CXXFLAGS` - The compiler flags to use for building C++ code.

The method is passed the following optional parameters which specify the environment to use for various commands:

- `COMPILE_ENVIRONMENT_OVERRIDES` [optional] - If given, a `env` style list of environment overrides to use when compiling.
- `LINK_ENVIRONMENT_OVERRIDES` [optional] - If given, a `env` style list of environment overrides to use when linking.

- `EXECUTION_ENVIRONMENT_OVERRIDES` [optional] - If given, a `env` style list of environment overrides to use when executing tests.

For more information, see the example tests in the LLVM test-suite repository under the `LNT/Examples` directory.

Capturing Linux perf profile info

When using the CMake driver in the test-suite, LNT can also capture profile information using linux perf. This can then be explored through the LNT webUI as demonstrated at http://blog.llvm.org/2016/06/using-lnt-to-track-performance.html .

To capture these profiles, use command line option `--use-perf=all`. A typical command line using this for evaluating the performance of generated code looks something like the following:

```
$ lnt runtest test-suite \
     --sandbox SANDBOX \
     --cc ~/bin/clang \
     --use-cmake=/usr/local/bin/cmake \
     --use-lit=~/llvm/utils/lit/lit.py \
     --test-suite ~/llvm-test-suite \
     --benchmarking-only \
     --build-threads 8 \
     --threads 1 \
     --use-perf=all \
     --exec-multisample=5 \
     --run-under 'taskset -c 1'
```

Bisecting: `--single-result` and `--single-result-predicate`

The LNT driver for the CMake-based test suite comes with helpers for bisecting conformance and performance changes with `llvmlab bisect`.

`llvmlab bisect` is part of the `zorg` repository and allows easy bisection of some predicate through a build cache. The key to using `llvmlab` effectively is to design a good predicate command - one which exits with zero on 'pass' and nonzero on 'fail'.

LNT normally runs one or more tests then produces a test report. It always exits with status zero unless an internal error occurred. The `--single-result` argument changes LNT's behaviour - it will only run one specific test and will apply a predicate to the result of that test to determine LNT's exit status.

The `--single-result-predicate` argument defines the predicate to use. This is a Python expression that is executed in a context containing several pre-set variables:

- `status` - Boolean passed or failed (True for passed, False for failed).
- `exec_time` - Execution time (note that `exec` is a reserved keyword in Python!)
- `compile` (or `compile_time`) - Compilation time

Any metrics returned from the test, such as "score" or "hash" are also added to the context.

The default predicate is simply `status` - so this can be used to debug correctness regressions out of the box. More complex predicates are possible; for example `exec_time < 3.0` would bisect assuming that a 'good' result takes less than 3 seconds.

Full example using `llvmlab` to debug a performance improvement:

```
$ llvmlab bisect --min-rev=261265 --max-rev=261369 \
  lnt runtest test-suite \
    --cc '%(path)s/bin/clang' \
    --sandbox SANDBOX \
    --test-suite /work/llvm-test-suite \
    --use-lit lit \
    --run-under 'taskset -c 5' \
    --cflags '-O3 -mthumb -mcpu=cortex-a57' \
    --single-result MultiSource/Benchmarks/TSVC/Expansion-flt/Expansion-flt \
    --single-result-predicate 'exec_time > 8.0'
```

Producing Diagnositic Reports

The test-suite module can produce a diagnostic report which might be useful for figuring out what is going on with a benchmark:

```
$ lnt runtest test-suite \
       --sandbox /tmp/BAR \
       --cc ~/llvm.obj.64/Release+Asserts/bin/clang \
       --cxx ~/llvm.obj.64/Release+Asserts/bin/clang++ \
       --use-cmake=/usr/local/bin/cmake \
       --use-lit=~/llvm/utils/lit/lit.py \
       --test-suite ~/llvm-test-suite \
       --cmake-cache Release \
       --diagnose --only-test SingleSource/Benchmarks/Stanford/Bubblesort
```

This will run the test-suite many times over, collecting useful information in a report directory. The report collects many things like execution profiles, compiler time reports, intermediate files, binary files, and build information.

