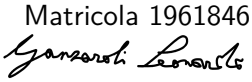
SAPIENZA
UNIVERSITÀ DI ROMA

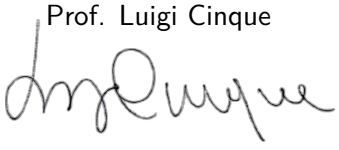
Emulazione di acceleratori hardware: un approccio basato su QEMU

Facoltà di Ingegneria dell'informazione, Informatica e Statistica
Corso di Laurea in Informatica

Leonardo Ganzaroli
Matricola 1961846



Responsabile
Prof. Luigi Cinque



Corresponsabile
Dr. Diego Bellani

Anno Accademico 2024/2025

Emulazione di acceleratori hardware: un approccio basato su QEMU
Relazione di tirocinio - Sapienza Università di Roma

Questa relazione è stata composta con LaTeX e la classe Sapthesis.

Email dell'autore: ganzaroli.leonardo@gmail.com

«Il cucchiaio non esiste?»
— Neo

Sommario

Le reti neurali vengono impiegate in modo sempre più esteso, ma il loro utilizzo comporta un costo computazionale considerevole. Per gestirlo si ricorre spesso agli acceleratori hardware. Lo sviluppo di questi acceleratori è tuttavia oneroso dal punto di vista temporale, ciò non è causato solamente dallo sviluppo dell'hardware e del software ma dipende anche da fattori come la complessità del metodo di interconnessione usato. Lo sviluppo del software stesso serba un ulteriore problema, può infatti cominciare soltanto quando lo sviluppo hardware è praticamente terminato.

Questo tirocinio esplora lo sviluppo della versione virtuale di un generico acceleratore hardware tramite opportuni strumenti di emulazione. L'obiettivo è valutare la possibilità di disaccoppiare, anche parzialmente, le fasi di sviluppo hardware e software. Nello specifico si vuole indagare sulla "fattibilità" di questo metodo, intesa come il grado di somiglianza tra la versione emulata e la sua controparte fisica. Particolare attenzione viene posta alle differenze a livello di comportamento funzionale, fedeltà del metodo di connessione e prestazioni (che potrebbero venire degradate dall'emulazione). Con i risultati ottenuti si vuole dunque comprendere se e in che misura l'emulazione possa essere usata come base per sviluppare il software parallelamente all'hardware, andando così a ridurre i tempi complessivi di progettazione.

Indice

Capitolo 1

Introduzione

Questo capitolo fornisce una panoramica sugli acceleratori hardware e sulla loro diffusione nella sezione 1.1, passa poi all'obiettivo finale del tirocinio con le relative motivazioni nella sezione 1.2.

1.1 Acceleratori hardware

Gli acceleratori hardware sono dei componenti hardware progettati appositamente per svolgere determinate funzioni, essi permettono di svolgere quelle funzioni in modo più efficiente rispetto alla classica combinazione di software e CPU. I possibili campi d'applicazione sono molteplici ma alcuni mostrano chiaramente il vantaggio di delegare i compiti, per esempio:

- In crittografia sono spesso richiesti dei calcoli particolarmente intensivi, evitare di svolgerli nella CPU comporta una notevole riduzione del carico sulla stessa.
- L'elaborazione grafica richiede molti calcoli che possono essere ottimizzati svolgendoli in parallelo, data la sua natura sequenziale la CPU non risulta la scelta migliore.
- L'elaborazione dei segnali in alcuni casi deve essere svolta in tempo reale e potrebbe richiedere delle operazioni aggiuntive, la CPU potrebbe non riuscire a gestire il tutto senza ritardi. L'acceleratore in figura 1.1 per esempio elaborava il vecchio segnale televisivo e permetteva quindi di guardare la TV direttamente da un computer.

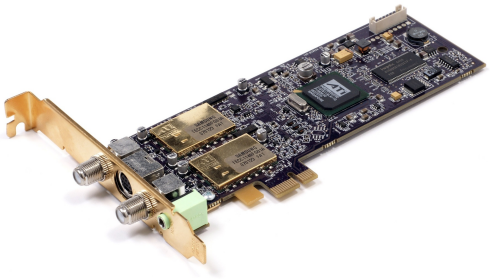


Figura 1.1. TV Wonder HD 650 di ATI Technologies

1.1.1 Evoluzione nel tempo

I primi acceleratori nacquero alla fine degli anni '70 / inizio anni '80 sotto forma di semplici coprocessori matematici atti a svolgere calcoli aritmetici complessi. Con l'avanzare del tempo e conseguentemente della tecnologia sono stati sviluppati acceleratori sempre più vari e potenti. Come si vedrà nella sezione 1.1.2 gli acceleratori più diffusi in questo periodo sono quelli che si occupano di grafica e quelli relativi all'IA. Per dare un'idea dell'evoluzione avvenuta seguono 3 esempi provenienti da anni diversi:

Intel 8087 (1980)

Si tratta del primo coprocessore per calcoli in virgola mobile progettato per i processori 8086 e 8088, il suo scopo principale è appunto quello di svolgere calcoli in virgola mobile tra cui i logaritmi e le varie funzioni trigonometriche [1].

Sun Crypto Accelerator 1000 (2002)

Lo scopo di questo acceleratore è quello di effettuare l'onerosa computazione di algoritmi crittografici usati in vari protocolli di sicurezza [2].

Pixel Visual Core (2017)

Presente negli smartphone Pixel 2 e Pixel 2 XL di Google, è un coprocessore che si occupa dell'elaborazione fotografica. Oltre al miglioramento delle foto permette di eseguire algoritmi di Machine Learning come il riconoscimento dei volti [3].

Con il tempo anche l'architettura generale degli elaboratori ha subito un'evoluzione, l'8087 aggiungeva 68 istruzioni al set dell'8086 ed era collegato in parallelo ad esso, quando arrivava una sua istruzione il coprocessore prendeva il controllo e ad operazione avvenuta mandava un segnale al processore per farlo ripartire [1].

Risulta evidente che questo metodo comporta uno spreco della CPU, infatti ogni volta che deve essere eseguita anche una semplice operazione come 0,1 + 0,2 il coprocessore (anche se per un tempo irrisorio) mantiene il controllo e non permette alla CPU di fare altro.

Considerando la complessità e la quantità di componenti dei sistemi odierni sarebbe impossibile continuare con questo metodo, per cercare di limitare problemi di questo tipo sono state create diverse soluzioni.

Nel caso gli acceleratori siano esterni al processore e collegati tramite bus si pone il problema dello scambio di dati, infatti nel caso questi siano molti bisogna continuamente fare richiesta alla CPU per poter usare il bus e trasferire un certo numero di dati creando di fatto una dipendenza. Per limitare il problema si può usare DMA, consiste nell'inserire un controllore che gestisca direttamente i trasferimenti e notifichi la CPU solo all'inizio ed al termine. Così facendo si va a fornire un certo grado di indipendenza all'acceleratore con conseguente possibilità di svolgere operazioni in parallelo alla CPU [4].

1.1.2 Popolarità

Come visto nella sezione precedente l'uso degli acceleratori porta evidenti vantaggi, questo potrebbe portare a domandarsi quanto siano diffusi e se possano aver contribuito allo sviluppo di altre aree tecnologiche.

Alla prima parte si può rispondere con "parecchio", nel 2023 il mercato globale degli acceleratori era valutato a 2,87 miliardi di dollari e si stima che nel 2033 raggiungerà 177 miliardi [5].

Alla seconda parte si può rispondere nel medesimo modo, prendendo come esempio gli acceleratori grafici si scopre che non solo si sono evoluti per soddisfare richieste sempre maggiori ma vengono anche usati per scopi diversi da quelli originali, rispettivamente:

- La continua ricerca da parte del mondo videoludico di una grafica sempre più realistica è stata fin'ora una delle motivazioni principali per lo sviluppo degli acceleratori grafici [6].
- Le GPU presenti negli acceleratori sono ormai lo strumento standard usato nella fase di addestramento delle reti neurali, un'altra applicazione non trascurabile è il mining delle criptovalute [7].

1.1.3 Metodi di interconnessione

Per concludere questa panoramica sugli acceleratori resta solamente da vedere il metodo di connessione fisico tra gli acceleratori e i sistemi a cui vanno collegati. Nella definizione vista nella sezione 1.1 non si accenna a questo concetto perché non esiste un metodo universale ma invece ce ne sono svariati, ognuno con i propri pro e contro, ma cosa più importante ci sono metodi più diffusi di altri. A seguire una breve descrizione di alcuni metodi [8]:

PCIe Approfondito nel capitolo 2, l'acceleratore nella figura 1.1 usa questo metodo.

Integrated on-chip Integrato direttamente all'interno di un System on a chip[1], per questo tipo di sistemi la comunicazione tra componenti interni avviene tramite delle connessioni specifiche ed i relativi protocolli. Un esempio sono gli acceleratori crittografici presenti nella famiglia di microcontrollori STM32L4 che usano le connessioni definite dallo standard "Advanced Microcontroller Bus Architecture" ed il protocollo "AHB" sempre definito nello stesso standard [9].

USB Anche se meno performante rispetto agli altri metodi viene comunque usato data la sua estrema diffusione, l'Intel Neural Compute Stick 2 rientra in questa categoria e si collega al PC semplicemente inserendolo in un connettore USB di tipo A [10].

