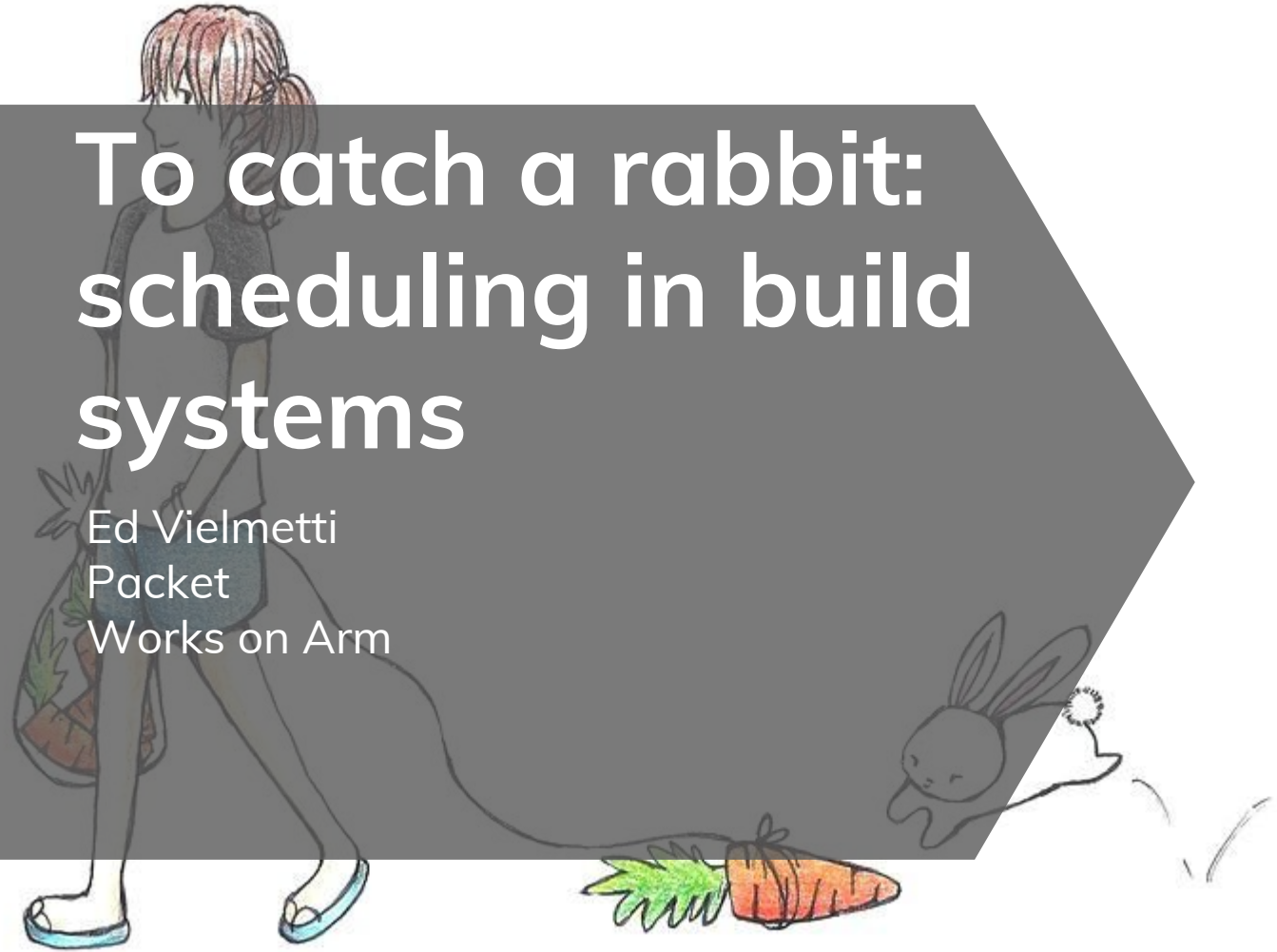
To catch a rabbit: scheduling in build systems

Ed Vielmetti
Packet
Works on Arm



Linaro connect
Bangkok 2019

About Works on Arm

Collaborative project to support open source on arm64 systems in the data center

Funded by Arm and Packet, hosted at Packet

Provides resources to open source projects

- Hardware for porting, testing and CI
- Hardware for hosted CI (Drone, Shippable, Codefresh)
- Targeted support for software ports
- Experience and assistance

Metrics of success

Measure inputs:

- Number of systems
- Number of projects
- Millions of hours of use
- System efficiency

Measure results

- Speed
- Throughput

Measure goals

- Number of rabbits caught?

