Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE

Corso di Laurea Magistrale in Informatica

Architettura cloud per lo sviluppo multi-piattaforma di Sistemi Operativi: Applicazioni specifiche

Relatore:
Chiar.mo Prof.
Alessandro Amoroso

Presentata da:
Francesco Ferretti

I Sessione
2015-2016

A tutti coloro che mi hanno aiutato a renderlo possibile ...

Introduzione

Il mondo dell'Internet of Things e del single board computing sono settori in forte espansione al giorno d'oggi e le architetture ARM sono, al momento, i dominatori in questo ambito.
I sistemi operativi e i software si stanno evolvendo per far fronte a questo cambiamento e ai nuovi casi d'uso che queste tecnologie introducono.
In questa tesi ci occuperemo del porting della distribuzione Linux Sabayon per queste architetture, la creazione di un infrastruttura per il rilascio delle immagini e la compilazione dei pacchetti software.

Indice

Elenco delle figure

Capitolo 1

Introduzione ai dispositivi ARM e Embedded

In questo capitolo andiamo a introdurre e a delineare gli aspetti del mondo dei chip a basso consumo energetico e alle nuove tipologie di calcolatori disponili sul mercato.

1.1 Contesto

Al giorno d'oggi il consumo energetico è un aspetto fondamentale dei sistemi, che siano essi elettronici o semplicemente meccanici.
Infatti, molto più che nelle decadi passate, si sente il peso dell'ambiente e l'aumento della popolazione ha come conseguenza l'incremento dell'energia necessaria al sostentamento di tutte le attività umane[1]. Parallelamente aumenta anche la richiesta di tecnologia per tutti gli aspetti della nostra vita quotidiana e lavorativa.
Questa richiesta, che non può essere soddisfatta da tecnologie come quelle dei processori con architettura x86 e x86_64, ha spianato la strada a nuove architetture e soluzioni per gli ambiti più disparati.
Le nuove architetture hanno avuto un doppio effetto, aumentando da un lato

[1]Da http://www.eia.gov/todayinenergy/detail.cfm?id=12251

potenzialità di dispositivi già impiegati da tempo e creando dall'altro nuovi settori inimmaginabili fino a oggi come smartphone e Internet of Things[1]. Questi nuovi ambiti sono stati il fulcro su cui si è basata l'evoluzione dell'interazione tra uomo e macchina e portando con loro nuove sfide e problematiche per software ed hardware.
Nel panorama frammentato possiamo però delineare ARM come architettura hardware che fa da padrona la quale, grazie alle sue caratteristiche, è al momento quella che ha il dominio del mercato[2].
Infatti il numero di persone che nel 2014 usavano un processore ARM erano superiori persino a quelle in grado di avere accesso a sistemi igenici basilari[3].

1.1.1 Il mondo ARM

ARM (acronimo che sta per Advanced RISC Machine) è una famiglia di processori a 32/64 bit progettata dalla ARM Holdings ed è l'architettura principe dei Personal Devices come smartphone, tablet, smartwatch, etc.
Il successo di questa architettura è da ricercare appunto nel suo approccio al design che si basa sul RISC ossia reduced instruction set computing[2].
Questo approccio comporta una riduzione, rispetto a CISC (complex instruction set computing) dei normali x86, del numero di transistor che devono essere integrati all'interno del processore. Riducendo così le prestazioni del processore, ma anche calore, energia e costi di produzione.
Molta dell'evoluzione delle nuove tecnologie, avvenuta durante questi anni, è dovuta all'impiego massiccio di ARM negli smartphone sempre più uno strumento indispensabile agli utenti.

Storia dell'azienda

In principio la A di ARM aveva come significato di Acorn Computers Ltd ossia la ditta in cui nacque il progetto di un coprocessore per il BBC Micro,

[2]Da http://ir.arm.com/phoenix.zhtml?c=197211&p=irol-mainmarkets
[3]Da http://www.bloomberg.com/bw/articles/2014-02-04/arm-chips-are-the-most-used-consumer-product-dot-where-s-the-money

serie di punta della loro linea di computer[4].
Il progetto fu creato nel 1983 da una squadra d'ingegneri sotto la supervisione di Sophie Wilson e Steve Furber per realizzare una versione migliorata del MOS Technology 6502. Infatti i dirigenti pensavano di poter utilizzare dei processori prodotti internamente per ottenere un vantaggio sui concorrenti rispetto alla produzione esterna dei componenti.
ARM1 venne alla luce nel 1985 ma solamente come prototipo, il primo processore effettivamente prodotto fu l'ARM2 (Figura 1.1) realizzato solo nel 1986. La CPU, con 32-bit e solo 30000 transistor[3], era in competizione con i top di gamma Motorola dell'epoca. Questo era dovuto al fatto che come scelta di progetto l'azienda decidette di rimuovere il microcodice all'interno della CPU e solo nel successore d'introdurre una cache.

Lo slancio creato da questi prodotti fece si che aziende come Apple si interessarono alla Acorn, con la quale iniziarono a lavorare per sviluppare una nuova versione dei processori ARM[5].
Da questa partnership nacquero i processori ARM6 nel 1992 che Apple usò come base per il suo palmare, il Newton (Figura 1.2)[6]. Questo primo tentativo di creare un device facilmente trasportabile e touchscreen però naufragò per il suo prezzo elevato e problemi con il riconoscimento della scrittura[7].

Nel frattempo Acorn, che per la partnership con Apple fu costretta a creare una società apposita per lo sviluppo di processori, utilizzò l'ARM6 per i propri computer RISC.
L'architettura riscosse molto successo anche perché impiegava solo 35000 transistor all'interno di un processore 32-bit a 12MHz e 10 MIPS che consumava molto meno di un watt, tutto ciò nel 1992[8].
Questo grazie anche alla strategia della società, che è ancora oggi quella di

[4]Da http://lowendmac.com/2007/acorn-and-the-bbc-micro-from-education-to-obscurity/

[5]Da http://www.arm.com/about/company-profile/milestones.php

[6]Da http://www.wired.com/2013/08/remembering-the-apple-newtons-prophetic-failure-and-lasting-ideals/

[7]Da http://www.macobserver.com/tmo/article/john_sculley_the_full_transcript_part2/

[8]Datasheet http://www.home.marutan.net/arcemdocs/ARM610.pdf

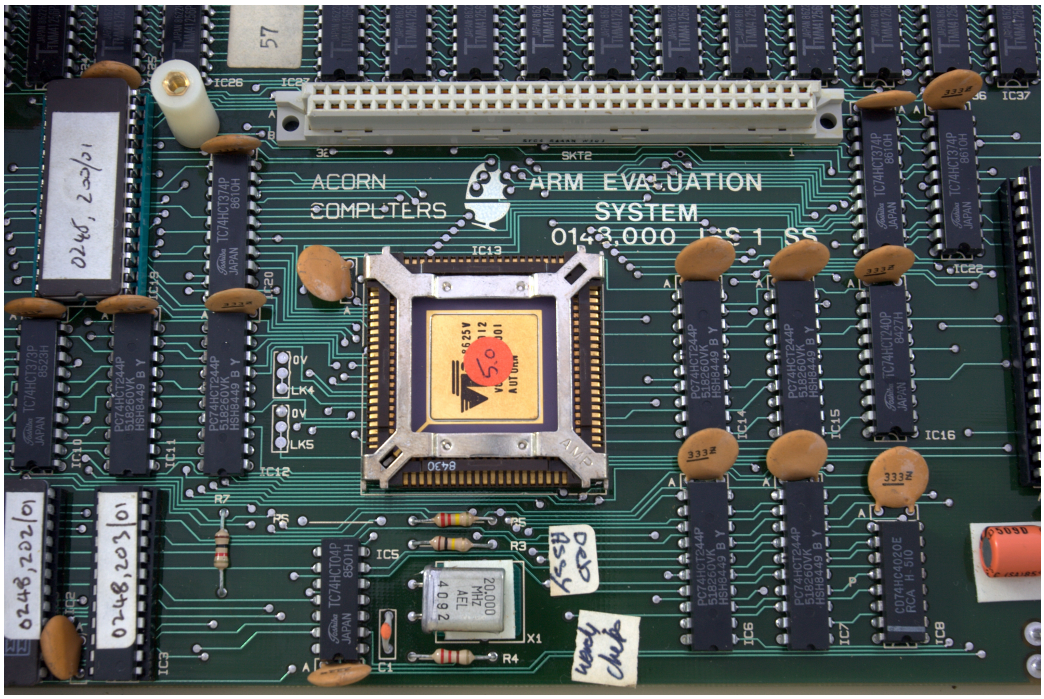

Figura 1.1: Il primo processore ARM inserito all'interno del BBC Micro.

Fonte immagine https://en.wikipedia.org/wiki/ARM_architecture.

creare solo i progetti del core ARM e di permettere quindi ai produttori di personalizzare il processore per i vari compiti specifici[9].

ARM6-7 e gli smartphone La possibilità di personalizzare il chip prima di metterlo in vendita ha portato ad una espansione notevole del campo delle applicazioni di questa architettura, creando addirittura nuovi mercati e dispositivi.
Il Newton fu un esperimento fallimentare però contribuì a creare, insieme al IBM Simon (Figura 1.3)[10], un interesse verso un genere di dispositivi personali chiamati personal digital assistant (PDA).
Questo convinse aziende come U.S. Robotics prima con Palm e Compaq poi

[9]Da http://www.arm.com/products/buying-guide/licensing/
[10]Da http://www.bloomberg.com/news/articles/2012-06-29/before-iphone-and-android-came-simon-the-first-smartphone

Figura 1.2: Il primo PDA della Apple equipaggiato con processore ARM.

Fonte immagine https://it.wikipedia.org/wiki/Apple_Newton.

con iPAQ, a sviluppare dispositivi PDA. Inizialmente Palm basava la sua offerta su processori come il Motorola 68328 per Pilot 1000[11] rappresentato in Figura 1.4, ma nel 2002 i dirigenti decisero di abbandonare questi core per gli ARM che avevano già ricevuto notevoli investimenti sin dai tempi del defunto Newton.

Palm fece da apripista nel mondo dei PDA ma i suoi dispositivi non ebbero una espansione di massa. Questo per via del fatto che erano dispositivi pensati per una fascia di utenza molto ristretta, e portavano con loro molte problematiche come nessuna memoria permanente, risorse computazionali limitate e scarsa possibilità di personalizzazione.

Nel 2001 invece, la Apple, decise di creare una nuova lineup di dispositivi per la riproduzione di audio ad alta qualità con batteria integrata per rilanciare il

[11]Da http://www.palminfocenter.com/news/8493/pilot-1000-retrospective

Figura 1.3: Il primo smartphone della storia.

Fonte immagine https://en.wikipedia.org/wiki/IBM_Simon.

marchio. Forte dei suoi rapporti precedenti con ARM decise così di utilizzare un ARM7TDMI[4] per il primo iPod Classic[12] (Figura 1.5).

Un dispositivo unico nel suo genere poiché coniugava il design di Apple ad un mix di tecnologie di cui l'ARM7 era il motore, garantendo una durata delle batterie di ben 12 ore.

Il notevole successo di questi dispositivi e la crescente richiesta di funzioni da parte degli utenti per i propri cellulari spinse Apple a siglare un accordo segreto con AT&T Mobility per lo sviluppo di un cellulare. Steve Jobs in-

[12]Da https://support.apple.com/it-it/HT204217

Figura 1.4: Il primo PDA prodotto da Palm.

Fonte immagine https://en.wikipedia.org/wiki/Pilot_1000.

fatti voleva trasportare i tablet e i PDA nel mercato di massa avendo visto il successo ottenuto con il Rokr e la partnership con Motorola
Venne così concepito un nuovo concetto di cellulare di massa: lo smartphone. Il 9 Gennaio 2007 Steve Jobs annunciò l'iPhone (Figura 1.6) sotto l'attenzione dei media e creando hype tra coloro che apprezzavano iPod, come CPU fu scelta appunto una Samsung ARM 1176JZ(F)-S v1.0[13].

La nuova gamma di dispositivi apriva un intero nuovo settore di mercato e il tutto inesplorato in cui le aziende vedevano potenziali di crescita enormi.

[13]Da https://www.engadget.com/2007/07/01/iphone-processor-found-620mhz-arm/

Figura 1.5: Il primo iPod di Apple, prodotto simbolo del suo rilancio.

Fonte immagine https://en.wikipedia.org/wiki/IPod_Classic.

1.1.2 ARM e single board computing

Oggi ARM è una azienda leader nella progettazione di system on a chip (SoC), ovvero sistemi che integrano diverse funzioni in un unico componente, grazie alla grande spinta ricevuta dal mondo degli smartphone.
Questo ha portato i costruttori a creare dispositivi per le applicazioni più disparate, sviluppando addirittura nuove tipologie di hardware anche in abiti dove non veniva nemmeno reputata necessaria l'elettronica.
La creazione di processori così performanti e con bassi consumi ha permesso di portare sistemi operativi general purpose come Linux in applicazioni come la domotica, da sempre campo dominato da dispositivi embedded che ne erano sprovvisti.
Infatti, uno tra i mercati più floridi di cui assistiamo la nascita al giorno d'oggi è quello di dispositivi delle dimensioni di una carta di credito, chiamati single board computer (Figura 1.7).

Figura 1.6: Il primo iPhone lanciato sul mercato nel 2007.

Fonte immagine https://en.wikipedia.org/wiki/IPhone_(1st_generation).

Questi computer sono abbastanza potenti da sopperire ai bisogni dell'utente medio e al contempo, grazie ai consumi bassi, possono essere impiegati per scopi come pilotare GPIO o fare rilevamenti tramite sensori[14].
Tutto ciò a beneficio anche degli sviluppatori, che oggigiorno non devono più descrivere nel dettaglio il software per una scheda embedded, ma possono invece interfacciarsi con un sistema operativo con una maggiore astrazione e dalle primitive complesse.

[14]https://www.raspberrypi.org/blog/raspberry-pi-zero/

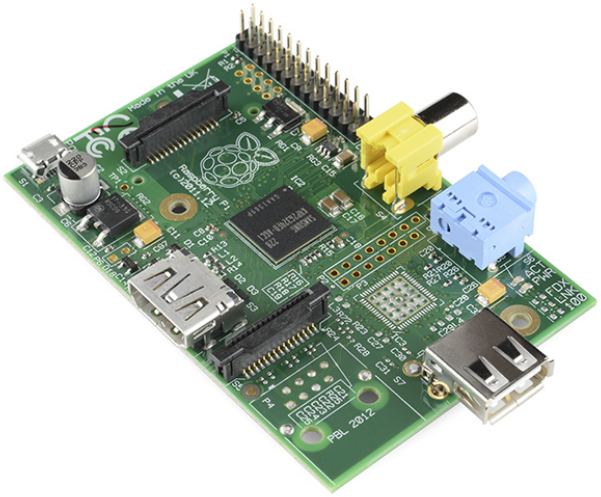

Figura 1.7: La prima e più famosa scheda single board computer, Raspberry PI modello A.