Cross-compiling

The best way to run the test-suite in a cross-compiling setup with the cmake driver is to use cmake's built-in support for cross-compiling as much as possible. In practice, the recommended way to cross-compile is to use a cmake toolchain file (see https://cmake.org/cmake/help/v3.0/manual/cmake-toolchains.7.html#cross-compiling)

An example command line for cross-compiling on an X86 machine, targeting AArch64 linux, is:

```
$ lnt runtest test-suite \
       --sandbox SANDBOX \
       --test-suite /work/llvm-test-suite \
       --use-lit lit \
       --cppflags="-O3" \
       --run-under=$HOME/dev/aarch64-emu/aarch64-qemu.sh \
       --cmake-define=CMAKE_TOOLCHAIN_FILE:FILEPATH=$HOME/clang_aarch64_linux.cmake
```

The key part here is the CMAKE_TOOLCHAIN_FILE define. As you're cross-compiling, you may need a –run-under command as the produced binaries probably won't run natively on your development machine, but something extra needs to be done (e.g. running under a qemu simulator, or transferring the binaries to a development board). This isn't explained further here.

In your toolchain file, it's important to specify that the cmake variables defining the toolchain must be cached in CMakeCache.txt, as that's where lnt reads them from to figure out which compiler was used when needing to construct metadata for the json report. An example is below. The important keywords to make the variables appear in the CMakeCache.txt are "CACHE STRING "" FORCE":

```
$ cat clang_aarch64_linux.cmake
set(CMAKE_SYSTEM_NAME Linux )
set(triple aarch64-linux-gnu )
set(CMAKE_C_COMPILER /home/user/build/bin/clang CACHE STRING "" FORCE)
set(CMAKE_C_COMPILER_TARGET ${triple} CACHE STRING "" FORCE)
set(CMAKE_CXX_COMPILER /home/user/build/bin/clang++ CACHE STRING "" FORCE)
set(CMAKE_CXX_COMPILER_TARGET ${triple} CACHE STRING "" FORCE)
set(CMAKE_SYSROOT /home/user/aarch64-emu/sysroot-glibc-linaro-2.23-2016.11-aarch64-
↪linux-gnu )
set(CMAKE_C_COMPILER_EXTERNAL_TOOLCHAIN /home/user/aarch64-emu/gcc-linaro-6.2.1-2016.
↪11-x86_64_aarch64-linux-gnu )
set(CMAKE_CXX_COMPILER_EXTERNAL_TOOLCHAIN /home/user/aarch64-emu/gcc-linaro-6.2.1-
↪2016.11-x86_64_aarch64-linux-gnu )
```

CHAPTER 5

Concepts

LNT's data model is pretty simple, and just following the *Quickstart Guide* can get you going with performance testing. Moving beyond that, it is useful to have an understanding of some of the core concepts in LNT. This can help you get the most out of LNT.

Orders Machines and Tests

LNT's data model was designed to track the performance of a system in many configurations over its evolution. In LNT, an Order is the x-axis of your performance graphs. It is the thing that is changing. Examples of common orders are software versions, Subversion revisions, and time stamps. Orders can also be used to represent treatments, such as a/b. You can put anything you want into LNT as an order, as long as it can be sorted by Python's sort function.

A Machine in LNT is the logical bucket which results are categorized by. Comparing results from the same machine is easy, across machines is harder. Sometimes machine can literally be a machine, but more abstractly, it can be any configuration you are interested in tracking. For example, to store results from an Arm test machine, you could have a machine call "ArmMachine"; but, you may want to break machines up further for example "ArmMachine-Release" "ArmMachine-Debug", when you compile the thing you want to test in two modes. When doing testing of LLVM, we often string all the useful parameters of the configuration into one machines name:

```
<hardware>-<arch>-<optimization level>-<branch-name>
```

Tests are benchmarks, the things you are actually testing.

Runs and Samples

Samples are the actual data points LNT collects. Samples have a value, and belong to a metric, for example a 4.00 second (value) compile time (metric). Runs are the unit in which data is submitted. A Run represents one run through a set of tests. A run has a Order which it was run on, a Machine it ran on, and a set of Tests that were run, and for each Test one or more samples. For example, a run on ArmMachine at Order r1234 might have two Tests, test-a which had 4.0 compile time and 3.5 and 3.6 execution times and test-b which just has a 5.0 execution time. As new runs are

submitted with later orders (r1235, r1236), LNT will start tracking the per-machine, per-test, per-metric performance of each order. This is how LNT tracks performance over the evolution of your code.

