THE LLVM TEST INFRASTRUCTURE

By Andrei Rimsa Álvares

DCC 888

Testing

1) Do you know any test framework?

2) What a good test infrastructure should provide?

- What can **testing** do for you?

"Program testing can be a very effective way to show the presence of bugs, but it is hopelessly inadequate for showing their absence"

Edsger W. Dijkstra♤

"Testing leads to failure, and failure leads to understanding"

Burt Rutan♧

♤: http://en.wikipedia.org/wiki/Edsger_W._Dijkstra
♧: http://en.wikipedia.org/wiki/Burt_Rutan

Goals of The LLVM Test Framework

- What can the **LLVM test framework** do for you?

1) Identify problems in your passes in early stages

2) Check your passes' performance

3) Verify the quality of your debugging information ***NEW***

Essential Tools

- What do you need to use the LLVM test framework?
  - All software required to build LLVM♤

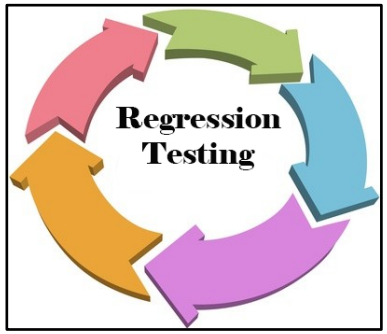
...

  - … and Python 2.5 or later♧

♤: http://llvm.org/docs/GettingStarted.html#software
♧: https://www.python.org

Organization

- The LLVM testing infrastructure contains two major categories of tests



Regression tests

Small pieces of code that test a specific feature or trigger a specific bug in LLVM

Whole programs (test-suite)

Pieces of code that can be compiled and linked into a stand-alone program ready to be executed

```
tmp — b...
$ tree -L 2 -C llvm
llvm
├── CMakeLists.txt
├── CODE_OWNERS.TXT
├── CREDITS.TXT
├── Debug+Asserts
├── LICENSE.TXT
├── LLVMBuild.txt
├── Makefile
├── Makefile.common
├── Makefile.config
├── Makefile.config.in
├── Makefile.llvmbuild
├── Makefile.rules
├── README.txt
├── autoconf
├── bindings
├── cmake
├── config.log
├── config.status
├── configure
├── docs
├── examples
├── include
├── lib
├── llvm.spec
├── llvm.spec.in
├── projects
│   └── test-suite
├── test
├── tools
├── unittests
└── utils
```

Regression Tests

- Regression tests can be used to check if LLVM was compiled properly (after the build with **make**):

```
$> make –C llvm/test
```

or

```
$> make check
```

- If clang was checked out and built in the LLVM tree, than regression tests can be executed simultaneously for both:

```
$> make check-all
```

Regression tests are checked out automatically with LLVM

Example: make check

```
$> make check

... # long time after ...

********************
Testing Time: 235.34s
********************
Failing Tests (5):
    LLVM :: CodeGen/X86/2009-06-05-VZextByteShort.ll
    LLVM :: CodeGen/X86/fma4-intrinsics-x86_64.ll
    LLVM :: CodeGen/X86/fp-fast.ll
    LLVM :: CodeGen/X86/vec_shift4.ll
    LLVM :: CodeGen/X86/vshift-4.ll

  Expected Passes    : 9224
  Expected Failures  : 54
  Unsupported Tests  : 34
  Unexpected Failures: 5
```

The regression tests let's us check if our installation of LLVM compiles correctly the programs. That is a good way to know if a new optimization is introducing unwanted bugs.

Regression Tests

- It is possible to execute tests with Valgrind (Memcheck) by passing parameters in the LIT_ARGS variable

```
$> make check LIT_ARGS="-v --vg --vg-leak"
```

Do you know what is valgrind?

