Universidad de San Carlos de Guatemala
Facultad de Ingeniería
Escuela de Ingeniería en Ciencias y Sistemas

TÉCNICAS DE OPTIMIZACIÓN EN TIEMPO DE EJECUCIÓN EN CÓDIGO INTERPRETADO POR MÁQUINAS VIRTUALES Y EMULADORES

Erik Vladimir Girón Márquez
Asesorado por el Ing. Luis Fernando Alvarado Cruz

Guatemala, agosto de 2012

UNIVERSIDAD DE SAN CARLOS DE GUATEMALA

FACULTAD DE INGENIERÍA

TÉCNICAS DE OPTIMIZACIÓN EN TIEMPO DE EJECUCIÓN EN CÓDIGO INTERPRETADO POR MÁQUINAS VIRTUALES Y EMULADORES

TRABAJO DE GRADUACIÓN

PRESENTADO A LA JUNTA DIRECTIVA DE LA
FACULTAD DE INGENIERÍA
POR

ERIK VLADIMIR GIRÓN MÁRQUEZ
ASESORADO POR EL ING. LUIS FERNANDO ALVARADO CRUZ

AL CONFERÍRSELE EL TÍTULO DE

INGENIERO EN CIENCIAS Y SISTEMAS

GUATEMALA, AGOSTO DE 2012

UNIVERSIDAD DE SAN CARLOS DE GUATEMALA

FACULTAD DE INGENIERÍA

NÓMINA DE JUNTA DIRECTIVA

DECANO	Ing. Murphy Olympo Paiz Recinos
VOCAL I	Ing. Alfredo Enrique Beber Aceituno
VOCAL II	Ing. Pedro Antonio Aguilar Polanco
VOCAL III	Ing. Miguel Ángel Dávila Calderón
VOCAL IV	Br. Juan Carlos Molina Jiménez
VOCAL V	Br. Mario Maldonado Muralles
SECRETARIO	Ing. Hugo Humberto Rivera Pérez

TRIBUNAL QUE PRACTICÓ EL EXAMEN GENERAL PRIVADO

DECANO	Ing. Murphy Olympo Paiz Recinos
EXAMINADOR	Ing. Juan Álvaro Díaz Ardavín
EXAMINADOR	Ing. Edgar Josué González Constanza
EXAMINADOR	Ing. José Ricardo Morales Prado
SECRETARIA	Inga. Marcia Ivónne Véliz Vargas

HONORABLE TRIBUNAL EXAMINADOR

En cumplimiento con los preceptos que establece la ley de la Universidad de San Carlos de Guatemala, presento a su consideración mi trabajo de graduación titulado:

TÉCNICAS DE OPTIMIZACIÓN EN TIEMPO DE EJECUCIÓN EN CÓDIGO INTERPRETADO POR MÁQUINAS VIRTUALES Y EMULADORES

Tema que me fuera asignado por la Dirección de la Escuela de Ingeniería en Ciencias y Sistemas, con fecha 1 de febrero de 2012.

Erik Vladimir Girón Márquez

Guatemala, 25 de Mayo de 2012

Ingeniero
Carlos Azurdia
Coordinador
Escuela de Ciencias y Sistemas
Facultad de Ingeniería
USAC

Respetable Ingeniero:

Por este medio me permito hacer de su conocimiento que he llevado a cabo una revisión completa del trabajo de tesis titulado "***Técnicas de optimización en tiempo de ejecución en código interpretado por máquinas virtuales y emuladores***" elaborado por el estudiante **Erik Vladimir Girón Márquez**, el cual, a mi juicio, cumple con los objetivos propuestos en el anteproyecto de tesis.

Por lo tanto, el autor de esta tesis, y el suscrito en calidad de asesor, nos hacemos responsables por el contenido y conclusiones del mismo.

Sin otro particular, me suscribo atentamente.

Ing. Luis Fernando Alvarado
Colegiado No. 9957
Asesor

Universidad San Carlos de Guatemala
Facultad de Ingeniería
Escuela de Ingeniería en Ciencias y Sistemas

Guatemala, 06 de Junio de 2012

Ingeniero
Marlon Antonio Pérez Turk
Director de la Escuela de Ingeniería
En Ciencias y Sistemas

Respetable Ingeniero Pérez:

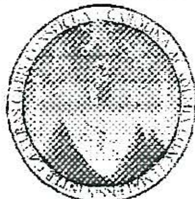
Por este medio hago de su conocimiento que he revisado el trabajo de graduación del estudiante **ERIK VLADIMIR GIRÓN MÁRQUEZ** carné **2003-13492,** titulado: **"TÉCNICAS DE OPTIMIZACIÓN EN TIEMPO DE EJECUCIÓN EN CÓDIGO INTERPRETADO POR MÁQUINAS VIRTUALES Y EMULADORES",** y a mi criterio el mismo cumple con los objetivos propuestos para su desarrollo, según el protocolo.

Al agradecer su atención a la presente, aprovecho la oportunidad para suscribirme,

Atentamente,

Ing. Carlos Alfredo Azurdia
Coordinador de Privados
y Revisión de Trabajos de Graduación

UNIVERSIDAD DE SAN CARLOS
DE GUATEMALA



FACULTAD DE INGENIERÍA
ESCUELA DE CIENCIAS Y SISTEMAS
TEL: 24767644

ESCUELA DE CIENCIAS Y SISTEMAS

*El Director de la Escuela de Ingeniería en Ciencias y Sistemas de la Facultad de Ingeniería de la Universidad de San Carlos de Guatemala, luego de conocer el dictamen del asesor con el visto bueno del revisor y del Licenciado en Letras, del trabajo de graduación titulado **"TÉCNICAS DE OPTIMIZACIÓN EN TIEMPO DE EJECUCIÓN EN CÓDIGO INTERPRETADO POR MÁQUINAS VIRTUALES Y EMULADORES"** presentado por el estudiante ERIK VLADIMIR GIRÓN MÁRQUEZ, aprueba el presente trabajo y solicita la autorización del mismo.*

"ID Y ENSEÑAD A TODOS"

Ing. Marlon Antonio Pérez Turk
Director, Escuela de Ingeniería en Ciencias y Sistemas

Guatemala, 06 de agosto 2012

Universidad de San Carlos
de Guatemala

Facultad de Ingeniería
Decanato

Ref.DTG.388.2012

El Decano de la Facultad de Ingeniería de la Universidad de San Carlos de Guatemala, luego de conocer la aprobación por parte del Director de la Escuela de Ingeniería en Ciencias y Sistemas, al trabajo de graduación titulado: **TÉCNICAS DE OPTIMIZACIÓN EN TIEMPO DE EJECUCIÓN EN CÓDIGO INTERPRETADO POR MÁQUINAS VIRTUALES Y EMULADORES**, presentado por el estudiante universitario: **Erik Vladimir Girón Márquez**, procede a la autorización para la impresión del mismo.

IMPRÍMASE.

Ing. Murphy Olympo Paiz Recinos
DECANO

Guatemala, agosto de 2012

/cc

ACTO QUE DEDICO A:

José Axel Girón Mayer	Tu ejemplo será siempre mi inspiración. Gracias padre.
Gloria Marina Márquez Valenzuela	Por tu bondad y paciencia, quien supo guiarme por el camino correcto. Gracias madre.
Amanda Girón, Axel Girón y Leslie Girón	Por todo su apoyo a mis hermanos biológicos y del alma.
Jose Carlos Girón, Alizon Estrada y Nicole Estrada	Mis sobrinos que agregan alegría a mi vida, gracias por su cariño.
Ana Luisa Pinott	Mi poesía hecha poetiza, mi presente y mi futuro, te amo.

AGRADECIMIENTOS:

Universidad de San Carlos de Guatemala

Mi alma máter; gracias por ser el canal de transmisión de valiosos conocimientos y sede de las mejores experiencias de mi vida. Id y enseñad a todos.

Pueblo de Guatemala

Por su interminable lucha en exigir y patrocinar educación libre y gratuita.

Ing. Luis Alvarado

Por su confianza y apoyo a este proyecto de graduación.

ÍNDICE GENERAL

ÍNDICE DE ILUSTRACIONES....VII
GLOSARIO....IX
RESUMEN....XXIII
OBJETIVOS....XXV
INTRODUCCIÓN....XXVII

1. INTRODUCCIÓN A LA EMULACIÓN Y VIRTUALIZACIÓN....1
    1.1. Conceptos básicos....1
        1.1.1. Compilador e intérprete....1
        1.1.2. Máquina virtual....1
        1.1.3. Emulador....2
    1.2. Niveles de abstracción....3
        1.2.1. Niveles de abstracción, caso TCP/IP....3
        1.2.2. Niveles de abstracción, caso sistema de archivos en UNIX....5
        1.2.3. Interfaces bien definidas....6
    1.3. Arquitectura de computadoras....9
        1.3.1. La arquitectura de conjunto de instrucciones ISA....10
    1.4. Taxonomía de máquinas virtuales....13

2. EMULACIÓN E INTERPRETACIÓN....15
    2.1. Introducción....15
    2.2. Interpretación básica....18
        2.2.1. Interpretación de arquitecturas CISC....22

2.2.2. Caso de estudio de emulador basado en intérprete, ZSNES....................23

3. EMULACIÓN, RECOMPILACIÓN DINÁMICA Y PROCESAMIENTO EN PARALELO....................27
3.1. Predecodificado....................27
3.2. Interpretación directa....................28
3.3. Traducción binaria o recompilación dinámica....................29
3.3.1. El problema de localidad y descubrimiento del código....................31
3.3.2. Traducción y predecodificado incremental....................33
3.4. Emulación de alto nivel y bajo nivel....................35
3.4.1. Criterios al elegir el método de emulación de alto nivel....................36
3.4.2. Ventajas y desventajas....................37
3.5. Procesamiento en paralelo....................38
3.5.1. Niveles de paralelismo....................40
3.5.1.1. Nivel de bit....................40
3.5.1.2. Nivel de instrucciones....................41
3.5.1.3. Nivel de datos....................41
3.5.1.4. Nivel de tareas....................42
3.5.2. Clases de computadoras en paralelo....................43
3.5.2.1. Multinúcleo....................43
3.5.2.2. Simétricas....................43
3.5.2.3. Cluster....................44
3.5.2.4. Grid....................44
3.6. Caso de estudio de emulador con recompilación dinámica, QEMU....................44

3.6.1. Recompilación dinámica en QEMU............47

4. MÁQUINAS VIRTUALES...49
4.1. Java Virtual Machine (JVM)...49
4.1.1. Componentes de la JVM...50
4.1.1.1. El cargador de Clases.............................50
4.1.1.2. El administrador de Seguridad.................50
4.1.1.3. Recolector de basura..............................50
4.1.1.4. Manejador de hilos..................................51
4.1.1.5. Entrada / salida (I/O)...............................51
4.1.1.6. Intérprete de bytecode.............................52
4.1.2. Implementación del intérprete de Java.......................53
4.1.2.1. Intérprete estándar..................................53
4.1.2.1.1. Identificación de los bloques..............................54
4.1.2.1.2. Construcción de un DAG…55
4.1.2.1.3. Generación de código desde el DAG....................55
4.1.2.2. Implementación de la JVM en ISA x86….56
4.1.2.2.1. Aritmética de enteros.........56
4.1.2.2.2. Aritmética de tipos dobles..56
4.1.2.2.3. Invocación de métodos......56
4.1.2.2.4. Creación de objetos...........57
4.1.2.2.5. Manipulación de campos de los objetos.....................57
4.1.2.2.6. Manipulación de variables locales...............................57
4.1.2.2.7. Excepciones......................58

4.2. Microsoft CLR..58
4.2.1. Common Type System (CTS).....................................59
4.2.2. Common Language Specification (CLS)......................59
4.2.3. Common Intermediate Language (CIL).......................59
4.2.4. Virtual Execution System (VES) 59
4.3. Esquemas de Virtualización...60
4.3.1. Virtualización pura...60
4.3.2. Virtualizacion en XEN..62
4.3.2.1. Mapa de memoria virtual tradicional........63
4.3.3. Virtualización por hardware..64
4.3.4. Paravirtualizacion...66

5. ANÁLISIS Y EVALUACIÓN DE LAS TÉCNICAS DE RECOMPILACIÓN DINÁMICA, VIRTUALIZACIÓN Y PARAVIRTUALIZACIÓN......................67
5.1. Selección de técnicas de recompilación y definición de la hipótesis...67
5.1.1. Definición de la hipótesis..67
5.1.2. Criterios del algoritmo a evaluar.................................67
5.1.3. Elección del algoritmo..68
5.1.3.1. Algoritmo de factorización de números primos...69
5.1.3.2. Implementación en el lenguaje Python y complejidad...69
5.1.4. Definición del área análisis, alcance y definición de la muestra...70
5.1.5. Técnicas de recompilación y emulación a evaluar......71
5.1.5.1. Especificación de la máquina anfitrión.....71

5.1.5.2. Especificación de la máquina virtual PowerPC 72
5.1.5.3. Especificación de la máquina virtual ARM...72
5.1.5.4. Especificación de la máquina virtual x86-64 72
5.1.5.5. Especificación de la máquina paravirtualizada x86-64 73
5.2. Análisis de datos y evaluación comparativa................................73
5.2.1. PowerPC emulado contra x64 nativo..........................73
5.2.2. ARM emulado contra x64 nativo.................................74
5.2.3. X64 emulado contra x64 nativo..................................75
5.2.4. X86-64 paravirtualizado contra x64 nativo..................76
5.3. Resultados y validación de la hipótesis......................................77
5.3.1. Resumen de totales de tiempo real de ejecución.......77
5.3.2. Eficiencia proporcional de los emuladores respecto a la ejecución nativa en el anfitrión............................79
5.3.3. Conclusiones de la evaluación comparativa...............81

CONCLUSIONES...83
RECOMENDACIONES...87
BIBLIOGRAFÍA..89
APÉNDICES..95

ÍNDICE DE ILUSTRACIONES

FIGURAS

1. Niveles de abstracción en capa OSI/TCP.....4
2. Organización por capas del sistema de archivos en UNIX.....5
3. Esquema de separación en capas definido en la arquitectura ISA.....11
4. Jerarquía de máquinas virtuales según su interacción con el sistema fuente.....14
5. Funcionamiento de una máquina virtual de proceso que emula un entorno operativo x86 dentro de una PowerPC.....17
6. Interacción y estructura en memoria de un intérprete.....19
7. Ejemplo del motor de un intérprete.....20
8. Ejemplo gráfico de predecodificado.....27
9. Esquema en memoria del método de Interpretación directa 29
10. Fuente en x86.....30
11. Ejemplo de traducción a opcode PPC 31
12. Ejemplo de tamaño variable de opcode.....32
13. Esquema del funcionamiento de precodificado incremental.....34
14. Diagrama de flujo para el proceso de traducción incremental.....35
15. Ley de Amdahl.....39
16. Paralelismo de instrucciones en RISC.....41
17. Esquema general de las operaciones VM *Entry* y VM *Exit*.....63
18. Modelo plano de memoria.....64
19. Pseudocódigo del algoritmo de factorización de números primos.....69
20. Análisis de complejidad del algoritmo seleccionado.....70

21. Comparativa PowerPC emulado contra x64 nativo................................74
22. Comparativa ARM emulado contra x64 nativo...75
23. Comparativa X64 emulado contra x64 nativo..76
24. Comparativa X64 emulado contra x64 nativo..77
25. Total del tiempo real de ejecución...78
26. Promedio del tiempo real de ejecución..78
27. Eficiencia proporcional respecto a la ejecución nativa............................80

TABLAS

I. Estructura de los códigos de operación para un procesador PowerPC….7
II. Porcentaje de emulación de procesadores personalizados en ZSNES‥26
III. Arquitecturas destino soportadas por el emulador QEMU......................46
IV. Arquitecturas en donde se puede ejecutar QEMU para emulación.…….47
V. Tabla de *opcodes* del *bytecode* para la máquina virtual de Java............52

GLOSARIO

ABI Application Binary Interface por sus siglas en inglés, describe una interfaz de bajo nivel entre una aplicación y el sistema operativo u otra aplicación.

API Application Program Interface, por sus siglas en inglés, se refiere a la serie de llamadas públicas que tiene un componente de software para que ésta pueda ser utilizada por otros componentes diferentes.

Android Sistema operativo para dispositivos móviles basado en Linux que ejecuta el entorno de usuario sobre una máquina virtual basada en Java.

Applet Aplicación Java incrustada dentro de una página web.

Arreglo Agrupación indexada consecutiva de datos.

Bit Es la unidad básica de información en computación y telecomunicaciones. Y representa la cantidad de información que puede ser almacenada por un dispositivo u otro sistema físico que puede existir únicamente en 2 estados, encendido (1) o apagado (0).

Big endian En computación, se refiere al ordenamiento de valor en un *byte*, en el que va en primer posición el *bit* más significativo.

Branch En ciencia computacional se refiere a un salto inesperado en la ejecución de un programa.

Byte Es una unidad de información en computación y telecomunicaciones, compuesto por 8 bits agrupados y es utilizado muchas veces para codificar un carácter.

Bytecode Códigos de operación a la que se compila el código Java para ser ejecutado por la máquina virtual de Java.

C Lenguaje de programación imperativo de alto nivel creado por Dennis Ritchie en los laboratorios de AT&T, con llamadas a funciones de bajo nivel, su sintaxis es basada en el lenguaje BCPL

C++ Lenguaje de programación imperativo orientado a objetos, con sintaxis heredada del lenguaje C.

CISC Complex Instruction Set Architecture por sus siglas en inglés, es una arquitectura de conjunto de instrucciones (ISA) en el que cada instrucción ejecuta varias operaciones de bajo nivel. Ejemplos de esto es la arquitectura Intel x86.

CLIB Conjunto de librerías estándar en los sistemas UNIX para aplicaciones hechas en C.

Compilador Un compilador es un programa de computadora, o un conjunto de programas, que transforman código fuente escrito en un lenguaje de computador (lenguaje fuente) hacia otro lenguaje de computador (lenguaje destino).

CPU Central Processing Unit por sus siglas en inglés, es la porción de un sistema de computadora que ejecuta las instrucciones de un programa de computadora, convirtiéndose en el elemento primario para la ejecución lógica y aritmética que éstas lleven.

Defragmentación Proceso de reagrupar segmentos de memoria, de tal manera que estén contiguos para reducir tiempo de búsqueda.

DMA Direct Memory Access por sus siglas en inglés. Es la capacidad de transferencia por bloques desde una posición de memoria a otra, sin interferir en la actividad o flujo normal del CPU.

DSP Direct Sound Processing, un tipo especial de procesador para ondas analógicas, usado regularmente para procesar audio.

Emulador En ciencia computacional, un emulador duplica las funciones de un sistema, usando un sistema diferente, de tal manera que el segundo sistema actúe como el primero, en contraste con los simuladores, que solo se enfoca del modelo abstracto del sistema que se simula.

GCC GNU Compiler Collection por sus siglas en inglés, es un conjunto de compiladores desarrollados por GNU y soporta múltiples lenguajes de programación. Es parte de las herramientas GNU y es estándar en muchos sistemas UNIX modernos.

Hardware Término general para referirse a los artefactos físicos de una tecnología, y por ende, los componentes físicos de un sistema computarizado.

Hash Función utilizada para mapear el valor de una variable de un dominio o tipo a otro dominio o tipo de menor longitud, por lo regular hacia *string*, preservando su unicidad y evitando colisiones.

Heap Área de memoria en el cual se almacena datos generada dinámicamente al ejecutarse el programa. Su crecimiento va en dirección contraria al *Stack.*

Hipervisor En computación, un *hiperrvisor*, también llamado Monitor de máquina virtual o VMM por sus siglas en inglés, es un segmento de software de virtualización de plataformas por *software*/*hardware* que permite la ejecución de múltiples sistemas operativos de forma concurrente en un sistema huésped.

