Bootstrap of the **Debian ppc64el** port:
History, Concepts, Techniques, and Challenges

Mauricio Faria de Oliveira
<mauricio.foliveira@gmail.com>

Linux Developer Conference Brazil
August 26th, 2018

This author/material does not represent any company/group or its views.

Contents

- History
  - Why little-endian 64-bit PowerPC?
  - Why Debian on little-endian 64-bit PowerPC?
- Concepts
  - What is the Debian ppc64el port?
  - What is bootstrap?
  - Overview of bootstrap process
- Techniques
  - QEMU/KVM, powernv & pseries platforms
  - Cross build on Debian powerpc port
  - git repo: scripts, patches, workarounds
  - Part 0: prepare build/host environment
  - Part 1: build cross toolchain
  - Part 2: build enough to boot and build
  - Part 3: build enough to build the toolchain
  - Part 4: build more
- Challenges
  - Source packages change quickly
  - Build dependencies: circular/cyclic
  - Build dependencies: recursive/self-hosted
  - Build problems: libtool, autoreconf
  - Rebuild The World, twice (ABI, GLIBC)
  - Small team of non-Debian developers
- Timeline – from dpkg arch to release!

Bootstrap of the **Debian ppc64el** port:

History, Concepts, Techniques, and Challenges

History: **why** LE 64-bit PowerPC?

(*2 reasons – these are my views; facts might relate; who knows it all? :-*)

So, back in ~2013 (and earlier years)...

Note on the word *traditionally* – IBM PowerPC-based servers have existed for a long time, and for most of that, used to be adopted in / focused on "enterprise" scenarios
(e.g., more static/scale-up, CPU/MEM workloads, storage, networking).
before the "cloud"-enterprises actually started
(e.g., more dynamic/scale-out, all sorts of workloads, incl. accelerators).
The word *traditionally* refers to the former, longer period of time.

History: **why** LE 64-bit PowerPC?

1) Linux on IBM PowerPC-based Servers (a.k.a. *Linux on Power*)

traditionally ran on a “different” environment/configuration

than x86_64 or standard things of scale-out computer industry.

(*or “weird” or “esoteric” as some have said – more fun adjectives*)

- Necessarily virtualized (i.e., no bare-metal mode) – OK for some, but
- As LPARs (logical partitions, a.k.a. “guests”, “VMs”) – what?
- In the PowerVM hypervisor (proprietary) – OK for some, but
- Non-”standard” tools/interoperability (HMC for LPAR/system/hardware management – no IPMI, libvirt/XML, not easy to interact/automate) – “how do I do <*something*> here?” (it used to be very “different”).

History: **why** LE 64-bit PowerPC?

2) The *processor wars* (technology battle on CPU clock, cache, pipeline, instructions per cycle, latency/throughput, speculative execution, etc.) were considered “done” and did not pay off (high investment, low return).

Computer industry going through changes.

From processor-level performance (CPU/MEM, clock/cache/bandwidth)
To system-level performance (e.g., that + I/O adapters, accelerators/GPUs)

History: **why** LE 64-bit PowerPC?

But, *traditionally*, Power servers did not have such range of I/O support.

(e.g., storage/networking for sure, but no GPUs nor many accelerators)

- Endianness (big-endian) made it difficult (at least for GPUs)
- Proprietary I/O bus technology (GX++ iirc) made it difficult
- More complex software/hardware interaction
  limited by virtualization requirement/no bare-metal mode
  made it difficult
- Needed to attract more I/O vendors to the platform

History: **why** LE 64-bit PowerPC?

So, *something* happened, ultimately directed at an open ecosystem
for both hardware and software.