Two jobs of build systems

What to do first?

- Compute dependencies
- Identify what depends on what

What to do in parallel?

- Pursue independent tasks
- Tackle common requirements

Interdependencies and priorities

Naive approach: identify priorities, attack in order.

- Proofs of concept
- Lengthy by-hand build instructions
- Ad hoc patches
- Architecture-specific forks

Evolved approach: identify dependencies, attack in depth.

- Repeatable builds
- Upstream patches
- Focus on shared dependencies
- CI/CD to build, test in parallel with development

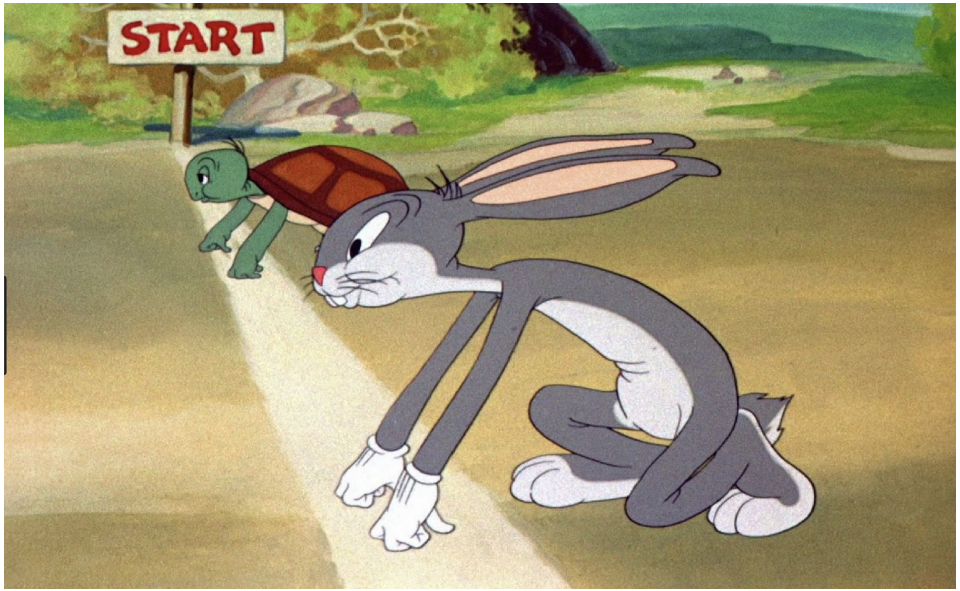
Everything depends on the build systems

It’s not enough that the software works!

Long-term adoption of new systems requires attention to stability

- Repeatability of builds
- High performance build infrastructure
- Robust test environments
- Scale out to support testing velocity

Can’t jump to hyperscale usage without solid foundation



Examples of build system components

In increasing order of complexity:

- Good old Make
- CMake, generating Makefiles
- CMake, generating ninja files, plus ninja (LLVM)
- gn, generating ninja files, plus ninja (Chromium)
- Bazel, with rules written in Skylark (Kubernetes)
- gg, which uses AWS Lambda as a back end (Stanford research)
- NixOS and nixpkgs
- SubmitQueue (Uber)

There are many others - if your favorite is not included here, don’t be alarmed.

Beyond Make: Dividing up the problem

Good old Make!

- Turing complete Makefile language
- Slow to start on big builds that don't do much
- Can't reason about dependencies without running the build

Divide and conquer

- One system to compute dependencies (CMake)
- A second system to run that build file (Make or Ninja)

So-called "meta-build system"

Example: LLVM's use of CMake plus Ninja to do builds.

Meta-build systems

What is a meta-build system? Sounds fancy!

- Determines dependencies
- Runs infrequently (or not at all, on small changes)
- Generates Makefiles or build files
- Divides the problem in half
- A good one lets you replace either half

Why bother?

- Project is large, hard to reason about
- Build times are too slow
- Codebase is very large, contributor count is high

Big system is a sign of success!

Building the build tools

“Gn”, a meta-build tool for Ninja

- Used to build Chromium
- Replaces “Gyp”
- Not yet ported to arm64

Bazel, a build system from Google

- Built with itself!
- Used for Tensorflow
- Written in Java
- Some Java pieces built with Bazel
- Not yet ported to arm64

Hyperscale tools, not ready yet!