Huésped En virtualización, se refiere a la máquina sobre la cual va a ejecutarse el emulador, también llamada anfitrión.

IBCS Intel Binary Compatible Standard, es un estándar para sistemas operativos UNIX que especifica una interfaz de llamadas al sistema entre el kernel y los programas de aplicación, permitiendo portabilidad binaria entre varios sistemas UNIX.

Inodos En computación, un Inodo es una estructura de datos en sistemas operativos basados en UNIX, y almacena información básica acerca de un archivo regular, directorio u otro objeto del sistema de archivos.

Interfaz En ciencia computacional, es un conjunto de operaciones nombradas de un sistema que pueden ser invocadas por por otros sistemas clientes. Generalmente se refiere a una abstracción que una entidad provee para su comunicación con el exterior, separando los métodos de comunicación externa de aquellos que operan internamente dicha entidad.

Internet Sistema global de redes de computadoras interconectadas, que usan el estándar TCP/IP para servir a millones de usuarios por todo el mundo. Es una red de redes que consiste en millones de redes privadas y públicas de gobierno, academia, y negocios; de alcance global o local y enlazados por varias tecnologías.

Intérprete En ciencia computacional, un intérprete normalmente se refiere a un programa de computador que ejecuta instrucciones escritas en otro lenguaje de programación.

Invitado En virtualización, máquina la cual se va a virtualizar o emular sobre otra máquina llamada huésped a través de una máquina virtual.

IRQ Interruption Request por su contracción en inglés, se refiere a las peticiones hechas desde código a interrupciones programables del CPU.

ISA Instruction Set Architecture por sus siglas en inglés, es la parte de una arquitectura de computadoras relacionada a la programación, incluyendo tipos de dato nativos, instrucciones, registros, modos de direccionamiento, arquitectura de memoria, interrupciones, manejo de excepciones y un I/O externo. Un ISA incluye la especificación de un conjunto de códigos de operación u *opcodes*.

Java Lenguaje de programación orientado a objetos que se compila para y se ejecuta sobre la *Java Virtual Machine.*

Javascript Lenguaje de programación de *scripting* orientado a objetos usado para permitir acceso programático a los objetos dentro de aplicaciones cliente y otras aplicaciones. Se usa principalmente del lado del cliente, implementado como un componente integrado en un navegador web, permitiendo el desarrollo de interfaces de usuario mejoradas y sitios web dinámicos.

JVM Es un conjunto de programas de computadora y estructuras de datos que usa el modelo de máquina virtual para la ejecución de programas y *scripts*. El modelo usado por una JVM utiliza un lenguaje intermedio llamado Java *Bytecode*.

Kernel Núcleo del sistema operativo el cual interactúa directamente con el *hardware* subyacente.

Linker En ciencia computacional, un *linker* o enlazador es un programa que toma uno o más objetos generados por un compilador y los combina en un solo programa ejecutable.

LISP Lisp (o LISP) es una familia de lenguajes de computadora, distinguido por su sintaxis orientada a paréntesis, y originalmente especificado en 1958, haciéndolo el segundo lenguaje de alto nivel más antiguo que aún esta en uso.

Little endian En computación, se refiere al ordenamiento del valor en un byte, en que el bit menos significativo de éste se coloca en primer posición.

Máquina de Turing En ciencia computacional, dispositivo teórico capaz de manipular símbolos sobre una cinta según una tabla de reglas; en donde, a pesar de su simplicidad, puede ser adaptada para simular la lógica de cualquier algoritmo computable.

Máquina Virtual Una máquina virtual o VM por sus siglas en inglés es una implementación en software de una máquina (o un computador) que ejecuta programas como si fuera la máquina física.

Microcódigo Código programable que se integra en un CPU para tareas diferentes a los códigos de operación nativo del mismo.

Multitarea Capacidad de un sistema operativo de ejecutar dos instancias de aplicación de manera casi paralela.

NMI Non-Maskable Interrupt por sus siglas en inglés, se refiere a un tipo de interrupción del procesador, el cual su funcionalidad no es enmascarable o intercambiable.

Opcode Operation Code, por su contracción en inglés, es una porción de una instrucción de lenguaje máquina que especifica una operación a realizar por un procesador, estando ésta especificada dentro de una arquitectura (ISA) del procesador en cuestión.

OSI Interconexión de sistemas abiertos, por sus siglas en inglés, es un esfuerzo para estandarizar las redes que inició en 1977 por la Organización Internacional para la estandarización ISO.

Paginación División de un segmento de memoria de un programa en páginas para reducir la fragmentación de memoria.

Paravirtualización Técnica de virtualización que consiste en presentar una interfaz por software a las máquinas virtuales, similar a la máquina anfitrión, aunque no idéntica.

Portabilidad Capacidad de un programa de poder ser ejecutado en varias plataformas o arquitecturas sin ser modificado en su código fuente.

PowerPC Performance Optimization With Enhanced RISC – Performance Computing, por siglas en inglés, es una arquitectura RISC creada en 1991 por Apple – IBM y Motorola, también se le llama también *Power ISA*.

Python Es un lenguaje orientado a objetos de propósito general de alto nivel, y su diseño se enfatiza en la legibilidad del código. Combinando flexibilidad junto con una sintaxis amistosa.

Reflectividad En ciencia computacional, es la capacidad de un programa de examinar y modificar la estructura y comportamiento de un objeto en tiempo de ejecución.

RISC Reduced Instruction Set Computer por sus siglas en inglés, representa un diseño de procesadores enfatizado en proveer instrucciones simples que hagan menos, pero que provean mayor rendimiento, tambien conocido como arquitectura de carga-almacenamiento.

Sandbox Área de ejecución aislado que no compromete la información externa a el código en éste.

SDK Software Development Kit por sus siglas en Inglés, conjunto de herramientas de desarrollo de software que simplifican la interfaz entre el código escrito y el entorno a donde se va a desarrollar.

Smalltalk Es un lenguaje orientado a objetos, tipificado dinámicamente y reflectivo, creado como lenguaje para presentar un nuevo paradigma en computación que ejemplificase la simbiosis humano – computador.

Socket Abstracción de software por la cual dos programas pueden intercambiar información desde un flujo de datos.

Software Término general usado en principio para referirse a los datos digitalmente almacenados en un computador, como programas y otro tipo de información que es capaz de ser leído, procesado y escrito por éstas máquinas.

Stack Espacio de memoria de crecimiento en orden de pila (Primero en entrar, último en salir), en donde se almacena el registro de activación y variables locales de sub rutinas.

Struct Construcción del lenguaje C que permite la agrupación de dos o más variables de distinto tipo en un solo registro.

Subrutina División funcional de un programa al cual se accede desde un llamado o salto del puntero de instrucciones hacia otra posición diferente y aleatoria.

TCP/IP Sigla que representa a un conjunto de protocolos de comunicación usados para el Internet y otras redes similares. Es nombrado así por que está compuesto de dos protocolos. El Transmission Control Protocol (TCP) y el Internet Protocol (IP) o protocolo de Internet.

Thread Hilo en inglés, se refiere a una unidad de ejecución de un programa que corre independiente al flujo principal de éste.

UDP User Datagram Protocol, por sus siglas en inglés, es un protocolo simple dentro del conjunto de protocolos TCP/IP, que no guarda estado de conexión ni segmenta los datos transmitidos.

UNIX Conjunto de sistemas operativos que conforman el estándar UNIX, el cual tiene como finalidad que el corazón del sistema operativo opere similarmente al sistema UNIX original, desarrollado por AT&T en 1969.

Userspace Segmento del sistema operativo que opera de tal manera que no interfiera con el núcleo o *kernel.*

VBlank *Vertical Blank* por su contracción en inglés, se refiere la generación de interrupciones del procesador a partir de un intervalo de actualización de pantalla en algunos sistemas de videojuegos

Virtualización Se refiere a la creación de una versión virtual de algún *hardware* o sistema real, ya sea una plataforma completa o alguna de sus partes como sistema operativo, sistemas de almacenamiento o interfaces de red.

VM Véase máquina virtual.

VMM Virtual Machine Monitor por sus siglas en inglés. Véase *hipervisor.*

WINE Wine es una aplicación de software libre que se enfoca en permitir a computadoras con sistemas Unix ejecutar programas escritos para el sistema operativo Microsoft Windows. Wine provee también una librería de software conocida como Winelib, con la cual los desarrolladores pueden compilar aplicaciones y portarlas a UNIX.

X64 Extensión de la arquitectura x86 para la ejecución de instrucciones de 64 bit.

X86 El término x86 se refiere a una familia de arquitecturas de conjunto de instrucciones (ISA) basado en el procesador Intel 8086. El término se derivo del hecho de que los nombres de muchos procesadores primitivos que eran compatibles con el 8086 terminaban también en 86, Esta arquitectura ha sido implementado por Intel, Cyrix, AMD y VIA entre otros.

RESUMEN

La virtualización de plataformas computacionales ha tomado un impulso significante desde inicios del presente siglo a nivel mundial, ese crecimiento en su utilización se debe a que ha simplificado la administración de sistemas a través de la centralización de los mismos, permitiendo un crecimiento horizontal de servidores y plataformas, en las que múltiples de estas máquinas virtuales pueden existir y ser ejecutadas sobre una misma plataforma de *hardware* –no necesariamente de la misma arquitectura--, aisladas entre sí, sin que el funcionamiento de una pueda verse afectado de manera adversa a la otra; simplificando, además, los costos de adquisición, mantenimiento y desarrollo.

Muchos tipos de máquinas virtuales pueden existir no sólo dentro de una máquina física de la misma arquitectura, ya que, por medio de técnicas de emulación, se puede virtualizar arquitecturas cruzadas, permitiendo además la compatibilidad binaria entre éstas.

De allí la importancia de que la interpretación entre diferentes códigos binarios de distintas arquitecturas, no sólo debe de ser consistente, sino eficiente.

Dado que, tanto la virtualización entre plataformas del mismo tipo de arquitectura, como la emulación para virtualizar otros tipos de arquitectura usando técnicas tradicionales, representa un gran costo en eficiencia para las máquinas físicas, se requiere de un enfoque diferente para su manipulación.

Las soluciones más factibles en la actualidad ha sido la paravirtualización entre una misma arquitectura, y la recompilación dinámica para arquitecturas cruzadas.

Este estudio se ha elaborado una evaluación comparativa entre los métodos de emulación y virtualización más eficientes para un algoritmo de alta complejidad computacional, contrapuestos con el método tradicional por interpretación y su ejecución nativa en la máquina física. Se concluye que la eficiencia alcanzada por la paravirtualización es significativamente más eficiente para la virtualización de una misma plataforma, y la recompilación dinámica para la virtualización entre arquitecturas diferentes.

OBJETIVOS

General

Recopilar, comparar, analizar y seleccionar las técnicas más eficientes de recompilación dinámica usada en emuladores y máquinas virtuales, con énfasis en la eficiencia temporal de su ejecución.

Específicos

1. Dar a conocer el funcionamiento general de una máquina virtual y un emulador.

2. Clasificar las máquinas y emuladores según su propósito y código destino.

3. Estudiar las fases por las que pasa un emulador y una máquina virtual genérica al momento de ejecutar código interpretado.

4. Investigar las diferentes técnicas de traducción de código ejecutado por las máquinas virtuales.

5. Analizar la eficiencia asintótica temporal y espacial de cada una de las técnicas y algoritmos de traducción previamente investigadas.

6. Seleccionar la apropiada técnica de optimización para cada máquina virtual según su clasificación de propósito y código destino, basados en implementaciones de código abierto.

INTRODUCCIÓN

Una máquina virtual es entendido formalmente como una máquina virtual completa en Turing (máquina invitado) capaz de ejecutar instrucciones escritas para ésta, dentro de otra máquina de Turing completa, que puede ser física o virtual (huésped).

La máquina virtual invitado puede manejar su propio conjunto de instrucciones, que no necesariamente sean las mismas que las de la máquina huésped, pero quedando siempre limitado al dominio computacional del huésped. En el mundo real, ésta solución entre relación invitado/huésped ha permitido generar soluciones bastante eficaces en cuanto a disponibilidad en servidores.

Aislando de esa manera diversas funcionalidades dentro de máquinas invitadas; que si sucediere algo nocivo a ésta, no afectase las funciones ejecutadas por otras invitadas dentro del mismo huésped, ni tampoco afectar al huésped (virtualización).

Provee también al huésped, a través de la máquina invitada, ejecutar código que su arquitectura no soporte (emulación), ampliando el dominio de compatibilidad tanto entre sistemas viejos y nuevos, y en otros casos, permitir la creación de máquinas virtuales capaces de ejecutar un código universal y traducirlo al código máquina (Compiladores JIT), separando las capas de *hardware* y software para que provea una mayor portabilidad al momento de generar código y no depender del conjunto de instrucciones de *hardware*.

Todo lo anterior se puede decir que son los beneficios de la emulación, sin embargo, éste cuenta con un grave problema; la eficiencia, ya que cuando la máquina virtual invitada ejecuta código, lo que hace es que tiene que traducirlo y enviarlo de forma comprensible para que la máquina huésped lo pueda entender, lo que provoca una sobrecarga sobre ésta, haciendo que el código se ejecute más lento cuando esta sobre la máquina virtual invitado, al contrario que si éste fuera compilado para la máquina huésped.

En esta investigación se pretende hacer un estudio teórico de las diversas técnicas de optimización de código para acelerar esa sobrecarga que recibe la máquina huésped cuando ejecuta código de la máquina virtual invitada, Identificando los aspectos y características, tomando el cuenta la finalidad de la máquina virtual, que puede ser ya sea un emulador de otra arquitectura, un emulador para la misma arquitectura, o un intérprete de código intermedio.

1. INTRODUCCIÓN A LA EMULACIÓN Y VIRTUALIZACIÓN

El estudio de la emulación y virtualización requiere conceptos básicos como los que a continuación se ejemplifica.

1.1. Conceptos básicos

Existen dos maneras principales de ejecución de código: por compilación o por interpretación.

1.1.1. Compilador e intérprete

En ciencia computacional, un compilador es un programa que genera código desde una entrada, escrito en un lenguaje "fuente", para traducirlo a un programa equivalente, en otro lenguaje, llamado lenguaje destino.[1] Un intérprete puede manejar el código sin necesidad de producir una salida binaria en disco, permitiendo mayor manipulación en memoria de la que es posible realizar en un binario estático. Entre estos intérpretes están las máquinas virtuales.

1.1.2. Máquina virtual

Una máquina virtual es un intérprete, que mezcla cualidades de compilador, al crear un entorno adecuado para la ejecución de código, aislado de la máquina real, pero dependiendo de los recursos de ésta.

1 AHO, SETHI, ULLMAN – Compilers, Principles, Techiniques and Tools – 1986 – Wesley p.2

Éste puede leer un conjunto de instrucciones que puede ser ajeno al de la máquina real, y tambien estar compilándolo internamente hacia el conjunto de instrucciones que la máquina real es capaz de entender.

La ventaja de las máquinas virtuales es su portabilidad, ya que permite crear código capaz de ser leído en diferentes arquitecturas, con tan solo portar la máquina virtual a la otra arquitectura.

Una de las principales desventajas, es la ineficiencia de estas, ya que al tener que generar código dinámicamente para otra arquitectura, el código ejecutado puede ser dos o más veces lento en orden de magnitud que si el código fuera compilado para la máquina real.

1.1.3. Emulador

La máquina virtual descrita con anterioridad, está fundamentada en el concepto de emulador, el cual es un programa de software capaz de imitar funcionalidades de *hardware* –principalmente el CPU (Unidad Central de Procesamiento, *Central Processing Unit* por sus siglas en inglés)-- con una arquitectura diferente a la máquina en donde se ejecuta, y actuar como aquella al ejecutar código que haya sido previamente compilado para dicho CPU a imitar.[2]

El concepto de emulador se fundamenta en la tesis de Church-Turing, la cual dice que cualquier entorno operativo computacional podrá ser emulado dentro de otro.[3]

2 Fuente: http://en.wikipedia.org/wiki/Emulator, fecha de consulta: 4 de enero de 2010
3 Fuente: http://es.wikibooks.org/wiki/La_tesis_de_Church-Turing/TextoCompleto, fecha de consulta: 4 de enero de 2010

1.2. Niveles de abstracción

Debido a los costoso que es la emulación y la ejecución de programas en máquinas virtuales, se debe enfatizar en el máximo aprovechamiento de los recursos del CPU disponibles.

Para aumentar la eficiencia en la complejidad computacional se hace uso de niveles de abstracción separados por interfaces bien definidas.

Esto ha permitido que se logre manejar capas de nivel más bajo (nivel máquina) desde capas de nivel más alto (cercanos a la abstracción humana), sin importarle a la capa más alta el funcionamiento interno de la capa mas baja, sino, que, por medio de interfaces bien definidas se logra establecer comunicación entre ambas capas de manera efectiva.

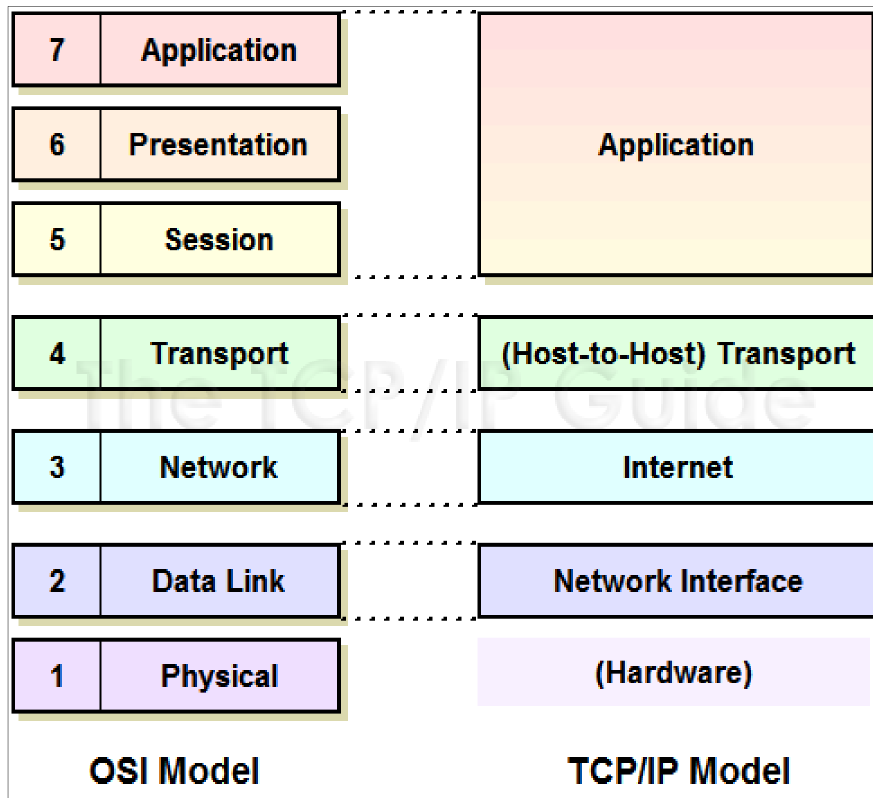
1.2.1. Niveles de abstracción, caso TCP/IP

Como ejemplo se puede tomar el modelo TCP/IP, el cual utiliza 4 capas sobrepuestas lógicamente, que equivale a las 7 capas del modelo OSI, y que, en el ascenso de niveles, logra mayor abstracción, teniendo en el nivel más bajo el *Hardware*, que comprende todos los dispositivos físicos.