- OpenPower Foundation (initially IBM, Google, NVIDIA, Mellanox, Tyan)
- Partnerships in hardware (licensable hardware/POWER8, designs, etc.)
- And software (more industry “open”/”standard”/”compatible” ways)

Then, in that direction,

- Linux scale-out ecosystem (bare-metal, KVM, libvirt, IPMI, BMC,
lots of modern programming languages, cloud environments, etc)
- **Little-endian** 64-bit PowerPC (with the **ELFv2** ABI)
(software/hardware portabilibity, specially GPUs back then.)

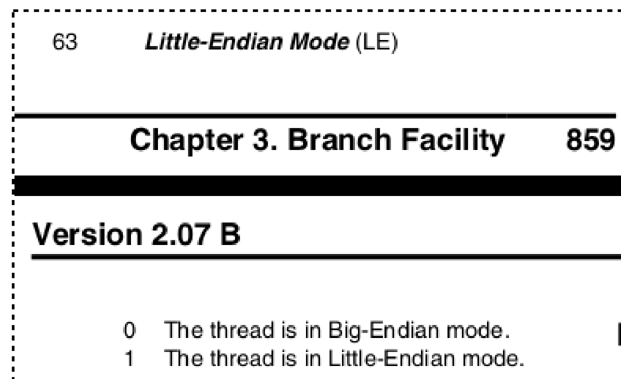
History: **why** LE 64-bit PowerPC?

OK, great! So let's do **little-endian** 64-bit PowerPC!

(endianness: the order in which the individual bytes of multi-byte words are read from / stored to memory – least or most significant byte first.)

These PowerPC processors are *bi-endian* (can run in either LE/BE mode).

So we *just change the endianness mode bit* in this processor register, and...



PowerISA v2.07B

History: **why** LE 64-bit PowerPC?

Uh-oh.

There is no software than runs on it.

(multi-byte memory access is now "*wrong*" for existing software, thus incompatible.)

Now this is a new architecture!

OK.

And let's run **Debian** on it too.

Start the Debian **ppc64el port**.

https://www.debian.org/logos/

History: Why **Debian**?

Ecosystem / Community / Business

1) Debian is one of the favorite Linux “community” distributions
for both development and production systems (good for OpenPower).

2) Ubuntu (based on Debian), similarly (community/devel/production),
and very famous in the cloud space (strategic imperative/move for IBM).

3) Ubuntu has customer support by Canonical; partnership with IBM
for a new Linux distro supported on IBM PowerPC-based servers
(*traditionally*, only RHEL and SLES).

Bootstrap of the **Debian ppc64el** port:

History, **Concepts**, Techniques, and Challenges

What is the **Debian ppc64el** port?

The **Debian** GNU/Linux distribution (e.g., package archive, installer, infra, ...) **ported** (i.e., patched) to the ***little-endian 64-bit PowerPC*** architecture.

This architecture runs on ***IBM / OpenPower servers*** with the ***IBM POWER*** processors (i.e., IBM POWER8 and later); potentially on non-IBM chips too (manufactured by other members/companies in OpenPower foundation.)

Also known as:

- ***powerpc64le*** in platform triplets (CPU field; e.g., ***powerp64le-linux-gnu***),
- ***ppc64le*** in `uname` machine hardware name,
- ***ppc64el*** in `dpkg` architecture.

What is **bootstrap**?

“A looped strap [...] on a boot to help pulling it on.”
(Merriam-Webster Dictionary; noun)

“A means of advancing oneself [...]”

“Self-generating or self-sustaining”

(Dictionary.com; adjective)

https://en.wikipedia.org/wiki/Bootstrapping

What is **bootstrap**?

Computer thing:

Process to (incrementally) build and run software
for/on a computer for which there is no software.

What is **bootstrap**?

Computer thing:

Process to (incrementally) build and run software

for/on a computer for which there is no software.

That is, *from the scratch,* in *incremental steps.*

What is **bootstrap**?

Computer thing:

Process to (incrementally) build and run software
for/on a computer for which there is <u>no</u> software.

That is, *from the scratch,* in *incremental steps*.