Fonte immagine https://en.wikipedia.org/wiki/Raspberry_Pi.

1.1.3 Sistemi operativi e infrastrutture cloud

La rivoluzione del concetto di PC è dovuta anche a due fattori non banali che sono:

- la crescita delle infrastrutture cloud;
- la creazione di sistemi operativi performanti per le nuove piattaforme.

Per quanto riguarda il cloud bisogna ricordare che lo sviluppo degli smartphone è stato spinto anche dalla possibilità di spostare su servizi esterni operazioni complesse ed energeticamente dispendiose.
Basti pensare al salvataggio delle foto e dei file in archivi cloud che stanno via via sostituendo il ricorso al supporto fisico.
Questa esternalizzazione di servizi, dalla macchina a server remoti, permette di creare nuove tipologie di applicativi, che spostano la potenza computa-

zionale e permettono ottimizzazioni oltre che l'espansione delle funzionalità disponibili.
Pensando invece ai sistemi operativi, la richiesta di ottimizzazione per architetture a basso consumo energetico ha portato ad una spinta all'unificazione delle piattaforme.
Per unificazione si intende che su diversi dispositivi venga eseguita la medesima versione del sistema operativo modificando solamente l'interfaccia utente.
Come prova di questa tendenza stanno uscendo tecnologie come Microsoft Continuum[15], che permette di utilizzare il proprio smartphone come sistema desktop semplicemente connettendolo ad un monitor.
Questo processo porta a dover creare applicativi che siano nativamente multi piattaforma oppure la richiesta di un livello di astrazione del sistema operativo stesso sempre nell'ottica di ridurre il carico di lavoro degli sviluppatori.

1.2 Scopo del lavoro e problemi correlati

Lo scopo del lavoro è quello di creare una infrastruttura per il rilascio delle immagini del sistema operativo Sabayon Linux e la compilazione di pacchetti per la piattaforma ARM.
Questo si compone di due parti principali:

- creazione di una repository di pacchetti;
- rilascio delle immagini per schede specifiche.

La creazione della base di pacchetti da inserire nella repository della distribuzione porta con se diverse problematiche.
Da un lato il sistema di gestione delle repository remoto è stato progettato per gestire pacchetti multi-piattaforma, dall'altro la compilazione del software è resa difficoltosa dalle limitate capacità computazionali di ARM.

[15]Da https://www.microsoft.com/it-it/windows/Continuum

Il parco software fornito deve essere il più completo possibile, ma solo una minima parte dei pacchetti presenti nelle repository può essere compilato per architettura ARM.
Inoltre la compilazione manuale dei pacchetti, sebbene sia il modo migliore di procedere per il popolamento iniziale dei pacchetti, non è la soluzione più efficiente per il mantenimento della repository. Questo perché l'aggiornamento dei software richiederebbe un costante intervento umano che oltre che costoso, potrebbe portare ad errori nel lungo periodo.
La manutenzione delle repository deve essere pensata e progettata per consentire anche un accesso multiplo alla macchina che si occupa della compilazione, evitando collisioni e gestendo la continuità del servizio.
Una possibile soluzione è quella di utilizzare delle virtual machine, ma questo richiede una infrastruttura predisposta a virtualizzare macchine ARM e un sistema di backup per le immagini.
La creazione delle immagini invece è un problema che deve essere affrontato per ogni scheda, risolvendo i problemi specifici modello per modello.
I kernel per ogni dispositivo sono un problema non da poco, creato dal fatto che ogni produttore decide le specifiche tecniche del proprio SoC richiedendo configurazioni ad-hoc.
Un'opzione è quella di compilare i kernel per ogni scheda partendo dalle specifiche del produttore, ma questo approccio è dispendioso sia per via del tempo impiegato che per le implicazioni riguardanti gli aggiornamenti di versione.
L'uso delle immagini fornite dai produttori è una opzione più semplice, ma impedisce una configurazione fine dei parametri che potrebbe essere richiesta per il funzionamento di alcune schede.
Infine, un ulteriore obiettivo che bisogna prefissare, è quello di creare un sistema riproducibile, questo poiché in caso di errore deve essere sempre possibile recuperare lo stato sia delle immagini che delle repository.

1.2.1 Struttura della tesi

Si procederà ora ad analizzare i sistemi Hardware disponibili sul mercato e le loro caratteristiche peculiari.
Andremo successivamente ad analizzare il lato software, identificando le varie alternative all'interno del mondo Open Source come sistemi operativi disponibili e la loro strutturazione.
Passeremo quindi ad analizzare le tecnologie utilizzate per la realizzazione del lavoro.
Si andranno allora a presentare le soluzioni proposte ed effettivamente utilizzate per la creazione delle repository e la loro gestione.
Infine esporremo la soluzione proposta per la generazione delle immagini, descrivendo il procedimento che si è reso necessario per supportare i diversi hardware.

Capitolo 2

Analisi dei sistemi esistenti

In questo capitolo verrà effettuata un'analisi delle tecnologie Hardware e software per single board computing, dispositivi embedded e mobili ad oggi disponibili.

2.1 Hardware in commercio

In questa sezione saranno approfondite nel dettaglio le caratteristiche della nomenclatura e architettura dell'hardware in oggetto.
Saranno quindi analizzate le piattaforme hardware disponibili in commercio e le loro caratteristiche tecniche.

2.1.1 Nomenclatura della famiglia ARM

La famiglia ARM ha una strutturazione di nomenclatura particolare che la rende di difficile comprensione per i non esperti.
I processori ARM non corrispondono direttamente ai numeri identificativi dell'architettura e le lettere utilizzate come suffissi possono confondere ulteriormente. Questo porta spesso nella situazione di dover cercare il tipo di processore specifico per scoprire dettagli tecnici e implementativi.
Come detto in precedenza tutto è iniziato con ARM2 che quasi istantanea-

mente ha rimpiazzato ARM1 nei dispositivi commerciali disponibili.
L'Acorn Archimedes fu il primo dispositivo RISC con un processore di tipo ARM2 ma venne prontamente sostituito anch'esso dalla nuova generazione di processori ARM3.
Apple, DEC e Intel, dopo i prodotti creati direttamente da ARM, finanziarono e produssero i processori ARM6 e ARM7 al fine di creare una nuova generazione di RISC in grado di soddisfare le richieste di mercato.
I processori ARM8 si alternarono nell'uscita con lo StrongARM di DEC/Intel e questo creò un'ulteriore frammentazione del mercato.
La fortuna di ARM arrivò con il rilascio dei ARM7TDMI il quale portava con se istruzioni a 16bit e il supporto per lo standard JTAG per il debug della scheda, che venne ampiamente utilizzato per i sistemi embedded.
La strategia particolare di ARM di essere una ditta di progettazione e non di produzione dell'hardware ha quindi creato una continua frammentazione della nomenclatura.
Dunque, per fare un po' di chiarezza, classifichiamo ARM1 come architettura ARMv1, mentre ARM2 e ARM3 fanno parte della famiglia ARMv2.
Si noti però che la differenza tra ARM2 e ARM3 è sostanziale, maggiore di quella tra ARM2 e ARM1. Non vennero mai prodotti ARM4 e ARM5, ma si passò subito a ARM6 e ARM7 che furono classificati ARMv3.
Gli StrongARM e i processori ARM8 sono classificati come ARMv4, mentre il processore ARM9TDMI e il famigerato ARM7TDMI appartengono a ARMv4T[4].
I processori ARM7EJ e ARM9 vengono classificati invece come ARMv5TE.
Questa nomenclatura poco coerente e anche difficoltosa da seguire resistette sino ad ARM11 il quale appartiene alla famiglia ARMv6.
La sua uscita infatti segnò il passaggio alla catalogazione sotto la tipologia Cortex.
Ci sono tre famiglie principali di Cortex:

- Cortex-A progettato con l'obiettivo di essere utilizzato per applicazioni general purpose;

- Cortex-R finalizzato ad applicazioni in tempo reale con una bassa latenza ma anche fortemente limitato dal punto di vista applicativo;
- Cortex-M per applicazioni a microcontrollore progettato con un set di istruzioni limitato e una pipeline di piccole dimensioni, pensato per sostituirsi ad altre soluzioni in ambito industriale.

Con questa nuova catalogazione la numerazione è diventata completamente non correlata, rendendo impossibile in alcun modo mettere in relazione processori di famiglie Cortex differenti.
In questo lavoro ci occuperemo prevalentemente dei processori Cortex-A dato che sono quelli pensati e progettati per far girare applicativi.

2.1.2 Specifiche di ARM di vecchia generazione

Gli ARM di vecchia generazione, a eccezione di StrongARM, sono ad architettura di von Neumann. Quest'ultima è un'architettura hardware per processori che comporta la memorizzazione dei dati del programma in esecuzione e delle istruzioni nello stesso spazio di memoria.
Un'ulteriore peculiarità inoltre è l'architettura a 26bit su cui si basavano i processori ARM. Questa tipologia di architettura metteva insieme il Program Counter (PC) e il Process Status Register (PSR) in un unico registro a 32bit. Così facendo si ottenevano numerosi vantaggi come, ad esempio, il ripristino di PC e flag di stato con una sola operazione, dando così la possibilità di eseguire le istruzioni più velocemente.
Lo svantaggio principale di questa tecnologia derivava dal fatto che ARM1 e ARM2 possedevano un solo Program Counter e bus d'indirizzamento a 26bit, riducendo quindi lo spazio della memoria totale indirizzabile a 64MB. Potremmo definire la limitazione blanda, dato che 64MB erano un enormità per l'epoca. La verità però è che sono stati necessari molti cambiamenti per modificare la sotto-struttura di ARM per renderla più efficiente.
Il sistema operativo progettato per girare su questa macchine a 26bit era RISC OS, originariamente pubblicato nel 1987 con il nome in codice di Arthur.

Tutte le versioni di RISC OS compresa la Castle sviluppata per PC Iyonix, hanno lavorato in modalità a 26bit che fu poi via via abbandonata.

ARMv1 e ARMv2 e ARMv2A

Possiamo accorpare ARMv1 e ARMv2 dato che il processore ARM1, unico della sua famiglia, fu più che altro un esperimento e condivise molti dettagli architetturali con ARM2.
Inoltre ARM1 non venne mai commercializzato e fu utilizzato solo in piccola quantità come coprocessore per il BBC Micro. Attualmente infatti attrae prevalentemente collezionisti per la sua taratura limitata.
ARM2, basato su ARM1, fu nettamente più utilizzato data la sua commercializzazione nell'Archimedes e il supporto a IOC, MEMC, VIDC.
Possiamo definire ARM2 è una versione rivisitata a 8MHz di ARM1 con all'interno un nuovo meccanismo per la moltiplicazione hardware e un'interfaccia a coprocessore per l'acceleratore di operazioni in virgola mobile.
Fece però subito vedere le proprie limitazioni data la mancanza del multitasking.
La terza iterazione di ARM (ARM3) soppiantò questa tecnologia fornendo una cache unificata a 4KB, una maggiore frequenza di clock e le istruzioni SWP.
Le istruzioni SWP sono funzioni atomiche per lo scambio dei dati tra i registri e la memoria senza possibilità d'interruzioni che permettono il supporto ai semafori.
La prima macchina che montava questa gamma di processori era la l'A5000 di Acorn, con un clock di 25MHz e 13,5 milioni d'istruzioni al secondo (MIPS), ma disponibile anche con 33MHz con 18MIPS[5].
ARMv2a invece aveva come differenza principale la riduzione della complessità della scheda con la prima sperimentazione da parte di ARM nella creazione di SoC.
Si tentò infatti di accoppiare in un'unico componente tutto lo stack fino a ora

realizzato in diverse parti dell'hardware, incorporando il processore ARM, e le tecnologie MEMC, VIDC, IOC.
Possiamo paragonare un ARM250 ad un ARM3 a cui però è stata rimossa la cache ma, con il supporto a SWP. Grazie anche ad un clock di ben 12MHz, poteva raggiungere circa 7MPIS[1].

ARMv3

Il passaggio ad ARMv3 segna anche l'inizio della produzione dell'ultima linea di computer prodotta da parte di Acorn, con l'arrivo appunto del RiscPC[2].
In questa versione vi è un cambiamento radicale dell'architettura dei processori, che vengono modificati pesantemente per sopperire alle mancanze dei modelli precedenti e alle aspettative degli utenti.
Si passò dall'architettura a 26bit a una più moderna a indirizzamento a 32bit spostando inoltre le flag su di un altro registro separato.
Il supporto però alla vecchia modalità con PC e PSR nella stessa locazione di memoria venne mantenuto, il che comportò una l'adozione di diverse scelte di progetto contrastanti. L'esempio più eclatante fu infatti la dimensione massima dello slot dei task a 28MB dove invece erano disponibili 128MB. Il processore infatti in modalità 26bit era in grado d'indirizzare solo 64MB.
I registri vennero ingranditi per via dell'aumento di modalità disponibili ma, a livello di programmazione, rimanevano sempre della stessa dimensione.
Vennero introdotte sei nuove modalità di elaborazione, e con l'occasione vennero anche degli stati originali da USR26, SVC26, IRQ26 e FIQ26 a User32, Supervisor32, IRQ32, FIQ32, Abort32, Undefined32[6].
Fu introdotto quindi anche il supporto a CPSR / SPSR[7] per salvare lo stato corrente dei registri in aggiunta a MRS e MSR per permetterne la lettura e la scrittura.

[1]Da https://en.wikipedia.org/wiki/List _of_ARM_microarchitectures
[2]Da http://chrisacorns.computinghistory.org.uk/Computers/RiscPC600.html

Infine alcune novità interessanti riguardavano il supporto alle operazioni statiche per gestire lo stop del clock e la possibilità di lavorare in modalità Endian-agnostic, ovvero senza premurarsi di controllare se la memoria fosse big endian o little endian.
Della famiglia ARMv3 fanno parte processori come ARM6 che fu distribuito in diverse flavour come l'ARM610[3]. Questo utilizzava ARM6 come processore primario ma consumava meno energia della sua controparte ARM60. Si distacca dai suoi fratelli ARM650 il quale aveva il supporto nativo all'I/O e la memoria onboard per i dispositivi embedded più performanti.
Un'ulteriore iterazione interessante fu l'ARM610 dato che venne molto utilizzato come processore per i RiscPC. Questo chip utilizzava un core ARM6 a 33MHz con 4KB di cache e inoltre una Memory Managment Unit, con una potenza computazionale di 27MIPS.
Infine ARM710 fu l'ultimo nato della generazione ARMv3, paragonabile ad un ARM610 rivisitato.
Le differenze principali si riscontravano prevalentemente nella progettazione fisica del processore che aveva una cache a 8KB, indirizzi per il buffer in scrittura raddoppiati e tecnologia TLB all'interno dell'MMU e numerosi cambiamenti interni nelle temporizzazioni. Questo ha portato il processore, con soli 40MHz a raggiungere 36MIPS.
Questo processore però segnava l'inizio della frammentazione della nomenclatura, portando con se un set di opzioni di personalizzazione che venivano identificate tramite i suffissi al nome dell'architettura.

