eman ta zabal zazu

Universidad del País Vasco Euskal Herriko Unibertsitatea

BILBOKO INGENIARITZA ESKOLA
ESCUELA DE INGENIERÍA DE BILBAO

MÁSTER UNIVERSITARIO EN

SISTEMAS ELECTRÓNICOS AVANZADOS

TRABAJO FIN DE MÁSTER

ESTADO DEL ARTE DE LOS HIPERVISORES PARA FPGAS Y SUS INTERFERENCIAS EN LA COMUNICACIÓN

Estudiante *Terradillos, Latorre, Julen*
Director/Directora *Lázaro, Arrotegui, Jesús*

Departamento **Tecnología Electrónica**
Curso académico *2020/2021*

Bilbao, 31, Septiembre, 2021

Resumen

Con el aumento exponencial de las necesidades tanto de usuarios como de empresas de aumentar la velocidad de comunicación de sus sitemas, los desarrolladores se han visto en la obligación de usar cada vez más FPGAs. Sin embargo, las FPGAs resultan ser un elemento muy caro y potente al que, en muchas ocasiones, no se llega a sacar el 100 % de la utilidad, de modo que, con el objetivo de conseguir un ahorro económico (y ya que las mismas lo permiten) las FPGAs se han empezado a virtualizar.

La virtualización es una tecnología bastante conocida, pues se emplea tanto en ordenadores de escritorio como en servidores, y esta misma tecnología se está trasladando ahora a arquitecturas ARM, haciendo posible la instalación de hipervisores en FPGAs.

En este trabajo se realizará un estudio sobre los diversos hipervisores disponibles actualmente en el mercado para, seguidamente, realizar la puesta en marcha de un hipervisor XNG sobre una ZYBO y una evaluación de las velocidades en sus comunicaciones de red.

La primera parte, recogida en los capítulos 5 a 8, es un estado del arte centrado en los virtualizadores que incluye una clasificación de los mismos, junto con una descripción de los distintos hipervisores y su correspondiente clasificación.

La segunda parte, englobada en los capítulos 9 a 10, contiene una breve descripción de algunos de los hipervisores que pueden ser usados en arquitecturas ARM.

La tercera y última parte de este trabajo se encuentra en el capítulo 11 y describe, primeramente, cómo instalar Linux en una tarjeta de desarrollo ZYBO y, en segundo lugar, cómo instalar el hipervisor XNG y una partición de Linux en la misma placa ZYBO. En este apartado también se presentan las mediciones de las velocidades de comunicación para cada uno de los supuestos y se comparan los resultados obtenidos.

Palabras clave: Hipervisor, FPGA, ARM, XNG, ZYBO

Laburpena

Erabiltzaileek eta enpresek beren sistemen komunikazio-abiadura handitzeko dituzten beharrak esponentzialki handitu direnez, garatzaileek gero eta FPGA gehiago erabili behar izan dituzte.

Hala ere, FPGAk oso elementu garesti eta indartsuak dira, eta, askotan, ez zaie %100 baliagarritasuna ateratzen; beraz, aurrezki ekonomikoa lortzeko helburuarekin (eta FPGA-ek ahalbidetzen dutenez), FPGAk birtualizatzen hasi dira. Birtualizazioa teknologia nahiko ezaguna da, mahaigaineko ordenagailuetan zein zerbitzarietan erabili ohi izan dena, eta egun teknologia hau ARM arkitekturetara eramaten hasi da, FPGAn hiperbisoreak instalatzeko aukera emanez.

Lan honetan, gaur egun merkatuan dauden hiperbisoreei buruzko ikerketa bat aurkeztuko da, eta, jarraian, ZYBO baten gainean XNG hiperbisore bat martxan jarriko da sareko komunikazioetako abiaduren ebaluazioa egiteko.

Lehenengo atala, 5. kapitulutik 8.era , birtualizatzaileetan zentratutako artearen egoera bat da, hauen sailkapen batekin batera, eta hiperbisore ezberdinen deskribapenak ere jasotzen ditu, dagokien sailkapenarekin.

Bigarren atalean, 9. eta 10. kapituluetan, ARM arkitekturetan erabil daitezkeen hiperbisoreetako batzuen deskribapen laburra jasotzen da.

Bukatzeko, lan honen hirugarren eta azken atala 11. kapituluan dago, eta, lehenik, Linux ZYBO garapen-txartel batean nola instalatzen den deskribatzen du, eta, bigarrenik, XNG hiperbisorea eta Linux partizio bat ZYBO plaka berean nola instalatu azaltzen du. Atal honetan, kasu bakoitzerako komunikazio-abiaduren neurketak ere aurkezten dira, eta lortutako emaitzak alderatzen dira.

Hitz-gakoak: Hipervisor, FPGA, ARM, XNG, ZYBO

Abstract

With the exponential rise in the needs of both users and companies to increase the communication speed of their systems, developers have been forced to use more and more FPGAs.

However, FPGAs turn out to be a very expensive and powerful element that, on many occasions, is not used to its full potential, so, in order to achieve economic savings (and since they allow it), FPGAs have begun to be virtualized. The virtualization is a well known technology, commonly used in both desktop computers and servers, whitch is now being transferred to ARM architectures making it possible to install hypervisors in FPGAs.

In this paper, a study of the various hypervisors currently available on the market will be carried out, followed by the implementation of an XNG hypervisor on a ZYBO and an evaluation of its network communication speeds.

The first part, contained in chapters 5 to 8, shows the state of the art mainly focused on virtualizers, including a classification of them, together with a description of the different hypervisors and their corresponding classification.

The second part, comprising chapters 9 to 10, contains a brief description of some of the hypervisors that can be used on ARM architectures.

The third and last part of this work is found in chapter 11 and describes, firstly, how to install Linux on a ZYBO development board and, secondly, how to install the XNG hypervisor and a Linux partition on the same ZYBO board. This section also presents the measurements of the communication speeds for each of the assumptions and compares the results obtained.

Keywords: Hypervisor, FPGA, ARM, XNG, ZYBO

Índice

1. Glosario de acrónimos **8**

2. Introducción **10**

3. Objetivos **12**

4. Conceptos previos **13**
4.1. Field Programmable Gate Array (FPGA) 13
4.1.1. Composición Interna de una FPGA 14
4.2. Microprocesador 16
4.3. Zynq Board (ZYBO) 17
4.4. Vivado 18
4.5. Microkernel 19
4.6. Iperf 20
4.7. Linux Embebido 20
4.8. First Stage Bootloader (FSBL) 20
4.9. Second Stage Bootloader (SSBL) 21
4.10. Kernel de Linux 21
4.11. Device Tree 21
4.12. Cross Compile 21

5. Virtualizadores de Hardware **22**

6. Clasificación de Virtualizadores para FPGAs **23**
6.1. Nivel de Fuente 23
6.2. Nivel de Nodo 23
6.3. Nivel de Multi-Nodo 24

7. Hipervisores **26**
7.1. Arquitectura de un hipervisor 27

8. Paravirtualización versus Virtualización completa **29**

9. Hipervisores más usados **30**
9.1. Jailhouse 30
9.2. OPTIMUS 31
9.3. Ker-ONE 33
9.4. Mini-NOVA 34
9.5. PikeOS 35
9.6. Codezero 37
9.7. XEN 38
9.8. Rodovisor 39
9.9. Hipervisor Tipo 0 39

9.10. OKL4 microvisor 41
9.11. Xtratum Next Generation (XNG) 41

10. Tabla de hipervisores **47**

11. Caso práctico **48**
11.1. Prerrequisitos 48
11.2. Resumen de la configuración 48
11.3. Puesta en marcha de Linux 49
11.4. Puesta en marcha de Xtratum 54
11.5. Análisis de la red 64

12. Conclusiones **67**

13. Trabajo Futuro **68**

A. **69**

B. **69**

Índice de figuras

1. Logic cell 15
2. Memoria de una FPGA. 15
3. Zybol 17
4. Microkernel 19
5. Tipos de hipervisores 27
6. Componentes de un Hypervisor 28
7. Arquitectura del hipervisor jailhouse. 30
8. Arquitectura del hipervisor OPTIMUS. 32
9. Arquitectura del hipervisor Ker-ONE 33
10. Arquitectura del hipervisorMini-NOVA. 34
11. Arquitectura del PikeOS. 36
12. Arquitectura del Codezero. 37
13. Arquitectura del XEN. 38
14. Arquitectura del Rodovisor. 40
15. Arquitectura del hipervisor de tipo 0. 40
16. Arquitectura del OKL4. 41
17. Arquitectura del XNG 42
18. Estados del hipervisor y transiciones. 44
19. Estados de las particiones y sus transiciones 45
20. Diseño de bloques para la configuración de Linux. 52
21. Selección ZYBO 56
22. Línea que habilita el AXI no seguro 56
23. Diseño de bloques 57
24. Usuario y pasword de Linux 60
25. root file system 61
26. Menú de configuración de root file system. 62
27. Sistema completo del PC-ZYBO. 64
28. Terminal de Windows ejecutando Iperf como servidor. 64
29. Resultados obtenidos al ejecutar Iperf en Linux como cliente y Windows como servidor. 64
30. Resultados obtenidos de ejecutar Iperf en XNG como cliente y Windos como servidor. 65
31. Resultados obtenidos de ejecutar Iperf en Linux como servidor y Windows como cliente. 65
32. Resultados obtenidos de ejecutar Iperf en XNG como servidor y Windows como cliente. 65
33. Gráfica con todas las muestras de las pruebas. En azul las muestras cuando Linux es el servidor. En rojo cuando XNG es servidor. En verde las muestras cuando Linux es cliente. En rosa las muestras cuando XNG es cliente 66

1. Glosario de acrónimos

ALU: Unidad Aritmetico Lógica.

API: Application Programming Interface.

ARM: Advanced RISC Machine.

BRAM: Block RAM.

BP: Baseband radio Processor.

CISC: Complex Instruction Set Computer.

DPR: Dynamic Partial Reconfiguration.

DRAM: Dinamic RAM.

DSP: Digital Signal Processor.

FPGA: Field Programable Gate Array.

FSBL: First Stage Boot Loader.

GIC: Generic Intrrup Controller.

HAM: Hypervisor Auxiliary Model.

HDL: Hardware Description Language .

I/O: Input and Output.

IBM: International Business Machines Corporation.

IOMMU: Input-Output Memory Management Unit.

IPC: Inter-Process Communication.

IPVI: Inter-Partition Virtual Interrupt.

LC: Logic cell.

LUT: Lookup table.

MMU: Memory Management Unit.

OS: Sistema Operativo.

PL: Programable-logic.

PS: Procesing-system.

RAM: Random Acces Memory.

RISC: Reduced Instruction Set Computer.

RTL: Register transfer level.

SOC: System on Chip.

SSBL: Second Stage Boot Loader.

TCB: task Control Block.

TSP: Time space partitioning.

vFPGA: virtual FPGA.

VHDL: Very High Speed Integrated Circuit Hardware Description Language.

VHSIC: Very High Speed Integrated Circuit.

vGIC: virtual Generic Intrrup Controller.

vIRQ: virtual Interrup Request.

VMM: Virtual Machine Monitor.

VM: Virtual Machine.

XCF: XNG Configuration Files.

ZYBO: Zynq Board.

2. Introducción

En los ultimos años las FPGAs han sido el centro de atención para investigadores y empresas debido a su alto potencial para desarrollar númerosas aplicaciones en todos los ámbitos posibles, ya que son capaces de procesar grandes cantidades de datos de forma paralela. Los microprocesadores, sin embargo, han estado destinados a aplicaciones de ejecución secuencial y con uso definido. La comunicación entre ambos dispositivos ha estado limitada por su velocidad de transferencia.

No obstante, ahora, las arquitecturas de las FPGAs cuentan con un microprocesador con el que comparten buses y memoria, de forma que se le facilita al microprocesador la ejecución de las tareas de tratamiento de datos, la configuración de entradas y salidas y, además, se reducen tiempos de reconfiguración de la FPGA. Algunas compañias están empleando estos dispositivos para acelerar sus sistemas (como, por ejemplo, Google, que utiliza esta herramienta para aumentar su velocidad de en los motores de busqueda [55]) y otras (como Amazon) usan los aceleradores para reducir el tiempo que tardan sus máquinas pick&place en mover cada objeto [33].

Hace algunos años, mientras las FPGAs estaban iniciando su desarrollo, los virtualizadores ya se habían establecido como líderes en el ámbito de computación virtual [22] permitiendo usar un ordenador sin poseerlo. Con el paso de los años este método se ha extendido a diferentes areas como redes definidas por software [15] o computación en la nube [18].

Aunque la virtualización en escritorios y servidores es ya madura, a la virtualización en sistemas embebidos le queda aún un largo camino por delante, a pesar de que crece rápidamente. La virtualización de los sistemas embebidos es una solución mucho más compleja que la virtualización en escritorios y servidores, ya que esta joven tecnología necesita diferentes técnicas para virtualizar al tener recursos muy limitados, pero promete ser una solución para los sistemas embebidos más desarrollados como control industrial en tiempo real, pues aporta complejidad del software, seguridad y robustez.

Uno de los ámbitos donde más se ha desarrollado el uso de sistemas de virtualización es en la electrónica de consumo, más concretamente la tecnología del teléfono móvil [24], donde los móviles han pasado de ser sistemas de comunicación por voz a ser sistemas más parecidos a un ordenador. En este sector, la virtualización ha conseguido la combinación de sistemas con requerimientos en tiempo real y el crecimiento de funcionalidades más diversas [25].

En el presente trabajo se van a comentar los siguientes aspectos:

- En el apartado 3, se van a comentar los objetivos.
- En el apartado 4, se van a tratar algunos conceptos que resultarán útiles para el entendimiento del trabajo.
- En el apartado 5, se hablará sobre qué son virtualizadores.
- En el apartado 6, se tratará de realizar una clasificación genérica de los virtualizadores.
- En el apartado 7, se comentarán más a fondo los hipervisores, así como su clasificación y características principales.
- En el apartado 8, se expondrá las diferencias que existen entre una virtualización completa y una paravirtualización.
- En el apartado 9, se comentarán determinados detalles de los hipervisores.
- En el apartado 10, se hará un breve resumen con los datos de los hipervisores vistos del apartado 9.
- En el apartado 11, se realizarán pruebas con el hipervisor XNG para ver las interferencias que genera al comunicar una placa de desarrollo ZYBO con la red.
- En el apartado 12, se expondrán las conclusiones del estudio de este trabajo.
- En el apartado 13, se comentarán distintas líneas de trabajo que sería interesante estudiar.

3. Objetivos

Este trabajo tiene tres objetivos:

- Por un lado, introducir al lector en el ámbito de los hipervisores para FPGAs.
- Por otro, hacer una breve descripción de los distintos modelos de hipervisores para saber cuál es el mejor para cada uso.
- Por último, hacer una puesta en marcha del hipervisor XNG y comparar su velocidad de comunicación TCP con la de un sistema que no use ningún hipervisor.

4. Conceptos previos

En el presente apartado se encuentran definidos o comentados distintos conceptos cuya comprensión resulta útil para poder entender el contenido del conjunto del trabajo.

