It was working yesterday!

Investigating regressions with **llvmlab bisect**

$whoami

- DevOps Engineer at Arm
  - Infrastructure for toolchains CI, test and benchmark
- LNT contributor

Getting Started

- When investigating a bug or performance change, finding which **commit** introduced it can be very helpful to understand the problem

- The process of looking into changes and finding which commit causes a given behaviour is called **code bisection**

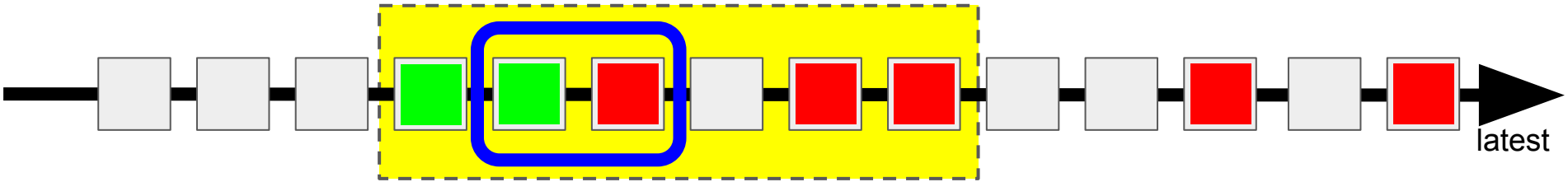
    - In projects with many commits a day (like LLVM, Clang, etc.), bisecting can be a **time consuming** task

    - Automated bisection can use clever ways to navigate you repository, helping to speed up the process

Code Bisection

- Is the iterative process of looking for which commit introduced a given change in behaviour, for example
  - crashes
  - performance regressions
  - when something was fixed, etc.

- Bisecting usually requires
  - A repository that contains sequential relationship metadata
  - A set of checks that help us to decide whether a given version is “good” or “bad”



Automated Code Bisection

- Source control tools commonly offer bisection as a feature
  - `git bisect`
  - `svn bisect`
  - `hg bisect`

- Pros
  - Fine grained bisection
  - Flexibility to build with all the options you want

- Cons
  - Need to rebuild every time
  - Broken revisions

Automated Code Bisection

- As source control tools are agnostic to what is being under bisection, all need to be setup by the user

- In projects with large code bases and many commits every day, like LLVM and Clang, the need of building each revision on demand can make this process time consuming

- **llvmlab bisect** is a tool that speeds up of bisecting LLVM and Clang

llvmlab bisect

llvmlab bisect

- Contributed in 2015 by Chris Matthews and Daniel Dunbar
- Written in Python, specifically for bisecting LLVM related projects
- Documentation here:
  - https://github.com/llvm/llvm-zorg/blob/master/llvmbisect/docs/llvmlab_bisect.rst

llvmlab bisect → Installation

```
$ virtualenv -p $(which python2.7) v
$ . v/bin/activate
```

optional

```
$ git clone https://github.com/llvm-mirror/zorg.git
$ cd zorg/llvmbisect
$ python setup.py install
$ llvmlab
Usage: llvmlab command [options]

...
```

llvmlab bisect → Basic Usage

```
$ llvmlab bisect <options> <test case>
```

1. obtain a build from the build cache
2. create a sandbox
3. run the test case (predicates)
4. navigate through versions and repeat the process to find the commit causing the issue