2.1.3 Specifiche della nuova generazione ARM

I processori ARM di nuova generazione si sono molto distaccati da quelli precedenti e questo ha comportato una profonda modifica del loro modo di funzionare.
Sebbene la potenza all'interno dei nuovi processori non manchi, l'obiettivo

[3]Datasheet http://www.home.marutan.net/arcemdocs/ARM610.pdf

principale è un altro. Infatti il focus principale è spostato sul consumo di batteria, da sempre ritenuto il tallone d'Achille dei normali processori per desktop e portatili.
Per far fronte alle richieste contrastanti di Hardware a basso consumo e alta potenza, ARM è stata costretta a passare a un'architettura Harvard.
L'evoluzione susseguita a questo cambio di tecnologia però ha modificato completamente il sistema ARM e lo spingendolo a evolversi sino ad arrivare ai livelli odierni.

ARMv4

In questa versione della famiglia ARM vennero inserite diverse migliorie significative come l'inserimento della moltiplicazione tra numeri con e senza segno e tra quelli con in formato long.
I processori di spicco di questa famiglia furono ARM8 e StrongARM SA110.
ARM8 vide un'estensione significativa della pipeline a cinque voci e un fetcher d'istruzioni speculativo offrendo così 50MIPS con un clock di appena 55MHz.
Il vero protagonista di questa generazione fu però StrongARM SA-110, sviluppato da una ditta non ARM, la Digital, che aumentò la pipeline ma trasformò l'architettura a una più moderna Harvard.
Questo si ottenne suddividendo le istruzioni e i dati in cache separate entrambe da 16KB.
L'ulteriore aumento del clock sino a 200MHz portò ad una velocità teorica di 230 MIPS con un consumo di poco inferiore al watt[8].
Tutto vanificato però dalle istruzioni sincrone del resto dell'hardware che, lavorando a velocità di clock nettamente inferiori, riducevano di molto le prestazioni reali del prodotto.

ARMv4T

La famiglia ARMv4T fu una svolta non da poco nel panorama ARM per via dell'introduzione alla tecnologia Thumb, come evocato dal suffisso T del nome.
Questa tecnologia fu sviluppata per migliorare la densità del codice compilato sin dal rilascio del primo processore ARM7TDMI.
Introducendo uno stato e set d'istruzioni proprio il processore dotato di tecnologia Thumb[9] esegue le operazioni che sono un sottoinsieme di quelle ARM però indirizzate 16bit.
Il risparmio di memoria veniva effettuato rendendo impliciti alcuni operandi e facendo delle limitazioni sulle possibilità delle operazioni rispetto all'insieme d'istruzioni standard ARM.
Questo portava a delle restrizioni a livello di operazioni come ad esempio la riduzione dell'accesso a solo metà di tutti i registri CPU o l'impossibilità di creare ramificazioni se non tramite istruzioni condizionali.
Il vantaggio riscontrato fu che le istruzioni più brevi aumentarono la densità globale del codice sacrificando però la concatenazione. Questo approccio risulta vincente in situazioni dove la dimensione di alcune memorie o porte in uscita è inferiore a 32bit dato che una minor quantità di codice deve essere caricata ed eseguita.
Inoltre la possibilità di eseguire codice più corto permette di potenziare il processore ARM anche in dispositivi meno costosi dato che porte a 32bit implicano un maggior costo del circuito. ARM7TDMI è una delle più grandi storie di successo della famiglia ARM.
In questo processore troviamo un core ARM7 con tecnologia Thumb, la possibilità di avere un sistema di Debug, un moltiplicatore e un in-circuit emulator (ICE).
ICE fornisce una finestra in un sistema embedded dove il programmatore utilizza un emulatore per caricare il programma e lo esegue passo per passo al fine di controllare l'evoluzione dei dati.
In termini tecnici, offre una pipeline a tre stadi, una cache unificata di di-

mensioni 8K (seconda core), e decodifica istruzioni Thumb. La ARM7TDMI offre anche la capacità di moltiplicazione migliorate viste la prima volta nello StrongARM, e riesce a raggiungere i 36 MIPS con 40MHz[10].
Il processore fu utilizzato da un gran numero di apparecchiature a bassa potenza dai router ai lettori MP3. I prodotti più famosi che montarono chip basati su ARM7TDMI furono l'iPod di Apple, il Game Boy Advance della Nintendo e il Lego Mindstorms NXT.

ARMv7-8

Con l'avvento della versione 7 di ARM il processore comincia a diventare un prodotto maturo e inizia a differenziare la gamma di processori attraverso i core Cortex.
Nella tabella 2.1 viene riportato una breve comprazione della famiglia Cortex.
Un'importante introduzione all'interno di questa versione di processori ARM

Profilo applicazione: Cortex-A	Profilo real-time: Cortex-R	Profilo Microcontrollore Cortex-M
32bit e 64bit	32bit	32bit
Set di istruzioni A32, T32 e A64	Set di istruzioni A32 e T32	Solo set di istruzioni T32 / Thumb
Sistema a memoria virtuale	Sistema a memoria protetta (memoria virtuale solo opzionale)	Sistema a memoria protetta
Supporto ai sistemi operativi complessi	Ottimizzato per sistemi real-time	Ottimizzato per applicazioni a microcolltrollore

Tabella 2.1: Tabella comparativa delle varie architetture Cortex disponibili sul mercato.

Fonte http://www.arm.com/products/processors/instruction-set-architectures/index.php

è stata la tecnologia Thumb-2 [11].
Questa soluzione è un'evoluzione del Thumb già visto in precedenza ed esten-

de le operazioni presenti con l'aggiunta di un insieme di comandi a 32bit e a dimensione variabile. Lo scopo principe di questa tecnologia è quello di ottenere una densità di codice simile a quella di Thumb ma con delle prestazioni migliorate.
Questo obiettivo venne raggiunto con l'aggiunta d'istruzioni per la manipolazione dei campi di bit, le tabelle ramificate e le istruzioni condizionali non presenti nella versione precedente. Inoltre un nuovo compilatore permette di gestire in maniera automatica le istruzioni scegliendo se usare Thumb o ARM normale nell'esecuzione del flusso di codice.
Il supporto a questa tecnologia è presente in Cortex-A e Cortex-R ma non per Cortex-M che supporta solo la vecchia versione.

L'ARM Cortex-A è un gruppo di processori ARM a 32-bit e 64-bit di ultima generazione pensati per l'uso applicativo.
La principale differenza tra la tipologia di Cortex-A e le altre è la presenza dell'MMU sul chip e il supporto alla memoria virtuale, richieste da molti sistemi operativi[12].

2.1.4 Single Board Computer

Un single-board computer è un dispositivo completamente costruito su di una singola scheda con tutte le componenti saldate su di essa.
Le prime schede prodotte con questa filosofia risalgono agli anni settanta ma è solo oggi, grazie all'avvento di processori ARM più potenti e performanti, che queste hanno effettivamente preso piede nel mercato di massa.
Infatti con l'avanzare di queste tecnologie e l'abbassamento generale dei costi si è riusciti a rendere appetibili questi prodotti anche a persone non specializzate.
A differenza dei computer desktop però queste schede spesso non possono essere espanse nelle componenti core come RAM, CPU e GPU anche se questa non è una limitazione così sentita dagli utilizzatori.

In questo lavoro ci occuperemo prevalentemente delle schede di largo consumo che hanno avuto la maggior risposta dal mercato di massa.

Raspberry PI

Raspberry PI (Figura 2.1) rappresenta la più famosa scheda single-board computer in commercio attualmente[4] e, con oltre cinque milioni di pezzi venduti in tutto il mondo[5], anche la più acquistata.

Un successo prorompente che ha avuto la capacità di attrarre comunità di

Figura 2.1: La nuova scheda quad-core di Raspberry PI 3, sono disponibili pin per il GPIO, 4 porte USB, una Porta Ethernet, uscita HDMI, uscita audio

Fonte immagine https://www.raspberrypi.org/blog/raspberry-pi-3-on-sale/.

sviluppatori e fan del prodotto dal prezzo competitivo di soli 35$ più tasse. Il progetto è pensato per essere una piattaforma Open Source completa di tutte le funzioni di un personal computer, con il focus su utenti di tutti i tipi dagli scolari ai progettisti di sistemi embedded.

[4]Da https://www.theguardian.com/technology/2015/feb/18/raspberry-pi-becomes-best-selling-british-computer

[5]Da https://www.raspberrypi.org/blog/five-million-sold/

L'enorme successo è anche dovuto all'hardware a basso consumo che monta questa scheda, che con ogni versione diventa via via più potente grazie appunto ai core ARM.
Si è partiti da una scheda che montava un processore ARM11 (famiglia ARMv6) che era paragonabile a quello montato in smartphone come iPhone 3GS.
Il SoC ARM1176JZF-S aveva 700MHz e 256MB di ram integrata.
Questo però è stato aggiornato nei modelli successivi che sono stati dotati di processori via via più potenti.
Sulla versione due fu inserito quand-core Cortex-A7 a 900MHz[6], mentre venne scelto un Cortex-A57 64bit a 1,2GHz per la terza version[7].
Le prestazioni durante il funzionamento della prima generazione furono apprezzabili ma non eccellenti, riuscendo a raggiungere nel mondo reale solo 0,041 GFLOPS.
Questo traguardo però l'ha portata ad essere una delle schede con le maggiori performance per watt oltre ad essere tra i meno costosi dispositivi general purpose[8].
In teoria Raspberry Pi 2 è all'incirca dalle 4 alle 6 volte più potente rispetto al suo predecessore, con lo stesso rapporto per la versione 3.
Ogni modello è stato fornito in due possibili versioni A e B, i quali differiscono per prestazioni ma anche per tipologia di porte montate lo schema di base però delle connessioni della scheda rimane sempre lo stesso.
I dispositivi d'IO sono in comunicazione con il SoC ARM, il quale è connesso con l'hub USB su cui troviamo montata anche la porta Ethernet.
Quest'ultima è per molti modelli solo un optional e corrisponde ad una porta RJ45 a soli 100Mb/s.
Nelle nuove versioni della scheda vi è la presenza inoltre di una scheda Wi-Fi e Bluetooth integrata.

[6]Da https://www.raspberrypi.org/products/raspberry-pi-2-model-b/
[7]Da https://www.raspberrypi.org/products/raspberry-pi-3-model-b/
[8]Da http://www.phoronix.com/scan.php?page=article&item=raspberry-pi-burst&num=1

La scheda possiede una GPU dedicata con un'uscita HDMI come interfaccia per lo schermo, in grado di funzionare in HD o Full HD a seconda della versione. Il sistema grafico integra la possibilità di decodificare molti formati video standard anche se nessuna versione supporta il nuovo H.265.
Infine la scheda non è fornita di un orologio, bensì si affida a programmi per la gestione del tempo, che acquisisce ad ogni boot.
Questa scelta è stata fatta per rimuovere la batteria di backup, ma l'orologio può comunque essere aggiunto tramite le porte GPIO tramite l'interfaccia I^2C.

UDOO

Figura 2.2: La scheda UDOO quad fornita di dissipatore attivo e pin per Arduino 1

Fonte immagine http://shop.udoo.org/eu/home/udoo-quad.html.

La scheda UDOO (figura 2.2) è un progetto italo-americano crowdfoun-

ded lanciato su Kickstarter il 9 aprile 2013[9]. La campagna ha riscosso molto successo ed è stata subito finanziata da migliaia di sottoscrittori.
La scheda infatti ha offerto due varianti al suo inizio, una dual core e una quad basate su Cortex-A9 con Wi-Fi, Ethernet già integrati.
L'obiettivo della scheda, sebbene con un costo più elevato delle altre controparti, era quello di fornire un dispositivo ad alte prestazioni a prezzi contenuti rispetto alle architetture x86, consentendo la riproduzione di video ad alta definizione e con prestazioni 3D interessanti.
Il processore i.MX 6 quad[10] fa uso di acceleratori hardware dedicato, al fine di soddisfare le richieste prestazionali multimediali. L'uso di acceleratori hardware grafici è un fattore chiave per ottenere alte prestazioni ma con un consumo energetico ridotto, lasciando così anche i processori relativamente liberi di svolgere i loro compiti.
Un ulteriore punto di forza di questa scheda è stata l'integrazione e la compatibilità con Arduino per il controllo di attuatori e sensori.

ODROID

La sigla ODROID, acronimo che sta per Open Droid, è il nome del progetto di una single board computer pensata per lo sviluppo dell'hardware e del software (Figura 2.3).
Questa scheda viene lanciata come prima piattaforma di sviluppo per i giochi su console con sistema operativo Android sul finire del 2009[11].
Con il passare del tempo la ditta si è focalizzata nella scelta del processore Exynos sviluppato da Samsung per creare il connubio tra potenza e consumo energetico richiesto dagli sviluppatori e utenti.
Questa scelta li ha portati a sviluppare una tra le schede più potenti in commercio, tanto che molte loro CPU vengono solitamente fornite con sistemi di

[9]Da https://www.kickstarter.com/projects/udoo/udoo-android-linux-arduino-in-a-tiny-single-board
[10]Da http://www.udoo.org/udoo-dual-and-quad/
[11]Da http://odroid.com/dokuwiki/doku.php

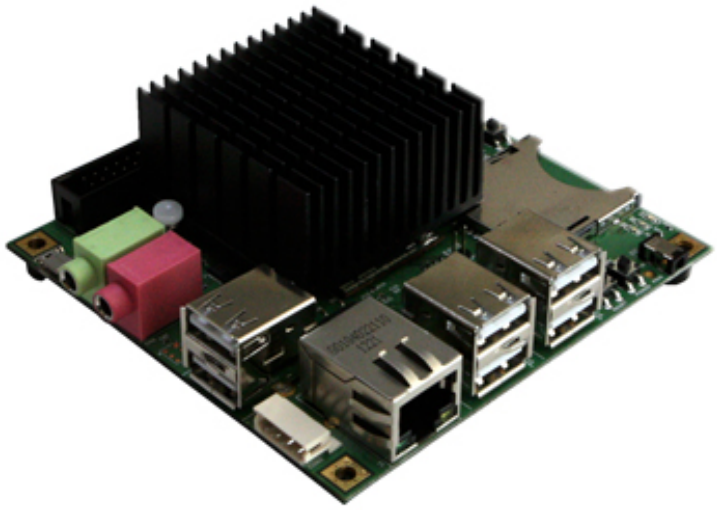

Figura 2.3: La scheda ODROID X2 dotata di processore Samsung Exynos4412

Fonte immagine http://www.hardkernel.com/.

dissipazione passiva.
La gamma si è ulteriormente ampliata con ODROID-X2, rilasciato nell'autunno 2012, che era definita dai produttori la più economica scheda quad core in commercio.
La scheda è stata ulteriormente aggiornata sino a montare un quad core a 2GHz basato su un Cortex-A53 della Armlogic con ben 2GB di ram DDR3 a 912MHz.
Viene definita ad oggi la più potente e meno costosa single board computer disponibile sul mercato, dotata di una GPU Mali 450MP e una scheda Ethernet Giga-bit.

