SAMSUNG INFRA RESEARCH LAB

Tool for Automatic Compiler Tuning (TACT) User Manual

Rev. 0.9.7

November 14, 2014

Contents

Chapter 1
Introduction

The Tool for Automatic Compiler Tuning (TACT) provides a framework for automatic application tuning based on given criteria. It provides a set of tools for program compilation and run in heterogeneous environment, automatic tuning of compiler options using genetic algorithm, and results analysis tools. This document describes the operation of the tool, provides instructions on adding new applications for tuning, configuring and starting tuning process, and analyzing the results. It also provides information about the tool internal structure.

The document is organized as follows. In this chapter we overview tool's main features and outline the document structure. In Chapter 2 we provide instructions for the tool installation. In Chapter 3 we describe TACT structure and its internal workflow. In Chapter 4 in the form of tutorial we provide a step-by-step guide to adding new sample application into the tuning system and starting the tuning. In Chapter 5 we provide a step-by-step guide to overall tuning process for x264 application. Chapter 6 is dedicated to result analysis techniques and tools, and in Chapter 7 we give a reference of all scripts and configuration files used in the system.

Chapter 2
Installation

2.1 Setting Up Required Tools

Most of TACT code that run on host machine is written in Ruby, so `ruby` interpreter version 1.9.1 or higher and `rubygems` should be installed (on Debian-based system, it can be installed with `sudo apt-get install ruby ruby-dev rubygems`). We are using `simple-xlsx` library for generating reports, and it requires `fast_xs` and `rubyzip` gems (`sudo gem install rubyzip fast_xs`). Configure script requires `rubytree` gem (`sudo gem install rubytree`). Also, for building Pareto graphs `gnuplot` is required (`sudo apt-get install gnuplot`).

2.2 Configuring System-Wide Parameters

To adjust the number of parallel jobs for compilation on host machine, edit `MAKE_FLAGS` variable in `$TACT_DIR/etc/build-config`. Typically it should be set to the number of cores found on host machine plus one. Note that the actual number of concurrent build processes may exceed that number since several build processes may run in parallel.

2.3 Setting Up Networking

TACT can be setup to use multiple nodes or just a single machine. In a single-machine setup TACT uses same host to compile and run programs. In multi-machine setup it executes apps on separate dedicated machines (test boards), while simultaneously compiling on host machine, which allows to speed up the tuning. Also such setup is required for cross-compilation. On

a single machine, simultaneous compilation and execution is also possible, but it may affect precision of the results. If you compile natively, and want a quick start, you may now skip the rest of the configuration and go straight to Tutorial in Chapter 4.

In multi-machine mode, TACT uses NFS for communication with test boards, so the first step in tool installation is to set up NFS server and clients. The only requirement to NFS setup is that tool's installation directory should be accessible with the same path on all systems involved in tuning. We suggest to run tuning under the same user on all nodes, and setup TACT into `$HOME/<HOST-SYSTEM-NAME>`. Then this folder should be mounted on every target node.

Also you'll need to set up key-based authorization to testboards in order to enable running tasks on boards without user authorization. Then, the tool can be installed in `$HOME/<HOST-SYSTEM-NAME>/tact`. We will refer this direcory as `$TACT_DIR`.

TACT uses `$TACT_DIR/task_manager/system.xml` configuration file for all network issues. In this file you should describe all target boards and intermediate hosts that are needed to reach the testboards. Example of `system.xml` file:

```
<?xml version="1.0"?>
<system_config>
<!-- Description of tuning server -->
<main_server network_name="condor" id="main" />
<!-- Description of intermediate servers if any -->
<inter_server connected_from="main" network_name="thrush" id="middle" />
<!-- Description of boards -->
<board id="tizen1" connected_from="middle" network_name="192.168.101.3"
     os="tizen" />
<board id="tizen3" connected_from="middle" network_name="tizen3" os="tizen" />
<!-- Description of boards classes -->
<board_class class_name="cortex-A8">
  <board1 id="tizen1" />
  <board1 id="tizen3" />
</board_class>
<!-- Username for tact on boards and intermediate servers -->
<tact_username value="tact"/>
</system_config>
```

Chapter 3
The Tool for Automatic Compiler Tuning Structure

This chapter describes the Tool for Automated Compiler Tuning main concepts and structure.

3.1 High-Level Structure Overview

Tool's installation main directory contains subdirectories with the following content:

`apps`
: Test applications for tuning, including their source code and build directories, various scripts required to build and run test applications, results parsers and test descriptions. Run logs, tuning results and generated reports are also stored here.

`bin`
: Tool binaries and scripts, including `tact` script which is main command-line user interface script for the tool.

`docs`
: Tool documentation.

`etc`
: Tool's configuration files that are invariant from test applications.

`lib`
: Libraries used by the tool.

`target`
: Binary utilities compiled for target (as reported by `uname -m`), by default these include `timeout` and `seq` programs, which are required for tool operation. As target test boards often have very limited set of utilities on their root file system, this directory may contain common programs required to run tests, e.g. `perl`.

`task_manager`
: Files used for communication with target boards.

Typically, tool's user only should be interested in contents of `apps` and `etc` subdirectories.

3.2 Applications, Tests and Benchmarks

Every test application directory consists of two levels: *application* and *test* level, where test is an instance of application. For example, application `app_name` would be located in `apps/app_name`, and contain one or more tests as its subdirectories. This structure allows to separate data that is specific to application itself from data specific for certain test. For example, we may want to tune the same application with different input data or using different compiler or compilation parameters, or try tuning with certain patches applied to the application source. With two-level structure we can create symbolic links to common shared libraries, application sources and test descriptions from specific test directories into application directory. We refer test directory in documentation (as well as in scripts) as `$TEST_DIR`, and application directory as `$APP_DIR`.

To help better understand these notions, we can draw an analogy with sports: the application and test would be in similar relationship as are athletics and sprint or jumping. As sprint and jumping may share same stadium, spectators and sportsmen, the rules, judges and equipment may be different for each discipline. Tests may also share some scripts, data, application binaries or sources, but have their own tuning goals or environment.