(er, from a computer for which there <u>is</u> software.)
(er, target computer for which there <u>is</u> *basic* boot.)

What is **bootstrap**?

Example scenario:

You have a brand new computer.

That runs a brand new computer architecture (e.g., different instructions).

And there is not (yet) any software that can run on it.

And other computers cannot (yet) build for it.

You have to start over.

All over.

What is **bootstrap**?

Example scenario:

You have a brand new computer.

That runs a brand new computer architecture (e.g., different instructions).

And there is not (yet) any software that can run on it.

And other computers cannot (yet) build for it.

You have to start over.

All over.

Build it all -- toolchain, kernel, userspace, applications.

You can only **build** with what you **built**. One step at at time.

Overview of **bootstrap** process

1) Cross Toolchain

First of all, you have to generate (executable) code.

- to the new architecture

- from an existing architecture.

This requires a **cross compiler.**

(actually, **cross toolchain** -- compiler, assembler, linker, ...)

Overview of **bootstrap** process

1) Cross Toolchain (terminology)

A toolchain is built on a build architecture,

it runs (i.e., builds) on a host architecture,

its output runs on a target architecture.

On a "normal" toolchain (native/(ly) built), all architectures are the same.

(For example, built, builds, **and** run on x86_64 or powerpc).

On a cross toolchain, some architectures are different.

(For example, built and builds on x86_64 or powerpc, run on powerpc64le.)

Overview of **bootstrap** process

1) Cross Toolchain

*(first, enable/patch it – a **lot** of work, not covered here.)*

So, build a cross toolchain (**build == host != target**)… *more details soon.*

Once you can generate code for the target architecture,

You can start to think about how to run it.

This requires building an “environment” similar to existing architectures

(that is, general purpose environments, not special purpose/without OS.)

Overview of **bootstrap** process

2) Cross compile the Kernel

(first, enable/patch it – a ***lot*** *of work, not covered here.)*

Well,

first you boot a kernel,

then you can run userspace/applications.

So, cross compile the kernel first.

Overview of **bootstrap** process

3) Cross compile Userspace

So, the kernel boots and hands it off to userspace

(initramfs, init system, login, shell, tools, applications, ...)

So, cross compile “userspace” (a lot of stuff).

Overview of **bootstrap** process

3) Cross compile Userspace

Er, but just enough of it for now (cross compiled).

Once you reach an environment you can actually use (e.g., shell, tools)

You want to build and run in native mode (not cross compile).

Because then you can run test suites (e.g., `make check`)

and hit/fix errors as early as you can (so they don't to propagate; e.g., libs).

And restart it all (*rebuild the world*) if required (e.g., toolchain changes).

Overview of **bootstrap** process

4) Native Toolchain (cross compiled)

You want to build natively, but there is not yet a native toolchain.

So, cross compile a native compiler/toolchain (**build != host == target**).

What? It's confusing at first.

The cross toolchain generates code that runs on the target architecture.

It just happens that code is another toolchain that also does it there.

So you can run a compiler/toolchain in the new environment.

Overview of **bootstrap** process

5) Native Toolchain (natively compiled)

You already have a cross compiled native toolchain.

Now build a natively compiled native toolchain (**build == host == target**).

Because you are not so sure (yet) about the new toolchain...

Whether it runs correctly in the new environment (e.g. bit/endian-ness).

Whether the new environment runs it correctly (kernel/shell/libraries).

Besides, building a toolchain exercises the new environment (a lot).

Useful to hit/fix errors (more of them), ensures things are OK early on.

Overview of **bootstrap** process

6) Build the rest (userspace/applications)

It seems simple, doesn’t it? :-)

You have a toolchain, you have the source code.

Now this should be just “normal”, right?

Well... *not really.*

Overview of **bootstrap** process

6) Build the rest (userspace/applications)

Building stuff just “normal” requires a “normal” environment:

- build dependencies available (incl. circular ones) when you build it,
- and (install/run-time) dependencies available when you install/run it.