BeagleBoard

BeagleBoard (Figura 2.4) è una scheda Open Source a basso consumo energetico prodotta da Texas Instrument con l'obiettivo di creare una piat-

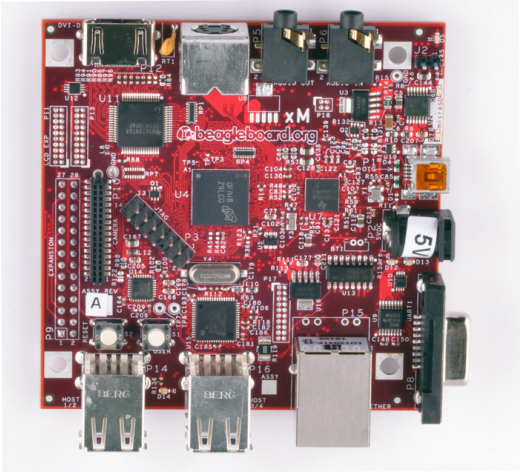

Figura 2.4: La scheda Open Source e Open Hardware Beagleboard-xm

Fonte immagine http://beagleboard.org/beagleboard-xm.

taforma aperta basata su un SoC OMAP.
La scheda è stata sviluppata da un team di ingeneri come piattaforma pensata per scopi didattici e di ricerca utilizzando hardware e software Open Source.
Infatti tutto l'hardware è stato rilasciato con licenza Creative Commons[12], che garantisce il riuso ma con l'attribuzione della paternità.
Il processore montato è un Cortex-A8 single core da 600MHz[13] con coprocessore NEON SIMD e una scheda video PowerVR SGX530 GPU in grado di fornire accelerazione 2D e 3D.
La RAM montata è da 256MB con ulteriori 256 di NAND come memoria flash interna. La forza di questa scheda sta infatti proprio nei consumi, dato che utilizza solo 2W di potenza erogabili da un connettore USB o da un alimentatore separato da 5V.

[12]Da https://en.wikipedia.org/wiki/BeagleBoard
[13]Da http://beagleboard.org/beagleboard-xm

2.2 Sistemi Operativi Linux

Per meglio comprendere il lavoro svolto e come questo influenzi il panorama dei sistemi operativi disponibili per single board computer andremo ora ad analizzare le principali soluzioni Open Source.
Ci focalizzeremo prevalentemente sui sistemi Linux per via del fatto che sono i più utilizzati all'interno di queste schede, rendendo così anche maggiormente incisivo il paragone tra la soluzione prodotta e quella adottata.

2.2.1 Arch

Arch è una distribuzione molto popolare e minimalista creata da Judd Vinet nel marzo del 2002[14].
Questa distribuzione è un'evoluzione di Slackware che ne costituisce la base fondante, e la estende per introdurre i pacchetti binari per l'installazione di software.
I pacchetti disponibili son creati per processori a 32bit e 64bit e ovviamente ad architettura ARM.
Viene però lasciato un sistema ibrido d'installazione di software simile agli Ebuild di Gentoo per la compilazione automatizzata dei pacchetti chiamata Arch Build System[15].
Il principio fondante su cui Arch si concentra è la semplicità del design, che si realizza nel creare un ambiente facile da comprendere per l'utente. Ciononostante il sistema manca di software grafici per semplificare l'installazione e la gestione dei pacchetti.
Questo la colloca come distribuzione pensata per gli utenti intermedi/avanzati dei sistemi Linux e Unix, per i quali fornisce strutture semplici per la gestione delle configurazioni.
La distribuzione può essere installata tramite uno script a linea di comando

[14]Da https://wiki.archlinux.org/index.php/Arch_Linux
[15]Da https://wiki.archlinux.org/index.php/Arch_Build_System

e nella versione di base non fornisce nessun tipo di software utente con interfaccia grafica come gli ambienti desktop.
Uno dei principali strumenti di gestione presenti all'interno di Arch è *pacman*[16], che si occupa dell'installazione del software tramite i pacchetti binari nelle repository risolvendo le dipendenze.
I software sono posizionati all'interno di una struttura di repository composte a livelli dove vengono organizzati tutti i programmi disponibili.
Le repository community sono inoltre un ulteriore vanto della comunità Arch che fornisce, oltre a quelle ufficiali, il sistema AUR[17].
In queste però non sono presenti pacchetti binari, bensì una collezione di file PKGBUILD[18] che forniscono alla macchina le istruzioni per installare il software desiderato tramite l'Arch Build System.
AUR esiste prevalentemente per problemi di licenza con le repository ufficiali, e per pacchetti upstream modificati.
Vi è una moltitudine di distribuzioni basate su Arch, di cui quella più importante per questa trattazione si può identificare in Arch Linux ARM.
Arch Linux ARM è una derivata che che ha come obiettivo quello di portare la distribuzione su schede ad architettura diversa da quella x86.
Gli obiettivi sono di nuovo la semplicità e il pieno controllo da parte dell'utente con una struttura leggera in grado di rispondere alle varie esigenze senza sacrificare però l'ottimizzazione software.
Offre supporto ad una vasta gamma di processori ARM da ARMv5TE alla versione 8[19].
Il ciclo di sviluppo di Arch e Arch Linux ARM è basato sul paradigma di rolling-release, che comporta una distribuzione aggiornata quotidianamente attraverso i pacchetti e non tramite major relase di tutto il sistema.
Al momento il supporto disponibile è presente per schede come BeagleBoard, CUBOX-i, PandaBoard, Raspberry Pi. Ma non manca anche il supporto per

[16]Da https://wiki.archlinux.org/index.php/Pacman
[17]Da https://wiki.archlinux.org/index.php/Arch_User_Repository
[18]Da https://wiki.archlinux.org/index.php/PKGBUILD
[19]Da https://archlinuxarm.org/

dispositivi come alcuni Chromebook Samsung e Acer basati su architettura ARM.

2.2.2 Debian

Debian è una distribuzione core di Linux tra le più famose ed utilizzate in ambiti disparati. Offre supporto nativo infatti a molte architetture inclusa la ARM.
Il motto di questa distribuzione è quello di essere universale, eseguibile in tutte le tipologie di PC esistenti.
È una distribuzione contenente solamente software free, di cui la maggior parte viene rilasciata tramite licenza General Public License, che può contare su repository tra le più fornite in assoluto.
Le operazioni di aggiunta dei pacchetti vengono effettuate tramite lo strumento chiamato *apt-get*[20], il quale si occupa dell'installazione e della gestione delle dipendenze.
Il gestore può essere utilizzato via linea di comando o anche tramite programmi con interfaccia grafica, per facilitare l'approccio a questo sistema da parte di utenti non esperti.
Il programma *apt-get* è un'interfaccia a *dpkg*[21] che si occupa della gestione del database dei pacchetti e dello decompressione e installazione di nuovi deb nel sistema.
Ad *apt-get* si contrappone spesso un software nominato *Aptitude*[22] che si sostituisce a questo e promette una migliore gestione delle dipendenze dei programmi installati.
Il supporto a schede come ad esempio BeagleBoard viene fornito direttamente upstream nella distribuzione, inoltre ci sono sempre nuovi sforzi per ampliare

[20]Da https://wiki.debian.org/apt-get
[21]Da https://wiki.debian.org/dpkg
[22]Da https://wiki.debian.org/it/Aptitude

il parco hardware su cui è possibile installare Debian.

Raspbian

Debian è una distribuzione facilmente estendibile e con molti utenti attivi nelle comunità. Questo fattore ha portato una base solida di utenza esperta e di entusiasti che ha fatto crescere di molto la distribuzione.
Dati questi presupposti al lancio di Raspberry PI fu creato Raspbian[23], un sistema operativo basato su Debian ottimizzato per girare sull'Hardware di Raspberry.
Raspbian non deve però essere inteso come un sistema operativo contente solo pacchetti core, bensì una rivisitazione ottimizzata di Debian, di modo da sfruttare al meglio l'hardware fornito.

UDOObuntu

Come per Raspbian, UDOOBuntu[24] è una versione ottimizzata del sistema operativo per Ubuntu (basato su Debian) per le schede della famiglia UDOO.
La distribuzione si pone come obiettivo quello di creare il migliore sistema operativo per le single board prodotte da UDOO.
Per incrementare la velocità del sistema sono stati modificati finemente le impostazioni di Boot e di gestione della RAM, inoltre è stato incluso il supporto nativo ai calcoli in virgola mobile di tipo Hard.
Il sistema ha come interfaccia grafica LXDE per offrire una maggiore velocità e reattività.

[23] https://www.raspbian.org/
[24] http://www.udoo.org/download-udoobuntu-2-rc2/

2.2.3 Fedora

Fedora[25] è una distribuzione in versione comunitaria di RedHat che ha come obiettivo primario l'integrazione e l'innovazione delle nuove tecnologie e di Linux.
Grazie infatti al team di sviluppo e al lavoro a stretto contatto con le comunità di ogni distribuzione, vengono trovate soluzioni e miglioramenti al software fornito, così da garantire sempre altissimi standard di qualità.
Questo modo di lavorare ha fatto si che questa fosse la distribuzione preferita da Linus Torvalds in persona[26]. L'interfaccia grafica del sistema di default è GNOME, ma sono presenti diverse versioni con diversi ambienti grafici.
Fedora è distribuita in tre diverse versioni principali:

- Server;
- Cloud;
- Workstation.

Il gestore di pacchetti presente in questa distribuzione è *DNF*, che si occupa della risoluzione delle dipendenze e dell'installazione dei software presenti nelle repository ufficiali e comunitarie.
DNF è un'interfaccia a rpm che si occupa di mantenere il database dei pacchetti e d'installare effettivamente il contenuto nel sistema.
I pacchetti disponibili all'interno del sistema sono di tipo RPM, i quali sono file compressi generati tramite un file di specifica. Questi file ci forniscono i passi per compilare e installare il software nel sistema.
Questo sistema operativo fornisce a livello upstream il supporto all'architettura ARM, con pacchetti già precompilati.

[25] https://fedoraproject.org/wiki/Fedora_Project_Wiki
[26] Da http://www.tuxradar.com/content/interview-linus-torvalds-linux-format-163

2.2.4 Toolchain e chroot

Una toolchain[27] è un sistema di software utilizzata per la compilazione di software per un architettura tramite un'altra. Queste tipologie di software vengono utilizzate per la creazione di pacchetti binari e delle immagini di sistema.
La maggior parte delle distribuzioni multi piattaforma al momento utilizza un sistema di cross-compilazione.
Ogni qualvolta si vuole compilare un pacchetto viene creata una nuova struttura di directory all'interno di una chroot separata del sistema[28].
Chroot è un modalità di esecuzione per sistemi Linux e Unix pensato per modificare una root montata però in una locazione di memoria differente da quella attuale. È una tecnica di sandboxing che previene le modifiche al sistema pur permettendo l'accesso tramite utente root.
Vengono quindi aggiunte le dipendenze di sistema necessarie al software desiderato generando così un ambiente pronto per la compilazione del nostro software. Il risultato prodotto viene pacchettizzato all'interno di un file binario compresso pronto per l'installazione.
La fase di creazione dell'immagine invece prevede l'esportazione della chroot corrente in un file compresso. In questa vengono installati tutti i pacchetti software selezionati dallo sviluppatore per poi essere rilasciata al pubblico sotto forma di file immagine.

[27] https://wiki.debian.org/ToolChain/Cross
[28] https://wiki.archlinux.org/index.php/DeveloperWiki:Building_in_a_Clean_Chroot

Capitolo 3

Tecnologie Utilizzate

In questa capitolo ci occuperemo di descrivere gli strumenti Software e Hardware utilizzati in questo lavoro.

3.1 Gentoo e Sabayon Linux

L'obiettivo di questo lavoro è quello di portare la distribuzione Sabayon all'interno di single board computer.
Per strutturare la migliore strategia da adottare cominciamo con l'analizzare il sistema operativo nel dettaglio.
Inizieremo con il descrivere il sistema operativo Gentoo Linux e il suo gestore di pacchetti per poi focalizzarci su Sabayon e le sue peculiarità.

3.1.1 Gentoo

Gentoo al contrario dei sistemi operativi sin qui trattati è una meta-distribuzione[1].
Con il termine meta-distribuzione intendiamo un sistema operativo ibrido in cui è possibile modificare e configurare ogni aspetto compresi kernel e pacchetti core. Ad esempio non vi sono limitazioni per quanto riguarda il kernel, che può essere semplicemente sostituito, passando da Linux ad un BSD.

[1] Da https://www.gentoo.org/get-started/about/

La sua struttura così diviene tra le più versatili e estensibili nel panorama Linux.
Il termine "Gentoo" deriva dal pinguino omonimo tra i più veloci nuotatori del genere dei Pygoscelis[2]. Venne scelto da Daniel Robbins per riflettere il concetto di velocità alla base del sistema. Infatti questo è pensato e progettato per essere compilato e ottimizzato per la macchina su cui esegue.
Al momento il sistema è gestito da una associazione non-profit denominata Gentoo Foundation che possiede tutti i diritti. Alla base di questa strutturazione vi è un concilio votato dagli sviluppatori attivi del progetto.

Portage

Il gestore di pacchetti di Gentoo, denominato Portage[3], ed è pensato e scritto per essere modulare, flessibile e facilmente mantenibile.
Portage ha come suo linguaggio principale Python a cui affianca Bash per le operazioni sul sistema. Questi sono stati scelti per la loro popolarità e la facile lettura da parte di tutta la comunità degli sviluppatori.
Questo packet manager si basa su di un concetto differente da quello delle altre distribuzioni, infatti nelle repository non vengono inseriti i precompilati, bensì file di specifica contenti i passi da seguire per installare il software. A questo scopo vengono utilizzati gli Ebuild che sono paragonabili ai PKGBUILD di Arch e ai file di specifica di Fedora. Questi vengono raccolti nella cartella "*/usr/portage*" dove vengono suddivisi in categorie.
Per verificare la differenza di versione la struttura delle directory nella macchina locale e quella remota viene utilizzato rsync, che si premura di sincronizzarsi con le repository remote.
Portage non è fornito di un'interfaccia grafica, anche se è teoricamente supportato da PackageKit[4]. Può essere dunque utilizzato solo tramite il comando *emerge*.