4.1. Field Programmable Gate Array (FPGA)

Una FPGA es un dispositivo semiconductor reprogramable formado por bloques lógicos programables embebidos en un mar de interconexiones. Las FPGAs no tienen ningún hardware fijo, sino que incluyen determinados componentes (puertas lógicas, flip-flops, multiplexores...) que se pueden interconectar para crear nuevos dispositivos. La descripción de estos circuitos se realiza a través del lenguaje de VHDL o Verilog. Las FPGAs pueden verse como un circuito en blanco esperando a ser configurado y, además, permiten hacer reconfiguraciones para mejorar el circuito, corregir errores o hacer adaptaciones en funciones de las necesidades del mercado. [41]

Diferencias entre las FPGAs y los CPLDs

Los dos dispositivos son chips reprogramables, sin embargo, la gran diferencia reside en su arquitectura:

- Las FPGAs contienen una mayor densidad de Logic Cell (LC) que los CPLDs.
- Las FPGAs están basadas en memoria RAM, lo que implica que tienen que ser reconfiguradas en cada reinicio, mientras que los CPLDs están basados en memoria EEPROM.
- Las FPGAs tienen un enrutamiento dedicado lo que les permite implementar de manera eficiente funciones aritméticas de diferentes maneras.

En general, las FPGAs permiten tener diseños digitales más grandes.

Diferencias entre las FPGAs y los microcontroladores

La principal diferencia entre estos dispositivos es que los microcontroladores cuentan con una circuitería ya definida y ejecutan instrucciones secuenciales definidas tras su fabricación. Por otro lado, las FPGAs no ejecutan instrucciones, sino que sus salidas se deben a los circuitos definididos en el diseño HDL y, por ello, permiten ejecuciones de forma paralela. Aunque los microcontroladores tengan perifericos que también se puedan ejecutar en paralelo con la CPU, siguen siendo menos configurables que las FPGAs.

Otra distinción importante es que las FPGAs son usadas cuando la velocidad en tiempo real es crítica, donde la cantidad de datos que hay que tratar es muy grande o cuando se requiera un alto número de proecesos en paralelo mientras que, por su parte, los microcontroladores son más faciles de programar y suelen ser suficientemente potentes para la gran mayoría de aplicaciones.

Finalmente, hay que atender a la diferencia de precio entre estos dispositivos. El valor de las FPGAs puede llegar a alcanzar varios cientos de euros, mientras que los microcontroladores se pueden encontar en el mercado por apenas un par de euros.

4.1.1. Composición Interna de una FPGA

A continuación se exponen los componentes internos de las FPGAs junto con una breve definición de los mismos [16] [41]:

Conexiones programables: Son pistas que atraviesan la parte lógica de las FPGAs y pueden unirse para conectar dos bloques internos. Existen dos tipos de conexiones:

- Conexiones directas: son conexiones que cruzan todo el SoC en todas las direcciones.
- Conexiones segmentadas: están asadas en líneas que puedes ser interconectadas formando una matriz.

A pesar de que las conexiones programables sean un elemento simple de manera aislada, permiten diseñar circuitos mucho más complejos. Las conexiones se le ocultan al usuario y pueden variar de generación en generación.

Bloques lógicos programables: Están constituidos por una o más funciones lógicas programables formadas por un Lookup Table (LUT) de 4-6 bits, un registro y una cadena de transporte todos ellos configurables. A la combinación de estos 3 elementos se le llama Logic Cell (LC). y una manera de medir la capacidad de una FPGA es medir la cantidad de LCs que tiene. Las LUTs también pueden usarse como memoria RAM, aunque lo óptimo sería usar la memoria RAM dedicada.

Memoria: Los diseños suelen utilizar una combinación de memoria embebida en el Programable Logic Array y memorias Doble Data Rate (DDR) externas. La Memoria puede ser implementada como registros discretos, como registros de desplazamiento, como RAM distribuida o como Block RAM (BRAM).

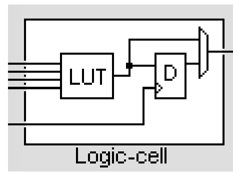


Figura 1: Logic cell [41].

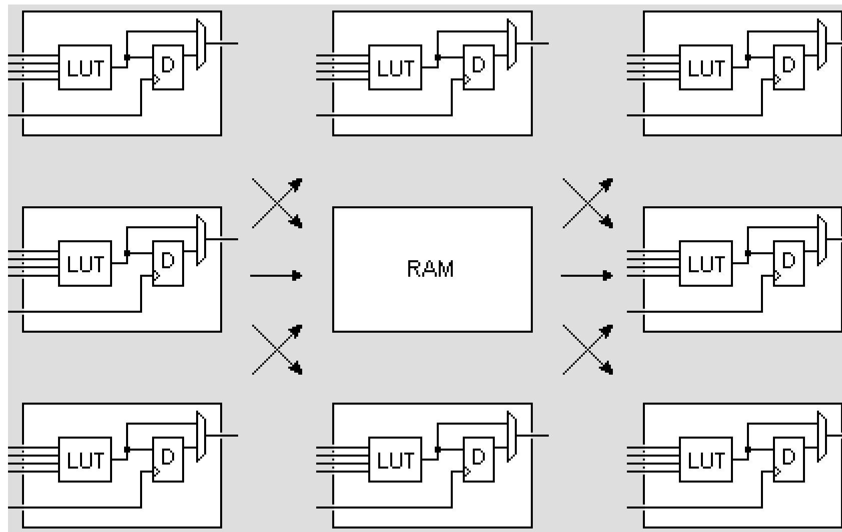


Figura 2: Memoria de una FPGA [41].

Bloque DSP: Las FPGAs más modernas tienen bloques Digital Signal Processor (DSP) que contienen multiplicadores discretos para realizar operaciones numéricas a muy alta velocidad.

Administración del reloj: Las operaciones síncronas requieren un flanco de reloj para registrar los resultados. La lógica usada por las FPGAs requiere uno o más sistemas de relojes para implementar lógica síncrona.

Bloques I/O: Son bloques que se encuentran junto a los pines de entrada y salida de las FPGAs. Su función es dar sincronía y retener el valor de la señal durante el tiempo necesario para que pueda ser leída o escrita. Además, poseen la circuitería para proteger la parte más interna del chip.

System On Chip (SoC): Solo los modelos más actuales de FPGA incluyen un microprocesador. Estos SoC incluyen una memoria externa, una memoria interna, unos periféricos comunes de I/O y un conjunto de interfaces conectados a la FPGA.

4.2. Microprocesador

Los microprocesadores son circuitos electrónicos integrados que se encargan del prosesamiento de datos, de realizar operaciones aritmetico-lógicas y de controlar los periféricos. Son los encargados de ejecutar todos los programas, desde el sistema operativo hasta aplicaciones de usuario. Se suelen clasificar según el número de instrucciones que son capaces de procesar en un tiempo determinado, la frecuencia del reloj y el número de bits utilizados por instrucción.

Componentes de un microprocesador

- Unidad de control: Encargada de interpretar las instrucciones.
- Undad Aritmetico-Lógica (ALU): Se encarga de procesar los datos dados por la unidad de control.
- Memoria: Espacio físico donde se guarda la información.
- Periféricos: Pines de I/O para comunicarse.

Arquitectura de un microprocesador

La arquitectura de un proceador es la forma en la que se han diseñado los elementos internos, y determina su rendimiento, las instrucciones que es capaz de manejar y su consumo. El uso de instrucciones simples reduce el cosnumo. La arquitectura determina la conexión entre la memoria y los canales, y las dos arquitecturas más conocidas son RISC y CISC.

Reduced Instruction Set Computing (RISC): La arquitectura cortex de Advanced RISC Machine (ARM) proporciona los diseños del procesador pero no los chips físicos. Estos diseños de procesador pueden modificarse antes de ser producidos por el fabricante. Esta arquitectura proporciona un conjunto de intrucciones para el desarrollo de software que permiten garantizar la interoperabilidad en diferentes dispositivos ARM. [13]

Complex Instruction Set Computing (CISC): Este tipo de arquitectura se caracteriza por tener un conjunto muy amplio de instrucciones y por permitir operaciones complejas entre operandos situdados en la memoria o en los registros internos. Dificulta el paralelismo entre instrucciones por lo que la mayoría de de sistemas CISC convierte las instrucciones complejas en instrucciones simples de tipo RISC.

4.3. Zynq Board (ZYBO)

El presente trabajo se realizará sobre una ZYBO, la placa de desarrollo más pequeña de la familia Xilinx Zynq 7000. Esta placa, desarrollada por Digilent, se basa en la arquitectura todo programable SoC, que integra un procesador de doble núcleo Cortex A9 de la familia ARM con una lógica programable de la serie 7 de Xilinx. [8]

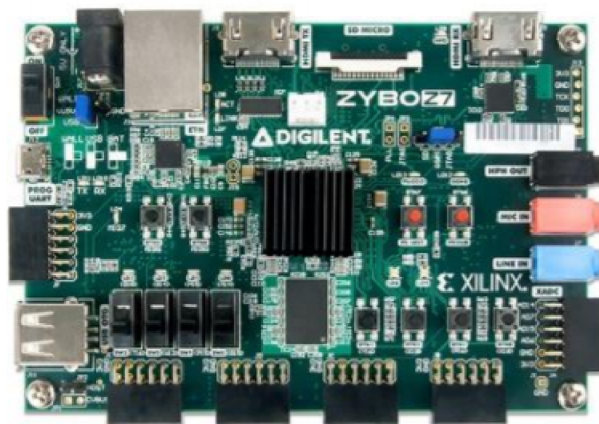


Figura 3: Placa de desarrollo Zynq Board (ZYBO).

La ZYBO es completamente compatible con las herramientas de desarrollo de Xilinx, a pesar de que Vivado, en una primera instancia, no pueda usarla y haya que descargar unos archivos [2] para poder hacerlo.

Esta placa de desarrollo cuenta con 3 formas de alimentación: mediante USB, mediante el uso de batería o mediante alimentación externa. Además, tiene distintos tipos de arranque: con tarjeta micrroSD, con QuadSPI Flash o JTAG. Tanto para seleccionar la alimentación como para seleccionar el arranque, existen unos jumpers para seleccionar el modo que se desea usar. En concreto, en este trabajo la alimentación se llevará a cabo por USB y se probarán los modos de arranque de tarjeta microSD y JTAG.

La ZYBO incluye 2 componentes de memoria que crean un único rango de interfaz de 32 bits de ancho con un total de 512MB de capacidad. Asimismo, cuenta con un puerto Ethernet que emplea dos LEDs para indicar que tiene conectado un RJ-45 y que la conexión es válida. A lo anterior se le suman otros periféricos como un conector HDMI, un conector VGA, conectores de audio, standar Pmods y Pmods de alta velocidad, entre otros (estos útimos no se van a utilizar en este trabajo). Finalmente, también cuenta con un botón que permite reiniciar el procesador, haciendo

que cargue de nuevo el contenido en la tajeta MicroSD, y otro botón que permite reiniciar la lógica configurable.

4.4. Vivado

Vivado Design Suite es un software desarollado por Xilinx para la síntesis y análisis de diseños HDL. Incluye un simulador, e introduce síntesis de alto nivel.

Describir hardware usando Vivado

Para describir el circuito de una FPGA, es necesario usar un lenguaje como VHDL, Verilog o lenguajes derivados de C.

El lenguaje VHDL, es el acrónimo de unir Very High Speed Integrated Circuit (VHSIC) y Hardware Description Languajge (HDL). Fue desarrollado por los investigadores del IEEE para describir circuitos digitales, y es un lenguaje que pasa por varias etapas hasta devolver el archivo que reconfigura la FPGA, llamado Bitstream. En este trabajo se utilizará el lenguaje VHDL.

- En primer lugar se debe crear el fichero VHDL donde se describirá el circuito.
- Testbench: Una vez se ha escrito el diseño del circuito se procede a simular, para ello es necesario describir un Testbench con las entradas y salidas. El Testbench no es sintetizable y, por lo tanto, solo se tendrá en cuenta cuando se le pida al programa que lo simule. Se utilizará el Testbench para simular todas las posibles entradas que puedan acarrear un problema al diseño, y así verificar que el circuito diseñado se ejecuta correctamente.
- Constrains: Luego, es necesario añadir el fichero de Constrains. En él se asignarán todas las entradas y salidas a pines físicos. Las Constrains pueden ser temporales, de frecuencia o características de I/O.
- Análisis RTL: El siguiente paso es llevar a cabo el análisis RTL (Register Tranfer Level). Este paso devuelve un circuito equivalente al descrito en HDL sin tener en cuenta dónde se implementará.
- Síntesis: Después se hace la síntesis, que se encarga de escoger los elementos que se van a usar y de interconectarlos. Al igual que en el análisis RTL, también devuelve un esquemático pero el de la síntesis muestra los recursos que utiliza de la FPGA.

- Implementación: En esta etapa previa a la generación al Bitstream se asignan, por un lado, tanto los bloques I/O como la localizacion de las entradas y salidas físicas y, por otro, las LUTs.
- Generación del Bitstream: En está última etapa se genera el archivo binario que se cargará en la FPGA.

4.5. Microkernel

El Kernel es una parte del OS que se encarga de conceder acceso al hardware para todo el software que lo solicita. Tiene que ejecutarse en modo privilegiado con acceso especial a los recursos del sistema, y puede servir como elemento de seguridad protegiendo del acceso de softwares maliciosos a todo el sistema.

Microkernel es la mínima cantidad de instrucciones primitivas para implementar un OS. Deja un número reducido de instrucciones dentro del espacio del Kernel (como la administración de memoria, la comunicación entre procesos, el programador, etc.) mientras que las otras partes (como los drivers, el sistema de carpetas o de redes) las deja en el espacio de ususario.

En un principio el Microkernel fue diseñado como un OS más seguro y fiable, una respuesta al concepto de Kernel monolítico que intentaba implementar todos los servicios del OS. La comparación entre estas dos estructuras queda reflejada en la figura 4.

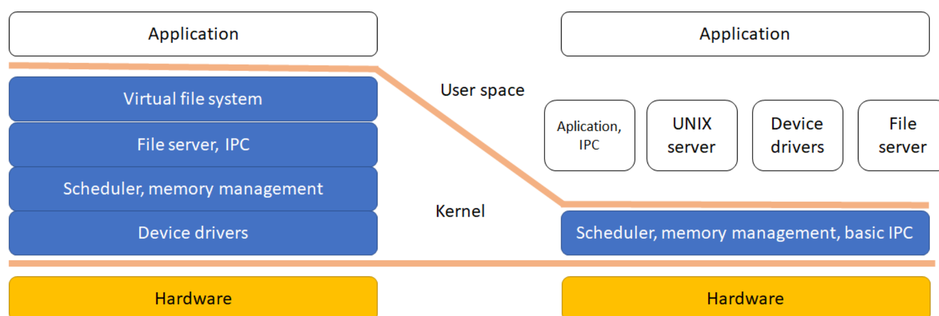


Figura 4: A la izquierda sistema operativo basado en Kernel monolítico. A la derecha sistema operativo basado en Microkernel.

Algunas investigaciones demuestran que los Microkernels pueden ser muy adecuados para el uso como hipervisores para la virtualización embebida [23]; sin embargo, muchos Microkernels solo admiten la para-virtualización, ya que esta tiene un mejor rendimiento y requiere

menor consumo de recursos. Además, requiere un mayor esfuerzo modificar el OS huésped que emplear la para-visulización, sin olvidar que esta modificación puede introducir bugs y hacer que las actualizaciones sobre el OS huésped sean más difíciles, caras y arriesgadas.

4.6. Iperf

El Iperf es una herramtienta informática que se usa en redes informáticas. Permite medir el rendimiento de la red generando tráfico TCP o UPD y, así mismo, el ajuste de varios parámetros relacionados con la sincronización, los protocolos y los búferes [3].

Iperf3

El Iperf3 es un rediseño de la versión original. Está implementado en Scratch con el principal objetivo de ocupar menos espacio, simplificar el código y ser una librería que puede ser usada por otros programas.

4.7. Linux Embebido

El Linux embebido se trata de una distribución de Linux muy ligera que tiene como objetivo los sistemas embebidos que tienen poca memoria y menor capacidad de procesado. En este sistema solo están incluidas las funciones más básicas y necesarias.

