Organising benchmarking LLVM-based compiler: Arm experience

Evgeny Astigeevich
LLVM Dev Meeting April 2018
Terminology

• **Upstream**: everything on llvm.org side.
• **Downstream**: everything on your side.
• **Benchmarking a compiler**: part of QA process where compiler quality requirements, such as generated code performance, code size, compilation time and others, are verified.
• **Bisecting a regression**: a process of identifying commits caused the regression.
• **Bare-metal application**: an application which runs without OS supporting it.
• **OS-hosted application**: an application which needs OS to run.
Benchmarking a compiler: get answers to

- Do my changes affect the compiler?
- Is the compiler improving?
- What caused regressions/improvements?
What is ARM Compiler 6?

- **Toolchain for development of bare-metal applications**
  - C/C++ and GNU assembly compiler based on Clang/LLVM (armclang)
  - Assembler for legacy Arm-syntx assembly
  - Linker
  - C++ libraries based on LLVM libc++
  - C libraries
  - ARM librarian (armar)
  - ARM image conversion utility (fromelf)
Why did we base our compiler on LLVM?
Why did we base our compiler on LLVM?
Cortex-A vs Cortex-R vs Cortex-M

**Cortex - A**
Highest performance
Optimised for rich operating systems

**Cortex - R**
Fast response
Optimised for high performance, hard real-time applications

**Cortex - M**
Smallest/lowest power
Optimised for discrete processing and microcontrollers
ARM Compiler product requirements

- Good quality of Cortex-A/R/M code.
- No significant regressions in releases.
The benchmarking process highly depends on how an interaction between upstream and downstream is organized.
Avoiding merge conflicts

We do development upstream as much as possible. The rough difference is ~20-50K SLOC.
Benchmarking Cortex-A code

- Cortex-A can run Linux => More benchmarks can be run
- Benchmarks are CPU-oriented => OS-hosted benchmarking can be used
- Llvm.org already has a working solution: BuildBot + LNT client/server tools
## Active Machines

<table>
<thead>
<tr>
<th>Machine</th>
<th>Latest Submission</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>llvm-juno-int__clang_DEV__aarch64:1349</td>
<td>1 hour ago</td>
<td>View Results</td>
</tr>
<tr>
<td>llvm-tk1-02__clang_DEV__thumbv7:1348</td>
<td>24 minutes ago</td>
<td>View Results</td>
</tr>
<tr>
<td>LNT-Broadwell-AVX2-O3__clang_DEV__x86_64:1344</td>
<td>12 minutes ago</td>
<td>View Results</td>
</tr>
<tr>
<td>Int-ctmark-aarch64-O0-g:1351</td>
<td>1 hour ago</td>
<td>View Results</td>
</tr>
<tr>
<td>Int-ctmark-aarch64-O3-filtro:1353</td>
<td>52 minutes ago</td>
<td>View Results</td>
</tr>
</tbody>
</table>

## Recent Submissions

<table>
<thead>
<tr>
<th>Run Order</th>
<th>Started</th>
<th>Duration</th>
<th>Machine</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>r330115</td>
<td>57 minutes ago</td>
<td>0:45:02</td>
<td>LNT-Broadwell-AVX2-O3__clang_DEV__x86_64:1344</td>
<td>View Results</td>
</tr>
<tr>
<td>r330111</td>
<td>59 minutes ago</td>
<td>0:06:40</td>
<td>Int-ctmark-aarch64-O3-filtro:1353</td>
<td>View Results</td>
</tr>
<tr>
<td>r330111</td>
<td>1 hour ago</td>
<td>0:02:44</td>
<td>Int-ctmark-aarch64-O0-g:1351</td>
<td>View Results</td>
</tr>
</tbody>
</table>
The infrastructure is similar to llvm.org infrastructure: BuildBot, LNT client/server tools.
LNT provides all needed benchmarking functionality out-of-the-box.
The internal LNT works with upstream Clang/LLVM repositories to get bisecting working.
Benchmarking Cortex-A code

- We use the internal and upstream LNTs to analyze significance of regressions.
Benchmarking Cortex-R/M bare-metal code
Daily Upstream ↔ Downstream synchronization

At 6:30am update upstream copy, build and test

All Passed

Merge to downstream

Manually resolve issues

Passed

End

6:30am was chosen because of the lowest development activity in upstream.
Repositories status

- Toolchain builds: b01, b02, b03.
Nightly downstream benchmarking

Product building

Build testable toolchain → Run benchmarks → Submit results to database → Compare results
“What caused regressions/improvements?”

• Manual bisecting:
  • An upstream commit needed to be merged to downstream. Not always possible.
  • Compiler binaries needed to be built per a merge. Not always possible.
The first solution: summary

• Pros:
  • Very simple to implement.
  • Upstream CI guards you from “bad” commits.
  • Merge conflicts are resolved when upstream is less active.
  • Nightly toolchain builds are based on a “stable” upstream trunk revision.