[2]http://animals.nationalgeographic.com/animals/birds/gentoo-penguin/
[3]Documentazione di Portage: https://devmanual.gentoo.org/index.html
[4]https://www.freedesktop.org/software/PackageKit/pk-matrix.html

Con questo strumento è possibile installare i pacchetti all'interno del sistema specificando opzioni per la compilazione.
Il software si occupa di gestire in maniera autonoma le dipendenze riportando all'utente di risolvere manualmente solo i conflitti che potrebbero bloccare l'installazione.
La gestione delle configurazioni di ogni software è lasciata in mano all'utente che ha piena libertà.
Questo è possibile grazie alle USE flag che sono utilizzate per gestire le opzioni di compilazione globali o specifiche di ogni software che viene installato.
Le flag permanenti nel sistema possono essere trovate in due locazioni separate:

- nel file *make.conf*, dove vengono gestiti tutti i profili di sistema;
- nei vari file di *package.use*, dove possiamo specificare le flag specifiche del programma.

Alcune flag inoltre sono disponibili solo dalla selezione di un profilo di sistema, che è un set di configurazioni standard fornite da Gentoo, tra cui si può scegliere.
Oltre ai metodi sopra citati si possono definire le flag di compilazione anche direttamente al lancio di emerge, passandole come variabili d'ambiente.
Quando si richiede d'installare un pacchetto, Portage si occupa della gestione delle dipendenze e se queste sono soddisfatte procede con lo scaricare e decomprimere il software sulla macchina.
Seguendo le istruzioni contenute nell'Ebuild il programma viene compilato all'interno di un sistema chiuso denominato Sandbox. Una Sandbox è uno strumento utilizzato in informatica per relegare un software all'interno di un'area al di fuori del sistema stesso così che le sue operazioni non abbiano effetto sulla macchina principale.
I file prodotti dal programma di compilazione vengono uniti, se non producono conflitti, a quelli di sistema.
Gli Ebuild che scandiscono le operazioni sono file scritti in linguaggio Bash

a cui è stata estesa la sintassi tramite delle apposite classi Python chiamate Eclass.
Nella parte iniziale dell'Ebuild sono presenti la dichiarazione del tipo di API Portage che il file utilizza per la compilazione e le classi che vengono importate per estendere la sintassi.
Di seguito possiamo trovare dei parametri che servono a dare una descrizione del pacchetto e la posizione online del sorgente.
Il sistema di slot di Portage ci permette di fornire pacchetti con lo stesso atomo ma con versioni differenti che non hanno conflitti di file.
Le keyword invece servono a indicare quale tipologia di architettura il software supporta.
Le dipendenze vengono divise in diverse tipologie di cui le più importanti sono quelle di esecuzione (RDEPEND) o di compilazione (DEPEND).
Successivamente a questa fase di preambolo vengono specificate le varie fasi tramite funzioni apposite fornite dalle API di Portage o dalle Eclass di estensione.
In ogni fase può essere scelto il comportamento del gestore di pacchetti e vengono specificate tutti i passi per creare l'eseguibile desiderato.
Subito dopo sono presenti le funzioni di compilazione che vengono interpretate durante l'esecuzione di Portage per portare a compimento l'installazione.
Oltre alle repository ufficiali di Gentoo, ogni utente può creare il proprio albero di Ebuild da inserire in Portage.
Queste operazioni sono rese più semplici da un software, chiamato layman, in grado di gestire la sincronizzazione di varie repository.

3.1.2 Sabayon

Sabayon[5] è una distribuzione Linux creata da Fabio Erculiani sulla base di Gentoo, e portata avanti dal team di sviluppo che fornisce supporto attivo agli utenti.
L'obiettivo del progetto è quello di fornire una distribuzione veloce, funzio-

[5]https://wiki.sabayon.org/index.php?title=En:Sabayon_Linux

nale e facile da utilizzare con un sistema di pacchetti precompilati.
Il sistema è basato su Gentoo Linux, che facilità la manutenzione del parco software disponibile garantendo al contempo una semplice interfaccia per l'estensione delle funzionalità.
La distribuzione è basata sul paradigma rolling-release per la gestione degli aggiornamenti della versione del sistema.
Sabayon, contrariamente a molte distribuzioni, fornisce ben due sistemi di gestione dei pacchetti in grado di operare contemporaneamente sulla macchina.
La sua forza sta infatti nella possibilità di estendere la distribuzione tramite Portage di Gentoo senza dover essere vincolati alla compilazione di tutto il parco software.

Entropy

Il gestore di pacchetti sviluppato proprio per questa distribuzione, chiamato Entropy[6], è stato scritto in Python per poter interfacciarsi nativamente alle classi di Portage.
Questo software ha come scopo quello di effettuare l'installazione dei pacchetti all'interno del sistema tramite file compressi, senza compromettere l'interoperabilità con Portage.
Il vantaggio principale di utilizzare file binari è appunto quello del risparmio del tempo di computazione che viene speso solo dal server per compilare e comprimere i pacchetti.
In questo modo si permette all'utente di personalizzare e investire risorse solo per qui pacchetti che necessitano di una configurazione fine sulla sua macchina.
Entropy utilizza un database SQLite per la memorizzazione delle tabelle contenenti dati dei pacchetti, le dipendenze e i file contenuti nel binario.
Viene utilizzato SQLite perché permette di creare un database relazionale

[6]http://wiki.sabayon.org/index.php?title=It:Entropy

senza bisogno di un servizio in background, aumentando inoltre la portabilità del software su diverse piattaforme.
Ogni cambiamento effettuato su questo database viene subito ricreato all'interno di quello di Portage, di modo da mantenere il sistema sempre coerente. Questo però ovviamente non avviene se la modifica viene effettuata sul sistema da parte di Portage ed è quindi necessaria una sincronizzazione tra i due sistemi.
Per la gestione delle repository Entropy viene utilizzato il software Enman, che si occupa di configurare e aggiungere ulteriori repository comunitarie a fianco di quelle ufficiali.
La generazione dei pacchetti lato server viene effettuata utilizzando il software Eit, il quale si occupa di sincronizzare lo stato locale della macchina creando i binari in base a quelli installati nel sistema.
Questo programma è ispirato al software di controllo versione sviluppato da Linus Torvalds chiamato Git per la gestione del codice sorgente di progetti.
Al momento, per le architetture a 64bit, questo viene eseguito in un server su cui vengono installati tutti i programmi richiesti e necessari presenti in Portage.
Quando viene installato un software Eit si occupa di controllare le differenze nello stato del database della macchina server, comprimere i file e inviarli ai mirror remoti.

Molecules

Per la creazione delle immagini software da rilasciare agli utenti è stato sviluppato un framework in Bash denominato Molecules per la generazione automatica.
Si parte da una immagine di base di Sabayon contenente tutte le parti necessarie al funzionamento basilare del sistema in cui verranno inseriti via via i software secondo le specifiche desiderate.
Il programma modifica quest'immagine aggiungendo i pacchetti desiderati ed

eventuali modifiche e configurazioni.
Il tutto avviene in un ambiente chroot da cui viene acquisito il prodotto delle modifiche all'immagine desiderata.

3.2 Docker

Docker[13] è un progetto Open Source tra i più famosi e conosciuti nell'ambito delle SaaS moderne. Il progetto nacque all'interno della ditta dotCloud e crebbe molto attraendo la comunità degli sviluppatori.
Il progetto è scritto in Go e fornisce il supporto alla virtualizzazione tramite container che espongono servizi, così da creare ambienti pronti per essere eseguiti in produzione.
Docker, a differenza dei sistemi tradizionali, non crea un ambiente virtuale con hardware emulato. Si riesce quindi a sfruttare le tecnologie già presenti all'interno dei sistema Linux per isolare i processi e utilizzare solo le risorse richieste dall'applicativo eseguito.
Ogni container è un sistema a se stante, in grado di coesistere con altre istanze all'interno della stessa macchina ospite senza alcun tipo di collisione.
Docker però non è solo un sistema di virtualizzazione ma anche di condivisione del prodotto degli utenti ad un progetto.
La sua strutturazione simile a quella di Git infatti ci permette di condividere le immagini create all'interno di repository remote simili a quelle del sistema di controllo versione.
Questa possibilità ci dà due vantaggi principali:

- la riproducibilità delle operazioni avvenute su una macchina dato che tutti condividono la stessa immagine[14];

- la possibilità di spostarci all'interno della storia dell'immagini create navigando anche solo livello per livello.

Grazie a queste funzionalità si viene a creare, a scapito dello spazio occupato, un backup efficiente delle immagini di sistema.

3.2.1 Utilizzo di Docker all'interno del lavoro

La funzionalità di Docker di unire diversi livelli in un unico sottoprodotto finale ha permesso di semplificare la creazione delle immagini.
Le funzionalità di stratificazione infatti ci permettono di lavorare solo su alcune parti dei file di sistema rendendo indipendente ogni fase della creazione dell'immagine.
Inoltre la virtualizzazione tramite container Docker ha permesso di eseguire le operazioni di compilazione dei pacchetti in ambienti separati dal sistema e al contempo riproducibili.

3.3 Sistemi di Virtualizzazione

I sistemi di virtualizzazione presentati qui di seguito saranno utilizzati all'interno del sistema per permettere l'esecuzione delle architetture ARM anche su processori tradizionali.

3.3.1 VirtualBox

VirtualBox[15] è un sistema di virtualizzazione sviluppato da Oracle per la gestione di macchine virtuali. È un software Open Source di cui esiste una versione commerciale, in grado di eseguire macchine a 32bit e 64bit di tipo tradizionale utilizzando diverse tecnologie.
Possono essere installati la maggior parte dei sistemi operativi esistenti indipendentemente da quello utilizzato nella macchina ospite. Inoltre il software è in grado di utilizzare diverse tecnologie di virtualizzazione al suo interno

differenti da quella sviluppata da Oracle.
Un suo punto di forza come sistema di supervisione per macchine virtuali è la semplicità nella creazione di script.
Questa funzionalità ha permesso di automatizzare la gestione delle macchine utilizzate come base per eseguire Docker.

3.3.2 QEMU

QEMU[16] è un Hypervisor per la gestione di macchine virtuali in grado di effettuare una virtualizzazione di diverse architetture hardware.
Il sistema infatti tenta di emulare la CPU desiderata traducendo automaticamente le chiamate dei software in un set d'istruzioni compatibile con la macchina ospite.
Ovviamente questa operazione è molto lenta ma grazie al modulo KVM si aumenta sensibilmente la velocità di esecuzione.
QEMU si occupa però non solo della traduzione ma anche di fornire un ambiente di emulazione del sistema completo per generare una separazione dei file tra la macchina ospite e quella virtuale.
All'interno di questo lavoro verrà appunto sfruttata la funzionalità di traduzione delle chiamate hardware per permettere l'esecuzione di codice per architetture ARM in macchine ospiti con processore standard.

Capitolo 4

Analisi degli approcci al problema

Nella progettazione del sistema e durante l'effettivo porting della distribuzione sono stati valutati diversi approcci al problema.
In questo capitolo ci occuperemo di analizzare nel dettaglio le varie soluzioni prese in considerazione, per meglio comprenderne i punti deboli e così trovare quella più efficace per i nostri scopi.
Cominceremo con lo studio delle varie infrastrutture possibili per la creazione dei pacchetti ARM per poi passare alle soluzioni per la produzione d'immagini installabili del sistema.

4.1 Infrastruttura

Per creare la distribuzione dobbiamo prima generare un insieme di pacchetti necessari per una immagine completa del sistema.
Andiamo quindi a delineare le varie infrastrutture possibili analizzando prima quella attualmente disponibile per i processori comuni a 64bit come riferimento, per poi focalizzarci su tecniche più specifiche.

4.1.1 Sistema attuale per la compilazione e rilascio

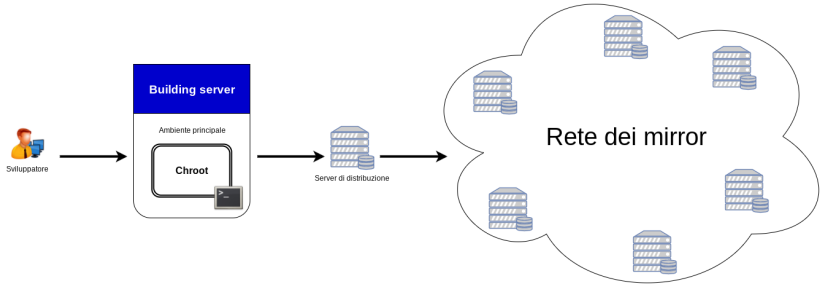

Figura 4.1: Rappresentazione dell'infrastruttura attuale una chroot all'interno di una macchina server builder per la compilazione dei pacchetti per processori standard.

La prima valutazione da effettuare per progettare il porting è l'analisi dell'infrastruttura attuale su cui si basa il sistema dei processori standard a 64bit.
La creazione dei pacchetti su Sabayon al momento, come detto già in precedenza, prevede un sistema basato su una chroot che viene eseguita all'interno di un server remoto in cui gli sviluppatori hanno accesso.
All'interno della chroot vengono generati i pacchetti compilandoli direttamente, per poi prendere gli eseguibili e inserirli in un file compresso.
Questo viene poi inviato ad un server che si occupa del dispaccio ad una serie di mirror che si occupano di distribuirli all'utente finale.
Nella chroot viene tenuto tutto il software installato contemporaneamente per mantenere l'integrità del database e per controllare eventuali problemi con le librerie.
Da questa soluzione possiamo prendere la struttura del sistema di distribuzione, dato che la strutturazione della repository è già pensata per fornire pacchetti multi piattaforma.
Il problema deriva prevalentemente dal sistema di compilazione che richiede

una rivisitazione per funzionare su architetture a x86 a 64bit.
Un ulteriore possibile soluzione è quella di migrare il servizio di compilazione su un'architettura nativa ARM, ma questo comporterebbe anche una perdita di potenza computazionale.

4.1.2 Infrastruttura standard a Toolchain

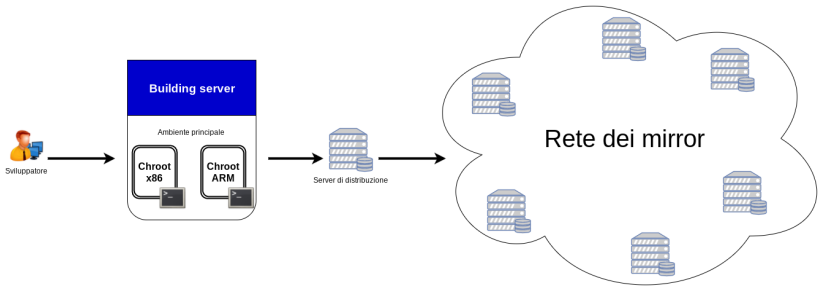

Figura 4.2: Rappresentazione dell'infrastruttura estesa con l'utilizzo di toolchain. Entrambe le chroot si trovano all'interno della stessa macchina ma le istruzioni dell'architettura arm vengono tradotte tramite crossdev.

Per creare la nostra base di pacchetti è possibile utilizzare un sistema simile alle altre distribuzioni utilizzando i software di toolchain presenti all'interno di Gentoo.
Per costruire rapidamente i pacchetti del sistema sarebbe quindi necessario creare una chroot dedicata ad ARM dove compilare tutti i pacchetti necessari.
Si è inizialmente partiti quindi da una chroot di base contenente una Gentoo stage3.
Questa versione di Gentoo è un file compresso contenente tutto l'ambiente minimale di Gentoo creato appositamente per l'installazione di un ambiente minimale.