Ascendiendo por la capa de la Interfaz de red, la cual, permite enlazar 2 dispositivos de red por medio de direccionamiento físico, elevando más el nivel de abstracción.

Encima de éste se encuentra la capa de Internet IP, que comprende el direccionamiento lógico y enrutamiento, elevando aún más la abstracción. Continuando con la capa de Transporte TCP, la cual segmenta de manera lógica el contenido que ha sido resuelto por dirección desde la capa de IP, para finalizar con la capa de Aplicación, que es controlada específicamente por las aplicaciones finales a través de *Sockets* TCP o UDP (figura 1).

Figura 1. **Niveles de abstracción en capa OSI/TCP**

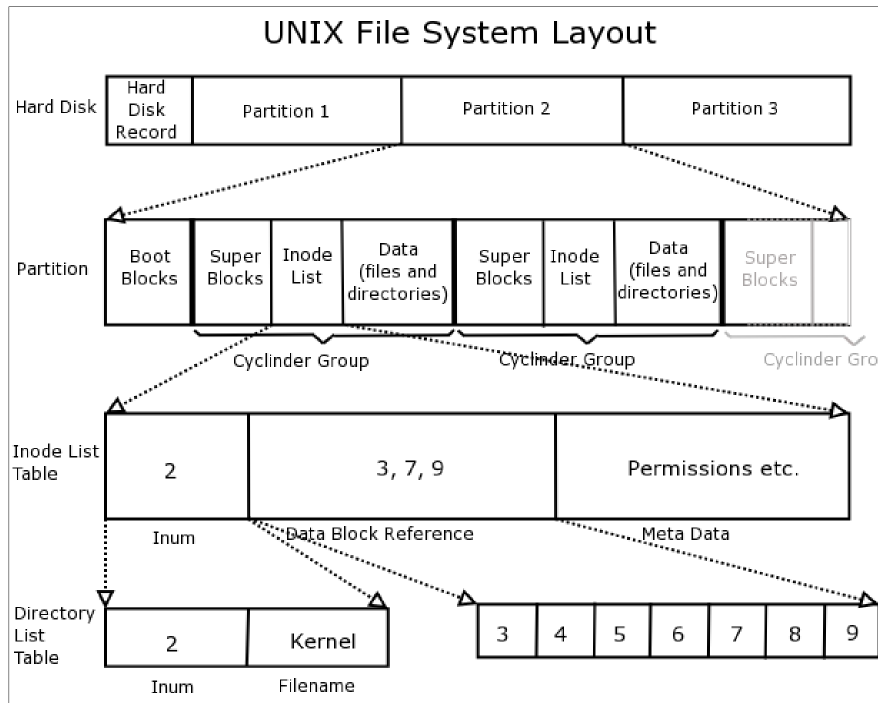


Fuente: *TCP IP Architechture and TCP IP Model,* disponible en http://www.tcpipguide.com/free/t_TCPIPArchitectureandtheTCPIPModel-2.htm, fecha de consulta, 26-08-2009.

1.2.2. Niveles de abstracción, caso sistema de archivos en UNIX

Otro ejemplo de abstracción de capas se encuentra en el manejo de dispositivos de almacenamiento primario, específicamente discos duros, los cuales físicamente están representados por sectores y pistas, para ser abstraídos por el sistema operativo y convertidos en *iNodos* y otras estructuras de datos, que a su vez, en una capa superior de aplicación, terminan siendo archivos y directorios (figura 2).

Figura 2. **Organización por capas del sistema de archivos en UNIX**



Fuente: *Linux Internals*, disponible en http://learnlinux.tsf.org.za/courses/build/internals/internals-all.html. fecha de consulta, 26-08-2009.

Los niveles de abstracción siempre representan un orden jerárquico desde el *hardware* hasta el *software*, en donde, en *hardware* se tiene todos aquellas características físicas del dispositivo y de su comunicación con otro *hardware*, en muchos casos el procesador. Por otro lado, en nivel software, se tiene todo aquellas características lógicas y su comunicación entre otras aplicaciones y el mismo sistema operativo.

1.2.3. Interfaces bien definidas

El siguiente aspecto para manejar la complejidad efectivamente son las interfaces bien definidas.

Estas son canales de comunicación que permiten desacoplar o separar los equipos que trabajan en los aspectos lógicos (*software*) de los equipos que trabajan en los aspectos físicos (*hardware*) de la computadora, lo que ha permitido aumentar la especialización en ambos ramos.

Los *opcode,* o códigos de operación, son una manera de desacoplamiento efectivo bajo comunicación de interfaces bien definidas; es decir, el conjunto de instrucciones que poseen los microprocesadores para manejar las operaciones lógicas y matemáticas que realicen con sus datos.

A través de este conjunto de instrucciones, un programador no tendría que preocuparse de cómo el hardware tenga implementado las funciones aritméticas o lógicas, así como del acceso, asignación y desasignación de datos que se vayan a procesar.

El programador debe estar solamente enterado en cómo poder comunicarse con el procesador, y en este caso, se define un lenguaje para ello, dicho lenguaje estaría compuesto por el conjunto de instrucciones del microprocesador.

Tabla I. **Estructura de los códigos de operación para un procesador PowerPC**

Formato /offset	**0**	**6**	**11**	**16**	**21**	**26**	**30**	**31**
D-form	opcd	tgt/src	src/tgt	immediate				
X-form	opcd	tgt/src	src/tgt	src	extended opcd			
A-form	opcd	tgt/src	src/tgt	src	src	extended opcd		Rc
BD-form	opcd	BO	BI	BD			AA	LK
I-form	opcd	LI					AA	LK

Fuente: elaboración propia.

Asumiendo que el programador no desee ni siquiera tener que ejecutar interrupciones del procesador para que su código fuese capaz de acceder a otro *hardware* ajeno al microprocesador, se podría agregar otra capa intermedia entre el *hardware* y la aplicación, la cual corresponde al sistema operativo, en donde el programador solo tendrá que comunicarse a más alto nivel con el sistema operativo, para que éste, asigne los recursos necesarios en dicha aplicación, y a la vez, a través de API de controladores, permitiese a éste acceso a más alto nivel al *hardware* encapsulando de cierta manera la complejidad subyacente de una directa comunicación con el *hardware*.

Ya estando los niveles de abstracción bien comunicados bajo interfaces bien definidas, se asume que es posible lograr hacer interfaz con otro sistema operativo como aplicación dentro de un sistema operativo, De ahí viene el aprovechamiento de las máquinas virtuales como aplicación. pues permiten flexibilidad al aislar y emular dispositivos virtuales, que se comunican con *hardware* real a través del sistema operativo sobre el que se está ejecutando.

Al ser estos dispositivos emulados, es posible entonces, crear un ambiente de *hardware* virtual sobre el que se pueda ejecutar otro sistema operativo que permita ejecutar aplicaciones, que estarán aislados del sistema operativo sobre el que corre la máquina virtual.

De esta manera evitando que si éstas colapsan, no colapse la plataforma sobre la que se ejecute la máquina virtual, y de esta manera, presentando una de las más grandes ventajas de la utilización de máquinas virtuales, la disponibilidad.

Debido a las características de emulación de *hardware* que presentan las máquinas virtuales, éstas no se limitan a ejecutar código de una misma arquitectura que la máquina huésped, pues, usando técnicas de emulación de *hardware* entre arquitecturas, permiten compatibilidad de software entre diferentes arquitecturas.

Por ejemplo, es posible tener una máquina virtual PowerPC ejecutándose sin problemas sobre una arquitectura x86, para lo cual evitaría desde la recompilación de un programa, hasta la codificación completa desde cero si esta hecho en un lenguaje no portable.

Sin embargo, éste no es el único tipo de máquina virtual que existe, pues, como se expresó previamente, también se tienen máquinas virtuales que emulen una máquina no física; en este caso, un entorno aislado y manejado por esta misma máquina virtual, la cual asigna sus recursos y accede a los dispositivos, y que cuente con su propio conjunto de instrucciones ajeno a cualquier otro conjunto de instrucciones de *hardware* existente. Un ejemplo de esto es la máquina virtual de Java.

1.3. Arquitectura de computadoras

Al emular una máquina virtual, es necesario tomar en cuenta la arquitectura de la máquina que se pretende interpretar –a la que se le suele llamar *Máquina Invitada*--, así como la máquina en donde se pretende ejecutar dicha máquina virtual –llamada comúnmente máquina huésped o anfitrión--.

Cuando se menciona las arquitecturas de procesadores se hace referencia únicamente a la funcionalidad, sin entrar mucho en detalle acerca de su funcionamiento interno (funcionamiento electrónico de más bajo nivel); definiéndose así como el conjunto de interfaces y el comportamiento lógico de sus recursos por medio de dichas interfaces.

Las arquitecturas pueden tener más de una implementación. Por ejemplo los diferentes modelos de la arquitectura *x86*, ya sea de una u otra marca, línea y modelo, todos tienen diferentes características, pero corresponden a una misma arquitectura base.

Los niveles de abstracción fácilmente se identifican como capas de implementación, que, por medio de interfaces bien definidas comunica cada arquitectura subyacente en cada capa, permitiendo comunicar entre aplicaciones y librerías por medio de interfaces.

Asimismo se logra comunicar estas aplicaciones y librerías con el sistema operativo subyacente, mientras, el mismo sistema operativo, el cual por medio de puertos de entrada y salida definidas por el *hardware* en su especificación, logra comunicarse con los dispositivos por medio de sus interfaces.

También el sistema operativo, logra comunicarse con la memoria principal por medio de los modos de direccionamiento soportados tanto por el microprocesador como por el mismo diseño del sistema operativo.

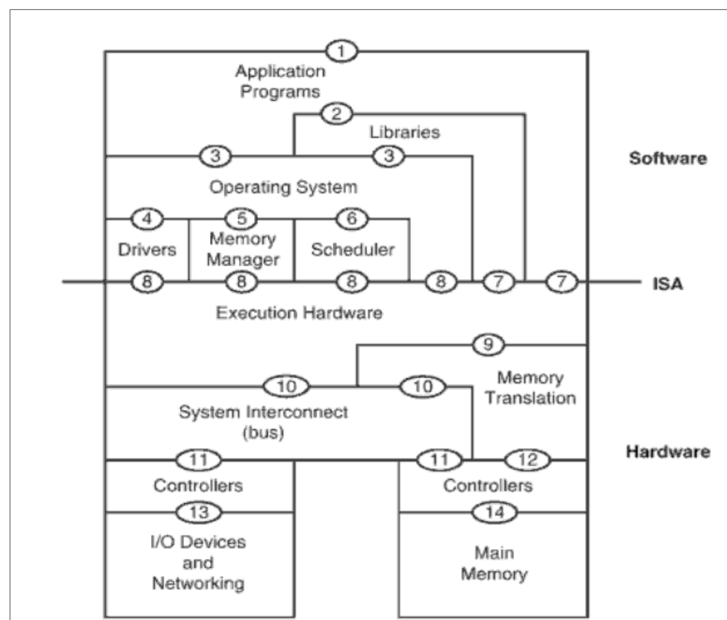
1.3.1. La arquitectura de conjunto de instrucciones ISA

La arquitectura de conjunto de Instrucciones (ISA, Instruction Set Architecture por sus siglas en inglés, Arquitectura de Conjunto de Instrucciones), utilizada desde la serie de Mainframes IBM 360, define una serie de interfaces entre el *hardware* y el software.

Su finalidad fue demostrar la necesidad de diferenciar y separar a estos dos componentes del computador, y a su vez, hacer énfasis en la compatibilidad de software para que éste sea ejecutado entre las diferentes implementaciones de la arquitectura que fue diseñada en un principio.

La arquitectura ISA está compuesta por dos modos de utilización o espacios: El espacio de Usuario o U*serspace*, conocido también *User ISA*, tiene como finalidad el intercambio de información entre el humano, y el *software*, por lo que es solo visible para las aplicaciones finales (figura 3).

Figura 3. **Esquema de separación en capas definido en la arquitectura ISA**



Fuente: *Virtual machines*: *versatile platforms for systems and processes* p 7.

Por otro lado, espacio de supervisor o *Kernel*, conocido también como *System ISA*, permanece invisible para las aplicaciones finales y es únicamente manipulado por el sistema operativo, en la que se realizan tareas de administración de recursos de *hardware* a bajo nivel, y que por lo tanto, solo le incumbe al entorno operativo llevar control sobre éstas.

Otra interfaz tratada en el ISA es el ABI (*Application Binary Interface* o Interfaz de Aplicaciones Binarias), la cual provee a una aplicación final el acceso a recursos de *hardware* de manera indirecta por medio de un conjunto de instrucciones específico, invocando llamadas predefinidas por el sistema operativo.

Esto permite al sistema operativo establecer prioridades en las llamadas de los programas al *hardware*, y autorizando a la aplicación final para su acceso. Normalmente es llamada por medio de una instrucción que transfiere el control al sistema operativo parecido a las llamadas de subrutinas. Pasando argumentos regularmente por el *Stack* o por los registros del mismo procesador.

Las ABI cubren detalles como el tipo de dato, tamaño, alineación de bits (*big endian* o *little endian*), la convención de llamada –la cual dictamina como se pasan los argumentos a las funciones, y los valores de retorno de las mismas--, el número de llamadas al sistema y como una aplicación debe hacer uso de las llamadas del sistema. En muchos casos cubre hasta el formato binario de los archivos objeto, es decir, los archivos binarios que serán leídos por el sistema operativo y serán pasados al procesador. [4]

Una implementación de ABI bastante común es la Especificación de Compatibilidad Binaria de Intel, o iBCS por sus siglas en inglés, desarrollada por desarrolladores de Unix para x86, la cual permite compatibilidad binaria entre distintas implementaciones. Siendo sustituida después por implementaciones específicas como el ABI de Linux.[5]

---

4 Fuente: *ABI*, disponible en http://en.wikipedia.org/wiki/Application_binary_interface, fecha de consulta: 7 de enero de 2010

5 Fuente: *iBCS*, dispoinble en http://www.everything2.com/index.pl?node=iBCS, fecha de consulta: 7 de enero de 2010

En los niveles mas altos de las capas, se encuentra la interfaz de programación de aplicaciones o API, la cual entra mas en el contexto de lenguajes de alto nivel.

Entre estas interfaces esta la librería estándar, la cual es una interfaz de más alto nivel para las llamadas del ABI para invocar los servicios más comunes del sistema operativo subyacente.

Esto permite mayor portabilidad entre diferentes sistemas que usen la misma API, con tan solo recompilar la aplicación final. Como ejemplo de estas podemos ver la librerías estándar del lenguaje C, también conocidas como *clib* en diversos sistemas operativos basados en Unix

1.4. Taxonomía de máquinas virtuales

Las máquinas virtuales, según sus características operativas y la arquitectura de conjunto de instrucciones que interprete, se clasifican en:

- Máquinas virtuales de Proceso: Aquellas que cuentan con ABI, éstas se subdividen en.

    - Máquinas virtuales de Proceso con el mismo ISA: *Sistemas multiprogramados* o sistemas que no necesitan *recompilación dinámica*, solamente optimización binaria, y se ejecuta de forma nativa.

    - Máquinas virtuales de Proceso con diferente ISA: Sistemas que necesitan recompilación dinámica.

- Máquinas virtuales de sistema: Aquellas que soportan un ISA completo, las que se subdividen en:

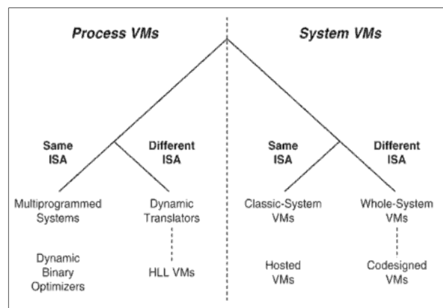
  - Máquinas virtuales de Sistema con el mismo ISA: Proveen entornos aislados y replicados del sistema huésped. Pueden ser de tipo:

    - ✓ Clásicos: No varia mucho la implementación del manejo de memoria virtual (VMM).

    - ✓ Hospedados: Tiende a variar la implementación del manejo de memoria virtual (VMM).

  - Máquinas virtuales de Sistema con diferente ISA: Máquinas virtuales de todo el sistema en donde la eficiencia temporal es secundario, importando más la exactitud de la emulación de la máquina virtual a ejecutar (figura 4).

Figura 4. **Jerarquía de máquinas virtuales según su interacción con el sistema fuente**



Fuente: SMITH *Virtual machines: versatile platforms for systems and processes,* p. 23.

2. EMULACIÓN E INTERPRETACIÓN

Una máquina virtual ya sea de proceso o de sistema, implica que cuando su implementación se encuentre en un ISA diferente, requiera traducción dinámica, --es decir, traducción de un ISA a otro, procurando mantener la misma funcionalidad y de ser posible la sincronización--, representando un reto aún mayor, ya que implica la conversión desde un conjunto de instrucciones específico de una arquitectura hacia la arquitectura huésped donde se ejecuta el entorno virtual.

Al emular un conjunto de instrucciones de una arquitectura a otra, todas las aplicaciones compiladas para el ISA de la máquina virtual deberán ser recompiladas dinámicamente al ISA huésped.

2.1. Introducción

Como ejemplo de emulación en máquinas virtuales de proceso, se puede tomar el sistema Digital FX 32, el cual ejecuta aplicaciones compiladas para la arquitectura Intel IA-32 en un sistema operativo Windows NT ejecutándose en un procesador con arquitectura DEC Alpha.

El método más directo y simple para la emulación de otro sistema es la interpretación, el cual consiste en un programa que ejecuta para la arquitectura destino, el proceso de obtención, decodificación y ejecución (ciclo *Fetch-Decode-Execute*) para la máquina virtual.

Por su complejidad temporal al interpretar cada código de operación de uno en uno, éste requiere mayor poder de procesamiento, e implica que es el proceso más lento, requiriendo hasta cientos de instrucciones de la arquitectura huésped solo para traducir una sola instrucción de la arquitectura a virtualizar.

Para mejorar el rendimiento, se debe utilizar métodos como el de traducción binaria o recompilación dinámica, que consiste en obtener bloques de el código de la arquitectura a emular, y convertir éstos en código equivalente para la máquina huésped, donde puede existir mucha sobrecarga en el proceso de traducción de bloques, pero una vez haya sido traducido dicho bloque de código, no será necesario volver a recompilarlo, permitiendo que en posteriores llamadas a dicho bloque, aumentando la eficiencia en ejecución del código previamente traducido que se haya almacenado en el caché de memoria.

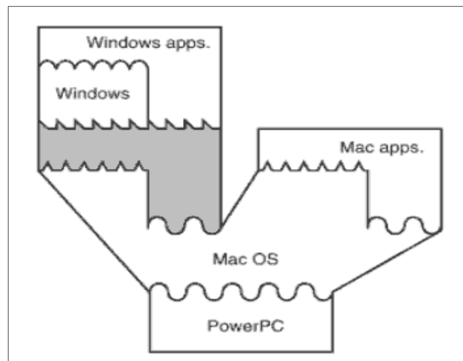
Como comparativa entre las diferencias en rendimiento de la interpretación contra la traducción dinámica, se debe mencionar que la interpretación tiene relativamente poca sobrecarga al momento de iniciar la traducción, pero que, a lo largo de toda la ejecución es un proceso lento. Por otro lado con la traducción binaria, se tiene una alta sobrecarga al iniciar la ejecución, pero baja sobrecarga durante toda la ejecución.

Respecto las máquinas virtuales de sistema completo, --cuando es necesario no solo la ejecución de aplicaciones dentro de un sistema operativo de otra arquitectura, sino también emular dicho entorne operativo entre diferentes ISA--, no solo es necesario emular el código de la aplicación, sino todo el entorno operativo incluyendo *hardware*.

