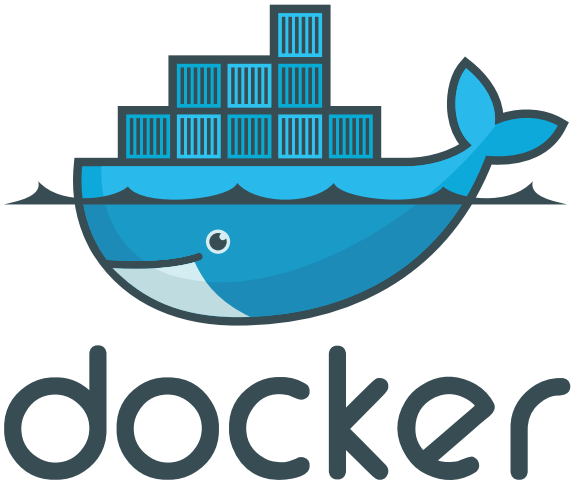

MSE | MASTER OF SCIENCE IN ENGINEERING

Hes·so
Haute Ecole Spécialisée de Suisse occidentale
Fachhochschule Westschweiz
University of Applied Sciences and Arts Western Switzerland

État de l'art à la mi-projet de semestre Docker and embedded systems - Ou comment ne pas cross compiler Docker sur ARM

Auteur :
Gary Marigliano

Encadrant :
Jean-Roland Schuler

Contact :
gary.marigliano@master.hes-so.ch

Mandant :
Haute École d'ingénierie et d'architecture de Fribourg

Version 0.0.3
10 juin 2016

Historique

Version	Date	Auteur(s)	Modifications
0.0.1	15.04.16	Gary Marigliano	Création du document
0.0.2	01.05.16	Gary Marigliano	Modifications première page, ajout historique, repositionnement des images
0.0.3	21.05.16	Gary Marigliano	Amélioration de la lisibilité générale

Table des matières

1. Introduction

1.1 Contexte

Ce document s'inscrit dans le cadre du projet de semestre Docker and embedded systems actuellement réalisé par moi-même. Un des buts de ce projet est de cross compiler Docker à partir de ses sources pour produire un binaire exécutable sur un Odroid XU3 (ARMv7).

Lien : `https://github.com/krypty/docker_and_embedded_systems`

Il est important de noter que la vitesse de développement de Docker est assez hallucinante. En effet, sur Github (`https://github.com/docker/docker`) les commits se succèdent à vitesse grand V. Entre chaque version de Docker qui sortent environ tous les mois, il est courant d'avoir plus de 3000 commits qui ont été *pushés*. Tout ceci pour dire qu'à la lecture de ce document, il est quasiment sûr que certaines pistes explorées soient définitivement obsolètes ou au contraire deviennent la voie à suivre due à une mise à jour quelconque.

Le chapitre 2 présente Docker afin de pouvoir comprendre les chapitres suivants.

1.2 Objectifs

De manière plus précise, ce projet vise à maîtriser les parties suivantes :

1. Construction d'un système Linux capable de faire tourner Docker et son *daemon* en utilisant Buildroot. Pour générer ledit système, on dispose d'un *repository* Gitlab hébergé à la Haute École d'ingénierie et d'architecture de Fribourg
2. Cross compilation de Docker et de son *daemon*, capable de faire tourner des containers

L'objectif de ce document est d'énumérer les différentes techniques tentées pour (cross-)compiler Docker sur une cible ARM. De cette manière, le lecteur, en cas de reprise du projet ou par simple curiosité, aura une idée des pistes à explorer ou à éviter.

2. Présentation de Docker

2.1 Introduction

Le but de ce chapitre est d'apporter un contexte au projet réalisé afin de comprendre ce qui est expliqué dans la suite de ce document. Si le lecteur souhaite connaître les tréfonds de Docker, il est recommandé de lire le rapport "Docker and Internet of Things (travail de semestre)" de M. Loic Bassang [5].

Docker est un outil qui permet d'empaqueter une application et ses dépendances dans un container léger, autosuffisant, isolé et portable. En intégrant leur application dans un container, les développeurs s'assurent que celle-ci va tourner sur n'importe quel environnement GNU/Linux. Ainsi, le temps passé à configurer les différents environnements (développement, test et production typiquement) est réduit, voire même unifié[6][2][4].

Les caractéristiques principales des containers Docker sont les suivantes :

- *légers* : les containers partagent le même kernel et librairies que le système d'exploitation hôte permettant ainsi un démarrage rapide des containers et une utilisation mémoire contenue
- *isolés* : les containers sont isolés grâce à des mécanismes offerts par le kernel. Voir section 2.5 pour plus de détails
- *éphémères et maintenables* : les containers sont conçus pour être créés et détruits régulièrement contrairement à un serveur ou une machine virtuelle pour lesquels un arrêt est souvent critique. Ils sont maintenables dans le sens où il est possible de revenir à une version précédente facilement et qu'il est aisé de déployer une nouvelle version

Les sections qui suivent abordent un peu plus en profondeur certains points clés de Docker qui ont été jugés importants.

2.2 Containers vs machines virtuelles

Suite à la lecture de la section précédente, il serait normal de se dire que ces containers partagent certaines caractéristiques avec les machines virtuelles.

Voici les différences majeures qu'il existe entre les containers et les machines virtuelles[4].

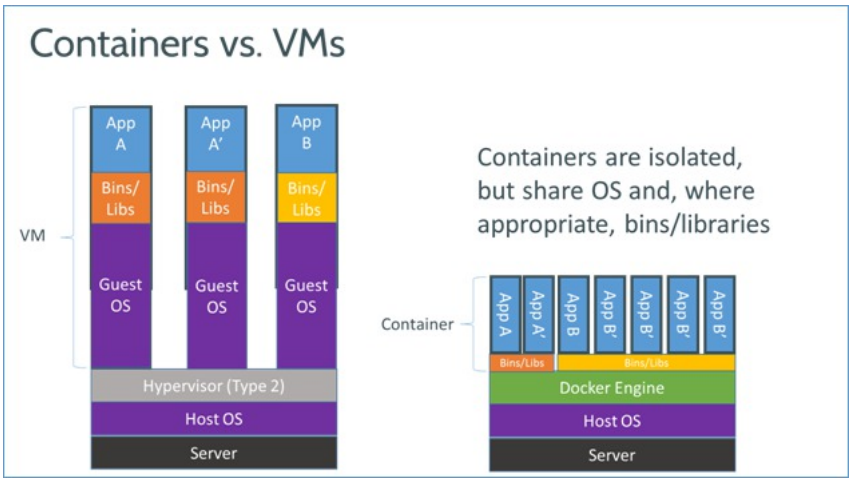

FIGURE 2.1 – Machines virtuelles vs Containers

Les machines virtuelles possèdent leur propre OS qui embarque ses propres binaires et librairies. Ceci engendre une perte d'espace disque importante surtout si les binaires ou libraires sont communes à plusieurs machines virtuelles. De plus, démarrer une machine virtuelle prend du temps (jusqu'à plusieurs

minutes). En outre, les machines virtuelles doivent installer leurs propres drivers afin de communiquer avec l'hyperviseur (logiciel s'exécutant à l'intérieur d'un OS hôte qui gère les machines virtuelles). Un avantage cependant est l'isolation complète d'une machine virtuelle qui ne peut communiquer avec les autres par défaut.