Si può iniziare la creazione dell'immagine anche da stage precedenti a questo, ma ciò comporterebbe la necessità di compilare anche la parte core del sistema tramite uno script per il bootstrap. Questo però avrebbe come ripercussione una maggior mole di lavoro, utile solo per la creazione di un'immagine molto personalizzata del sistema di cui al momento non abbiamo bisogno.
Una volta scaricato l'ambiente bisogna predisporre una chroot adatta per creare la base di pacchetti. Questa viene ottenuta inserendo Entropy e le sue dipendenze all'interno della stage3 di Gentoo.
Sfortunatamente per ottenere la base di cui necessitiamo bisogna creare un sistema per la cross-compilazione dei programmi in un ambiente x86 per ARM.
Il programma che si occupa di fare ciò è chiamato Crossdev e il suo compito è quello di convertire i binari compilati su architettura x86 in un formato compatibile con ARM.

Limiti dell'infrastruttura a Toolchain

Questo sistema è il più semplice da realizzare dal punto di vista di progettazione, ma porta con se anche molti svantaggi.
Innanzitutto l'utilizzo della chroot per creare pacchetti rende difficoltoso il backup del sistema, dato che questa operazione dovrebbe essere eseguita manualmente o tramite script creati ad-hoc.
Il problema principale è che molti programmi, come ad esempio Python o GCC, eseguono dei test per verificare funzioni e la correttezza dei bytecode generati. Questi test però spesso falliscono dato che i programmi di compilazione tentano di eseguirli su macchine non ARM.
Inoltre l'approccio non tiene conto delle problematiche riguardanti Eit, il quale almeno per il momento, viene utilizzato per tenere traccia del database dell'intero sistema e potrebbe avere problemi di consistenza di librerie se utilizzato in chroot sempre nuove. Questo porta ad avere chroot di dimensioni considerevoli e che, aggiornamento dopo aggiornamento, vengono portate

ad uno stato sempre meno riproducibile.
Infine sistemi di cross-compilazioni sono efficaci, ma riducono la scalabilità del sistema, che deve essere eseguito su una singola macchina con accesso multi-utente.

4.1.3 Infrastruttura nativa ARM

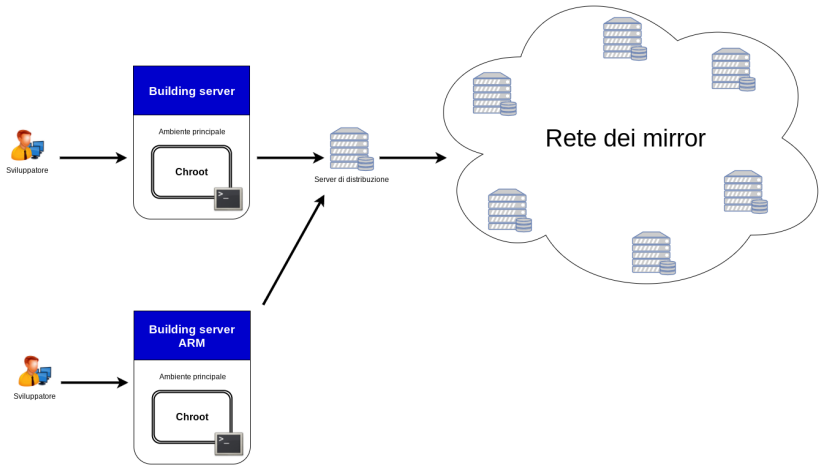

Figura 4.3: Schema dell'infrastruttura che utilizza un buildserver ARM. Questo esegue nativamente una chroot senza bisogno di tradurre le istruzioni e inserisce i pacchetti nel vecchio sistema di distribuzione dei contenuti.

Dato che a disposizione dell'organizzazione è presente anche un server ARM, un'ulteriore semplice soluzione al problema sarebbe quella di creare un ambiente chroot nativo.
Così facendo si potrebbero eliminare eventuali problematiche derivanti dai sistemi di Toolchain, creando inoltre un ambiente molto simile a quello attuale, diminuendo i tempi di apprendimento per i mantainer già coinvolti nel progetto.
Sarebbe quindi necessario creare su di un server remoto un'immagine chroot

dove verrebbero compilati e pacchettizzati i software desiderati. Questi verrebbero poi aggiunti nella repository remota semplicemente utilizzando Eit sul sistema di distribuzione attuale.

Limiti dell'infrastruttura ARM

Sebbene questa soluzione possa sembrare la più adatta è anche la più insidiosa.
I problemi del sistema così strutturato sarebbero prevalentemente legati alla scarsità di risorse dei sistemi ARM che, anche se il divario si sta via via colmando, mancano di potenza computazionale e memoria.
Gli hosting remoti che forniscono server ARM si trovano spesso infatti a vendere agli utenti macchine con poca RAM e poco hard disk, con costi proibitivi per soluzioni più performanti.
La scarsità di memoria provoca in molte schede un blocco del sistema che, per via della dimensione ridotta dell'hard disk, non possono avere un'area di SWAP.
Inoltre la scarsa quantità di memoria sia di RAM che di hard disk impedisce la compilazione di pacchetti avidi di queste risorse.
Infine la lentezza del sistema di compilazione comporta anche tempi maggiori per la creazione dei pacchetti, dai più semplici ai più complessi.
Sebbene la compilazione sia un processo spesso non supervisionato dallo sviluppatore, quando si tenta di risolvere un bug può capitare di dover ricompilare più volte lo stesso pacchetto. Le limitazioni hardware in questo caso comporterebbero un maggior tempo speso da parte di una sola persona per risolvere un problema, creando così un collo di bottiglia.

ARM con distcc

Per ovviare almeno in parte ai problemi di scarsa potenza e di RAM ridotta si può pensare di utilizzare il software distcc per distribuire il carico di

lavoro tra i nodi della rete.
Distcc è un software per lo smistamento del carico di lavoro pensato per velocizzare la compilazione di un software scritto in C[1] su un cluster di macchine.
Ogni macchina con installato al suo interno distcc possiede un servizio in background che negozia con il nodo master la quantità di codice e da compilare e che successivamente ritorna i file oggetto.
Questo però ci porta a due maggiori problemi che sono: la scarsità di macchine ARM e i prezzi relativamente elevati della creazione di un cluster di macchine basato su questa architettura.
Si potrebbe ovviare al problema utilizzando anche macchine x86, ma questo comporterebbe, come già detto, probabili problemi di compilazione dei sorgenti.
Inoltre un cluster di compilazione, se non è composto di sole macchine server affidabili, comporta la possibilità di dover comunque compilare su di una macchina sola il software, eliminando così ogni vantaggio atteso. Questo perché le macchine slave coinvolte potrebbero essere spente o offline al momento in cui vi è la richiesta da parte del master.
Un ulteriore collo di bottiglia è dato anche dalla banda e la latenza introdotta dallo scambio sulla rete dei prodotti della computazione, questo ovviamente se i nodi non si trovano nella stessa rete e vicini geograficamente.

4.1.4 Infrastruttura tramite macchine virtuali

Come abbiamo visto le macchine fisiche ci pongono davanti a diverse problematiche derivanti dall'utilizzo di architetture differenti o dalle limitazioni della architettura ARM stessa.
Una soluzione efficace per ovviare a tutti questi inconvenienti potrebbe derivare dall'utilizzo di macchine virtuali per compilare i nostri pacchetti.
Le macchine virtuali sono molto utili in questo caso poiché permettono di

[1] *distcc* può essere utilizzato anche per i suoi derivati come C++ ad esempio

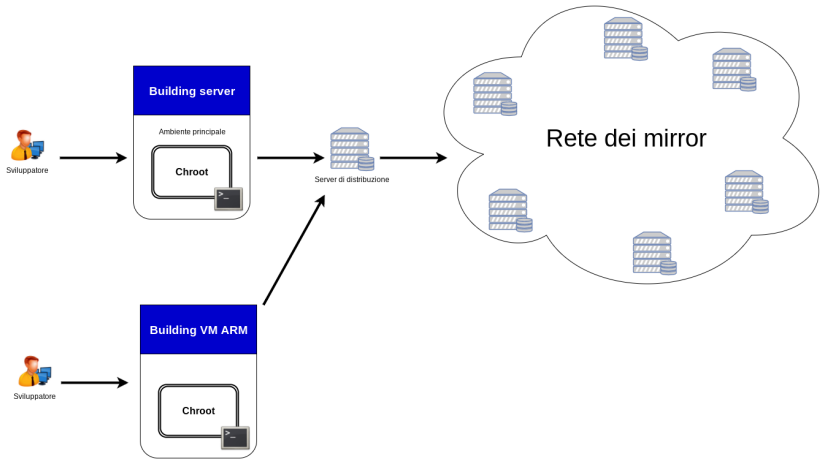

Figura 4.4: Rappresentazione dell'infrastruttura contenente una chroot all'interno di una macchina virtuale ARM.

spostare la complessità del sistema in un ambiente trasportabile ma soprattutto non vincolato dall'hardware.
Il sistema sarebbe quindi in grado di girare su qualsiasi architettura senza l'utilizzo di programmi di cross-compilazione, velocizzando anche l'esecuzione grazie alle primitive di virtualizzazione delle CPU.
Il problema più annoso di questa strutturazione però è l'acquisto di un server virtuale ARM.
Al momento esistono solo poche ditte che forniscono infatti questo servizio e le macchine virtuali messe a disposizione hanno poca larghezza di banda e poca potenza computazionale.
Inoltre queste macchine non possono essere migrate facilmente dato che richiedono necessariamente QEMU come sistema di virtualizzazione, unico in grado di supportare virtual machine di tipo ARM.
L'unica soluzione praticabile sotto questo punto di vista è quella di creare una macchina virtuale all'interno di un server proprietario installando il software opportuno.

Infine, sebbene la soluzione migliorerebbe la situazione riguardante i backup, il sistema degli snapshot non ci da la possibilità di navigare attraverso le modifiche fatte al sistema ma solo di salvare un certo numero di stati della macchina.

4.1.5 Infrastruttura su container Docker

Durante il porting della distribuzione il sistema di creazione dei pacchetti per processori standard è cambiato in maniera considerevole.
L'introduzione delle repository comunitarie all'interno di Sabayon ha portato al suo interno un paradigma di gestione delle immagini di sistema completamente nuovo.
Grazie alla sue funzioni ereditate dai software di controllo versione come Git, Docker[17] ha permesso di sfruttare infrastrutture esistenti e gratuite per la compilazione dei pacchetti.
Sebbene però questo sistema sia molto interessante dal punto di vista della riduzione del costo dell'infrastruttura, della multi-utenza efficiente e della semplificazione dei backup, i servizi che forniscono la virtualizzazione dei container al momento non supportano macchine Docker su architettura ARM.

4.2 Creazione delle immagini

Oltre a quelle derivanti dalla creazione di un'infrastruttura efficace ed efficiente per la compilazione, la creazione delle immagini comporta anch'essa difficoltà non trascurabili.
Abbiamo visto prima come, per via di restrizioni da parte dei fornitori di servizi, fosse impossibile utilizzare il sistema di compilazione implementato nelle repository comunitarie attraverso Docker. Questo ci impedisce di utilizzare la stessa infrastruttura per creare i pacchetti, ma non di utilizzare Docker al posto di Chroot per generare le immagini installabili.

Infatti questa tecnologia ci svincola dall'utilizzo delle chroot come base per la creazione delle immagini, rendendo più agevole sia i backup che le modifiche all'immagine da distribuire.
DockerHub, servizio basato su Docker appunto, ci permette di gestire in maniera efficiente i nostri file dell'immagine di sistema fornendo spazio di archiviazione per i container purché queste siano pubblici.
Docker ci permette di estendere facilmente altre immagini create sulla piattaforma semplicemente creando degli script che ci indichino come un'immagine debba essere modificata.
Il problema di questo sistema rimane appunto la virtualizzazione tramite container, che può essere effettuata solo tramite macchine che girano su architettura ARM.
Inoltre il software Molecules, utilizzato per la generazione delle immagini, non è attualmente in grado di utilizzare container. Questo rende necessario predisporre un passo di conversione da container a immagine per garantire il funzionamento dell'attuale infrastruttura.

4.2.1 Pacchetti kernel specifici

La diversità delle schede presenti sul mercato ha comportato anche la creazione di diverse strategie per l'avvio delle immagini.
La maggior parte dei sistemi generalisti come Arch Linux forniscono supporto creando un kernel non specifico per ogni macchina e rilasciando così un'immagine generica in grado di funzionare ovunque.
Il problema di questo approccio è la necessità di compilare e rilasciare un kernel contenente molti moduli inutili, tenendo in considerazione che la maggior parte di queste schede non può modificare il proprio hardware nel tempo.
Un altro approccio è quello di creare pacchetti specifici con il kernel di ogni architettura e compilarli singolarmente.
Questo però implica una moltiplicazione del lavoro necessario al porting di ogni scheda dato che ogni kernel va compilato con flag particolari e va ef-

fettuato il reverse engineering del sistema operativo supportato ufficialmente.

Capitolo 5

Soluzione adottata

Dopo aver analizzato le soluzioni possibili andremo ora a focalizzarci su quelle che sono state scelte come approccio per la compilazione dei pacchetti che forniranno la base del nostro sistema.
Andremo quindi a definire l'infrastruttura creata e come questa sfrutti al meglio le risorse disponibili in termini di hardware e software.
Passeremo quindi a delineare il sistema di produzione delle immagini che verrà utilizzato per tutte le schede ufficialmente supportate.

5.1 Infrastruttura

Per creare il nostro sistema automatizzato di compilazione dei pacchetti si è scelto di sviluppare una soluzione che funzioni indipendentemente dall'hardware a disposizione.
Anche se è possibile avere accesso a delle schede per la compilazione effettiva del software bisogna ovviare alle restrizioni che l'hardware nativo comporta.
Per via di queste limitazioni si è optato per un sistema ibrido realizzato da una combinazione di script semplici per la gestione delle macchine sia virtuali che fisiche in maniera efficace ed efficiente.

5.1.1 Macchine virtuali

Le problematiche riguardanti gli hardware disponibili hanno fatto subito sorgere la necessità di poter gestire le operazioni di creazione della distribuzione su macchine virtuali.
Sono stati quindi utilizzati tre software per la virtualizzazione del sistema così da renderlo più flessibile e adattabile all'hardware a disposizione, tenendo comunque in considerazione la possibilità di eventuali cambiamenti futuri dell'infrastruttura.

I container Docker

Nel sistema sviluppato per la compilazione dei pacchetti si utilizza un'immagine del sistema creata in un container Docker.
Si è scelto di utilizzare questo sistema per i seguenti motivi:

- sistemi di backup gratuiti ed efficienti;
- possibilità di eseguire container su altri sistemi operativi;
- portabilità dell'immagine;
- gestione delle modifiche multi utente semplificate.

