Building Container Images with OpenEmbedded and the Yocto Project

Scott Murray

scott.murray@konsulko.com

Konsulko Group

About Me

- Linux user/developer since 1996
- Embedded Linux developer starting in 2000
- Principal Software Engineer at Konsulko Group
- Konsulko Group
  - Services company specializing in Embedded Linux and Open Source Software
  - Hardware/software build, design, development, and training services.
  - Based in San Jose, CA with an engineering presence worldwide
  - https://konsulko.com

Agenda

- Quick overview of OpenEmbedded / Yocto Project
- Containers
- What can OE bring to the table?
- Example OE container build configurations
  - Full distribution and application containers
  - Nesting images (pre-installed application sandboxes)

Caveats

- I am not a container expert, and this presentation does not cover the mechanics of using the discussed container images in detail
- Container technology is progressing rapidly, it’s entirely possible I’ve missed something of interest (Please let me know!)
- An intermediate level of OpenEmbedded / Yocto Project knowledge is assumed

OpenEmbedded & The Yocto Project

- OpenEmbedded (OE) is a build system and associated metadata to build embedded Linux distributions.
- The Yocto Project (YP) is a collaboration project founded in 2010 to aid in the creation of custom Linux based systems for embedded products. It is a collaboration of many hardware and software vendors, and uses OpenEmbedded as its core technology. A reference distribution called “poky” (pock-EE) built with OE is provided by the Yocto Project to serve as a starting point for embedded developers.

Notable OE / YP Features

- Broad CPU architecture support
- Strong vendor support
- Highly customizable, layered configuration metadata
- Focus on constrained embedded devices, so support for small images
- Regular release schedule
- Integrated license and source publishing compliance tools
- Working towards full binary reproducibility

Containers

- Operating system level virtualization as opposed to virtual machines
- Linux implementations typically are based on namespaces and cgroups
  - LXC
  - Docker
  - runc
  - systemd-nspawn
- Newer Clear / Kata containers are based on lightweight VM technology
- Container images can be full Linux distribution installs, or small images containing a single application and its dependencies

Konsulko Group

Containers (continued)

- Common use cases:
  - Running an application that has incompatible dependencies from the host machine
  - Sandboxing an application to isolate it from the host machine
  - Implementing microservices where application containers are started based on demand
- Typical container construction
  - Start with a minimal Debian, Ubuntu, or Alpine Linux image
  - Add required packages
  - Potentially compile non-upstream available packages (e.g. via Dockerfile commands)
  - Prune container down by removing unneeded files
    - Small size is very desirable
    - Reduces security attack surface, maintenance, and migration time

Container Drawbacks?

- Reproducibility
  - Base containers changes may not be obvious, e.g Docker labels may change
  - Package versions on Debian, Alpine, etc. changing
    - It’s not uncommon to see “apt-get update && apt-get upgrade -y”, etc. in Dockerfiles
    - Pinning package versions can break if the base distro doesn’t archive older versions
  - Even if automating with Dockerfile(s) or other scripting, effort required to ensure result is reproducible
- Transparency / Security
  - You have to trust the builders of the base container
  - Security is dependent on the providers of the base container, i.e. distribution update policies
  - Often quoted problem of library updates potentially affecting many containers

Konsulko Group

Container Drawbacks? (continued)

- License compliance scheme
  - Potentially can be pulled from package manager, but no particularly turn-key solutions
- Customization
  - Patching a package or tweaking its configuration flags requires manual or scripted rebuild
  - Building for an unsupported architecture requires delving into the distribution’s build process

So is OE / YP a solution?

- Reproducibility
  - Image builds can be straightforwardly reproduced using fixed metadata
- Transparency / Security
  - Entire build process is bootstrapped from scratch
  - Typically 18 months support per release versus 5 years for Debian stable, ~2 years for Alpine
- License compliance scheme
  - Image license manifests and license text archiving
  - Source archiving
- Customization
  - Layered metadata and build process allows adding almost any customization
  - Any architecture with a BSP layer can be targeted

So is OE / YP a solution? (continued)

- Package availability
  - Debian, Ubuntu several 10’s of K, Alpine ~5K
  - OE ~2300 in oe-core and meta-openembedded, many more in other layers
  - OE node.js and Python module availability is not as broad
- Ease of use
  - It’s possible, but quite involved to reproduce something like the apt-get, apk install user experience with an OE built package feed
  - Small, relatively fixed content images are going to be easier to handle
- Resources
  - OE is a new toolset to learn
  - Building images can require significant hardware resources
  - Long term maintenance may involve dedicating resources