Test Suites

LNT uses the idea of a Test Suite to control what metrics are collected. Simply, the test suite acts as a definition of the data that should be stored about the tests that are being run. LNT currently comes with two default test suites. The Nightly Test Suite (NTS) (which is run far more often than nightly now), collects 6 metrics per test: compile time, compile status, execution time, execution status, score and size. The Compile (compile) Test Suite, is focused on metrics for compile quality: wall, system and user compile time, compile memory usage and code size. Other test suites can be added to LNT if these sets of metrics don't match your needs.

Any program can submit results data to LNT, and specify any test suite. The data format is a simple JSON file, and that file needs to be submitted to the server using either the lnt import or submit commands, see *The lnt Tool*, or HTTP POSTed to the submitRun URL.

The most common program to submit data to LNT is the LNT client application itself. The `lnt runtest nt` command can run the LLVM test suite, and submit data under the NTS test suite. Likewise the `lnt runtest compile` command can run a set of compile time benchmarks and submit to the Compile test suite.

CHAPTER 6

Accessing Data outside of LNT: REST API

LNT provides REST APIs to access data stored in the LNT database.

Endpoints

The API endpoints live under the top level api path, and have the same database and test-suite layout. For example:

```
http://lnt.llvm.org/db_default/v4/nts/machine/1330
Maps to:
http://lnt.llvm.org/api/db_default/v4/nts/machines/1330
```

The machines endpoint allows access to all the machines, and properties and runs collected for them. The runs endpoint will fetch run and sample data. The samples endpoint allows for the bulk export of samples from a number of runs at once.

Endpoint	Description
/machines/	List all the machines in this testsuite.
/machines/*id*	Get all the runs info and machine fields for machine *id*.
/runs/*id*	Get all the run info and sample data for one run *id*.
/orders/*id*	Get all order info for Order *id*.
/samples?runid=1&runid=2	Retrieve all the sample data for a list of run ids. Run IDs should be pass as args. Will return sample data in the samples section, as a list of dicts, with a key for each metric type. Empty samples are not sent.
/samples/*id*	Get all non-empty sample info for Sample *id*.

Write Operations

The machines, orders and runs endpoints also support the DELETE http method. The user must include a http header called "AuthToken" which has the API auth token set in the LNT instance configuration.

The API Auth token can be set by adding *api_auth_token* to the instances lnt.cfg config file:

```
# API Auth Token
api_auth_token = "SomeSecret"
```

Example:

```
curl --request DELETE --header "AuthToken: SomeSecret" http://localhost:8000/api/db_
↪default/v4/nts/runs/1
```

CHAPTER 7

Importing Data

Importing Data in a Text File

The LNT importreport command will import data in a simple text file format. The command takes a space separated key value file and creates an LNT report file, which can be submitted to a LNT server. Example input file:

```
foo.exec 123
bar.size 456
foo/bar/baz.size 789
```

The format is "test-name.metric", so exec and size are valid metrics for the test suite you are submitting to.

Example:

```
echo -n "foo.exec 25\nbar.score 24.2\nbar/baz.size 110.0\n" > results.txt
lnt importreport --machine=my-machine-name --order=1234 --testsuite=nts results.txt␣
↪report.json
lnt submit http://mylnt.com/default/submitRun report.json
```

LNT Report File Format

The lnt importreport tool is an easy way to import data into LNTs test format. You can also create LNTs report data directly for additional flexibility.