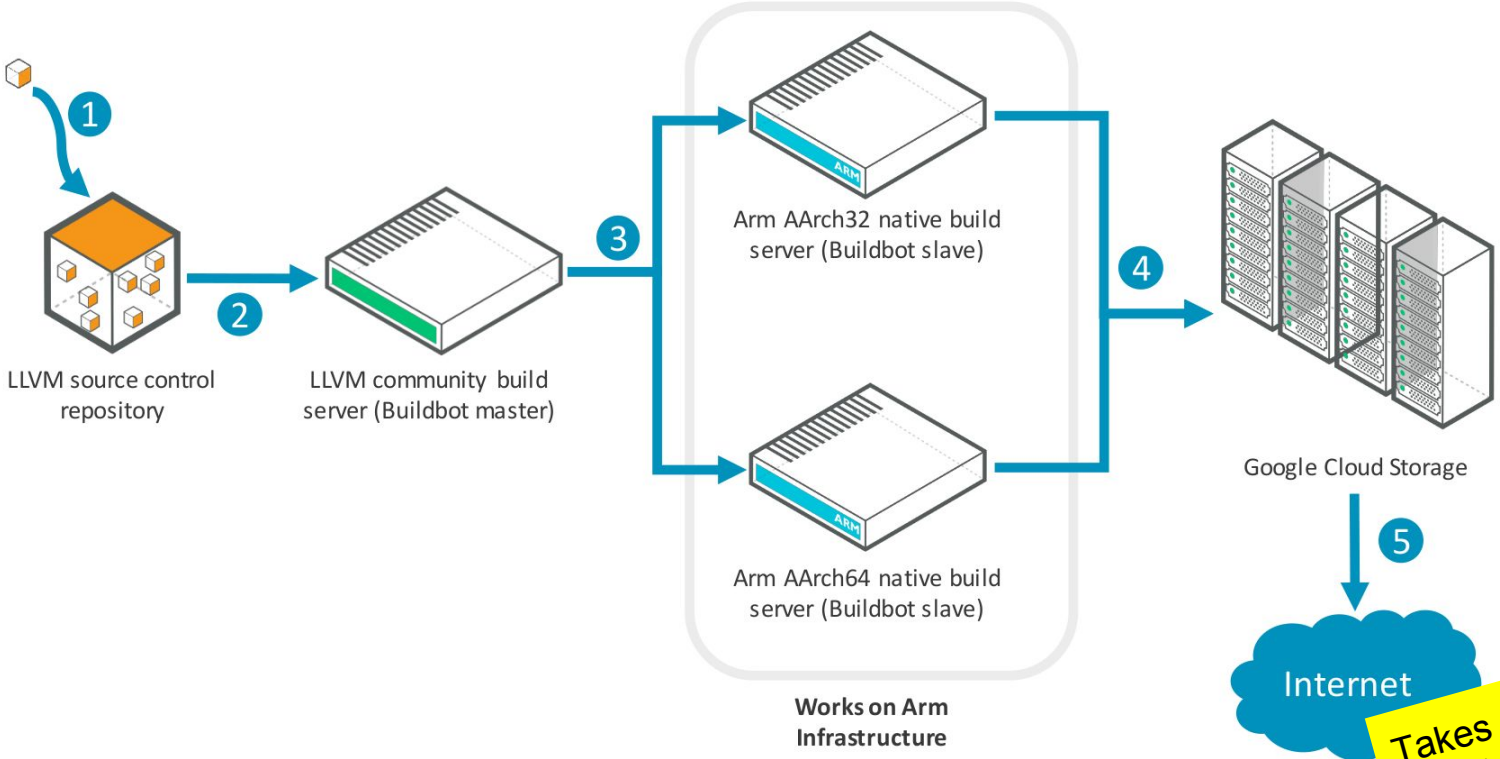
llvmlab bisect → Concepts

- **Build cache**
- Sandbox
- Predicates
  - Variables
  - Test filters

llvmlab bisect → Build Cache

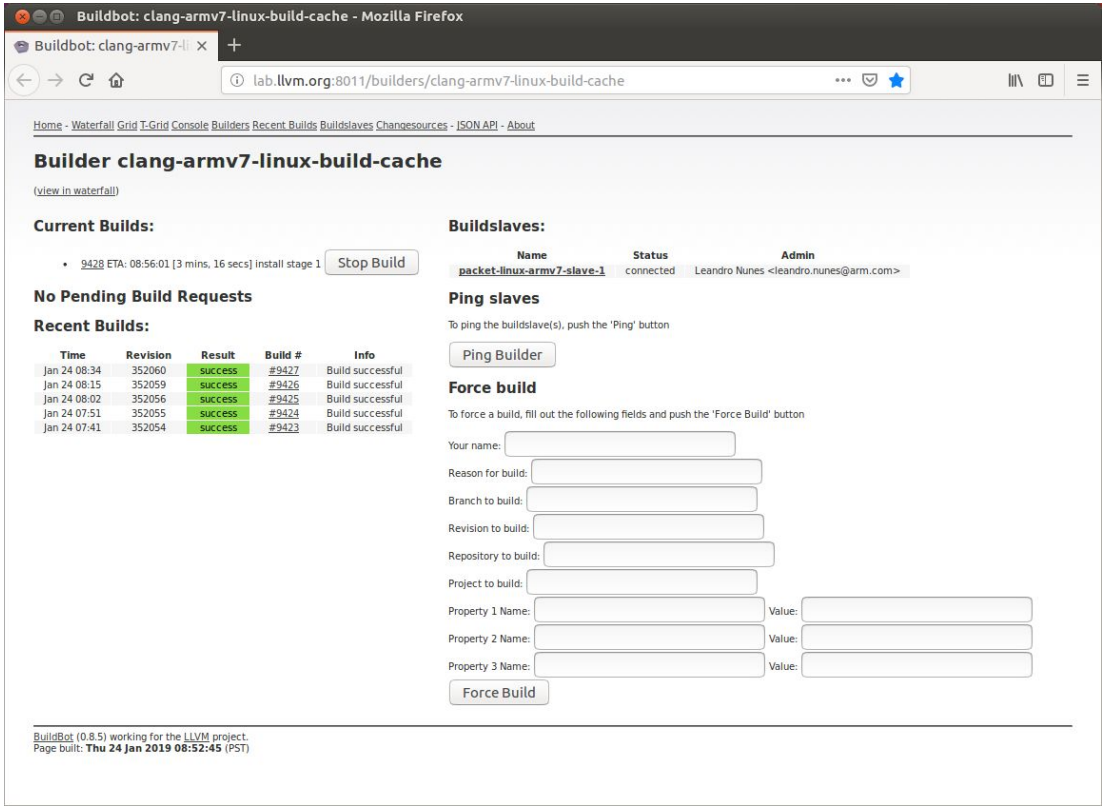
- The build cache hosts pre-built packages, generated by CI systems like Jenkins and Buildbot
- Various types of packages grouped in different **builders** (x86, Armv7, AArch64, etc.)
- Packages are stored in Google Cloud Storage
- Armv7 and AArch64 native toolchains were recently introduced
  - http://lab.llvm.org:8011/builders/clang-armv7-linux-build-cache
  - http://lab.llvm.org:8011/builders/clang-aarch64-linux-build-cache

llvmlab bisect → Populate Build Cache



https://community.arm.com/tools/b/blog/posts/accelerating-open-source-llvm-development

llvmlab bisect → Populate Build Cache



llvmlab bisect → Explore Build Cache

- Listing existing “build names” or “builds”

```
$ llvmlab ls
clang-aarch64-linux
clang-armv7-linux
clang-cmake-aarch64
clang-cmake-armv7a
clang-cmake-mips
clang-cmake-mipsel
clang-stage1-configure-RA
clang-stage1-configure-RA_build
clang-stage2-Rthinlto
clang-stage2-cmake-RgTSan
clang-stage2-configure-Rlto
clang-stage2-configure-Rlto_build
clang-stage2-configure-Rthinlto_build
```

llvmlab bisect → Build Cache

- Using a specific builder

```
$ llvmlab bisect -b clang-aarch64-linux <test case>
```

llvmlab bisect → Concepts

- Build cache
- **Sandbox**
- Predicates
  - Variables
  - Test filters

llvmlab bisect → Sandbox

- Each revision pulled from the build cache is extracted on a temporary directory
  - This temporary directory is the “sandbox”
