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

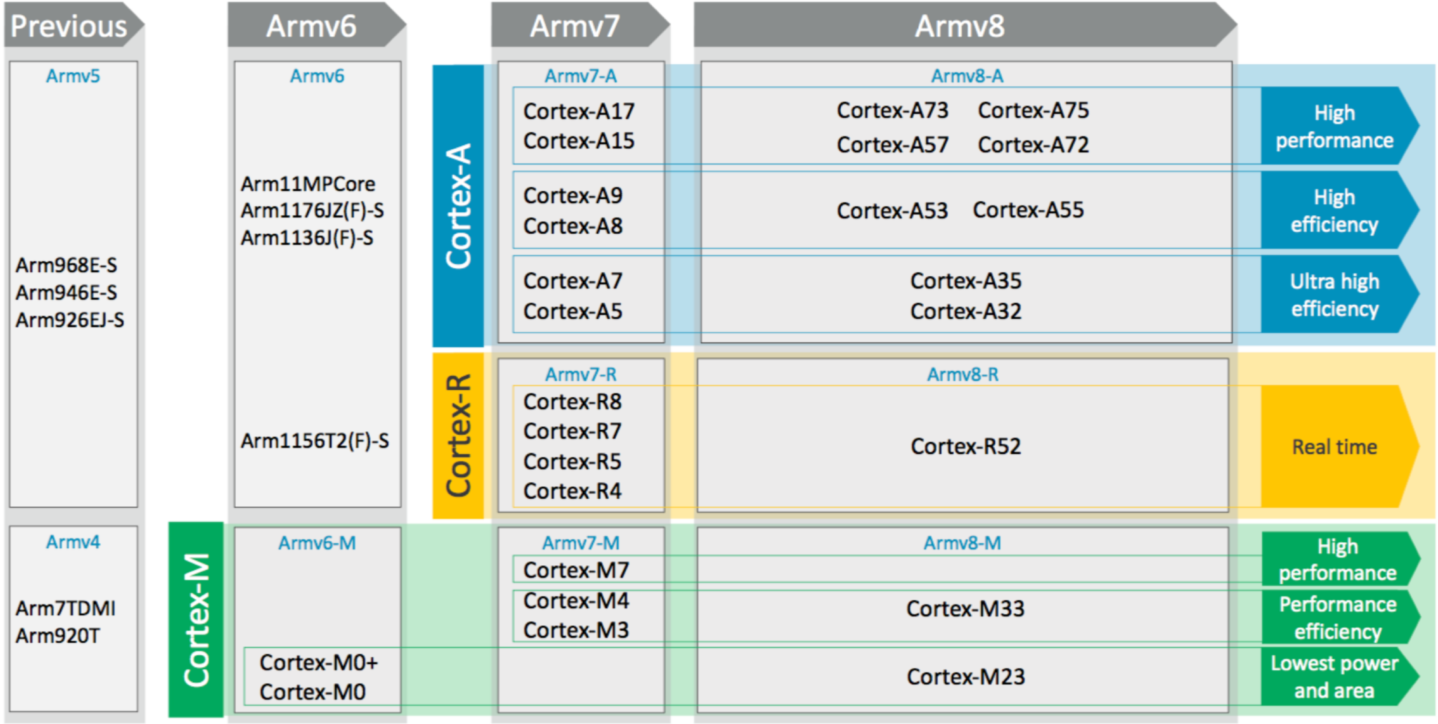
- Do my changes affect the compiler?
- Is the compiler improving?
- What caused regressions/improvements?

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-syntax 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?



	Previous	Armv6	Armv7	Armv8	
Cortex-A	Armv5	Armv6	Armv7-A	Armv8-A	
			Cortex-A17 Cortex-A15	Cortex-A73 Cortex-A75 Cortex-A57 Cortex-A72	High performance
		Arm11MPCore Arm1176JZ(F)-S Arm1136J(F)-S	Cortex-A9 Cortex-A8	Cortex-A53 Cortex-A55	High efficiency
	Arm968E-S Arm946E-S Arm926EJ-S		Cortex-A7 Cortex-A5	Cortex-A35 Cortex-A32	Ultra high efficiency
Cortex-R			Armv7-R	Armv8-R	
		Arm1156T2(F)-S	Cortex-R8 Cortex-R7 Cortex-R5 Cortex-R4	Cortex-R52	Real time
Cortex-M	Armv4	Armv6-M	Armv7-M	Armv8-M	
			Cortex-M7		High performance
	Arm7TDMI Arm920T		Cortex-M4 Cortex-M3	Cortex-M33	Performance efficiency
		Cortex-M0+ Cortex-M0		Cortex-M23	Lowest power and area