Too complex to reason about

Build systems are

- Complex systems in their own right
- Undervalued by those who depend on them
- A primary source of portability issues

The more I prepared to give this talk,

The less I am prepared to say I understand them.

gg - building 1000-fold in parallel with Lambda

gg is a research build system

- Originated at Stanford
- In very limited use

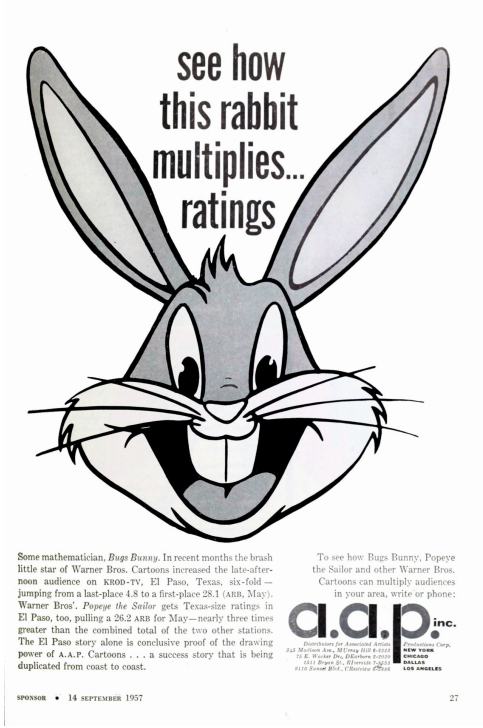
Two-part build system

- Divide big project into “thunks”
- Dispatch each job to AWS Lambda

Parallelism at very large scale

Amdahl’s Law

- Limited by slowest serial path



NixOS - reproducible builds

NixOS - purely functional operating system

Nixpkgs - build system

- Caching of builds
- Reproducible results
- Fast and exact repeatability

NixOS runs on x86 and arm64.

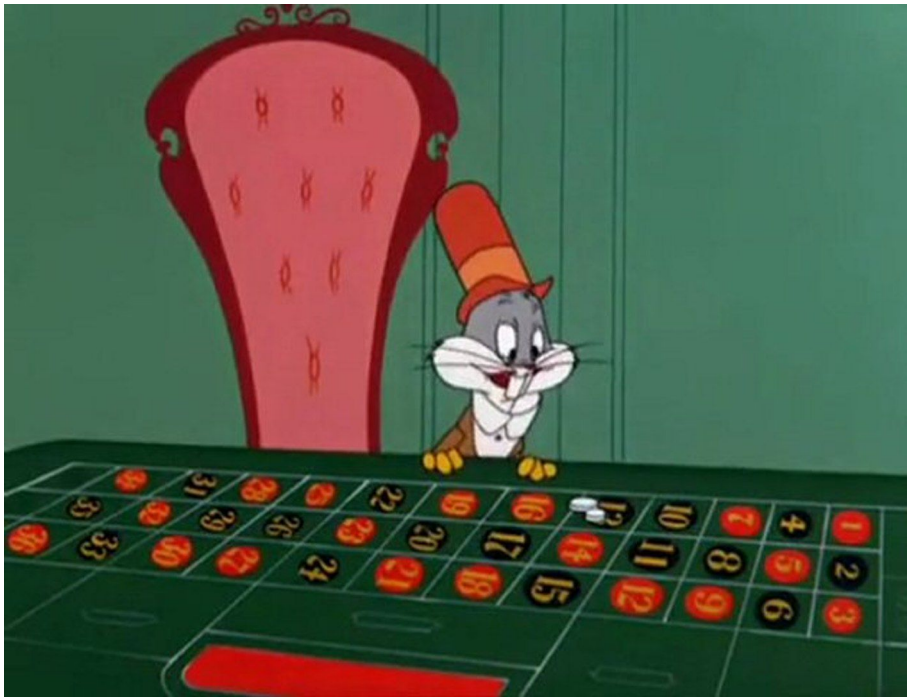
Keeping pace with speculative builds

Uber “SubmitQueue”

- Speculative execution of builds
- Merge combinations of work
- Keep mainline green

OpenStack “Zuul”

- Gating tests for changes
- Merge changes to dependencies
- Identify and reject failures quickly



CI/CD goals

Why do we put together CI/CD systems?