- By default, sandboxes are kept under /tmp and deleted just after the test execution on that specific revision is completed
- It is possible to preserve sandboxes by using “-s <directory path>” option on command line

llvmlab bisect → Sandbox

- Using a custom sandbox

```
$ llvmlab bisect -s ~/llvm_bisect_sandbox <test case>
```

llvmlab bisect → Concepts

- Build cache
- Sandbox
- **Predicates**
  - **Variables**
  - **Test filters**

llvmlab bisect → Predicates

- The commands used to guide your bisecting process
- Can be provided by command line or as a shell script
  - Can also use any other command line tool available on your local system

```
$ llvmlab bisect "%(path)s/bin/clang test.c"
```

llvmlab bisect → Variables

- Used in your test script to point to values that will be replaced by the bisecting tool

- These are all the variables currently available
  - **sandbox:** the path to the sandbox directory.
  - **path:** the path to the build under test.
  - **revision:** the revision number of the build.
  - **build:** the build number of the build under test.
  - **clang:** the path to the clang binary of the build if it exists.
  - **clang++:** the path to the clang++ binary of the build if it exists.
  - **libltodir:** the path to the directory containing libLTO.dylib, if it exists

llvmlab bisect → Variables

- When provided via **command line**, they will be used as named arguments on Python printf() syntax
  - `"%(path)s"`
  - `"%(sandbox)s"`
  - `"%(revision)s"`

- When used in a **shell script**, they will be injected as `$TEST_<VAR NAME>`
  - `${TEST_PATH}`
  - `${TEST_SANDBOX}`
  - `${TEST_REVISION}`