Les containers s'exécutent de manière isolée par dessus l'OS hôte qui partage ses ressources (kernel, binaires, librairies, périphériques...). Plus légers, les containers démarrent en quelques secondes seulement. Sur une machine, il est tout à fait possible de lancer des milliers de containers similaires, car l'empreinte mémoire est réduite et l'espace disque occupé est partagé si les containers sont semblables. Ceci est expliqué plus en détail à la section 2.4. Les containers sont isolés, mais ils peuvent aussi communiquer entre eux si on leur a explicitement donné l'autorisation.

Si le lecteur désire connaître plus de détails concernant la virtualisation, il est conseillé de lire le chapitre 3.2 du rapport "Docker and Internet of Things (travail de semestre)" de M. Loic Bassang [5] qui amène une bonne introduction aux différents types de virtualisation.

2.3 Docker images et Docker containers

Avec Docker, une application est encapsulée avec toutes ses dépendances et sa configuration dans une **image**.

Pour construire cette image, on utilise un Dockerfile. Il s'agit d'un fichier qui décrit les étapes de création et de configuration nécessaires à l'obtention de l'application configurée. C'est dans ce fichier qu'on retrouve l'OS à utiliser, les dépendances à installer et toutes autres configurations utiles au bon fonctionnement de l'application à déployer.

Typiquement un Dockerfile permettant de lancer un serveur web Nginx qui affiche un "hello world" ressemble à ceci :

```
1  FROM alpine  # image de départ
2  MAINTAINER support@tutum.co  # mainteneur du Dockerfile
3  RUN apk --update add nginx php-fpm && \  # installation des dépendances
4      mkdir -p /var/log/nginx && \
5      touch /var/log/nginx/access.log && \
6      mkdir -p /tmp/nginx && \
7      echo "clear_env = no" >> /etc/php/php-fpm.conf
8  ADD www /www  # ajout des sources de l'application
9  ADD nginx.conf /etc/nginx/  # ajout d'un fichier de configuration
10 EXPOSE 80  # ouverture du port 80
11 CMD php-fpm -d variables_order="EGPCS" && (tail -F
   ↪ /var/log/nginx/access.log &) && exec nginx -g "daemon off;" # commande
   ↪ à lancer au lancement du container
```

Source : `https://github.com/tutumcloud/hello-world/blob/master/Dockerfile`

Un Dockerfile est en quelque sorte la recette de cuisine qui permet de construire une image Docker.

Une fois l'image construite, on peut exécuter l'application dans un container. Un container Docker est donc une instance de l'image fraîchement créée. La figure 2.2 montre les relations entre un Dockerfile, une image et un container.

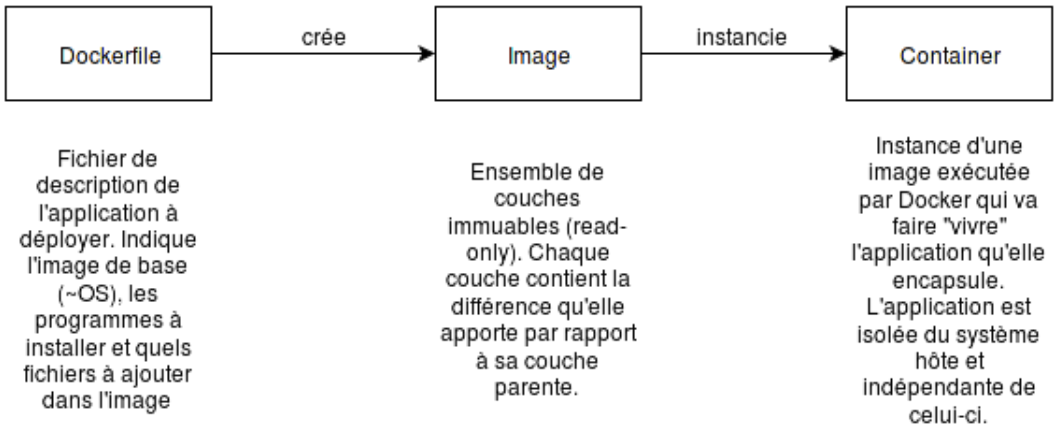

FIGURE 2.2 – Dockerfile, image et container

2.4 Système de fichiers en couche

Chaque image Docker est composée d'une liste de couches (*layers*) superposées en lecture seule[3]. Chaque couche représente la différence du système de fichiers par rapport à la couche précédente. Sur la figure 2.3, on peut voir 4 couches (identifiables par une chaîne de caractères unique, par exemple `91e54dfb1179`) et leur taille respective.

FIGURE 2.3 – Couches d'une image Docker

À la création d'un container, une nouvelle couche fine est ajoutée. Cette couche, appelée "container layer" est accessible en écriture durant l'exécution du container. La figure 2.4 le montre clairement.

Un mot supplémentaire sur une nouvelle caractéristique arrivée avec Docker 1.10 (mars 2016) ; avant cette version, Docker attribuait des UUID [1] générés aléatoirement pour identifier les couches d'une image. Désormais, ces UUID sont remplacés par des hash appelés *secure content hash*.