Cortex-A vs Cortex-R vs Cortex-M

Cortex - A

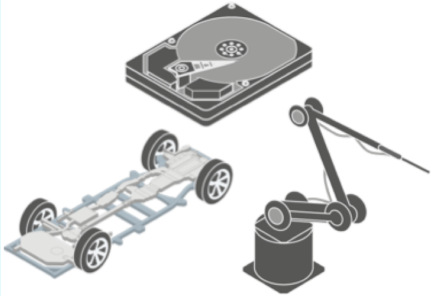
Highest performance

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

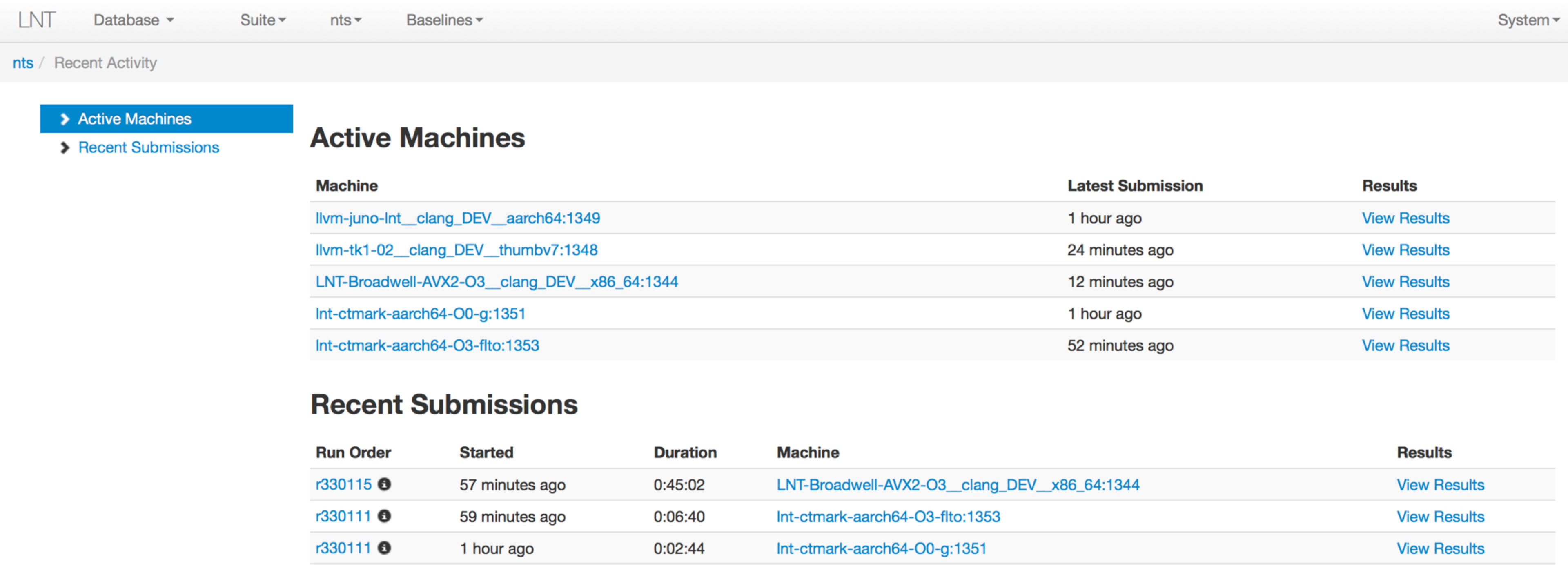
We do development upstream as much as possible.

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

lnt.llvm.org



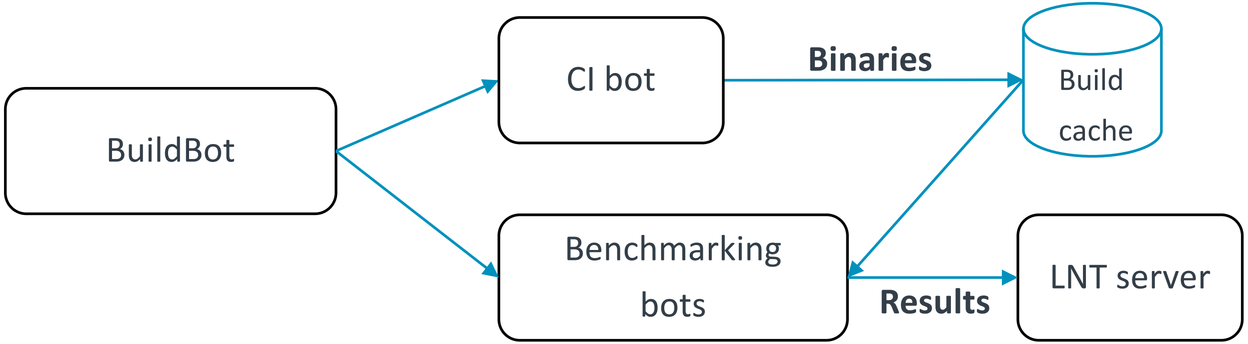
Active Machines

Machine	Latest Submission	Results
llvm-juno-lnt__clang_DEV__aarch64:1349	1 hour ago	View Results
llvm-tk1-02__clang_DEV__thumbv7:1348	24 minutes ago	View Results
LNT-Broadwell-AVX2-O3__clang_DEV__x86_64:1344	12 minutes ago	View Results
lnt-ctmark-aarch64-O0-g:1351	1 hour ago	View Results
lnt-ctmark-aarch64-O3-flto:1353	52 minutes ago	View Results

Recent Submissions

Run Order	Started	Duration	Machine	Results
r330115	57 minutes ago	0:45:02	LNT-Broadwell-AVX2-O3__clang_DEV__x86_64:1344	View Results
r330111	59 minutes ago	0:06:40	lnt-ctmark-aarch64-O3-flto:1353	View Results
r330111	1 hour ago	0:02:44	lnt-ctmark-aarch64-O0-g:1351	View Results