Each test instance in its turn can be a *test suite* that consists of multiple *benchmarks*. In this case a composite value is computed from all benchmark results and is used as an optimization criterion in tuning. At the same time, performance of each separate benchmark still can be found in report tables, so the user can find out which tests contributes the most to performance change.

3.3 Application and Test Directory Structure

Test or application directories have the following structure:

`bin`
: Contains scripts for building, running test applications, parsing its output, computing hashes, etc. Typically, this directory contains only *interface* scripts, that are required by TACT for its operation.

`etc`
: Holds test configuration files. They contain compiler options to tune, number of generations for options evolution, number of testboards, build pools,

program timeout, benchmark test suite description (including weights for each test) and other configuration data.

`private-bin, private-etc`
User auxiliary helper scripts and test-specific configuration data, that is not directly required by TACT interface can be stored here.

`log`
Contains subdirectories with run logs. Each directory name is given by the daytime when tuning session was started. The latest results are always available through symlink named `current`.

`pool`
Contains build pools. Each pool contains separate directories where program is built, installed, and it's output is stored. Pool is rebuilt for every compiler options combination that is tested on a testboard. The notion of build pools is described in detail in Section 3.5.

`src`
Holds the application source code. All pools are built from the same source code located in this directory.

`shared`
Holds sources of programs that are shared among tests or pools of one test, e.g. libraries required by application, as well as their binary install tree. Everything that could be compiled only once for all tests (or pools) should be put in there. Typically, this directory is present only at application level directory.

Application- and test-level directories (`$APP_DIR` and `$TEST_DIR`) have similar structure; moreover, standard-API scripts required by the tool (like `compute-binary-hash`) are first searched in `$TEST_DIR`. If missing, they are searched in `$APP_DIR`, and, finally, in `$TACT_DIR`. This way, using directory hierarchy we implement some sort of inheritance model, so the scripts at "deeper" level may redefine those at higher levels. Inheriting standard scripts doesn't require symlinks, but source and data directories should be linked explicitly.

Scripts and data are not strictly assigned either to test or application level, but code and data should be shared among tests as much as possible. This way, if user updates application from its repository, changes would become available to all tests, as they share the same source tree.

3.4 Main tuning parameters

TACT implements a genetic algorithm to find the "best" options for compiling programs with the GNU Compiler Collection (GCC) C and C++ compilers. First, it reads a configuration file with tuning options and compiler options to be tuned, and generates a pool of strings ("chromosomes") consisting of random set of compiler options. Then, for each option string it compiles

target application with given option string, possibly runs it on a test board if needed. Then it calls custom script to evaluate its "fitness" which is located in `$APP_DIR/bin/parse-results.rb` or `$TEST_DIR/bin/parse-results.rb`. This script should inherit base parser – `$TACT_DIR/lib/ResultsParserBase.rb`. "Fitness" value typically corresponds to application performance (e.g. *fps* for graphical applications): the greater is its value, the greater chance the option combination gets to reproduce, but also code size or combination of size/performance can be taken. In such manner all compile option strings from the population pool is being evaluated, and then new generation starts. Option combinations from the population pool from the previous generation are used to produce option combinations for the next generation: two "parent" option sets are chosen randomly from the population (though strings with greater fitness get greater chance to be picked), and new option set is constructed by picking at random option values either from first parent or from second. Then, the process is repeated until the fixed number of generations pass or user stops tuning. TACT has several main tuning paramers, that are specified in `$TEST_DIR/etc/tuning.conf`:

`AFTER_CROSSOVER_MUTATION_RATE`
: Mutation rate for new entities after crossover.

`ARCHIVE_SIZE`
: The number of "alive" entities for population before breeding.

`BUILD_CONFIG`
: Specifies variables to be exported in shell. For example:

```
<build_config name="CC" value="@static_params[:compiler] + '/bin/gcc'" />
```

: Exports full path to compiler executable, based on `COMPILER` variable.

`COMPILER`
: Path to compiler installation directory.

`CROSSOVER_RATE`
: The percent of entities made by crossover from total new entities.

`DO_PROFILING`
: Specifies whether test should be compiled with usage of profile info.

`GREATER_IS_BETTER`
: Specifies whether greater result is better for fitness (as example – higher frames per second is better).

`MEASURE`
: Specifies desired tuning type (performance, size or pareto).

`MIGRATION_RATE`
: Specifies a probability of chromosome migration across populations.

`MUTATION_RATE`
: During the tuning "chromosomes" (option strings) are not only being cross-bred with each other, but also being "mutated" by changing randomly single options in a string. This parameter specifies the probability of such event for every single option in a chromosome.

NUM_GENERATIONS
: Specifies how many generations to run before exit. This can be set to an arbitrary large number, since the tuning can be stopped at any time by pressing Ctrl+C. However, usually good results can be expected after about 15-20 generations. Another way to find out when to finish tuning is looking at how average generation fitness changes: if the tuning still progresses and new good option combinations are still being found, then average fitness should increase or decrease, depending on what is better. During the tuning generation average fitness can be tracked with the following command (executed in $TEST_DIR):

```
tail -f log/current/progress.log | grep "Average fitness"
```

PARETO_BEST_SIZE
: The number of cluster centers on pareto frontier.

PARETO_SUMMARY_CHART_GENERATION_NUMBER
: The number of pareto frontiers on summary chart, that represents tuning progress.

POPULATION_SIZE
: The number of chromosomes in a population.

POPULATIONS
: TACT can use several distinct populations within the same generation: chromosomes can cross-breed only with those of the same population. It would be identical to having NUM_POPULATIONS distinct runs, if not for migration: best chromosomes can randomly migrate from one population to another, exchanging genes. Several simultaneous branches of evolution help to prevent from being stuck in a local maximum and speeds up evolution process. The number of populations should be a multiple of the number of test boards used. Please see Section 3.6 for details on using multiple test boards.

```
<populations>
  <join_results name="local">
    <population board_id="localhost"/>
    <population board_id="localhost"/>
  </join_results>
</populations>
```

  This section should contain one or more subsections join_results. All joined boards should have the same performance. For each population should be specified target board or board class.

REPETITIONS
: The number of repetition of test runs.