En esos casos la interpretación no es opción, pues es demasiado costosa en ciclos de procesador, por lo que la traducción binaria juega un papel importante dentro de ésto.

Un ejemplo de máquinas virtuales que implementan traducción binaria son las soluciones de virtualización entre plataformas como Qemu o *PearPC* que permite utilizar sistemas de legado Mac OS basados en PowerPC en una arquitectura x86, o el producto comercial *VirtualPC for Mac,* que permite realizar la operación inverso, utilizar sistemas operativos basados en x86 dentro de uno basado en PowerPC (figura 5).

Figura 5. **Funcionamiento de una máquina virtual de proceso que emula un entorno operativo x86 dentro de una PowerPC**



Fuente: SMITH *Virtual machines: versatile platforms for systems and processes,*, p. 21

Muchas implementaciones de máquinas virtuales se basan en emulación, por lo que, formalmente, se define la emulación como el proceso de de implementar las interfaces y funcionalidades de un sistema o subsistema hacia otro sistema con diferentes interfaces y funcionalidades.

La emulación del conjunto de instrucciones es una principal característica en la implementación de las máquinas virtuales ya que la máquina virtual debe soportar un programa compilado de forma nativa para un conjunto de instrucciones diferente al del entorno donde este se ejecuta.

La máquina virtual deberá ejecutar código de un conjunto de instrucciones invitado o fuente, hacia otro conjunto de instrucciones huésped o destino, reproduciendo así su comportamiento en la máquina huésped.

2.2. Interpretación básica

La ciencia computacional define formalmente a un intérprete, como un programa capaz de leer una cadena infinita de instrucciones, y ejecutarla progresivamente mientras la lee.

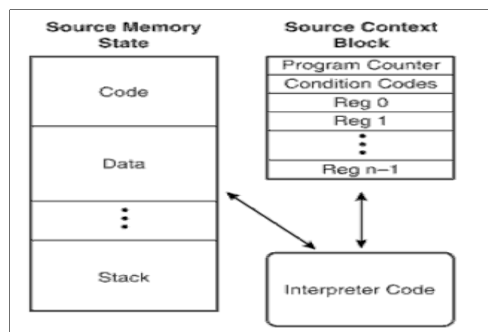
La interpretación de código se ha llevado a cabo desde que se inició la definición de los lenguajes formales de programación. Los ejemplos más comunes van desde LISP pasando por Smalltalk, y finalizando en Javascript y Python*;* donde los intérpretes han logrado facilitar la implementación de lenguajes dinámicos debido a sus versatilidad al ejecutar código residente en memoria en forma de representaciones intermedias y estructuras de datos.

Debido a ello se logra bastante facilidad para ejecutar semánticamente construcciones de lenguaje, que, de otra manera, sería bastante complicado hacerlos en un lenguaje compilado de una pasada, como por ejemplo las funciones lambda en los lenguajes funcionales y algunos de múltiples paradigmas.

Un intérprete emula y opera por completo un estado de una máquina implementando un ISA fuente, hacia una máquina con ISA destino, incluyendo todos los registros y la memoria principal de dicho estado.

Para ello el emulador mantiene en una región de memoria de la máquina huésped tanto el código como el área de datos, y mantiene una *tabla de contexto* que, contiene además los registros de propósito general, el puntero de código o *program counter*, código de condiciones y otros registros varios (figura 6).

Figura 6. **Interacción y estructura en memoria de un intérprete**



Fuente: SMITH *Virtual machines: versatile platforms for systems and processes*, p. 30

El modo de operación normal, llamada *decode-and-dispatch* de los intérpretes es leer instrucción por instrucción el código de la fuente, y mientras lee, modificar el estado en memoria de la máquina fuente de acuerdo a la instrucción que se este ejecutando. Ésto se encuentra dentro de un ciclo infinito central que decodifica la instrucción y luego la despacha a la rutina de interpretación según la instrucción leída.

En pseudocódigo se puede representar al ciclo principal como se muestra en la figura 7.

Figura 7. **Ejemplo del motor de un intérprete**

```
// Ciclo principal
mientras(!halt && !interrupt){
      inst = codigo[pc];
      opcode = extraer(inst, 31, 6);
      seleccionar(opcode){
            caso LoadWordAndZero: LoadWordAndZero(inst);
            case ALU: ALU(inst);
            case Branch:  branch(inst);
            ….
      }
}
// Ejemplo de funcion para un codigo de operacion
LoadWordAndZero(inst){
      RT = extraer(inst, 25, 5);
      RA = extraer(inst, 20,5);
      desplazamiento = extraer(inst, 15, 16)
      si (RA == 0) fuente =0;
      sino fuente = regs[RA];
      dir = fuente + desplazamiento;
      regs[RT] = (data[direccion]<<32) >> 32;
      PC = PC + 4;
}
```

Fuente: elaboración propia.

En un lenguaje de alto nivel se observa el proceso de decodificación y despacho, en el cual se puede verificar que el ciclo principal esta ejecutado dentro de la construcción *mientras* (*while* en inglés), y se coloca de ejemplo el código de la función *LoadWordAndZero*, básica en los procesadores PowerPC, el cual carga una palabra de 32 bits hacia un registro de 64 bits, y anula los primeros 32 bits más significativos.

En el proceso de decodificación, el contador de programa está almacenado en la variable PC, el cual es un índice hacia un arreglo que contiene la fuente binaria. La palabra apuntada por ese índice es la instrucción fuente que necesita ser ejecutada.

El campo *Opcode* representa el código de operación o instrucción a ejecuta, representado por un campo de 6 bit iniciando desde el bit 31. y es extraído usando una combinación de desplazamiento y enmascarado por la función *extraer*.

Los campos designativos de registro se usan como índices para determinar el valor real de los operandos, y a menos que la mera instrucción reasigne el contador de programa, como lo puede llegar a hacer *opcode Branch*, el PC continuará incrementándose hasta llegar a la siguiente instrucción secuencial antes de regresar a la rutina principal de decodificar y despachar.

El proceso de interpretación es directo y sencillo, sin embargo el costo en rendimiento es alto, incluso si el intérprete estuviese escrito en lenguaje ensamblador.

Por ejemplo, para una única instrucción como *LoadWordandZero* se necesitó de al menos 8 lineas de código de alto nivel, se traducen en más de 8 instrucciones de bajo nivel.

Otra manera alterna de efectuar el proceso de *decode-dispatch* consiste en descentralizar el proceso de decodificación del ciclo principal, y asignárselo a cada *dispatch*, de manera que dentro del procedimiento donde se interpreta el *opcode*, al final se le adjunte la operación de *dispatch*, y por medio de una tabla de *opcodes*, se pueda “saltar” al finalizar la interpretación del *opcode* hacia el nuevo *opcode*, de esa manera se elimina la necesidad de utilizar un ciclo centralizado, y permite atar uno a uno cada *opcode* durante la ejecución. A esto se le llama método directo *Threaded*.

Sin embargo, aunque se haya eliminado el ciclo centralizado de decodificado y despacho, todavía queda como problema de eficiencia la búsqueda en la tabla de *opcodes*, pues aún es un lugar en donde está centralizada parte de la ejecución. Aparte de que aún se cuenta con la ineficiencia de tener que extraer los campos de instrucción, ya que se repite el proceso cada vez que se ejecuta un *opcode*.

2.2.1. Interpretación de arquitecturas CISC

Los ejemplos previos se han hecho en base a un conjunto de instrucciones basado en la Arquitectura con un Conjunto de Instrucciones Reducido (RISC por sus siglas en inglés); implementado en la ISA PowerPC.

Ésta arquitectura, en general, tienen la ventaja de tener un formato regular para las instrucciones, es decir, comúnmente, todas las instrucciones tienen la misma longitud, en donde los registros casi siempre están posicionados de la misma manera, y en el mismo segmento.

Sin embargo, la arquitectura comercialmente más exitosa para computadores de consumo general, es la CISC, o Arquitectura del conjunto completo de instrucciones, específicamente las implementaciones hechas por Intel, conocida como *x86* o su variante de 64 bits *x64/x86-64/amd64.*

Tiene la desventaja de utilizar instrucciones de longitud variable, e incluso de campos dentro de la instrucción con longitud variable.

Esto fue hecho de tal manera de mantener compatibilidad binaria entre diferentes versiones de microprocesadores a lo largo de la evolución de la arquitectura, lo que ha permitido que código realizado para un procesador antiguo 386 corra sin modificaciones sobre un procesador basado en los modelos *Core 2*[6].

Tambien ha permitido la evolución del tamaño de palabra del procesador, de 16, 32 o a 64 bits, manteniendo el código compatible con mínimas modificaciones.

2.2.2. Caso de estudio de emulador basado en intérprete, ZSNES

Zsnes es un emulador de código abierto de la consola de videojuegos Super Nintendo exclusivamente para máquinas x86.

Dicho emulador está escrito mayormente en *Assembler*, con componentes escritos también en C y C++ para la arquitectura Intel x86.

6 Esto es aplicable siempre que se hable de código compilado para 32 bit.

Las características del CPU que emula ZSNES son las siguientes:

- Procesador:
    - Ricoh 5A22, basado en un núcleo 65c816 a 16-bit
- Ciclos del Reloj
    - Entrada:21.47727 Mhz
    - Bus: 3.58 MHz, 2.68 MHz, 1.79 Mhz
- Buses
    - Bus de dirección de 24-bit and 8-bit, Bus de datos de 8-bit
- Otros
    - DMA y HDMA, IRQ Temporizado

El procesador personalizado '5A22', producido por Ricoh; está basado en el procesador CMD/GTE 65c816 (Un clon del procesador WDC 65816), el cual tiene un núcleo de ejecución 65c816 con bus de velocidad variable, y tiempos de acceso determinados por la dirección a la cual se accede, con una velocidad máxima teórica de de 3.58 Mhz.

Además el CPU tiene acceso a 128 KB de RAM de trabajo.

- El CPU contiene otro tipo de *hardware* de soporte, incluyendo:
    - *Hardware* para interactuar con otros puertos de entrada
    - *Hardware* para generar interrupciones NMI en intervalos de *V-Blank*

    - *Hardware* para generar interrupciones IRQ dependiendo las posiciones de pantalla.
    - Unidad DMA soportando dos modos primarios, DMA general para transferencias por bloques a 2.68MB/s, y HDMA para transferir pequeños conjuntos de datos al final de cada línea de pantalla.
    - Registros de multiplicación y división por *hardware*.

Adicional al CPU, éste software tiene que emular además otro tipo de *hardware*, y procesadores personalizados encontrados dentro de los cartuchos, entre los que se encuentran:

- Modos gráficos 0,1,2,3,4,5,6,7
    - Soporte para gráficas de tamaño 8x8, 16x16, 32x32, y 64x64 (girados en cualquier dirección).
    - Mosaicos en 8x8 y 16x16.
    - Modos de mosaico en 32x32,64x32,32x64,64x64.
    - Efectos HDMA.
    - Modo 7 para rotación y escala.
    - Prioridades de fondo.
    - Prioridades de sprites.
        - ✔ Adición y sustracción del área de fondo.
        - ✔ Efectos de mosaico.
    - Sonido estéreo digital a 16 bit.

- CPU de sonido SPC700, con su propio conjunto de instrucciones.
- Procesador de efectos de sonido DSP entre los que se encuentran: efectos de eco, efectos de volumen.
- Efectos de ruido y modulación.

- Microprocesadores personalizados.

Tabla II: **Porcentaje de emulación de procesadores personalizados en ZSNES**

Procesador personalizado	**Porcentaje de ejecución**
C4	100%
Nintendo DSP-1	100%
Nintendo DSP-2	100%
Nintendo DSP-3	80%
Nintendo DSP-4	95%
OBC-1	100%
SA-1	90%
S-DD1	100%
Seta DSP 10	99%
Seta DSP 11	80%
SPC7110	100%
S-RTC	95%
SuperFX	90%

Fuente: *Zsnes Documentation,* http://zsnes-docs.sourceforge.net/html/readme.htm, fecha de consulta: 05-11-2009.

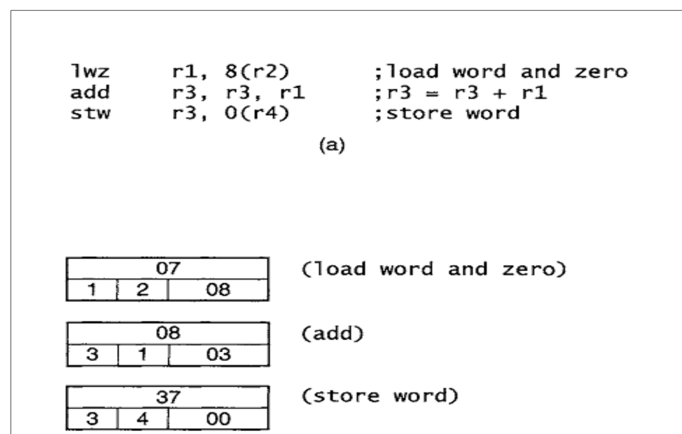
3. EMULACIÓN, RECOMPILACIÓN DINÁMICA Y PROCESAMIENTO EN PARALELO

El proceso de emulación por interpretación tradicional suele ser costoso, a continuación se presenta una solución elegante a este problema.

3.1. Predecodificado

Consiste en re-ordenar inicialmente cada instrucción de código a leer, de tal manera que se evite las operaciones de extracción que se ejecutaban antes, y, por medio de éste nuevo orden sea más rápido para la máquina huésped acceder a la operación actual al tener predefinidos los campos de longitud variable dentro de campos de un *struct* de longitud fija (figura 8).

Figura 8. **Ejemplo gráfico de predecodificado**



Fuente: SMITH, *Virtual machines: versatile platforms for systems and processes,* p 58

Esto implica además la creación de un código intermedio desde el programa fuente. Por ejemplo se puede tomar las instrucciones *Add (suma)*, y *Substract* (resta) de PowerPC, las cuales son una combinación entre un mismo *opcode* con bits de *opcode* extendidos.

Al incluirlos en una estructura con campos fijos se evita tener que extraer y ejecutar operaciones costosas de enmascarado y desplazamiento de bits en cada paso.

Las operaciones *branch* en la traducción hacia esta representación intermedia, el contador de programa destino (el de la representación intermedia) varia respecto al contador de programa fuente.

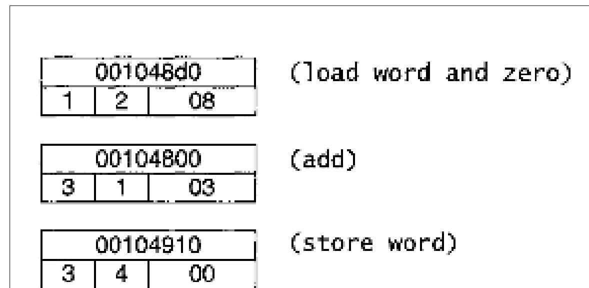
Esto no es mucho problema para el intérprete, pues en el destino van a ser código de tamaño fijo, y es suficiente solo re acomodar los punteros del contador de programa para que se vayan adaptando éstos al nuevo esquema.

3.2. Interpretación directa

Se le llama así al reemplazo de una o mas direcciones de una tabla de direcciones previo a llegar al segmento de destino, por la dirección real en memoria donde se encuentra dicho segmento.

Esto se hace debido a que el costo computacional al revisar dicha tabla es mucho más alto a que cuando ya se tiene reemplazadas todas las direcciones previamente (figura 9).

Figura 9. **Esquema en memoria del método de Interpretación directa**



Fuente, SMITH *Virtual machines: versatile platforms for systems and processes,* p 59.

Aunque este método suele ser más rápido, es necesario depender de la localidad exacta de dicho segmento de memoria donde se almacena la rutina, lo que limita la portabilidad, pues, si el intérprete es portado a otra arquitectura, habrá necesidad de volver a regenerar las direcciones para la máquina destino en donde se ejecutará.

Una solución a esto es colocar solo una dirección fija en una rutina base, y las demás direcciones serán relativas a éstas.

3.3. Traducción binaria o recompilación dinámica

La traducción binaria trata de mejorar el rendimiento de la interpretación, tomando en cuenta que con el pre decodificado se conoce que el código binario fuente es convertido a una representación intermedia, el cual, sus instrucciones son ejecutadas siempre por la rutina que la interprete de una misma manera.

Con la técnica de traducción, en vez de usar de llamar una rutina cada vez que se lea dicho código de operación, se traduzca directamente el código de operación al conjunto de instrucciones de la máquina destino[7].

Por ejemplo, para llevar a cabo la traducción binaria desde una instrucción x86 a PowerPC, se debe primero almacenar el conjunto de registros de estado del contexto del procesador leídos y almacenarlos directamente en registros del procesador de la arquitectura destino (PPC).

De tal manera que se pueda conservar los datos del registro de contexto en un registro r1, el puntero hacia la imagen en memoria de la máquina fuente r2, y el contador del programa r3. así que para traducir las instrucciones en x86 (figura 10) se tenga como resultado en PPC (figura 11).

Figura 10. **Fuente en x86**

```
addl %edx, 4(%eax)
movl 4(%eax), %edx
add %eax, 4
```

Fuente: elaboración propia.

7 Fuente: *Ngemu Forums*, disponible en http://forums.ngemu.com/web-development-programming/20491-dynamic-recompilation-introduction.html, fecha de consulta: 5 de diciembre de 2009

Figura 11. **Ejemplo de traducción a *opcode* PPC**

```
;addl %edx, 4(%eax)
lzw     r4,0(r1)       ; carga %eax desde el bloque de registros
addi    r5,r4,4 ;suma 4 a %eax
lwzx    r5,r2,r5       ; carga operando desde memoria
lwz     r4;12(r1)      ; carga %edx desde el bloque de registro
add     r5,r4,r5       ; hace suma
stw     r5,12(r1)      ; coloca resultado en %edx
addi    r3,r3,3 ; actualiza contador de programa (3 bytes)
;movl 4(%eax), %edx
lwz     r4,0(r1)       ;carga %eax desde el bloque de registros
addi    r5,r4,4 ;suma 4 a %eax
lwz     r4,12(r1)      ;carga %edx desde el bloque de registro
stwx    r4,r2,r5       ;almacena %edx en memoria
addi    r3,r3,3 ;actualiza el contador del programa
add %eax, 4
lwz     r4,0(r1)       ;carga %eax desde el bloque
addi    r4,r4,4 ;suma
stw     r4,0(r1)       ;devuelve resultado a %eax
addi    r3,r3,3 ;actualiza contador
```

Fuente: elaboración propia.

Para la traducción binaria es necesario contar con un mapeo uno a uno de los registros de la arquitectura fuente a los registros de la arquitectura destino. No sólo para los registros generales que se usen para datos, sino también para los registros que marcan el estado actual de ejecución como el registro del apuntador a *stack* o el contador de programa.

3.3.1. El problema de localidad y descubrimiento del código

El método de traducción binaria aplica únicamente cuando se utiliza antes de la ejecución del código ajeno, en donde solamente la información estática es usada. Sin embargo existen otros casos donde éste enfoque es demasiado dificultoso para implementarse.

Con el enfoque estático resulta demasiado complicado predecir si la dirección destino será la válida en el mejor caso.

El caso no ideal es que el registro no contenga dirección alguna sino cualquier otro valor arbitrario, ya que en repetidas ocasiones, cuando se traduce una secuencia de instrucciones, se encuentra un salto indirecto, en donde la dirección destino se encuentra en un registro.

Otro problema que se presenta es que, dependiendo del compilador y el *linker*, puede darse el caso que se tenga código de relleno antecediendo a cada llamada a procedimientos para poder alinear destinos de salto en palabras, o en lineas de cache por razones de rendimiento.