Internal LNT



BuildBot

CI bot

Binaries

Build cache

Benchmarking bots

Results

LNT server

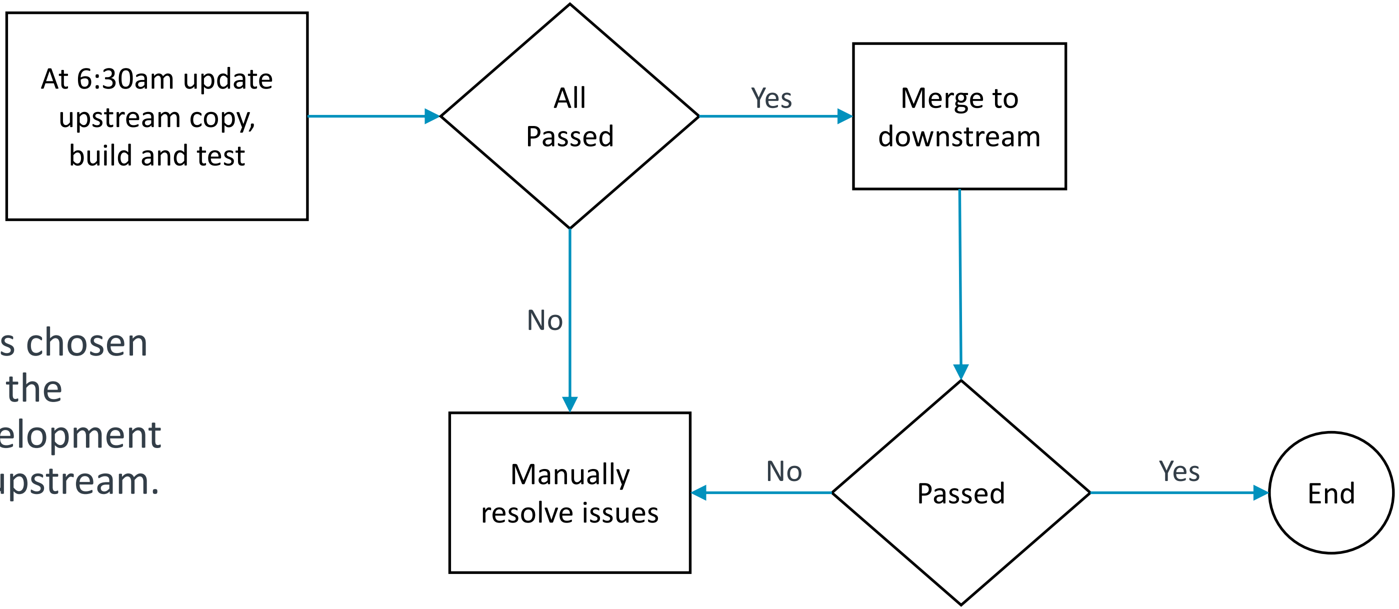
- The infrastructure is similar to llvm.org infrastructure: BuildBot, LNT client/server tools.
- LNT provides all needed benchmarking functionality out-of-the-box.
- The internal LNT works with upstream Clang/LLVM repositories to get bisecting working.

Benchmarking Cortex-A code

- We use the internal and upstream LNTs to analyze significance of regressions.

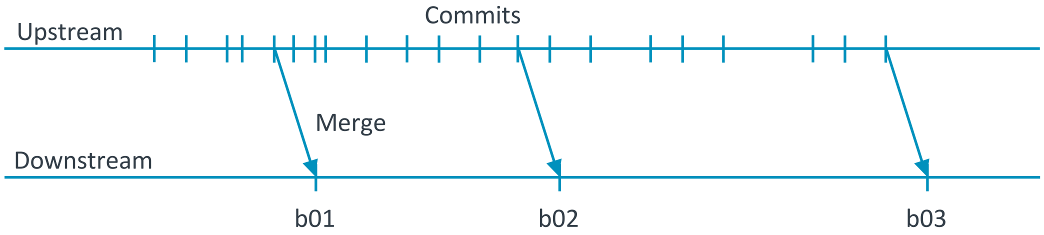
Benchmarking Cortex-R/M bare-metal code

Daily Upstream ⇔ Downstream synchronization



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

Repositories status



- Toolchain builds: b01, b02, b03.

Nightly downstream benchmarking

Product building

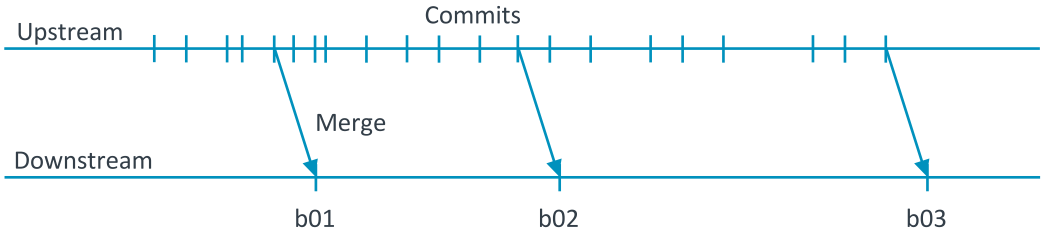
Build testable toolchain

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 ~~a~~ 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