I^2C Si tratta di un protocollo di comunicazione seriale usato ampiamente nel mondo embedded, usa un filo per i dati ed uno per il clock e presenta una struttura master-slave [11]. Questo metodo è usato dal coprocessore DS2465, esso si occupa di hashing, nel particolare di SHA-256 [12].

PCIe rimane uno dei metodi più usati [13], questo è dovuto sia al fatto che si tratta della naturale evoluzione di PCI (già famoso) sia alle sue ottime caratteristiche riguardo: retrocompatibilità, scalabilità, velocità di trasferimento, larghezza di banda e gestione degli errori [14].

1.2 Obiettivo del tirocinio

Per spiegare l'obiettivo di questo tirocinio va introdotto il principale problema riguardante gli acceleratori, il tempo. La progettazione ed il successivo sviluppo

[1] Circuito integrato che contiene alcuni/tutti gli elementi principali di un computer/sistema elettronico in un singolo pezzo di materiale semi-conduttivo.

richiedono mesi, se non addirittura anni, nel caso di un sistema di tipo ASIC (Application specific integrated circuit) ci vogliono mediamente dai 9 ai 18 mesi [15]. A tutto ciò va aggiunto il processo di sviluppo del lato software che può iniziare solamente quando è pronto almeno un prototipo del prodotto finale portando così ad ulteriori ritardi.

L'obiettivo finale è quello di capire quanto risulti complicato creare un acceleratore hardware virtuale e fino a quale grado di somiglianza con la controparte fisica si possa arrivare. Con il risultato ottenuto si cercherà di verificare se è possibile disaccoppiare lo sviluppo hardware da quello software andando così a ridurre i tempi di sviluppo complessivi. Verrano prodotti un prototipo di acceleratore con connessione PCIe che verrà poi emulato da QEMU insieme al sistema, il relativo driver per Linux ed una libreria utilizzabile dall'utente.

I capitoli 2, 3 e 4 forniscono le basi teoriche necessarie e danno una panoramica sugli strumenti usati, il capitolo 5 mostra gli elementi creati nel processo di implementazione ed il capitolo 6 tira le conclusioni sul tirocinio sostenuto.

Capitolo 2

PCIe

Questo capitolo fornisce una visione generale sullo standard PCIe, la sezione 2.1 è introduttiva mentre il resto del capitolo fornisce la teoria essenziale.

2.1 Concetti di base

PCIe (Peripheral Component Interconnect Express) è un'interfaccia di I/O ad alte prestazioni e ad uso generale, progettata per una vasta gamma di piattaforme informatiche e diretto successore di PCI e delle sue varianti PCI-X e CompactPCI. Mantiene alcune delle caratteristiche del suo predecessore ma cambia totalmente topologia [16].

È bene notare che questa interfaccia permette di connettere una vasta gamma di dispositivi tra cui antenne per la connessione Wi-Fi e schede di espansione che aggiungono porte fisiche. La connessione avviene tramite gli slot fisici comunemente presenti sulle schede madri dei computer fissi (visibili in figura 2.1), ma PCIe è usato anche in altri standard e interfacce che ne estendono il campo d'azione come:

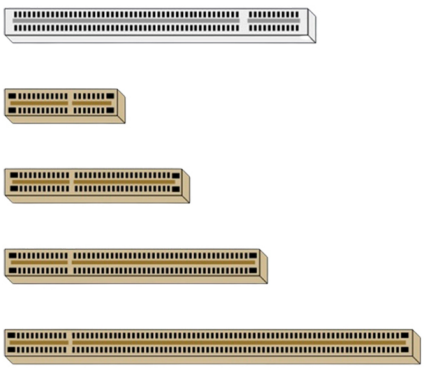


Figura 2.1. Rappresentazione degli slot fisici, dall'alto: PCI, PCIe (x1, x4, x8, x16)

- Le ExpressCard sono schede di espansione inseribili negli appositi slot presenti nei computer portatili, una delle sue versioni presenta un connettore PCIe (x1) che permette di collegare logicamente un dispositivo esterno alla topologia PCIe interna [17].

- Adattando PCI all'ambito industriale è nato PXI, l'evoluzione del primo in PCIe ha portato il secondo a diventare PXIe che mantiene la parte fisica di PXI ma prende quella logica ed elettrica di PCIe [18].

Prima di passare alle specifiche è opportuno aprire una parentesi sull'evoluzione di questo standard, dall'anno di creazione fino ad oggi si sono succedute ben 7 generazioni (con un ottava programmata) e l'elemento migliorato maggiormente è

il trasferimento di dati, passando da 2,5 GT/s[1] a 128 GT/s. A partire dalla sesta generazione c'è stato anche un cambio della codifica di linea[2], si è passati da NRZ[3] a PAM-4[4] + FEC[5] [16].

2.2 Topologia

Come accennato nella sezione precedente la topologia di PCIe differisce da quella del suo predecessore. PCI ha una struttura basata su bus paralleli mentre PCIe è seriale e presenta delle somiglianze con le reti di calcolatori. In particolare usa una forma di comunicazione a pacchetti ed è possibile dividere la sua architettura in 3 livelli logici che presentano delle similitudini con i livelli inferiori del modello OSI [16]:

Livello PCIe	**Descrizione del livello**
Transazione	Gestione dei pacchetti e instradamento
Data Link	Controllo integrità dei pacchetti e gestione dei link
Fisico	Trasmissione fisica dei dati

Livello OSI	**Descrizione**	**Livello PCIe corrisp.**
Trasporto	Gestione della comunicazione logica tra nodi finali	Transazione (parzialmente)
Rete	Instradamento dei pacchetti	
Data Link	Gestione della connessione tra dispositivi fisicamente connessi	Data Link
Fisico	Trasmissione fisica dei dati	Fisico

La somiglianza non si ferma ai soli livelli ed al tipo di comunicazione, infatti alcuni tra gli elementi di base presenti in questa topologia presentano somiglianze con elementi comunemente presenti nelle reti:

Lane Formata da 2 coppie differenziali[6], ossia 4 tracce (2 per trasmettere e 2 per ricevere).

Link Un semplice canale di comunicazione tra due componenti, deve contenere almeno una lane. Si può vedere come un classico collegamento con cavo di rame.

Root Complex Il punto di connessione principale tra CPU, memoria e gerarchia PCIe. Ha il compito di gestire la comunicazione tra i componenti collegati e generare le richieste R/W dirette ai dispositivi PCIe. Nei PC generalmente o viene integrato nel processore o si trova fisicamente sulla scheda madre all'interno del circuito integrato (northbridge) che connette la CPU ai bus ed ai dispositivi più veloci.

[1] T/s è un'unita di misura informale che indica il numero di operazioni di trasferimento dati effettuate al secondo.

[2] Processo di adattamento di un segnale al mezzo di comunicazione usato.

[3] Codifica di linea in cui i valori 0 e 1 vengono rappresentati da due livelli distinti di tensione senza ritorno allo stato neutro tra un bit e l'altro [19].

[4] Modulazione in cui l'ampiezza della portante (impulsiva) varia in base in base all'ampiezza del segnale modulante [19], PCIe usa la versione a 4 livelli.

[5] Tecnica usata per rilevare e correggere errori nelle comunicazioni digitali.

[6] Formata da 2 tracce affiancate, esse portano un segnale con stessa ampiezza ma polarità opposta [20].

Endpoint Ogni singola funzione fornita da ogni dispositivo fisicamente connesso alla gerarchia è un endpoint, si distinguono in:

Legacy Compatibili con le vecchie specifiche PCI ed il relativo software legacy. Oggigiorno rientrano in questa categoria anche molti dei dispositivi creati durante le prime generazioni di PCIe che usano funzionalità cadute in disuso come gli interrupt `INTx`.

Nativo PCIe Progettati esclusivamente per PCIe, nessuna retrocompatibilità con PCI. Praticamente tutti i dispositivi moderni che usano PCIe.

Root Complex Integrated Endpoint Integrati all'interno del Root Complex, si comportano come quelli nativi ma non presentano una connessione fisica tramite link. Devono rispettare alcune regole aggiuntive come implementare necessariamente la capacità "MSI" (o la sua versione estesa) e non implementare i sotto-registri della capacità "Express" relativi ai link [16].

A livello generale sono paragonabili agli endpoint presenti nelle reti (PC, server, stampanti, ecc. . .).

Switch Insieme di bridge PCI-to-PCI virtuali, permette di collegare tra loro i vari componenti. In alcune schede madri sono presenti vicino agli slot PCIe. Nelle schede di espansione che aggiungono altri slot PCIe sono generalmente integrati nella stessa.

Bridge PCIe to PCI/PCI-X Permette di collegare una gerarchia di dispositivi PCI/PCI-X a quella PCIe.

Ripetitore Ritrasmette ad una potenza maggiore il segnale ricevuto in ingresso, permette così di estendere la distanza di comunicazione.

Un esempio di topologia è visibile nella figura 2.2, inoltre il link preso in esame nella stessa è composto da 4 lane, esso viene definito x4 ed è associato all'omonimo slot presente in figura 2.1. Secondo lo standard non sono supportati numeri di lane arbitrari ma solamente: 1, 2, 4, 8, 12, 16 e 32 [16].