Xilinx provee 3 formas para construir una distribución embebida de Linux:

- Petalinux: Las herramientas de Petalinux no permiten generar una distribución como tal, sino que se usan para generar imagenes de Linux.
- The Yocto Project: Se trata de un proyecto open source que permite crear sistemas personalizados basados en Linux independientes de la arquitectura del hardware.
- Xilinx Open Source Linux: Permite construir y personalizar cada bloque del sistema de manera individual. Esta formado por U-Boot, Firmware ARM fiable, Kernel de Linux, GDB, GCC, librerías y otros softwares.

4.8. First Stage Bootloader (FSBL)

Configura el procesador y todos los periféricos. Carga el SSBL desde la memoria no-volátil a la memoria externa RAM y le da el control a ella.

4.9. Second Stage Bootloader (SSBL)

Se encarga de cargar el Kernel de Linux desde la memoria no-volátil. El SSBL le da el control al Kernel.

4.10. Kernel de Linux

Carga el sistema de capetas ROOT y ejecuta el sistema init. Como mínimo, para que funcione un Bootloader, es necesario que tenga las siguientes funciones [19].:

- Inicializar el hardware.
- Proveer parámetros de inicio para el OS.
- Iniciar el OS.

4.11. Device Tree

Es la estructura de información que se encarga de describir el hardware de una máquina concreta para que el Kernel pueda usarlo.

4.12. Cross Compile

Es un compilador capaz de crear un código ejecutable para otra plataforma que no sea en la que el compilador está corriendo.

5. Virtualizadores de Hardware

En este apartado se explicarán brevemente los distintos tipos de virtualizadores que existen.

La virtualización no es un concepto nuevo, se originó a finales de los años 60, cuando International Business Machiness Corporation (IBM) invertía un alto esfuerzo en crear soluciones compartidas en tiempo real, permitiendo manejar un ordenador sin poseerlo en realidad.

Inicialmente la virtualización de hardware se definía como técnicas de mapeo y arquitecturas que permitían cierta capacidad de independencia entre la aplicación y la arquitectura [42]. No obstante, esta cuestión ha variado con el tiempo, tal y como veremos en el siguiente apartado de este trabajo.

División Temporal: Este tipo de virtualización permite introducir una aplicación muy grande en un dispositivo que no dispone de suficiente capacidad de hardware. Consiste en dividir la aplicación en partes pequeñas para que puedan entrar en el dispositivo, al dividir las tareas significa que estas no deben solo ser correctas, sino que deben cumplirse sus tiempo límite [56].

Ejecución Virtualizada: Permite lograr cierta independencia del dispositivo con respecto a la familia. Se dividen las aplicaciones en múltiples tareas de comunicación y se utiliza un sistema de tiempo de ejecución para controlarlas [14].

Máquinas Virtuales: Las aplicaciones son diseñadas para funcionar en una arquitectura de computación abstracta. Después, esta arquitectura puede ser traducida en la arquitectura original usando herramientas de reasignado o usando un intérprete [42].

Sin embargo, las técnicas más modernas utilizadas para la virtualización de FPGAs se parecen más a la virtualización de software. Los principales objetivos del uso de la virtualización en las FPGAs son similares al de las CPUs [46]: uso del mismo dispositivo para múltiples usuarios [38], administración de recursos [34], ejecución, flexibilidad, aislamiento, escalabilidad, seguridad [44], habilidad del sistema para correr a pesar de fallos y productividad del programador.

6. Clasificación de Virtualizadores para FPGAs

El anterior criterio de clasificación puede quedar obsoleto debido al diseño de nuevas aplicaciones o requisitos. Se propone una clasificación más atemporal basada en la abstracción de niveles de los sistemas de computación que usan la virtualización [46].

6.1. Nivel de Fuente

Se pueden diferenciar dos tipos:

Arquitectura incrustada: Mejora la productividad y el tiempo de compilación, aunque también se usa para apoyar la portabilidad. Permite otro nivel de programación que se coloca encima del nivel más bajo de la FPGA . Divide el proceso de compilación en dos partes reduciendo el tiempo para generar un acelerador. Esta arquitectura puede ser usada por sistemas multi-core, sistemas de procesado customizados o por LUTs de grano fino. Un ejemplo de estos sistemas son los proveedores industriales como Microblaze [53].

Virtualización I/O: Es una capa intermedia entre las aplicaciones y los dispositivos que permite compartir los mismos recursos con la misma interfaz. Crea un canal entre el dispositivo y el usuario el cual no tiene porque corresponderse con el canal físico [45]. Se usa para mejorar las medidas de seguridad, facilitar la interfaz del I/O u optimizar el tiempo de acceso. Es muy parecido a la virtualización en una CPU.

6.2. Nivel de Nodo

Representa la infraestructura necesaria para gestionar una única FPGA, y se pueden encontrar tres tipos:

Virtual Machine Monitor (VMM): Tratan a la CPU como a un maestro y a la FPGA como a un periférico, permitiendo a los programadores llamar al hardware de la FPGA como una llamada a una librería . Aportan aislamiento entre los procesos o las máquinas virtuales, aunque también pueden centrarse en conseguir resiliencia, seguridad o administración de recursos [51].

Shells/Hipervisor: Se puede referir al Sistema Operativo (OS) de una FPGA o a un hipervisor para una Virtual FPGA (vFPGA). Se consigue aislamiento, una mejor gestión de recursos y la resolución de dependencia del controlador [40]. La vFPGA es un software que simula una FPGA y puede ejecutar programas como si fuese una FPGA real. Un ejemplo de

una vFPGA se encuentra en donde Robert et al. [31] utilizan un virtualizador unificado para compilar una capa dentro de las FPGAs para protegerlas de técnicas de ingeniería inversa.

Scheduling: Permite compartir la FPGA en el dominio del tiempo. Esta técnica no se puede aplicar a todas las FPGAs, ya que el estado de la información que necesita ser guardado no es trivial. Las técnicas más comunes son: con derecho preferente, sin derecho preferente y cooperativo. La opción sin derecho preferente consigue simplificar el diseño y puede ser implementada a bajo costo, mientras que la manera cooperativa ofrece un cambio de contexto cuando la aplicación alcanza un punto de control de ejecución. La mayoría son sin derecho preferente y se centran en la optimización del dominio del tiempo.

6.3. Nivel de Multi-Nodo

Su objetivo es repartir un trabajo de aceleración entre varias FPGAs. Se puede hacer comunicando una FPGA con otra, usando una CPU como servidor y FPGAs como clientes o usando las CPUs como servidores y clientes y que las FPGAs sean periféricos de la CPU.

Clúster personalizado: Los datos son pasados de una FPGA a otra cuando terminan el proceso, como una FIFO, aunque no generan latencia para comunicarse con otras FPGAs. En algunos casos se requiere que el usuario escriba el código en programas especiales para construir las FIFOs. [47]

Frameworks: La CPU se encarga de configurar las FPGAs y de administrar los datos, mientras que las FPGAs se encargan de la computación. Numerosas técnicas de gestión de datos han sido estudiadas para distribuir sistemas usando CPU, y muchas de esas técnicas pueden ser usadas para las FPGAs (como MapReduce [43]).

Servicios en la Nube: Abstraen completamente los detalles de dónde se está haciendo la computación. Al usuario solo se le garantiza el servicio, por lo que la computación la puede realizar una FPGA en lugar de una CPU. Este método difiere de los demás en que no provee una FPGA, pero sí una aplicación/servicio web. Un ejemplo de este modo de virtualización son los sistemas de Huawei [28] y Amazon [12].

Arquitecturas Híbridas: Existen arquitecturas híbridas que pueden soportar muchas formas de conectividad basadas en la FPGA a nivel de nodo. La técnica más común de administrar una FPGA es utilizar

OpenStack para proveer la FPGA y eprmitir al usuario conectar y programar la FPGA usando una dirección IP o MAC de la FPGA o vFPGA [20].

7. Hipervisores

Este apartado se usará para describir que es un hipervisor y los distintos tipos que existen.

Como ha quedado explicado, la función de los hipervisores es gestionar las FPGAs locales y su encapsulación [32]. Los hipervisores se pueden clasificar en 4 tipos:

Tipo 0: El hipervisor está dentro del hardware. No ofrece una completa virtualización [29]. El objetivo de este hipervisor son los procesadores de banda base (BP). Sobre este hipervisor en concreto se hablará más en 9.9.

Tipo 1 (hipervisor nativo o bare metal hypervisor): Se instala directamente en el hardware físico y no está conectado con el sistema operativo del huésped. El consumo de recursos es pequeño porque los procesos informáticos no se ejecutan a través del sistema operativo del huésped. Si uno de los OS falla, el hipervisor se encarga de que solo ese se anule, impidiendo que afecte a los demás. Del hipervisor pueden colgar tanto OS como aplicaciones individuales [40].

Este tipo de hipervisores son altamente eficaces, debido a que tienen que ser muy pequeños y extremadamente eficientes para poder usar la memoria, ya que la velocidad de está suele ser el limitador principal.

Gracias a su pequeño tamaño ganan en seguridad y también en fiabilidad, por ser el hipervisor el único que se ejecuta en modo privilegiado. Además, este sistema permite la interacción entre todos los sistemas que cuelgan del hipervisor.

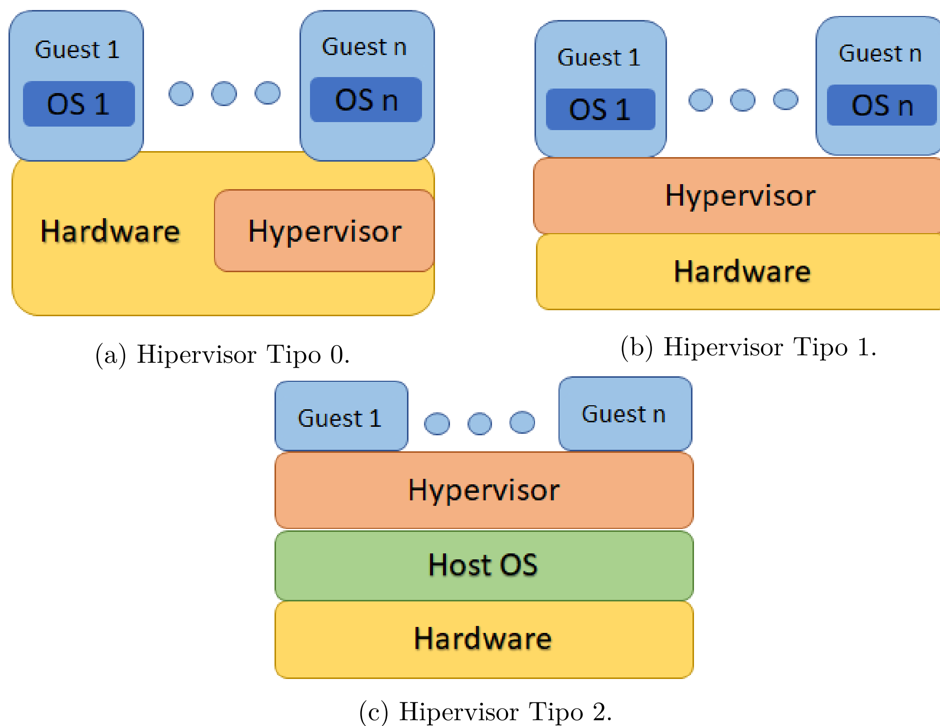
Los hipervisores Tipo1 funcionan como un OS ligero que se encarga de gestionar los recursos y controlar uno o varios OS sobre él.

Tipo 2 (hosted hypervisor): Requiere un OS que a su vez se base en el hardware físico. En el caso de que el OS principal falle, ese dispositivo deja de funcionar. Es el método de virtualización ideal para probar software sin riesgo de afectar nada de lo que haya en la máquina anfitriona. Su uso mas común suele ser doméstico.

En términos generales es una simulación de hardware para los huéspedes, y es el sistema operativo el que se encarga de gestionar los recursos y de la programación de tareas.

Hipervisores Híbridos: El hipervisor a veces interactúa sobre el hardware pero otras veces usa servicios que le proporciona el OS anfitrión.FR

A grandes rasgos, para la virtualización de sistemas embebidos, debido a sus recursos de hardware limitados, el hipervisor de Tipo 1 es preferible al Tipo 2, siendo este segundo más popular para virtualizar escritorios y servidores.



(a) Hipervisor Tipo 0.

(b) Hipervisor Tipo 1.

(c) Hipervisor Tipo 2.

Figura 5: Hipervisores: a) Tiene el hipervisor incrustado en el hardware, b) su hipervisor actúa como una capa de programación pero interactúa directamente con el hardware y, c) el hipervisor interacúa como una capa de programación entre el sistema operativo y los usuarios.

7.1. Arquitectura de un hipervisor

Un hipervisor es una capa que abstrae el sistema virtualizado del hardware de la máquina anfitriona. Lo que permite a cada invitado acceder a los recursos virtuales.

Los recursos o módulos mínimos para iniciar un sistema operativo anfitrión, que se pueden observar en la figura 6, son: un núcleo para

arrancar, memoria persistente y uno o varios dispositivos de red. La memoria está asociada habitualmente al disco físico de la máquina y el dispositivo de red a la tarjeta de red existente.

También deben presentar una serie de aplicaciones para ejecutar multiples invitados simultaneamente. Una capa que controle las funciones en modo Kernel. Los periféricos de I/O deben ser emulados en el Kernel. Las interrupciones deben ser capturadas por el hipervisor para poder ser enrutadas hacia el invitado adecuado. Como ejemplo se propone el caso de un error que obliga a bloquear el sistema, solo debería afectar a la máquina virtual que genera la interrupción.

Por último, una VMM necesita un enlacer de memoria que asocie las direcciones físicas de la memoria y un programador qué se encarge de seleccionar a que sistema se le asocia cuánto del tiempo actual. En vez de llamar a una excepción cuando la aplicación intenta acceder a los recursos en el momento que no están disponibles, el OS huésped llama al hipervisor.

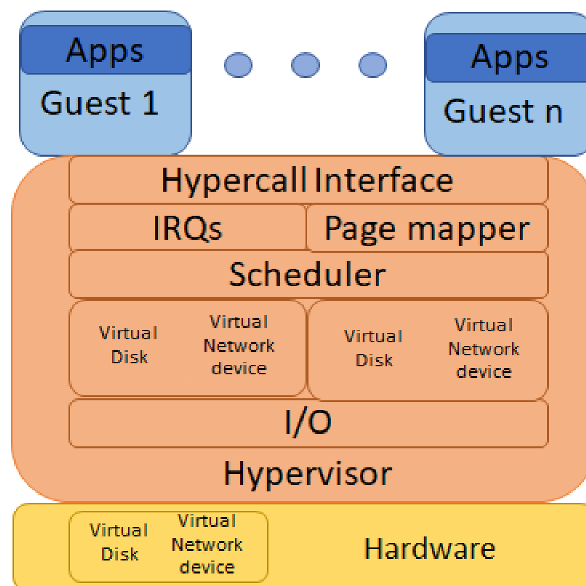


Figura 6: Componentes de un hipervisor.

8. Paravirtualización versus Virtualización completa

La paravirtualización es una técnica que introduce un software entre el OS huésped y el hypervisor. El OS huésped necesita ser porteado a la interface para poder correr el hypervisor [21]. Todas las instrucciones privilegiadas en el OS huésped deben ser reemplazadas por llamadas al hypervisor, después el hipervisor ya se encarga de hacer las llamadas al procesador para encargarse de esa instrucción.