Finalizando que para arquitecturas como la x86, es que no se sabe en dónde inicia o finaliza una instrucción debido a su tamaño variable de *opcode* (figura 12).

Figura 12. **Ejemplo de tamaño variable de *opcode***

```
31 c0 – 8b -  b5 00 00 03 08 8b bd 00 00 03 00
```

Fuente: elaboración propia.

No se puede determinar si el *8b* representa el inicio de una instrucción *movl* o si es parte de la instrucción anterior, seguida por una instrucción *mov* iniciando con b5.

3.3.2. Traducción y pre decodificado incremental

Una solución al problema de localidad y descubrimiento de código es traducir éste mientras se ejecuta, es decir, hacer una traducción dinámica incremental, que consistirá en pre-decodificar y traducir segmentos de código mientras el programa los vaya alcanzando.

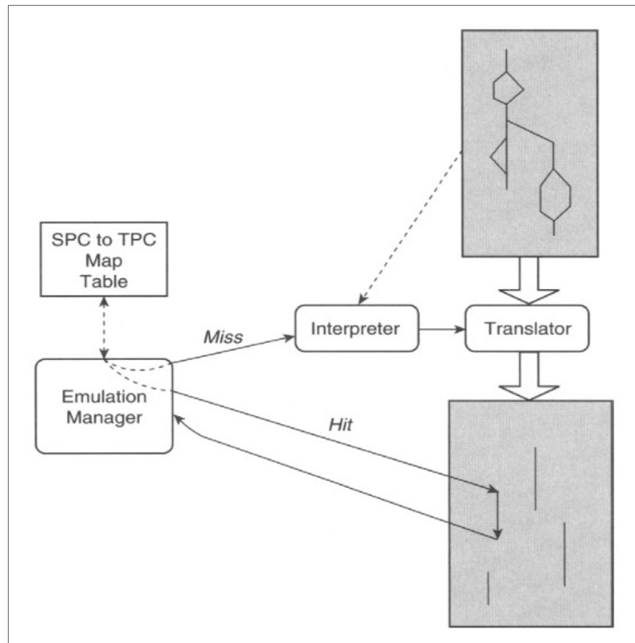
La manera genérica de implementar esto. es contar un traductor *T* que trabaja de la mano con un intérprete *I*, el cual será controlado por un manejador de emulación *EM*, quien se encarga de verificar una tabla de mapeo (la cual puede ser un *Hash*[8]) entre el contador de programa fuente *SPC* y el contador de programa destino *TPC*.

El proceso consiste en que, incrementalmente, el traductor se mantendrá traduciendo código a petición del manejador de emulación en un proceso de prueba - error, es decir, el manejador, al existir un salto indirecto, comprobará que el segmento de código apuntado por el contador de programa destino ya haya sido traducido. Si es valido se dirige al segmento traducido.

De otra manera solicitará al traductor convertir dicho segmento binario a su traducción para el código destino, y al intérprete, la localidad de la nueva dirección, ya que, estando el intérprete en contacto directo con el código binario, logrará precodificar y mapear el contador de programa, de tal manera que al volver a ejecutar el salto, se acceda correctamente al segmento de código al que originalmente se quería acceder con el salto realizado por el código traducido (figura 13).

8 Fuente: *Ngemu Forums*, disponible en: http://forums.ngemu.com/web-development-programming/20491-dynamic-recompilation-introduction-2.html#post240799, fecha de consulta: 12 de diciembre de 2010

Figura 13. **Esquema del funcionamiento de precodificado incremental**



Fuente: SMITH, *Virtual machines: versatile platforms for systems and processes* p 56.

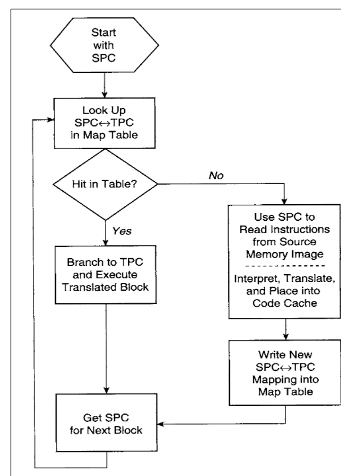
Este código traducido se mantendrá en un caché de código en memoria, y regularmente se traducirá un segmento a la vez en un bloque llamado bloque básico dinámico, el cual es separado al finalizar cada instrucción de salto indirecto, determinado únicamente por el flujo de programa, pues será la unidad básica hacia donde regularmente se resolverá la dirección de los saltos indirectos.

Un problema que puede ocurrir es que al estar dividido entre saltos, es que estos bloques de instrucciones sean destino de otro salto indirecto justo a la mitad del bloque y no al principio.

Esto hará necesario mantener una estructura de datos adicional que contenga rangos adicionales de bloques de traducción y dirijan el contador de programa destino hacia el sitio dentro del bloque al que apunta su código.

La siguiente figura describe mejor el proceso de traducción:

Figura 14: **Diagrama de flujo para el proceso de traducción incremental**



Fuente: SMITH, *Virtual machines: versatile platforms for systems and processes* p 59.

3.4. Emulación de alto nivel y bajo nivel

La emulación de alto nivel, o HLE por sus siglas en ingles, es un enfoque en la construcción de emuladores, principalmente emuladores de sistemas interactivos como consolas de videojuegos, la cual se centra no tanto en emular con exactitud el conjunto de instrucciones y la precisión temporal del sistema, así como su *hardware*, sino en recrear la funcionalidad que provee dicho *hardware*.

Es decir, no se enfocar en simular el método de como se realizó, sino se enfoca en el resultado del método. Por el contrario, el método tradicional, el que ha sido descrito a lo largo del presente trabajo, se le llama Emulación de Bajo Nivel o LLE por sus siglas en inglés.

3.4.1. Criterios para elegir el método de emulación de alto nivel

Para poder implementar un emulador de alto nivel, será necesario que la plataforma a emular tenga un alto nivel de abstracción en vez de código máquina puro. Y que este nivel de abstracción esté implementado directamente en el *hardware* o el sistema operativo de la plataforma de tal manera que, en base a una funcionalidad, devuelva un resultado inmediato sin importar como se procese. Además de tener cierto nivel de estandarización semántica.

Esto es debido a que, las máquinas que son candidatas a ser emuladas de alto nivel, son máquinas lo suficientemente complejas, en las que en su diseño se ha decidido fragmentar o separar la funcionalidad específica en diferentes dispositivos de *hardware* independientes, teniendo un canal de comunicación de alto nivel entre estas y el microprocesador a través de microcódigo, como en algunos emuladores de consolas de videojuegos.

Esto ha logrado reducir la complejidad de implementación de hardware para manejo de gráficas tridimensionales, las cuales requieren de gran poder de procesamiento de aritmética de decimales, y asignarle esta tarea al chip gráfico específicamente, así como la carga de texturas, y de esa manera evitando la sobrecarga del microprocesador.

Dado que la emulación de esas características del chip gráfico en un emulador estándar ya sea intérprete o de traducción binaria, seria lo demasiado complejo como para ser no práctico, tanto desde el punto de vista de procesamiento, como del tiempo requerido para su desarrollo; se recurre a delegación de la tarea del manejo de gráficas directamente orientado a resultados.

Es decir emulación de alto nivel aplicado en la funcionalidad del chip gráfico. Para que así se traduzca solamente las instrucciones de alto nivel que el procesador envía al GPU y la funcionalidad sea ejecutada en la máquina huésped ya traducida, por *APIs* especializados en el manejo de gráficas en tres dimensiones y consumidos por el *hardware* de la máquina huésped.

3.4.2. Ventajas y desventajas

Entre las ventajas se encuentran que al enfocarse únicamente en interpretar la funcionalidad del *hardware*, se cuenta con un método que logra acelerar no solo la ejecución del código, sino simplificar el desarrollo al evitar que se necesite un mayor conocimiento de la funcionalidad interna del *hardware* y la implementación del mismo, al hacer uso de *APIs* ya desarrollados para dichas funcionalidades en la máquina destino.

Al contrario, se tiene que evaluar qué *hardware* y plataforma es apto para desarrollar este tipo de emulación, ya que, requiere un mayor nivel de estandarización en la comunicación del procesador y los dispositivos que se pretende usar la emulación de alto nivel.

Al estar mas en una capa de aplicación, también se corre el riesgo de que se emule solo lo que se ha probado, es decir, que nuevo código que ejecute funcionalidad no conocidas por el *hardware* requerirán modificaciones inesperadas en el motor de emulación de alto nivel.

3.5. Procesamiento en paralelo

Se define como la capacidad de una entidad de ejecutar dos o más tareas atómicas, al mismo tiempo. Tarea atómica se define como una tarea que no puede ser dividida en dos o más subtareas. En el contexto computacional, se puede mapear el concepto de tarea atómica como Hilo, para lo cual el concepto de procesamiento en paralelo se convierte en Computación en paralelo, y se definirá como la ejecución de 2 o más hilos dentro de un mismo marco de tiempo.

Este concepto podría llegar a confundirse con multitarea, sin embargo, se aclara que multitarea es la ejecución de 2 o más procesos al mismo tiempo[9], tomando en cuenta el concepto de proceso, es una unidad de tareas computacionales compuesta de uno o varios hilos. Por lo tanto, multitarea es más un concepto de sistemas operativos que un concepto de procesamiento atómico como será tomado por una máquina virtual.

La finalidad del procesamiento en paralelo es utilizar todas las unidades de procesamiento (CPU) de una computadora, y coordinarlas de tal manera que puedan realizar dos tareas, que puedan realizarse en paralelo, y ejecutarlas al mismo tiempo, para evitar encolarlas y ejecutarlas en serie.

9 Fuente: GUGLIEMETI, M., *Multitareas,* disponible en http://www.mastermagazine.info/termino/6039.php, fecha de consulta: 25 de enero de 2010

Para ello se debe contar tanto con ayuda del *hardware*, es decir, debe contar con 2 o más núcleos o CPUs –e incluso un conjunto de ordenadores independientes interconectados en red-- para un paralelismo puro, así como la capacidad del lenguaje y herramienta de programación para ordenar que se ejecutará en ese modo.

Prueba de la importancia del procesamiento en paralelo es la tendencia actual de crear microprocesadores comerciales, para computadoras personales, de 2 o más núcleos, que en sí son CPUs independientes capaces de realizar 2 o mas tareas simultáneas según la cantidad de núcleos, y según la cantidad de hilos simultáneos que sea capaz de manejar cada núcleo.

La capacidad del procesamiento en paralelo, estará limitada por la Ley de Amdahl, la cual dice que la velocidad máxima S –que será un factor de aceleración del algoritmo si se ejecutara secuencialmente-- que se puede alcanzar por un programa estará limitado solamente por la pequeña porción del programa que no se puede paralelizar –la cual la define como (Q)– , es decir, el complemento de la porción del programa que si es paralelizable (P, en donde P + Q = 1); y dicha velocidad es inversamente proporcional a ésta. para lo cual esta dada la siguiente fórmula (figura 15):

Figura 15. **Ley de Amdahl**

$$\frac{1}{(1-P)+\frac{P}{N}}.$$

Fuente: *Ley de Ahmdal*, disponible en http://www.tomechangosubanana.com/tesis/escrito-1-split/node18.html, fecha de consulta: 5 de noviembre de 2009.

3.5.1. Niveles de paralelismo

Los niveles de paralelismo más comunes son los siguientes:

3.5.1.1. Nivel de bit

El paralelismo a nivel de bit se define como la capacidad de un CPU de manejar 2 o más bits simultáneamente.

Ejemplo de ello es el avance que se ha tenido en el aumento de la longitud de palabra –Cantidad de bits simultáneos— en el bus de datos de cada procesador, ya que, históricamente, los CPUs de 4 bits fueron sustituidos por CPUs de 8, a su vez, estos fueron sustituidos por 16, 32, o 64 bits.

Este aumento en el tamaño de la longitud de palabra del bus de datos corresponde a la necesidad de ejecutar con una sola instrucción, operaciones aritméticas que involucren números reales de mayor exactitud, o de mayor orden.

Esto es para evitar que operaciones adicionales –como suma con acarreo de bits en x86[10]— que se necesiten, sean obviadas al contar con mayor espacio para operarse dentro de la misma palabra sin tener que ejecutar otra instrucción adicional para llevar control de éstas.

---

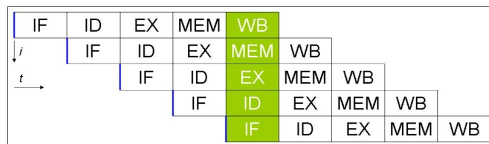
10 Fuente: *Add with carry*, dispoinble en http://siyobik.info/index.php?module=x86&id=4, fecha de consulta: 28 de enero de 2010

3.5.1.2. Nivel de instrucciones

Se define como el reagrupamiento y reordenamiento de una serie de instrucciones que vienen una a una; es decir, en serie; para que éstas puedan ser ejecutadas en paralelo sin romper el flujo normal del algoritmo que represente éste conjunto de instrucciones.

En los CPUs con arquitectura RISC, es común encontrar este paralelismo aprovechando que cada instrucción se separa en 5 fases —Obtener instrucción (*fetch*), decodificar (*decode*), ejecutar (*execute*), acceder a memoria (*memory access*) y reescribir (*write back*)–, de tal manera que se puede llegar a ejecutar hasta 5 instrucciones a la vez en paralelo cuando cada instrucción esta por cada una de las fases.

Figura 16. **Paralelismo de instrucciones en RISC**



Fuente: elaboración propia.

3.5.1.3. Nivel de datos

Conocido también como paralelismo cíclico (*Loop Level Paralellism* en inglés), y relaciona la ejecución de un mismo código a través de varios CPUs, operando datos diferentes distribuidos en cada uno de los CPUs. Ésta se enfoca en la naturaleza paralela y distribuida de los datos.

En el caso de los compiladores que tomen ventaja de características de procesamiento en paralelo a nivel de datos, y optimizar el código para ésta, es necesario, especificarle en el código en donde se desea ejecutar este tipo de paralelismo, ya que no es trival que el compilador prediga dónde y cuando aplicar dicha optimización dada la naturaleza cambiante del un programa secuencial[11].

3.5.1.4. Nivel de tareas

Éste nivel de paralelismo, también conocido como paralelismo funcional y paralelismo de control, consiste en la distribución de tareas, procesos de ejecución, o hilos (*Threads*), entre cada uno de los diferentes CPUs, para, que de una manera semi-independiente, puedan realizarse varias tareas a la vez, quedando la responsabilidad de sincronización al mismo código, por medio de construcciones del lenguaje apropiadas para su manejo.

Un hilo o *Thread* se define como uno o muchos flujos de control dentro de un proceso[12], en donde cada uno de estos tiene una prioridad, y son ejecutados paralelamente al *thread* que lo creó, donde puede compartir entre éstos datos en común. Cabe mencionar que se debe implementar técnicas de sincronización entre los *threads*, para evitar que éstos puedan escribir datos simultáneamente, y enviar el proceso central a condiciones de carrera irrecuperables que incluso pueden alterar el funcionamiento normal de los procesos ajenos a éste.

---

11 Fuente: AYCOCK, C, *What is Loop Level paralelism*?, disponible en http://insidehpc.com/2006/03/11/what-is-loop-level-parallelism/, fecha de consulta: 28 de enero de 2010

12 Fuente: *Java 1.4 Documentation*, disponible en http://java.sun.com/j2se/1.4.2/docs/api/java/lang/Thread.html, fecha de consulta: 28 de enero de 2010

3.5.2. Clases de computadoras en paralelo

El paralelismo computacional en hardware es a nivel de núcleo, a nivel de procesador, a nivel de nodos y a nivel de redes, y se clasifican de la siguiente manera.

3.5.2.1. Multinúcleo

Son aquellas, que, en un procesador, cuentan con varias unidades de ejecución, o núcleos, que pueden ejecutar desde una instrucción por ciclo, al contrario de las *superescalares*, que pueden ejecutar más de una instrucción por ciclo.

3.5.2.2. Simétricas

También llamados *Symmetric Multiprocessor* o SMP por sus siglas en inglés. Y caen dentro de ésta categoría todas aquellas computadoras que cuentan con varios CPUs físicos interconectados de tal manera que todos éstos comparten la memoria RAM y se conectan vía un Bus.

Este bus hace que la escalabilidad en éstos sea nula, ya que un SMP puede contar 32 procesadores máximo debido a la congestión que podía causarse en el bus si el número de procesadores es mayor, reduciendo el rendimiento general del sistema.

3.5.2.3. Cluster

Cluster es la utilización de más de una computadora física, con características en común a nivel de software, conectadas por medio de red, para realizar una tarea en particular común entre éstas, tarea la cual se puede dividir en varios "pedazos" para poderse ejecutar simultáneamente por dichas máquinas con el fin de acelerar su procesamiento.

3.5.2.4. Grid

Se le llama así a una ampliación de *cluster*, específicamente en donde muchas computadoras comparten recursos por medio de Internet para resolver cierta tarea no crítica pero que si requiere de gran poder de procesamiento. Con tarea no critica se puede definir a todas aquellas que no requiera respuesta en tiempo real (dadas las latencias y otras dificultades de conectividad que pueda tener Internet).

3.6. Caso de estudio de emulador basado en recompilación dinámica, QEMU

QEMU es un emulador y virtualizador genérico y de código abierto. Cuando se usa como emulador, QEMU puede ejecutar sistemas operativos y programas hechos para una máquina (por ejemplo un dispositivo ARM) en una máquina diferente (por ejemplo, una computadora personal convencional con ISA x86), logrando un alto rendimiento debido a que implementa recompilación dinámica. Cuando se usa como virtualizador, QEMU logra un rendimiento bastante cercano a la máquina nativa, ejecutando el código de la máquina invitada directamente en el CPU del huésped.

Para ello se necesita un controlador en el anfitrión llamado “Acelerador QEMU” o también KQEMU, el cual aprovecha las extensiones del sistema operativo para virtualizar.

En este modo se requiere que tanto el huésped como el anfitrión sean máquinas con un procesador con ISA compatible con x86.

QEMU tiene dos modos de operación:

- Emulación en modo sistema: en este modo, QEMU emula un sistema completo, incluyendo el procesador y periféricos y puede usarse para cargar un sistema operativo diferente sin necesidad de reiniciar el ordenador, o también para depuración de código del sistema.

- Emulación en modo usuario: en este modo QEMU puede lanzar procesos de Linux compilados para un CPU en otro CPU. Permitiendo por ejemplo, usar WINE en arquitecturas no x86 o para compilación y depuración cruzada.

Entre las arquitecturas destino soportadas por el emulador de QEMU se encuentran (tabla III):

Tabla III. **Arquitecturas destino soportadas por el emulador QEMU**

CPU Destino	Emulación del modo Usuario	Emulación en modo Sistema
Alpha	En desarrollo	En desarrollo
ARM	Si	Si
CRIS	Si	En Pruebas
m68k (Coldfire)	Si	Si
MicroBlaze	En Pruebas	En Pruebas
MIPS	Si	Si
MIPS64	No esta soportado	Si
PowerPC	En Pruebas	Si
PowerPC64	En Pruebas	En desarrollo
SH-4	En Pruebas	En desarrollo
SPARC	Si	Si
SPARC64	En desarrollo	En desarrollo
x86	Si	Si
x86_64	En Pruebas	Si

Fuente: elaboración propia.

Entre las arquitecturas del huésped (máquina que ejecuta QEMU) que soporta este emulador están (tabla IV):

Tabla IV: **Arquitecturas en donde se puede ejecutar QEMU para emulación**

CPU del huésped	Estado
Alpha	No soportado
ARM	En prueba
CRIS	No soportado
HPPA	desarrollo
IA64	No soportado
m68k	No soportado
MIPS	En prueba
MIPS64	No soportado
PowerPC	Completo
PowerPC64	En prueba
SH-4	No soportado
SPARC	En prueba
SPARC64	No soportado
x86	Completo
x86_64	Completo

Fuente: elaboración propia.