2.3 Link Training ed enumerazione

L'inizializzazione dei link (Link Training, compresa nella figura 2.3) è una procedura che coinvolge tutti gli elementi della gerarchia, si verifica quando tutti i dispositivi sono alimentati ed hanno ricevuto un clock di riferimento [21]. Si può dividere in 4 fasi:

Detect Ogni dispositivo attiva su ogni sua lane un circuito specifico che gli permette di capire se è collegato ad un altro dispositivo, in caso affermativo inizia ad inviare dati.

Polling Il Root Complex e gli endpoint trasmettono un insieme di dati predefiniti, questa parte termina quando ogni dispositivo riesce ad interpretare i dati ricevuti e rispondere di conseguenza.

Configuration Viene stabilita la grandezza dei link, si assegna un numero sia ai link che alle lane ed avviene l'associazione delle seconde ai primi.

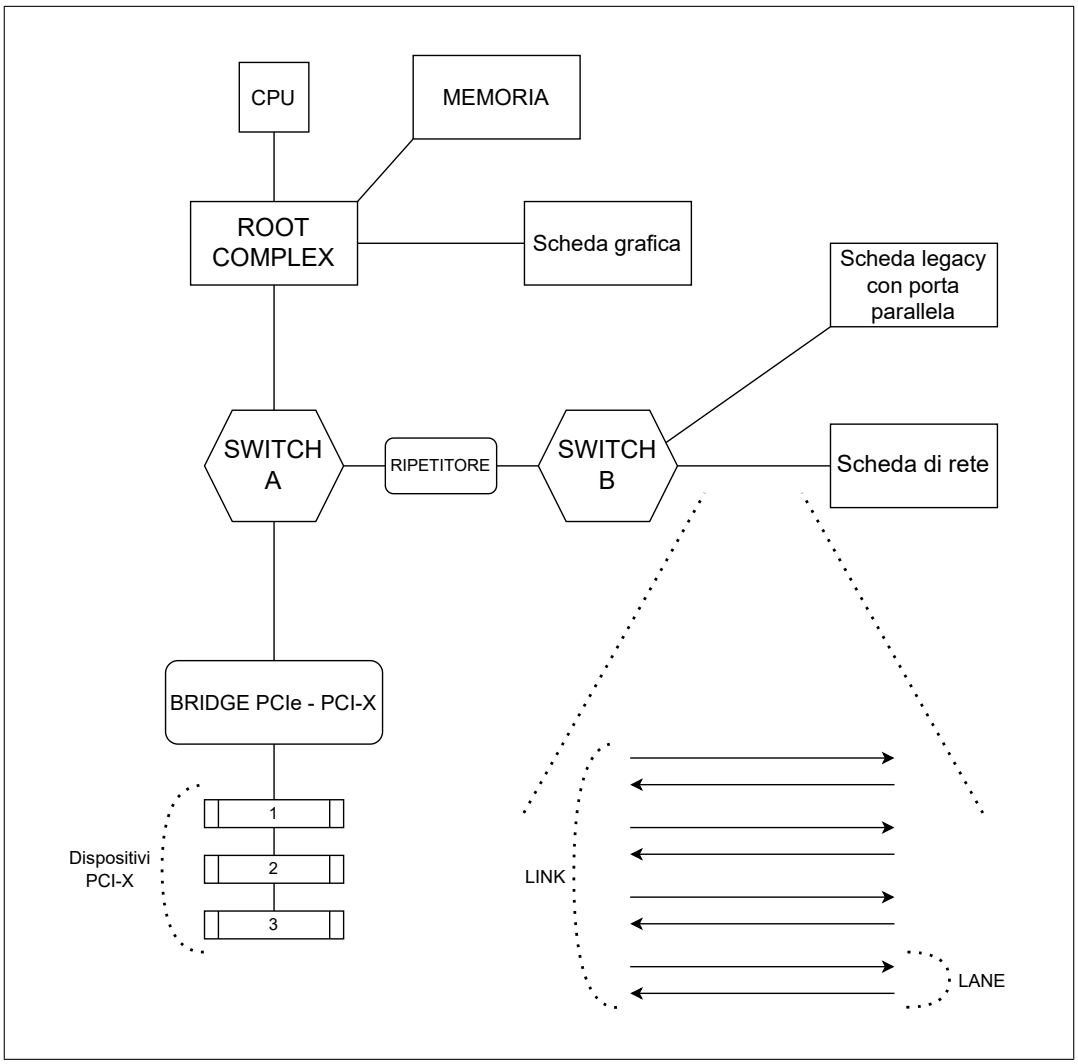


Figura 2.2. Esempio di topologia

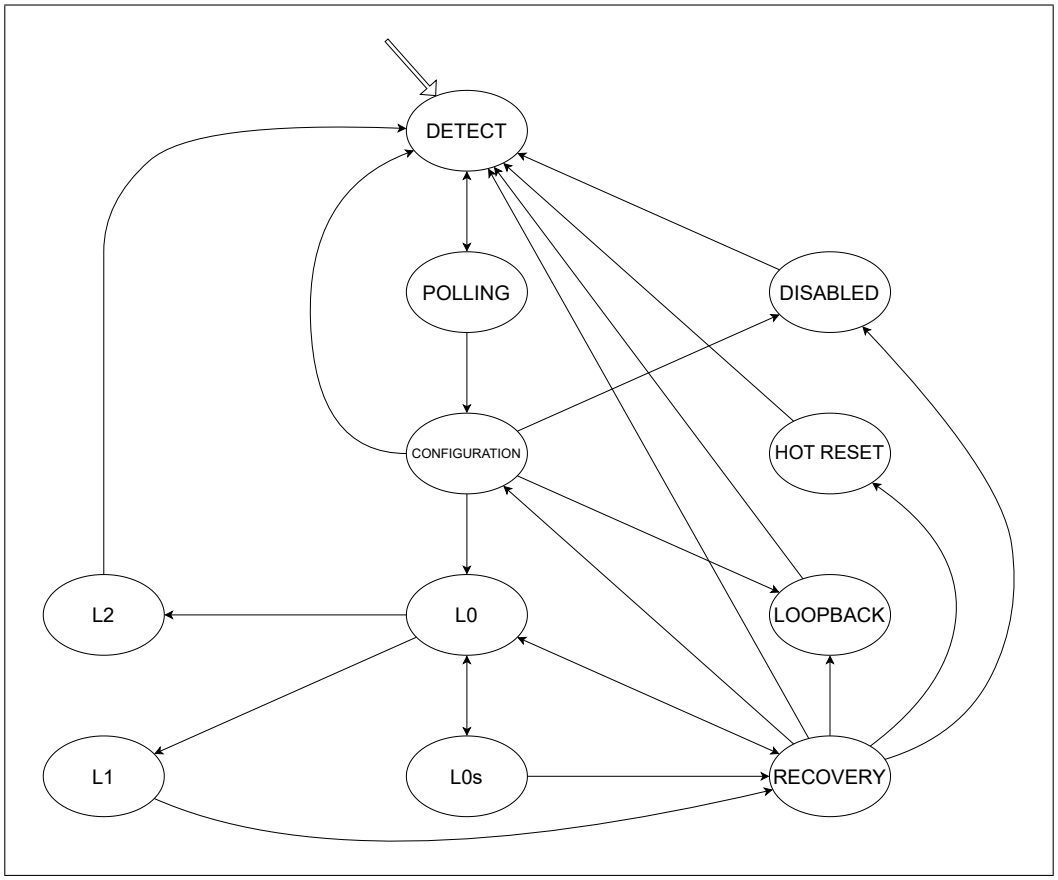


Figura 2.3. Link Training and Status State Machine

Link Equalization Opzionale (per questo non presente nella figura 2.3), eseguita solo se tutti i dispositivi supportano almeno la velocità standard della 3° generazione. Tramite dei preset si cerca la più alta velocità trasmissiva possibile senza sacrificare la qualità del segnale trasmesso.

Nella figura 2.3 sono presenti anche i vari stati in cui si possono trovare i link dopo la fine del processo visto prima, nel dettaglio:

L0 rappresenta il normale funzionamento.

L0s, L1 e L2 sono associati a diversi livelli di risparmio energetico.

Loopback è usato per la diagnostica.

Disable indica che il link non è abilitato.

Hot Reset serve per eseguire un reset logico.

Recovery fa ripartire il processo di training in caso di errori.

Immediatamente dopo il Link Training avviene l'enumerazione dei dispositivi, ad ognuno viene assegnato un codice identificativo con forma:

Dominio	N. Bus	N. Dispositivo	N. Funzione
2 byte	1 byte	5 bit	3 bit

Il dominio ha una scopo solo in presenza di Root Complex multipli, permette infatti di distinguere le varie gerarchie presenti nel sistema e risulta superfluo se ne è presente solo una. Il numero di bus viene fornito da una divisione in bus logici della topologia, così facendo è possibile semplificare l'instradamento dei pacchetti ed allo stesso tempo avere retrocompatibilità con PCI. Il numero di dispositivo serve a distinguere i vari endpoint presenti in un bus ed il numero di funzione è presente per permettere ad un dispositivo di esporre più funzioni logiche distinte [16].

Questo processo avviene in modo semplice, vedendo il tutto come se fosse un grafo basta svolgere una visita in profondità partendo dal Root Complex e tenendo nota del livello in cui ci si trova.