- Or execute individual tests or subsets of a test

```
$> llvm-lit llvm/test/Integer/BitPacked.ll
```

```
$> llvm-lit llvm/test/CodeGen/ARM
```

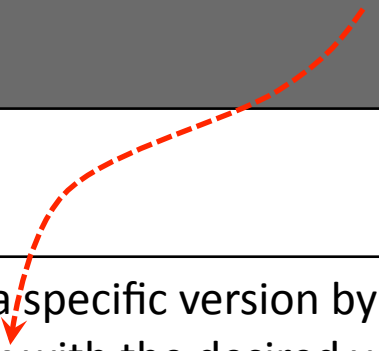
We will not learn how to create regression tests♤

♤: http://llvm.org/docs/TestingGuide.html#writing-new-regression-tests

Whole Program (test-suite)

- LLVM does not include the **test-suite** by default. You must check it out manually in the *llvm/projects* directory

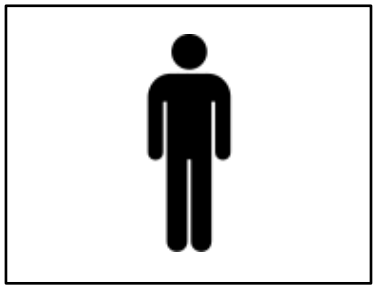
```
$> cd llvm/projects

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



You can checkout a specific version by changing the **trunk** with the desired version, for example **tags/RELEASE_34/final**

The Structure of the Test-Suite

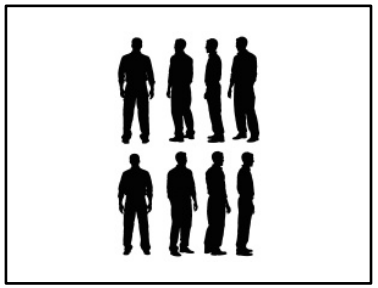
- The **test-suite** itself has an internal organization



SingleSource

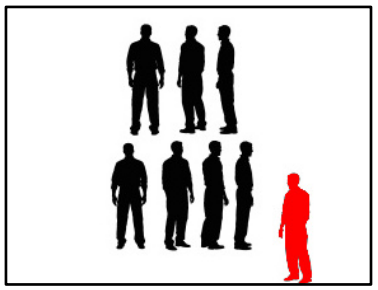
Contains programs that consist of a single source file (small benchmarks)

Can you guess why we have this organization?



MultiSource

Contains entire programs with multiple source files (large benchmarks and whole apps)



External

Contains only Makefiles for building code external to LLVM (such as SPEC benchmarks)

Whole Program (test-suite)

- In order to make the **test-suite** available, you must reconfigure and recompile LLVM

```
$> cd ..
$> ./configure # your options here
$> make –j8
```

- Or, before that, you can setup your external test-suite (such as SPEC 2006)

```
$> cd ..
$> ./configure –with-externals=<directory> # other options here
$> make –j8
```

1) The directory must contain a specific subdirectory name, for example, SPEC 2006 should be placed in the **speccpu2006** subdirectory

2) Check `configure`'s output for a **yes** in the external section, stating that it has found the external test-suite correctly

Executing the **test-suite**

- To execute the test-suite, just go to its directory and type **make**:

```
$> cd projects/test-suite
$> make –j8
```

1) Note that the compiled files will not be placed in this directory structure, but in a temporary directory

2) This step is only required once, unless the test code or configure script changes

- You can also dispatch this process in some subdirectory of the test-suite to narrow the test scope, such as running only **SingleSource** benchmarks

```
$> cd SingleSource/Benchmarks
$> make –j8
```

Executing Other Types of Tests

- In addition to the regular tests, the test-suite module provides a mechanism for compiling the programs in different ways; for example, to run the **nightly tests**

```
$> make TEST=nightly
```

- To run this kind of test, LLVM looks in *projects/test-suite* for a file called **TEST.<value of TEST variable>.Makefile** that can modify build rules to yield different results

```
$~/llvm/projects/test-suite> ls TEST.*.Makefile
TEST.aa.Makefile                TEST.lineinfo.Makefile
TEST.beta-compare.Makefile      TEST.llc.Makefile
TEST.buildrepo.Makefile         TEST.llcdbg.Makefile
TEST.dbg.Makefile               TEST.m2regllcdbg.Makefile
TEST.dbgopt.Makefile            TEST.nightly.Makefile
TEST.example.Makefile           TEST.optllcdbg.Makefile
TEST.ipodbgopt.Makefile         TEST.simple.Makefile
TEST.jit.Makefile               TEST.typesafe.Makefile
TEST.libcalls.Makefile          TEST.vtl.Makefile
```

Generating Test Output

- You can run the tests with the **test** target, which adds per-program summaries to the output that are easily *grepable*