- Rapid feedback on change requests
- Reproducible builds
- Faithful tests that reflect real world conditions
- Risk management for dependencies
- Time-efficient builds of complex systems
- Cost-efficient build infrastructure
- Manageable complexity

In general, no single system provides all of these.

How do you measure dependencies?

Our customers

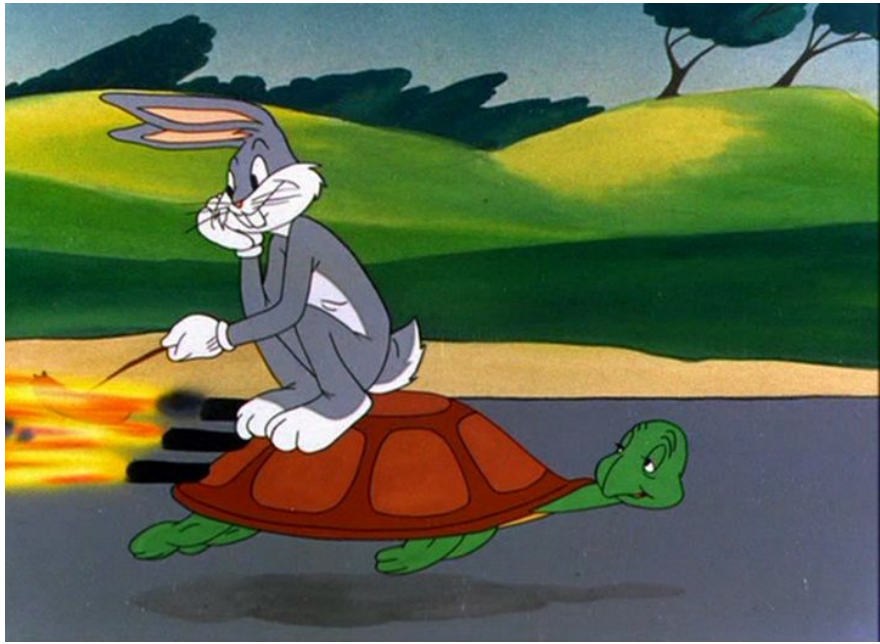
- Asking for A, B, C
- Most important! Do first!
- Not asking for X, Y, Z

Open source developers

- Working on A, B, C
- Depending on X, Y, Z for daily use

Good tooling attracts developers

Bad tooling frustrates them!



CI/CD challenges for Arm systems

What is particularly difficult for Arm-on-Arm build environments?

- Lack of native support in hosted build systems
- Shortage of high-powered cores in high-memory machines
- Imperfect emulation in QEMU
- Poor understanding of the depths of dependency chains
- Complexity of cluster build environments
- Ubiquity of x86-first, x86-only design thinking in cloud environments