Por otra parte, la virtualización completa ofrece una simulación completa del hardware, en lo que se refiere a conjuntos de intrucciones, perifericos I/O, interrupciones, acceso a memoria, etc. Cualquier sistema operativo puede ejecutarse en el hipervisor como un OS huésped sin ninguna modificación. Cuando el OS huésped ejecute una instrucción en modo privilegiado o haga una llamada al sistema, el hipervisor atrapará esa operación y emulará la instrucción o la llamada como si el OS huésped estuviera corriendose en hardware real.

La paravirtualización permite tener menor sobrecarga y mayor rendimiento, pero requiere más coste de desarrollo y puede contener más bugs. Ya que, en los sistemas embebidos, el recurso del hardware está altamente restringido.

9. Hipervisores más usados

En este apartado se comentarán algunos hipervisores conocidos para sistemas embebidos.

9.1. Jailhouse

Jailhouse es un hipervisor de Tipo 1 que está diseñado para poder trabajar con Linux, aplicaciones bare-metal o sistemas huésped modificados. Se trata de sofware libre publicado por Siemens en 2013.

La idea detrás de este hipervisor es faciltar aislamiento a través de un diseño simple [27]. No emula elementos que no existen, simplemente separa los recursos de harware existentes en celdas. Siempre hay una celda que contiene el Linux y es llamada Root Cell, el resto de celdas contienen sofware dedicado llamado Inmate. Este hipervisor es capaz de soportar las arquitecturas x86, ARMv7, ARMv8 [4].

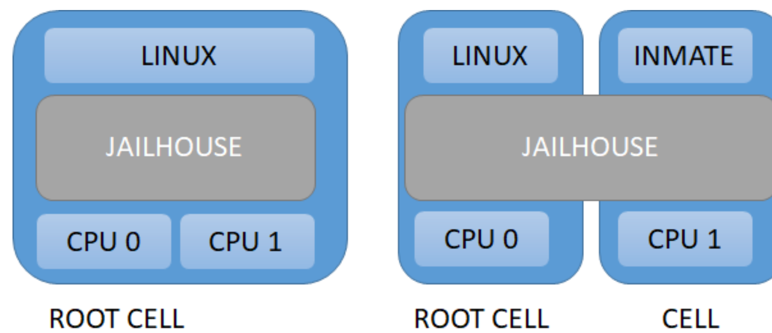


Figura 7: Arquitectura del Jailhouse. A la izquierda está un hipervisor con una única partición y dos cores. A la derecha dos particiones cada una con su propio core.

Sus principales características son:

- Solo asigna un huésped a cada core o conjunto de cores, es decir, que un mismo core no puede tener más de un huésped.
- Todas las particiones son estáticas. No permite reconfiguración.
- Una vez arrancado el hipervisor congela la configuración del harware.
- Favorece la virtualización asistida por hardware frente a las funciones de software.

Para ponerlo en marcha, Jailhouse usa una estructura compuestá por 3 archivos:

- hypervisor_memory: contiene la locaclización en memoria del Jailhouse.
- config_memory: se encarga de apuntar la región en la memoria donde está almacenada la configuración del hardware.
- system: Este archivo se usa para describir la configuración inicial de la celda que almacena el Linux.

Para describir cada una de las celdas también se usa una estructura formada por arrays que contiene información básica de ellas:

- Un bitmap que enumera las CPUs de las celdas.
- Un array que almacena las direcciones físicas, las direcciones del huésped, el tamaño y el acceso a las flags.
- Un array que describe las interrupciones.
- Un bitmap que le da acceso a las I/O a la celda.

Por último, Jailhouse tiene una estructura que usa para describir el hipervisor como un conjunto. Está definido en el contenido de la imagen del binario y contiene la información de la dirección del punto de entrada del hipervior, su tamaño en memoria y número de posibles CPUs.

Actualmente Jailhouse no tiene archivos de configuración legibles para los humanos. Utiliza estos archivos para generar los binarios. No existen controles para saber si una descripción está bien hecha o no, por lo que es posible generar algo inutilizable.

9.2. OPTIMUS

OPTIMUS es un hipervisor diseñado por la universidad de Michigan y la Universidad de Ciencia y Tecnología de Hong Kong. Toda la información sobre este hipervisor ha sido extraida de este único paper [35].

Se trata de un hipervisor de Tipo 1 novel, que tiene como objetivo principal la configuración de FPGAs como aceleradores en la nube.

Características principales de OPTIMUS:

- Programabilidad: OPTIMUS permite unir las direcciones del espacio de memoria entre el software y el hardware, haciendo que se deba confiar en las abstracciones sencillas de la memoria en los espacios de direcciones unificadas.

- Aislamiento: En vez de centrarse en el aislamiento de la Dynamic RAM (DRAM) de la FPGA como otros hipervisores, OPTIMUS tiene en cuenta que su aislamiento de memoria se realiza en presencia de aceleradores Direct Memory Access (DMA), ya que da un asilamiento limitado para la virtualización por hardware. OPTIMUS proporciona un fuerte aislamiento DMA dentro de un solo espacio de direcciones I/O Memory Management Unit (IOMMU).

- Escalabilidad: Para conseguir escalabilidad OPTIMUS usa un árbol multiplexador jerárquico por defecto.

- Eficiencia: Este hipervisor busca conseguir que el ancho de banda de cada acelerador virtual sea lo más cercano posible al ancho de banda total de la FPGA y busca minimizar la latencia agregada por el mismo.

- Equidad: OPTIMUS logra dar a cada acelerador la misma cantidad de ancho de banda en tiempo real a la hora de trasmitir datos.

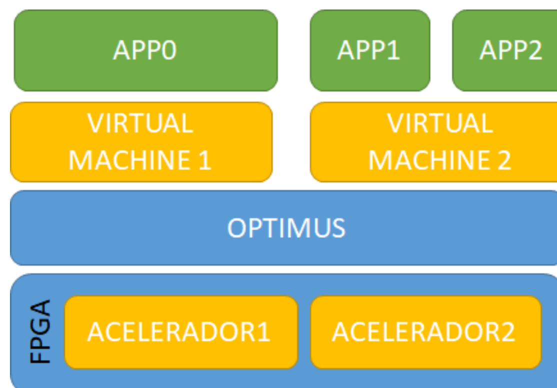


Figura 8: Arquitectura de OPTIMUS.

Este hipervisor configura la FPGA de memoria compartida como un conjunto fijo de aceleradores ofreciendo multiplexación espacial, siendo capaz de expandir su escalabilidad compartiendo un acelerador físico entre varios aceleradores.

9.3. Ker-ONE

Se trata de de un hipervisor de Tipo 1. Ker-ONE sigue el principio de mínima autoridad y baja complejidad, y está diseñado sobre la base de las siguientes suposiciones [52]:

- Solo se consideran arquitecturas mono-core.
- Ocupa principalmente OS simples.
- Todas las tareas críticas se ejecutan en un único huésped, de esta manera Ker-ONE comparte el anfitrión con uno de los huéspedes.

La arquitectura del Ker-ONE está compuesta por un Microkernel con privilegios de supervisor y un entorno de nivel de usario.

El Microkernel se encarga de progrmar, manejar la memoria y las comunicaciones entre las máquinas virtuales, entre otras funciones. Se busca que el Microkernel sea lo más pequeño posible para reducir los ataques al sistema.

El entorno de usuario está compuesto por drivers, sistema de archivos, bootloders...

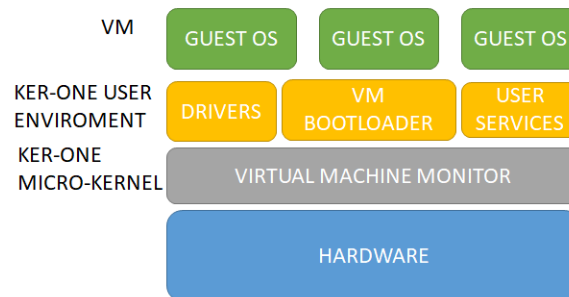


Figura 9: Arquitectura del Ker-ONE

Ahora, se comentarán algunos recursos del sistema:

Memoria: Ker-ONE permite virtualizar la memoria bajo 3 niveles:

- host: para el hipervisor.
- guest Kernel: para los Kernels del sistema operativo del huésped.
- guest use: para las aplicaciones.

Con este sistema de niveles se garantiza que el OS Kernel del huésped está protegido de las aplicaciones del usuario.

Interrupciones: Para la gestión de las interrupciones Ker-ONE genera registros virtuales que son parecidos a los físicos y, de está forma permite mantener las interrupciones originales del OS huésped.

Comunicación entre particiones: Para reducir la complejidad Ker-ONE utiliza un sistema asíncrono de comunicaciones para agilizar la comunicación entre las máquinas virtuales. Esto se logra compartiendo una página de memoria accesible para las dos partes.

Capacidad en tiempo real: Ker-ONE ha sido diseñado para ejecutar RTOS y varios GPOSs haciendo que las tareas de RTOS, sean consideradas críticas debido a sus limitaciones en tiepo real.

9.4. Mini-NOVA

Mini-NOVA es una versión simplificada del mirco-hypervisor NOVA, que ha sido porteado a la arquitectura ARM Cortex A9. El principio sobre el cual está diseñado es el de reducir su complejidad para, de este modo, reducir la sobrecarga, tener un Task Control Block (TCB) más pequeño y aumentar la seguridad, lo que lo hace más flexible y portable a otros dispositivos. Está basado en la plataforma de Zynq 7000, lo que incluye un dual-core ARM y una FPGA, permitiendo Dynamic Partial Reconfiguration (DPR) [50].

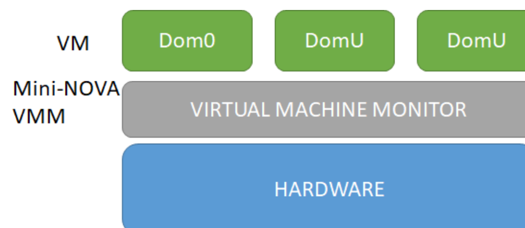


Figura 10: Arquitectura del Mini-NOVA.

Por cada huésped se inicializa una máquina virtual (VM), corriendo en un dominio de usuario aislado. Se usa un monitor de máquina virtual para crear el entorno virtualizado para cada máquina virtual. Este entorno provee las siguientes propiedades:

Virtualización de la CPU: Mini-NOVA se encarga de guardar los estados del hardware que están siendo usados en cada VM mediante una estructura de datos. Esta estructura actúa como una CPU virtual. Mini-NOVA permite que la máquina virtual acceda a los recursos para programarlos, a excepción del hardware que afecta al Microkernel o al de otras máquinas virtuales.

Virtualización de las interrupciones: Por cada VM se genera un Generic Interrupt Controller (GIC) que genera diferentes tipos de interrupciones. Estas interrupciones son atrapdas por el Kernel y son distribuidas por el virtual GIC (vGIC) de la VM actual. Cada vGIC está asociado a una máquina virtual y guarda una lista de los estados y las interrupciones usadas por la VM.

Administración de la memoria: La gestión de la memoria es controlada por el memory management unit (MMU). El MMU aplica un sistema para las direcciones virtuales de tipo tabla con 3 niveles:

- Nivel privilegiado.
- Acceso Completo.
- Sin acceso.

Cada VM tiene su propio espacio de memoria aislado. Cada VM tiene su propia tabla que solo mapea la parte de espacio de memoria en el que Mini-NOVA lo aloja.

Programador: Todos los huéspedes están organizados en dos grupos de ejecución: la cola de ejecutados, y la de suspendidos. La cola de ejecutados está compuesta por los OS/aplicaciones que se están ejecutando. Los suspendidos no tienen que estar necesariamente programados para evitar malgastar recursos de la CPU.

Cada VM está ejecutada con un nivel de prioridad, consiguiendo de esta manera que una aplicación que tiene características de tiempo real o unas limitaciones más apretadas se puedan ejecutar inmediatamente.

9.5. PikeOS

Aunque el nombre PikeOS pueda confundirse con un OS, se trata de otro hipervisor comercial de Tipo 1 desarrollado por la compaña SYSGO [5], basado en la separación del Kernel. Puede tener muchos tipos de particiones con diferentes OS, llamados GuestOS, y aplicaciones.

PikeOS es un clon comercial de la familia L4 Microkernel. Gracias a su arquitectura de seguridad multicapa de datos, separación de las aplicaciones y control de la información.

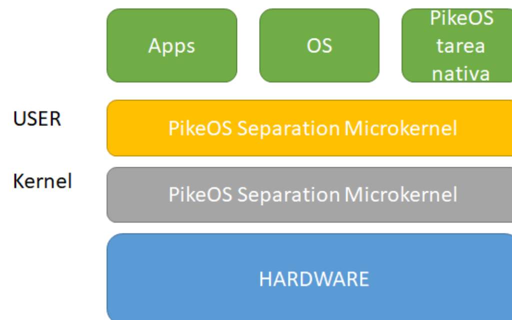


Figura 11: Arquitectura del hipervisor PikeOS.

Al tratarse de un hipervisor comercial no hay muchos datos sobre su funcionamiento más allá de su arquitectura y sus características principales:

- Previene que se propaguen los errores de una aplicación a cualquier otra parte del sistema.
- Incorpora un sistema de encriptación y verificación binaria.
- El sistema tiende a favorecer las aplicaciones más críticas sin dejar de lado las tareas menos estrictas.
- Tiene un monitoreo de la salud del sistema con el que intercepta los errores de las aplicaciones y del hardware.

9.6. Codezero

Codezero es un Microkernel L4 escrito en scratch. Su objetivo son los sistemas embebidos y su propósito principal es el de actuar como un hipervisor embebido seguro [30]. La filosofía bajo este proyecto es la de crear el microkerel más simple, que sea genérico y aplicable a aplicaciones muy diferentes [1]. Es interesante mencionar que se trata de un hipervisor distribuido bajo GNU Licencia Pública General versión 3.

Se trata de un hypervisor de Tipo 1 usando la típica abstracción del hardware.

La virtualización con Codezero está implementada a traves de contenedores. Cada contenedor está aislado de los otros y tiene su propio entorno de ejecución y su propio set de recursos. Usa 3 formas de Inter-Process Communication(IPC): short IPC, full-IPC(256 bytes) y extended IPC(2048 bytes). Codezero ha sido diseñado para sistemas embebidos y soporta procesadores multicore y diseños ARM.

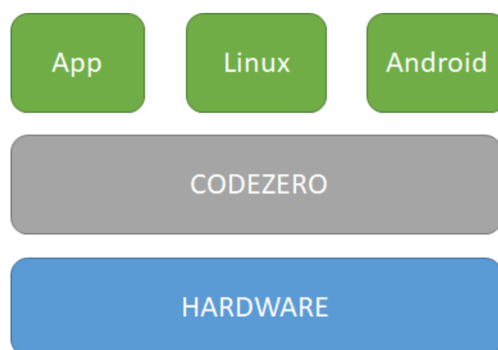


Figura 12: Arquitectura del Codezero.

Codezero está formado por los siguientes directorios [39]:

- Loader: Es el directorio que contiene el bootloader y la libreria ELF y es el último en cargarse dentro del sistema.
- Conts: Se trata de los directorios que contienen al espacio de usuario.
- SRC: En estos directorios están incluidos los archivos necesarios para ejecutar el Microkernel.

9.7. XEN

XEN comenzó siendo parte del proyecto Xenoverse de la universidad de Cambridge en 1990. Fue publicado como código abierto en los 2000 y fue respaldado por la fundacion Linux en 2013. Se ha convertido en la solución por defecto para los OS basados en Linux. Se trata de un hipervisor gratuito y con una comunidad activa que desarrolla nuevas funciones [54].

Se trata de un hipervisor de de Tipo 1 que divide sus OS por dominios. Utiliza un dominio especial llamado Dom0 para controlar el funcionamiento del hipervisor. Este dominio proporciona una estructura especializada para el hipervisor y su uso es obligado para que funcione correctamente. Dom0 usa software privilegiado dentro de su Kernel así como, controladores especiales que pueden acceder al hardware.