3.6.1. Recompilación dinámica en QEMU

QEMU tiene un traductor dinámico, el cual, como los traductores binarios, cuando encuentra un segmento de código, lo traduce al ISA del huésped.

QEMU utiliza técnicas especiales de recompilación dinámica –tomando en cuenta el costo temporal que éste proceso puede tener al momento de traducir--, hacen que éste sea relativamente fácil de portar a otras arquitecturas mientras mantiene un rendimiento aceptable durante el proceso de portación.

La idea general que utiliza QEMU es partir cada instrucción x86 en pocas instrucciones más simples, cada instrucción simple está implementada por un segmento de código en C (como se puede ver en el archivo `*target-i386*/*op.c*' del código fuente del emulador).

Luego, una herramienta de compilación (`*dyngen*') toma el archivo objeto correspondiente(`*op.o*') para recrear un generador de código dinámico que concatena las instrucciones simples hasta construir una función.

La clave para obtener rendimiento óptimo es que los parámetros constantes pueden ser pasados a las operaciones simples. Para ello, re localizaciones tontas de ELF se generan con GCC por cada parámetro constante, y para ello, la herramienta (`*dyngen*') puede localizar dichas re-localizaciones y generar el código apropiado en C para resolverlos mientras se construye el código dinámico.

4. MÁQUINAS VIRTUALES

Las máquinas virtuales tanto de lenguajes como de sistemas ejemplifican técnicas de recompilación dinámica, como las que se describen a continuación.

4.1. *Java Virtual Machine* (JVM)

La JVM es una implementación en software de una arquitectura de computadora orientada a la ejecución de código intermedio, el cual es un lenguaje estándar para código que, originalmente puede haber sido código Java u otro lenguaje de alto nivel soportado.

Por esta característica, el código que se escribe para ésta, debe ser *portable* entre diferentes arquitecturas de *hardware*, dependiendo únicamente de la implementación de la JVM para cada arquitectura de *hardware* específica, debido a que las JVM ejecutan el mismo conjunto de instrucciones y se apoyan en librerías escritas con APIs estándar para que sea igual la ejecución tanto en un sistema embebido con un procesador modesto, como para sistemas grandes que aprovechen cualquier tipo de paralelismo.

Entre sus especificaciones incluye estricta portabilidad del código intermedio, sin embargo no especifica claramente la manera de hacer el código *portable* entre las diferentes arquitecturas, en donde por ejemplo, en la especificación del lenguaje Java no se define las prioridades que debe tener dicha JVM, y tampoco obliga a que todas las JVM tengan recolector de basura.

4.1.1. Componentes de la JVM

Regularmente la JVM está compilado como un programa monolítico, pero está diseñado como un conjunto de componentes, entre los que se puede mencionar

4.1.1.1. El cargador de clases

Su tarea es cargar las clases y organizarlas para que estén integradas con el resto del Entorno Java.

4.1.1.2. El administrador de seguridad

Previene que código no confiable sea ejecutado en la JVM, y se implementa regularmente por un *sandbox*, el cual coloca una barrera de seguridad alrededor de la aplicación a usar, y se usa mayormente para *applets*.

4.1.1.3. Recolector de basura

La especificación de Java no obliga la implementación de un recolector de basura, pero es necesario contar con uno, ya que por sí sola la JVM no puede detectar memora que no está en uso, por lo que se requiere un barrido regular para liberar espacio y defragmentar el *heap* de memoria, y de esa manera haya más memoria disponible para cada objeto y mejorar el rendimiento.

4.1.1.4. Manejador de hilos

Por su diseño, Java es un lenguaje que soporta múltiples hilos o *multithreading*, para encapsular la concurrencia, y aprovechar los recursos de las máquinas físicas con capacidad de procesamiento en paralelo.

Cada hilo es un contexto de ejecución, es decir, cada uno cuenta con su propia área de *stack* del procesador, su conjunto de registros de CPU, y su stack Java.

Las APIs de java proveen la funcionalidad necesaria para iniciar, detener, cambiar prioridad e interrumpir los hilos, manejados por un despachador (*dispatcher*) de hilos --quien se encarga de intercambiar el control de un hilo a otro-- y un planificador (*scheduler*) el cual les asigna prioridad a cada uno de éstos.

El mecanismo físico de *threads* en el que se basa la JVM está dado por el sistema operativo –*Kernel Threads*--, o en caso que no esté soportado, por librerías que estén enlazadas a la JVM que provean dicha funcionalidad.

4.1.1.5. Entrada / salida (I/O)

Se encarga principalmente de acceder a las interfaces del entrada/salida del sistema operativo, y proveerla a la JVM.

El subsistema gráfico se encarga del muestreo de gráficas por medio del *Abstract Window Toolkit* o AWT por sus siglas en inglés. Y es un subsistema basado en eventos.

4.1.1.6. Intérprete de *bytecode*

La máquina virtual de Java, análogo a las máquinas físicas con microprocesador, interpreta un conjunto de instrucciones de operación llamadas *Bytecode*, éstas son operaciones que se encuentran en un *byte*, y por lo tanto permite hasta 256 códigos de operación diferentes.

Entre los opcodes están principalmente los que aplican las operaciones de carga desde memoria, almacenamiento en memoria y operaciones lógicas y aritméticas básicas para cada tipo de dato.

Tabla V: **Tabla de opcodes del *bytecode* para la máquina virtual de Java.**

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	nop	aconst_null	iconst_m1	iconst_0	iconst_1	iconst_2	iconst_3	iconst_4	iconst_5	lconst_0	lconst_1	fconst_0	fconst_1	fconst_2	dconst_0	dconst_1
1	bipush	sipush	ldc	ldc_w	ldc2_w	iload	lload	fload	dload	aload	iload_0	iload_1	iload_2	iload_3	lload_0	lload_1
2	lload_2	lload_3	fload_0	fload_1	fload_2	fload_3	dload_0	dload_1	dload_2	dload_3	aload_0	aload_1	aload_2	aload_3	iaload	laload
3	faload	daload	aaload	baload	caload	saload	istore	lstore	fstore	dstore	astore	istore_0	istore_1	istore_2	istore_3	lstore_0
4	lstore_1	lstore_2	lstore_3	fstore_0	fstore_1	fstore_2	fstore_3	dstore_0	dstore_1	dstore_2	dstore_3	astore_0	astore_1	astore_2	astore_3	iastore
5	lastore	fastore	dastore	aastore	bastore	castore	sastore	pop	pop2	dup	dup_x1	dup_x2	dup2	dup2_x1	dup2_x2	swap
6	iadd	ladd	fadd	dadd	isub	lsub	fsub	dsub	imul	lmul	fmul	dmul	idiv	ldiv	fdiv	ddiv
7	irem	lrem	frem	drem	ineg	lneg	fneg	dneg	ishl	lshl	ishr	lshr	iushr	lushr	iand	land
8	ior	lor	ixor	lxor	iinc	i2l	i2f	i2d	l2i	l2f	l2d	f2i	f2l	f2d	d2i	d2l
9	d2f	i2b	i2c	i2s	lcmp	fcmpl	fcmpg	dcmpl	dcmpg	ifeq	ifne	iflt	ifge	ifgt	ifle	if_icmpeq
A	if_icmpne	if_icmplt	if_icmpge	if_icmpgt	if_icmple	if_acmpeq	if_acmpne	goto	jsr	ret	tableswitch	lookupswitch	ireturn	lreturn	freturn	dreturn
B	areturn	return	getstatic	putstatic	getfield	putfield	invokevirtual	invokespecial	invokestatic	invokeinterface		new	newarray	anewarray	arraylength	athrow
C	checkcast	instanceof	monitorenter	monitorexit	wide	multianewarray	ifnull	ifnonnull	goto_w	jsr_w	breakpoint					
D																
E																
F															impdep1	impdep2

Fuente: *Java 7 VM Specification*, disponible en: http://docs.oracle.com/javase/specs/jvms/se7/html/jvms-7.html, fecha de consulta: 5 de diciembre de 2009.

4.1.2. Implementación del intérprete de Java

La máquina virtual de java se implementa de varias maneras, entre las cuales están:

4.1.2.1. Intérprete estándar

Es la forma más natural del intérprete sin ningún ajuste de velocidad, su velocidad de ejecución puede llegar a ser hasta 20 veces más lenta que si el mismo algoritmo a ejecutar fuera escrito en C++. Regularmente sirve de referencia para JVM más optimizadas, y, por su amplia portabilidad por estar escrito en el lenguaje C, se utiliza también para portar JVMs entre diferentes arquitecturas de una manera rápida.

El intérprete *Just-in-Time* (JIT, por sus siglas en ingles) a diferencia del intérprete estándar, es un compilador en tiempo de ejecución, con capacidad de traducción binaria y recompilación dinámica, el cual se encarga de traducir el *bytecode* de la máquina virtual de Java, a código máquina de la arquitectura huésped, es decir, para una JVM ejecutándose en una arquitectura x86.

El JIT se encargará de traducir, por cada llamada a un método, el código de dicho método del *bytecode* a códigos de operación legibles para la arquitectura x86, compilando el código justo a tiempo.

En donde luego de ser compilado, la misma JVM solo llama al código como si fuera un procedimiento nativo, optimizando todo el proceso de ejecución.

El JIT de la JVM, como recompilador dinámico, puede incurrir en un leve retraso en su inicialización debido a que muchas veces es gran cantidad de código que se debe pre compilar previo a la ejecución de todo el programa.

Los pasos de compilación generales son los siguientes:

- Dividir el *bytecode* en bloques básicos para construir un grafo de flujo de control.
- Construir un grafo acíclico dirigido (DAG por sus siglas en inglés)por cada bloque básico.
- Generar código desde el DAG construido.

4.1.2.1.1. Identificación de los bloques

Los bloques básicos son aquella secuencia de sentencias en el que el flujo de control entra al inicio, y sale al final, sin ningún tipo de interrupción o detención.

Se puede identificar primero, determinando un conjunto de líderes, es decir, una instrucción que le sigue a una asignación de memoria o invocación de método, o instrucción rama que, eventualmente será también un líder. Así también la primer sentencia de un método es un líder. Luego el bloque básico consistirá en todas las sentencias que sigan a éste líder, hasta el próximo líder, sin incluirlo.

4.1.2.1.2. Construcción de un DAG

DAG o gráfico acíclico dirigido se le llama a una representación intermedia de un bloque básico y permite visualizar cómo un valor calculado es usado por sentencias subsecuentes de cualquier bloque básico, en donde cada nodo interior del DAG representa el resultado de ese cálculo y los nodos hijos son los argumentos para dicho cálculo. Y donde cada hoja es, ya sea, una variable, o una literal constante, mientras que los nodos internos, operaciones.

Éste se construye por medio de una exploración a profundidad del árbol de bloques básicos. Iniciando desde el primer bloque básico más a profundidad, para consecuentemente, seguir construyendo los nodos predecesores debido a que sigue un orden de Pila debido a que la misma JVM es una máquina que se basa en una Pila (*Stack*).

4.1.2.1.3. Generación de código desde el DAG

El primer paso de la generación de código es ordenar el listado de DAG en cada bloque básico, en donde se deberá aplicar para cada bloque básico la revisión de registros disponibles, y liberarlos de ser necesarios con el algoritmo *freereg*.

Considerando la naturaleza de la técnica JIT, en donde ningún algoritmo de alojamiento de registros se usa hasta que un registro requiere el uso de dicho registro, y, de esa manera, un registro predeterminado es liberado y usado.

4.1.2.2. Implementación de la JVM en ISA x86

Se enumera a continuación los detalles concernientes a la implementación sobre la arquitectura Intel x86.

4.1.2.2.1. Aritmética de enteros

Regularmente para tipos enteros y *long* es necesario representarlos como palabras de 64 bits, tal como lo especifica la JVM, en donde las *subrutinas* de suma, resta, multiplicación y división es implementada en *subrutinas* de software; y otras operaciones aritméticas y lógicas se implementan con operaciones *multi byte* usando pares de registros.

4.1.2.2.2. Aritmética de tipos dobles

Se implementa la operación de tipos dobles con rutinas de software, que hacen uso de la pila para ingresar los operandos, y obteniendo el resultado desde el registro de punto flotante (*Floating Point Register*).

4.1.2.2.3. Invocación de métodos

Los métodos estáticos y virtuales se invocan ingresando los argumentos a la pila, y resolviéndolo usando una rutina de alto nivel, que devolverá ya sea la posición en la pila de la rutina (para métodos virtuales) o una referencia de método para ser llamado por la instrucción CALL (para métodos estáticos) y depositarla en el registro EAX del procesador; cargar el código de dicho método al que apunta EAX, usar la llamada CALL , y decrementar el puntero de pila.

Para el valor de retorno en métodos, se suele utilizar regularmente *eax*, *ebx* y *ecx* para especificar el valor de retorno ya sea de tipos de dato primitivos, o de referencias a objetos. Por métodos virtuales se entiende a todos aquellos métodos dinámicos que son llamados al instanciar a un objeto.

4.1.2.2.4. Creación de objetos

Los objetos son creados con la palabra reservada *new* en java, en donde, en tiempo de ejecución, se debe de resolver la referencia para cargar e instanciar la clase del objeto que va a ser creado, para ello se debe utilizar parte del módulo de cargador de Clases de la JVM. De ello parte que también se requiera una *subrutina* por software para la palabra reservada *new*, la cual se encargará de ejecutar todas las tareas necesarias y devolver la referencia al objeto en el registro EAX.

4.1.2.2.5. Manipulación de campos de los objetos

Las rutinas para la manipulación de campos son similares a las utilizadas para la invocación de métodos, y consiste en retornar la dirección del campo en el registro EAX antes de llamar a la rutina de alto nivel para

4.1.2.2.6. Manipulación de variables locales

Se utiliza regularmente la referencia a la variable dentro del segmento de *stack* en el que está dicha variable, y se opera sobre éstas.

4.1.2.2.7. Excepciones

La instrucción *throw* buscará dentro de la pila del hilo en ejecución algún manejador de excepciones, es decir, aquellas conformadas por la construcción *try*... *catch*... En caso se encuentre, se buscara el siguiente en la anterior pila. Si se encuentra, se invocará el manejador, de otra manera, se detendrá la ejecución del hilo.

Las excepciones aritméticas y de puntero nulo (*ArithmeticException* y *NullPointerException*) son manejadas a nivel de manejador de señales, el cual, así como en el anterior caso, buscará la actual pila un manejador de señales, y si no encuentra, terminará la ejecución de dicho hilo.[13]

4.2. Microsoft CLR

La máquina virtual de Microsoft para su plataforma .NET, la CLR o *Common Language Runtime* por sus siglas en inglés, interpreta *bytecode* orientado a objetos y se basa en pilas, así como la JVM de java, con la diferencia de que, de forma nativa, soporta la generación de bytecode en tiempo de ejecución a través del uso de las clases en el *namespace System.Reflect.Emit*.

Siendo este bytecode generado, equivalentemente veloz al bytecode compilado en C#, y comparable en velocidad al cliente de *Sun Microsystems Hotspot Java* para Java.

13 Fuente: MANDAL, *Design and Implementation of Java Virtual Machine*, 2005

Dicha máquina virtual es la implementación de CLI, Un estándar internacional para la ejecución de entornos de desarrollo en el que los lenguajes y las librerías trabajan en conjunto sin problemas entre ellos. Éste especifica un entorno de ejecución virtual que protege los componentes de CLI de los del sistema operativo subyacente. Su arquitectura se divide en:

4.2.1. ***Common Type System* (CTS)**

Define todo el conjunto de tipos de variable en un programa que cumpla con el CLI.

4.2.2. ***Common Language Specification* (CLS)**

Define el subconjunto de tipos CTS que pueden ser utilizados para llamadas externas.

4.2.3. ***Common Intermediate Language* (CIL)**

Lenguaje intermedio al cual es compilado el código fuente, común entre todos y legible por el VES.

4.2.4. ***Virtual Execution System* (VES)**

La máquina virtual que ejecuta el código intermedio y yace entre el sistema operativo y el código del programa.

4.3. Esquemas de virtualización

La virtualización y paravirtualización conciernen a máquinas virtuales de arquitecturas de procesadores reales, las que permiten emular ya sea entornos operativos enteros (virtualización de un ISA completo), incluyendo *hardware* y sistema operativo, o solo entornos parciales, como en el caso de la paravirtualización, que aprovecha las características de varios sistemas operativos para crear entornos virtuales del mismo, usando el mismo *hardware*.

4.3.1. Virtualización pura

Técnica de virtualización que implementa traducción binaria para la ejecución del código de la máquina invitada. En la cual inspecciona cada bloque básico y las reescribe con instrucciones privilegiadas – aquellas instrucciones del modo protegido en el caso del x86, que solo pueden ser ejecutadas por el kernel--, Usando la virtualización pura, el VMM intercepta las instrucciones del CPU que no son capaces de virtualizarse y las reescribe de tal manera que estas funcionen virtualizadas.

En los procesadores x86, existen diferentes niveles o "*rings*" en los que el software puede ser ejecutado: el *Ring* 0 es usado por el kernel del sistema operativo huésped , mientras que el *Ring* 3 es usado comúnmente por software en modo de usuario.

El *Ring* 0, por ser el que tiene acceso menos restringido al CPU es aprovechado por el VMM al integrarse al Kernel por medio de algún controlador que tenga que instalarse en éste.

Al contrario del sistema operativo huésped, el sistema operativo invitado no se ejecuta en *Ring* 0, y aunque éste esta programado para eso, se le simula interceptando las instrucciones especiales de este *Ring* 0 y se reescriben usando el controlador en el huésped.

La ventaja de la virtualización pura es que casi todos los sistemas operativos pueden ejecutarse como invitados sin necesidad de que sean modificados, debido a que el VMM reescribe las instrucciones, además que no requiere ningún *hardware* que soporte virtualización.

La desventaja es que es la opción mas lenta en tiempo de ejecución entre las otras 2 alternativas de virtualización debido a la traducción que se debe realizar.[14]

Entre los productos comerciales que implementan este esquema de virtualización se puede mencionar a Vmware, VirtualPC, y las opciones de código abierto: Qemu y Virtualbox.

Regularmente este esquema es el que más recursos demanda, donde puede acelerarse debido a extensiones de *hardware* especializadas para ello, en el caso de los procesadores basados en la arquitectura x86, se utilizaría las extensiones VT-x en el caso de *CPUs* intel, o SVM en el caso de los procesadores AMD.

El software de virtualización disponible para soportar esta aceleración por *hardware* se puede nombrar a XEN, KVM, Vmware y Virtualbox.

---

14 Fuente: GARRAUX, Oliver *How Virtualization Works and its Effects on IT*

4.3.2. Virtualización en XEN

XEN es una monitor de máquina virtual (VMM por sus siglas en inglés) que permite consolidación de servidores de alto rendimiento, facilidades de hospedaje localizados en un mismo punto, servicios web distribuidos, plataformas computacionales seguras y movilidad de aplicaciones, permitiendo a los usuarios instanciar dinámicamente un sistema operativo para ejecutar lo que se desee[15]

Para lograr virtualizar, XEN se basa en la modificación del código fuente del kernel host los dispositivos como PCI, VGA, IDE, entre otros, XEN los emula utilizando el módulo *ioemu*. Los controladores de dispositivos de alto rendimiento están implementados directamente en XEN.