OE / YP container support

- Container image type
  - Added in pyro / 2.3 release
  - IMAGE_FSTYPES = “container”
  - Produces a tar.bz2 with no kernel components or post-install scripts
  - Required PREFERRED_PROVIDER_virtual/kernel to be set to “dummy”
- meta-virtualization layer
  - Provides
    - LXC, runc, Docker (currently 18.03.0 in master/thud and sumo branches)
    - OCI image-tools
    - Kernel configuration fragments for linux-yocto
  - Currently no support for building OCI / Docker images during OE build
    - Difficult with Docker itself, since it needs its daemon running
    - Still investigating this myself, open to suggestions

OE / YP container support (continued)

- Togán Labs' Oryx Linux
  - Commercially supported OE based distribution
  - Container support using runc on target
  - https://www.toganlabs.com/oryx-linux/

Examples

- Build bootstrap container
  - Contains the tools to run OE / YP builds, i.e. self-hosting
  - Lighter container version of build-appliance VM image
- Alpine-like container image
  - Attempt to match base contents and size
- Application container image
  - Typical microservice single application
- Nested application sandbox
  - A host image built with container tools and pre-loaded with application container(s)

Build Bootstrap Container Example

Quick and dirty with local.conf

```
MACHINE = “qemux86-64”
IMAGE_FSTYPES = "container"
PREFERRED_PROVIDER_virtual/kernel = "linux-dummy"
IMAGE_LINGUAS_append = " en-us"
CORE_IMAGE_EXTRA_INSTALL += "packagegroup-self-hosted-sdk packagegroup-self-hosted-extended"
```

Notes

- Resulting core-image-minimal for qemux86-64 is ~150 MB
- Builds some graphical packages that go unused
- Further tinkering required to prune out some things
- Lack of post-install scripts means volatile directories (/var/volatile/*, etc.) do not get created
    - Can run /etc/rcS.d/S37populate-volatile.sh
    - Fixable with ROOTFS_POSTPROCESS or bbappend to base-files and fsperms.txt tweaking
- User for building needs to be created / managed
- Access to build tree needs to be managed
    - Docker volume(s), mounts, etc.

Image definition: build-container.bb

```
SUMMARY = "A minimal bootstrap container image"

IMAGE_FSTYPES = "container"

inherit core-image

IMAGE_INSTALL = " \
        packagegroup-core-boot \
        packagegroup-self-hosted-sdk \
        packagegroup-self-hosted-extended \
        ${CORE_IMAGE_EXTRA_INSTALL} \
"

IMAGE_LINGUAS = "en-us"
IMAGE_TYPEDEP_container += "ext4"

# Workaround /var/volatile for now
ROOTFS_POSTPROCESS_COMMAND += "rootfs_fixup_var_volatile ; "

rootfs_fixup_var_volatile () {
        install -m 1777 -d ${IMAGE_ROOTFS}/${localstatedir}/volatile/tmp
        install -m 755 -d ${IMAGE_ROOTFS}/${localstatedir}/volatile/log
}
```

Convenience MACHINE definition: containerx86-64.conf

```
require conf/machine/qemux86-64.conf

PREFERRED_PROVIDER_virtual/kernel = "linux-dummy"

MACHINE_ESSENTIAL_EXTRA_RDEPENDS = ""
```

Alpine-like Container Example

Quick and dirty with local.conf

```
MACHINE = “qemux86-64”
IMAGE_FSTYPES = "container"
PREFERRED_PROVIDER_virtual/kernel = "linux-dummy"
TCLIBC = “musl”
```

Resulting image manifest

```
base-files qemux86_64 3.0.14
base-passwd core2_64 3.5.29
busybox core2_64 1.29.2
busybox-hwclock core2_64 1.29.2
busybox-syslog core2_64 1.29.2
busybox-udhcpc core2_64 1.29.2
eudev core2_64 3.2.5
init-ifupdown qemux86_64 1.0
initscripts core2_64 1.0
initscripts-functions core2_64 1.0
libblkid1 core2_64 2.32.1
libkmod2 core2_64 25+git0+aca4eca103
libuuid1 core2_64 2.32.1
libz1 core2_64 1.2.11
modutils-initscripts core2_64 1.0
musl core2_64 1.1.20+git0+c50985d5c8
netbase core2_64 5.4
packagegroup-core-boot qemux86_64 1.0
sysvinit core2_64 2.88dsf
sysvinit-inittab qemux86_64 2.88dsf
sysvinit-pidof core2_64 2.88dsf
update-alternatives-opkg core2_64 0.3.6
update-rc.d noarch 0.8
v86d qemux86_64 0.1.10
```

Konsulko Group

Notes

- Resulting core-image-minimal for qemux86-64 is ~4.8 MB
  - ~8.5 MB with package management support via opkg
  - Almost 100 MB with package management support via rpm / dnf
- Further pruning is possible
  - Custom distro configuration
  - Set FORCE_RO_REMOVE to remove update-alternatives, etc. if not using package management