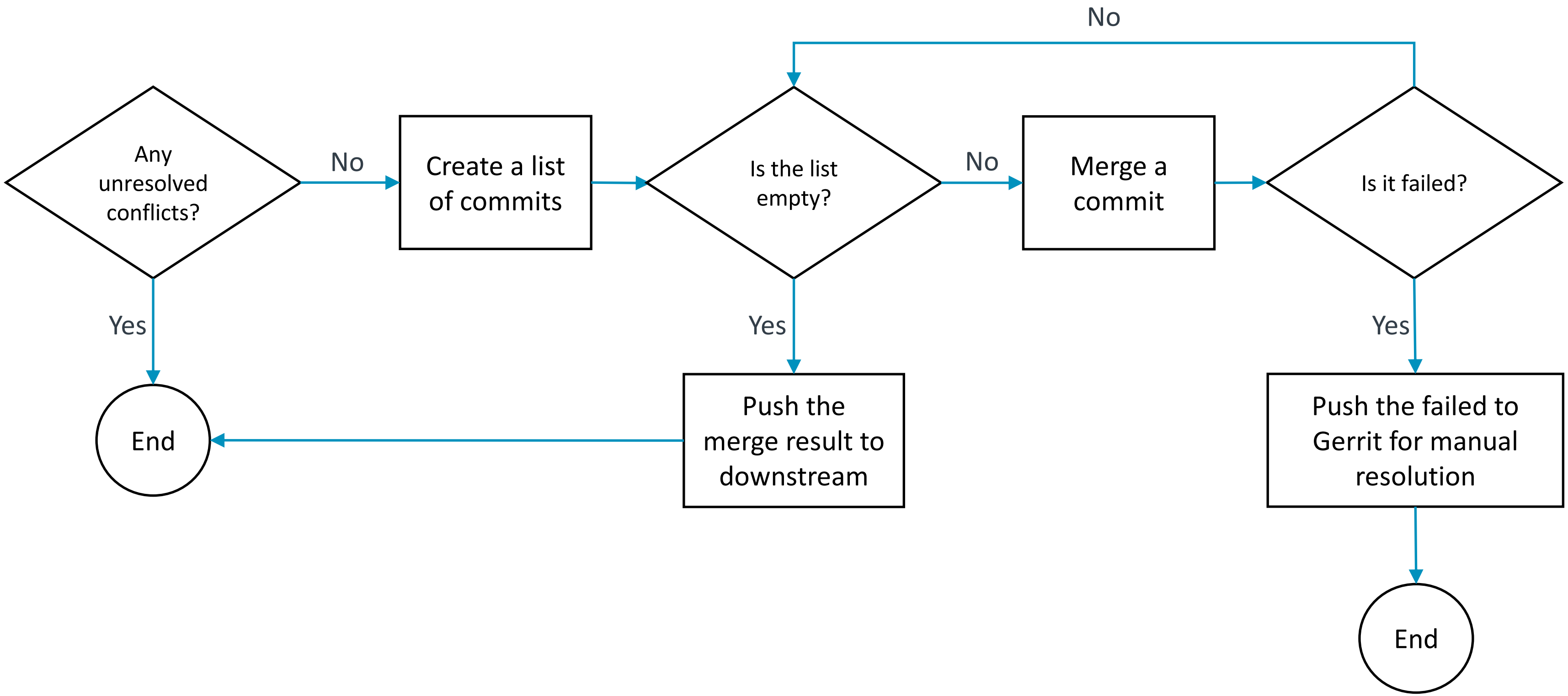
- Solution: Continuous Integration

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.

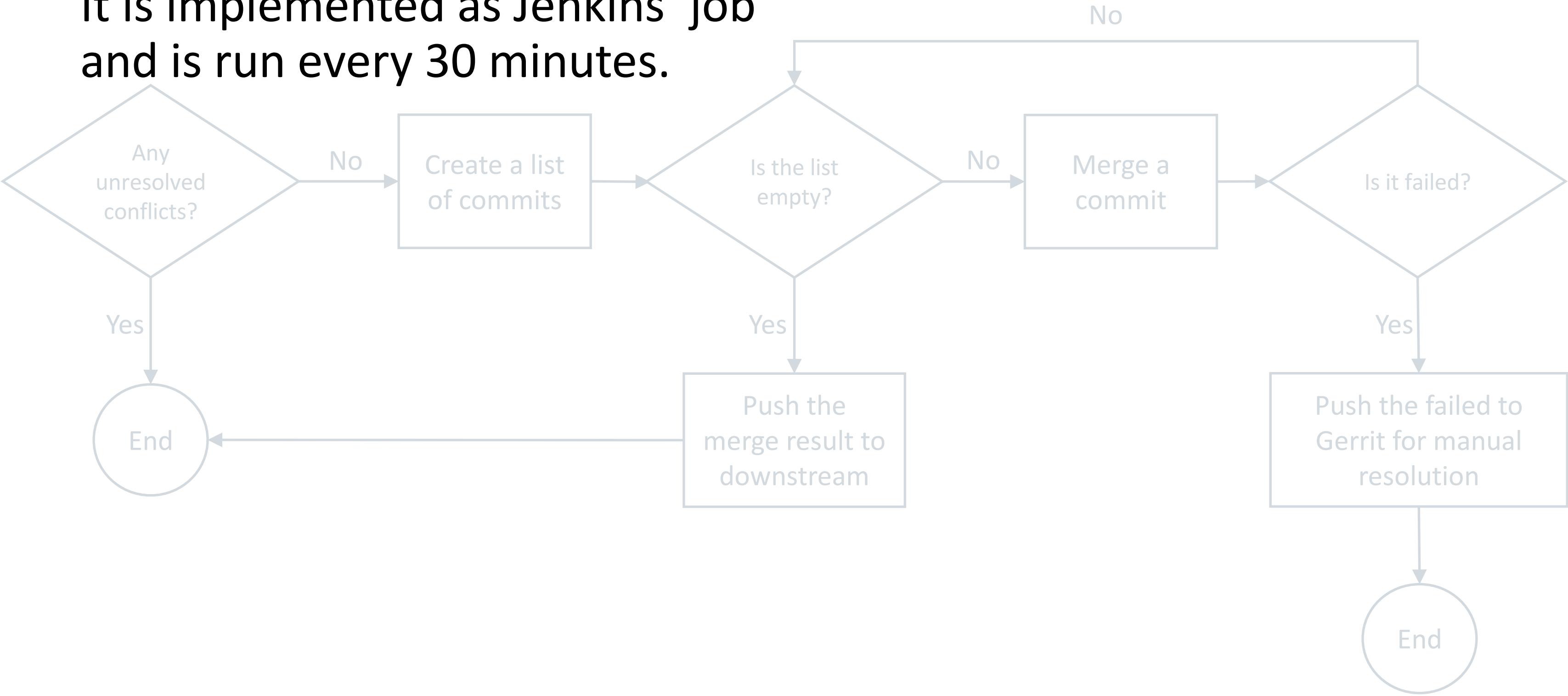
    - https://en.wikipedia.org/wiki/Continuous_integration

