Build infrastructure on Intel x86-64

Contents

Build infrastructure on Intel x86-64

Introduction

The current Apertis infrastructure is largely made of hosts based on the Intel x86-64 architecture, often using virtualized machines.

The only exceptions are:

- OBS workers used to build packages natively for the ARM 32 bit and ARM 64 bit architectures
- LAVA workers, which match the reference hardware platforms[1]

While LAVA workers are by nature meant to be hosted separately from the rest of the infrastructure and are handled via geographically distributed LAVA dispatchers[2], the constraint on the OBS workers is problematic for adopters that want to host downstream Apertis infrastructure.

Why host the whole build infrastructure on Intel x86-64

Being able to host the build infrastructure solely on Intel x86 64 bit (usually referred to as `x86-64` or `amd64`) machines enables downstream Apertis to be hosted on a standard public or private cloud solution as these usually only offer x86-64 machines.

Deploying the OBS workers on cloud providers would also allow for the implementation of elastic workload handling.

[1] https://www.apertis.org/reference_hardware/
[2] https://gitlab.apertis.org/infrastructure/apertis-lava-docker/blob/master/apertis-lava-dispatcher/README.md

Elastic scaling, and the desire to ensure that the cloud approach is tested and viable for downstreams, means that the deployment approach described in this document is of interest for the main Apertis infrastructure, not just for downstreams.

Some cloud providers like Amazon Web Services have recently started offering ARM 64 bit servers as well. As a result it should be possible to adopt a hybrid approach, mixing foreign builds on x86-64 and native ones on ARM machines.

In particular, Apertis is currently committed to maintain native workers for all the supported architectures, but is aiming for a hybrid set up where foreign packages get built on a mix of native and non-native Intel x86 64 bit machines.

Downstreams will be able to opt for fully native, hybrid or Intel-only OBS worker setups.

Why OBS workers need a native environment

Development environments for embedded devices often rely on cross-compilation to build software targeting a foreign architecture from x86-64 build hosts.

However, pure cross-compilation prevents running the unit tests that are shipped with the projects being built, since the binaries produced do not match that of the build machine.

In addition, supporting cross-compilation across all the projects that compose a Linux distribution involves a considerable effort, since not all build systems support cross-compilation, and where it is supported some features may still be incompatible with it.

From the point of view of upstream projects, cross-compilation is in generally a less tested path, which often leads cross-building distributors to ship a considerable amount of patches adding fixes and workarounds.

For this reason all the major package-based distributions like Fedora, Ubuntu, SUSE and in particular Debian, the upstream distribution from which Apertis sources most of its packages, choose to only officially support native compilation for their packages.

The Debian infrastructure thus hosts machines with different CPU architectures, since the build workers must run hardware that matches the architecture of the binary packages being built.

Apertis inherits this requirement, and currently has build workers with Intel 64 bit, ARM 32 and 64 bit CPUs.

CPU emulation

Using the right CPU is fortunately not the only way to execute programs for non-Intel architectures: the QEMU project[3] provides the ability to emulate a multitude of platforms on an x86-64 machine.

QEMU offers two main modes:

- system mode: emulates a full machine, including the CPU and a set of attached hardware devices;
- user mode: translates CPU instructions on a running Linux system, running foreign binaries as if they were native.

The system mode is useful when running entire operating systems, but it has a severe performance impact.

The user mode has a much lighter impact on performance as it only deals with translating the CPU instructions in a Linux executable. For instance, running an ARMv7 ELF binary on top of the x86-64 kernel running on a x86-64 host.

Using emulation to target foreign architectures from x86-64

The build process on the OBS workers already involves setting up a chroot where the actual compilation happens. By combining it with the static variant of the QEMU user mode emulator it can be used to build software on a x86-64 host targeting a foreign architecture as if it were a native build.

The binfmt_misc[4] subsystem in the kernel can be used to make the emulation transparent so that emulation happens automatically and transparently when a foreign binary is executed. Packages can then be built for foreign architectures without any changes.

The emulation-based compilation is also known as Type 4 cross-build[5] in the OBS documentation.

The following diagram shows how the OBS backend can distribute build jobs to its workers.