Example custom distro configuration: schooner.conf

```
require conf/distro/poky.conf

DISTRO = "schooner"
DISTRO_NAME = "Schooner"
DISTRO_VERSION = "1.0-${DATE}"
DISTRO_CODENAME = "master"
SDK_VENDOR = "-schoonersdk"

MAINTAINER = "Scott Murray <scott.murray@konsulko.com>"

TARGET_VENDOR = "-schooner"

TCLIBC = "musl"

DISTRO_FEATURES = "acl ipv4 ipv6 largefile xattr ${DISTRO_FEATURES_LIBC}"

VIRTUAL-RUNTIME_dev_manager ?= ""
VIRTUAL-RUNTIME_login_manager ?= ""
VIRTUAL-RUNTIME_init_manager ?= ""
VIRTUAL-RUNTIME_initscripts ?= ""
VIRTUAL-RUNTIME_keymaps ?= ""
```

Konsulko Group

Application Container Example

Base application image: app-container-image.bb

```
SUMMARY = "A minimal container image"
LICENSE = "MIT"
LIC_FILES_CHKSUM = "file://${COREBASE}/meta/COPYING.MIT;md5=3da9cfbcb788c80a0384361b4de20420"

IMAGE_FSTYPES = "container"

inherit image

IMAGE_TYPEDEP_container += "ext4"

IMAGE_FEATURES = ""
IMAGE_LINGUAS = ""
NO_RECOMMENDATIONS = "1"

IMAGE_INSTALL = " \
        base-files \
        base-passwd \
        netbase \
"

# Workaround /var/volatile for now
ROOTFS_POSTPROCESS_COMMAND += "rootfs_fixup_var_volatile ; "

rootfs_fixup_var_volatile () {
        install -m 1777 -d ${IMAGE_ROOTFS}/${localstatedir}/volatile/tmp
        install -m 755 -d ${IMAGE_ROOTFS}/${localstatedir}/volatile/log
}
```

lighttpd application image: app-container-image-lighttpd.bb

```
SUMMARY = "A lighttpd container image"
LICENSE = "MIT"
LIC_FILES_CHKSUM = "file://${COREBASE}/meta/COPYING.MIT;md5=3da9cfbcb788c80a0384361b4de20420"

require app-container-image.bb

# Note that busybox is required to satisfy /bin/sh requirement of lighttpd,
# and the access* modules need to be explicitly specified since RECOMMENDATIONS
# are disabled.
IMAGE_INSTALL += " \
        busybox \
        lighttpd \
        lighttpd-module-access \
        lighttpd-module-accesslog \
"
```

Konsulko Group

Resulting image manifest

```
base-files qemux86_64 3.0.14
busybox core2_64 1.29.2
libattr1 core2_64 2.4.47
libcrypto1.1 core2_64 1.1.1
libpcre1 core2_64 8.42
lighttpd core2_64 1.4.50
lighttpd-module-access core2_64 1.4.50
lighttpd-module-accesslog core2_64 1.4.50
lighttpd-module-dirlisting core2_64 1.4.50
lighttpd-module-indexfile core2_64 1.4.50
lighttpd-module-staticfile core2_64 1.4.50
musl core2_64 1.1.20+git0+c50985d5c8
netbase core2_64 5.4
```

nginx application image: app-container-image-nginx.bb

```
SUMMARY = "A nginx container image"
LICENSE = "MIT"
LIC_FILES_CHKSUM = "file://${COREBASE}/meta/COPYING.MIT;md5=3da9cfbcb788c80a0384361b4de20420"

require app-container-image.bb

IMAGE_INSTALL += "nginx"

# Add /var/log/nginx and /run/nginx
ROOTFS_POSTPROCESS_COMMAND += "rootfs_add_nginx_dirs ; "

rootfs_add_nginx_dirs () {
        install -m 755 -d ${IMAGE_ROOTFS}/${localstatedir}/log/nginx
        install -m 755 -d ${IMAGE_ROOTFS}/run/nginx
}
```

Notes

- bash may get pulled into images because of script detection during packaging
- If the application expects to exec /bin/sh, busybox may need to be added manually as a dependency
- The lack of post-install scripts means some tweaking may be required to e.g. create volatile directories

Nested Application Sandbox Example

Motivation

- So far we’ve been building container images on their own
- Useful for “docker import” on target, or “docker compose”, etc., then fetching over the network to target
- What if we wanted to build a container image into a target image for a device?
    - Building factory images for devices running application sandboxes
- Somewhat constrained by tooling
    - Currently only systemd-nspawn seems straightforwardly doable
    - Other systems might be supported by using post-install scripts to import container images