Per quanto riguarda i sistemi di backup, la macchina può essere comodamente salvata sul servizio DockerHub, il quale fornisce spazio di archiviazione gratuito a tutti i progetti pubblici.
Dato che il nostro lavoro si articola su un progetto Open Source è stato possibile quindi utilizzare questo spazio di archiviazione come se fosse una repository Git, permettendo così di salvare lo stato del container creato.
Ad intervalli periodici vengono quindi inviate le modifiche dell'immagine a DockerHub, così da rendere inoltre più semplice l'operazione di fork[1] da par-

[1] Operazione effettuata da un utente per creare una copia della repository di un progetto in un proprio spazio. Viene utilizzata per lavorare e creare modifiche proprie ad un progetto già esistente.

te di altri utenti.
La possibilità di eseguire il container su altre distribuzioni ci permette di compilare anche su macchine con sistema operativo non Sabayon, permettendo così di utilizzare architetture ARM ancora prima di aver creato la base della distribuzione.
Il sistema di DockerHub inoltre crea la possibilità di collaborare semplicemente ad un progetto non solo gestendo l'accesso ad una repository da parte dei diversi utenti, ma fornendo anche un CDN[2] efficiente per distribuire i container creati. Tutti questi vantaggi sono accentuati anche dall'utilizzo particolare che viene fatto della piattaforma Docker in questo progetto.
Infatti un'ulteriore funzionalità che, ai fini del progetto, si è rivelata tra le più utili risiede nella possibilità di montare alcune cartelle direttamente nella macchina host, permettendo quindi l'isolamento delle informazioni.
In questa modo, mentre l'immagine procedeva con la sua evoluzione nella compilazione dei pacchetti, i file sorgenti si trovavano in una locazione separata, riducendo quindi il tempo di download complessivo. Inoltre le configurazioni potevano così essere gestite tramite un set di script presenti su un'altra repository Git.
L'insieme di script per la compilazione manuale è presente all'interno di una repository Git remota e viene utilizzato per automatizzare la creazione dell'ambiente sulla macchina locale.
Lo script *entropy_container* è quello deputato a questo compito e innanzitutto si occupa di scaricare la versione più aggiornata del nostro container all'interno della macchina locale, dopo di che controlla l'esistenza di un'istanza di quest'ultimo fornendo quindi una interfaccia alla shell del sistema. In caso il sistema non sia avvitato alcun container questo viene creato e avviato sulla macchina corrente.
In caso il programma rilevi che c'è un container in fase di salvataggio questo viene unito all'interno all'immagine di partenza utilizzando il comando:

[2]Un Content Distribution Network è un servizio che ha come obiettivo quello distribuire il contenuto agli utenti finali garantendo disponibilità e prestazioni nell'accesso alla risorsa.

```
$ docker commit <CID> <IMG>
```

dove CID è l'id (hash) del container e IMG è l'immagine che vuole essere aggiornata.
Inoltre si occupa di creare un ambiente di test, comunemente chiamato playground, dove l'utente può testare le transazioni quando il container nel nodo corrente è fortemente utilizzato.
Finite le operazioni di modifica manuale sul container questo viene salvato e registrato tramite il comando di commit.
Il sistema di strumenti utilizzato permette un miglior coordinamento da parte di più operatori sulla stessa macchina, riducendo il tempo degli interventi manuali e monitorandoli nelle operazioni effettuate.
Gli script sono inoltre pensati per astrarre dai concetti funzionali di Docker creando così un sistema più semplice anche per i neofiti.

Sistema per l'aggiornamento automatico

Dato che però si vuole minimizzare anche l'intervento umano, inutile quando ripetitivo, all'interno del set di script vengono inserite delle funzioni per l'aggiornamento automatico dei pacchetti.
Questo avviene appunto tramite *emerge* che si occuperà di compilare e generare i file per il pacchetto da comprimere e inviare alle repository.
Lo script tiene traccia dei pacchetti attualmente installati sulla macchina e, grazie all'ausilio del programma *flock* si garantisce la mutua esclusione degli operatori al container.
Quando la routine viene avviata questa sincronizza l'albero dei pacchetti Portage e delle ulteriori repository aggiunte tramite Layman.
Dopo aver quindi generato una lista dei pacchetti da aggiornare viene effettuata la compilazione di ogniuno di essi senza tralasciare il grafo completo

relativo alle dipendenze inverse del pacchetto[3]
Le dipendenze inverse sono un problema annoso dato che un pacchetto aggiornato potrebbe esportare librerie e funzioni per altri programmi a lui collegati. Quindi ogni qualvolta un programma si aggiorna tutti i programmi a esso collegati devono essere ricompilati per evitare inconsistenza nei link alle librerie.

5.1.2 Il sistema host virtuale

Il container Docker ha però bisogno di essere eseguito all'interno di una macchina fisica che funga da ospite. Questa può essere o di tipo ARM o con processore x86, ma nel secondo caso abbiamo bisogno di un'ulteriore astrazione per la compilazione dei binari.
In questo caso è stata creato un container Docker ibrido ARM con all'interno un binario statico di QEMU compilato per AMD64. Questa tecnica sfrutta la caratteristica peculiare di QEMU di essere un traduttore per le chiamate dell'infrastruttura, sollevandoci quindi da tutti quei problemi legati ai software per la cross-compilazione; questo a patto però che nella macchina che verrà usata sia specificato come il sistema deve convertire le chiamate. Ciò avviene impostando sulla macchina virtuale i "magic byte" (Listato 5.1) necessari per identificare i binari di tipo ARM, e specificare l'interprete qemu-static-arm per la sua codifica[18].

```
echo ':arm:M::\x7fELF\x01\x01\x01\x00\x00\x00\x00\x00\x00\x00\
    ↪ x00\x00\x02\x00\x28\x00:\xff\xff\xff\xff\xff\xff\xff\x00\
    ↪ xff\xff\xff\xff\xff\xff\xff\xff\xfe\xff\xff\xff:/usr/bin/
    ↪ qemu-arm-binfmt:' > /proc/sys/fs/binfmt_misc/register
```

Listing 5.1: I magicbyte di Qemu

[3]Viene lanciato *emerge* con le opzioni *–accept-properties=-interactive -t –verbose –update –nospinner –oneshot –complete-graph –buildpkg dove –update e –complete-graph* che ci garantiscono l'intera ricompilazione grafo delle dipendenze inverse relative al pacchetto che vogliamo compilare; con *–buildpkg* genera il .tbz2 oggetto da inserire nelle repository.

In questo modo indirizziamo il kernel all'utilizzo di QEMU per convertire le chiamate se vengono incontrati le intestazioni ELF che corrispondono all'architettura ARM.
Per garantire la trasportabilità del sistema e l'esecuzione anche su servizi IaaS esterno si è deciso di strutturare l'infrastruttura per far si che possa essere eseguito all'interno di una macchina virtuale generata automaticamente. È stato quindi creato un insieme di script per automatizzare l'avvio di container che compilino i nostri pacchetti all'interno di una macchina virtuale. Quest'ultima è una macchina VirtualBox definita tramite un file Vagrant[19] che ci permette di creare script per l'esecuzione di un pool di macchine virtuali.

5.1.3 Macchine fisiche

Data l'infrastruttura di macchine virtuali sviluppate, passiamo ora a definire come queste vengono eseguite sull'hardware disponibile.
Al momento della creazione della base di pacchetti il sistema è stato eseguito su architettura ARM nativa attraverso una Raspberry Pi 2. In questa fase le macchine Docker sono state eseguite all'interno attraverso Raspbian.
I nodi iniziali sono poi stati ampliati spostando il carico su un server ARM dedicato remoto che si occupa di gestire le fasi di compilazione automatiche descritte prima.
Il server in questo modo non ha bisogno di eseguire tecnologie ibride per compilare i pacchetti, ma ha comportato diversi svantaggi dal punto di vista dell'hardware a disposizione. Infatti la scarsa memoria di archiviazione e RAM, oltre che la poca potenza della CPU hanno impedito la compilazione di diversi software su questo server.
Inoltre è stato necessario limitare in maniera forte l'utilizzo della RAM da parte dei container per evitare che la macchina si bloccasse e venisse richiesto un ripristino manuale del sistema.

5.1.4 Distribuzione del carico di lavoro

```
FROM sabayon/armhfp
...
RUN equo up && equo u && equo i distcc gcc base-gcc
...
CMD ["/usr/bin/distccd", "--allow", "0.0.0.0/0", "--user", "
    ↪ distcc", "--log-level", "notice", "--log-stderr", "--no-
    ↪ detach"]
```

Listing 5.2: Installazione e configurazione di distcc per il container Docker

Dato che l'infrastruttura con solo il nodo master limitava fortemente la velocità di esecuzione di tutto il sistema, si è scelto d'introdurre la possibilità di distribuire il carico di lavoro su diversi nodi a disposizione.
È stata quindi creata un'immagine Docker apposita che all'avvio lancia un servizio che si occupa di raccogliere le richieste di compilazione, eseguirle e restituire i file oggetto al nodo master della rete.
Questa immagine ci permette di distribuire il carico di lavoro senza dover configurare finemente ogni nodo nella rete.
In Listing 5.2 è presente un esempio della configurazione utilizzata per la generazione del container slave per la compilazione.
In questa maniera distcc gestisce autonomamente la distribuzione del carico da parte del nodo master per compilare pacchetti basati su linguaggio C e derivati.
In figura 5.1 possiamo vedere l'infrastruttura come in produzione in questo momento.

5.2 Creazione dei pacchetti

Per creare la base di pacchetti sono state selezionati opzioni di profilo e le flag di USE necessarie a compilare il sistema, creando così un set di configurazioni che può essere esteso semplicemente modificando la macchina.

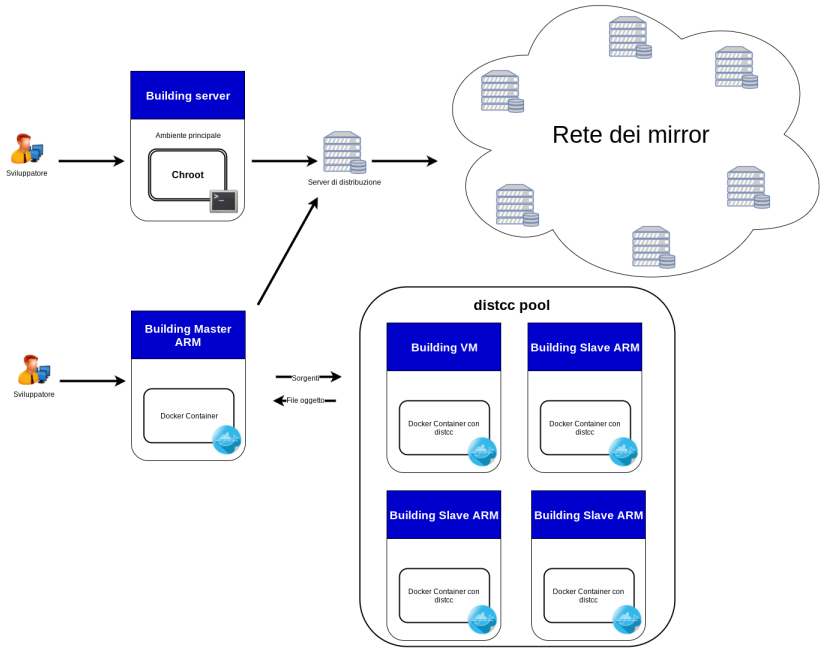

Figura 5.1: Rappresentazione dell'infrastruttura attuale con il pool di macchine per il distcc e il nodo master che si occupa di coordinare e gestire la compilazione.

Sono stati quindi compilati i pacchetti di base del sistema, i software LAMP, i programmi per il networking e quelli Desktop (XFCE e LXQT).
Gentoo permette appunto un'estrema configurazione dei software da installare e questo ci ha quindi permesso di creare pacchetti personalizzati in base alla scheda.
Infatti molte schede hanno bisogno di compilare i software con un particolare set flag USE in modo da ottimizzare l'esecuzione per il SoC a disposizione. Questi sono inseriti all'interno della repository con nel nome dell'atomo il suffisso della scheda di riferimento.
Si è scelta questa soluzione poiché risultava più semplice creare una sola repository per architettura ARM contenente tutti i software.

5.3 Costruzione delle immagini

Dopo aver definito l'infrastruttura di compilazione dei pacchetti, andiamo ora ad illustrare il procedimento per generare le immagini Sabayon partendo dallo stage3 di Gentoo.
Per aumentare la flessibilità della nostra struttura delle immagini, vengono divise in 3 layer principali (Figura 5.2):

- stage3 Sabayon;
- Sabayon base;
- immagini per i dispositivi specifici.

Andiamo ora a definire nel dettaglio il sistema, spiegando come si possa passare da un livello all'altro di queste immagini.

5.3.1 Stage3 Sabayon

Le immagini su DockerHub sono in continua evoluzione e vengono aggiornate dagli sviluppatori delle distribuzioni con tutti i nuovi software disponibili.
Per evitare problemi con le dipendenze, ma anche per avere un sistema controllato e stabile, viene quindi importata una versione di Gentoo non modificata dal loro DockerHub testata e funzionante.
In questa fase vengono effettuate solo alcune piccole operazioni di conversione come:

- aggiornamento del database di portage;
- configurazione del profilo dell'immagine;
- rimozione di alcuni pacchetti come ssh, openrc, etc. che verranno aggiunti di nuovo in seguito se necessario.

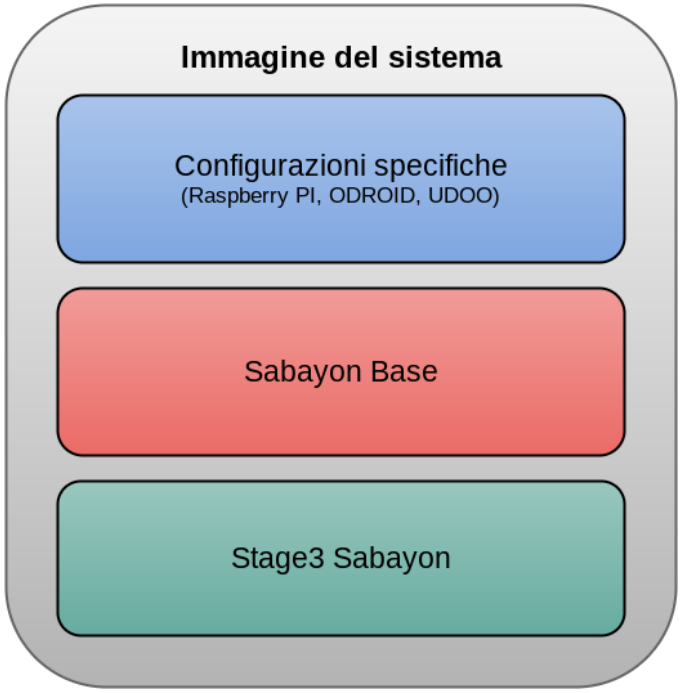

Figura 5.2: Strutturazione dei livelli dell'immagine Docker della distribuzione.