El resto de dominios son llamados colectivamente DomU. Pueden ser desde OS de alto nivel (como Linux), OS en tiempo real (como FreeRTOS) o incluso código nativo [7].

XEN ha añadido acceso directo a las I/O para un dominio, con el que es capaz de dar un rendimiento casi nativo, sin embargo, este método no cumple el concepto de virtualización, además de que puede generar problemas en la seguridad [49].

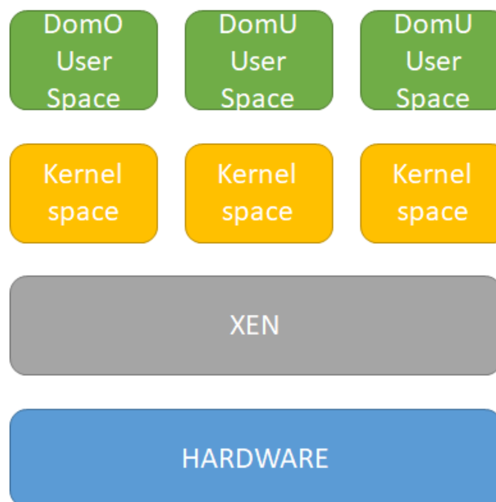


Figura 13: Arquitectura del XEN.

9.8. Rodovisor

Se trata de un hipervisor de Tipo 1 que soporta virtualización completa junto con capacidades en tiempo real y es capaz de ejecutar varias aplicaciones. Estaba basado en el estándar de aviación ARINC-653, de este estándar se hablará más en el apartado 9.11, oponiendose al resto de hipervisores Rodovisor está orientado a objetos.

Rodovisor garantiza un aislamiento espacial y temporal entre cada partición, de forma que cuenta con áreas separadas de memoria y tiempo preasignado para las actividades de ejecución, protegiendo al hipervisor de las VMs y a cada VM de las demás [17].

Rodovisor se encarga de manejar las interrupciones para después despacharlas a la máquina virtual que corresponda. Este hipervisor divide las interrupciones en 3 clases:

- Interrupciones generadas por el hardware gestionadas primero por el hipervisor hasta las VMs.
- Interrupciones relacionadas con el estado de la CPU detectadas por alguna VM que son gestionadas desde el huésped OS al hipervisor y de nuevo a la VM.
- Interrupciones virtuales que se reenvían a las VM que contienen los dispositivos virtuales.

Como caso especial de las interrupciones está la virtualización del reloj, que para mantener el aislamiento temporal usa dos relojes físicos, uno a nivel de hipervisor y otro a nivel de partición.

Rodovisor puede ejecutar diferentes particiones de software como GPOS, RTOS o bare-metal aplicaciones. Así mismo, en su propio espacio de direcciones ejecuta un espacio de usuario de controlador de dispositivo.

9.9. Hipervisor Tipo 0

Se trata de un hipervisor particular, pues es el único de la lista que pertenece al grupo 0 y no tiene nombre. El hipervisor no se ejecuta directamente sobre el hardware sino que pertenece al hardware, por lo que requiere su propio sistema de microprocesador, memoria, GPIO, etc.

Este hipervisor implementa un sistema de programación basado en franjas de tiempo, en donde un temporizador genera interrupciones para el hipervisor [29].

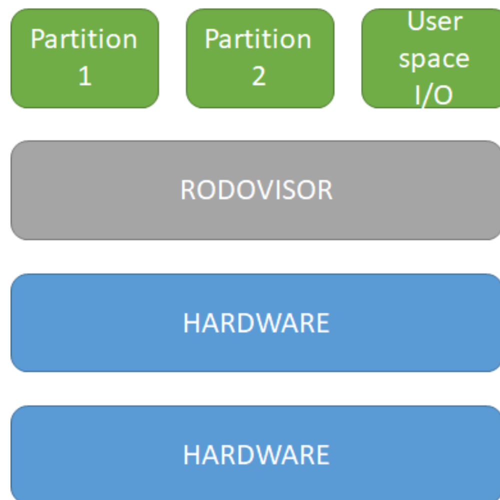


Figura 14: Arquitectura del Rodovisor.

No es posible acceder a las llamadas al sistema a partir de las interrupciones, por lo que se ha implementado un Hypervisor Auxiliary Module (HAM). El HAM se utiliza también para configurar el MMU de la aplicación y maneja el saludo con el hipervisor.

Comparándolo con los hipervisores de Tipo 1, sus mayores ventajas son que aumenta la latencia de las llamadas al sistema y que puede ejecutar reconfiguraciones de particiones en paralelo reduciendo el tiempo en el cambio entre particiones.

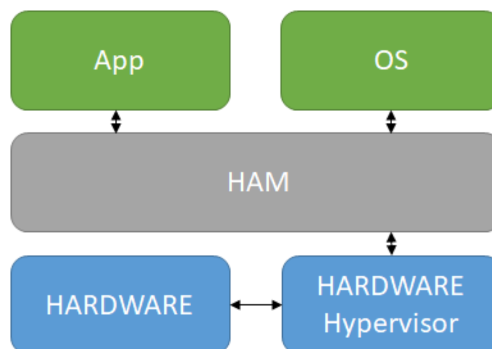


Figura 15: Arquitectura del hipervisor de Tipo 0.

9.10. OKL4 microvisor

OKL4 es un hipervisor de Tipo 1, está diseñado en base a la idea de fusionar lo mejor de los hipervisores y los Microkernels, buscando proveer de una plataforma de virtualización con la eficiencia del mejor hipervisor y con la capacidad de dar soporte a cualquier sistema [26].

El microvisor está diseñado para modelar hardware lo más real que pueda, teniendo las siguientes características:

- El microvisor debe poder virtualizar varias máquinas virtuales en las que el OS invitado pueda ejecutar programas. Puede ejecutar OS completos, standalone o bare-metal applications.
- Para compartir la memoria entre las particiones, OKL4 usa un sistema de ficheros bidireccional FIFO haciendo que el sistema sea muy fácil de manejar. Sin embargo, este sistema tiene varias limitaciones como que es un único canal, tiene poca eficiencia y carece de un mecanismo de información-respuesta [48].
- Las I/O las gestiona como registros de componentes virtuales y como interrupciones virtuales.
- Las particiones de este sistema usan un mecanismo llamado Interprocess Communication (IPC) que resulta muy eficiente para que las particiones se comuniquen y cooperen, el IPC se comporta como un camino de I/O.

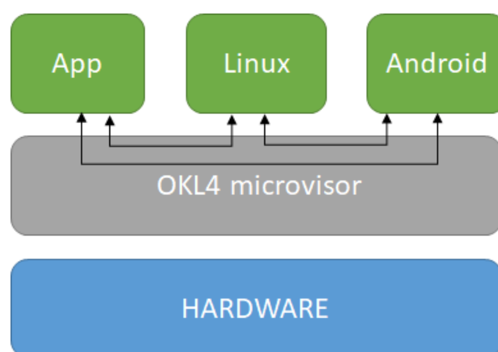


Figura 16: Arquitectura del OKL4.

9.11. Xtratum Next Generation (XNG)

En está sección se van a describir los aspectos más importantes del hipervisor XNG de una manera más detallada que en los anteriores, ya que este será el hipervisor que se va a usar para el caso práctico.

Toda la documentación de esta sección está basada en la documentación oficial y la documentación de Fentiss [36] [37] [6].

El hipervisor XNG, al igual que muchos de este trabajo, es un hipervisor de Tipo 1. Es un hipervisor que es capaz de correr varias particiones a la vez aunque estas tengan diferentes OS.

XNG está parcialmente basado en el estandar de aviación ARINC-653. Este estandar no está pensado para definir como debe operar un hipervisor, pero muchas de sus partes están ligadas a la funcionalidad proporcionada por un hipervisor.

El estandar de aviación ARINC-653 se usa como software de desarrollo para sistemas a bordo, ya sean aviones o naves espaciales, que tienen un software muy crítico. Este estandar define cómo debe funcionar un anfitrión cuando tiene numerosas aplicaciones con diferentes softwares con sus respectivos niveles de severidad en un mismo hardware. Define el conjunto de funcionalidades que un RTOS debe tener, así como los procedimientos para demostrar el cuplimiento del comportamiento de RTOS [10].

Arquitectura

En este capítulo se introducir brevemente la estructura de XNG.

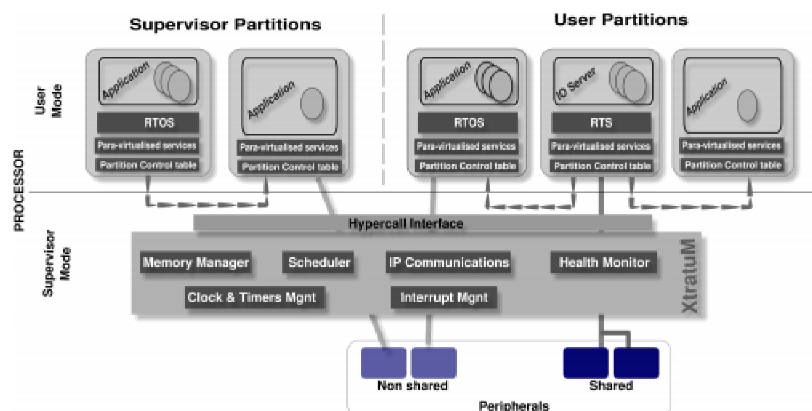


Figura 17: Arquitectura del XNG.

Los principales componentes de su arquitectura son:

- **Hipervisor:** XNG provee servicios de virtualización a las particiones. Debe ejecutarse en modo supervisor y se encarga de

virtualizar la CPU, memorias, interrupciones y algunos periféricos. Los componentes intrínsecos son los siguientes:

- Administrador de memoria: XNG se encarga de dar la memoria necesaria a cada partición. Utilizando los mecanismos de hardware puede garantizar aislamiento.
- Programador: Las particiones son programadas mediante una política de programación cíclica.
- Administrador de interrupciones: Se puede configurar al XNG para que permita que las interrupciones puedan ser atendidas por una única partición. Además, XNG aumenta la cantidad de interrupciones con interrupciones virtuales (vIRQ) que son generadas por distintos eventos relacionados con los servicios del hipervisor.
- Administrador de reloj y tiempo.
- Comunicador de particiones internas: Las comunicaciones entre las particiones pueden hacerse mediante dos métodos. El primero, llamado simplemente puertos de comunicación de las particiones, permite enviar mensajes de datos finitos y la comunicación se realiza mediante dos puertos:
  - Sampling Port: El mensaje se mantiene en el puerto hasta que se sobreescribe.
  - Queing Port: El mensaje se almacena en una cola hasta que es leído por otra partición. Funciona como una cola FIFO.

  El segundo, llamado Inter-Partition Virtual Interrupt (IPVI), permite enviar vIRQs a otras particiones.
- Monitoreo de la salud: Esta parte de XNG está encargada de detectar y reaccionar a eventos extraños o estados. Estos eventos son capturados por la partición y comunicados al hipervisor. Así, XNG puede intentar resolver el error o, por lo menos aislarlo.
- Rastreo de instalaciones: XNG provee unos mecanismos que retienen las trazas generadas por las particiones y el propio hipervisor, facilitando el debugging durante la fase de desarrollo.

- **Apliaction Programming Interfaces (API):** Define servicios para-virtuaizados del XNG. El acceso a estos servicios tiene que hacerse a través de hypercalls.

- **Particiones:** Una partición es un entorno donde se está ejecutando un programa o OS manejado por el hipervisor. Las particiones consisten en uno o más procesos, que comparten acceso a los recursos dependiendo de la aplicación.

Estados del Sistema

Boot state: En este estado se carga la imagen del XNG en la memoria principal y se transfiere el control al hipervisor. En este momento el programador no se ha habilitado y las particiones no se han ejecutado.

Normal state: Al final del estado de boot, el hipervisor comienza a ejecutar el código de las particiones y el programador se habilita.

Halt state: A este estado se entra si el monitor de salud detecta un error o si una de las particiones invoca una hypercall. En este estado el programador y las interrupciones se deshabilitan. El procesador entra en un loop sin fin, la única forma de salir del loop es a través de un reset del hardware.

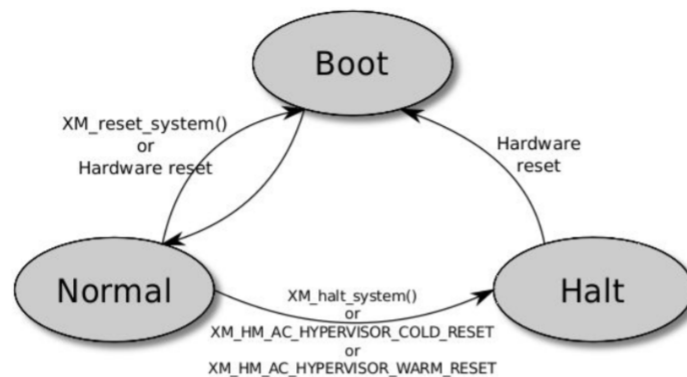


Figura 18: Estados del hipervisor XNG y sus transiciones.

Estados de las particiones

Una vez que el XNG se encuentra en el estado normal, las particiones arrancan.

Boot: Es el estado en el que comienzan todas las particiones y es el estado en donde la partición se inicializa.

Normal: Una vez la partición se ha inicializado pasa al estado normal. Este estado se puede subdividir a su vez en 3 estados.

- Ready: La aplicación está preparada para ejecutar el código pero no está en su franja temporal.

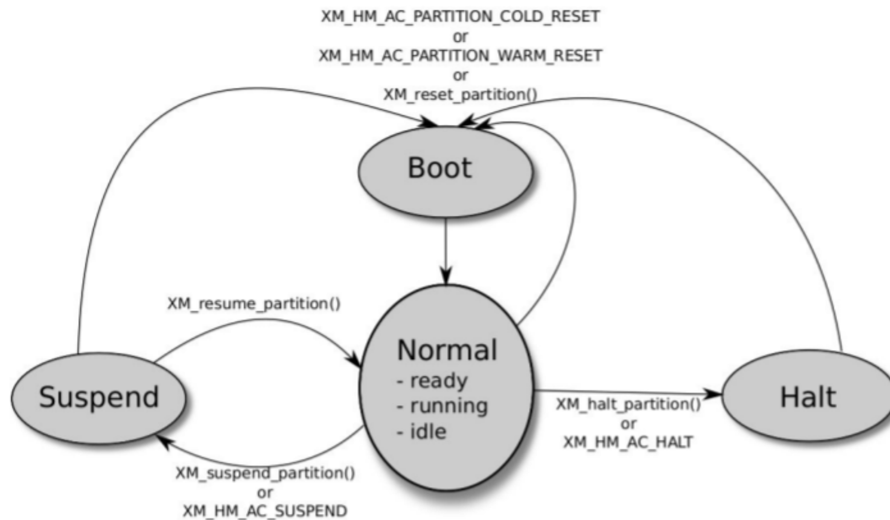


Figura 19: Estadaos de las particiones

- Running: El procesador está corriendo la aplicación.
- Idle: Si la partición no desesa usar su intervalo de tiempo asignado puede ceder al procesador y esperar una interrupción o el próximo intervalo de tiempo. En este estado solo se entra a través de una hypercall.

Halt: En este estado todos los recursos de la partición son liberados. No es posible volver al estado normal. A este estado puede ir la propia partición a través de un error o puede ser llevado a este estado a través de la partición de sistema.

Suspended: En este estado la aplicación no puede recibir interrupciones ni programarse. Todas las interrupciones quedan como pendientes. Si la partición vuelve al estado normal, las interrupciones pendientes se mandarán a la aplicación. La aplicación puede volver al estado ready usando las hypercalls.