Approaches

- Simple nesting
  - Based on method outlined by Jérémy Rosen in “Yoctoception: Containers in the embedded world”: https://www.slideshare.net/ennael/embedded-recipes-2018-yoctoception-containers-in-the-embedded-world-jrmy-rosen
  - Restricted to common MACHINE, DISTRO, TCLIBC configuration
- Multiconfig based approach
  - More flexibility with respect to different configuration between host and container images
  - https://www.yoctoproject.org/docs/latest/dev-manual/dev-manual.html#dev-building-images-for-multiple-targets-using-multiple-configurations
  - Caveat that multiconfig dependencies are a recent addition to OE

Konsulko Group

Nesting - Simple Example

lighttpd container recipe: app-container-lighttpd.bb

```
SUMMARY = "Package lighttpd app container image"
LICENSE = "MIT"
LIC_FILES_CHKSUM = "file://${COREBASE}/meta/COPYING.MIT;md5=3da9cfbcb788c80a0384361b4de20420"

DEPENDS = "app-container-image-lighttpd"

FILESEXTRAPATHS_prepend = "${DEPLOY_DIR}/images/${MACHINE}:"

SRC_URI = "file://app-container-image-lighttpd-${MACHINE}.ext4"
SRC_URI[md5sums] = ""

do_fetch[deptask] = "do_image_complete"
do_compile[noexec] = "1"

do_install () {
    install -d ${D}/var/lib/machines
    install ${WORKDIR}/app-container-image-lighttpd-${MACHINE}.ext4 ${D}/var/lib/machines
}

RDEPENDS_${PN} += "systemd-container"
```

Host system image: container-host-image.bb

```
SUMMARY = "A minimal container host image"
LICENSE = "MIT"
LIC_FILES_CHKSUM = "file://${COREBASE}/meta/COPYING.MIT;md5=3da9cfbcb788c80a0384361b4de20420"

inherit core-image

IMAGE_INSTALL = " \
        packagegroup-core-boot \
        app-container-lighttpd \
"
```

Nesting - Multiconfig Example

local.conf

```
BBMULTICONFIG = "host container"
```

multiconfig/host.conf

```
MACHINE = "qemux86-64"
DISTRO_FEATURES_append = " systemd"
DISTRO_FEATURES_BACKFILL_CONSIDERED = "sysvinit"
VIRTUAL-RUNTIME_init_manager = "systemd"
VIRTUAL-RUNTIME_initscripts = ""
```

multiconfig/container.conf

```
MACHINE = "containerx86-64"
DISTRO = "schooner"
TMPDIR = "${TOPDIR}/tmp-container"
```

lighttpd container recipe: app-container-lighttpd-multiconfig.bb

```
SUMMARY = "Package lighttpd app container image"
LICENSE = "MIT"
LIC_FILES_CHKSUM = "file://${COREBASE}/meta/COPYING.MIT;md5=3da9cfbcb788c80a0384361b4de20420"

do_compile[noexec] = "1"

do_install[mcdepends] =
"multiconfig:host:container:app-container-image-lighttpd:do_image_complete"

do_install () {
    install -d ${D}/var/lib/machines
    install ${TOPDIR}/tmp-container/${DEPLOY_DIR_IMAGE}/app-container-image-lighttpd.ext4 \
        ${D}/var/lib/machines
}

RDEPENDS_${PN} += "systemd-container"
```

Host system image: container-host-image-multiconfig.bb

```
SUMMARY = "A minimal container host image"
LICENSE = "MIT"
LIC_FILES_CHKSUM = "file://${COREBASE}/meta/COPYING.MIT;md5=3da9cfbcb788c80a0384361b4de20420"

inherit core-image

IMAGE_INSTALL = " \
        packagegroup-core-boot \
"

do_image[mcdepends] = "multiconfig:host:container:app-container-image-lighttpd:do_image_complete"

ROOTFS_POSTPROCESS_COMMAND += "rootfs_install_container ; "

rootfs_install_container () {
    install -d ${IMAGE_ROOTFS}/${localstatedir}/lib/machines
    install ${TOPDIR}/tmp-container/deploy/images/${MACHINE}/app-container-image-lighttpd-${MACHINE}.ext4 \
        ${IMAGE_ROOTFS}/${localstatedir}/lib/machines
}
```

Notes

- I hit a couple of multiconfig issues experimenting that need some investigation
  - Had to change TMPDIR when TCLIBC differed between host and container configs
  - multiconfig dependency works when used in an image recipe per documentation, but currently seems a bit fragile, saw failures in non-image recipe
- multiconfig shows a lot of promise due to the flexibility it gives

Questions?