When those are not available... there are *quite a few* tricks, hacks, incremental builds (of the same package(s)), and unique procedures to build/install/run stuff... in order to satisfy the dependencies.

This is a real ~~*nightmare*~~ ~~*challenge*~~ ***adventure****!* ;-)

Bootstrap of the **Debian ppc64el** port:

History, Concepts, **Techniques**, and Challenges

QEMU/KVM, powernv & pseries

The very first thing..

An early version of the OPAL firmware (OpenPower Abstraction Layer),
for POWER7+ systems (not publicly available, only with legal agreement),
which were available for the development teams (POWER8 not there yet).

It allows the system to boot/run in bare-metal mode (and run KVM guests!)
as an alternative to the PowerVM hypervisor (in LPAR/virtualized mode).

QEMU/KVM, powernv & pseries

The bare-metal "platform" name is **powernv** (non-virtualized),
provided by the OPAL firmware, capable of running KVM guests.

The virtualized "platform" name is **pseries** (power),
provided/emulated by QEMU, as in the PowerVM hypervisor.

This provided backward-compatibility for existing Linux distros
(and most of the software) supported on BE 64-bit PowerPC,
originally under the PowerVM hypervisor (pseries), now under QEMU/KVM.

VM: *faster* dev/test/crash/reboot/debug cycles – good for development.

Cross Build on Debian **powerpc** port

KVM guest running the **Debian *powerpc* port** (BE 32-bit), on top of
KVM host running Fedora/powerpc64 (BE) with patched kernel/qemu/etc.

This is used to **cross** build a minimal **native** build/host environment.

Reasons for Debian as (**cross**) build/host environment:
- maximize patch reuse (e.g., toolchain/pkg builds on build/host and target)
- minimize differences between build/host and target environment
- simpler/more common build scripts (e.g., dpkg, apt, etc.)

git repo: scripts, patches, workarounds

1) Build scripts

- Part **0** (cross): Prepare build/host environment (dpkg arch/multi-arch)
- Part **1** (cross): Build cross toolchain
- Part **2** (cross): Build enough to boot and build (natively)
- Part **3** (native): Build enough to rebuild the toolchain (natively).
- Part **4** (native): Build *more* (buildd, missing build deps, FTBFS, etc.)

2) Patches

- **pkg/.patch** files – source code changes (i.e., endianness/abi fixes)
- **pkg/.debpatch** files – packaging changes (i.e., files in the **debian/** dir)

git repo: scripts, patches, workarounds

3) Workarounds

- Variables for per-package workarounds: **$<pkg>_*deps|vars|version***
  - ***$<pkg>_deps***: remove specific build dependencies
  - ***$<pkg>_vars***: export environment variables during build
  - ***$<pkg>_version***: specific source code version to build (e.g., simpler)

- Multiple package build stages: **<pkg>_*stageN***
  (the *"_" char is not allowed in Debian package names; used here as separator*)
  - Different build stages for different workarounds (decreasingly)
  - Variables: ***$<pkg>_stageN_deps|vars|version***

- Package array: ***pkgs=(pkg1 pkg2_stage1 pkg3 pkg2 ... pkgN)***
- Package build loop: ***for pkg in $pkgs; do <patch/work'd/build $pkg>; done***

Part 0: **prepare** build/host environment

Steps:

1) Build/Install **dpkg** with patch to add **ppc64el** architecture
2) **dpkg –add-architecture ppc64el**
   (*multiarch – to support installing target packages on build/host system)*
3) **dpkg-architecture –host-arch ppc64el**
   (***DEB_{BUILD,HOST,TARGET}_**** env vars for the cross binutils/gcc & glibc)
4) Package repository for target packages

Part 1: build **cross toolchain**

Steps:

1) **binutils** (*assembler, linker, etc.*)
2) **gcc** stage 1 – gcc to build the glibc headers/start files only

   it does not need glibc

   it can build parts of glibc (not the complete glibc)
3) **glibc** stage 1 – glibc to build the gcc that can build the complete glibc

   it is headers and start files (C runtime)

   it needs kernel headers (C types, system calls, errno, threads, etc.)
4) **gcc** stage 2 – gcc to build the complete glibc
5) **glibc** stage 2 – complete/final glibc
6) **gcc** stage 3 – complete/final gcc that can use/link to complete glibc

Suggested reading:
https://crosstool-ng.github.io/docs/toolchain-construction
https://developers.redhat.com/blog/2014/12/19/bootstrapping-power8-little-endian-and-common-pitfalls/
https://www.airs.com/blog/archives/492

Part 2: build enough to **boot** and **build**

The **deboostrap** tool installs a basic Debian **suite** in a **dir/** from a package **repository**.

Next steps come from its package-set **variants** and the package install **order/errors**.

cross-env # debootstrap \

--arch=ppc64el \ *(set target architecture)*

--foreign \ *(unpack only, do not configure/run anything; see --second-stage)*

--variant=minbase|buildd|<skip> \ *(essential+apt | build-essential | base install; **fix errors**)*

--include=... \ *(dhcp, ssh, vim, curl)*

unstable \ (package suite/release)

./rootfs \ (mount point for disk image to boot)

file:///pkg-repo (package repository)

native-env # /debootstrap/debootstrap –second-stage *(configure packages)*

Part 2: build enough to **boot** and **build**

Prepare environment to boot:

- Create disk image file (dd if=/dev/zero of=rootfs.ext4 ...)
- Create its filesystem (mkfs.ext4 -F rootfs.ext4)
- Mount it (mount -o loop ...)
- Create /etc/inittab to run a first-boot script
- Create script (remove itself, run debootstrap –second-stage, set password, poweroff.)
- Get a kernel image
- Boot kernel + disk image with qemu-kvm (first-boot sets environment up)
- The native environment (disk image) is ready.
- Boot it and go build more stuff!

See **wiki.debian.org/ppc64el/Installation** for details.

Part 3: build enough to **build toolchain**

Now that the native environment is ready and usable...

Build the build-dependencies of the toolchain packages, and build it.

Install it so it's used as the default toolchain (and update the disk image).

Part 4: build **more**...

What to build now? Where to go?

The Build Daemon (*buildd*).

It automatically builds packages, and reports failures:

- Missing build dependencies
- Problems installing build dependencies
- Other problems during build (FTBFS – *fail(ure) to build from source*)

Set it loose (build/following Debian Archive news) and fix the errors it reports.

Initially internal/downstream, using our patches and package repository.

Later, using our package repository for build-deps only, build from Debian sources.

Bootstrap of the **Debian ppc64el** port:
History, Concepts, Techniques, and **Challenges**

Source packages change quickly

During the early bootstrap process, with a smaller team,

it was hard to keep up with the changes in the Debian archive

(unstable/sid sources, it’s where we had to live on).

Changes like (build) deps versions broke our repository.

Changes like code update/rebase broke our patches (internal-only then).

Some **packages stop installing**. (bad, but a rebuild may fix it.)

Some **patches stop applying**. (worse! work required to rebuild.)

Source packages change quickly

The **workaround** used is **snapshot.debian.org** as source/binary repository.

This allowed us to stick to source code/deps of a given point in time.

And port consistently (without breakages), until ready / good enough

to update to a more recent snapshot.

Eventually, with buildd, we moved to the rolling archive / non-snapshot.

Build deps: circular/cyclic

For example..

Let's build Package A.

Package A *build-depends* on Package B (i.e., to build A, you first must build/install B)

Ok, let's build Package B *first*.

Package B *build-depends* on Package A.

Ok, let's build Package A *firs*..

Eh? Package A *again*?! Uh-oh.