Per la figura 2.2 una possibile enumerazione, ipotizzando la presenza di funzioni multiple, potrebbe essere:

Elemento	Dominio	N. Bus	N. Dispositivo	N. Funzione
Root Complex	0000	0	0	0
Scheda grafica	0000	0	1	0
Scheda grafica (2°)	0000	0	1	1
Scheda grafica (3°)	0000	0	1	2
Switch A	0000	0	2	0
Switch B	0000	1	0	0
Bridge	0000	1	1	0
Scheda di rete	0000	2	0	0
Scheda di rete (2°)	0000	2	0	1
Scheda legacy	0000	2	1	0
Disp. PCI-X 1	0000	3	0	0
Disp. PCI-X 2	0000	3	1	0
Disp. PCI-X 3	0000	3	2	0

2.4 Configurazione tramite software

Il procedimento visto nella sezione precedente permette quindi di inizializzare i vari link e scoprire gli elementi presenti nella gerarchia, tuttavia, questo non è ancora sufficiente. Per essere utilizzato ogni dispositivo deve esporre le informazioni necessarie per la sua configurazione, la sua inizializzazione e l'associazione al giusto driver. Per fare ciò deve implementare dei registri, ossia delle zone di memoria interne ma comunque accessibili dall'esterno, che contengano tutto il necessario. Il loro insieme viene definito spazio di configurazione, nel caso PCIe questo spazio equivale a 4096 Byte e va ad estendere il vecchio spazio PCI (visibile in figura 2.4). A differenza di PCI questo spazio esteso viene mappato direttamente nella memoria e permette di accedere ai singoli registri tramite opportune API fornite dal sistema operativo [16]. Ogni funzione di ogni dispositivo ne ha uno.

Resta comunque disponibile il vecchio metodo di accesso usato da PCI che permette di accedere solo ai primi 256 Byte, ossia allo spazio non esteso. Come visto nella figura 2.4 le sezioni principali sono le 2 liste delle capacità e l'header PCI[7], quest'ultimo è quello che contiene tutte le informazioni sul dispositivo.

Sono presenti 2 tipi di header, il tipo 0 è quello di nostro interesse e viene usato per gli endpoint, è visibile nella tabella 2.1, mentre il tipo 1 è per gli elementi di interconnessione come gli switch. Tra i 2 tipi ci sono comunque molti registri

[7]Alcuni bit e registri vengono impostati read-only e con valore 0 perché associati a funzionalità obsolete non più supportate in PCIe [16], visibile nella pagina successiva.

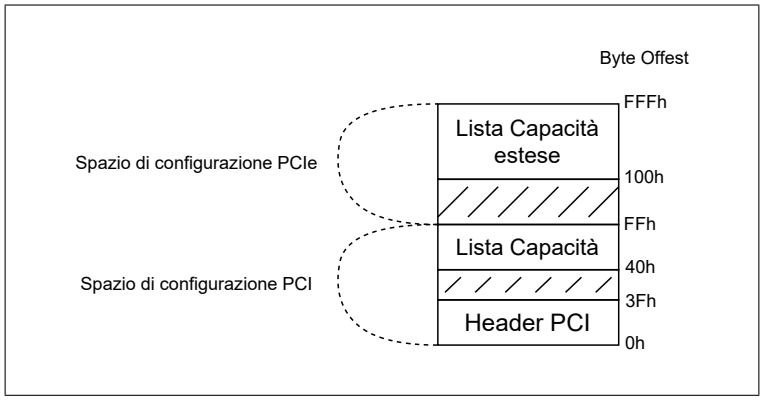


Figura 2.4. Layout dello spazio

31-24	23-16	15-8	7-0	Bit
∗ ID Dispositivo		∗ ID Venditore		
∗ Stato		∗ Comando		
∗ Codice di Classe			∗ ID Revisione	
∗ BIST	∗ Tipo di header	∗ Timer di latenza	∗ Dimensione linea Cache	
∗ BARs				
Puntatore a CIS per CardBus				
ID Sottosistema		ID Venditore Sottosistema		
Indirizzo base ROM di espansione				
			∗ Puntatore alle capacità	
Latenza massima	Tempo minimo di accesso al bus	∗ Pin di interrupt	∗ Linea di interrupt	

Tabella 2.1. Header PCI di tipo 0

in comune, indicati con ∗ nella tabella 2.1, ma la maggior parte differiscono e contengono informazioni che risultano specifiche al tipo di funzione comunemente svolta dal dispositivo associato all'header.

Breve descrizione dei registri[8] presenti nell'header di tipo 0:

I vari campi ID Identificano nel dettaglio il dispositivo.

Comando Controlla varie funzionalità del dispositivo.

Stato Contiene informazioni sullo stato del dispositivo e su eventuali errori.

Codice di Classe Contiene Classe, Sottoclasse e Interfaccia del dispositivo, insieme identificano la funzione associata allo spazio.

BIST (Built-in Self Test) Usato per auto-diagnostica interna.

Tipo di header Indica il tipo di header e se il dispositivo è multifunzione.

[8] I registri sono sempre little-endian [22].

Bit	0-7	8-15	16-19	20-31	...
Normali	ID	Puntatore nodo successivo	...		
Estesi	ID		Versione	Offset capacità successiva	...

Struttura dei nodi delle liste di capacità[9]

Timer di latenza e Puntatore a CIS per CardBus Obsoleti.

Dimensione linea Cache Presente solo per retrocompatibilità.

Indirizzo base ROM di espansione Gestisce un'eventuale ROM di espansione presente sul dispositivo, essa contiene del codice che deve essere copiato in RAM e poi eseguito. Questo codice normalmente o permette di usare il dispositivo durante il boot (prima del caricamento dei driver) oppure serve per il corretto avvio del dispositivo [23, 24].

Puntatore alle capacità Punta alla linked list delle capacità.

Latenza massima e tempo minimo di accesso al bus Obsoleti.

Pin e Linea di interrupt Usati solo per interrupt legacy.

BARs Indicano al sistema operativo quante regioni di memoria e/o I/O (con relativa dimensione) devono essere allocate in memoria per consentire il corretto funzionamento del dispositivo. L'header di tipo 0 può contenerne al massimo 6, quello di tipo 1 al massimo 2 [16].

2.4.1 Capabilites

Le capacità (capabilities) sono delle estensioni che permettono di aggiungere funzionalità ad un dispositivo PCI senza andare a modificare la struttura base dello spazio di configurazione. Sono organizzate in una lista che generalmente parte dall'offset visto nella figura 2.4, ogni nodo contiene sempre un ID ed un puntatore al nodo successivo e in alcuni casi anche dei registri aggiuntivi [25, 26]. Quelle estese invece sono prerogativa dei dispositivi PCIe, hanno struttura simile a quelle normali ma offrono generalmente delle funzionalità più complesse [16], la loro lista parte dall'offset visto nella figura 2.4.

In ogni caso i dispositivi PCIe devono sempre avere le capacità "Power Management" ed "Express" [16].

Tra tutte quelle disponibili ne spiccano alcune: "Express" indica che il dispositivo è PCI express, "Power Management" fornisce un'interfaccia con cui è possibile controllare il consumo elettrico del dispositivo [26], "MSI" (e la sua estensione MSI-X) permette al dispositivo di generare un interrupt scrivendo un certo valore nell'indirizzo fornito dal sistema operativo [26]. Anche per quelle estese vale lo stesso concetto: "Advanced Error Reporting" aggiunge un controllo degli errori avanzato, "Device Serial Number" assegna al dispositivo un numero di serie univoco, "Resizable BAR" consente la negoziazione tra dispositivo e sistema operativo riguardo la dimensione della memoria mappata.

[9] Entrambi i tipi di nodi devono essere allineati a 4 byte, per quelli normali si fa ciò settando i 2 bit meno significativi di ogni puntatore a 0 [16, 26].

Capitolo 3

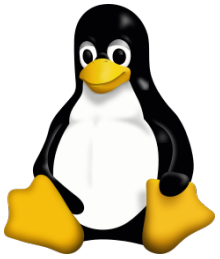
Linux

Questo capitolo fornisce una panoramica sul kernel Linux e in particolare su come gestisce i driver.

3.1 Panoramica

Con "Linux" normalmente si intende, erroneamente, un sistema operativo completo, ma in realtà si tratta di un kernel, ossia un componente software che sta alla base di un sistema operativo e si occupa di gestire l'hardware e fornire delle funzioni software basilari. I principali sistemi operativi costruiti su questo kernel sono detti distribuzioni (o distro) e sono svariate ma ci sono molti altri sistemi operativi, come Android, che usano lo stesso kernel ma hanno una struttura totalmente diversa da esse [27, 28]. Un ulteriore caso d'uso differente si può trovare in ambito embedded dove a causa delle limitate risorse viene spesso usato direttamente il kernel in combinazione con pochi programmi specifici.

Android in particolare viene ormai usato in una vasta gamma di dispositivi: Smartphone, Tablet, Smartwatch, automobili e Smart TV. Non sono da meno le varie distribuzioni, alcune presentano una grande flessibilità e in alcuni casi un'estrema leggerezza che ne permette l'uso in apparecchiature di rete come server e router ma anche nei PC, in particolare in quelli usati per la programmazione o con qualche anno alle spalle [27]. Nel penultimo caso in particolare entrano spesso in gioco alcuni degli strumenti prodotti per il progetto GNU, molte distribuzioni infatti li includono propriamente o li integrano con altri componenti. Alcuni di questi strumenti sono: la shell Bash, l'ambiente desktop Gnome, la famiglia di compilatori gcc ed il debugger gdb.