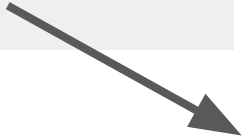
llvmlab bisect → Variables

- Using a variable on command line

```
$ llvmlab bisect "%(path)s/bin/clang crash.c"
```

- Using a variable on shell script

```
$ llvmlab bisect bash run.sh
```



```
#!/bin/bash

${TEST_PATH}/bin/clang crash.c
```

llvmlab bisect → Test Filters

- Extra values to be used to evaluate in the bisection process

- The available filters are
  - result: boolean value, `True` when the current predicate result is PASS
  - user_time
  - sys_time
  - wall_time

llvmlab bisect → Test Filters

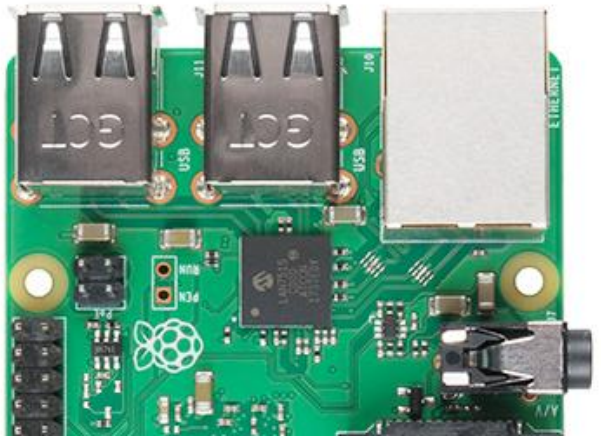
- Using a test filter

```
$ llvmlab bisect "%% result and user_time < .5 %%" <test case>
```

llvmlab bisect

- Useful command line options
  - **`--very-verbose`** enables detailed logging
  - **`--reuse-sandbox`** prevent build cache items to be extracted if already present
  - **`--min-rev=`**`NNNN` sets the minimum revision to be used
  - **`--max-rev=`**`NNNN` sets the maximum revision to be used

Demonstrations



Demonstration #1

- “Clang crashes when calling a function while both omitting a parameter and misspelling a parameter”
  - https://bugs.llvm.org/show_bug.cgi?id=40286

Demonstration #1 → Command Line

```
llvmlab bisect                                                  \
--reuse-sandbox                                                 \
--very-verbose                                                  \
--max-rev=352299                                                \
-s ~/Project/bisect_sandbox/                                    \
-b clang-armv7-linux                                            \
/bin/sh -c '%(path)s/bin/clang -fsyntax-only test.c 2>&1 |  \
            grep "undeclared identifier"'
```

Demonstration #1 - Notes

- In a real world situation (i.e. omitting `--reuse-sandbox`) it will test 23 versions of the toolchain, taking around 3 minutes to download and extract the packages (Raspberry Pi 3B+)
  - Total time is around 1h 10min (23 toolchains to test * 3 minutes each)

- Based on our experience generating the toolchains for the build-cache, building the toolchains takes around 10 minutes
  - Total time would be 3h 50min (23 toolchains to test * 10 minutes each)

- Also important to consider that not every revision is able to build

Demonstration #2

- “DAGCombiner hangs in an infinite loop”
  - https://bugs.llvm.org/show_bug.cgi?id=39098

Demonstration #2 → Command Line

```
llvmlab bisect                                  \
--reuse-sandbox                                 \
--very-verbose                                  \
--max-rev=352299                                \
-s ~/Project/bisect_sandbox/                    \
-b clang-armv7-linux                            \
bash run.sh
```

```
#!/bin/sh

ulimit -t 10; \
${TEST_PATH}/bin/llc -O0 test.ll -debug-pass=Executions
```

Final Remarks

Final remarks

- Automated bisecting is a valuable tool to easily find what commit triggered a change in behaviour
- Using **llvmlab bisect** can save a lot of time as it uses pre-compiled toolchains, stored in the cloud (the build cache)
- The build cache now contains native toolchains for for **armv7-linux** and **aarch64-linux**
- For the upcoming changes regarding the move from svn to git on LLVM repositories, changes will be needed to keep llvmlab working

Works on Arm

- The infrastructure that builds the contents of the build cache uses resources from **Works on Arm**
- **Works on Arm** offers free of charge Arm machines to open source projects to run build and testing jobs
- Application is as easy as opening a GitHub ticket!

https://www.worksonarm.com

Thanks!