```
$> make TEST=nightly test
```

- Or with the **report** or **report.<format>** (*html*, *csv*, *text* or *graphs*) targets
  - The exact content depends on the type of *TEST* chosen
  - The format is guided by the file in the projects/test-suite called **TEST.<value of TEST variable>.report**

```
$> make TEST=nightly report
```

```
$> make TEST=nightly report.html
```

Can you guess what is a nightly test?

The Nightly Test

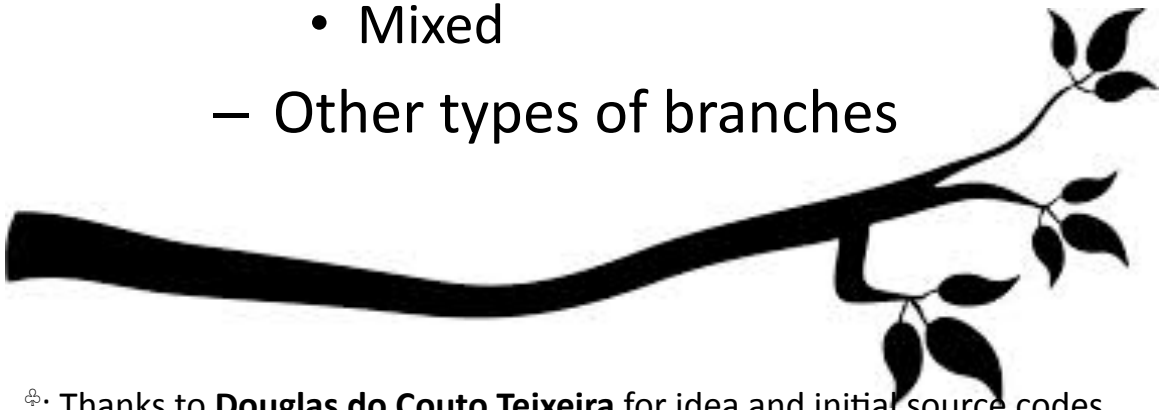
- This is the name of the battery of tests used to check if LLVM is compiling programs correctly[◇].
  - Compares GCC and LLVM
  - Permits to test a new variation of llc
- Today, we can use the nightly tests as a guide to write our own customized tests.
  - We can reuse the Makefile, for instance.

Program	GCCAS	Bytecode	LLC compile	LLC-BETA compile	JIT codegen	GCC	LLC	LLC-BETA	JIT
Bubblesort	0.0058	2320	0.0018	*	0.0030	0.0537	0.0524	*	0.1527
FloatMM	0.0048	2656	0.0044	*	0.0044	0.8071	0.8059	*	0.9073
IntMM	0.0047	2544	0.0032	*	0.0031	0.0013	0.0014	*	0.1086
Oscar	0.0109	4368	0.0049	*	0.0056	0.0038	0.0021	*	0.2128
Perm	0.0044	2448	0.0000	*	0.0001	0.0542	0.0430	*	0.1100
Puzzle	0.0128	7952	0.0026	*	0.0052	0.1678	0.1652	*	0.4764

◇: The name is due to historical reasons. Usually developers code some new optimization and fires new tests that will run throughout the night, while they sleep. In the morning, they check if the new optimization is alright.

Branch Counter Pass

- We shall illustrate the construction of a custom test via a pass that counts the kinds of branches that we may find in a typical program.♧
- We will consider the following types of branches
  - Unconditional branches
  - Branch with comparison instructions
    - Variable/Variable
    - Constant/Constant
    - Mixed
  - Other types of branches



Which type of branch is this?