Build deps: circular/cyclic

In bootstrapped architectures, all packares are available (e.g., previous versions).

In bootstrapping/new architectures, *that is not the case*.

That is a trivial example (but it happens).

Some packages might get *very complex* to analyze and resolve

- Introduce more packages: A -> B -> C -> D -> (...) -> A
- Inner loops / multiple circular build-deps: A -> B -> C -> D -> C -> (...) -> A

Build deps: circular/cyclic

The **workaround** is to *identify and remove not strictly required* build deps (e.g., packages that do not cause build failures, just a "*no*" in ***./configure***), then build both packages, and rebuild the first (build deps available now.)

packages=(

pkgA_stage1 *# build A without B*

pkgB *# build B*

pkgA *# rebuild A now with B*

)

pkgA_*stage1_deps*="pkgB" # remove pkgB from build-deps of packageA

build $packages

Build deps: recursive/self-hosted

It happens that a package *build-depends* on ***itself***.

For real; a strictly necessary build-dep; can't remove it.

This is the case with several programming languages.

For example, compiler of language X *is written in* language X.

So, it needs a compiler of language X to build (i.e., self-hosted).

Build deps: recursive/self-hosted

The **workaround**(s) for that are usually more difficult.

It might involve one or multiple steps:

- Build a simpler version, and use that to build a better one. (e.g., gcc, glibc)
- Inject an externally/manually built binary, and ignore build-deps.
- Build the compiler using alternative compilers (real example!)

Build deps: recursive/self-hosted

Build the compiler using alternative compilers (real example!)

- Target: OpenJDK Java 1.7, needs "Java" 1.7 (not necessarily OpenJDK)

- Step 1: Use **GCC** to build **GCJ** for Java **1.5** (GNU Compiler for Java)
- Step 2: Use it to build **GCJ/1.6 (or OpenJDK Java 1.5 ?)**
- Step 3: Use it to build **GCJ/1.7 (or OpenJDK Java 1.6 ?)**
- Step 4: Use it to build **OpenJDK Java 1.7**

Build problems: libtool, autoreconf

Several packages depend on **libtool** to detect the architecture,

and configure the respective build options, etc. for libraries.

Several packages ship their *own (outdated) copy* of libtool files, and use it.

This led to several problems (not always seen as build errors, unfortunately)

when the package did not detect the **little-endian** architecture correctly,

and fail to build a shared library in **big-endian** mode (thus disable building it).

Usually, the problem is shared libraries missing; not all packages build-check.

Build problems: libtool, autoreconf

The **solution** is to run **autoreconf**, which uses the (*updated*) system libtool.

There is even a **debhelper** option to do that automatically.

However, some packages *fail to build* with that.

(e.g., need other fixes, depend on older versions of something.)

One **workaround** is to patch just the lines to detect **ppc64el** right.

Another **workaround** (*never upstream*) is a **linker** patch, to *force* linkage

in little-endian mode even if big-endian mode is specified (**LD_FORCE_LE=1**).

Rebuild the World, twice: **ABI, GLIBC**

Rebuild the world means to rebuild **everything**, to start over.

It usually happens in case of changes to the *toolchain, ABI, C library* that affect all software that has already been built.

We did that **twice**.

You should have most of your bootstrap process automated.

Just saying.

Rebuild the World, twice: **ABI, GLIBC**

First time: the introduction of the **ELFv2 ABI** for LE 64-bit PowerPC.

The ELFv1 ABI, used in **BE** 64-bit PowerPC, dated 20ish years ago.

Designed in different paradigms for programming languages, programs,

even the hardware itself (e.g., long running loops, long functions, few calls).

More recent software/programming languages are more modular/shorter.

But it cannot change so not to break ABI compatibility (20ish years/programs).

Hey! The **LE** 64-bit PowerPC was totally new, no software built and published.

The ABI **could** change!