SINGLE_OPTION_MUTATION_RATE
: Mutation rate for options.

THREADS_PER_TESTBOARD
: The number of pools used in tuning is evaluated as NUM_TESTBOARDS $*$ THREADS_PER_TESTBOARD. To benefit from parallel build and execution,

`THREADS_PER_TESTBOARD` should be always set to value greater or equal to 2, otherwise compilation and run can not be pipelined, since compiled data should retain in pool until run is completed. For those applications which doesn't have enough distinct object files to compile (so they can not use parallel `make`), their effective compile time may improve from further increasing this parameter value.

3.5 Build Pools

Build pools reside in `$TEST_DIR/pool/<POOL_NUMBER>` directory. Pools can have any name, but usually in tuning they are nubered from 1 to number of pools. First TACT tries to allocate a free pool from `$TEST_DIR/pool` directory. If all pools are busy, then it waits until one becomes available. After the pool is allocated, it calls `rebuild-pool` script (provided for given application by the user) to rebuild an application with given options. All the data that belongs to a single evaluation run of compile options string is stored in these directories. Typically, there are three subdirectories in each pool: `build`, `run` and `log`. In `build` directory the application is built, in `run` it is installed, and `log` contains output from building, running application, parsing and verifying results. Upon pool deallocation these logs are copied from pool directory and are appended to corresponding tuning log in `$TEST_DIR/log/current`.

3.6 Parallel Build and Execution

Pools are created and maintained by TACT automatically, and typically user should not deal with them, unless debugging scripts while adding new test or application into the system. However, the user needs to know about pools to specify the right number of them in configuration file in order to achieve good tuning performance.

Support for parallel compilation and execution greatly speeds up the tuning process. The parallelism is exploited on two levels. First, it allows building application at the same time as TACT waits for result of execution of previously compiled application. Second, it allows using of several test boards in the tuning process so to run tuned applications simultaneously on all of them. A task queue management, remote task execution and synchronization among host and test boards is done through `$TACT_DIR/task_manager/runb` script which can run any given command on any board or class of boards in the FIFO queue.

Note that test boards used for tuning should not be necessarily of the same model, vendor or have the same CPU speed. The test board parallelism is implemented at the level of TACT populations, and only performance re-

sults obtained on the same test board or class of boards are "competing" with each other, so evolution branches progress independently in each population. However, with migrations allowed between populations, best compiler option combinations from each population can spread themselves through other populations and continue their competition with the "native species" of those populations.

3.7 Workflow of the Tuning Process

The tuning tool command-line interface consists of one main script – `tact`.

On host machine, tuning is started with `tact start-tuning` command, which basically runs tuning process with given parameters. Here we will describe the internal workflow of that script. When executed, it performs the following actions:

Allocate a pool. A free build pool is allocated from the `$TEST_DIR/pool` directory. The current pool is exported as `$POOL_DIR` environment variable and is available to user scripts. In pool directory file `in_use` is created, which indicates that pool can not be allocated with other instances of tuning.

Rebuild pool. The script `$TEST_DIR/bin/rebuild-pool` is called, which rebuilds the minimal number of object files in `$POOL_DIR/build` directory and installs test application in `$POOL_DIR/run`.

Send task to a test board. Communication with test board is done via `$TACT_DIR/task_manager/runb` scirpt. The exclusive access to test board is granted by locking on file `$TACT_DIR/task_manager/boardsBOARD_NAME`. Then, commands to execute application is written to `$POOL_DIR/board_run` file. Then, host just waits until `runb` finished run on board.

Run pool. As soon as queue reach desired task, `$POOL_DIR/board_run` will be executed on board. This script contain needed exports, like the number of test repetitions and timeout. Full test output is captured to `$POOL_DIR/log/ run.log`. After run is finished, task will be removed from queue and it proceeds to the next task.

Parse results. After run finished saved test output is passed to application-specific `parse-results.rb` script, which is searched in `bin` of either `$TEST_DIR` or `$APP_DIR`. This script converts arbitrary plain-text output of the test to unified XML representation according to test description in `$TEST_DIR/etc/test-descr.xml`. The XML result file is saved as `$POOL_DIR/log/run.xml`. The advantage of using XML representation is unification of result verification and analysis tools for all test applications.

Verify XML results. In the process of verification the results correctness is checked, i.e. output hashes from different repetitions are compared with each other and with those in a reference file. If hashes don't match, then

status for a run is set to *error*. Though this verification script can be test-specific, usually the standard script from `$TACT_DIR/bin` is used. An XML file with verified run results is written to `$TEST_DIR/log/current/runs`. The latter contains XML files for all runs, named with generation, population and run number on which they were obtained.

Compute score. Verified XML file with run results is read by application-invariant `compute-score` script, which computes a composite value for benchmark suite according to evaluation method specified in test's `etc/test-descr.xml` (the most common method used is geometric mean). The computed value is updated in XML log file, and written to `log/current/results.log` along with compile options string. Finally, this value is passed back to main evolution script and assigns it as a fitness value to correspondent options combination.

Free pool. Logs for this run are appended to global logs in `log/current`, and the pool is released by removing file `in_use`.

The current status (one of the actions described above) of tuning on a given pool is reflected in `status` file in pool directory. A convenient way to continuously monitor status of all pools in `top`-like style is provided by `watch-status` script found in `$TACT_DIR/bin`.

You can tune applications by performance, size or pareto. Tuning by performance is described below in example section. Tuning by size and pareto have some differences. For tuning by size you need one additional script – `compute-size`, and you don't need testboards. In case of pareto tuning you need both `compute-size` script and testboards, also for paretoo tuning you should add some specific options to config, such as archive size.

Tuning progress and results for pareto are also stored in directory `$TEST_DIR/current/archives/generationG_P`, where G and P are generation and population numbers respectively. You may be interested in `results.pdf` and `archive.log` which contains graphical progress and best options archive respectively.

3.8 Packages

TACT has functionality to run a set of tests with specified options like SPEC. First you shoud create symlinks for desired tests in `$TACT_DIR/packages/package_name/test_set/` directory. Then you should create config for package: `$TACT_DIR/packages/package_name/etc/tuning.conf`. Options are the same as for tuning, but package needs only prime options, baselines and desired board specified in `BOARD_ID` parameter. For test a package you should run `tact package-run` from package directory. Then TACT automatically run all tests with `baseline`s and produce report.