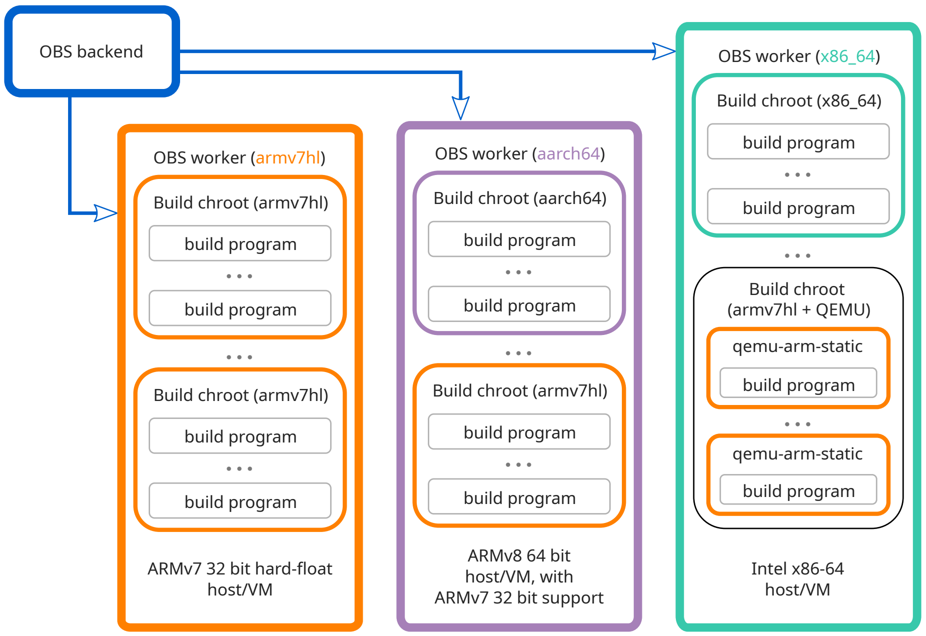
Each CPU instruction set is marked by the code name used by OBS:

- `x86_64`: the Intel x86 64 bit ISA, also known as `amd64` in Debian
- `armv7hl`: the ARMv7 32 bit Hard Float ISA, also known as `armhf` in Debian
- `aarch64`: the ARMv8 64 bit ISA, also known as `arm64` in Debian

[3] https://www.qemu.org/

[4] https://en.wikipedia.org/wiki/Binfmt_misc

[5] https://en.opensuse.org/openSUSE:Build_Service_Concept_CrossDevelopment#Types_of_crossbuild



Particularly relevant here are the `armv7hl` jobs building ARMv7 32 bit packages that can be dispatched to:

1. the native `armv7hl` worker machine;
2. the `aarch64` worker machine, which supports the ARMv7 32 bit ISA natively and thus can run binaries in `armv7hl` chroots natively;
3. the `x86_64` worker machine, which uses the `qemu-arm-static` binary translator to run binaries in `armv7hl` chroots via emulation.

Some ARM 64 bit server systems do not support the ARMv7 32 bit ISA natively, and would thus require the same emulation-based approach used on the x86-64 machines to execute the ARM 32 bit jobs.

Mitigating the impact on performance

The most obvious way to handle the performance penalty is to use faster CPUs. Cloud providers offer a wide range of options for x86-64 machines, and establishing the appropriate cost/performance balance is the first step. It is possible that the performance of an emulated build on a fast x86-64 CPU may be comparable or even faster than a native build on a older ARMv7 machine.

In addition, compilation is often a largely parallel task:

1. big software projects like WebKit are made of many compilation units that can be built in parallel
2. during large scale rebuilds each package can be built in parallel

Even if some phases of the build process do not benefit from multiple cores, most of the time is spent on processing the compilation units which means that increasing the numbers of cores on the worker machines can effectively mitigate the slowdown due to emulation on large packages.

For large scale rebuilds, scaling the number of machines is already helpful, as the build process for each package is isolated from the others.

A different optimization would be to use some selected binaries for the native architecture during the qemu-linux-user emulation. For instance, a real cross-compiler can be injected in the build chroot and make it pretend to be the "native"compiler in the otherwise emulated environment.

This would give the best possible performance as the compilation is done with native `x86-64` code, but care has to be taken to ensure that the cross-compiler can run reliably in the foreign chroot, and keeping the native and emulated versions synchronized can be challenging.

Risks

Limited maturity of the support for cross-builds in OBS

Support for injecting the QEMU static emulator in the OBS build chroot seems to be only well tested on RPM-based systems, and there may be some issues with the DEB-based approach used by Apertis.

A feasibility study was done by Collabora in the past demonstrating the viability of the approach, but some issues may need to be dealt with to deploy it at scale.

Versioning mismatches between emulated and injected native components

If native components are injected in the otherwise emulated cross-build environment to mitigate the impact on performance, particular care must be made to ensure that the versions match.