Inoltre per rendere compatibile l'immagine con Entropy bisogna modificare anche le impostazioni per abilitare Python versione 2.7.
Viene quindi scritta quest'immagine e inserita nel dominio di Sabayon all'interno di DockerHub.

5.3.2 Sabayon base

Partendo dalla stage3 viene quindi creata un'immagine di base contenente tutti i pacchetti necessari alla conversione di Gentoo in Sabayon.
In questa fase come primo passo vengono importate le configurazioni necessarie e si procede all'inserimento delle repository ufficiali Sabayon.
Successivamente vengono compilate le dipendenze e viene generato il database per il gestore di pacchetti. Partendo da un file SQLite vuoto, vengono

registrati nel database tramite Entropy tutti i software presenti in quel momento nel sistema.
Viene quindi selezionato il profilo per ARM desiderato e contemporaneamente introdotti i software QEMU per la conversione delle chiamate.
Infine vengono aggiornati i pacchetti installati da Portage nel sistema con le versioni presenti nelle repository remote di Entropy per evitare inconsistenze.
Questa fase di costruzione ci permette di generare l'immagine più importante poiché sarà la nostra base per la creazione del sistema su macchine fisiche ma anche quella per i sistemi di compilazione automatici.

5.3.3 Immagini per i dispositivi fisici

Le immagini dei livelli creati sino ad ora non sono installabili e quindi richiedono un ulteriore passo prima di venire rilasciate.
In una prima fase è necessario configurare tutti i servizi per il funzionamento basilare della scheda.
Saranno quindi reintrodotti servizi come ssh, sudo e ntp attraverso le repository ufficiali di Entropy. Sono inoltre definite le configurazioni di base tramite uno script che però verranno effettuate solo al primo avvio dell'immagine nella scheda.
Viene quindi creato un file Docker per la creazione di un'immagine che inserisca queste configurazioni nell'immagine.
A questo punto, abbiamo definito una struttura a livelli da cui possiamo attingere per creare le immagini e supportare le architetture specifiche. Tutte le modifiche infatti saranno isolate all'interno di un solo stadio Docker che ci servirà per la costruzione dei nostri file.
Ogni immagine specifica potrà essere creata sia da architettura ARM che da architettura AMD64, purché questa supporti la tecnologia *binfmt_misc*.
Quest'ultima è una funzionalità del kernel Linux necessaria per la conversione dei comandi passati automaticamente ad un emulatore dell'architettura.

Raspberry PI

Cominciamo ad analizzare il sistema parlando della prima immagine creata per Raspberry PI.
Dopo aver compilato tutti i pacchetti necessari all'avvio della macchina con processore ARM è stato creato il Dockerfile (riportato in alcuni punti salienti nei Sorgenti A.1 e A.2) per generare l'immagine del nostro dispositivo aggiungendo semplicemente un livello alle immagini precedenti.
In questo modo possiamo definire dei parametri specifici dell'architettura, come la posizione delle partizioni e la definizione delle variabili di ambiente per OpenGL. Si deve quindi invocare Docker all'interno della posizione contenente i file di definizione che provvederà ad assemblare così un container che funga da immagine del sistema.
Per poter però utilizzare l'immagine all'interno del software Molecules per la generazione di un supporto installabile, è necessario estrarre tutto il contenuto del container appena creato in modo che questo possa utilizzarlo come se fosse una chroot.
Per estrarre l'immagine viene utilizzato il comando[17]:

```
docker export sabayon/raspberrypi2 | docker import - sabayon/
    ↪ raspberrypi2-compact
```

Attraverso la funzione di *export* il sistema si occupa di esportare il container appena creato come somma di tutti i livelli dell'immagine. Questo viene poi successivamente importato come nuova immagine definita da un singolo layer derivante dall'unione di tutti gli strati. L'immagine compattata poi verrà estratta da Molecules, utilizzando il programma *undocker*[4].

ODROID e le altre schede

Il procedimento utilizzato per la creazione della scheda Raspberry può essere utilizzato genericamente per altre schede esistenti come la ODROID-

[4]Da https://github.com/larsks/undocker

X2.
Per ogni scheda infatti che si vuole supportare bisogna creare un archi contenente il kernel che si vuole introdurre tramite Docker.
Dopo questa fase bisogna comprendere e definire la struttura dell'immagine che potrebbe richiedere offset di byte per permettere al sistema di leggere la tabella delle partizioni dell'immagine nel modo corretto. Bisogna quindi per ogni release tramite Molecules specificare i passi da eseguire per creare un'immagine del sistema installabile su scheda SD.

5.3.4 Rilascio delle immagini

Con il procedimento sin qui descritto riusciamo ad automatizzare la creazione di ogni livello dell'immagine per ogni scheda, eliminando così l'intervento umano per queste fasi.
Questo ci porta ad un notevole vantaggio, ovvero quello di poter effettuare release automatiche per ogni dispositivo utilizzando sempre Molecules come strumento software per la generazione, limitando quindi l'impatto della modifica all'infrastruttura esistente per il supporto alla nuova architettura.

Conclusioni

In questa tesi ci si è occupati di analizzare e sviluppare una soluzione per il porting della distribuzione Sabayon Linux su architettura ARM.
Per poter raggiungere questo obiettivo il problema è stato diviso in due parti:

- progettazione di un'infrastruttura per la compilazione multi piattaforma;
- creazione di un sistema di rilascio automatizzato delle immagini.

Nella costruzione della infrastruttura sono stati valutati diversi approcci come il metodo classico tramite toolchain, l'utilizzo di hardware nativo e la compilazione tramite virtual machine.
Il sistema di toolchain è stato scartato da subito come tecnica poiché troppo complesso e con difficili tecniche di backup dei file della macchina.
L'hardware nativo di contro ci ha messo di fronte alle criticità derivanti dalle limitazioni forti date dai processori delle macchine a disposizione.
Infine si è analizzato il sistema tramite virtual machine, valutando tre tipologie di soluzioni:

- Docker come sistema per la gestione dei container;
- VirtualBox come hypervisor di macchine virtuali;
- QEMU per la traduzione delle istruzioni.

Docker come gestore di container ci permette di eseguire queste macchine virtuali molto più snelle di quelle classiche fornendoci, oltre che lo spazio di

archiviazione cloud per i backup tramite DockerHub, anche un sistema per la gestione della multi-utenza.
VirtualBox è invece un Hypervisor in grado di gestire una virtualizzazione completa del sistema, ma purtroppo non supporta macchine virtuali con architettura ARM.
Infine QEMU è un sistema di virtualizzazione molto efficace per la traduzione delle chiamate in formato ARM ma con lo svantaggio di essere poco automatizzabile per quanto riguarda i backup.
Si è scelto quindi di creare un ibrido di queste tecnologie utilizzando una immagine Docker creata appositamente per la compilazione. VirtualBox viene utilizzato come Hypervisor per la gestione delle macchine virtuali su architetture x86 e QEMU per la traduzione delle istruzioni del container.
Inoltre l'utilizzo di Docker ci ha permesso di poter migrare semplicemente l'immagine da una macchina all'altra, svincolandoci da una architettura predefinita e dandoci la possibilità di utilizzare nodi di varia natura in maniera del tutto trasparente.
Infine è stato creato un sistema per la distribuzione del carico di lavoro tramite *distcc*, attraverso la definizione di un'immagine Docker contenente tutte le configurazioni necessarie per la definizione automatica della rete.
In seguito alla definizione dell'infrastruttura è stata definita una pipeline per la creazione delle immagini da rilasciare per ogni single board computer partendo da una base Gentoo Stage3.
Questa immagine è stata ampliata per essere trasformata in una base Sabayon aggiungendo tutti i software necessari al funzionamento del secondo gestore di pacchetti.
Dalla base comune sono quindi derivate le immagini proprie per le schede inserendo i kernel con configurazioni specifiche per ogni scheda da supportare. Si è scelto di non compilare i kernel ma semplicemente di estrarli dalle immagini ufficiali data la difficoltà nel gestire le configurazioni particolari di ogni scheda.
Il progetto così strutturato fornisce una base per la compilazione non solo di

Sabayon, ma anche di altre distribuzioni attraverso un metodo molto efficace ma sopratutto riproducibile.
Il crescente interesse infatti per l'architettura ARM per sistemi single board computer e portatili comuni ci porta alla necessità di snellire la pipeline di produzione del sistema.
In questa maniera molti più sviluppatori potrebbero rilasciare la loro versione di distribuzioni Linux aumentando così l'interesse per i dispositivi ad architettura RISC; questo poiché lo svincolo da sistemi di toolchain e chroot permette una più semplice configurazione degli ambienti e una mole di lavoro inferiore per gli sviluppatori.
Inoltre l'utilizzo di piattaforme cloud per l'archiviazione di immagini permette di ridurre la complessità dell'infrastruttura multi utente e la possibilità di esternalizzazione con più facilità i sistemi di compilazione.
Il progetto potrebbe essere estendendo il supporto ad altre famiglie di single board computer senza però compromettere l'ottimizzazione derivante dall'utilizzo di un kernel con configurazioni specifiche.
Infine la metodologia qui esposta potrebbe essere utilizzata per snellire la pipeline di produzione di sistemi operativi basati su Gentoo come Google Chrome OS.

Appendice A

Sorgenti

```
FROM sabayon/armhfp

# Set locales to en_US.UTF-8
ENV LC_ALL=en_US.UTF-8
...
# Installing packages without questions
RUN equo i media-libs/raspberrypi-userland app-admin/rpi-update
    ↪ && \
...
# Perform post-upgrade tasks (mirror sorting, updating
    ↪ repository db)
ADD ./scripts/setup.sh /setup.sh
RUN /bin/bash /setup.sh  && rm -rf /setup.sh

RUN mkdir -p /lib/firmware
ADD ./rpi3_wifi.tar /lib/firmware/

# Set environment variables.
ENV HOME /root

# Define working directory.
WORKDIR /
```

Listing A.1: Dokerfile per Raspberry PI 3

```
#!/bin/bash

/usr/sbin/env-update
. /etc/profile

setup_bootfs_fstab() {
        # add /dev/mmcblk0p1 to /etc/fstab
        local boot_part_type="${1}"
        echo "/dev/mmcblk0p1  /boot  ${boot_part_type}  defaults
            ↪  0 2" >> /etc/fstab
}

setup_rootfs_fstab() {
        echo "/dev/mmcblk0p2 / ext4 noatime 0 1" >> /etc/fstab
}

die() { echo "$@" 1>&2 ; exit 1; }

echo 'SUBSYSTEM=="vchiq",GROUP="video",MODE="0660"' > /etc/udev/
    ↪ rules.d/10-vchiq-permissions.rules
eselect opengl set raspberrypi-userland

rm -rfv /etc/fstab
setup_bootfs_fstab "vfat"
setup_rootfs_fstab

echo "y" | SKIP_BACKUP=1 /usr/sbin/rpi-update

exit 0
```

Listing A.2: Script eseguiti sul container dal Dockerfile

Bibliografia

[1] C. Perera, C. H. Liu, S. Jayawardena, and M. Chen, "A survey on internet of things from industrial market perspective," *IEEE Access*, vol. 2, pp. 1660–1679, 2014.

[2] E. Blem, J. Menon, and K. Sankaralingam, "Power struggles: Revisiting the risc vs. cisc debate on contemporary arm and x86 architectures," in *High Performance Computer Architecture (HPCA2013), 2013 IEEE 19th International Symposium on*, pp. 1–12, IEEE, 2013.

[3] I. A. S. L. P. D. VLSI Technology, *Acorn Risc Machine: Family Data Manual*. Arm Family Data Manual, Prentice Hall PTR, 1990.

[4] J. A. Langbridge, *Professional Embedded ARM Development*. Birmingham, UK, UK: Wrox Press Ltd., 1st ed., 2014.

[5] "Fastest micro in the world," *New Scientist*, vol. 114, no. 1565, pp. 41 –, 1987.

[6] D. Seal, *ARM architecture reference manual*. Pearson Education, 2001.

[7] "Arm information center - mode bits." `http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0290g/I27695.html`. (Accessed on 17/01/2016).

[8] T. Burd and R. Brodersen, *Energy Efficient Microprocessor Design*. Springer US, 2012.

[9] "Arm information center - thumb." http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0210c/CACBCAAE.html. (Accessed on 10/02/2016).

[10] "Datasheet arm7tdmi." http://infocenter.arm.com/help/topic/com.arm.doc.ddi0234b/DDI0234.pdf. (Accessed on 07/03/2016).

[11] "Arm information center - thumb2." http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0344c/Beiiegaf.html. (Accessed on 10/02/2016).

[12] M. Williams, "Armv8 debug and trace architectures," in *System, Software, SoC and Silicon Debug Conference (S4D), 2012*, pp. 1–6, IEEE, 2012.

[13] D. Merkel, "Docker: lightweight linux containers for consistent development and deployment," *Linux Journal*, vol. 2014, no. 239, p. 2, 2014.

[14] C. Boettiger, "An introduction to docker for reproducible research," *ACM SIGOPS Operating Systems Review*, vol. 49, no. 1, pp. 71–79, 2015.

[15] P. Li, "Selecting and using virtualization solutions: our experiences with vmware and virtualbox," *Journal of Computing Sciences in Colleges*, vol. 25, no. 3, pp. 11–17, 2010.

[16] F. Bellard, "Qemu, a fast and portable dynamic translator.," in *USENIX Annual Technical Conference, FREENIX Track*, pp. 41–46, 2005.

[17] "Docker docs." https://docs.docker.com/. (Accessed on 08/02/2016).

[18] "Manual - qemu." http://wiki.qemu.org/Manual. (Accessed on 12/02/2016).

[19] "Documentation - vagrant by hashicorp." `https://www.vagrantup.com/docs/`. (Accessed on 08/01/2016).

[20] A. Sloss, D. Symes, and C. Wright, *ARM system developer's guide: designing and optimizing system software*. Morgan Kaufmann, 2004.

[21] J. Goodacre and A. N. Sloss, "Parallelism and the arm instruction set architecture," *Computer*, vol. 38, no. 7, pp. 42–50, 2005.

Ringraziamenti

Ringrazio innanzitutto la mia famiglia che mi ha supportato in ogni maniera possibile in questo percorso di studi e nella realizzazione di tutti i miei sogni senza mai risparmiarsi.
Grazie anche ai docenti che mi hanno sostenuto e guidato nella crescita culturale e professionale, insegnandomi nuovi modi di pensare e comprendere il mondo.
Un ringraziamento speciale va ai miei amici sempre pronti a sopportare le mie lunghe assenze e i miei sproloqui in qualsiasi ambito.
Grazie infine ad Elena la mia roccia e centro di gravità che mi guida nell'orbita verso il futuro.