Chapter 4
Tutorial: Adding New Application and Starting Tuning

In this chapter we show how to add new application for tuning into the system. In this step-by-step guide we will explain the required modifications to configuration files and interface scripts, and how to start tuning of the application.

4.1 An Example Program for Tuning

Change directory to `tact/apps/tutorial` in your base tool directory. We well refer this directory as `$APP_DIR`, and base TACT directory as `$TACT_DIR`.

In this tutorial we will tune a very simple example program, located in `apps/tutorial/src/test.c`. In this program we use preprocessor defines to emulate effects of compiler optimization options. The number of iterations in program main loop is being adjusted so to make program run faster or slower depending on `-D...` options passed to the compiler. The valid parameters are `-DFAST`, `-DEVEN_FASTER`, `-DSLOW` and `-DMISCOMPILE`. The first option will decrease program run time; the second will decrease it even more, but only if specified together with the first option; and the last two options respectively will slow program down and emulate a miscompile. This way, the automatic tool should be able to find combination of `-DFAST` and `-DEVEN_FASTER` as the best one, and filter out `-DSLOW` and `-DMISCOMPILE`.

Also our program will output "internally measured performance" (as it might have output *fps* value if it were a graphical application). In this example we will be maximizing this "performance" (though we as well might have opted for minimizing runtime). The program will also have "built-in" correctness verification routine, that will "compute" hash of its output. Again, for graphical application it might have computed output hash of the last frame in a framebuffer.

```
int main()
{
#ifdef MISCOMPILE
  printf("Segmentation fault.\n"); exit(139);
#endif

// Program main loop (the number of iterations adjusted accordingly)
for (...) { }

// Output "internally measured performance (e.g. fps)"
#ifdef SLOW
  printf("5 - slow\n");
#else
#ifdef FAST
#ifdef EVEN_FASTER
  printf("90 - very fast\n");
#else
  printf("30 - fast\n");
#endif
#else
  printf("10 - normal\n");
#endif
#endif

  // Print an "output hash" value
  printf("HASH=123\n");
  return 0;
}
```

4.2 Setting Up Scripts for Tutorial

First of all we need to write `$APP_DIR/bin/init-pool` script, which TACT will call to build tutorial program source. This script will be provided the following three environment variables as an input:

- `POOL_DIR` – path to build pool
- `TEST_DIR` – path to test directory (in our case `$TACT_DIR/apps/tutorial/tests/default`)
- `FLAGS` – compiler flags

The sample `init-pool` script looks like this:

```
# A directory with test-private configs
CONFIG_DIR="$TEST_DIR/private-etc"

# Include file with compiler configuration. It sets some compiler variables
# (e.g. $C_FLAGS, $LD_FLAGS, adds proper -I and -L options if needed, etc)
. $CONFIG_DIR/build-config
```

```
# Build binary with given $FLAGS
$CC $FLAGS $TEST_DIR/src/test.c -o $POOL_DIR/run/tutorial
```

For large applications this script would contain `configure --prefix=$POOL_DIR/run && make && make install` commands.

Second script, `$APP_DIR/tests/default/bin/rebuild-pool`, is invoked when TACT needs to rebuild pool with new compiler options (passed as `$FLAGS`). For our simple application, it will coincide with `init-pool` script, however, ideally it should contain commands to rebuild only the minimum number of files in order to optimize build time. The usual technique to achieve that is to rebuild only those object files containing hot functions that matter most for the performance. To do that, those object files should be removed, and then `make` command will rebuild only the missing files.

The reason why this script is located at deeper level, in `default` test directory, is that different tests may have different hot functions, therefore, this script is test-specific.

Next script is `$APP_DIR/tutorial/bin/compute-binary-hash`. This script computes hash of an application binary and is used by TACT to determine whether different compiler options result in identical binaries (this hash is only used at the post-analysis reducing of the result, and identical binary still may run multiple times during tuning). We will use `md5sum` to compute hash:

```
# POOL_DIR - path to pool

# This script should print out a single binary hash for test binaries,
# e.g. md5sum of .so file of the library under optimization

md5sum $POOL_DIR/run/tutorial | awk '{ print $1 }'
```

Then, a script is needed to execute our application on a target test board. This script is located in `$APP_DIR/tests/default/private-bin/target-run-single-test`. For tutorial application this script can look like this:

```
# POOL_DIR - path to pool

# This script should run the test using the executables from $POOL_DIR/run
$POOL_DIR/run/tutorial
```

Technically, TACT calls its interface script `$APP_DIR/tests/default/bin/target-run-test`, which can be customized, but we will use its standard variant that is designed for running test suites and utilizes `target-run-single-test` as a helper that runs single benchmark from a test suite, while benchmarks that constitute a test suite are listed in `private-etc/test-set`. Since it's a user script, its path is prefixed with *"private"*. Since we have the only test, our `private-etc/test-set` will contain the only test name, *"tutorial"*.

Also, we need a script for parsing results of the test. This script is also test-specific and is located in `$APP_DIR/tests/default/bin/parse-results.rb`.