Les différences principales entre un UUID et un hash sont :

- Un UUID est généré aléatoirement, donc deux images exactement identiques auront deux UUID différents alors qu'en utilisant un hash, le résultat sera identique
- Avec les UUID, même si la probabilité est rare [2], il est possible de générer deux fois le même UUID ce qui peut poser des problèmes lors de la construction des images

1. UUID : `https://fr.wikipedia.org/wiki/Universal_Unique_Identifier`
2. Probabilité de doublon : `https://en.wikipedia.org/wiki/Universally_unique_identifier#Random_UUID_probability_of_duplicates`

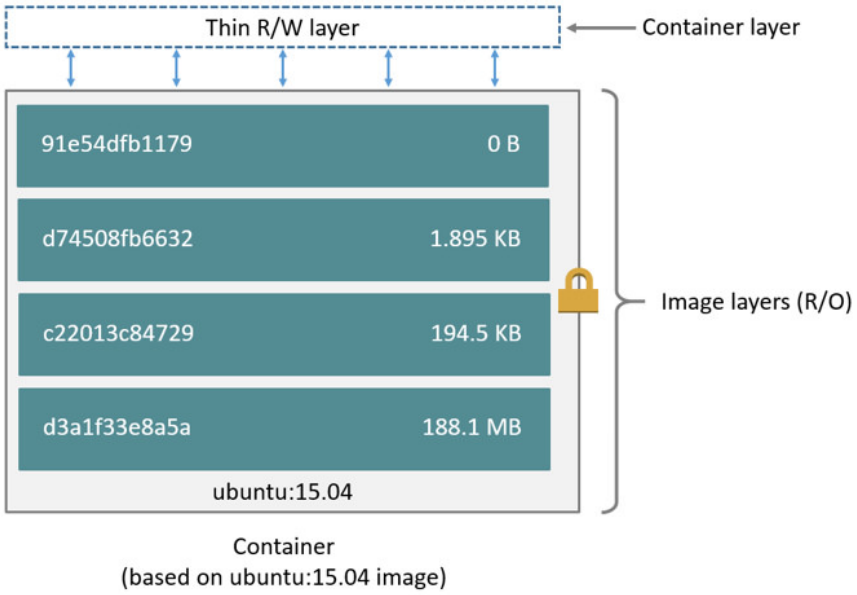

FIGURE 2.4 – Couches d'un container Docker

— Une image téléchargée chez une personne A aura un UUID différent que la même image téléchargée chez une personne B. Impossible de s'assurer de l'intégrité de l'image téléchargée en se basant sur l'UUID. En utilisant le hash, on s'assure du même résultat si l'image est identique

Par conséquent, Docker avance les avantages suivants :

— Intégrité des images téléchargées et envoyées (sur Docker Hub par exemple)
— Évite les collisions lors de l'identification des images et des couches
— Permet de partager des couches identiques qui proviendraient de *build* différents

Le dernier point est relativement intéressant. En effet, si deux images de base (Ubuntu et Debian) sont différentes, mais qu'une couche supérieure est identique (par exemple l'ajout d'un même fichier texte) alors cette couche supérieure peut être partagée entre les deux images (puisqu'elle possède le même hash). Ceci peut potentiellement offrir un gain d'espace disque conséquent si plusieurs images partagent plusieurs couches identiques. Ceci n'aurait pas pu être possible avec les UUID, car les deux couches auraient produit deux UUID différents. Un exemple est visible à la figure 2.5

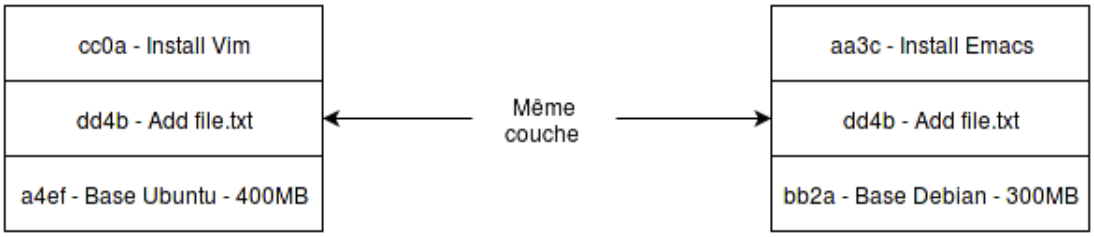

FIGURE 2.5 – Couche partagée entre deux images Docker

D'autres explications plus détaillées sur le système de fichiers en couches ont fait l'objet du chapitre 5.4.1 du rapport "Docker and Internet of Things (travail de semestre)" de M. Loic Bassang.

2.5 Isolation

Docker met en avant le fait que ses containers soient isolés du système hôte. Pour ce faire, Docker utilise des mécanismes fournis par le kernel. On peut citer parmi ces mécanismes les *namespaces* et *cgroups*.

2.5.1 Les namespaces

Les namespaces permettent d'isoler certaines fonctionnalités d'un système d'exploitation utilisant Linux. Comme chroot permet aux processus de voir comme racine un dossier isolé du système et non pas la "vraie" racine, les namespaces isolent certains aspects du système comme les processus, les interfaces réseaux, les points de montage, etc.

Jusqu'à très récemment (docker $<$ 1.10.0), Docker supportait les namespaces suivants[1] :

- PID namespace, chaque conteneur a ses propres id de processus
- UTS namespace, pour avoir son propre hostname
- IPC namespace, qui permet d'isoler les Communications Inter-Processus
- Network namespace, chaque conteneur peut avoir sa propre interface réseau, son IP, ses règles de filtrage

Docker a désormais ajouté le support d'un nouveau namespace : user namespace. Celui-ci permet à un processus d'avoir les droits root au sein d'un namespace, mais pas en dehors. Avant, Docker lançait les containers en root ce qui pouvait poser des problèmes de sécurité si un processus dans le container venait à en sortir ; il se retrouverait root sur le système hôte. Avec la prise en charge de ce namespace, un container Docker a l'impression d'être root alors qu'il n'est, en réalité, qu'un utilisateur normal sur le système hôte.

2.5.2 cgroups - Control Groups

Cgroups (control groups) est une fonctionnalité du kernel pour limiter, prioriser, isoler et contrôler l'utilisation des ressources (CPU, RAM, utilisation disque, utilisation réseau...). Pour limiter les ressources, cgroups propose de créer un groupe (profil) qui décrit les limitations à respecter. Par exemple, Si on crée un groupe appelé "groupe 1" et qu'on exige de lui qu'il n'utilise qu'au maximum 25% de la charge CPU et n'utilise qu'au maximum 100 MB de RAM. Alors, il devient possible de lancer des programmes qui appartiennent à ce groupe et qui respectent les limites fixées.

Lorsqu'on utilise la commande `docker run` de Docker pour lancer un container, Docker peut utiliser cgroups et ainsi limiter les ressources du container [3].

2.6 Contraintes liées au monde de l'embarqué

On entend par système embarqué, un système qui est/peut être léger, autonome, à puissance limitée, à stockage réduit, avec un OS minimal et souvent connecté.