First, make sure you've understood the underlying *Concepts* used by LNT.

```
{
    "format_version": "2",
    "machine": {
        "name": _String_      // machine name, mandatory
        (_String_: _String_)* // optional extra info
    },
    "run": {
```

```
        "start_time": "%Y-%m-%dT%H:%M:%S", // mandatory, ISO8061 timestamp
        "end_time": "%Y-%m-%dT%H:%M:%S",   // mandatory, ISO8061 timestamp, can equal␣
↪start_time if not known.
        (_String_: _String_)* // optional extra info about the run.
        // At least one of the extra fields is used as ordering and is
        // mandatory. For the 'nts' and 'Compile' schemas this is the
        // 'llvm_project_revision' field.
    },
    "tests": [
        {
            "name": _String_,   // test name mandatory
            (_String_: _Data_)* // List of metrics, _Data_ allows:
                                // number, string or list of numbers
        }+
    ]
}
```

A concrete small example is

```
{
    "format_version": "2",
    "machine": {
       "name": "LNT-AArch64-A53-O3__clang_DEV__aarch64",
       "hardware": "HAL 9000"
    },
    "run": {
       "end_time": "2017-07-18T11:28:23.991076",
       "start_time": "2017-07-18T11:28:33.00000",
       "llvm_project_revision": "265649",
       "compiler_version": "clang 4.0"
    },
    "tests": [
       {
           "name": "benchmark1",
           "execution_time": [ 0.1056, 0.1055 ],
           "hash": "49333a87d501b0aea2191830b66b5eec"
       },
       {
           "name": "benchmark2",
           "compile_time": 13.12,
           "execution_time": 0.2135,
           "hash": "c321727e7e0dfef279548efdb8ab2ea6"
       }
    ]
}
```

Given how simple it is to make your own results and send them to LNT, it is common to not use the LNT client application at all, and just have a custom script run your tests and submit the data to the LNT server. Details on how to do this are in *lnt.testing*

Default Test Suite (NTS)

The default test-suite schema is called NTS. It was originally designed for nightly test runs of the llvm test-suite. However it should fit many other benchmark suites as well. The following metrics are supported for a test:

- `execution_time`: Execution time in seconds; lower is better.

- `score`: Benchmarking score; higher is better.
- `compile_time`: Compiling time in seconds; lower is better.
- `execution_status`: A non zero value represents an execution failure.
- `compilation_status`: A non zero value represents a compilation failure.
- `hash_status`: A non zero value represents a failure computing the executable hash.
- `mem_bytes`: Memory usage in bytes during execution; lower is better.
- `code_size`: Code size (usually the size of the text segment) in bytes; lower is better.

The *run* information is expected to contain this:

- `llvm_project_revision`: The revision or version of the compiler used for the tests. Used to sort runs.

Custom Test Suites

LNTs test suites are derived from a set of metadata definitions for each suite. Simply put, suites are a collections of metrics that are collected for each run. You can define your own test-suites if the schema in a different suite does not already meet your needs.

To create a schema place a yaml file into the schemas directory of your lnt instance. Example:

```
format_version: '2'
name: size
metrics:
  - name: text_size
    bigger_is_better: false
    type: Real
  - name: data_size
    bigger_is_better: false
    type: Real
  - name: score
    bigger_is_better: true
    type: Real
  - name: hash
    type: Hash
run_fields:
  - name: llvm_project_revision
    order: true
machine_fields:
  - name: hardware
  - name: os
```

- LNT currently supports the following metric types:
  - `Real`: 8-byte IEEE floating point values.
  - `Hash`: String values; limited to 256, sqlite is not enforcing the limit.
  - `Status`: StatusKind enum values (limited to 'PASS', 'FAIL', 'XFAIL' right now).
- You need to mark at least 1 of the run fields as `order: true` so LNT knows how to sort runs.

CHAPTER 8

Developer Guide

This developer guide aims to get you started with developing LNT. At the moment, a lot of detailed info is missing, but hopefully this will get you started.

Installation

See *Quickstart Guide* for setting up an installation. Use the "develop" option when running ~/lnt/setup.py.

Running LNT's Regression Tests

LNT has a growing body of regression tests that makes it easier to improve LNT without accidentally breaking existing functionality. Just like when developing most other LLVM sub-projects, you should consider adding regression tests for every feature you add or every bug you fix. The regression tests must pass at all times, therefore you should run the regression tests as part of your development work-flow, just like you do when developing on other LLVM sub-projects.