This script reads test output from its standard input, parses it and writes XML file of certain format to standard output. The easiest way to write a parser script is to extend Ruby base class `ResultsParserBase` and override `user_handle_single_line` method for our test. This method takes each single line of test output as parameter and determines beginning of output of a new benchmark from a test suite[1], test name, and test result by parsing lines using regular expressions. It should call `note_next_test` when it sees first line of the next benchmark output, `set_current_test_name` when it finds out a current benchmark name, and `store_current_value` with appropriate parameter when it has parsed a performance value or output hash. The base parser class will take care about the rest and output XML file in an appropriate format.

```
#!/usr/bin/ruby

class AppResultsParser < ResultsParserBase
    def user_handle_single_line(line)
          # Extract test value and begin new test
          matches = line.match(/^(\d+)\s-\s(slow|normal|fast|very fast)/)
          if matches && matches.length == 3
             # When encountering the first line of the output of new test,
             # switch test number
             note_next_test

             # In tutorial we have only one test name, so set it to 'tutorial'
             set_current_test_name("tutorial")

             # first matched value is a "performance value" -- result of a
                  test run
             res = matches[1].to_f
             store_current_value('value', res)
          end
     end
end
```

Finally, we need to set up configuration files for application tuning:

- `$APP_DIR/tests/default/etc/test-descr.xml` – Test description.
  This XML file describes names of benchmarks constituting this test, their weights, method for calculating composite value for a test suite and whether optimal value is the smaller or the greater one. For our example summary method is the arithmetic mean, greater value is better (if we were optimizing for time, it would be smaller) and we have only one test `tutorial` with weight 1:

```
<benchmark_description
    summary_method="mean"
```

[1] In documentation we assume that an *application* may have several *tests*, and those could be a *test suites*, consisting of multiple *benchmarks*. In TACT functions naming here we use *benchmark* and *test* terms interchangeably.

```
    greater-is-better="true">
    <test name="tutorial" weight="1.0"/>
</benchmark_description>
```

- `$APP_DIR/tests/default/etc/tuning.conf` – configuration file with compiler options to be tuned for TACT. In our case we put in here four defines: `-DFAST`, `-DEVEN_FASTER`, `-DSLOW` and `-DMISCOMPILE`. The header of this file (all the strings before *flags* section) is a configuration options for TACT.

```
<?xml version="1.0"?>
<config>
    <prime command=""
          flags="-O2 -mfpu=neon -mfloat-abi=softfp" />

    <baseline description="-O1 -mfpu=neon -mfloat-abi=softfp"
             command=""
             flags="-O1 -mfpu=neon -mfloat-abi=softfp" />

    <baseline description="-O2 -mfpu=neon -mfloat-abi=softfp"
             command=""
             flags="-O2 -mfpu=neon -mfloat-abi=softfp" />

    <baseline description="-O3 -mfpu=neon -mfloat-abi=softfp"
             command=""
             flags="-O3 -mfpu=neon -mfloat-abi=softfp" />

    <baseline description="-Os -mfpu=neon -mfloat-abi=softfp"
             command=""
             flags="-Os -mfpu=neon -mfloat-abi=softfp" />
    <build_config name="TARGET" value="arm-unknown-linux-gnueabi" />
    <build_config name="HOST" value="arm-unknown-linux-gnueabi" />
    <build_config name="CC" value="@static_params[:compiler] +
         '/bin/arm-unknown-linux-gnueabi-gcc'" />
    <compiler value="/home/user/x-tools/gcc-4.6.2" />
    <populations>
        <join_results name="local">
            <population board_id="localhost"/>
        </join_results>
    </populations>
    <population_size value="10" />
    <mutation_rate value="0.07" strategy="uniform_replace"/>
    <migration_rate value="0.2" />
    <greater_is_better value="false" />
    <repetitions value="1" />
    <do_profiling value="false" />
    <num_generations value="10" />
    <num_testboards value="1" />
    <threads_per_testboard value="4" />
    <measure value="performance"/>


    <!-- A list of flags that will be "evolved" by TACT -->
    <flags>
```

```
    <!-- These flags can have value "-D..." or just an empty string. -->
    <flag type="enum" value="-DSLOW | "/>
    <flag type="enum" value="-DFAST | "/>
        <flag type="enum" value="-DEVEN_FASTER | "/>
        <flag type="enum" value="-DMISCOMPILE | "/>
    </flags>
</config>
```

In `flags` e.g. each compiler flag is declared as a separate tag. There are several possible flag types. Type *simple* used in this example means a flag that can have two possible values: the one specified in *value* attribute (when turned on) and an empty string (when turned off).

```
<flag type="gcc_flag" value="-fmodulo-sched"/>
```

The optimization flag of *gcc_flag* type should start with `-f` or `-m` prefix and has two possible values: `-f`*value* and `-fno-`*value* (the same applies to `-m` prefix). The negation is created automatically in TACT. For example, `-fmodulo-sched` can be transformed to `-fno-modulo-sched`.

```
<flag type="enum"
     value="-fira-region=one|-fira-region=all|-fira-region=mixed"/>
```

Type *enum* means TACT will use exactly one of the values (separated by "|") of the specified string.

```
<flag type="param" value="--param large-stack-frame" default="256"
     min="200" max="312" step="8" separator="=" />
```

Type *param* allows to tune GCC parameters. The declaration syntax is self-explanatory. However, some GCC parameters have stronger restrictions, and can be declared only as *enum* type:

```
<flag type="enum" value="--param l1-cache-line-size=0|--param
     l1-cache-line-size=16|--param l1-cache-line-size=32|--param
     l1-cache-line-size=64|--param l1-cache-line-size=128|--param
     l1-cache-line-size=256"/>
```

4.3 Starting Tuning

First, if you want to work with a test board, you need to write configuration file for accessing a test board (a sample file can be found in Chapter 2).

On your host machine, change directory to `$TACT_DIR`, and run the following commands:

```
‘bin/set-env‘
cd apps/tutorial/tests/default/
```

Similarly, the first command will set up the tuning environment on host, and the second changes dir to test directory, so `tact` knows which test we're tuning.

The following commands will prepare `log` directory for current run, and initialize pools building them with baseline flags (typically `-O2`):

```
tact init-test
tact init-pools
```

Now, we will run test application on each board and save reference output hashes and reference results in `log/current/ref/`:

```
tact reference-runs
```

Finally, we can start the tuning with the following command:

```
tact start-tuning
```

For results analysis, please proceed to Step 9 in Chapter 5 (tuning for Pareto-optimal set of parameters).

Chapter 5
Tutorial: Pareto-tuning of x264 application

This tutorial shows how to perform Pareto-based tuning for performance / code size tradeoff.