Dans le monde de l'embarqué, il existe plusieurs problèmes récurrents lorsqu'on développe, déploie et maintient une application sur une cible. On peut citer les problèmes suivants :

- Cross-compilation souvent obligatoire
- Aucune interface graphique
- Installation et configuration des dépendances sur la cible
- Mises à jour de l'application et de ses dépendances
- Tests et journalisation (logs)
- Limiter l'utilisation en CPU, RAM et disque

Bien que ces problèmes peuvent être aussi présents dans le cas d'un développement desktop, ils ne sont pas aussi préoccupants.

3. Docker - Runtime constraints on resources : `https://docs.docker.com/engine/reference/run/#runtime-constraints-on-resources`

Docker peut être utile dans le cadre d'une application embarquée, car il permet une maintenance plus aisée de l'application, car on peut **versionner son installation et sa configuration ainsi que celle de ses dépendances**. Ceci permet de plus facilement mettre à jour une application, mais également de pouvoir revenir à une version précédente. De plus, si un accès réseau est disponible, il est même possible d'administrer Docker à distance depuis un poste de développement en se connectant au *deamon* Docker de la cible.

Cependant, il reste quelques freins et prérequis pour pouvoir utiliser Docker sur une carte embarquée :

GNU/Linux : Il faut un système GNU/Linux et si possible une distribution qui intègre Docker dans ses packages

Images compatibles : Les images doivent être compatibles avec la plateforme de la cible. Les images x64 ne fonctionnent pas sur ARM. Actuellement, la majorité des images sont basées sur une image de base x64. Dans la plupart des cas, il suffit de trouver l'équivalent ARM de l'image de base. Par exemple, à la lecture d'un Dockerfile, il suffit de remplacer `FROM ubuntu` par `FROM armhf/ubuntu` pour que l'image arrive à se construire. Dans les autres cas, il faudra adapter les instructions du Dockerfile.

Espace disque limité : Il faut veiller à l'utilisation de l'espace disque. En effet, la plupart des images se basent sur des Ubuntu (400 MB) ou des Debian (300 MB) ce qui peut être trop volumineux pour un système embarqué. De plus plusieurs versions de ces images peuvent être téléchargées si les Dockerfiles le spécifient. Par exemple, `FROM ubuntu:14.04` pour l'image 1 et `FROM ubuntu:15.10` pour l'image 2. On favorisera l'utilisation d'une image de base légère, comme Alpine Linux [4], commune à plusieurs applications ou processus tournant sur la cible.

4. Alpine Linux : `http://www.alpinelinux.org/`

3. Objectif 1 - Construction d'un système GNU/Linux Docker-ready

Pour rappel, on souhaite générer un système minimal contenant Docker installable sur la cible, un ODROID-XU3 Lite . On verra donc les ingrédients et pistes à suivre pour concevoir un système construit à partir de Buildroot capable de faire tourner Docker et son *daemon*.

3.1 Générer le système

Comme évoqué à la section 1.1, on dispose d'un Odroid XU3 sur lequel il faut générer un système GNU/Linux. Dans le cadre d'un cours, la Haute École d'ingénierie et d'architecture de Fribourg met à disposition un *repository* git qui contient tout ce qu'il faut pour générer un tel système.

Toutes les ressources nécessaires à la génération du système se trouvent ici :

- Procédure de génération du système du cours CSEL et adresse du *repository* git : *p.02.2_mas_csel_environnement_linux_embarque_exercices.pdf*
- Script de génération de la carte utilisé : `https://github.com/krypty/docker_and_embedded_systems/blob/master/write_system_on_sd.sh` ou Appendix A
- Le PDF *01_IntroOdroidXu3.pdf* du cours SeS

Le système généré ne peut, dans sa configuration actuelle, permettre à Docker de se lancer. Pour pouvoir le faire, on a deux moyens à disposition : Buildroot et le kernel.

Grossièrement, Buildroot permet d'ajouter des packages et de configurer son système tandis que le kernel autorise l'ajout de modules ou de drivers.

3.2 Vérifier que le système peut faire tourner Docker

Il faut en premier lieu mettre la main sur un binaire ARM Docker statiquement lié qui intègre le *daemon*. En effet, à l'heure actuelle, lorsqu'on compile Docker pour ARM de la manière officielle, le binaire résultant n'intègre pas le *daemon*, mais uniquement le client (qui permet de se connecter à un *daemon* distant). Voir également à la section 4.1.4

Le seul binaire de ce type trouvé actuellement est téléchargeable ici : `https://github.com/umiddelb/armhf/raw/master/bin/docker-1.9.1`.

Copiez ce fichier sur la cible et tentez de le lancer avec :

```
1 chmod +x docker-1.9.1
2 ./docker-1.9.1 deamon
```

S'il y a des erreurs, c'est sûrement qu'il manque un ou plusieurs modules kernel. Pour vérifier que la configuration actuelle du noyau est correcte. On trouve sur le *repository* de Docker un script qui indique quels modules sont manquants.

Ce script est à télécharger ici : `https://github.com/docker/docker/blob/master/contrib/check-config.sh`

Voici un exemple de sortie où l'on voit qu'il manque certains modules :

```
/ # ./check-config.sh
info: reading kernel config from /proc/config.gz ...

Generally Necessary:
- cgroup hierarchy: nonexistent??
    (see https://github.com/tianon/cgroupfs-mount)
- CONFIG_NAMESPACES: enabled
- CONFIG_NET_NS: enabled
- CONFIG_PID_NS: enabled
- CONFIG_IPC_NS: enabled
- CONFIG_UTS_NS: enabled
- CONFIG_DEVPTS_MULTIPLE_INSTANCES: missing
- CONFIG_CGROUPS: enabled
- CONFIG_CGROUP_CPUACCT: enabled
- CONFIG_CGROUP_DEVICE: enabled
- CONFIG_CGROUP_FREEZER: enabled
- CONFIG_CGROUP_SCHED: missing
...
- CONFIG_NETFILTER_XT_MATCH_CONNTRACK: missing
- CONFIG_NF_NAT: missing
- CONFIG_NF_NAT_NEEDED: missing
- CONFIG_POSIX_MQUEUE: enabled

Optional Features:
- CONFIG_USER_NS: missing
- CONFIG_SECCOMP: enabled
- CONFIG_CGROUP_PIDS: missing
- CONFIG_MEMCG_KMEM: enabled
...
- CONFIG_EXT3_FS: enabled
- CONFIG_EXT3_FS_XATTR: missing
- CONFIG_EXT3_FS_POSIX_ACL: enabled
- CONFIG_EXT3_FS_SECURITY: enabled
    (enable these ext3 configs if you are using ext3 as backing filesystem)
- CONFIG_EXT4_FS: enabled
- CONFIG_EXT4_FS_POSIX_ACL: enabled
- CONFIG_EXT4_FS_SECURITY: enabled
- Storage Drivers:
  - "aufs":
    - CONFIG_AUFS_FS: missing
  - "btrfs":
    - CONFIG_BTRFS_FS: enabled (as module)
  - "devicemapper":
    - CONFIG_BLK_DEV_DM: enabled
    - CONFIG_DM_THIN_PROVISIONING: missing
  - "overlay":
    - CONFIG_OVERLAY_FS: enabled (as module)
  - "zfs":
    - /dev/zfs: missing
    - zfs command: missing
    - zpool command: missing
```

La suite consiste à modifier le kernel pour y ajouter les modules manquants, de reflasher le système et tester à nouveau si Docker se lance.