Tux, la mascotte di Linux

3.2 Gestione dei driver

Il metodo principale per implementare i driver in Linux è attraverso i moduli, essi sono componenti software che forniscono funzionalità aggiuntive al kernel. Possono essere aggiunti staticamente durante l'avvio del kernel oppure dinamicamente a runtime. Nel secondo caso l'inserimento e la rimozione avvengono rispettivamente mediante i comandi `insmod` e `rmmod`, il file da passare come argomento a questi

comandi deve avere estensione `ko` ed è il risultato di un tipo di compilazione specifica dipendente anche dalla versione del kernel. Si possono dividere in 3 classi basandosi sui tipi di dispositivi a cui sono associati [22]:

A Caratteri Dispositivi che lavorano con flussi sequenziali di dati, l'accesso è diretto senza la presenza di buffer intermedi. Rientrano in questa categoria: porte seriali, porte parallele e terminali ma anche dispositivi senza controparte fisica come `/dev/zero` e `/dev/random`.

A blocchi Dispositivi che gestiscono i loro dati organizzandoli in blocchi di dimensione fissa e permettono operazioni I/O ad accesso casuale (bufferizzate ed astratte) su di essi, principalmente sono i dispositivi di archiviazione.

Di rete Dispositivi che permettono la comunicazione con altri sistemi, generalmente usano una comunicazione a pacchetti (Ethernet, Wi-Fi, Bluetooth, NFC e simili).

Questa suddivisione ha ancora più importanza se avviene la creazione di un dispositivo virtuale (associato a quello fisico, presente nella cartella `/dev`) dato che questi presentano la medesima classificazione. L'utente può interagirci con le classiche operazioni eseguibili su un generico file (`open, close, read, write, ecc...`) a patto che siano implementate nel driver associato. Oltre al loro "tipo" i dispositivi virtuali hanno 2 numeri identificativi associati (`Major` e `Minor`) e sta sempre al driver gestirli.

Nel caso il driver sia per un dispositivo collegato tramite bus si troveranno sempre 3 funzioni e dei metadati:

Init() Eseguita quando il modulo viene caricato, per fare ciò va passata alla macro `module_init`. Il suo compito principale è quello di registrare il driver come tale ed eventualmente specificare con quali tipi di dispositivi virtuali può funzionare.

Exit() Eseguita quando il modulo viene rimosso, rimuove il driver da quelli registrati. Va passata alla macro `module_exit`.

Probe() Eseguita al rilevamento di un dispositivo associato al driver, nel caso sia per un dispositivo PCI presenta la firma `static int probe(struct pci_dev *, const struct pci_device_id *)`.

Metadati Autore, descrizione e licenza (quest'ultima sempre obbligatoria). Definite rispettivamente con le macro: `MODULE_AUTHOR`, `MODULE_DESCRIPTION` e `MODULE_LICENSE`.

A tutto ciò vanno aggiunte le eventuali implementazioni delle operazioni già citate.

3.2.1 PCI(e)

C'è bisogno di un approfondimento riguardo i moduli che fanno da driver per i dispositivi PCIe (e PCI), dato che è il metodo di collegamento usato dal prototipo. Il linguaggio usato per sviluppare i moduli è il C e le librerie necessarie per trasformarli in driver specifici sono presenti nella cartella `/usr/include/linux`.

Durante l'inizializzazione del kernel vengono avviati vari sottosistemi, tra questi sono presenti l'implementazione della specifica ACPI, che definisce come un sistema operativo debba interfacciarsi con il firmware riguardo la gestione dell'energia e

la configurazione dell'hardware [29], ed il sottosistema PCI (che funziona anche con la topologia PCIe come visto nella sezione 2.3). La collaborazione tra questi 2 sottosistemi permette di svolgere l'enumerazione, creare i vari spazi di configurazione ed associare ad ogni dispositivo trovato il giusto driver. Per fare ciò ogni driver deve fornire una lista dei dispositivi che supporta, così facendo si può cercare una corrispondenza con i registri ID (sezione 2.4) dei dispositivi[1] [30].

La lista viene implementata tramite un array ed ogni suo elemento è del tipo `pci_device_id`, esso è ottenibile dalle macro:

- `PCI_DEVICE(ID_Venditore, ID_Dispositivo)` se si vuole indicare uno specifico dispositivo
- `PCI_DEVICE_CLASS(Codice_di_Classe, Maschera_Classe)` se si vuole indicare un'intera classe di dispositivi

Una volta completa si procede ad esportare la lista nella sua interezza con la macro `MODULE_DEVICE_TABLE(pci, array)` [22].

Per registrare il driver invece c'è bisogno di una struct di tipo `pci_driver` che contenga almeno: un nome, l'array visto prima e i puntatori alle funzioni `Probe()` e `Remove()`.

Questa struttura viene usata dalle funzioni `Init()` ed `Exit()` per registrare/cancellare il modulo come driver [22], nel caso non si debbano svolgere ulteriori operazioni si può usare la funzione wrapper `module_pci_driver`.

Struttura minimale del driver:

```
static struct pci_device_id ids[ ] = {
    { PCI_DEVICE(ID_venditore, ID_dispositivo) },
    { 0 },
};

MODULE_DEVICE_TABLE(pci, ids);

static struct pci_driver driver = {
    .name = "nome_driver",
    .id_table = ids,
    .probe = funzione_probe,
    .remove = funzione_remove,
};

module_pci_driver(driver);
```

3.3 Strumenti e nozioni utili

Per facilitare l'uso dei dispositivi PCI e PCIe è normalmente presente il pacchetto pciutils, esso contiene 2 comandi eseguibili da terminale estremamente importanti:

lspci che stampa una lista di tutti i dispositivi rilevati, con alcune opzioni vengono descritti in modo dettagliato.

setpci che permette di leggere e modificare i singoli registri dell'header.

[1] Questo metodo permette l'associazione anche se il driver per un certo dispositivo viene caricato successivamente all'enumerazione iniziale.

Esempio d'uso:

```
$ lspci

    00:00.0 Host bridge
    00:14.0 USB controller
    01:00.0 SD Host controller
    ...

$ lspci -vv -s 0000:01:00.0

    01:00.0 SD Host controller: ...
    Subsystem: ...
    Control: I/O- Mem+ BusMaster+ SpecCycle- ...
    Status: Cap+ 66MHz- UDF- FastB2B- ParErr- ...
    ...

$ setpci -s 0000:01:00.0 BASE_ADDRESS_0

    a1301000
```

In ambienti minimali non è assicurato che questo pacchetto sia presente, l'alternativa è accedere direttamente allo spazio di configurazione tramite filesystem. Andando in `/sys/bus/pci/devices` è possibile trovare una cartella per ogni dispositivo, chiamata con l'indirizzo dello stesso. Al suo interno è presente il file denominato config che corrisponde allo spazio di configurazione, è possibile aprirlo ed interagirci come un generico file, si può quindi leggere con i vari linguaggi di programmazione.

Esempio minimale in C e in Perl, apertura del file, lettura (4 byte) dell'indirizzo presente nel registro BAR0 usando il suo offset (0x10) e stampa del valore letto a schermo:

```
int fd = open("/sys/bus/pci/devices/0000:01:00.0/config",
           O_RDONLY);
uint32_t val;
pread(fd, &val, 4, 0x10);
printf("%08x", val);
```

```
open F, "<", "/sys/bus/pci/devices/0000:01:00.0/config";
seek F, 0x10, SEEK_SET;
read F, $d, 4;
printf "%08x\n", unpack("V", $d);
```

Entrambi stampano `a1301000`.

Capitolo 4

Emulazione

Questo capitolo fornisce una panoramica su QEMU e su come può essere usato per creare ed emulare dispositivi custom.

4.1 Nozioni fondamentali

Prima di proseguire è bene definire alcuni concetti per evitare fraintendimenti, in questo specifico contesto distinguiamo:

Virtualizzazione Far credere ad un programma o ad un intero sistema operativo di avere il totale controllo della macchina, questo è possibile astraendo gli effettivi componenti hardware del sistema ospitante in modo da creare dell'hardware virtuale. Con questa tecnica le singole istruzioni macchina (del programma/sistema operativo in questione) vengono comunque eseguite inalterate dal processore. Un esempio di virtualizzatore è VirtualBox.

Emulatore Programma che riproduce il funzionamento di un altro computer, concettualmente è come una macchina di Turing universale che ne esegue un'altra. La differenza con un virtualizzatore sta nel fatto che in questo caso l'hardware presentato al sistema emulato è creato totalmente via software. QEMU è uno di questi ma ne esistono svariati ed alcuni tra questi non emulano un classico PC, per esempio PCSX2 (PlayStation 2) e DeSmuME (Nintendo DS).

Simulatore Programma che può riprodurre sistemi in molto estremamente dettagliato, a livello logico o analogico. Due esempi sono Verilator, che simula al livello dei componenti logici, e ngspice, che simula circuiti elettronici a livello analogico.