The LNT regression tests make use of lit and other tools like FileCheck. At the moment, probably the easiest way to get them installed is to compile LLVM and use the binaries that are generated there. Assuming you've build LLVM into $LLVMBUILD, and installed lnt in $LNTINSTALL you can run the regression tests using the following command:

```
PATH=$LLVMBUILD/bin:$LNTINSTALL/bin:$PATH llvm-lit -sv ./tests
```

If you don't like temporary files being created in your LNT source directory, you can run the tests in a different directory too:

```
mkdir ../run_lnt_tests
cd ../run_lnt_tests
PATH=$LLVMBUILD/bin:$LNTINSTALL/bin:$PATH llvm-lit -sv ../lnt/tests
```

For simple changes, adding a regression test and making sure all regression tests pass, is often a good enough testing approach. For some changes, the existing regression tests aren't good enough at the moment, and manual testing will be needed.

Optional Tests

Some tests require additional tools to be installed and are not enabled by default. You can enable them by passing additional flags to lit:

-Dpostgres=1 Enable postgres database support testing. This requires at least postgres version 9.2 and the *initdb* and *postgres* binaries in your path. Note that you do not need to setup an actual server, the tests will create temporary instances on demand.

-Dmysql=1 Enable mysql database support testing. This requires MySQL-python to be installed and expects the *mysqld* and *mysqladmin* binaries in your path. Note that you do not need to setup an actual server, the tests will create temporary instances on demand.

-Dtidylib=1 Check generated html pages for errors using tidy-html5. This requires pytidylib and tidy-html5 to be installed.

-Dcheck-coverage=1 Enable `coverage.py` reporting, assuming the coverage module has been installed and `sitecustomize.py` in the virtualenv has been modified appropriately.

Example:

```
PATH=$LLVMBUILD/bin:$LNTINSTALL/bin:$PATH llvm-lit -sv -Dpostgres=1 -Dmysql=1 -
↪Dtidylib=1 ../lnt/tests
```

CHAPTER 9

Performance profiles

LNT has support for storing and displaying performance profiles. The intent of these profiles is to expose code generation differences between test samples and to allow easy identification of hot sections of code.

Principles of profiles in LNT

Profiles in LNT are represented in a custom format. The user interface operates purely on queries to this custom format. Adapters are written to convert from other formats to LNT's profile format. Profile data is uploaded as part of the normal JSON report to the LNT server.

Producing profile data

Profile generation can be driven directly through python API calls (for which `lnt profile` is a wrapper) or using the `lnt runtests` tool.

Producing profile data via `lnt runtests test-suite`

Note: Profile collection via LNT is currently only supported on **Linux systems** as the only adapter that has currently been written uses Linux's `perf` infrastructure. When more adapters have been written, LNT can grow support for them.

If your test system is already using `lnt runtests` to build and run tests, the simplest way to produce profiles is simply to add a single parameter:

```
--use-perf=all
```

The `--use-perf` option specifies what to use Linux Perf for. The options are:

- `none`: Don't use `perf` for anything
- `time`: Use `perf` to measure compile and execution time. This can be much more accurate than `time`.
- `profile`: Use `perf` for profiling only.
- `all`: Use `perf` for profiling and timing.

The produced profiles live alongside each test executable, named `$TEST.perf_data`. These profiles are processed and converted into LNT's profile format at the end of test execution and are inserted into the produced `report.json`.

Producing profile data without `lnt runtests test-suite`

A supported usecase of LNT is to use the LNT server for performance tracking but to use a different test driver than `lnt runtests` to actually build, run and collect statistics for tests.

The profiling data sits inside the JSON report submitted to LNT. This section will describe how to add profile data to an already-existing JSON report; See *Importing Data* for details of the general structure of the JSON report.

The first step is to produce the profile data itself in LNT format suitable for sending via JSON. To import a profile, use the `lnt profile upgrade` command:

```
lnt profile upgrade my_profile.perf_data /tmp/my_profile.lntprof
```

`my_profile.perf_data` is assumed here to be in Linux Perf format but can be any format for which an adapter is registered (this currently is only Linux Perf but it is expected that more will be added over time).