• Cons
  • No CI. Testing and benchmarking is started after the full toolchain is built.
  • Downstream benchmarking results are always outdated.
  • Complex merge conflicts can take more than one day and block synchronization.
  • Bisecting is very difficult.
The first solution worked well enough

- Not many commits into Arm related areas => Not many merge conflicts
- Not many optimization works => No need to automate manual tasks
- Not many embedded benchmarks => Not many regressions
But...
Increased upstream development activity (100+ commits per day) => More merge conflicts
Complex merge conflicts => Merges were blocked for days => Delayed benchmarking => A snowball effect
Any building infrastructure instabilities => No toolchain => Delayed benchmarking
More benchmarking configurations => More regressions
At the end of 2016 our solution stopped working...

Engineers might spend a week on bisecting regression. Then it was too late to report.

This resulted a lot of internal regression reports (50+) to be created but nothing was investigated and reported upstream.
We wanted to have fun but the benchmarking was a real pain in the neck.
The Optimization Team

• The team responsible for benchmarking and for implementing optimizations.
• 2 engineers (inc. a team lead): only benchmarking related tasks, no optimization tasks.
• 3 engineers (inc. a team lead): some optimization tasks.
• 4 engineers (inc. a team lead): capable to deliver great results.
Problem #1: regressions

- Solution: Continuous Integration
Continuous Integration

• In software engineering, continuous integration (CI) is the practice of merging all developer working copies to a shared mainline several times a day.

  • https://en.wikipedia.org/wiki/Continuous_integration
Any unresolved conflicts? → Create a list of commits → Is the list empty? → Merge a commit → Is it failed? → Push the merge result to downstream → End → Push the failed to Gerrit for manual resolution → End

Yes

No

Yes

No

No
New Upstream ⇔ Downstream schema

It is implemented as Jenkins’ job and is run every 30 minutes.

- Any unresolved conflicts? [Yes] [No]
  - No: Create a list of commits
  - Yes: End

- Is the list empty? [Yes] [No]
  - No: Merge a commit
  - Yes: Push the merge result to downstream

- Is it failed? [Yes] [No]
  - No: Push the failed to Gerrit for manual resolution
  - Yes: End
Commits are sorted by date to solve a multirepo problem

- Any unresolved conflicts?
  - Yes: End
  - No: Create a list of commits

- Is the list empty?
  - Yes: Push the merge result to downstream
  - No: Merge a commit

- Is it failed?
  - Yes: Push the failed to Gerrit for manual resolution
  - No: End
New Upstream ⇔ Downstream schema

If the result is not empty, the push triggers building armclang in Jenkins.
A conflict is annotated with 3-way conflict markers to help an engineer to resolve it.
Results

• On average, a merge contains 2-3 upstream commits.
• On average, bisecting time reduced from a day to a few hours. We still need to build armclang per commit.
• Most of merge conflicts are easy to resolve.
But we still...

• Did a lot of manual building.
• Did manual bisecting.
• Found that more hardware needed for regression analysis and benchmarking.
• Found hardware dependent regressions.
Build cache

- Our build cache is built on Artifactory.
Regression tracking system

Jenkins

- Auto-merge job
- CI job
- Auto-bisecting job
- Benchmarking jobs

Armclang binaries
Build cache

Results
LNT server
Regression tracking system

Jenkins

Auto-merge job

CI job

Armclang binaries

Build cache

Results

LNT server

Auto-bisecting job

On Cortex-R/M benchmarks are noiseless.
Hardware (bare-metal boards)

- The process of initialization can take more time than an actual benchmark run.
Hardware (bare-metal boards)

• We use performance simulators where it is possible.
• We moved from vendor-specific boards to FPGA boards.
Benchmarking system

Cortex-A benchmarking
- CI(BuildBot) + manual bisecting
- upstream sources

Cortex-R/M benchmarking
- CI(Jenkins) + auto-bisecting
- downstream sources
- Toolchain full overnight benchmarking
Dealing regressions

• Time is your enemy. 😞
• A good report is the key. Focus on creating a reproducer.
• Can be a workaround/downstream patch on a branch but not on the trunk.
Preventing regressions

• Be part of the community.
  • Monitor llvm mailing lists
  • Help with assessing impact
    – But we always don’t have time 😞.

• Open question: how to automate?
Future works

• Unify our systems
• Public build cache
Future works

• Unify our systems
• Public build cache
Public build cache
Questions
The Arm trademarks featured in this presentation are registered trademarks or trademarks of Arm Limited (or its subsidiaries) in the US and/or elsewhere. All rights reserved. All other marks featured may be trademarks of their respective owners.

www.arm.com/company/policies/trademarks