1. **Set up the required tools.** Please be sure to follow instructions in Section 2.1 to install all the tools and modules required by TACT system.
2. **Set up the environment.** Change dir to your TACT top installattion directory and run `‘bin/set-env‘` to set `$TACT_DIR` and other required environment variables.
3. **Set up the networking.** There is a quick way to start the tuning on a single host machine that will be used for both compilation and execution. In this case you may now skip to Step 5. Otherwise please follow installation instructions in Chapter 2 to set up NFS, SSH and public/private keys on your testboard(s) and tuning host machine, and proceed to the next step.
4. **Configure access to hardware nodes involved in tuning.** Edit the file `$TACT_DIR/task_manager/system.xml` to match your configuration. For example, for one server and one test board directly connected to it you can create the following configuration:

```
<system_config>
<main_server id="mypc" network_name="localhost"/>
<board id="beagle1" network_name="beagle" connected_from="mypc"/>
</system_config>
```

5. **Go to test directory.** Change dir to x264 test direcory:

```
cd $TACT_DIR/apps/x264/tests/default
```

6. **Set up test application's configuration.** Edit `etc/tuning.conf` and set `board_id` tag value to match your testboard id from `system.xml` (or, if you have opted for single machine to do the tuning at Step 3, then just set `id` to `"localhost"`). Also set `measure` value to `"pareto"`, and adjust `compiler` parameter value to point at your GCC's installation base directory (for system default compiler it should be `"/usr"`). Also check that

all other parameters suit your tuning goals. The configuration parameters are described in Chapter 7 in section with test-specific scripts.

7. **Set up build pools and do reference runs.** Run the following commands (please be sure to run them from `./tests/default` directory itself, not from its subdirectories):

```
tact init-test
tact init-pools
tact reference-runs
```

This will build x264 with each set of compiler flags specified in `baseline` tag of `tuning.conf` and perform reference runs. The code size and performance for reference runs will be remembered for later comparison, as well as result hash for detecting miscompiles.

8. **Start tuning.** Finally, start tuning:

```
tact start-tuning
```

During the tuning process you can watch the progress and current results in directory `$TACT_DIR/apps/x264/tests/default/log/current/archives`. Files `generationN_K/output.pdf` show current Pareto front (N and K correspond to generation and population numbers, respectively). Also in `joint_archives` directory there's the same data joined from all populations. Compiler flags and results for Pareto-front can be found in `*.xml` files in the same directory. The results on Pareto-front are clustered by using *k-means* method, and are saved to `log/current/best` directory.

9. **Create tuning report.** To create tuning report in `xlsx` format, run the following command:

```
tact verify
```

For performance tuning, this command picks the best tuning results from all generations. In case of Pareto-tuning, the best Pareto-front is built at the tuning time. ~~Then, it verifies the best flags combinations by running them once again on the testboards~~ (currently verification is not supported). In both cases, at this stage the report for best results is generated in `log/current/results.xlsx`. For Pareto-tuning, it contains the performance and code size for each solution in resulting Pareto-front, as well as comparison with baseline flags specified in `tuning.conf`.

10. **Reduce result by binary-hash.** Refine the tuning results by removing all compiler flags that do not affect the application binary:

```
tact reduce-flags
```

The results will be placed in `log/current/best-reduced/*.xml`

11. **Reduce by "score".** In case of Pareto-tuning, this procedure will try removing those options without which the solution is still Pareto-optimal. I.e. it will strive to move the solution in bottom-left direction on Pareto-graph by dropping as much options as possible. This process is started by this command:

```
tact reduce-by-score
```

After all flags that do not make the solution worse in the sense of Pareto are removed, it continues removing the options that contribute the least to Pareto-optimality (i.e. at each step allowing drift in top-right direction by the minimum amount). This way, the most important options in the sense of Pareto are retained, and get removed only at the latest stages of the process. The results of this process are saved here:

```
log/current/best-byscore-reduced/*.xml.report
```

12. **Results analysis.** By analyzing `*.xml.report` files, the most important compiler flags can be identified. The first table in the report reflects the proccess of dropping a locally least useful option at each step, in the reverse order ("least useful" is either for Pareto-optimality or for the performance, depending on the tuning goal). Each line in that table corresponds to flags set that includes all flags starting from the top of the table to that line. The best reduced flag combination is shown in the line marked with an asterisk (i.e. dropping other options than those below that line causes degradation). The table can be also read in natural top-down order: each line in the table adds one compiler flag, and the numbers show the effect of adding that flag compared to the previous line and to -O2. Usually, it takes only first few flags from this table to retain most of tuned performance, while keeping it clear where the improvement comes from.

Chapter 6
Tools for Analysis of Tuning Results

6.1 Verifying Tuning Results

After tuning it is possible to verify best tuned results. It's only needed to initialize testboards, create reference runs in current configuration (checker needs output hashes to compare), and to run

```
tact verify <LOG_DIR_NAME>
```

where a parameter is a name of subdir of `log/` with tuning results. Verifier will select best 5 options sets for each board and re-run the test compiled with this options. In `log/log_dir_name` the next folders will be created:

- `best`
  This folder will contain text files with best option sets for each testboard. Also XML files for runs with best options are copied here from `runs` folder.
- `verified`
  This directory will contain XML files created while verifying best option sets. Each option set is verified on each testboard.
- `report`
  This directory will contain report tables in XML format (compatible with MS Excel). There will be given one table per testboard using "old" tuning results and one table for each best option sets on each testboard containing verifying results.

6.2 Single Run

This feature can be used when all configuration and reference-runs are done.
`tact single-run compiler_options testboard_num`
This command allows to build application using specified options and run it

on specified testboard. XML file with run results is placed at `log/current/single.xml`

6.3 Running Using Option Sets From a Text File

This feature can be used when all configuration and reference-runs are done.
`tact verify-file filename`
This command allows to build application using options written in text file, and then run them on testboard(s). One string of text file should contain one option set. The result XMLs are placed in `verified` directory, and result table (in XML compatible with MS Excel format) is placed in `report` directory.

6.4 Diagnosing Miscompiles

With TACT you can find out which of GCC options may produce miscompiled or very slow code and then exclude this options from search. This can be done by running `process-log results.log | sort -gk2 | less`. This combination will parse results log, count total and average score for each option and then sort options by their average score. Total score of option is sum of results of tests which were built with this option. Average score counts similarly. If aplication miscompiled then its score would be 1000000000. So if after x runs, where x pretty big, (200 is pretty big in many cases) score of some option equal to $x * 1000000000$, then this option is bad and better to exclude this option from tuning.conf. For example dom test of webkit always fail with segmentation fault if it was built with `-fno-reg-struct-return`.