La configuration d'un kernel Linux n'est volontairement pas expliquée, car d'une part cette information se trouve facilement sur Internet ou sur les documents indiqués plus haut et d'une autre, car ce n'est pas le but de ce rapport.

4. Objectif 2 - Techniques de compilation essayées

Après avoir généré un système capable d'exécuter Docker et son *deamon*, il faut (cross-)compiler Docker et le déployer sur la cible. Ce chapitre montre les diverses techniques tentées pour y parvenir.

Voici les techniques essayées :
- La manière officielle
- Compiler directement sur une machine ARM en utilisant la manière officielle
- Compiler en émulant une machine ARM sur un PC de bureau avec QEMU et une image Debian
- Compiler en émulant une machine ARM sur un PC de bureau avec QEMU et une image Raspbian
- Compiler en émulant une machine ARM sur un PC de bureau avec QEMU et chroot
- Compiler Docker sans Docker

4.1 La manière officielle

C'est la manière recommandée et qui, un jour, sera celle qu'il faudra employer. Mais aujourd'hui, elle ne permet que de cross compiler un binaire ARM Docker qui n'embarque pas le *deamon*.

4.1.1 Principe utilisé

Pour compiler Docker de la manière officiellement supportée, on doit utiliser Docker. En effet, le Makefile fourni va lancer un container Docker qui va contenir un système d'exploitation ainsi que tous les prérequis et dépendances puis lancer la compilation de Docker à l'intérieur de ce container.

4.1.2 Cheminement général

Sur une machine GNU/Linux

```
1 git clone https://github.com/docker/docker
2 cd docker
3 git checkout v1.10.3 -b tmp_build # vous pouvez remplacez v1.10.3 par la
  ↪ dernière version (tag) stable
4 make build
5 make binary # pour générer le binaire sur la plateforme sur laquelle on est
  ↪ en train de compiler (probablement x64)
6 make cross # pour générer le binaire ARM
```

Le binaire se trouve dans le dossier ./bundle. On se place volontairement sur un tag de release pour avoir un minimum de stabilité dans les fonctionnalités embarquées.

4.1.3 Schéma

Comme on peut le voir à la figure 4.1, pour compiler Docker, il faut disposer de Docker sur son PC. En faisant une commande `make`, Docker va créer un container basé sur une image Ubuntu et va installer tous les outils de compilation nécessaires. Une fois que cela est fait, Docker utilise ce container pour lancer la compilation. Le binaire est ensuite récupéré dans le dossier `./bundle`. Il ne reste plus qu'à copier le binaire sur la cible.

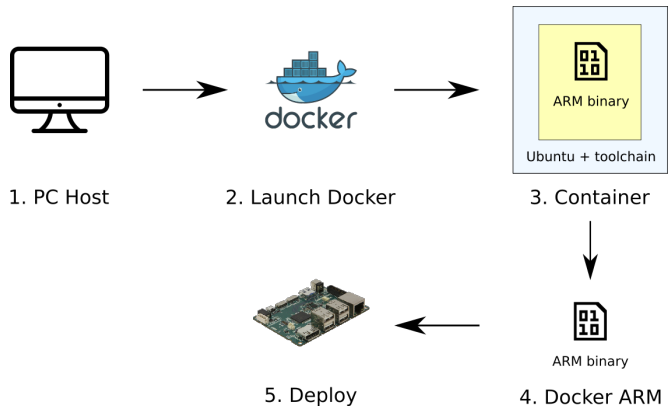

FIGURE 4.1 – Docker in Docker

4.1.4 Limitations

Actuellement, il est possible de générer un binaire Docker x64 et ARM, mais seule l'architecture x64 intègre le *deamon* nécessaire à la création de containers.

Le binaire ARM est dit `CLIENT_ONLY` dans le sens où il peut être le client d'un *deamon* Docker remote (instancié sur une autre machine).

4.2 Compiler directement sur une machine ARM en utilisant la manière officielle

Même principe que la technique précédente à la différence que le PC Host est remplacé par la cible, une carte ARM. On va donc avoir besoin d'une carte qui propose une distribution GNU/Linux qui intègre Docker. Dans le cas de l'Odroid XU3, il existe Archlinux ARM [1].

4.2.1 Principe utilisé

Voir section 4.1.

Remarque : dans ce cas, seul `make binary` est nécessaire, car on n'est pas obligé de cross compiler pour d'autres plateformes.

4.2.2 Cheminement général

Voir section 4.1.

4.2.3 Schéma

Similaire à la section 4.1.

1. Archlinux ARM : `https://archlinuxarm.org/`

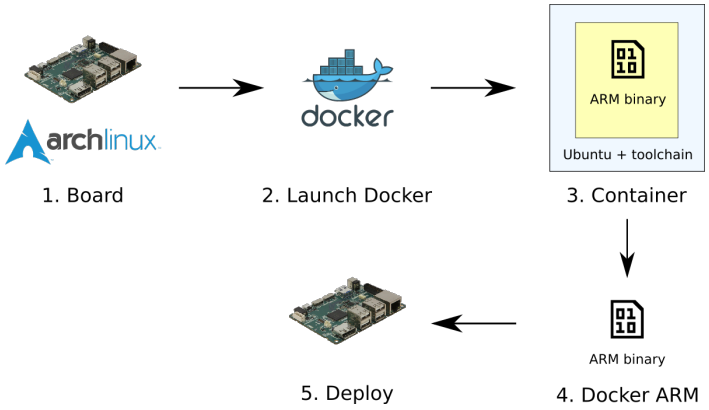

FIGURE 4.2 – Docker in Docker on ARM

4.2.4 Limitations

A l'issue de plusieurs tentatives de compiler Docker en utilisant un Odroid C1 (personnel), il n'a pas été possible de finir une compilation, car ces dernières se terminaient en gelant le système au bout d'une heure.