Rebuild the World, twice: **ABI, GLIBC**

The ELFv2 ABI, used in **LE** 64-bit PowerPC, builds on top of the ELFv1, with some changes to address modern programming languages, shorter and more functions, smaller stack size, and a lot of cool stuff.

The initial "build the world" used the ELFv1 ABI, so to validate toolchain/libc/kernel/etc changes for endianness, platform, firmware, etc (a lot of stuff already) alone.

The first rebuild the world used the ELFv2 ABI, once the components were considered good enough/stable.

Rebuild the World, twice: **ABI, GLIBC**

Second time: the change in **GLIBC** start version for LE 64-bit PowerPC.

This symbol is carried in executables/libraries that use the C library (“all”).

The Linux distros initially supporting LE 64-bit PowerPC (Ubuntu, SLES)

used GLIBC 2.18, but later RHEL joined too! and it used GLIBC 2.17.

In order to keep binaries compatible/consistent across the architecture,

that version number had to change. It was a bigger problem as some

Linux distro(s) had to rebuild the world too, due to another Linux distro.

Cool patches from RedHat to “fake” GLIBC version on (re)build/run-time!

Small team of non-Debian Developers

Our whole team (< 10 always) had little to no experience with Debian at all.

It's been a great and hard learning time, with help from Debian community.

Later on, around the end of 2013, the people at Canonical involved in the bootstrap of Ubuntu Server 14.04 for ppc64el, who had plenty and great experience with Debian, took a good share of the work in many packages.

This helped tremendously.

They helped us with upstreaming our work to Debian (new process to us), and we helped with upstreaming to Debian for them too, later on.

Several Debian community/maintainers were very supportive as well.

Timeline – from dpkg arch to release

- 2013/06/13 (zero): internal: patch adds ppc64el to dpkg/cputable
- 2013/08/06 (~2mo): debian: patch adds ppc64el to dpkg/cputable [1]
- 2013/08/26 (~2mo): IBM: POWER8 announce: Hot Chips conference [2]
- <*lots of work at IBM, Debian community, and Canonical*>
- 2014/04/17 (~10mo): ubuntu-announce: Ubuntu Server 14.04 LTS [3]
- 2014/06/10 (~12mo): IBM: POWER8 general availability [4]
- 2014/06/12 (~12mo): debian/linux: patch adds ppc64el kernel support [5]
- 2014/08/27 (~14mo): debian-devel-announce: ppc64el in unstable [6]
- 2014/09/12 (~15mo): buildd.debian.org/stats: ppc64el builds 90%+ of archive
- 2014/09/26 (~15mo): debian-devel-announce: ppc64el in testing [7]
- 2014/10/05 (~16mo): debian-devel-announce: ppc64el in debian-installer [8]
- 2015/04/26 (~22mo): debian-announce: ppc64el in Debian 8 "Jessie" release [9]

[1] https://bugs.debian.org/718945
[2] https://www.hotchips.org/archives/2010s/hc25/
[3] https://lists.ubuntu.com/archives/ubuntu-announce/2014-April/000182.html
[4] https://www-03.ibm.com/press/us/en/pressrelease/44123.wss
[5] https://salsa.debian.org/kernel-team/linux/commit/57356b0a9edf
[6] https://lists.debian.org/debian-devel-announce/2014/08/msg00012.html
[7] https://lists.debian.org/debian-devel-announce/2014/09/msg00002.html
[8] https://lists.debian.org/debian-devel-announce/2014/10/msg00002.html
[9] https://lists.debian.org/debian-announce/2015/msg00001.html

Takeaway

"… the obstacles in the path

are not obstacles;

they are the path."

– Jane Lotter

Bootstrap of the **Debian ppc64el** port:
History, Concepts, Techniques, and Challenges

Mauricio Faria de Oliveira
<mauricio.foliveira@gmail.com>

Linux Developer Conference Brazil
August 26th, 2018

This author/presentation does not represent any company or its views.