6.5 Checking config and compiler

TACT includes command `tact check` for checking some useful cases:

- unknown options in `tuning.conf`
- binary hash is equal for all pools
- -O2 and -O0 have different hashes for init-pool and rebuild-pool
- compilation fails with unknown option
- compute-size cannot calculate size if compilation fails

This command helps to avoid some errors on tuning stage.

6.6 Evovis

Evovis (evolution visualizer) is a script that reads a log produced by TACT and plots the distribution of flags values among the populations.

6.6.1 Requirements

Evovis is written in perl and requires the `Chart::Gnuplot` perl module (which requires gnuplot itself). To install gnuplot on debian you can do:

```
aptitude install gnuplot
```

To install `Chart::Gnuplot` (The second line should be typed in the CPAN shell):

```
perl -MCPAN -we "shell"
install Chart::Gnuplot
```

6.6.2 Example

The simplest way of running this script is as follows:

```
evovis -o example /path/to/results.log
```

This command will generate a lot of png files and several html files in the current directory. The main html file will be called `example.html` and will contain a link for each flag.

This line will produce a better result in some sense:

```
evovis -o example --format svg --percent --noyscale --noxscale --alltics
     --nounused --cmaxscale -c $TACT_DIR/etc/evovis-params.conf
     /path/to/results.log
```

6.6.3 Options

Generally evovis should be run like this:

```
evovis [OPTIONS] /path/to/results.log
```

`--help`
Print a brief help.

`-o output-name`
Set the prefix of generated file names.

`-c configuration-file`
Use this configuration file. Configuration file is a list of patterns. Its syntax is described in the next subsection. By default it uses `$TACT_DIR/etc/evovis.conf`

`-v`
Be verbose. It is useful for debugging configuration files or the script itself.

`--percent`
Use percent of population instead of count.

`--noyscale`
Don't scale Y axis.

`--noxscale`
Don't scale X axis.
These options can make each plot uglier but they simplify the comparison of different plots. It is also recommended to use `--alltics` for this purpose.

`--alltics`
Display all tics on X axis.

`--nounused`
Don't add tics for unused values. (e.g. if we have two values of some parameter, 128 and 64, then all values between them will be on the X axis without this flag)

`--nocscale`
Don't scale color. This option makes the brightest color correspond to 0.

`--cbest`
Use the best individual (instead of the average fitness) to color the bars.

`--clocal`
Makes the meaning of color local to specific population and generation. It also makes harder to compare different plots.

`--cplacelocal`
Makes the meaning of color local to specific population but not generation.

`--cmaxscale`
Makes black correspond to average+(average−best). Seems to be meaningless with `--nocscale`. The effect should be close to the effect of `--gamma`.

`--onegen`
Merge generations.

`--onepop`
Merge populations

`--format format-eg-png`
Set the format of images. The default value is png.

`--size "w,h"`
`"1,1"` is a good choice

`--gamma g`
Gamma correction of bar color.

`--font "Arial,12"`
Font.
`--ticfont "Arial,12"`
Tics font.

6.6.4 Configuration File

Configuration file is a list of patterns, against which the flags will be matched. It may also contain rules that define which part of a flag is its name and which is its value.

For example, the default configuration file looks like this:

```
--param ([^=]+)=(\S+) -> $1 $2
([^= ]+)=(\S+) -> $1 $2
-fno-(\S+) -> -f$1 no
-f(\S+) -> -f$1 yes
-DNO_(\S+) -> -D$1 no
-D(\S+) -> -D$1 yes
```

There are patterns on the left hand side of the `->` and rewriting rules on the right hand side. `$1` corresponds to the first pair of parentheses in the pattern and `$2` to the second. The flag should be rewritten to something that looks like `name value`, i.e. a name and a value delimited by a space. So neither name nor value can contain space symbols. The patterns are ordinary perl regular expressions.

The substrings of a compile string correspondig to the patterns will be searched and eliminated from it until there is no matching substring left in the compile string. So if you want to ignore some flags, just make sure there is no pattern it can be matched against.

There is a configuration file that doesn't look at binary flags (`-f...` and `-fno-...`, it is named `$TACT_DIR/etc/evovis-params.conf`

6.7 Reducing the Resulting Compiler Option Set

The set of flags produced by TACT may be redundant: some flags may do nothing but they will be present in compile string anyway. This script tries to reduce the sets of flags for best runs by eliminating the flags whose presence doesn't affect the hash of binary files. It tries to achieve this by performing as few as possible compilations but it still may take a long time (about a day).

This script reduces the best flags sets, so you should run the verifier before running this script. To launch the reducing, type:

```
tact reduce-flags
```

It will take the best runs from `$TEST_DIR/log/current/best`, recommendations from `$TEST_DIR/log/current/reduce-flags-recommendation`, cache from `$TEST_DIR/log/current/reduce-flags-cache` and the reference run from `$TEST_DIR/log/current/ref/1.xml`. Then it will reduce compile strings of best runs and write the results to `$TEST_DIR/log/current/reduce-flags-results`. It will also put xml files with compilation results to `$TEST_DIR/log/current/best-reduced`.

Recommendation file is just a list of pairs `uselessness flag-name`, one pair per line. The uselessness of each flag will be adjusted and written to this file after each run of this script to help next runs to eliminate useless flags more quickly.

Cache directory contains xml files that correspond to compilations made by this script. Sometimes it may become out of date, so you should delete the cache if the script asks you to.

The results will look like this:

```
BEFORE: (hash) /path/to/original.xml
a huge list of flags

AFTER: (hash) /path/to/new.xml
a small list of flags

Additional gcc runs: 1000
Total runs (including cached): 100500
```

Chapter 7
TACT Scripts Reference

7.1 Script for Importing SPEC2000 Tests

The folder `$TACT_DIR/specstrap` contains scripts that import SPEC 2000 tests into TACT for tuning. To do that, the following steps are required:

- Configure TACT environment by running `‘bin/set-env‘`
- Export `SPEC_DIR` environment variable with a path to your SPEC dir. e.g. `/home/username/spec2k`
- Run script `run.sh`
- Now you can find SPEC2K tests in `tact/apps` dir.