New Upstream ⇔ Downstream schema



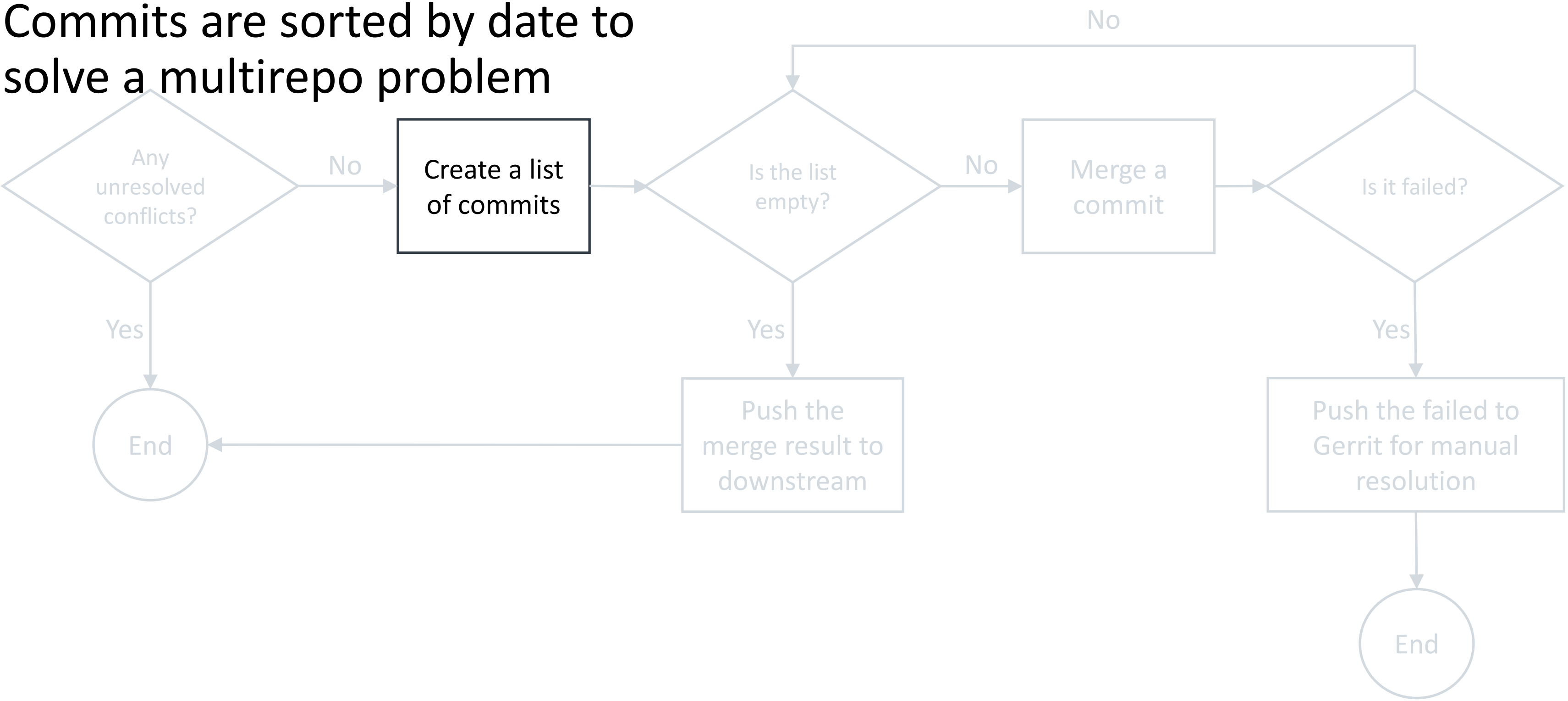
New Upstream ⇔ Downstream schema

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

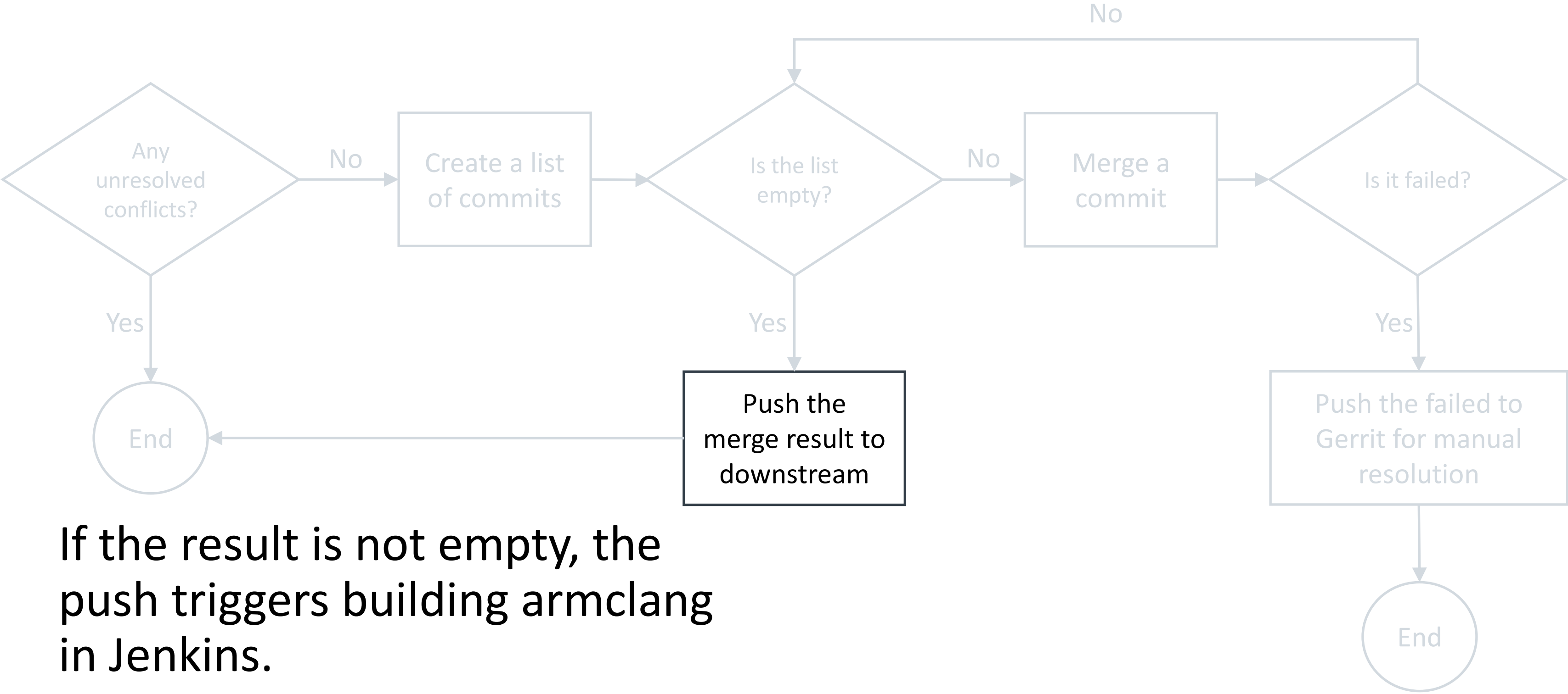


New Upstream ⇔ Downstream schema

Commits are sorted by date to solve a multirepo problem

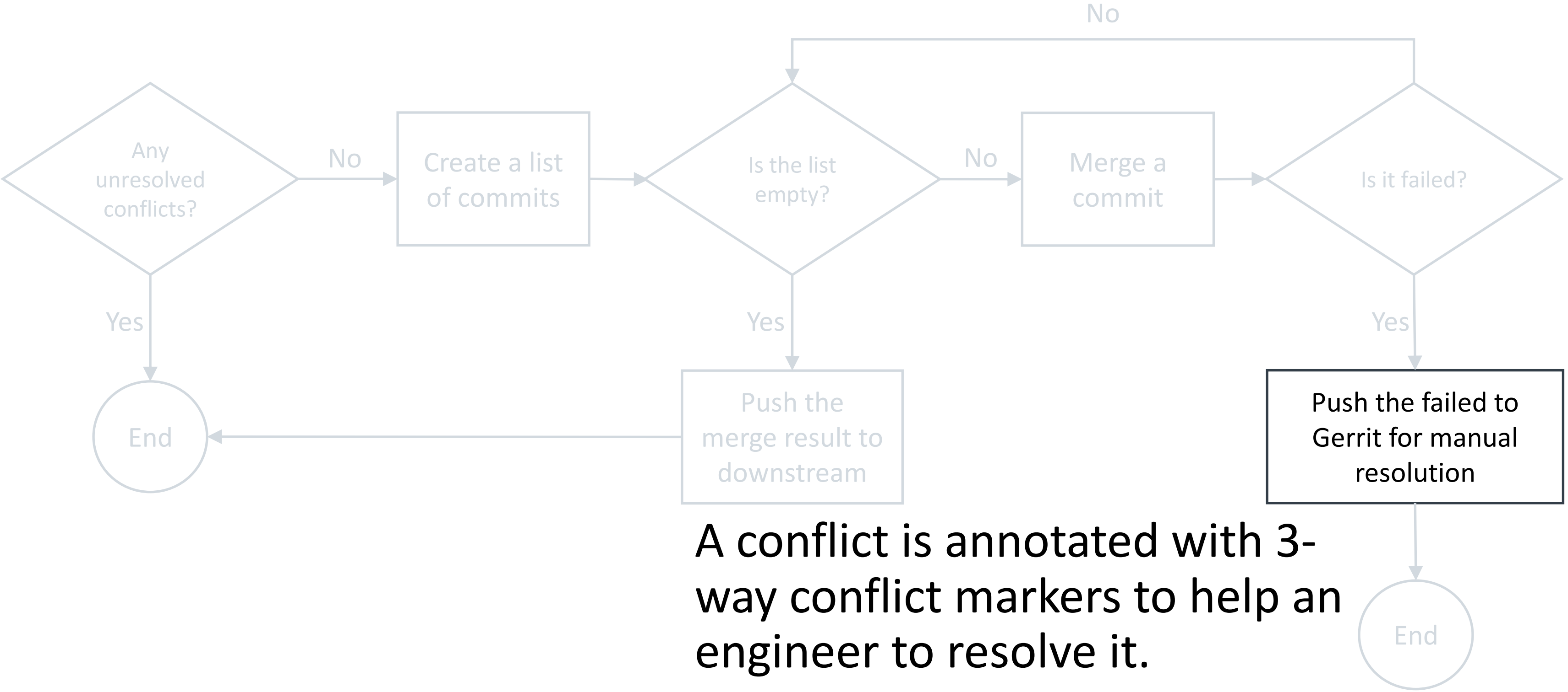


New Upstream ⬄ Downstream schema