Évidemment, le temps de compilation est énormément long et on est limité par la machine avec laquelle on compile. Une autre possibilité pour l'avenir serait de passer par un prestataire Cloud qui fournit des machines ARM. Scaleway semble proposer ce genre d'offre [2]Scaleway : `https://www.scaleway.com/`

4.3 Compiler en émulant une machine ARM sur un PC de bureau avec QEMU et une image Debian

4.3.1 Principe utilisé

Cette technique (et les suivantes basées sur celle-ci) est différente. Ici, on utilise une machine virtuelle QEMU [3] pour compiler Docker. On ne fait donc plus de cross compilation, mais de l'émulation. On verra dans au chapitre 5 que l'on peut utiliser QEMU pour exécuter des containers ARM sur une machine x64.

4.3.2 Cheminement général

Il faut d'abord installer QEMU sur une machine. Ici Arch Linux est utilisé, mais cela devrait être sensiblement la même chose pour d'autres distributions.

```
1 pacman -S qemu qemu-arch-extra
```

Pour la suite, les tutoriels suivantes ont été suivis :

1. A QEMU image for debian armel, `http://www.n0nb.us/blog/2012/03/a-qemu-image-for-debian-armel/`.
2. Debian Wheezy armhf images for QEMU, `https://people.debian.org/~aurel32/qemu/armhf/`

On ne va pas s'étendre sur cette solution, car Debian ARM ne propose pas de package Docker et cette VM Debian ne permet pas dans sa configuration actuelle d'exécuter le binaire ARM proposé à la section 3.2.

2. .
3. QEMU, émuler une machine complète : `http://wiki.qemu.org/Main_Page`

4.3.3 Schéma

Comme on peut le voir à la figure 4.3, on reprend le même principe expliqué à la section : La manière officielle.

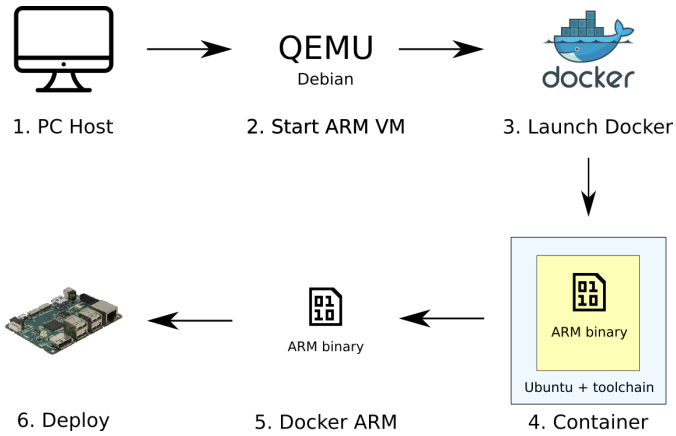

FIGURE 4.3 – Docker QEMU Debian

La principale différence vient du fait qu'on encapsule le processus dans une machine virtuelle QEMU afin d'éviter d'utiliser une carte ARM. Une fois le système émulé, on lance Docker pour compiler Docker.

4.3.4 Limitations

Cette solution n'est pas une solution à suivre. C'est, à mon avis, une bonne idée, mais inutilisable comme présenté ici. Par contre, cela peut devenir intéressant avec d'autres distributions qui intègrent un package Docker et dont le kernel est correctement configuré. Par exemple, Raspbian (Une Debian modifiée pour les Raspberry Pi) ou Archlinux ARM. Les solutions suivantes reprennent cette hypothèse.

Autre point négatif, cette solution (et les dérivées utilisant QEMU) nécessite de disposer d'un binaire ARM Docker, celui de la section 3.2. Par conséquent, comme on ne sait pas comment produire ce binaire, on ne maîtrise pas l'ensemble de la pipeline de compilation.

4.4 Compiler en émulant une machine ARM sur un PC de bureau avec QEMU et une image Raspbian

4.4.1 Principe utilisé

Même principe que la technique précédente à la différence près que l'image de la VM n'est plus une Debian, mais une Raspbian [4], utilisée par les Raspberry Pi.

4.4.2 Cheminement général

Les étapes sont relativement les mêmes que pour la technique précédente. C'est principalement les fichiers à télécharger qui diffèrent.

Voici les tutoriels que j'ai suivis :

4. Raspbian : `https://www.raspbian.org`

1. Émuler le Raspberry Pi sous Debian avec Qemu, `http://www.jdhp.org/hevea/tutoriel_rpi_qemu/tutoriel_rpi_qemu.html`
2. Émuler une Raspberry Pi sous Linux avec Qemu, `https://assos.centrale-marseille.fr/clubrobot/content/%C3%A9muler-une-raspberry-pi-sous-linux-avec-qemu`

4.4.3 Schéma

Similaire à la technique précédente.

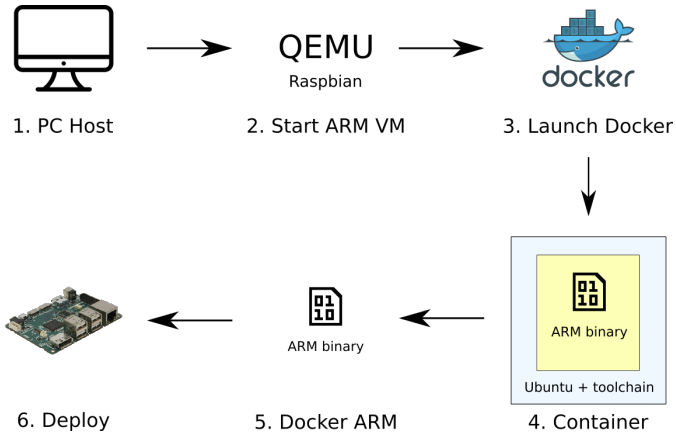

FIGURE 4.4 – Docker QEMU Raspbian

4.4.4 Limitations

Similaires à la technique précédente.

Conclusion : aucune amélioration par rapport à la technique précédente. Impossible de compiler Docker.

4.5 Compiler en émulant une machine ARM sur un PC de bureau avec QEMU et chroot

4.5.1 Principe utilisé

On va utiliser la commande `chroot`[5] afin d'isoler une machine QEMU dans un répertoire de notre machine de bureau.

4.5.2 Cheminement général

Le tutoriel utilisé est le suivant : `https://github.com/RoEdAl/linux-raspberrypi-wsp/wiki/Building-ArchLinux-ARM-packages-ona-a-PC-using-QEMU-Chroot`. Une copie de ces commandes se trouve à l'Appendix B.

4.5.3 Schéma

Comme on peut le voir à la figure 4.5, la première couche représente le dossier *root* de notre machine et le niveau suivant, les dossiers qu'il contient. Dans un dossier (à créer) appelé *docker_qemu*, on retrouve les fichiers utilisés par le tutoriel évoqué à la section précédente.