`/tmp/my_profile.lntprof` is now an LNT profile in a space-efficient binary form. To prepare it to be sent via JSON, we must base-64 encode it:

```
base64 -i /tmp/my_profile.lntprof -o /tmp/my_profile.txt
```

Now we just need to add it to the report. Profiles look similar to hashes in that they are samples with string data:

```
{
   "Machine": {
     ...
   },
   "Run": {
     ...
   },
   "Tests": [
        {
            "Data": [
                0.1056,
                0.1055
            ],
            "Info": {},
            "Name": "nts.suite1/program1.exec"
        },
        {
            "Data": [
                "eJxNj8EOgjAMhu99Cm9wULMOEHgBE888QdkASWCQFWJ8e1v04JIt+9f//
↪7qmfkVoEj8yMXdzO70v/
↪RJn2hJYrRQiveSWATdJvwe3jUtgecgh9Wsh9T6gyJvKUjm0kegK0mmt9UCjJUSgB5q8KsobUJOQ96dozr8tAbRApPbssOeCcm8
↪RGUUwXt7iviPEDLJN92yh62LR7I8aBUMysgLnaKNFNzzMo8y7uGplQ4sa/j6rfn60WYaGdRhtT9fP5+JUW4=
↪"
            ],
            "Info": {},
```

```
            "Name": "nts.suite2/program1.profile"
        }
    ]
}
```

Supported formats

Linux Perf

Perf profiles are read directly from the binary `perf.data` file without using the `perf` wrapper tool or any Linux/GPL headers. This makes it runnable on non-Linux platforms although this is only really useful for debugging as the profiled binary / libraries are expected to be readable.

The perf import code uses a C++ extension called cPerf that was written for the LNT project. It is less functional than `perf annotate` or `perf report` but produces much the same data in a machine readable form about 6x quicker. It is written in C++ because it is difficult to write readable Python that performs efficiently on binary data. Once the event stream has been aggregated, a python dictionary object is created and processing returns to Python. Speed is important at this stage because the profile import may be running on older or less powerful hardware and LLVM's test-suite contains several hundred tests that must be imported!

Note: In recent versions of Perf a new subcommand exists: `perf data`. This outputs the event trace in CTF format which can then be queried using babeltrace and its Python bindings. This would allow to remove a lot of custom code in LNT as long as it is similarly performant.

Adding support for a new profile format

To create a new profile adapter, a new Python class must be created in the `lnt.testing.profile` package which subclasses the `ProfileImpl` class:

class `lnt.testing.profile.profile.`**`ProfileImpl`**

static `checkFile` (*fname*)
Return True if 'fname' is a serialized version of this profile implementation.

static `deserialize` (*fobj*)
Reads a profile from 'fobj', returning a new profile object. This can be lazy.

`getCodeForFunction` (*fname*)
Return a *generator* which will return, for every invocation, a three-tuple:

```
(counters, address, text)
```

Where counters is a dict : (e.g.) `{'cycles': 50.0}`, text is in the format as returned by getDisassemblyFormat(), and address is an integer.

The counter values must be percentages (of the function total), not absolute numbers.

`getDisassemblyFormat`()
Return the format for the disassembly strings returned by getCodeForFunction(). Possible values are:

- `raw` - No interpretation available - pure strings.
- `marked-up-disassembly` - LLVM marked up disassembly format.

getFunctions()
Return a dict containing function names to information about that function.

The information dict contains:

- `counters` - counter values for the function.
- `length` - number of times to call getCodeForFunction to obtain all instructions.

The dict should *not* contain disassembly / function contents. The counter values must be percentages, not absolute numbers.

E.g.:

```
{'main': {'counters': {'cycles': 50.0, 'branch-misses': 0}, 'length': 200},
 'dotest': {'counters': {'cycles': 50.0, 'branch-misses': 0}, 'length': 4}}
```

getTopLevelCounters()
Return a dict containing the counters for the entire profile. These will be absolute numbers: `{'cycles': 5000.0}` for example.

getVersion()
Return the profile version.

serialize(*fname=None*)
Serializes the profile to the given filename (base). If fname is None, returns as a bytes instance.

static upgrade(*old*)
Takes a previous profile implementation in 'old' and returns a new ProfileImpl for this version. The only old version that must be supported is the immediately prior version (e.g. version 3 only has to handle upgrades from version 2.