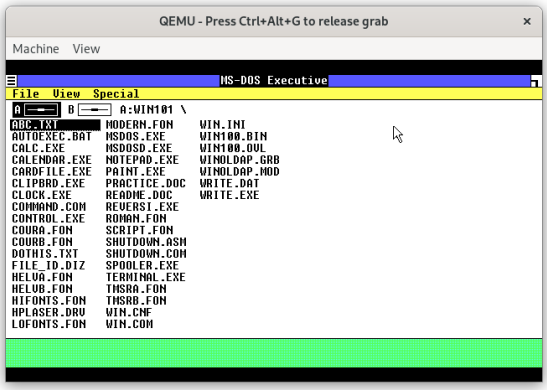
4.2 QEMU

Come visto nella sezione precedente QEMU è un emulatore, presenta diverse modalità di utilizzo ma le 2 principali sono l'emulazione di sistema che consente di simulare un'intera macchina comprendente CPU, memoria, ecc... e l'emulazione in modalità utente per eseguire processi singoli. Nel primo caso c'è anche la possibilità di usare un ipervisore[1] in modo che il sistema emulato sia eseguito direttamente sulla CPU fisica e non su quella emulata, questo però solo nel caso in cui il sistema emulato e

[1] Componente che crea e gestisce le macchine virtuali, il modulo KVM è quello che permette al kernel Linux di funzionare come tale.

quello ospitante abbiano la stessa architettura. Infatti QEMU permette di emulare anche architetture diverse da quella nativa del sistema dove viene eseguito [31].

L'emulazione (con o senza la virtualizzazione a supporto) risulta essere utile in numerosi ambiti. Ad esempio simulando sistemi datati si possono rendere accessibili programmi legacy al pubblico andando così a preservarli [32], oppure creando un ambiente controllato diventa possibile studiare i malware senza che arrechino danni al sistema [33] e con lo stesso criterio si possono emulare sistemi embedded avendo modo di simulare guasti fisici agli stessi evitando rischi reali [34].



Emulazione di Windows 1.0 con QEMU

4.2.1 Emulazione di dispositivi

QEMU permette di emulare un gran numero di dispositivi, nel nostro caso ci concentreremo su quelli basati su PCIe ma il discorso è pressoché identico per gli altri tipi.

I dispositivi vengono gestiti con un paradigma ad oggetti, per inserire una generica istanza di dispositivo nel sistema emulato deve essere presente nel codice sorgente di QEMU un file che lo descriva (nel linguaggio C) e vanno poi aggiunte le opportune opzioni al comando usato per far partire l'emulazione.

L'opzione da aggiungere è `-device nome_dispositivo,bus=numero_bus`, ne va aggiunta una per dispositivo.

Il file si può dividere in frontend e backend, il primo descrive come si presenta il dispositivo al sistema e per scriverlo bisogna attenersi al modello QOM (QEMU Object Model) mentre il secondo definisce cosa fa il dispositivo [31].

Esempio minimale di file, dispositivo PCI:

```
typedef struct PCIDevState {
    PCIDevice parent_obj; MemoryRegion bar0; uint64_t b0;
} ExamplePCIDev;

uint64_t b0_read(void *ptr, hwaddr addr, unsigned size) {
    return ((ExamplePCIDev *) ptr)->b0;}

void b0_write(void *ptr, hwaddr addr, uint64_t val, unsigned size) {
    ((ExamplePCIDev *) ptr)->b0 = val;}

const MemoryRegionOps b0_ops = {
    .read = b0_read, .write = b0_write,
    .endianness = DEVICE_NATIVE_ENDIAN,
    .valid = {.min_access_size = 8, .max_access_size = 8,},
    .impl = {.min_access_size = 8, .max_access_size = 8,},};
```

```
void pcidev_realize(PCIDevice *pdev, Error **errp) {
    ExamplePCIDev *s = (ExamplePCIDev *)pdev;
    memory_region_init_io(&s->bar0, OBJECT(s), &b0_ops, s, "b0", 8);
    pci_register_bar(pdev, 0, 0, &s->bar0);}

void pcidev_class_init(ObjectClass *class, void *data) {
    PCIDeviceClass *k = PCI_DEVICE_CLASS(class);
    k->realize = pcidev_realize;
    k->vendor_id = PCI_VENDOR_ID_QEMU;
    k->device_id = 0x1234;
    k->class_id = PCI_CLASS_OTHERS;}

void pcidev_register_types(void) {
    static const TypeInfo pcidev_info = {
        .name = "esempio_dispositivo_pci",
        .parent = TYPE_PCI_DEVICE,
        .instance_size = sizeof(ExamplePCIDev),
        .class_init = pcidev_class_init,};
    type_register_static(&pcidev_info);}
type_init(pcidev_register_types)
```

4.2.2 Limitazioni

Come visto nella sezione 4.1 usare QEMU comporta un'assenza di hardware tangibile nel contesto del sistema emulato, questo non solo rallenta l'esecuzione a causa del sovraccarico introdotto per creare e gestire l'hardware tramite software, che peggiora ulteriormente se il sistema emulato ha un'architettura differente rispetto al sistema ospitante, ma crea anche dei problemi [35].

Quelli di nostro interesse sono sostanzialmente 2 e comportano una notevole perdita di realismo, il primo è che i vari stati di risparmio energetico diventano solamente simbolici mentre il secondo che alcuni eventi normalmente presenti in un sistema non si presentano mai nell'ambiente emulato. Un esempio sono i ritardi che si possono presentare durante la lettura/scrittura dei dati, infatti senza collegamenti fisici non si presentano mai né la congestione della rete né altri problemi che richiedono la ritrasmissione dei pacchetti. Per gli stessi motivi diventano pressoché inutili anche alcune delle capacità estese come "Latency Tolerance Reporting"[2] e "L1 PM Substates"[3]. Quest'ultima può anche essere presa come esempio per evidenziare un'altra mancanza, infatti alcune delle capacità PCIe sono sì presenti in QEMU ma solamente come "segnaposto", ovvero sono presenti nel codice sorgente solo per completezza e all'atto pratico non posseggono né funzioni di supporto né un vero comportamento [36].

Questo implica che se si sta sviluppando un dispositivo ad-hoc non è assicurata la presenza di librerie già pronte per ogni funzionalità ma potrebbe presentarsi la necessità di svilupparle da zero.

[2] Definisce una latenza massima tollerabile rispetto alle operazioni di lettura e scrittura.
[3] Divide lo stato L1 visto nella figura 2.3 in 2 sottostati distinti.

Capitolo 5

Implementazione

In questo capitolo vengono presi in esame i singoli elementi del prototipo creato ed i test effettuati su di esso.

5.1 Prototipo

5.1.1 Acceleratore

L'acceleratore creato è molto semplice e prende ispirazione da una generica scheda grafica. In figura 5.1 è possibile vedere a grandi linee il funzionamento, i 3 BAR presenti hanno le seguenti funzioni: il primo (4 byte) viene usato per indicare la funzione da eseguire e farla partire, il secondo (8 byte) contiene un puntatore al buffer in cui è presente la matrice su cui operare (allocato dal driver)[1] e l'ultimo (256 byte) viene usato per passare degli argomenti alla funzione scelta. Per dichiarare l'uso di PCIe come metodo di connessione si è proceduto in modo simile all'esempio visto nella sezione 4.2.1, aggiungendo inoltre: le capacità necessarie viste nella sezione 2.4.1, "MSI" e "Device Serial Number".

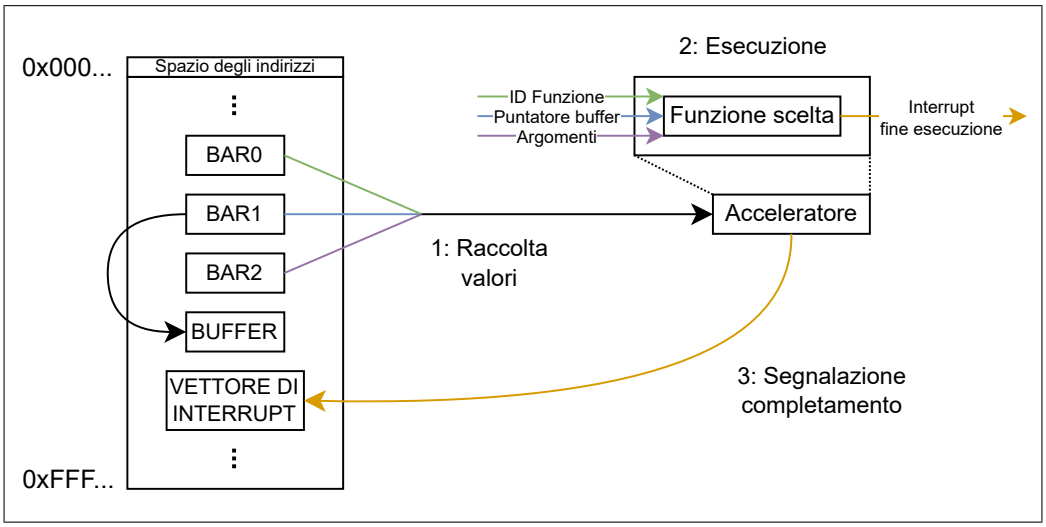


Figura 5.1. Svolgimento di un'operazione

[1] L'indirizzo fornito dal driver non può essere usato direttamente, viene chiesto a QEMU di "tradurlo" con la funzione `cpu_physical_memory_map`. Si ottiene così un puntatore valido che permette all'acceleratore di accedere direttamente al buffer (presente nella memoria emulata).