```
%1:

%2 = load i32* %i, align 4
%3 = icmp sle i32 %2, 500
br i1 %3, label %4, label %45
```

T	F

♧: Thanks to **Douglas do Couto Teixeira** for idea and initial source codes

Branch Counter Pass

```
#define DEBUG_TYPE "branch-counter"

#include "llvm/Pass.h"
#include "llvm/IR/Constants.h"
#include "llvm/IR/Function.h"
#include "llvm/IR/Instructions.h"
#include "llvm/ADT/Statistic.h"
#include "llvm/Support/InstIterator.h"
#include "llvm/Support/raw_ostream.h"

using namespace llvm;

STATISTIC(UnconditionalBranches, "Unconditional branches.");
STATISTIC(ConstantAndVarBranches, "Branches with one variable and one constant.");
STATISTIC(ConstantAndConstantBranches, "Branches with two constants.");
STATISTIC(VarAndVarBranches, "Branches with two variables.");
STATISTIC(OtherBranches, "Other branches.");
STATISTIC(TotalBranches, "Total branches.");

namespace {
  struct BranchCounter : public FunctionPass {
    static char ID;
    BranchCounter() : FunctionPass(ID) {}
    virtual bool runOnFunction(Function &F);
  };
}

char BranchCounter::ID = 0;
static RegisterPass<BranchCounter> X("branch-counter", "Branch Counter Pass");
```

LLVM provides this facility to gather statistics during the execution of passes

BranchCounter.h

Branch Counter Pass

```
#include "BranchCounter.h"
bool BranchCounter::runOnFunction(Function &F) {
  for (inst_iterator I = inst_begin(F), E = inst_end(F); I != E; ++I) {
    if (BranchInst* BI = dyn_cast<BranchInst>(&*I)) {
      // Count this branch in the total.
      TotalBranches++;
      // Count unconditional branches.
      if (!BI->isConditional())
        UnconditionalBranches++;
      // Count the other types os branches
      else if (ICmpInst* CI = dyn_cast<ICmpInst>(BI->getCondition())) {
        bool const_op0 = dyn_cast<ConstantInt>(CI->getOperand(0)) != 0;
        bool const_op1 = dyn_cast<ConstantInt>(CI->getOperand(1)) != 0;
        // Both operands are constants.
        if (const_op0 && const_op1)
          ConstantAndConstantBranches++;
        // Both operands are variables.
        else if (!const_op0 && !const_op1)
          VarAndVarBranches++;
        // A variable and a constant operands.
        else
          ConstantAndVarBranches++;
      // Count other types of branches.
      } else
        OtherBranches++;
    }
  }
  return false;
}
```

BranchCounter.cpp

Branch Counter Pass

- To gather statistics information about the branches in a program, we will load and execute our pass with the **–stats** modifier in a sample test case

```
$> opt -load dcc888.dylib -branch-counter -stats -disable-output
Bubblesort.linked.rbc
===-------------------------------------------------------------===
                  ... Statistics Collected ...
===-------------------------------------------------------------===

 3 branch-counter - Branches with one variable and one constant.
 6 branch-counter - Branches with two variables.
24 branch-counter - Total branches.
15 branch-counter - Unconditional branches.
```

How to automate this process for any test case?

Custom Testing

- In order to make this a custom test, we build a *TEST* case that can be used in the test-suite
- We will extract statistics about the branches
- We will call this *TEST* "**branches**" and we will write the following files that must be placed inside *<LLVM>/ projects/test-suite*

1) TEST.**branches**.Makefile
2) TEST.**branches**.format

Custom Testing

- You can use one of the LLVM's test Makefile as a template:

```
CURDIR  := $(shell cd .; pwd)
PROGDIR := $(PROJ_SRC_ROOT)
RELDIR  := $(subst $(PROGDIR),,$(CURDIR))

$(PROGRAMS_TO_TEST:%=test.$(TEST).%): \
test.$(TEST).%: Output/%.$(TEST).report.txt
     @cat $<