These can all be overcome with diligence and effort

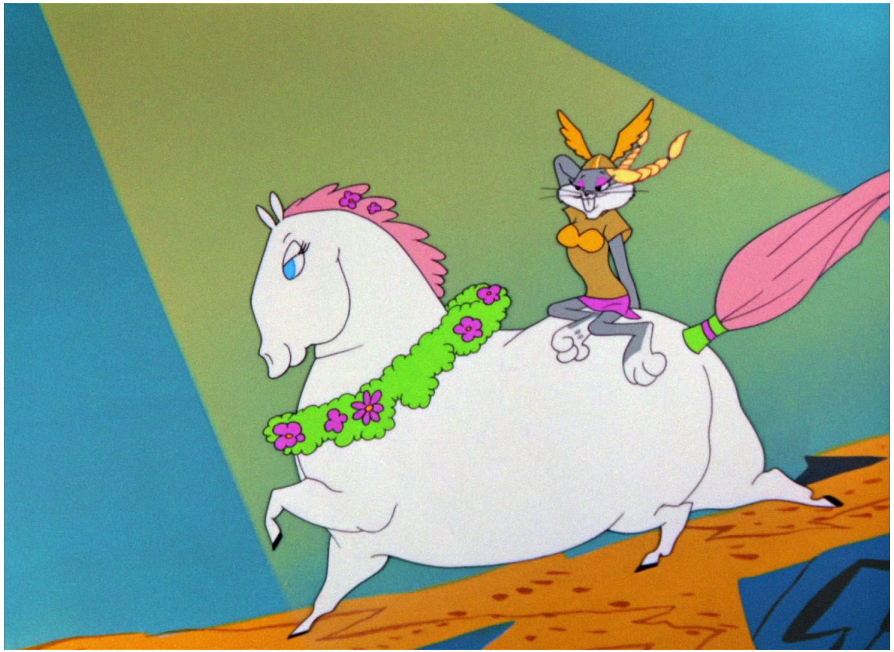
To catch a unicorn

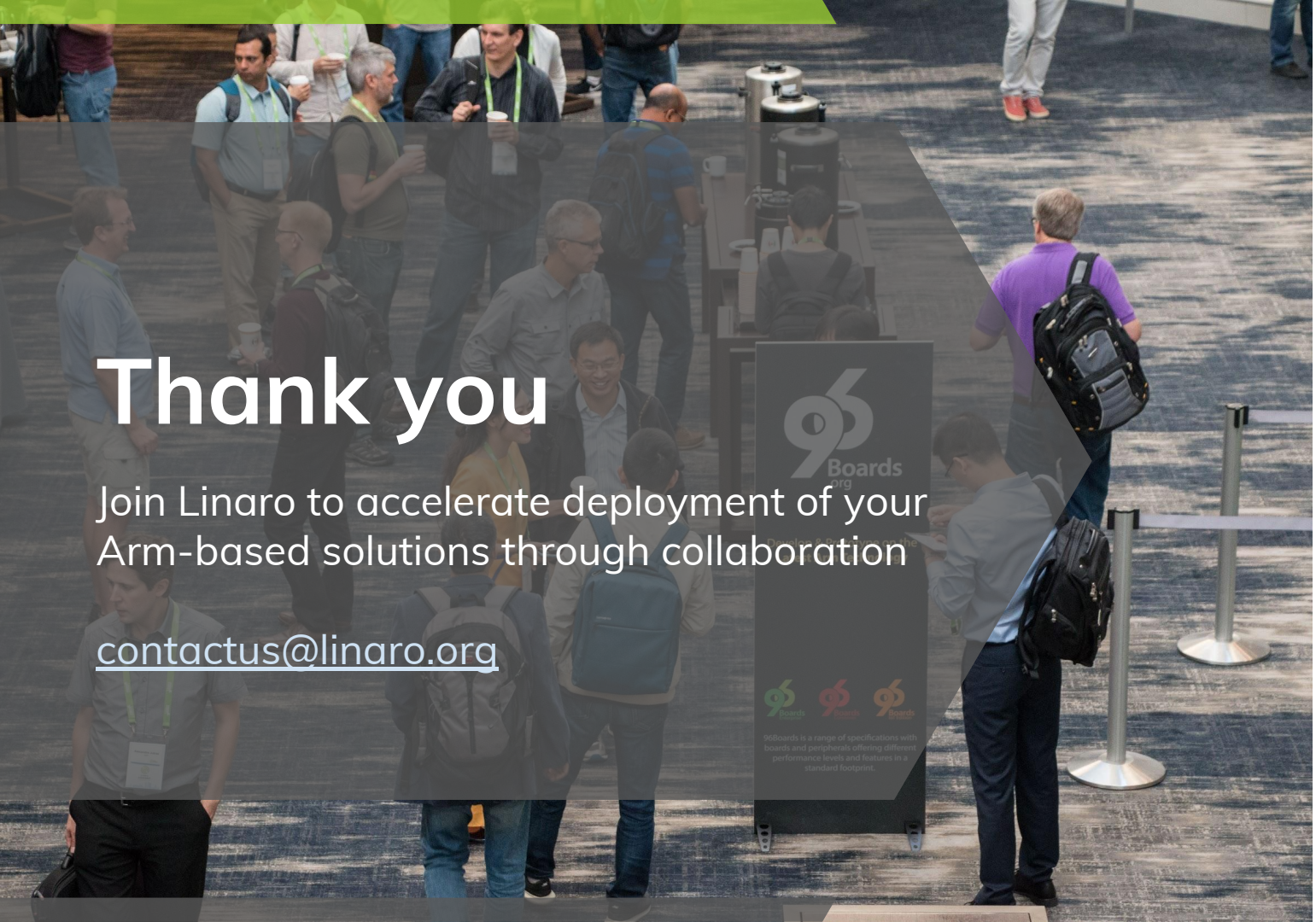
Goals?

- Catch developers
- Attract talent
- Provide competitive advantage
- Enable growth

Unicorn = billion dollar startup

If everything goes right!





Thank you

Join Linaro to accelerate deployment of your Arm-based solutions through collaboration

contactus@linaro.org

Linaro connect
Bangkok 2019