5. chroot : `https://fr.wikipedia.org/wiki/Chroot`

Dans le répertoire *archlinux_rpi2*, on retrouve le système Arch Linux ARM extrait dans lequel on vient copier `qemu-arm-static` permettant d'émuler le système ARM.

En clair, tous les dossiers en dessus de *archlinux_rpi2* appartiennent à l'OS hôte et tous les fichiers en dessous, appartiennent au système invité, Arch Linux ARM.

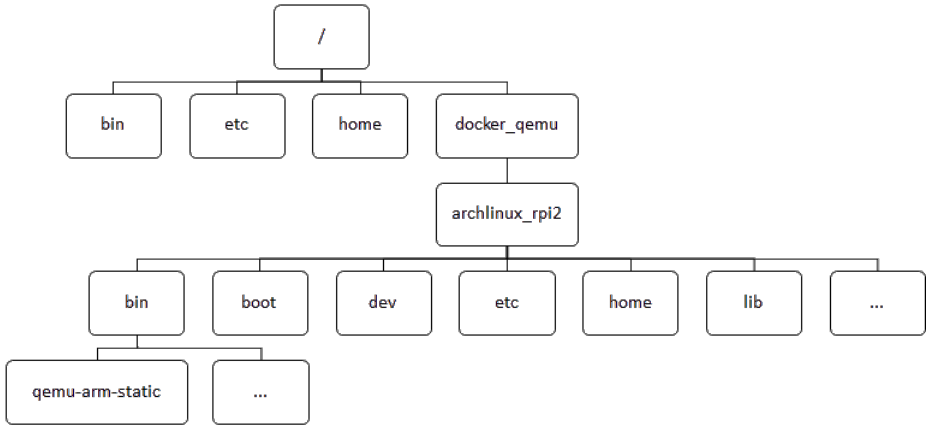

FIGURE 4.5 – Docker QEMU

4.5.4 Limitations

Bien qu'il ait été possible de booter sur la machine virtuelle, la compilation de Docker n'a pas pu être lancée, car cette procédure a "détecté" qu'on se trouvait en `chroot` et s'est arrêtée.

4.6 Compiler Docker sans Docker

Avec cette technique, on essaie de court-circuiter la compilation officielle (Docker in Docker) en compilant Docker sans Docker. Pour ce faire, il faut veiller à avoir une machine qui puisse compiler Go, car Docker est écrit dans ce langage.

Attention : Il a d'abord été tenté de compiler Docker pour une machine x64. Cette technique était avant tout exploratoire et ne permet donc pas de compiler un binaire ARM.

4.6.1 Principe utilisé

On compile le programme de manière "classique" sans passer par un container Docker.

4.6.2 Cheminement général

Installation de Go sur Arch Linux :

```
1 pacman -S go
```

Ensuite, on effectue les commandes suivantes :

```
1 git clone https://github.com/docker/docker docker_tmp
2 export AUTO_GOPATH=1
3 cd docker_tmp
4 git checkout v1.10.3 -b gary
5
```

```
6  # parmi les prérequis, il manquait btrfs-progs
7  yaourt btrfs-progs
8
9  # on ignore devicemapper, car il y avait des erreurs.
10 export DOCKER_BUILDTAGS='exclude_graphdriver_devicemapper'
11 ./hack/make.sh binary
12 # le binaire se trouve dans ./bundles/1.10.3/binary
```

4.6.3 Schéma

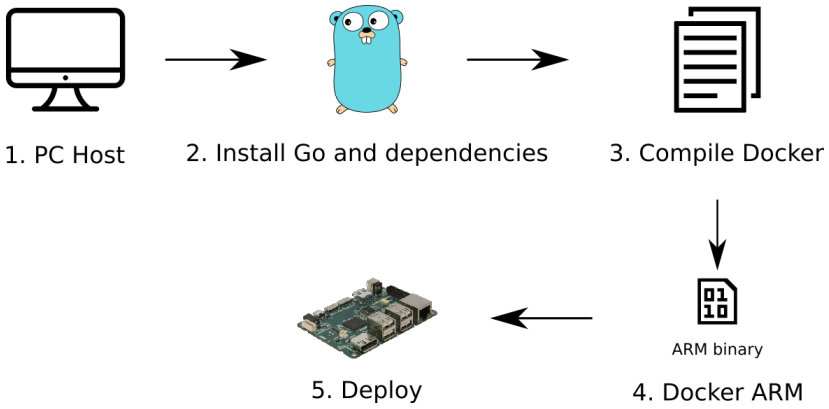

FIGURE 4.6 – Docker sans Docker

La figure 4.6 montre la différence par rapport à la manière officielle. En effet, Docker n'est plus utilisé, mais uniquement ses sources.

4.6.4 Limitations

Cette technique n'est pas une bonne idée pour les raisons suivantes :

- Il faut installer et configurer soi-même les outils de compilations alors qu'avec la manière officielle, c'est automatiquement fait dans un container.
- Lors des tests, il n'a été tenté de cross compiler vers ARM. C'est possible, mais cela n'a été pas testé.
- Cette solution n'apporte rien de plus par rapport à la manière officielle. C'est-à-dire qu'elle produit (ou produirait dans le cas d'une cross compilation ARM) le même binaire qu'avec la manière officielle. Donc, le *deamon* ne serait de toute façon pas disponible.

Néanmoins, cette solution a permis de comprendre qu'il fallait peut-être modifier le code source de Docker afin d'autoriser l'intégration du *deamon* tout en conservant la technique "Docker in Docker" afin de se faciliter la vie pour installer les outils de compilation.

4.7 Conclusion sur les techniques de compilation

Comme on a pu le voir, plusieurs techniques et tentatives ont été essayées. Aucune n'a permis de compiler ou de cross compiler Docker pour obtenir un binaire ARM statiquement lié.

Néanmoins, on sait que c'est possible, car d'une part des distributions comme Arch Linux ARM intègrent un package Docker incluant le *deamon* et d'une autre, car on a pu mettre la main sur un binaire isolé, voir section 3.2.

Encore une fois, il n'est pas impossible que Docker supporte officiellement le *deamon* pour la plateforme ARM d'ici quelque temps et ce document sera obsolète.

Pour la suite du projet, j'ai installé Archlinux ARM sur l'Odroid XU3 et vais continuer avec une partie concernant la sécurité de Docker et de ces containers.