Particiones de sistema

XNG define dos tipos de particiones: normales y de sistema. Algunas hypercalls no pueden ser llamadas por una partición normal. A diferencia de las normales, las particiones de sistema son capaces de monitorear el estado del sistema y de otras particiones, pero una particion de sistema sigue sin tener acceso directo al hardware.

Programador de particiones

XNG usa un programa de particiones de forma cíclica fija. No permite que una partición use el procesador por más tiempo que el programado para evitar el detrimento de las otras particiones. El tiempo de uso del procesador permitido para una partición está definido en las XNG Configuration Files (XCF) durante la fase de diseño. Si la partición tiene varias actividades que concurren, es la partición la que debe implementar un sistema para organizar su programación. XNG no se hace cargo de la programación interna de cada partición.

Las XCF son tablas creadas durante el diseño del sistema que definen la localización de los elementos de cada partición y el hipervisor durante la ejecución. Esta configuración solo se puede acctualizar reiniciando el hipervisor.

Beneficios

Los beneficios por usar XNG son los siguientes:

- El aislamiento de los componentes del software refuerza el aislamiento de fallos. Esta propiedad es la clave para el desarrollo de sistemas críticos de seguridad.
- Permite la integración de distintos tipos de aplicaciones en el mismo hardware reduciendo el peso y volumen.
- La integración de diferentes sistemas no propaga criticidad del más crítico al menos crítico.
- La reutilización de las particiones es posible.
- La reutilización del software legado es posible.

10. Tabla de hipervisores

La siguiente tabla es un resumen de los hipervisores vistos anteriormente:

Hipervisor	Arquitecturas	Huesped OS	Licencia
Jailhouse	x86, ARM	Linux, bare-metal applications, adapted OS	GPL
OPTIMUS	ARM	Linux	Open source
Ker-ONE	ARM	µC/OS, FreeRTOS	Open source
mini-NOVA	ARM	RTOS	Open-source
PikeOS	PowerPC, x86, ARM, MIPS, SPARC, SuperH	PikeOS native, Linux, POSIX, AUTOSAR, Android, RTEMS, OSEK, ARINC 653 APEX, ITRON	Comercial
CODEZERO	ARM	Linux	GPL
XEN	x86, ARM	High level OS, real-time OS, bare-metal code	GPL
Rodovisor	ARM	GTOS, RTOS, bare-metal code	GPL
Tipo 0	ARM	?	?
OKL4	x86,ARM	Linux, bare-metal code	Commercial
XNG	x86, ARM	Linux, XRE, RTOS, Lithos, RTEMS	Comercial

Cuadro 1: Tabla de los hipervisores. En el caso del Tipo 0, aparece con dos ”?” debido a que no se tiene suficiente información sobre él.

11. Caso práctico

En este apartado se va a probar el hipervisor Xtratum con una tarjeta ZYBO, para ello se implementará una red TCP y se medirá su velocidad tanto con el hipervisor como sin él. Además, se instalarán dos versiones de Linux dentro de la ZYBO para facilitar la instalación de IPERF.

La instalación del hipervisor y los OS se harán sobre una tajeta MicroSD. Tras esto, se instalará el IPERF y se harán las pruebas correspondientes usando un programa de comunicación por puerto serie.

Primero, se explicará como crear las dos targetas microSD y después se haran las pruebas.

11.1. Prerrequisitos

Antes de la puesta en marcha de cualquiera de los sistemas es necesaria la instalación de las siguientes librerías y programas. Para esta práctica el OS que se usará para construir ambas tarjetas microSD es Linux.

- Librerías:
    - libncurses5-dev
    - libncursesw5-dev
    - u-boot-tools
    - device-tree-compiler
    - flex
    - libssl-dev
    - bison
- Programas:
    - Vivado 2017.4 incluir SDK.
    - Añadir la placa ZYBO para que la reconozca el Vivado [2].
    - Algún programa de comunicación por puerto serie como Putty.

11.2. Resumen de la configuración

Para la configuración del Linux en la ZYBO se seguirá el siguiente esquema:

1. Configurar las variables de entorno para que el cross compile pueda compilar.

2. Construir el U-Boot.
3. Construir el Kernel de Linux.
4. Crear el FSBL y la imagen del Boot utilizando vivado.
5. Construir el Device Tree Blob.
6. Modificar el espacio de la tarjeta microSD.
7. Puesta en marcha del Linux para corroborar su funcionamiento.
8. Intsalar Iperf.

Para la configuración del hypervisor XNG se seguirán los siguientes pasos.

1. Configurar las variables de entorno de los archivos del hipervisor del Linux-bsp.
2. Prerequisitos para adaptar la ZYBO a los ejemplos.
3. Construir los ficheros ps7, FSBL y dtc usando vivado.
4. Construir el Kernel de Linux.
5. Instalar por JTAG.
6. Modificar el espacio de la tarjeta microSD.
7. Puesta en marcha de la partición del XNG.
8. Modificar los archivos rootfs para instalar iperf.

11.3. Puesta en marcha de Linux

Configuración de variables

Para que el Cross Compiler pueda ejecutarse es necesario crear una variable global. Dentro de un terminal de Linux se debe escribir el siguiente código.

```
export CROSS_COMPILE =arm-linux-gnueabihf-
```

En el caso de que no se use esta versión de Vivado es posible que la variable sea diferente.

También es necesario configurar correctamente la variable de entorno $PATH, y, para ello, es necesario ejecutar el siguinente comando.

```
source <Directorio de Xilinx>/Vivado/<Version>/settings.sh
```

Construir el U-Boot

Para cosntruir el U-Boot primero, hay que descargarlo en la dirección que aparece en el apéndice A, después es necesario descomprimirlo y navegar hasta el directorio llamado u-boot-xlnx.

Para crear el arrancador hay que ejecutar los siguientes comandos:

```
make zynq_zybo_config
make
```

Si alguno de los comandos no se ejecuta correctamente, es necesario revisar que la variable CROSS_COMPILE esté bien escrita, e incluso podría ser necesario escribir el path completo.

Una vez completado correctamente el paso anterior, dentro del directorio Tools se creará un archivo llamado Mkimage. Es necesario que este archivo sea accesible cuando se construya el Kernel, es por eso, que a la variable $PATH se le añade el path del Kernel.

```
cd tools
export PATH=$PATH:‘pwd‘
```

También es necesario cambiar la extensión del archivo U-Boot, de forma que el Xilinx SDK pueda usarlo, ejecutando los siguientes comandos:

```
cd ..
mv u-boot u-boot.elf
```

Construir el Kernel de Linux

Primero, es necesario descargar el directorio del enlace que aparece en el apéndice A y, después, hay que descomprimir el archivo y navegar hasta el directorio linux-xlnx-zymp-dt-fixes-for-4.10.

Posteriormente, se descarga el fichero de configuración xilinx_zynq_defconfig que aparece en el apéndice A y se copia en arch/arm/configs.

Para construir el Kernel es necesario configurarlo ejecutando los siguientes comandos:

```
make ARCH=arm xilinx_zynq_defconfig
make ARCH=arm menuconfig
```

No es necesario modificar la configuración del menu.

Para construir la imagen del Kernel hay que escribir el siguiente comando:

```
make ARCH=arm UIMAGE_LOADER=0x8000 uImage
```

Cuando el proceso termine el Kernel debe quedar compilado en uImage, que se encuentra en el directorio arch/arm/boot/.

Para ejecutar los módulos del Kernel y montarlos en el sistema de archivos:

```
make ARCH=arm modules
sudo make arch=arm INSTALL_MOD_PATH=mnt/ modules_install
```

Crear el FSBL y la imagen del Boot

Primero, hay que ejecutar Vivado, crear un nuevo proyecto y elegir ZYBO como placa.

Segundo, hay que crear un nuevo diseño y añadir Zynq Processing System, y seguidamente pinchar sobre Run Block Automation para añadir el Processor System Reset. Clickar sobre Run Connection Automation y conectar los dos bloques como aparece en la imagen 20.

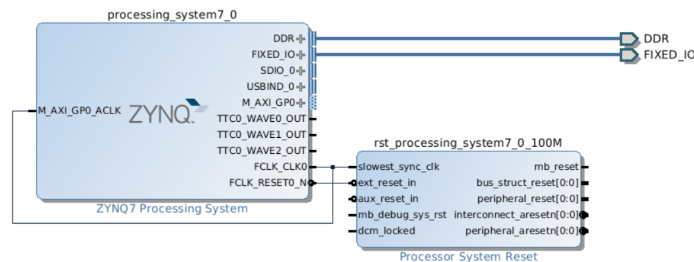


Figura 20: Diseño de bloques para la configuración de Linux.

Tercero, en la pestaña de fuentes, click derecho y se le da a Create HDL Wrapper. Después, hay que generar el Bitstream y, una vez Vivado termine de generarlo, hay que exportar el hardware, incluir el Btstream y lanzar el SDK.

Cuarto, Se crea un new application project, se le nombra fsbl y click a next. Hay que elegir Zynq FSBL y clickear Finish. En el explorador del proyecto click derecho en la carpeta llamda fsbl y elegir Create Boot Image. En la sección de Boot image partitions clickear Add, navegar por los directorios de U-Boot y clickear en el archivo u-boot.elf que antes se generó. Click en Create Image y ya se tendria el archivo BOOOT.bin

Construir el Device Tree Blob

El Device tree blob se tiene que crear usando el SDK con el mismo hardware y bitstrea que se han exportado.

Se descarga el devicetree.dts del enlace que aparece en el apéndice A.

En el panel del SDK clickar en Xilinx Tools y Repositories. Se despliega una pantalla hay que clickar en el boton New de Local Repositories. Navegar hasta el directorio de device-tree-xlnx y pulsar OK. Esto permite que el directorio de device-tree sea accesible para el SDK.

Después, clickar en File>New>Board Support Package. En el área de Board Support Package OS hay que elegir device_tree y Finish. Aparece una nueva pantala en la que solo hay que dar click a OK.

Click derecho en el directorio del proyecto que se llama device_tree_bsp_0 e importar. Click en General>File System, navegar por el directorio del devicetree.dts y clickar OK. Elegir el fichero y pulsar Finish.

Por útimo, es necesario usar el archivo que acabamos de generar navegando por los directorios <project>/<project>.sdk/device_tree_bsp_0/ y compilar el devicetree.dtb usando el siguiente comando:

```
dtc -I dts -O dtb -o devicetree.dtb devicetree.dts
```

Crear una Tarjeta SD

Para Formatear la tarjeta SD se va a usar una de las herramientas mas comunes: GParted.

Es necesario que a tarjeta este dividida en 3 partes:

- 4MB sin asignar.
- Un area formateada con fat16 o fat32 etiquetada como BOOT.
- Un area formateada como ext4 con por lo menos 1GB etiquetada como ROOT_FS

En el BOOT se guardan los siguientes archivos:

- BOOT.bin
- devicetree.dtb
- uImage

En ROOT_FS es necesario crear un sistema de archivos para Linux. Para ello se puede descargar la versión de Linaro 17.02 del siguiente enlace que aparece en el apéndice A.

Es necesario descomprimirlo y transferir los archivos de manera correcta para mantener los permisos.

```
sudo tar xf linaro-jessie-developer-20161117-32.tar.gz –strip-components=1
-C <path>/ROOT_FS/
```

Encender la ZYBO

Para poder arrancar la ZYBO es necesario itroducir la tarjeta SD dentro de la ZYBO, colocar el jumper en modo de programación para SD y conectar la ZYBO al ordenador para comezar las comuncaciones se usa un programa de comunicacion por puerto serie.

Se puede verificar que está correctamente lanzado el Linux ejecutando algún comando.

Instalación de Iperf

Para instalar Iperf basta con escribir en la terminal lo siguiente:

```
sudo apt-get install iperf
```

11.4. Puesta en marcha de Xtratum

La instalación del hipervisor Xtratum se puede hacer tanto por el JTAG como por la tarjeta SD [11]. Al igual que para instalar Linux, es necesario instalar las librerías y los programas mencionados en los prerequisitos. A la hora de la descarga el Xtratum viene dividido en dos carpetas. Una donde está el hipervisor y sus ejemplos, que en este trabajo se llama **xtratum_path** y otra donde se ubica el linux y sus ejemplos de instalación que se llama **linux_path**.

Modificación de archivos

Para que se pueda compilar el Kernel es necesario modificar dos archivos. El primero se encuentra en los archos de linux_path y se llama env.sh. En él se modificarán las variables de entorno como PATH, XNG_PATH y CROSS_COMPILER para hacerla coincidir con las del ordenador. De la misma manera, es necesaro modificar el fichero xtratum_path/cfg/xng_cfg.mk para modificar las variables TARGET_CCPATH y TARGET_CCPREFIX.

Prerequisitos para la instalación del hipervisor

Como se va ha utilizar una placa distinta a la de los ejemplos que aparecen en las carpetas del hipervisor es necesario hacer algunas modificaciones.

Primero se crea un nuevo ejemplo basado en los ejemplos existentes. Por ejemplo se puede copiar y pegar el de Zedboard_example y renombrarlo como Zybo_example.

Segundo, dentro de XNG_confi/xml/hypervisor.xml modificar la frecuencia del osciador a 50.0MHz y modificar la dirección de la UART a 0xE0001000.

Tercero, adaptar los componentes para esta placa.

- Se necesita un device-tree específico para la placa por lo que FENTISS ha proporcionado uno para cada tipo de arranque: zynq-zed-RAMDISK para el arranque por JTAG y zynq-zed-EXT4 para el arranque por microSD.
- Es necesario crear un nuevo bitstream, ps7 y FSBL.elf.

Para generar los ficheros de bitstream, ps7 y FSBL.elf es neceario seguir los siguientes pasos:

1. Abrir la ubicación donde se encuentra Vivado y ejecutar los siguientes comandos para lanzarlo:

```
cd /opt/Xilinx/Vivado/2017.4
source settings64.sh vivado
```

2. Hay que crear un nuevo proyecto eligiendo ZYBO como la placa sobre la que se va a trabajar.
3. Se crea un nuevo diseño de bloques. En el se añaden los componentes: ZYNQ7 Processing System y AXI Timer.
4. Se hace doble click sobre el ZYNQ7 Processor System y se habilita el acceso no seguro al bus AXI. Clickando sobre PS-PL Configuration>AXI Non Secure Enablement = 1; Desplegar GP Master AXI Interface y validar las dos líneas de M AXI GP0 interface y M AXI GP1 interface. Por último, se clicka en OK para salir de esa ventana.

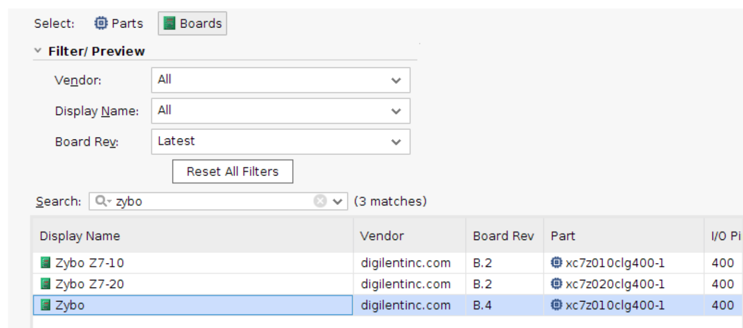


Figura 21: Selección de la placa ZYBO al inicio de la creación del proyecto en vivado.