If the result is not empty, the push triggers building armclang in Jenkins.

New Upstream ⇔ Downstream schema



Any unresolved conflicts?

No

Create a list of commits

Is the list empty?

No

Merge a commit

Is it failed?

No

Yes

End

Yes

Push the merge result to downstream

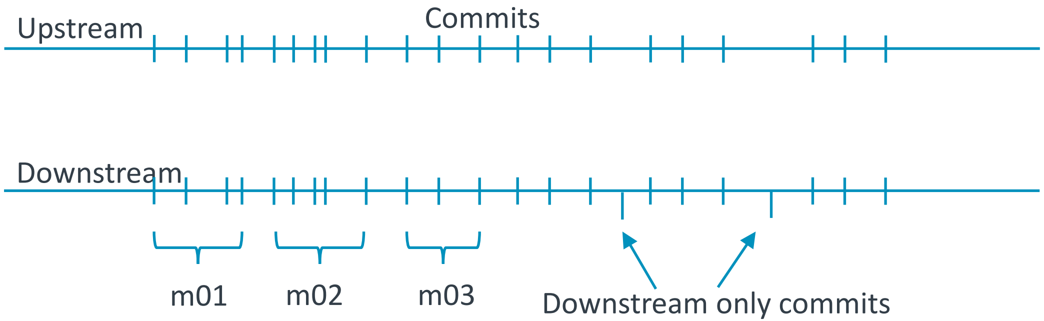
Yes

Push the failed to Gerrit for manual resolution

End

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.