7.2 Setup script

`$TACT_DIR/task_manager/system_setup` – this script can automatically setup nfs and TACT user on all boards and intermediate servers. *The script is deprecated, and won't be supported in the future.*

This script has 3 parts:

- `system_setup --export_nfs` – configures nfs on server.
- `system_setup --create_users` – creates users on all intermediate servers and target boards.
- `system_setup --create_mounts` – configures target boards and makes desired mounts and directory structure on them.

7.3 Scripts Reference

All scripts can be divided in two groups – application specific scripts, that are common for the whole application, and test specific scripts, that can differ on each test.

Main configure file for all scripts – `$TEST_DIR/etc/tuning.conf`. Its options described in Section 3.4 (main tuning parameters) and at the end of Section 4.2 (compiler flags).

7.3.1 Application-Specific Scripts

- `$APP_DIR/bin/`compute-binary-hash – compute binary hash of application.
  Parameters:
  `POOL_DIR` – path to pool
  Result:
  As a result this script should print to stdout md5 or other hash of application binary or library (*.so).
- `$APP_DIR/bin/`init-pool – initial build of application
  Parameters:
  - `POOL_DIR` – path to pool
  - `TEST_DIR` – path to test
  - `FLAGS` – compiler flags

  Result:
  This script should take application sources from `$TEST_DIR/src` build them in `$POOL_DIR/build` and then install to `$POOL_DIR/run`. Also this script should include `build-config` script from `$TEST_DIR/private-etc`, which contains build-time options.
- `$APP_DIR/bin/`init-shared – initial build of shared libraries
  Parameters:
  - `APP_DIR` – path to application
  - `TEST_DIR` – path to test

  Result:
  This script should take shared libraries sources from `$APP_DIR/shared/src` build them in `$APP_DIR/shared/build` and then install to `$APP_DIR/shared/run`. Also this script should include `$TEST_DIR/private-etc/build-config` script, which contains build-time options.
- `$APP_DIR/bin/`init-target – target setup
  Contains actions that should be done on target before start tunung, for example, initialize ramdisc, prepare data for test or move cursor to the bottom of screen.

- `$APP_DIR/bin/`verify-results – verify test results for errors and/or miscompile. Usually you don't need to edit this script.

7.3.2 Test-Specific Scripts

- `$TEST_DIR/bin/`compute-binary-hash – compute binary hash of application
  Parameters:
  `POOL_DIR` – path to pool
  Result:
  As a result this script should print to stdout md5 or other hash of application binary or library (*.so). Usually it's symlink for `$APP_DIR/bin/compute-binary-hash`.
- `$TEST_DIR/bin/`compute-size – compute binary size of application
  Parameters:
  `POOL_DIR` – path to pool
  Result:
  As a result this script should print to stdout size of application binary or library (*.so). Usually it's symlink for `$APP_DIR/bin/compute-size`.
- `$TEST_DIR/bin/`parse-results.rb – class on Ruby for parsing results.
  This class extends `ResultsParserBase`. You should rewrite `user_handle_single_line` function for your test. This function takes each single line of test output as parameter and should determine when new test begin, test name and test result by parsing lines with regular expressions.
- `$TEST_DIR/bin/`rebuild-pool – rebuild pool with new options.
  Parameters:
  - POOL_DIR – path to pool
  - FLAGS – list of flags

  Result:
  This script should delete sufficient object files and/or binaries and then rebuld pool in `$POOL_DIR` with given flags.
- `$TEST_DIR/bin/`target-run-test – run test on target.
  Parameters:
  - POOL_DIR – path to pool
  - REPETITIONS – number of repetitions

  Result:
  This script runs test from `POOL_DIR/run` on target `REPETITIONS` times. Usually you don't need to edit this script.
- `$TEST_DIR/etc/`tuning.conf – configure file for tuning.
  In this file should be specified compiler options for tuning. Also this file has some TACT params, such as:

  - `COMPILER` – path to compiler installation dir
  - `NUM_GENERATIONS` – number of generations
  - `NUM_POPULATIONS` – number of populations in one generation
  - `NUM_TESTBOARDS` – number of testboards
  - `MEASURE` – type of measure for test (performance, size or pareto)
  - `MIGRATION_RATE` – migration rate
  - `MUTATION_RATE` – mutation rate
  - `POPULATION_SIZE` – number of entities in population
  - `POPULATIONS` – number of populations
  - `THREADS_PER_TESTBOARD` – number of threads for one testboard

  Tact runtime options:

  - `BOARD_ID` – id of board for tuning
  - `DO_PROFILING` – do profiling when building pool
  - `REPETITIONS` – number of repetitions for running test
  - `TIMEOUT` – timeout for running all `REPETITIONS` on target. If `TIMEOUT` not specified then it will be no timeout for test.

  Tact pareto options:

  - `ARCHIVE_SIZE` – size of best entities archive
  - `PARETO_BEST_SIZE` – number of best entities, that will be found by clastering

- `$TEST_DIR/etc/`test-descr.xml – description of tests.
  This XML file describes names of tests, weights of tests and method how to count final result of test set.
- `$TEST_DIR/private-bin/`target-run-single-test – run test on testboard for one time.
  Parameters:

  - POOL_DIR – path to pool
  - TEST_NAME – name of the test

  Result:
  This script should run `$TEST_NAME` from `$POOL_DIR` on target for one time and exit code should be the same with test, so command for running test should be last on script, or exit code should be saved e.g. if you run test as `run_test`:

```
run_test
EXIT_CODE=$?
# some other actions here
...
exit $EXIT_CODE
```

- `$TEST_DIR/private-etc/`build-config – compiler configure file. In this file compiler and common compilation options such as include/library flags, common compiler flags and common configure options are specified.

- `$TEST_DIR/private-etc/`build-functions – useful functions for build shared libraries and pool. Usually you don't need to edit this script.
- `$TEST_DIR/private-etc/`test-set – name of test set.