Your subclass can either implement all functions as specified, or do what `perf.py` does which is only implement the `checkFile()` and `deserialize()` static functions. In this model inside `deserialize()` you would parse your profile data into a simple dictionary structure and create a `ProfileV1Impl` object from it. This is a really simple profile implementation that just works from a dictionary representation:

class `lnt.testing.profile.profilev1impl.`**ProfileV1**(*data*)
ProfileV1 files not clever in any way. They are simple Python objects with the profile data layed out in the most obvious way for production/consumption that are then pickled and compressed.

They are expected to be created by simply storing into the `self.data` member.

The `self.data` member has this format:

```
{
 counters: {'cycles': 12345.0, 'branch-misses': 200.0}, # Counter values are␣
↪absolute.
 disassembly-format: 'raw',
 functions: {
   name: {
     counters: {'cycles': 45.0, ...}, # Note counters are now percentages.
     data: [
       [463464, {'cycles': 23.0, ...}, '      add r0, r0, r1'}],
       ...
     ]
   }
  }
}
```

Viewing profiles

Once profiles are submitted to LNT, they are available either through a manual URL or through the "runs" page.

On the run results page, "view profile" links should appear when table rows are hovered over if profile data is available.

Note: It is known that this hover-over effect isn't touchscreen friendly and is perhaps unintuitive. This page should be modified soon to make the profile data link more obvious.

Alternatively a profile can be viewed by manually constructing a URL:

```
db_default/v4/nts/profile/<test-id>/<run1-id>/<run2-id>
```

Where:

- `test-id` is the database TestID of the test to display
- `run1-id` is the database RunID of the run to appear on the left of the display
- `run2-id` is the database RunID of the run to appear on the right of the display

Obviously, this URL is somewhat hard to construct, so using the links from the run page as above is recommended.

CHAPTER 10

Indices and tables

- genindex
- modindex
- search

Module Listing

`lnt.testing` – Test Data Creation

Utilities for working with the LNT test format.

Clients can easily generate LNT test format data by creating Report objects for the runs they wish to submit, and using Report.render to convert them to JSON data suitable for submitting to the server.

class `lnt.testing.`**`Report`**(*machine*, *run*, *tests*)
Information on a single testing run.

In the LNT test model, every test run should define exactly one machine and run, and any number of test samples.

`update_report`(*new_samples*)
Add extra samples to this report, and update the end time.

class `lnt.testing.`**`Machine`**(*name*, *info={}*)
Information on the machine the test was run on.

The info dictionary can be used to describe additional information about the machine, for example the hardware resources or the operating environment.

Machines entries in the database are uniqued by their name and the entire contents of the info dictionary.

class `lnt.testing.`**`Run`**(*start_time*, *end_time*, *info={}*)
Information on the particular test run.

The start and end time should always be supplied with the run. Currently, the server uses these to order runs. In the future we will support additional ways to order runs (for example, by a source revision).

As with Machine, the info dictionary can be used to describe additional information on the run. This dictionary should be used to describe information on the software-under-test that is constant across the test run, for example the revision number being tested. It can also be used to describe information about the current state which could be useful in analysis, for example the current machine load.

class lnt.testing.**TestSamples**(*name*, *data*, *info={}*, *conv_f=<type 'float'>*)

Test sample data.

The test sample data defines both the tests that were run and their values. The server automatically creates test database objects whenever a new test name is seen.

Test names are intended to be a persistent, recognizable identifier for what is being executed. Currently, most formats use some form of dotted notation for the test name, and this may become enshrined in the format in the future. In general, the test names should be independent of the software-under-test and refer to some known quantity, for example the software under test. For example, 'CINT2006.403_gcc' is a meaningful test name.

The test info dictionary is intended to hold information on the particular permutation of the test that was run. This might include variables specific to the software-under-test . This could include, for example, the compile flags the test was built with, or the runtime parameters that were used. As a general rule, if two test samples are meaningfully and directly comparable, then the should have the same test name but different info paramaters.

The report may include an arbitrary number of samples for each test for situations where the same test is run multiple times to gather statistical data.

Python Module Index

l

Index