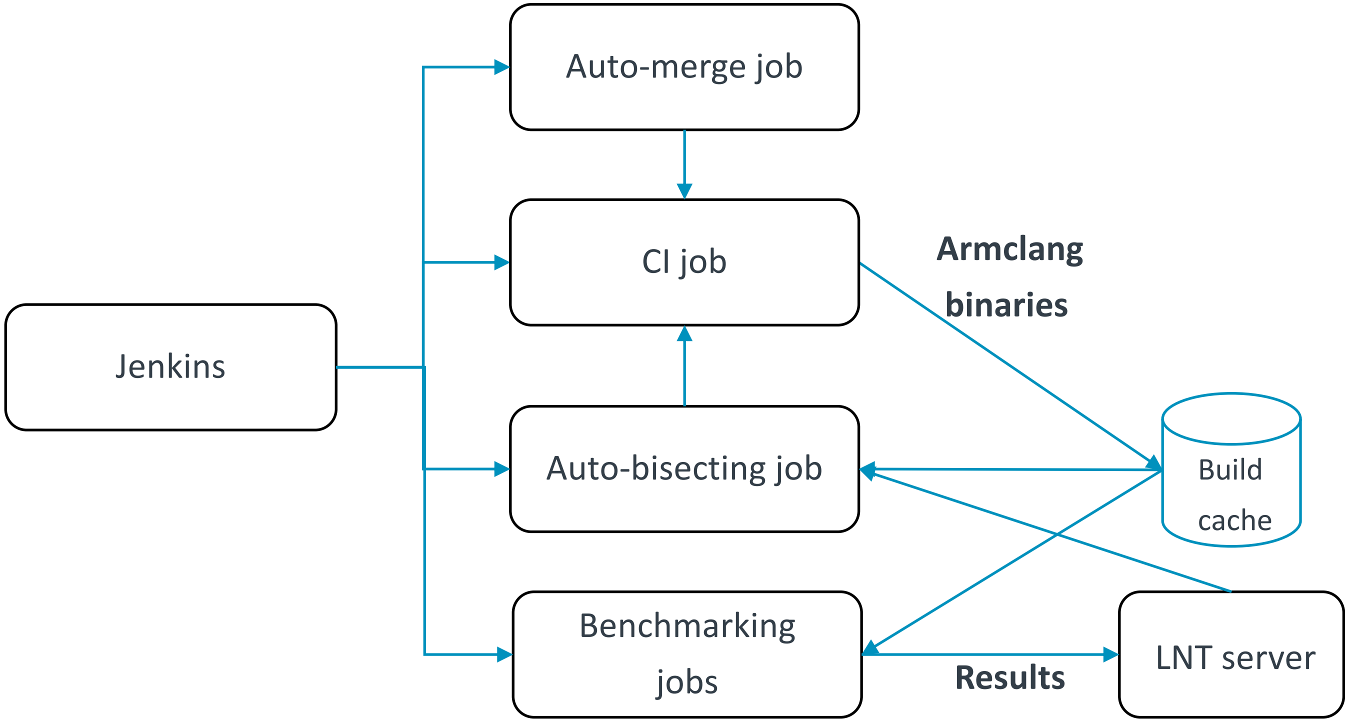
But we still...

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

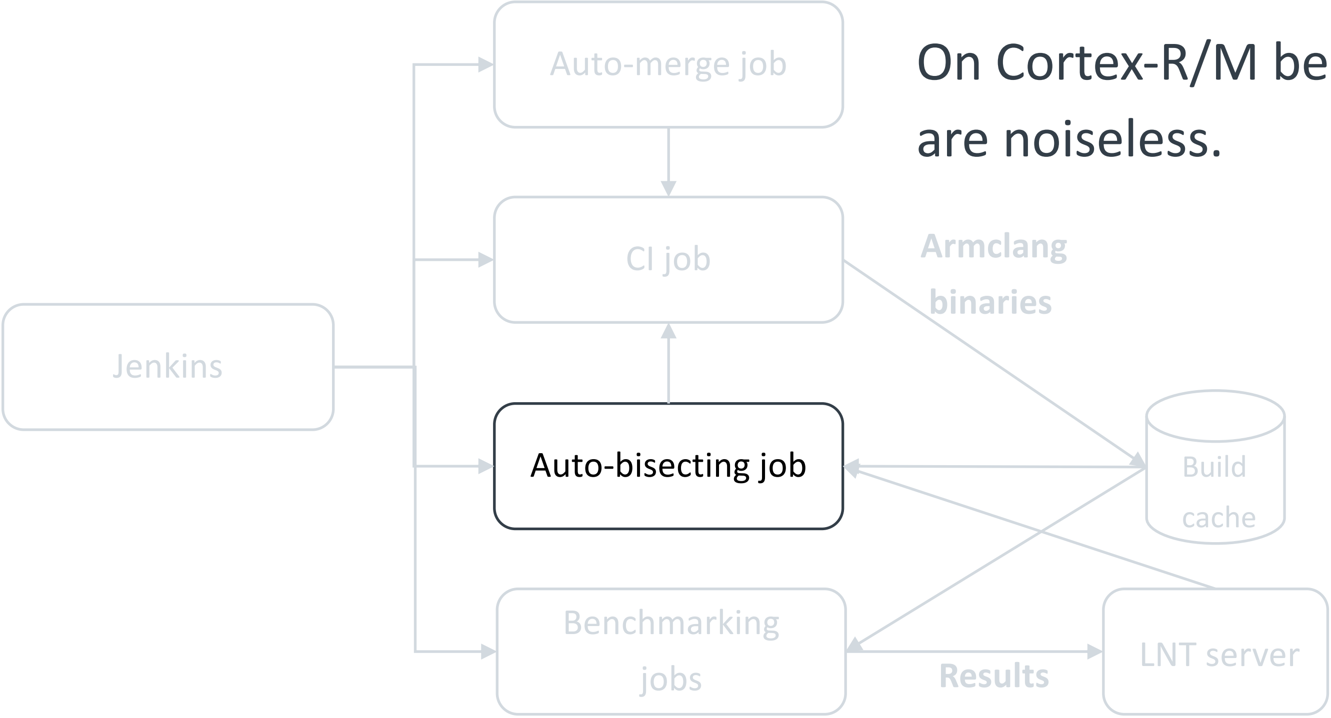
Build cache

- Our build cache is built on Artifactory.