Impact of performance loss on timing-depended tests

Some unit tests shipped in upstream packages can be very sensitive to timing issues, failing on slower machines. If the performance impact is non-trivial, the emulated environment may be subject to the same failures.

However, this is not specific to the emulated environment: Apertis often faces this kind of issues where some tests that pass on the main Apertis infrastructure fail due to timing issues on the slower workers that downstream distributions may use.

To mitigate the impact on downstream distributors, the flaky tests usually get fixed or, if the effort required is too large, disabled.

Emulation bugs

The emulator may have bugs that may get triggered by the build process of some packages.

Since upstream distributors use native workers those issues may not be caught before the triggering package is built on the Apertis infrastructure.

Debugging this kind of issues is often not trivial.

Approach

These are the high level steps to be undertaken to be able to run the whole Apertis build infrastructure on x84-64 machines:

- Set up an OBS test instance with a single `x86-64` worker
- Configure the test instance and worker for `armhf` and `aarch64` emulated builds
- Test a selected set of packages by building them for `armhf` and `aarch64`
- Set up other `x86-64` workers and test a rebuild of the whole archive, ensuring that all the packages can be build from using the emulated approach
- Devise mitigations in case some packages fail to build in the emulated environment
- Measure and evaluate performance impact comparing build times with those on the native workers currently in use in Apertis, to decide whether scaling the number of workers is sufficient to compensate the impact
- Test mitigation approaches over a selected set of packages and evaluate the gains
- Do another rebuild of the whole archive to ensure that the mitigations didn't introduce regressions
- Refine and deploy the chosen mitigation approaches to, for instance, ensure that the injected native binaries are kept synchronized with the emulated ones they replace

There's a risk that no mitigation end up being effective on some packages so they keep failing in the emulated approach. In the short term those packages will be required to be built on the native workers in a hybrid set up, but they would be more problematic in a hypothetical downstream setup with no native workers as they can't be built there. In that case, pre-built binaries coming from an upstream with native workers will have to be injected in the archive.

Alternatively, it may be possible to mix type 3 and 4 crossbuilds[6] by modifying the failing packages to make them buildable with a real cross-compiler. This solution requires a much higher maintenance cost as packages do not generally support being built in that way, but it may be an option to be able to do full builds on x86-64 in the few cases where emulation fails.

[6] https://en.opensuse.org/openSUSE:Build_Service_Concept_CrossDevelopment#Types_of_crossbuild

Evaluation Report

A full archive-wide build was run on the Azure Cloud setup, using `x86-64` virtual machines. A cloud optimized setup was built, comprising of the following major components:

- Azure provided Linux Virtual Machines (Debian Buster)
- Docker (as provided by the Linux distribution vendor)
- Linux 4.19 and above
- binfmt-support
- QEMU Emulator

Given the task at hand, to run emulation for `ARM` architecture on `x86-64`, we chose the following cloud hardware class for our OBS worker setup.

- OBS-Server VM: Standard DS14 v2 (16 vcpus, 112 GiB memory)
- Worker VM: Standard F32s_v2 (32 vcpus, 64 GiB memory)

The provisioned `OBS-Server` VM hosted all of the OBS services, dockerized to run easily and efficiently in a cloud environment. For the workers, we provisioned 3 `Worker` VMs, each VM running 5 worker instances per architecture, with 3 architectures this resulted in a total of 15 worker instances per virtual machine. In total, we ran 45 worker instances for our build farm. This includes 30 worker instances doing emulated builds, 15 for the 32-bit ARM architecture and 15 for the 64 bit architecture. The remaining 15 worker instances were allocated for native `x86` builds.

All services used Azure provided *Premium SSD* disk storage. Azure Networking was tweaked to allow full intercommunication in-between the VMs.

The OBS Build setup was populated with the Apertis v2021dev3 release for the `development, target and sdk` components. The combined number of packages for the 3 repository components is: `4121`

- development => 3237 packages
- target => 465 packages
- sdk => 419 packages

Of the mentioned repositories, `development` and `target` repository are built for 3 architectures: `x86-64`, `armv7hl` and `aarch64`, while `sdk` repository is built only for the `x86-64` architecture.

The full archive-wide rebuild of Apertis v2021dev3 was completed in around 1 week, with the above mentioned setup. There weren't any build failure specific to the setup above, to the `emulated build` setup in particular. Some packages failed to build while running their respective build time tests.

To summarize, *Emulated Builds* worked fine with 2 caveats mentioned below

- Performance: Given the emulation penalty, builds were 4-5 times slower than native.

- Failing packages: Given the performance penalty due to emulation, some of the tests failed due to timeouts