5.1.2 Driver

La creazione del driver e del dispositivo virtuale associato sono avvenute dopo aver correttamente immesso l'acceleratore nel sistema emulato. Per creare il driver stesso, come visto nelle sezioni 3.2 e 3.2.1, si parte da un modulo e si aggiungono i giusti elementi. Si può scomporre in 3 parti:

Gestione del driver che contiene le funzioni necessarie per registrare/rimuovere il driver stesso, quelle per creare e gestire il buffer condiviso e quelle necessarie per il dispositivo virtuale associato.

Gestione dell'acceleratore dove avviene l'inizializzazione dello stesso, la creazione delle risorse necessarie (come l'allocazione di memoria per i BAR) e la gestione degli interrupt.

Operazioni per l'utente in cui ci sono tutte le operazioni necessarie all'utente come quella che fornisce un puntatore al buffer (`mmap`) e quella che permette di interagire con l'acceleratore (`ioctl`).

N.B. In questo caso si ha buffer = matrice, è stata fatta questa scelta perché l'implementazione e la gestione risultano semplici. Ci sono altri metodi che permettono di lavorare con più matrici ma richiedono una gestione più raffinata da parte del driver, uno è quello di dichiarare buffer multipli e permettere all'utente di scegliere quale usare per una certa operazione, per fare ciò oltre a dover passare al dispositivo il puntatore di volta in volta bisognerebbe anche inserire degli offset associati ai vari buffer nella memoria fornita da `mmap`. Un altro metodo possibile è quello di avere più dispositivi virtuali ma in questo caso oltre a dover fornire un buffer diverso ad ogni istanza bisognerebbe anche inserire dei lock per garantire la mutua esclusione riguardo l'uso del dispositivo.

5.1.3 Libreria

La libreria permette in primis di accedere e modificare il buffer visto in precedenza, tramite il puntatore ottenuto con `mmap`, ma anche di far svolgere all'acceleratore le sue funzioni su di esso. Nella libreria sono infatti presenti le stesse operazioni eseguibili dall'acceleratore, ma qui sono essenzialmente una serie di `ioctl` che tramite il driver scrivono tutte le informazioni necessarie dentro i BAR e successivamente fanno partire l'operazione scelta. Queste funzioni possono essere eseguite singolarmente o in alternativa è presente una coda con cui è possibile creare una sequenza di operazioni da eseguire sul buffer.

Avendo ora visionato tutti i componenti presenti si può notare come siano presenti dei collegamenti logici tra di essi:

Programma utente ↔ Libreria ↔ Dispositivo virtuale ↔ Driver ↔ Dispositivo emulato

5.2 Test

Dopo aver terminato la creazione del prototipo resta un'ultima fase, ossia raccogliere dei dati ed analizzarli, andando così a scoprire se l'uso di questo metodo risulti effettivamente conveniente. Per avere un qualcosa con cui confrontarlo è stata creata una seconda libreria (in C) che implementa direttamente le stesse operazioni presenti nell'acceleratore e su di essa verranno raccolti gli stessi dati[2], questo metodo è stato

[2]Per avere dei valori equi è necessario usare la flag di ottimizzazione `-O2` per i test con la libreria, questo perché QEMU la usa a sua volta.

scelto perché risulta la modalità più immediata (seppur grossolana) per riprodurre il comportamento di un acceleratore.

Come metro di giudizio è stato scelto il tempo totale necessario per svolgere un'operazione, qui ci si concentrerà sulla conversione di una matrice RGB in scala di grigi seguita dalla convoluzione con filtro di Sobel[3]. Dato che il sistema emulato e quello ospite hanno entrambi architettura x86_64 si può sfruttare l'istruzione macchina `RDTSC` che restituisce il timestamp del processore, ossia il numero di cicli di clock trascorsi dall'ultima reimpostazione. Per ottenere il tempo trascorso (in secondi) con questo metodo basta prendere i valori dati da `RDTSC` prima e dopo l'operazione, sottrarre al secondo il primo e dividere tutto per la frequenza del processore (2,6 GHz in questo caso). Bisogna però fare un'osservazione, misurare il tempo all'interno di una macchina virtuale non è cosa semplice e i valori ottenuti potrebbero essere non veritieri o comunque sballati [37], con QEMU si può arginare il problema abilitando l'ipervisore che, almeno con `RDTSC`, porta ad avere dei valori quantomeno attendibili essendoci accesso diretto alla CPU.

Ovviamente per avere una visione completa non ci si può soffermare su un solo caso e soprattutto a poche ripetizioni, quindi vengono prese in esame 3 matrici di grandezza crescente e per ognuna si eseguono le operazioni dette sopra in 3 casi diversi: con l'acceleratore, direttamente con la libreria e sempre con la libreria ma al di fuori dell'emulazione. Ogni singolo caso viene ripetuto 1000 volte portando ad un totale di 9000 misure che sono riportate negli istogrammi in figura 5.2, la figura è organizzata in una sorta di griglia: le righe sono le 3 matrici, le colonne i 3 casi. Ognuna ha comunque un titolo quindi risulta semplice distinguerle.

Dai dati si evince, oltre al naturale aumento dei tempi medi di esecuzione con matrici più grandi, che la situazione muta:

- Con matrici piccole l'esecuzione esterna è la più veloce, segue quella interna (+6,6%)[4] mentre il dispositivo non solo è il più lento (+38,4%) ma presenta anche una notevole variabilità nei tempi rilevati.
- Per matrici medie la situazione cambia, il dispositivo è il più veloce, l'esecuzione esterna è leggermente più lenta (+3%) mentre quella interna arriva ad essere maggiore del 20,2%.
- Per quelle grandi si consolida il concetto, l'uso del dispositivo diventa decisamente più vantaggioso, l'esecuzione esterna arriva a +10,1% e quella interna a +32,5%.

In tutti i casi è comunque presente una coda a destra che tende a crescere con l'ingrandimento della matrice, si differenzia leggermente la libreria interna che per matrici medio-grandi ha un andamento simile ad una distribuzione bimodale. La presenza di queste code può dipendere da tanti fattori ma si può ipotizzare su quali siano i più influenti: per il dispositivo va sicuramente tenuto conto dell'overhead causato dalla sua emulazione, per gli altri 2 casi invece va considerata l'eventuale presenza di altri processi attivi che possano influenzare lo scheduling.

[3] Usato per il riconoscimento dei contorni in un'immagine.

[4] Percentuali relative ai tempi medi, espresse rispetto al caso migliore.

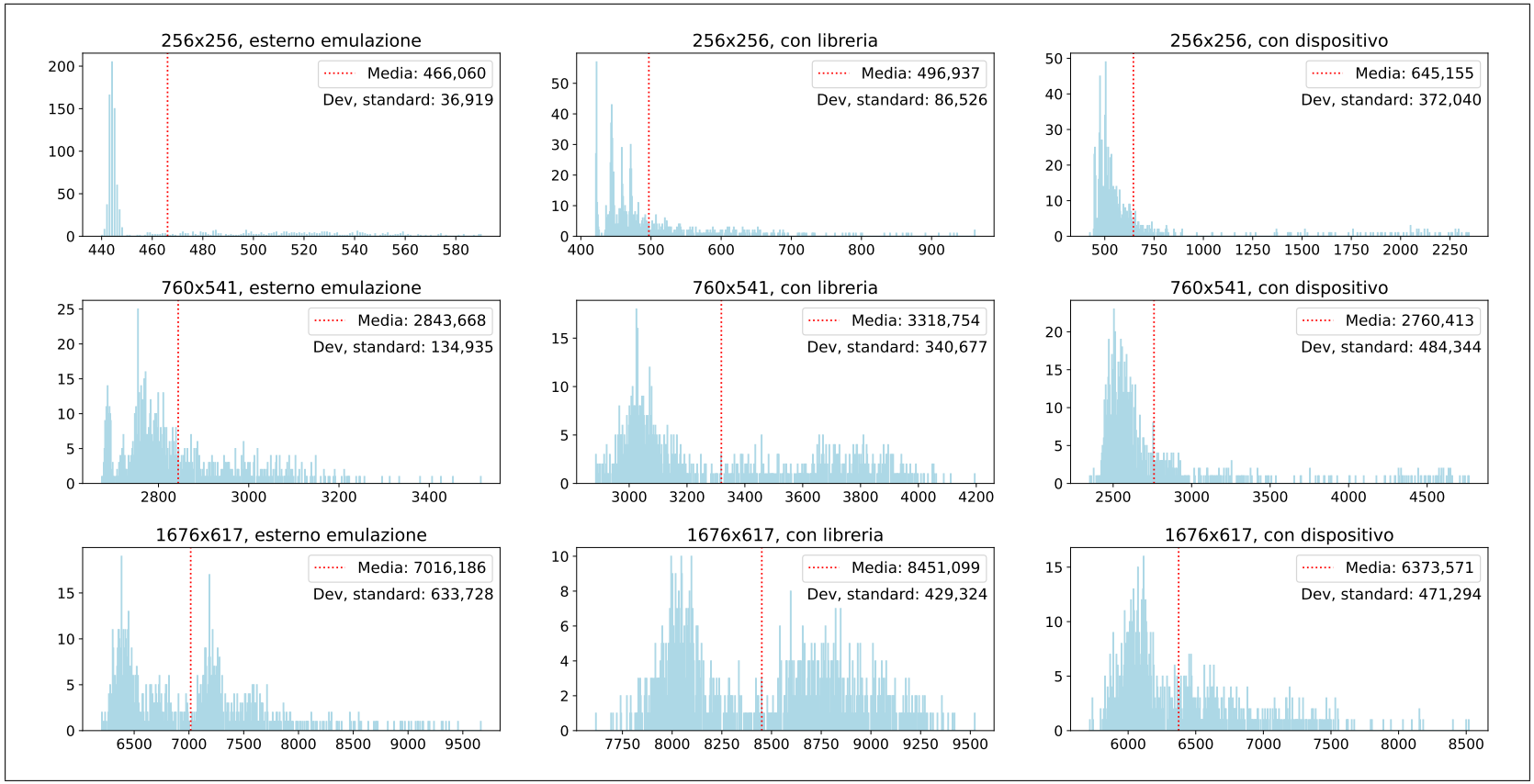


Figura 5.2. Tempi di esecuzione in μS

Capitolo 6

Conclusioni

6.1 Risultati ottenuti

L'obiettivo di questo elaborato è stata l'esplorazione dell'emulazione come supporto allo sviluppo di acceleratori hardware, con particolare attenzione posta alla possibilità di disaccoppiare le fasi di progettazione hardware e software degli stessi.