Regression tracking system



Regression tracking system



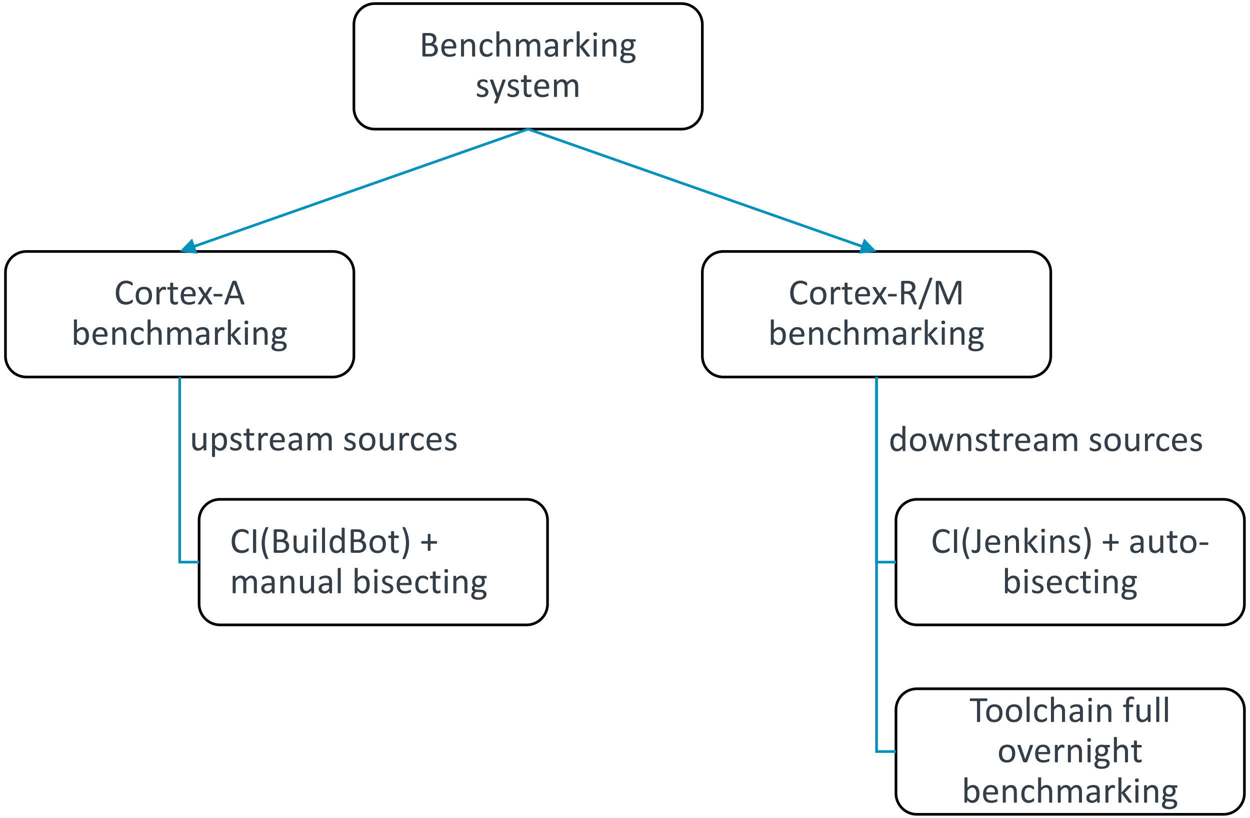
On Cortex-R/M benchmarks are noiseless.

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

upstream sources

CI(BuildBot) + manual bisecting

Cortex-R/M benchmarking

downstream sources

CI(Jenkins) + auto-bisecting

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?

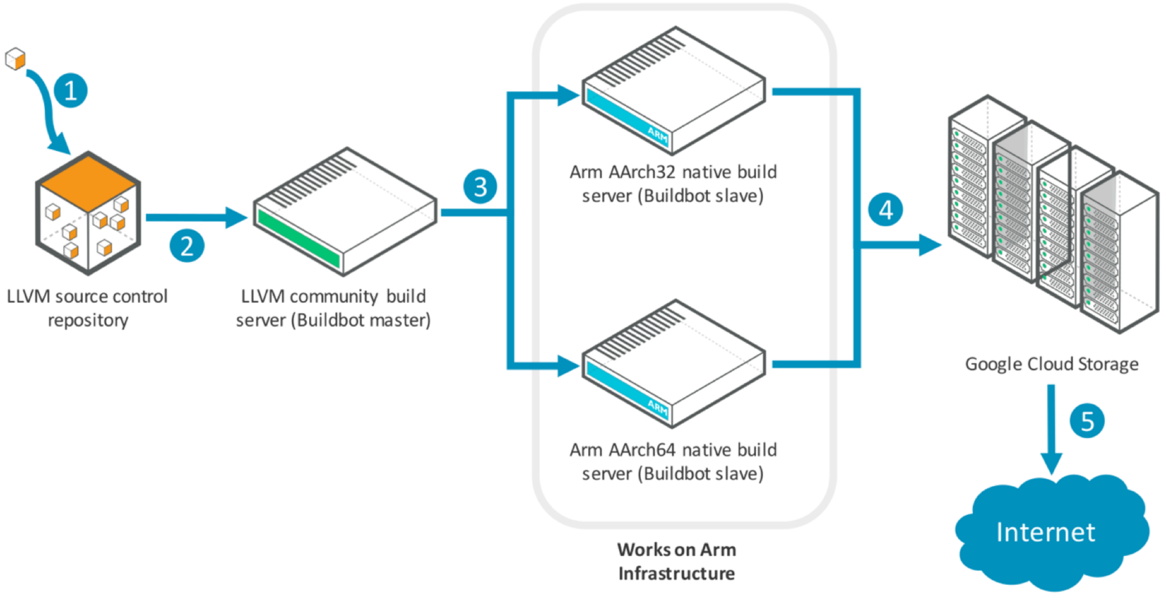
Future works

- Unify our systems
- Public build cache

Future works

- Unify our systems
- Public build cache

Public build cache



Questions