THE LLVM TEST INFRASTRUCTURE

By Andrei Rimsa Álvares
Testing

• 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

1) Do you know any test framework?

2) What a good test infrastructure should provide?

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
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
```
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/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"
```

• Or execute individual tests or subsets of a test

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

Do you know what is valgrind?

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)

  **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)

Can you guess why we have this organization?
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.

```bash
$~/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.

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

♢: 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
---|---

_votes: 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");
Branch Counter Pass

```cpp
#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 of 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++;
      } else
        OtherBranches++;
    }
  }
  return false;
}
```
Branch Counter Pass

• To gather statistics information about the branches in a program, we will load and execute our pass with the \texttt{-stats} modifier in a sample test case

\begin{verbatim}
$> opt -load dcc888.dylib -branch-counter -stats -disable-output Bubblesort.linked.rbc

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.
\end{verbatim}

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:

```makefile
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.libcalls.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 | egrep '======|branch-counter -'

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

Tells the test-suite how to execute our pass
Custom Testing

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

```bash
# 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 # separators, 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'],
    []
);
```

Do you understand how we are collecting statistics?
Custom Testing

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

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

$> make TEST=branches report

$> cat report.branches.txt
```

<table>
<thead>
<tr>
<th>Name</th>
<th>Unconditional</th>
<th>Mixed</th>
<th>Two Consts</th>
<th>Two Vars</th>
<th>Other</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bubblesort</td>
<td>15</td>
<td>3</td>
<td>*</td>
<td>6</td>
<td>*</td>
<td>24</td>
</tr>
<tr>
<td>FloatMM</td>
<td>19</td>
<td>7</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>26</td>
</tr>
<tr>
<td>IntMM</td>
<td>18</td>
<td>6</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>24</td>
</tr>
<tr>
<td>Oscar</td>
<td>37</td>
<td>8</td>
<td>*</td>
<td>8</td>
<td>*</td>
<td>53</td>
</tr>
<tr>
<td>Perm</td>
<td>14</td>
<td>6</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>20</td>
</tr>
<tr>
<td>Puzzle</td>
<td>170</td>
<td>60</td>
<td>*</td>
<td>3</td>
<td>*</td>
<td>233</td>
</tr>
<tr>
<td>Queens</td>
<td>19</td>
<td>14</td>
<td>*</td>
<td>*</td>
<td>1</td>
<td>34</td>
</tr>
<tr>
<td>Quicksort</td>
<td>19</td>
<td>2</td>
<td>*</td>
<td>10</td>
<td>*</td>
<td>31</td>
</tr>
<tr>
<td>RealMM</td>
<td>18</td>
<td>6</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>24</td>
</tr>
<tr>
<td>Towers</td>
<td>20</td>
<td>9</td>
<td>*</td>
<td>1</td>
<td>*</td>
<td>30</td>
</tr>
<tr>
<td>Treesort</td>
<td>28</td>
<td>6</td>
<td>*</td>
<td>10</td>
<td>*</td>
<td>44</td>
</tr>
</tbody>
</table>
We can also generate the report in HTML format.
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.