La virtualización en XEN implica 2 operaciones principales, *VM Entry* y VM *exit. VM Entry* es la transición desde el VMM a la máquina virtual invitada. Iniciando en modo ROOT el VMX (sin privilegios y adecuada para la ejecución de máquinas virtuales para no interferir el espacio de memoria del kernel) En la cual entra y carga el estado de la máquina invitado, así como el criterio de salida del VMCS (*Virtual Machine Control Structure*).

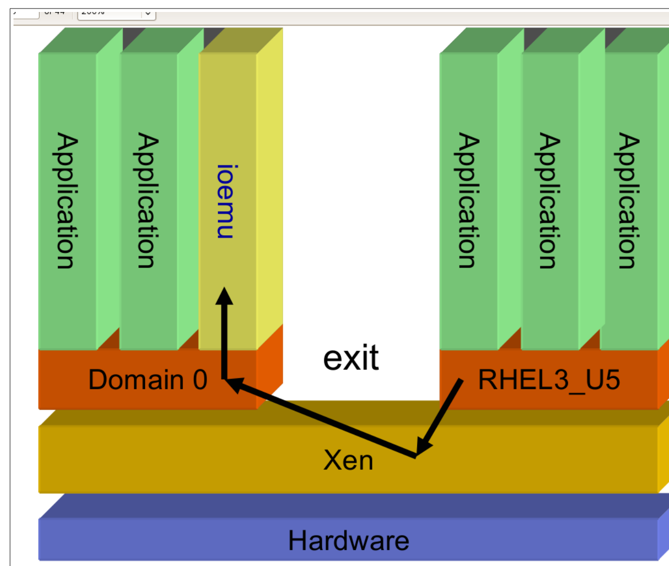
VM Exit es la transición desde la máquina virtual invitada hacia el VMM, realizando una operación ROOT VMX (con todos los privilegios para modificar el espacio de memoria del *kernel),* guarda el estado de las máquinas invitadas en el VMCS y carga el estado de la VMM desde el VMCS.[16]

---

15 Fuente: BARHAM, *Xen and the Art of Virtualization*
16 Fuente: NARENDAR B. *Understanding Intelo Virtualization Technology (VT)*

El costo temporal asociado a las operaciones de *VM Entry* son bastante altas, pues implican según la especificación de la VTX hasta 11 páginas de condiciones que necesitan ser evaluadas antes de realizar dicha operación.

Figura 17. **Esquema general de las operaciones VM *Entry* y VM *Exit***

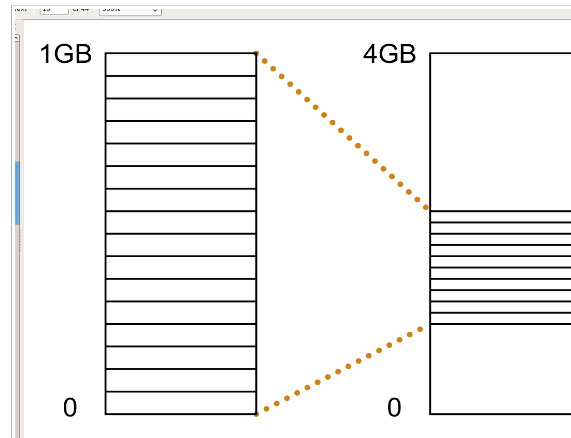


Fuente: elaboración propia.

4.3.2.1. Mapa de memoria virtual tradicional

El mapeo de memoria entre modelo plano del sistema operativo (figura 18) y la memoria de la máquina virtual es normalmente mantenida por el mismo sistema operativo, en donde se permite al CPU caminar por las tablas de paginación automáticamente tal como el sistema operativo lo especifique, ocurriendo fallos de página solo en el caso que no se encuentre la página solicitada, o exista alguna violación de acceso.

Figura 18. **Modelo plano de memoria**



Fuente: elaboración propia.

4.3.3. Virtualización por *hardware*

La virtualización por *hardware* permite a los sistemas operativos huésped ejecutarse en el *Ring* 0 por medio de extensiones de virtualización agregadas en procesadores x86.

El soporte de extensiones de virtualización por *hardware* se incrementó durante la primer década de los 2000, convirtiéndose casi en una capacidad ubicua de los procesadores X86 tradicionales, en las que casi cualquier CPU de esta arquitectura contiene algún tipo de soporte de virtualización, entre las que se puede mencionar las tecnologías: Intel VT-x y AMD SVM, y las extensiones de *Cell* de *PowerPC.* En la virtualización asistida por hardware, se divide el *ring* 0 en dos partes: Una en donde se ejecuta el VMM, y la segunda en la que el sistema operativo virtualizado se ejecuta.

Eso da la ventaja de virtualizar por hardware tanto la virtualización pura como la paravirtualización. Cualquier sistema operativo invitado puede virtualizarse por *hardware* ya que, como en el caso de la virtualización pura, éste no necesita modificarse.

La tecnología Intel VT, (*Intel Vanderpool Technology*) además agrega nuevas instrucciones al CPU como VMXON, VMXOFF, VMLAUNCH, VMRESUME, VMCALL, entre otras, las cuales son instrucciones especializadas para realizar tareas costosas como el VMENTRY (realizada por un VMLAUNCH) y VMLEAVE (realizada por un VMRESUME) que se mencionaron previamente. Además cada máquina invitada tiene un VMCS para su estado.

Parecido a la tecnología de Intel, la tecnología de SVM (*Secure Virtual Machine*) de AMD asigna las instrucciones VMRUN, VMCALL, entre otras para realizar las mismas tareas.

Otra característica en los procesadores de dicha arquitectura ha sido la ampliación del bus de datos, de 32 a 64 bits, así como las características de seguridad que implementan en la placa. Entre las que se puede mencionar la capacidad de privilegio dinámico en modo *ring0* como las presentes en los procesadores AMD SVM e Intel LT.

Una ventaja ha sido la integración de varios núcleos “cores” o procesadores independientes dentro de una misma placa, haciendo posible asignar un solo núcleo para las tareas de virtualización y otro para el uso general del sistema huésped, aunque esto puede variar según el sistema operativo y las prestaciones del CPU.

Para la comunicación con *hardware* se aprovechan las capacidades de entrada y salida por medio del la unidad de administración de memoria (I/O MMU), teniendo, a la fecha, tecnologías IOMMU de Intel y AMD, y la capacidad futura de la tecnología *PCI Express* que permitirá soporte para virtualización.

4.3.4. Paravirtualización

La paravirtualización que fue ideada después que la virtualización pura, consiste en presentar un entorno virtual con similares prestaciones de *hardware*, pero puede que no sean iguales. Es decir, una capa de comunicación directa entre el *hardware* físico y el VMM que permita acceso directo a los dispositivos, sin necesidad de emularlos como ocurre con la virtualización pura.

Este modo de virtualización implica, que para ganar un tiempo de ejecución bastante cercano a la máquina huésped, el VMM tendrá que ejecutar sistemas operativos invitados modificados levemente, para que éstos tengan acceso directo a los recursos de *hardware* de la máquina huésped, en donde el VMM será quien asigne los recursos en vez del sistema operativo huésped. Sin embargo, no requerirá modificación del ABI (Interfaz de aplicación binaria por su siglas en inglés), por lo que el código (ya sea de aplicaciones o librerías) que ejecute los sistemas operativos huésped pueden ir sin modificación alguna.

Aunque, por la naturaleza de la paravirtualización y su eficiencia se podría obviar la necesidad de la virtualización asistida por *hardware,* su uso, en conjunto con la virtualización asistida por *hardware* descrita anteriormente, permite la aceleración de ejecución de entornos virtualizados, a una velocidad casi transparente como si se ejecutara de forma nativa en la máquina huésped.

5. ANÁLISIS Y EVALUACIÓN DE LAS TÉCNICAS DE RECOMPILACIÓN DINÁMICA, VIRTUALIZACIÓN Y PARAVIRTUALIZACIÓN

Para encontrar la técnica más eficiente de emulación y virtualización, se procedió a asumir una hipótesis, evaluar las técnicas más comunes respecto a la hipótesis planteada, observar sus tiempos de ejecución, analizar su tendencia, y concluir, tal como se muestra a continuación.

5.1. Selección de técnicas de recompilación y definición de la hipótesis

Se describe a continuación los criterios para la selección de las técnicas, y definición de la hipótesis.

5.1.1. Definición de la hipótesis

El tiempo de ejecución de un algoritmo de alta complejidad en una máquina virtual paravirtualizada será menor a dos veces el tiempo de ejecución medido sobre la máquina nativa, mientras que el tiempo de ejecución en la máquina virtual emulada será mayor y variará según la arquitectura.

5.1.2. Criterios del algoritmo a evaluar

El algoritmo a elegir cumplirá con los siguientes requisitos para que la evaluación sea válida:

- Alta complejidad temporal:

    - Ya sea utilizando recursividad o iteraciones, el algoritmo a elegir deberá ser lo suficientemente pesado tanto para la máquina invitado como para la anfitrión, para que nos pueda lanzar estadísticas temporales medibles en la escala de milisegundos a segundos.
    - Uso pesado de operaciones aritméticas:
    - Para poder evaluar la eficiencia de ejecución en operaciones matemáticas en el CPU huésped y validar la arquitectura menos complicada para esa tarea.

- No concurrente:

    - Se debe evitar a toda costa que dicho algoritmo se ejecute en paralelo en arquitecturas que presentan capacidad SMP, ya que la ejecución en múltiples CPU está fuera del alcance de la evaluación a realizar.

5.1.3. Elección del algoritmo

El algoritmo que mejor se ajusta a los criterios especificados es el algoritmo de factorización de números primos, por su su sencillez en codificación y complejidad en ejecución, pues comprende de la ejecución de operaciones aritméticas pesadas como división y modulo, ciclos anidados y procesamiento no concurrente.

5.1.3.1. Algoritmo de factorización de números primos

La representación para el algoritmo de números primos en pseudocódigo es de la siguiente manera (figura 19):

Figura 19. **Pseudocódigo del algoritmo de factorización de números primos**

```
Para todo x de 1 a 10000 hacer:
      lista_factores = (vacío)
      ciclos : = 2
      mientras  ciclos < x:
            si el residuo de ( modulos / ciclos) = 0 entonces:
                  x := x / ciclos
                  agregar_a_lista(ciclo)
            de otra manera:
                  ciclos := ciclos + 1
      imprimir(lista)
```

Fuente: elaboración propia.

5.1.3.2. Implementación en el lenguaje *Python* y complejidad

El análisis de complejidad para el algoritmo en *Python* se describe en la figura 20:

Figura 20. **Análisis de complejidad del algoritmo seleccionado**

<table>
<tr><td>

```
def primefactors(x):
       factorlist=[]
       loop=2
       while loop <= x:
              if x%loop == 0:
                     x /= loop
       factorlist.append(loop)
              else:
                     loop += 1
       return factorlist
for i in range (10000):
       print primefactors(i)
```

</td><td>

```
(3) n + 1
(4) n + 1
(5) n + 1
(6) (n + 1) * n
.
.
.

.

(1) n
(2) n  (ir a 3)
```

</td></tr>
<tr><td colspan="2">

Complejidad temporal asintótica calculada:

O((n + 1)*n) = O(n ^ 2 + n) ~ **O(n^2)**

</td></tr>
</table>

Fuente: elaboración propia.

5.1.4. Definición del área análisis, alcance y definición de la muestra

Para la realización del análisis se necesitó la generación de tiempos de procesador, respecto a la ejecución de un algoritmo de factorización de números primos realizado en el lenguaje de programación Python versión 2.6, obteniendo la muestra por medio de la utilidad GNU time para el sistema operativo Debian GNU/Linux sobre la máquina virtual invitado, ejecutándose sobre QEMU sobre el sistema Arch Linux.

Dichas evaluaciones se ejecutaron en tres escenarios diferentes:

- Emulación por recompilación dinámica para el ISA ARM
- Emulación por recompilación dinámica para el ISA PowerPC

- Emulación por recompilación dinámica para el ISA x86-64
- Emulación por paravirtualización para el ISA x86-64

Las mismas evaluaciones se harán sobre un *invitado* sin capacidad de concurrencia y paralelismo, Sobre la que se compararán contra la ejecución nativa sobre *Arch Linux* en el ordenador huésped que cuenta con un ISA x86-64 también en modo no SMP.

5.1.5. Técnicas de recompilación y emulación a evaluar

Se procedió a realizar una evaluación de rendimiento entre distintas arquitecturas usando recompilación dinámica y paravirtualización, comparándola con el rendimiento de la máquina invitado, definiendo la configuración estándar para cada máquina virtual como se describe a continuación:

5.1.5.1. Especificación de la máquina anfitrión

Las especificaciones de *hardware y software* son las siguientes.

- CPU: AMD FX(tm)-6200 *Six-Core processor AuthenticAMD*
- Memoria RAM: 8192 MiB
- Espacio de Intercambio**:** 2048 MiB
- Sistema operativo**:** Arch Linux; Linux kernel v. 3.3.2-1-ARCH SMP
- Versión de glibc**:** 2.15-10
- Versión de Qemu**:** 1.0.1-1

5.1.5.2. Especificación de la máquina virtual *PowerPC*

Las especificaciones de *hardware y software* son las siguientes.

- CPU: PowerPC 740/750 PowerMac G3
- RAM: 128 MiB
- Espacio de Intercambio : 756 MiB
- Sistema Operativo: Debian Lenny PPC 64; Linux kernel v. 2.6.26-2-*powerpc*
- Versión de glibc: 2.7.18lenny7

5.1.5.3. Especificación de la máquina virtual ARM

Las especificaciones de *hardware y software* son las siguientes.

- CPU: ARM926EJ-S rev 5 v51
- RAM: 128 MiB
- Espacio de Intercambio : 756 MiB
- Sistema Operativo: Debian Lenny ARM; Linux kernel v. 2.6.26-2-*versatile*
- Versión de glibc: 2.7.18lenny7

5.1.5.4. Especificación de la máquina virtual x86-64

Las especificaciones de *hardware y software* son las siguientes.

- CPU: QEMU *Virtual CPU* version 1.0.1 64-bit @ 3812.7 MHz
- RAM: 128 MiB
- Espacio de Intercambio: 756 MiB

- Sistema Operativo: Debian Lenny amd64; Linux kernel v. 2.6.26-2-amd64
- Versión de glibc: 2.7.18lenny7

5.1.5.5. Especificación de la máquina paravirtualizada x86-64

- CPU: *QEMU Virtual CPU* version 1.0.1 64-bit @ 3812.7 MHz
- RAM: 128 MiB
- Espacio de Intercambio: 756 MiB
- Sistema Operativo: Debian Lenny amd64; Linux kernel v. 2.6.26-2-amd64
- Versión de glibc: 2.7.18lenny7
- Tecnología de paravirtualización: KVM

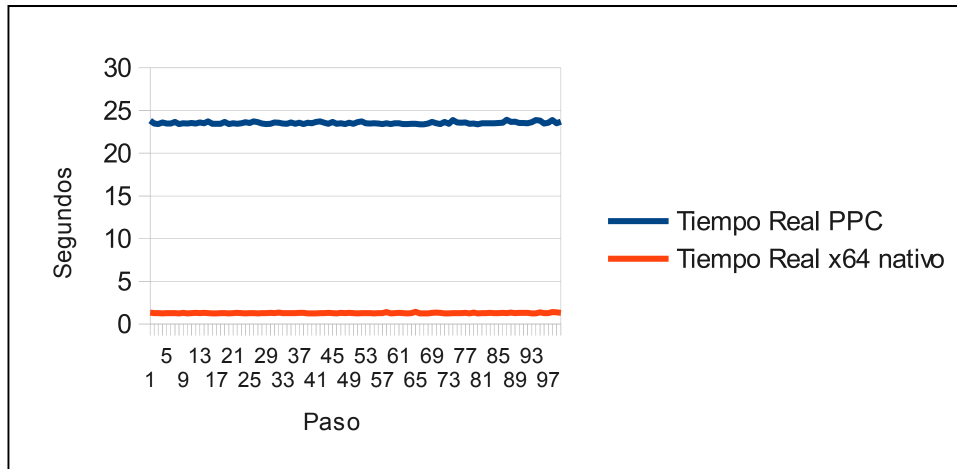
5.2. Análisis de datos y evaluación comparativa

El resumen de los tiempos medidos es de la siguiente manera:

5.2.1. PowerPC emulado contra x64 nativo

Al ejecutar el programa en la máquina virtual PPC bajo los parámetros de configuración nativa, se observa el tiempo real de ejecución del mismo de 2354.22 segundos en 100 pasos, con un promedio de 23.5411 (figura 21) segundos por paso y una desviación estándar de 0.1198 segundos, comparado con la ejecución nativa en el *invitado* con arquitectura x64 que presenta una ejecución de 128.75 segundos en 100 pasos, con un promedio de 1.2875 segundos por paso y una desviación estándar de 0.0330 segundos. La ejecución por tanto fue 18.28 veces más lenta en la plataforma virtualizada que en la nativa.

Figura 21. **Comparativa *PowerPC* emulado contra x64 nativo**



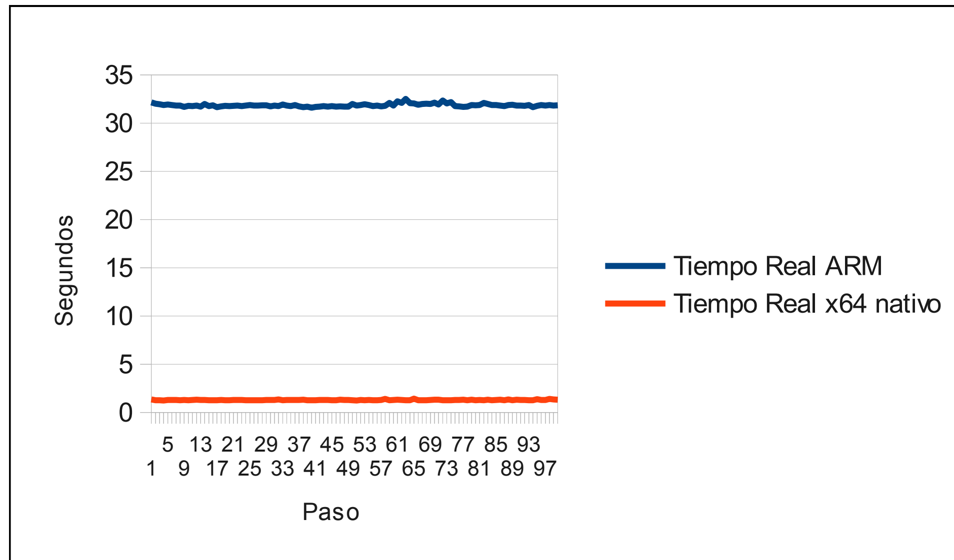
Fuente: elaboración propia.

5.2.2. ARM emulado contra x64 nativo

La ejecución del algoritmo en la máquina virtual ARM bajo los parámetros de configuración nativa, se observa el tiempo real de ejecución del mismo de 3185.75 segundos en 100 pasos, con un promedio de 31.8575 (figura 22) segundos por paso y una desviación estándar de 0.1509 segundos, comparado con la ejecución nativa en el *invitado* con arquitectura x64 que presenta una ejecución de 128.75 segundos en 100 pasos, con un promedio de 1.2875 segundos por paso y una desviación estándar de 0.0330 segundos.

La ejecución por tanto fue 24.74 veces más lenta en la plataforma virtualizada que en la nativa.