La realizzazione del prototipo, del driver associato e della libreria utente per usarlo, ha dimostrato che è possibile riprodurre in maniera relativamente fedele il comportamento funzionale di un dispositivo fisico, seppur con la presenza di alcune limitazioni specifiche intrinseche all'emulazione stessa e/o a QEMU. Queste limitazioni però intaccano solamente degli ambiti specifici, il fatto che la parte fisica non sia presente crea problemi solamente se si vuole emulare qualcosa nel campo embedded o real-time, mentre molte delle funzionalità PCIe mancanti/ridotte sono comunque meno usate rispetto a quelle presenti e sono di interesse, ovviamente, solo se si usa PCIe come metodo di connessione. Un aspetto cardine dell'analisi svolta è stato il confronto tra questo sistema ed un altro metodo più semplice, ossia implementare il tutto direttamente con una libreria. Ovviamente il secondo metodo è più semplice a livello implementativo, infatti richiede delle conoscenze meno approfondite ed è anche più immediata la modifica del codice ed il debug.

Con i dati ottenuti è possibile concludere che entrambi i metodi siano comparabili, nel dettaglio si può constatare come con input di dimensioni importanti e/o operazioni particolarmente intensive la libreria tende, a livello temporale, ad essere più lenta rispetto al dispositivo emulato mentre nei casi opposti risulta vero il contrario. Tutto questo risponde al quesito iniziale, l'emulazione risulta uno strumento più che valido in questo ambito ed un buon punto di partenza per lo sviluppo iniziale del software.

6.2 Possibili lavori futuri

Il lavoro svolto ha dimostrato l'efficacia dell'emulazione, ma resta comunque un margine per ulteriori approfondimenti. Un primo ambito potrebbe riguardare l'uso di altri metodi di connessione e/o altri ambienti emulati, ma anche l'implementazione di dispositivi con funzionalità più complesse potrebbe portare alla scoperta di nuove informazioni.

Bisogna comunque tenere conto che con il passare del tempo si potrebbe verificare un superamento delle varie limitazioni oggi presenti e allo stesso tempo potrebbero nascere nuovi strumenti che non le presentano già dal principio.

Ringraziamenti

Questo punto delinea la fine di un lungo ed arduo percorso che non sarebbe mai stato portato a compimento senza il continuo sostegno della mia famiglia, dei miei amici e della grande tenacia che ho scoperto di possedere. I giusti riconoscimenti vanno dati anche al prof. Marini per avermi instradato verso questo tirocinio e a Diego per essere stato un fondamentale punto di riferimento prima e durante la stesura di questo documento.

Bibliografia

[1] *8087 Math Coprocessor Datasheet*, Intel, Oct. 1989.

[2] *Sun Crypto Accelerator 1000 Installation and User's Guide, Version 2.0*, Oracle Corporation, Oct. 2005.

[3] O. Shacham and M. Reynders, "Pixel Visual Core: image processing and machine learning on Pixel 2," https://blog.google/products/pixel/pixel-visual-core-image-processing-and-machine-learning-pixel-2, Google, Oct. 2017, accessed: 2025-06-24.

[4] A. S. Tanenbaum and T. Austin, *Architettura dei calcolatori: Un approccio strutturale*, 5th ed. Pearson Education Italia, 2006.

[5] Fact.MR, "Hardware Acceleration Market Outlook (2023 to 2033)," Fact.MR, Market Research Report, Jul. 2023.

[6] B. Zaid, "Analysis of the PC Video Games and GPU Market," https://medium.com/@baraaz/analysis-of-the-pc-video-games-and-gpu-market-dbdaecc2b56f, Jan. 2023, accessed: 2025-06-25.

[7] W. Hwu and S. Patel, "Accelerator Architectures — A Ten-year Retrospective," *IEEE Micro*, 2018.

[8] B. Peccerillo, M. Mannino, A. Mondelli, and S. Bartolini, "A survey on hardware accelerators: Taxonomy, trends, challenges, and perspectives," *Journal of Systems Architecture*, 2022.

[9] STMicroelectronics, "STM32L4 Advanced Encryption Standard hardware accelerator, Revision 2.0," 2016.

[10] Intel, "Intel® Neural Compute Stick 2: Product Brief," 2019.

[11] Texas Instruments, *A Basic Guide to I^2C*, Nov. 2022.

[12] Maxim Integrated, "DS2465 SHA-256 Coprocessor Datasheet," 2012.

[13] Fact.MR, "PCI-E Connectors Market Forecast, Trend Analysis & Competition Tracking - Global Review 2021 to 2031," Fact.MR, Market Research Report, Nov. 2021.

[14] A. Verma and P. K. Dahiya, "Pcie BUS: A State-of-the-Art-Review," *IOSR Journal of VLSI and Signal Processing*, Jul. 2017.

[15] J. Jaeger, "FPGA-based rapid prototyping of ASIC, ASSP, and SoC designs," *EDN (via plDesignLine / Design & Reuse)*, Oct. 2009.

[16] *PCI Express ® Base Specification, Revision 5.0, Version 1.0*, PCI-SIG, May 2019.

[17] *ExpressCard ® Standard, Release 2.0*, PCMCIA, 2009.

[18] PXI Systems Alliance, *PXI Express Hardware Specification, Revision 1.1*, May 2018.

[19] Intel, "AN 835: PAM4 Signaling Fundamentals," https://www.intel.com/content/www/us/en/docs/programmable/683852/current/nrz-fundamentals.html, Dec. 2019, accessed: 2025-06-27.

[20] L. Harvie, "Understanding and Implementing Differential Pair Routing in High-Speed PCBs," https://runtimerec.com/understanding-and-implementing-differential-pair-routing-in-high-speed-pcbs, Mar. 2025, accessed: 2025-06-27.

[21] J. B. Yoon and N. Malone, "PCIe Link Training Overview," Texas Instruments, Tech. Rep., Aug. 2022.

[22] J. Corbet, A. Rubini, and G. Kroah-Hartman, *Linux Device Drivers*, 3rd ed. O'Reilly Media, 2005.

[23] R. Budruk, D. Anderson, and T. Shanley, *PCI Express System Architecture*. MindShare Inc., 2003.

[24] J. Liu, "PCI Expansion ROM," http://liujunming.top/2021/06/30/PCI-Expansion-ROM, Jun. 2021, accessed: 2025-07-29.

[25] *PCI Code and ID Assignment Specification, Revision 1.11*, PCI-SIG, Jan. 2019.

[26] *PCI Local Bus Specification, Revision 3.0*, PCI-SIG, Feb. 2004.

[27] Royal Holloway University of London, Department of Computer Science, "An Introduction to Linux," https://intranet.royalholloway.ac.uk/computerscience/computerdocumentation/introductiontolinux.pdf, accessed: 2025-07-01.

[28] Boston University, "Introduction to Linux," https://www.bu.edu/tech/files/2018/05/2018-Summer-Tutorial-Intro-to-Linux.pdf, 2018, accessed: 2025-07-01.

[29] *Advanced Configuration and Power Interface Specification, Release 6.6*, UEFI Forum Inc., May 2025.

[30] C. Kuehl, "How the Linux Kernel Detects PCI Devices and Pairs Them With Their Drivers," https://codeofconnor.com/how-the-linux-kernel-detects-pci-devices-and-pairs-them-with-their-drivers, 2019, accessed: 2025-07-02.

[31] The QEMU Project, "Qemu documentation," https://www.qemu.org/docs/master, accessed: 2025-07-03.

[32] A. Acker, "Emulation practices for software preservation in libraries, archives, and museums," *Journal of the Association for Information Science and Technology*, 2021.

[33] W. Stallings, *Sicurezza dei computer e delle reti*. Pearson Italia, 2022.

[34] Y. Li, P. Xu, and H. Wan, "A Fault Injection System Based on QEMU Simulator and Designed for BIT Software Testing," *Applied Mechanics and Materials*, Feb. 2013.

[35] E. Bugnion, J. Nieh, and D. Tsafrir, *Hardware and Software Support for Virtualization*. Morgan & Claypool, 2017.

[36] QEMU Developers, "Qemu source code," https://github.com/qemu/qemu/tree/master, accessed: 2025-07-03.

[37] V. Kuznetsov, "An introduction to timekeeping in Linux VMs," https://opensource.com/article/17/6/timekeeping-linux-vms, Jun. 2017, accessed: 2025-08-10.