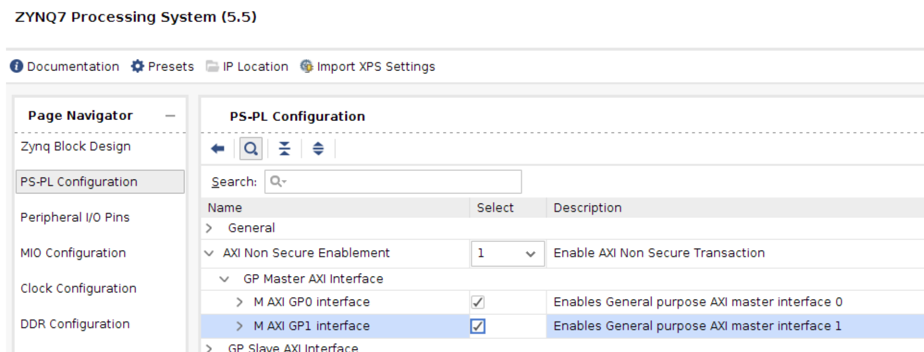


Figura 22: Línea que habilita el AXI no seguro en el ZYNQ7 Processing System

5. Run Block Automation y Run Connection Automation. El resultado deberia ser como en el de la figura 23.
6. Click derecho sobre el diseño y HDL wrapper.
7. Generar el Bitstream.
8. Exportar el diseño de hardware con el bitstream. File>Export>Export Hardware. En este momento ya están los archivos de bitstream y ps7, para ejecutar el XNG via JTAG.
9. Lanzar el SDK. File>Launch SDK adjuntando el bitstream.
10. Crear un nuev proyecto: File>New>Application Project.
11. En esta ventana nueva se le da un nombre al proyecto y NEXT, en la siguiente se elige Zynq FSBL y FINISH.
12. Es necesario modificar uno de los archivos para que la dirección del start address de inicio del XCF esté en el registro r0 del procesador

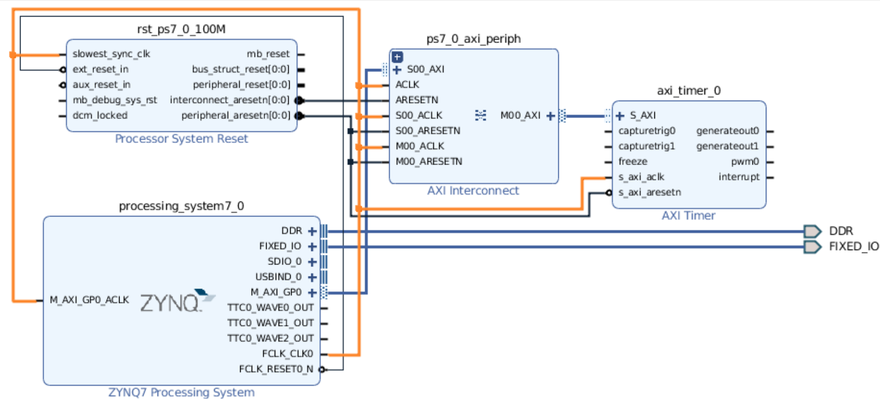


Figura 23: Diseño de bloques con el ZYNQ7 processor system y el AXI Timer.

antes de que empiece.

13. Hay que habilitar el modo Release antes de la construcción del FSBL. File tab>Properties>C/C++build>Manage Configurations>Release>Set Active>Apply y close.

14. Construir el proyecto. Project tab>Build Project.

Con esto ya se tienen los tres ficheros dentro de la carpeta del proyecto.

Construir el Kernel de linux

Una vez modificados los ficheros, hay que cambiar al directorio de linux-xlnx-master. Se cargan las variables de entorno y se aplica la configuración para construir el Kernel de Linux.

```
source ../env.sh
make xilinx_zynq_XNG_defconfig
make
```

Como resultado se obtiene un fichero que es necesario convertirlo en un fichero con extensión .bin

```
arm-none-eabi-objcopy -O binary vmlinux vmlinux.bin
```

El siguiente paso consiste en compilar el divice-tree. Se van a compilar todos los device-tree pero solo se van a usar dos: zynq-zybo-RAMDISK y zynq-zybo-EXT4, el primero se usará para cargar y ejecutar el programa por JTAG y el segundo por la microSD.

```
make dtbs
```

El último paso es cargar el programa que carga el linux.

```
cd ../linux_ldr/
make
```

Como salida de este proceso se han obtenido 3 ficheros:

- El Kernel de Linux (vmlinux.bin) que se encuentra en la carpeta actual.
- El device-tree (zynq-zybo-RAMDISK, zynq-zybo-EXT4) que está en la carpeta arch/arm/boot/dts.
- El cargador de Linux (ldr.bin) que se encuentra en ../linux_ldr/.

Instalar por JTAG

Para cargar el programa a través del JATG es necesaario verificar que el jumper está colocado en la posición correcta. Se alimenta la placa por el puerto USB.

Es necesario modificar el archivo Zybo_example/JATG/xsdb.tcl con los cambios en los nombres de los archivos que se han hecho como en el del device tree.

Dentro del archivo de linux_path se cargan las variables de entorno.

```
source env.sh
```

Se compila el ejemplo.

```
cd Zybo_example/XNG_conf/
make
```

Se corre el programa a traves del JTAG.

```
cd ../JATG/
xsdb
```

Una vez se han cargado todos los ficheros del XNG se puede abrir un puerto serie, en este caso se usará putty para ver como se carga el Linux y poder interactuar con él.

Instalar por microSD

Para cargar el programa por la tarjeta microSD es necesario igual que antes cargar las variables de entorno de la carpeta linux_path y compilar el ejemplo.

```
source env.sh cd Zybo_example/XNG_conf/
make
```

Se crea el archivo imagen boot(.bif) y se genera el archivo imagen boot.bin

```
cd ../SDCARD/ tclsh generate_boot_image.tcl make
```

Ahora, se inserta la tarjeta microSD en el ordenador y se formatea en dos partes:

- Una parte de 100mb (como mínimo) llamada boot.
- Otra de 500mb (como mínimo) llamda rootfs

El fichero creado antes, llamado boot.bin, se introduce en la partición llamada boot.

```
cp output/boot.bin /media/<users>/boot
```

Se extrae el fichero root file system a la partición rootfs.

```
sudo tar -C /media/<user>/rootfs/ -xzvf ../Vivado_output/rootfs.tar.gz
sync
```

Se introduce la tarjeta SD dentro de la ZYBO. Es necesario colocar correctamente el jumper en la posición para que la placa arranque en modo microSD. Una vez hecho esto, se puede encender la placa. Es necesario tambien habilitar un puerto serie para poder comuncarse con la placa. De nuevo se utilizará putty.

La placa arranca correctamente ya que pide el nombre de usuario y contraseña que en este caso son: USER: root y PASWORD root.

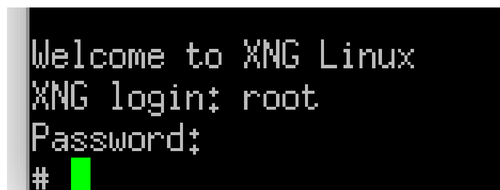


Figura 24: Usuario y pasword del Linux para poder usar Linux de la ZYBO. La linea de Pasword no aparece porque es invisible pero tanto el usuario como el pasword es el mismo: root.

Instalación del Iperf

En este caso, no vale con instalar Iperf usando la linea de comandos ya que está distribución para Linux embebido no tiene los binarios tipicos (ls, wget o mkdir) sino que lleva una versión más ligera de cada uno. Por lo tanto, es necesario modificar los archivos de rootfs para poder instalarlo.

Lo primero que se debe hacer, es descargar y descomprimir el Buildroot stable release que se encuentra en el apéndice B.

Segundo, se abre un termnal en el directorio de buildroot y se ejecuta el siguiente comando para crear la configuración de archivos basados en zynq_zed.

```
make zynq_zed_defconfig
```

Tercero, se abre el menú de configuración.

```
make menuconfig
```

Cuarto, para que la ejecución sea más rápida, se van a deshabilitar las construcciones que no se necesitan de U-boot y Kernel. Para deshabilitar el Kernel hay que entrar Kernel>Linux Kernel y pusando la tecla "n"dejarlo deshabilitado. Para deshabiltar el U-Boot, hay que entrar en Bootloaders>U-Boot y pulsar la tecla "n".

```
Target options  --->
Build options  --->
Toolchain  --->
System configuration  --->
Kernel  --->
Target packages  --->
Filesystem images  --->
Bootloaders  --->
Host utilities  --->
Legacy config options  --->
```

(a) Menú principal de configuración de root file system.

```
[ ] Linux Kernel
```

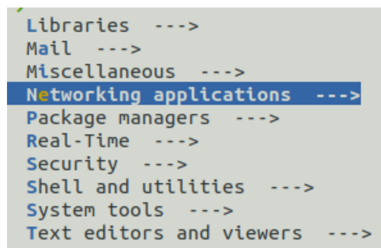
(b) Menú de Kernel de la configuración de root file system.

```
[ ] afboot-stm32
[ ] Barebox
    *** grub2 needs a toolchain w/ wchar ***
[ ] mxs-bootlets
[ ] optee_os
[ ] s500-bootloader
[ ] U-Boot
```

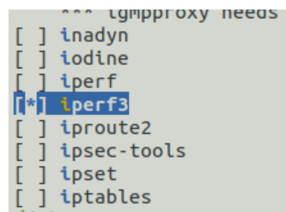
(c) Menú de Bootloader de la configuracion de root file system.

Figura 25: Menú de configuración de los archivos binarios para eliminar la generación de U-boot y del Kernel del root file system.

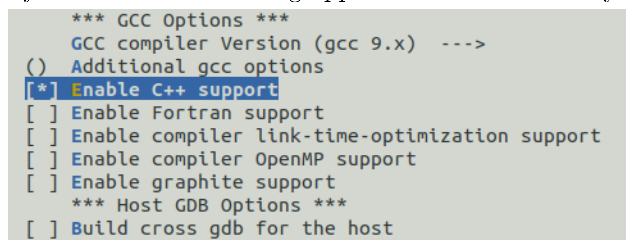
Quinto, dentro de Target packages>Networking applications se encuentra el iperf3. Para instalarlo solo hay que añadirlo pulsando la tecla "y". Para que pueda correr correctamente el Iperf3, es necesario añadir el siguiente binario para que iperf pueda ejecutarse correctamente: Toolchain>Enable C++ support pulsando la tecla "y". Ahora ya se puede guardas y salir del menú.



(a) Menú de configuración de Target Packages de root file system.



(b) Menú de configuración de Networking applications de root file system.



(c) Menú de configuración de Toolchain de root file system.

Figura 26: Menú de configuración de root file system para añadir iperf en el root file system.

Sexto, se compila el root file system con el siguiente comando:

```
make
```

Esto puede tardar bastante, y puede que salten fallos después de que se compile correctamente el archivo rootsf.tar, pero para este trabajo solo se necesita que llegue a ese punto.

Séptimo y último paso es introducir el rootfs dentro de la tarjeta microSD.

```
cd /output/images/ sudo tar -C /media/<user>/rootfs -xvf rootfs.tar
```

11.5. Análisis de la red

El sistema en conjunto está formado por una CPU y una ZYBO. Ambos sistemas están conectados a un router a través de sus puertos de ethernet. La CPU será la encargada de mostrar las salidas de los resultados. Para esto, se abrirá un puerto serie que mostrará la salida del OS que estemos ejecutando en la ZYBO, el otro terminal será el símbolo de sistema. Para que pueda funcionar correctamente Iperf debe ser instalado tanto en los Linux, como en el Windows.

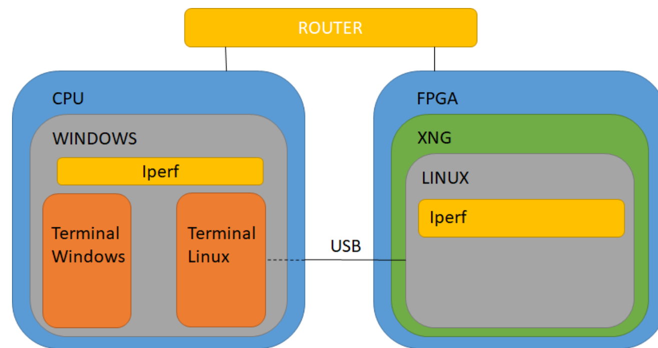


Figura 27: Sistema completo de PC-ZYBO.

Para que Iperf analice la conexión es necesario elegir un servidor y un cliente.Como primer caso se va a elegir el propio Windows del PC como servidor y la ZYBO como cliente. La IP del PC: 192.168.1.40.

Se abre un terminal dentro de windows y se ejecuta iperf.

```
C:\Users\julen\OneDrive\Desktop\ipef3\iperf-3.1.3-win64>iperf3 -s
-----------------------------------------------------------
Server listening on 5201
-----------------------------------------------------------
```

Figura 28: Terminal de Windows ejecutando Iperf como servidor.

Se empezará por ejecutar Iperf con la tarjeta microSD que solo tiene el Linux. Para poder ver que buenos resultados se le deja actuar durante 60 segundos.

```
[ ID] Interval           Transfer     Bandwidth       Retr
[  4]   0.00-60.01  sec  3.59 GBytes   514 Mbits/sec   40             sender
[  4]   0.00-60.01  sec  3.59 GBytes   514 Mbits/sec                  receiver
```

Figura 29: Resultados obtenidos al ejecutar Iperf en Linux como cliente y Windows como servidor.

Se cambia de tarjeta microSD a la que tiene el hipervisor XNG y se realiza la misma prueba. Igual que antes se le deja actuar 60 segundo.

```
[ ID] Interval           Transfer     Bitrate         Retr
[  5]   0.00-60.02  sec  1.69 GBytes   242 Mbits/sec    0             sender
[  5]   0.00-60.02  sec  1.69 GBytes   242 Mbits/sec                  receiver
```

Figura 30: Resultados obtenidos de ejecutar Iperf en XNG como cliente y Windos como servidor.

Analogamente, se realiza la prueba con la tarjeta que no tiene el hipervisor como servidor y el Windows como cliente.

```
- - - - - - - - - - - - - - - - - - - - - - - - -
[ ID] Interval           Transfer     Bandwidth
[  5]   0.00-60.01  sec  4.56 GBytes   653 Mbits/sec                  sender
[  5]   0.00-60.01  sec  4.56 GBytes   653 Mbits/sec                  receiver
-----------------------------------------------------------
Server listening on 5201
-----------------------------------------------------------
```

Figura 31: Resultados obtenidos de ejecutar Iperf en Linux como servidor y Windows como cliente.

Por último, se realiza la misma prueba usando el hipervisor XNG.

```
- - - - - - - - - - - - - - - - - - - - - - - - -
[ ID] Interval           Transfer     Bitrate
[  5]   0.00-60.03  sec  3.50 GBytes   501 Mbits/sec                  receiver
-----------------------------------------------------------
Server listening on 5201
-----------------------------------------------------------
```

Figura 32: Resultados obtenidos de ejecutar Iperf en XNG como servidor y Windows como cliente.

Servidor	Cliente	Datos transferidos	Bitrate
Windows	Linux	3.59 MBytes	514 Mbits/s
Windows	XNG	1.69 GBytes	242 Mbits/s
Linux	Windows	4.56 GBytes	653 Mbits/s
XNG	Windows	3.50 GBytes	501 Mbits/s

Cuadro 2: Comparación en la comunicación tras usar un hipervisor.