5. Exécuter des containers ARM sur une machine x64

Dans ce chapitre, on va voir comment exécuter des containers basés sur des images ARM sur une machine x64. Ceci permet plusieurs choses intéressantes comme :

- Tester une image sur son poste avant de la déployer sur une cible
- Pouvoir utiliser des images prévues pour ARM sur une machine x64, voire d'en effectuer le portage
- Utiliser le container ARM comme outil de cross compilation
- Pouvoir développer une application ARM et la tester sur le même PC sans devoir la déployer sur la cible à chaque compilation
- ...

Lien utile : `http://blog.hypriot.com/post/close-encounters-of-the-third-kind/`

On va reprendre l'exemple du blog de Hypriot et lancer un container httpd ARM (un serveur web et une page HTML toute simple).

Il faut d'abord installer QEMU. Pour ce faire, voir section 4.3.2.

Ensuite, il faut créer un Dockerfile qui se base sur l'image ARM (ici `hypriot/rpi-busybox-httpd`) à lancer et une archive contenant `qemu-static-arm`.

Création d'un dossier de travail :

```
mkdir hypriot-qemu
cd hypriot-qemu
touch Dockerfile
```

Dockerfile à créer sur une machine x64 :

```
FROM hypriot/rpi-busybox-httpd
ADD qemu-arm-static.tar /
```

Pour l'archive, il faut effectuer les manipulations suivantes :

```
sudo cp /usr/bin/qemu-arm-static .
sudo chown gary:gary qemu-arm-static
mkdir usr
tar -cvf qemu-arm-static.tar usr
mkdir usr/bin
tar -uvf qemu-arm-static.tar usr/bin
mv qemu-arm-static usr/bin
tar -uvf qemu-arm-static.tar usr/bin/qemu-arm-static
```

Si tout s'est bien passé, vous devriez obtenir une sortie similaire :

```
tar vtf qemu-arm-static.tar
drwxrwxr-x gary/gary         0 2016-04-15 21:35 usr/
drwxrwxr-x gary/gary         0 2016-04-15 21:35 usr/bin/
-rwxr-xr-x gary/gary   2936324 2016-04-15 21:26 usr/bin/qemu-arm-static
```

Il ne reste plus qu'à lancer le container :

```
docker build -t rpi-busybox-httpd .
```

```
2 docker run -d -p 80:80 rpi-busybox-httpd # essayez un autre port s'il est
  ↪ déjà pris
```

Rendez-vous ensuite sur `http://localhost:80` pour observer la page web.

6. Conclusion

À la fin de la lecture de ce rapport, on se rend compte qu'il n'y a aucune méthode testée qui a permis de (cross-)compiler Docker sur la cible. Néanmoins, ce document montre les techniques utilisées et aidera peut-être en cas de reprise de cette partie du projet.

Heureusement, tout ceci a permis de mieux comprendre comment Docker était construit et comment il fonctionnait de manière générale. De plus, la génération d'un système depuis zéro a permis d'apprendre beaucoup sur Linux et les systèmes embarqués en général.

La suite du projet s'articulera autour de la sécurité de Docker et de sa configuration.

Bibliographie

[1] Docker 1.10 et les user namespace, mai 2016. http://linuxfr.org/users/w3blogfr/journaux/docker-1-10-et-les-user-namespace.

[2] Docker (software), mai 2016. https://en.wikipedia.org/wiki/Docker_%28software%29.

[3] Docker Inc. Understand images, containers, and storage drivers, mai 2016. https://docs.docker.com/engine/userguide/storagedriver/imagesandcontainers/.

[4] Docker Inc. What is Docker ?, mai 2016. https://www.docker.com/what-docker.

[5] Loïc Bassang. Docker and Internet of Things. Technical report, Haute École d'ingénierie et d'architecture de Fribourg , 2015.

[6] Katherine Noyes. Docker : A 'Shipping Container' for Linux Code, mai 2016. https://www.linux.com/news/docker-shipping-container-linux-code.

Appendices

A. Script : Write system on SD

```
#!/bin/bash

#Warning: please replace sdc by the location of your sd card
# Please review this script before use it. It can harm your system :-)

sudo dd if=/dev/zero of=/dev/sdc bs=4k count=32768
sudo parted /dev/sdc mklabel msdos
sudo parted /dev/sdc mkpart primary ext4 131072s 2228223s
sudo parted /dev/sdc mkpart primary ext4 2228224s 4325375s
sudo mkfs.ext4 /dev/sdc2 -L usrfs
#sudo mkfs.ext4 /dev/sdc1 -L rootfs
sync
sudo dd if=/tftpboot/xu3-bl1.bin of=/dev/sdc bs=512 seek=1
sudo dd if=/tftpboot/xu3-bl2.bin of=/dev/sdc bs=512 seek=31
sudo dd if=/tftpboot/xu3-tzsw.bin of=/dev/sdc bs=512 seek=2111
sudo dd if=/tftpboot/xu3-u-boot.bin of=/dev/sdc bs=512 seek=63
sudo dd if=/tftpboot/xu3-uImage of=/dev/sdc bs=512 seek=6304
sudo dd if=/tftpboot/exynos5422-odroidxu3.dtb of=/dev/sdc bs=512 seek=22688
sudo dd if=/tftpboot/xu3-rootfs.ext4 of=/dev/sdc1
sync
sudo resize2fs /dev/sdc1
```

B. Building ArchLinux ARM packages on a PC using QEMU Chroot

Source : `https://github.com/RoEdAl/linux-raspberrypi-wsp/wiki/Building-ArchLinux-ARM-packages-ona-a-PC-using-QEMU-Chroot`

On a PC with ArchLinux or Manjaro installed :

Build and install binfmt-support package. Build and install qemu-user-static package. Install arch-install-scripts package.

Download and extract the root filesystem for Raspberry Pi or Raspberry Pi 2 :

```
wget http://archlinuxarm.org/os/ArchLinuxARM-rpi-latest.tar.gz
bsdtar -xpf ArchLinuxARM-rpi-latest.tar.gz -C archlinux-rpi
```

or

```
wget http://archlinuxarm.org/os/ArchLinuxARM-rpi-2-latest.tar.gz
bsdtar -xpf ArchLinuxARM-rpi-2-latest.tar.gz -C archlinux-rpi2
```

Enable ARM to x86 translation :

```
update-binfmts --enable qemu-arm
```

Copy the **QEMU** executable :

```
cp /usr/bin/qemu-arm-static archlinux-rpi/usr/bin
```

Chroot :

```
arch-chroot archlinux-rpi /bin/bash
```

Install base-devel and distcc packages :

```
pacman -Suy base-devel distcc
```

Exit and chroot again as normal alarm user :

```
exit
arch-chroot archlinux-rpi runuser -l alarm
```

Build your package.