$(PROGRAMS_TO_TEST:%=Output/%.$(TEST).report.txt):  \
Output/%.$(TEST).report.txt: Output/%.linked.rbc $(LOPT) \
     $(PROJ_SRC_ROOT)/TEST.branches.Makefile
     $(VERB) $(RM) -f $@
     @echo "----------------------------------------------------------------------" >> $@
     @echo ">>> ========= '$(RELDIR)/$*' Program" >> $@
     @echo "----------------------------------------------------------------------" >> $@
     @-$(LOPT) -load dcc888$(SHLIBEXT) -branch-counter -stats  \
               -time-passes -disable-output $< 2>>$@
summary:
     @$(MAKE) TEST=branches

.PHONY: summary
REPORT_DEPENDENCIES := $(LOPT)
```

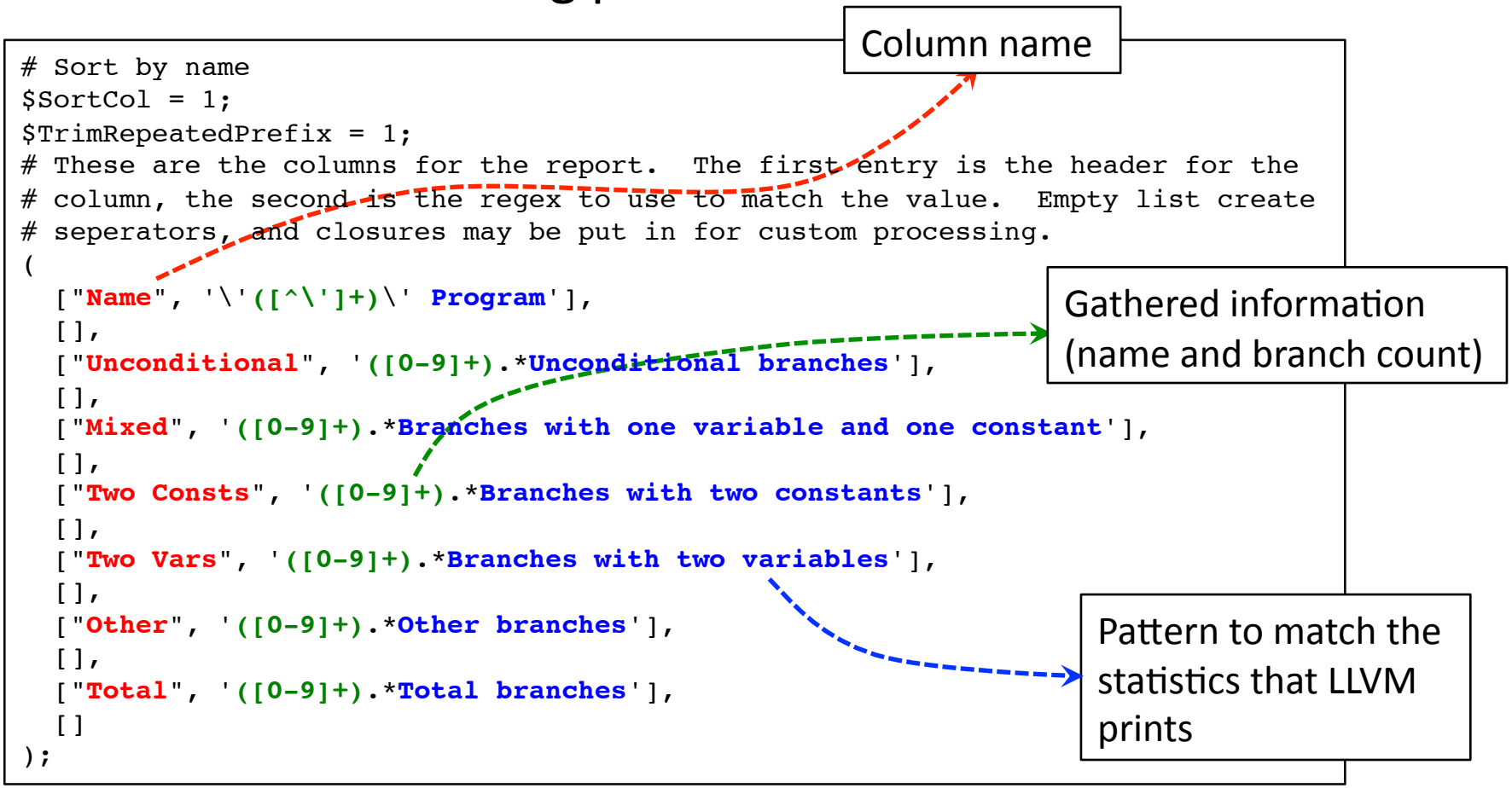
Tells the test-suite how to execute our pass

TEST.branches.Makefile

Custom Testing

Do you understand how we are collecting statistics?

- You can also use one of the LLVM's formaters as a starting point:




```
# Sort by name
$SortCol = 1;
$TrimRepeatedPrefix = 1;
# These are the columns for the report.  The first entry is the header for the
# column, the second is the regex to use to match the value.  Empty list create
# seperators, and closures may be put in for custom processing.
(
  ["Name", '\'([^\']+)\' Program'],
  [],
  ["Unconditional", '([0-9]+).*Unconditional branches'],
  [],
  ["Mixed", '([0-9]+).*Branches with one variable and one constant'],
  [],
  ["Two Consts", '([0-9]+).*Branches with two constants'],
  [],
  ["Two Vars", '([0-9]+).*Branches with two variables'],
  [],
  ["Other", '([0-9]+).*Other branches'],
  [],
  ["Total", '([0-9]+).*Total branches'],
  []
);
```

TEST.branches.report

Reading Information

- You can organize the report in multiple different ways:

Empty list work as blank columns separating data groups.

Read the time taken for each pass (remember to use -time-passes in the Makefile)

```
# Sort by name
$SortCol = 1;

(
# Name
 ["Name:" , '\'([^\']+)\' Program'],
 [],
# Times
 ["Total"   , '([0-9.]+) \([^)]+\)[ 0-9]+Total'],
 ["LoopInf", '([0-9.]+) \([^)]+\)[ 0-9A-Za-z]+Natural Loop Information'],
 ["Scev"    , '([0-9.]+) \([^)]+\)[ 0-9A-Za-z]+Scalar Evolution Analysis'],
 ["DomT"    , '([0-9.]+) \([^)]+\)[ 0-9A-Za-z]+Dominator Tree Construction'],
 [],
# Pass stats:
 ["#tags"   , '([0-9]+).*Number of tagged store instructions'],
 ["#insts" , '([0-9]+).*Number of instructions'],
 []
);
```

Custom Testing

- Now, just execute our newly created *TEST* in the test-suite or any of its subdirectories

Go to the previous slides, and find these columns names there.

```
$> cd test-suite/SingleSource/Benchmarks/Stanford

$> make TEST=branches report

$> cat report.branches.txt
```

Name	Unconditional	Mixed	Two Consts	Two Vars	Other	Total
Bubblesort	15	3	*	6	*	24
FloatMM	19	7	*	*	*	26
IntMM	18	6	*	*	*	24
Oscar	37	8	*	8	*	53
Perm	14	6	*	*	*	20
Puzzle	170	60	*	3	*	233
Queens	19	14	*	*	1	34
Quicksort	19	2	*	10	*	31
RealMM	18	6	*	*	*	24
Towers	20	9	*	1	*	30
Treesort	28	6	*	10	*	44

HTML Report

```
$> make TEST=branches report.html
```

We can also generate the report in HTML format

Name	Unconditional	Mixed	Two Constants	Two Variables	Other	Total
Bubblesort	15	3	*	6	*	24
FloatMM	19	7	*	*	*	26
IntMM	18	6	*	*	*	24
Oscar	37	8	*	8	*	53
Perm	14	6	*	*	*	20
Puzzle	170	60	*	3	*	233
Queens	19	14	*	*	1	34
Quicksort	19	2	*	10	*	31
RealMM	18	6	*	*	*	24
Towers	20	9	*	1	*	30
Treesort	28	6	*	10	*	44

Final Remarks

- The LLVM test infra-structure really makes it easy to carry on very professional experiments.
- It is easy to generate report and collect the most diverse suite of statistics.
- And it is easy to incorporate new benchmarks in the test suite.
  - We can follow the examples already there.