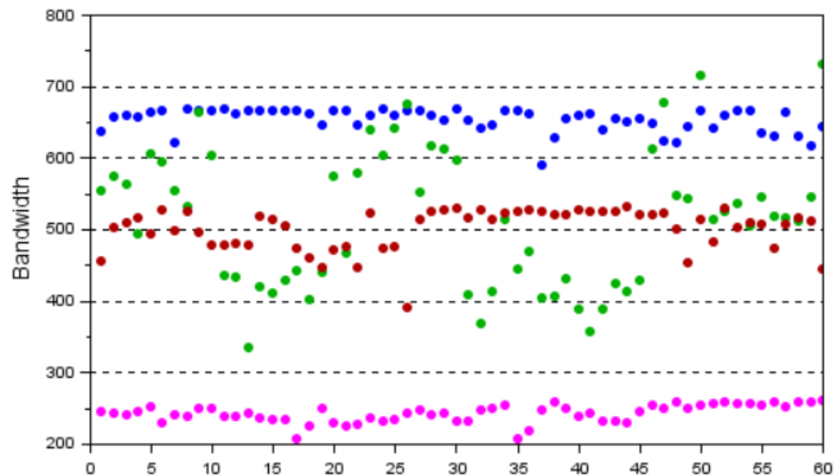


Figura 33: Gráfica con todas las muestras de las pruebas. En azul las muestras cuando Linux es el servidor. En rojo cuando XNG es servidor. En verde las muestras cuando Linux es cliente. En rosa las muestras cuando XNG es cliente

Antes de hacer ninguna comparativa es necesario destacar que aunque los dos OS sean Linux, se trata de dos distribuciones diferentes.

Por un lado, como se puede ver en la tabla 2 y la figura 33 cuando Windows es el cliente el uso del hipervisor empeora la velocidad de las comunicaciones reduciendo la velocidad a más de la mitad. No obstante, haciendo una comparativa con el trabajo de Sara et al. [9], donde analizan las interferencias del hipervisor XEN, se observó que el uso del mismo provoca que las velocidades de la red se se reduzcan unicamente a dos tercios de la velocidad que correspondería en caso de no utilizar hipervisor.

Por otro lado, cuando el servidor es la placa, el hipervisor generá interferencias pero esta vez la velocidad de comunicación solo se reduce una cuarta parte.

Todas estas medidas han sido tomadas con la placa conectada a una velocidad máxima de 1000Mbits/s. Este proceso de conexión lo realiza la placa de manera automática siendo de 10,100 o 1000Mbits/s [8].

12. Conclusiones

Como se ha visto la mayoría de hipervisores aunque aparentemente sean muy parecidos, presentan grandes diferencias en cuanto a estructura y diseño. La mayoría de hipervisores para FPGAs resultan ser los de Tipo 1, principalmente por el hecho de ahorrar espacio en memoria y en procesado. Muchos de ellos resultan útiles para aislar o asegurar sistemas que resultan críticos como en el ambito aeroespacial.

En este trabajo se ha demostrado que el uso de un hipervisor puede interferir en la comunicación con otro dispositivo, haciendo que un sistema con hipervisor reciba datos a una velocidad la mitad de rápida que uno que no use. No obstante, a la hora de trasmitir datos al exterior, a pesar de que sigue siendo más rápido usar la ZYBO sin hipervisor, la diferencia no es tan grande.

Personalmente ha sido un trabajo duro realizar este proyecto principalmente debido a la falta de conocimiento sobre el tema, sin embargo, una vez finalizado, resulta gratificante haber podido aprender tanto de él, no solo en lo referente a hipervisores sino tambien, aprender a trabajar con Linux y a aprender más de la placa de desarrollo ZYBO.

13. Trabajo Futuro

Una opción que resultaría interesante es explotar más las posibilidades de XNG, crear otro cliente de tipo Linux y probar las comunicaciones internas del mismo. Otra opción, sería realizar está misma prueba pero usando una aplicación standalone.

Ya que se conocen los resultados de XNG por este proyecto y los resultados de Xen, sería interesante probar otros hipervisores y realizar la misma prueba para determinar cuál es el mejor hipervisor para una aplicación que requiera usar uno y dependa mucho de la comunicación con el exterior.

Por último, otra línea de trabajo podría ser usar otra placa. En este trabajo se ha usado la placa ZYBO, que es la más pequeña de la serie Zynq7000, habría que estudiar como afecta el XNG a las diferentes FPGAs, mas áun, se podría probar con FPGAs de otras marcas.

A.

Enlaces para descargar los archivos necesarios para instalar Linux en una ZYBO.

Descargar el U-Boot para instalar Linux:
https://github.com/SDU-Embedded/u-boot-xlnx

Para construir el Kernel de Linux:
https://github.com/SDU-Embedded/linux-xlnx/releases/tag/zynmp-dt-fixes-for-4.10

Fichero de configuración de xilinx_zynq_defconfig:
(https://github.com/SDU-Embedded/linux-xlnx/releases/tag/zynmp-dt-fixes-for-4.10)

Para descargar el devicetree.dts:
https://github.com/SDU-Embedded/linux_zynq/blob/master/linux_zybo/devicetree.dts

Para descargar el sistema de archivos ROOTFS:
http://releases.linaro.org/debian/images/developer-armhf/17.02/

B.

Enlaces de descarga para instalar XNG con Linux en una ZYBO

Buildroot stable release:
https://buildroot.org/downloads/buildroot-2020.08.tar.gz

Referencias

[1] *CODEZERO*. https://github.com/drasko/codezero.

[2] *Installing Vivado Board Files for Digilent Boards*.

[3] *iperf*. https://iperf.fr/.

[4] *Siemens/Jailhouse*. https://github.com/siemens/jailhouse.

[5] *SYSGO*. https://www.sysgo.com/pikeos.

[6] *User Guide of the Hypervisor XtratuM*.

[7] *Xenproject*. https://xenproject.org/.

[8] *Zybo Reference Manual*. https://reference.digilentinc.com/programmable-logic/zybo/reference-manual.

[9] Alonso, S., Lazaro, J., Jimenez, J., Muguira, L., and Largacha, A. Analysing the interference of xen hypervisor in the network speed. In *2020 XXXV Conference on Design of Circuits and Integrated Systems (DCIS)* (nov 2020), IEEE.

[10] Alptekin, A., Yilmazer, Y., Usug, U., Koca, F., and Incki, K. Design for ARINC 653 conformance: Architecting independent validation of a safety-critical RTOS. In *2014 IEEE/AIAA 33rd Digital Avionics Systems Conference (DASC)* (oct 2014), IEEE.

[11] Alvarez, A. *Linux on XNG/Zynq7000*. FENTISS, May 2021.

[12] Amazon. *Instancias F1 de Amazon EC2*. https://aws.amazon.com/es/ec2/instance-types/f1/.

[13] ARM Developer. *Introducing the arm architecture*. https://developer.arm.com/, 2019.

[14] Asiatici, M., George, N., Vipin, K., Fahmy, S. A., and Ienne, P. Virtualized execution runtime for FPGA accelerators in the cloud. *IEEE Access 5* (2017), 1900–1910.

[15] Blenk, A., Basta, A., Reisslein, M., and Kellerer, W. Survey on network virtualization hypervisors for software defined networking. *IEEE Communications Surveys & Tutorials 18*, 1 (2016), 655–685.

[16] Churiwala, S., Ed. *Designing with Xilinx FPGAs*. Springer, 2017.

[17] Dídimo, A. T. A., Lobo, T., Cabral, P. C. J., and Montenegro, S. Rodosvisor — an arinc 653 quasi-compliant hypervisor: Cpu, memory and i/o virtualization. *Proceedings of 2012 IEEE 17th International Conference on Emerging Technologies and Factory Automation (ETFA 2012)* (2012).

[18] Deshmukh, P. P., and Amdani, S. Y. Survey of memory streaming techniques for virtual machine in cloud environment. In *2017 International Conference on Energy, Communication, Data Analytics and Soft Computing (ICECDS)* (aug 2017), IEEE.

[19] Digi International. *U-Boot Reference Manual.*

[20] Erol, A., Yazar, A., and Schmidt, E. G. OpenStack generalization for hardware accelerated clouds. In *2019 28th International Conference on Computer Communication and Networks (ICCCN)* (jul 2019), IEEE.

[21] Fornaeus, J. Device hypervisors. In *Proceedings of the 47th Design Automation Conference on - DAC '10* (2010), ACM Press.

[22] Goldberg, R. P. Survey of virtual machine research. *Computer 7*, 6 (jun 1974), 34–45.

[23] Hand, S., Warfield, A., andEvangelos Kotsovinos, K. F., and Magenheimer, D. Are virtual machine monitors microkernels done right? *HotOS* (2005).

[24] Heiser, G. Hypervisors for consumer electronics. *2009 6th IEEE Consumer Communications and Networking Conference* (2009).

[25] Heiser, G. Virtualizing embedded systems - why bother? *2011 48th ACM/EDAC/IEEE Design Automation Conference (DAC)* (2011).

[26] Heiser, G., and Leslie, B. The okl4 microvisor:convergence point of microkernels and hypervisors. *Conference: Proceedings of the 1st ACM SIGCOMM Asia-Pacific Workshop on Systems, ApSys 2010, New Delhi, India, August 30, 2010* (2010).

[27] Hernandez, C., Flieh, J., Paredes, R., Lefebvre, C.-A., Allende, I., Abella, J., Trillin, D., Matschnig, M., Fischer, B., Schwarz, K., Kiszka, J., Ronnback, M., Klockars, J., McGuire, N., Rammerstorfer, F., Schwarzl, C., Wartet, F., Ludemann, D., and Labayen, M. SELENE: Self-monitored dependable platform for high-performance safety-critical systems. In *2020 23rd Euromicro Conference on Digital System Design (DSD)* (aug 2020), IEEE.

[28] HUAWEI. *FPGA Accelerated Cloud Server.* https://www.huaweicloud.com/en-us/product/fcs.html.

[29] Jansen, B., Korkmaz, F., Derya, H., Hubner, M., Ferreira, M. L., and Ferreira, J. C. Towards a type 0 hypervisor for dynamic reconfigurable systems. In *2017 International Conference on ReConFigurable Computing and FPGAs (ReConFig)* (dec 2017), IEEE.

[30] Jones, M. Virtualization for embedded systems. *https://developer.ibm.com/tutorials/l-embedded-virtualization/* (2011).

[31] Kirchgessener, R., Stitt, G., George, A., and Lam, H. Virtualrc: A virtual fpga platform for applications and tools portability. *Proceedings of the ACM/SIGDA 20th International Symposium on Field Programable Gate Aarrays* (2012).

[32] Knodel, O., Genssler, P. R., and Spallek, R. G. Virtualizing reconfigurable hardware to provide scalability in cloud architectures. *The Tenth International Conference on Advances in Circuits, Electronics and Micro-Electronics* (2017).

[33] Kosuge, A., Yamamoto, K., Akamine, Y., and Oshima, T. An SoC-FPGA-based iterative-closest-point accelerator enabling faster picking robots. *IEEE Transactions on Industrial Electronics 68*, 4 (apr 2021), 3567–3576.

[34] Le, D.-C., Oh, E.-Y., Cho, G.-S., Lee, K.-C., Kim, S.-H., and Youn, C.-H. Performance analysis of adaptive resource allocation scheme for OpenCL-based FPGA virtualization system. In *2019 International Conference on Information and Communication Technology Convergence (ICTC)* (oct 2019), IEEE.

[35] Ma, J., Zuo, G., Loughlin, K., Cheng, X., Liu, Y., Eneyew, A. M., Qi, Z., and Kasikci, B. A hypervisor for shared-memory FPGA platforms. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (mar 2020), ACM.

[36] Masmano, M. *Sofware User Manual: XNG hypervisor*, May 2021.

[37] Masmano, M. *Sofware User Manual: XNG hypervisor extended*, May 2021.

[38] Mbongue, J. M., Shuping, A., Bhowmik, P., and Bobda, C. Architecture support for FPGA multi-tenancy in the cloud. In *2020 IEEE 31st International Conference on Application-specific Systems, Architectures and Processors (ASAP)* (jul 2020), IEEE.

[39] Mishra, G., Ashwini, V., and Sunkara, S. Virtualization on ARM embedded platform codezero hypervisor-a case study. In *2018 International Conference on Networking, Embedded and Wireless Systems (ICNEWS)* (dec 2018), IEEE.

[40] Morabito, R., Kjällman, J., and Komu, M. Hypervisors vs. lightweight virtualization: a performance comparison. *IEEE International Conference on Cloud Engineering* (2015).

[41] Nicolle, J. P. Fpga4fun. *https://www.fpga4fun.com/SiteInformation.html*.

[42] Plessl, C., and Platzner, M. Virtualization of hardware - introduction and survey. *Proc. Int. Conf. on Engineering of Reconfigurable Systems and Algorithms (ERSA)* (2004).

[43] Shan, Y., Wang, B., Yan, J., Wang, Y., Xu, N., and Yang, H. Fpmr: Mapreduce framework on fpga a case study of rankboost acceleration. *ARC* (2010).

[44] Stitt, G., Karam, R., Yang, K., and Bhunia, S. A uniquified virtualization approach to hardware security. *IEEE Embedded Systems Letters 9*, 3 (sep 2017), 53–56.

[45] Suzuki, J., Hidaka, Y., Higuchi, J., Baba, T., Kami, N., and Yoshikawa, T. Multi-root share of single-root i/o virtualization (SR-IOV) compliant PCI express device. In *2010 18th IEEE Symposium on High Performance Interconnects* (aug 2010), IEEE.

[46] Vaishnav, A., Pham, K. D., and Koch, D. A survey on FPGA virtualization. In *2018 28th International Conference on Field Programmable Logic and Applications (FPL)* (aug 2018), IEEE.

[47] Vesper, M., Koch, D., Vipin, K., and Fahmy, S. A. JetStream: An open-source high-performance PCI express 3 streaming library for FPGA-to-host and FPGA-to-FPGA communication. In *2016 26th International Conference on Field Programmable Logic and Applications (FPL)* (aug 2016), IEEE.

[48] Wang, R., Xu, L., Bai, Y., Cheng, K., Wang, Z., Luan, G., Yang, H., and Wang, W. Research on asynchronous inter-VM communication mechanism based on embedded hypervisor. In *2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)* (jul 2018), IEEE.

[49] Wang, W., Bolic, M., and Parri, J. pvFPGA: Accessing an FPGA-based hardware accelerator in a paravirtualized environment.

In *2013 International Conference on Hardware/Software Codesign and System Synthesis* (Sept. 2013), IEEE.

[50] XIA, T., PREVOTET, J.-C., AND NOUVEL, F. Mini-NOVA: A lightweight ARM-based virtualization microkernel supporting dynamic partial reconfiguration. In *2015 IEEE International Parallel and Distributed Processing Symposium Workshop* (may 2015), IEEE.

[51] XIA, T., PREVOTET, J.-C., AND NOUVEL, F. Hypervisor mechanisms to manage FPGA reconfigurable accelerators. In *2016 International Conference on Field-Programmable Technology (FPT)* (dec 2016), IEEE.

[52] XIA, T., TIAN, Y., PRÉVOTET, J.-C., AND NOUVEL, F. Ker-one: A new hypervisor managing fpga reconfigurable accelerators. *Journal of System Architecture* (2019).

[53] XILINX. *MicroBlaze Soft Processor Core.* https://www.xilinx.com/products/design-tools/microblaze.html.

[54] XILINX. Enabling virtualizationwith xen hypervisor onzynq ultrascale+ mpsocs. *Xilinx All Programable* (2016).

[55] XU, N.-Y., CAI, X.-F., GAO, R., ZHANG, L., AND HSU, F.-H. FPGA-based accelerator design for RankBoost in web search engines. In *2007 International Conference on Field-Programmable Technology* (dec 2007), IEEE.

[56] ZOU, X., CHENG, A. M., LI, Y., AND JIANG, Y. A temporal partition-based linux CPU scheduler. In *2014 IEEE Intl Conf on High Performance Computing and Communications, 2014 IEEE 6th Intl Symp on Cyberspace Safety and Security, 2014 IEEE 11th Intl Conf on Embedded Software and Syst (HPCC,CSS,ICESS)* (aug 2014), IEEE.