Figura 22. **Comparativa ARM emulado contra x64 nativo**



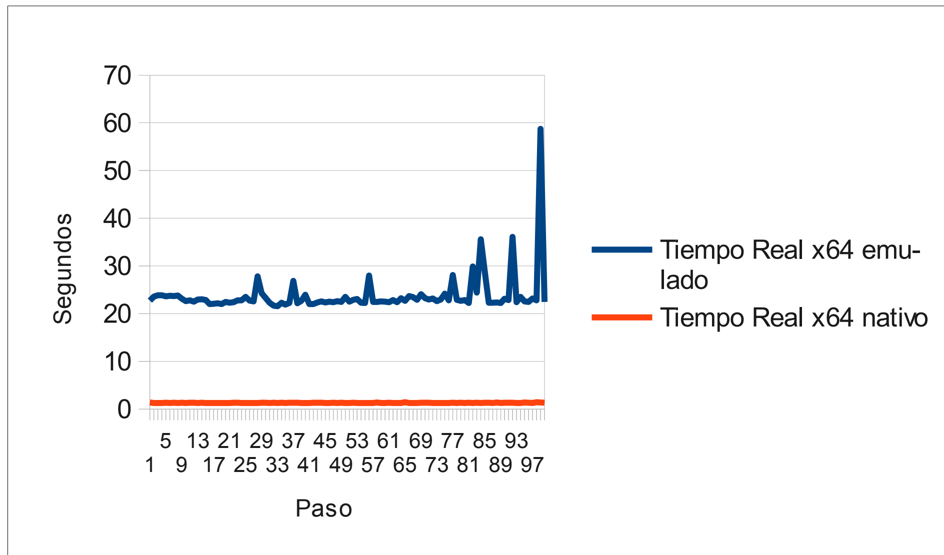
Fuente: elaboración propia.

5.2.3. X64 emulado contra x64 nativo

El algoritmo ejecutado en la máquina virtual x64 emulado bajo los parámetros de configuración nativa, se observa el tiempo real de ejecución del mismo de 2373.79 segundos en 100 pasos, con un promedio de 23.7379 (Figura 23) segundos por paso y una desviación estándar de 4.2164 segundos, comparado con la ejecución nativa en el *invitado* con arquitectura x64 que presenta una ejecución de 128.75 segundos en 100 pasos, con un promedio de 1.2875 segundos por paso y una desviación estándar de 0.0330 segundos.

La ejecución por tanto fue 18.4372 veces más lenta en la plataforma virtualizada que en la nativa.

Figura 23. **Comparativa X64 emulado contra x64 nativo**



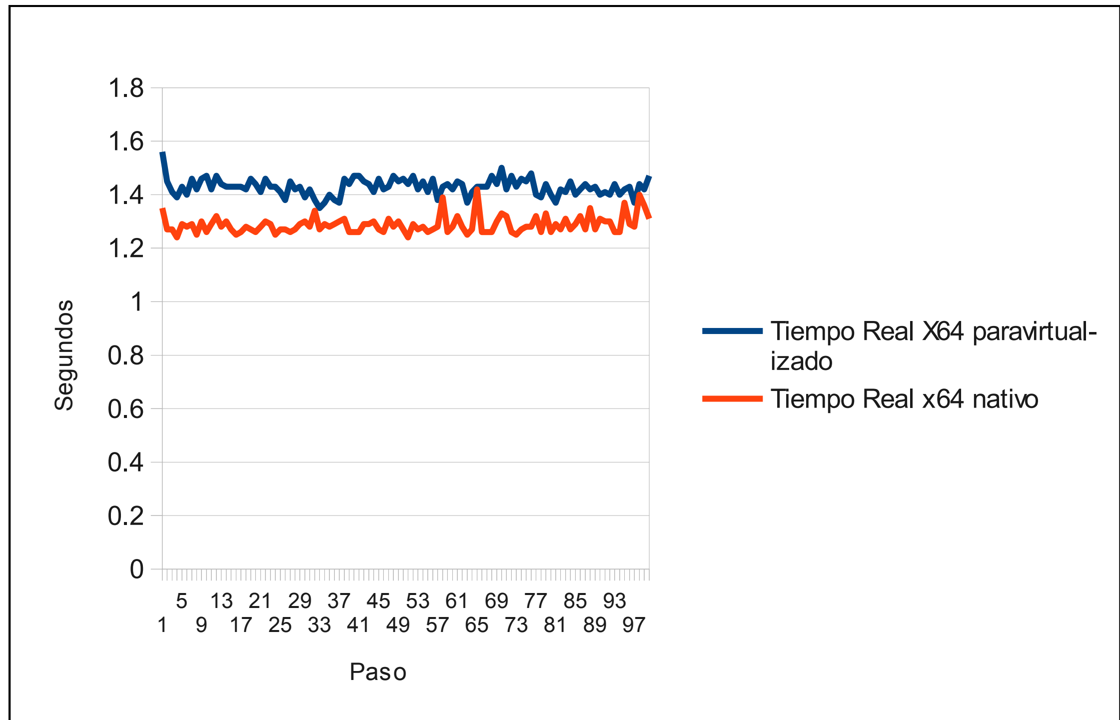
Fuente: elaboración propia.

5.2.4. X86-64 paravirtualizado contra x64 nativo

El programa ejecutado en la máquina virtual x64 emulado bajo los parámetros de configuración nativa, se observa el tiempo real de ejecución del mismo de 142.92 segundos en 100 pasos, con un promedio de 1.4292 (Figura 24) segundos por paso y una desviación estándar de 0.03212 segundos, comparado con la ejecución nativa en el *invitado* con arquitectura x64 que presenta una ejecución de 128.75 segundos en 100 pasos, con un promedio de 1.2875 segundos por paso y una desviación estándar de 0.0330 segundos.

La ejecución por tanto fue 1.11 veces más lenta en la plataforma virtualizada que en la nativa.

Figura 24. **Comparativa X64 emulado contra x64 nativo**



Fuente: elaboración propia.

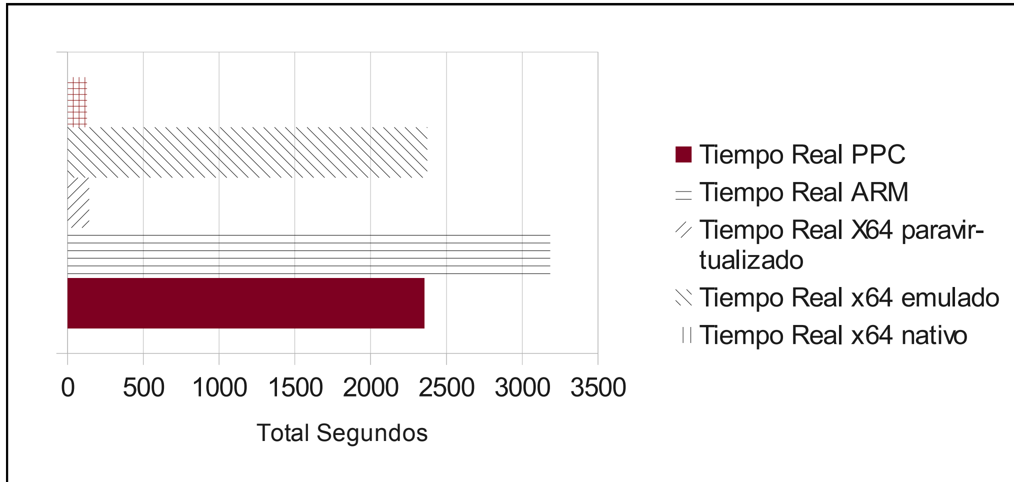
5.3. Resultados y validación de la hipótesis

La comparativa de resultados obtenidos es la siguiente:

5.3.1. Resumen de totales de tiempo real de ejecución

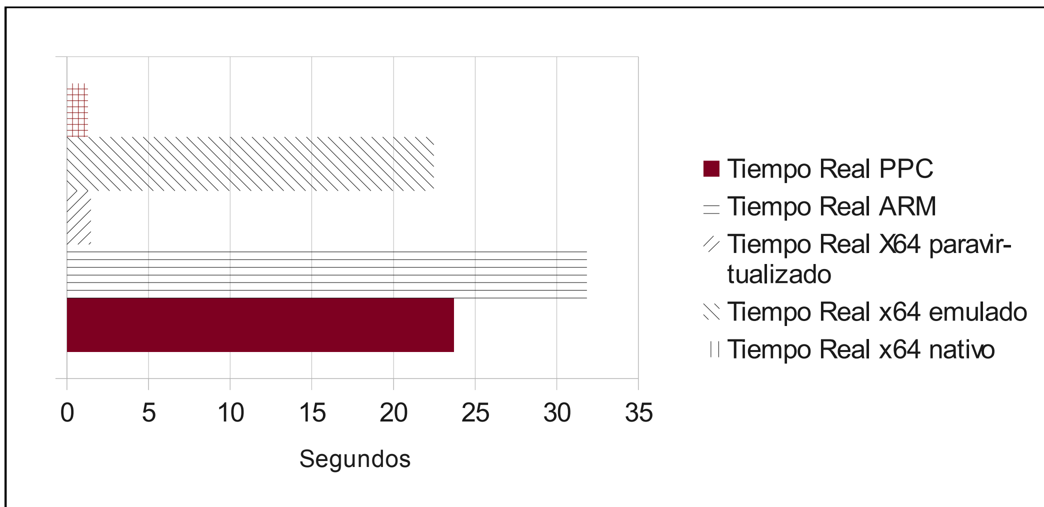
Los totales y promedios del tiempo real de la ejecución para el conjunto de instrucciones no Intel x86 resultaron de la siguiente manera:

Figura 25. **Total del tiempo real de ejecución**



Fuente: elaboración propia.

Figura 26. **Promedio del tiempo real de ejecución**



Fuente: elaboración propia.

- El tiempo de ejecución más alto se obtuvo al evaluar la ejecución del algoritmo sobre la máquina virtual ARM, con un promedio de 31.8575 segundos y una desviación estándar de 0.1509 segundos.

- La versión completamente emulada del conjunto de instrucciones x64 en la máquina virtual obtuvo un promedio de tiempo de 23.7379 segundos, con una desviación estándar de 4.21 segundos.

- Seguido por el conjunto de instrucciones *PowerPC*, del cual se promedió un tiempo de 23.5422 segundos por iteración o pasada, con una desviación estándar de 0.119 segundos.

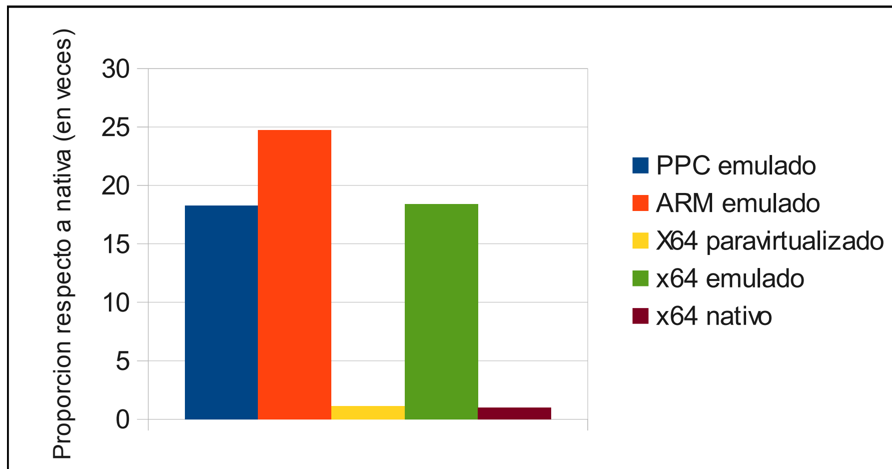
- Por último la ejecución paravirtualizada de x64 obtuvo un tiempo promedio de ejecución de 1.4292 segundos por iteración, con una desviación estándar de 0.03212.

Tomando como referencia, que la ejecución nativa de un ciclo del algoritmo tomó 1.2875 segundos con una desviación estándar de 0.03301 segundos.

5.3.2. Eficiencia proporcional de los emuladores respecto a la ejecución nativa en el anfitrión

El análisis de la eficiencia proporcional (siendo valor unitario un ciclo de ejecución en la máquina anfitrión) presenta los siguientes resultados (figura 27):

Figura 27. **Eficiencia proporcional respecto a la ejecución nativa**



Fuente: elaboración propia.

- Para la emulación ARM se requirió 24.74 veces el tiempo de ejecución total en los 100 pasos del algoritmo a lo que tomó de forma nativa en la máquina anfitrión.

- En la ejecución sobre PowerPC se requirió 18.28 veces más tiempo de al ejecutar los 100 ciclos.

- Para la emulación sobre x64, se requirió 18.43 veces más tiempo al ejecutar los 100 ciclos de procesamiento.

- En la paravirtualización de x64, se completó la ejecución en 1.11 veces el tiempo que tomó para la máquina nativa.

5.3.3. Conclusiones de la evaluación comparativa

En las evaluaciones realizadas entre diferentes arquitecturas se refleja una ventaja en eficiencia al ejecuta el algoritmo en una arquitectura PowerPC respecto a emular la máquina nativa x64, Esto fue provocado por la optimización matemática que presentan los binarios en un PowerPC, y su conjunto reducido de instrucciones, comparado a la cantidad de instrucciones emuladas en una arquitectura x64 tipo CISC.

El recompilador dinámico con menor eficiencia fue el de ARM, esto es debido a que es una arquitectura orientada a dispositivos móviles de bajo consumo de energía, lo cual hace que, incluso de forma nativa la ejecución de instrucciones lógico aritmética pesadas no sea tan eficiente como con procesadores PowerPC que están orientados a realizar esa tarea.

Según lo observado en la información, también se puede concluir de que la emulación con recompilación dinámica entre una misma arquitectura comparada con su emulación virtualizada, presenta una deficiencia en tiempo de ejecución bastante amplia, estando un 18% más lento al no utilizar las extensiones de paravirtualización del procesador y sistema operativo.

Se toma como válida la hipótesis planteada al observar que la velocidad de emulación por paravirtualización es inferior a dos veces el tiempo necesario (1.18 veces aproximadamente) en contra parte con su versión emulada para la misma arquitectura.

CONCLUSIONES

1. La capacidad de emular computadoras dentro de otras computadoras, tanto en el caso de los emuladores como de las máquinas virtuales, ha cambiado el paradigma desde la administración de servidores, como en el caso de los VMM, hasta la implementación de máquinas virtuales para máquinas abstractas que ayuden a crear un lenguaje intermedio genérico para aplicaciones multiplataforma, como en el caso de la JVM.

2. La emulación permite no solo ahorro de costos --como en el caso de los múltiples servidores para una sola tarea, al no existir necesidad de comprar *hardware* físico, sino incrustarlos todos estos servicios en servidores virtuales dentro de un mismo equipo--, sino además ahorro de energía –al no consumir más energía en equipo--, y también ahorro de espacio –al no tener que tener un cuarto dedicado para todo el conjunto de servidores--.

3. Los beneficios son demasiados comparados con las pequeñas desventajas, como la eficiencia temporal en la ejecución de una máquina virtual dentro de otra, las cuales se están logrando superar debido a técnicas de optimización del código en tiempo de ejecución, así como también el soporte de paralelismo en los microprocesadores de nueva generación, y sus capacidades de virtualización, las cuales facilitan el trabajo y le quitan carga a llamadas más costosas en el sistema operativo.

4. Las principales arquitecturas y comercialmente más exitosas son: ARM para móviles y sistemas embebidos de bajo consumo de energía, x86 para servidores y escritorio, y *PowerPC* para servidores y sistemas de videojuegos. Las tres diferentes entre sí, deben interactuar para que se permita la ejecución de binarios de una arquitectura a otra, ya que por ejemplo, el desarrollo de ARM para el sistema Android comúnmente se realiza sobre una máquina basada en el conjunto de instrucciones Intel x86 para luego ser probada sobre una máquina virtual.

5. La emulación y virtualización deben ser lo suficientemente eficientes para que su código pueda ser ejecutado lo más cercanamente posible al tiempo de ejecución que tomaría en correr en la máquina nativa.

6. Se realizó la evaluación comparativa diversas técnicas de emulación y recompilación dinámica para las arquitecturas anteriormente descritas, sin embargo como se refleja en los resultados presentados, no son lo suficientemente óptimas comparadas a su contra parte paravirtualizada, dado que la ejecución –incluso de código dentro de la misma arquitectura sobre un emulador con recompilación dinámica-- era varias veces más ineficiente que en la máquina nativa y su versión paravirtualizada, creando la necesidad de un crecimiento vertical respecto al poder de procesamiento para poder ejecutar el mismo código en las mismas máquinas virtuales.

7. Se refleja en los resultados, que, mientras la recompilación dinámica presenta una manera rápida de ejecutar código entre diferentes arquitecturas mediante emulación, la virtualización usando extensiones específicas del procesador y paravirtualización, como se refleja en los resultados de éste trabajo, es la manera más eficiente de virtualizar una plataforma dentro de otra, ambas de la misma arquitectura.

RECOMENDACIONES

1. Para la adquisición de equipo de cómputo para virtualizar, ya sea en menor o mayor escala, se debe elegir el que disponga de un procesador con extensiones de virtualización (como los que implementan las extensiones VTx de Intel por ejemplo), pues los beneficios en eficiencia son bastante notables.

2. Activar y utilizar cuando estén disponibles, las extensiones de virtualización que provea el procesador del sistema anfitrión, ya que esto representará un gran ventaja en eficiencia en el momento de virtualizar.

3. Al emular arquitecturas diferentes al que provee el anfitrión, utilizar los motores que provean compiladores JIT o recompiladores dinámicos, debido a que es la solución más eficiente para emulación inter arquitectura.

BIBLIOGRAFÍA

1. *Add with Carry* [en línea]. <http://siyobik.info/index.php?module=x86&id=4>. [Consulta: 5 de noviembre de 2009].

2. *Application Binary Interface* [en línea]. <http://en.wikipedia.org/wiki/Application_binary_interface>. [Consulta: 1 de septiembre de 2009].

3. ATTARDI, G.; CISTERNINO, A.; COLOMBO, D. *Common Language Runtime: A new virtual machine, Journal of Object Technology*, Citeseer 2004.Vol. 3. Num. 2. p.1-5

4. AYCOCK, Christopher C. *What is Loop Level paralelism?* [en línea]. <http://insidehpc.com/2006/03/11/what-is-loop-level-parallelism/>. [Consulta: 3 de septiembre de 2009].

5. BARHAM, Paul. *Xen and the Art of Virtualization* [en línea]. <www.cl.cam.ac.uk/research/srg/netos/papers/2003-xensosp.pdf>. [Consulta: 25 de agosto de 2009].

6. *Bytecode instructions of the Java™ Virtual Machine* [en línea]. <http://www.stackframe.com/documents/bytecode.html>. [Consulta: 7 de diciembre de 2009].

7. CRAIG, I. *Virtual Machines.*, Londres: Springer, 2006, p. 27 ISBN 1852339691

8. CROSBY, Simon. *The virtualization Reality.* Revista Queue, Vol. 4, Diciembre-Enero 2006,2007, p. 34-41, 2006.

9. DIBBLE, Peter. *Real Time Java Platform Programming.* California: Prentice Hall 2002, p. 14-36. ISBN 0130282618,

10. *Dynamic Recompilation Introduction* [en línea]. <http://forums.ngemu.com/web-development-programming/20491-dynamic-recompilation-introduction.html>. [Consulta: 10 de diciembre de 2009].

11. *Dynamic Recompilation Introduction* [en línea]. <http://forums.ngemu.com/web-development-programming/20491-dynamic-recompilation-introduction-2.html#post240799>. [Consulta: 10 de diciembre de 2009].

12. GARRAUX, Oliver. *How virtualization works and its effects on IT information technology fundamentals* [en línea]. <http://www.garraux.net/files/cit112_virtualization.pdf>. [Consulta: 1 de febrero de 2010].

13. GUGLIELMETI, Marcos. *Multitareas* [en línea] <http://www.mastermagazine.info/termino/6039.php>. [Consulta: 5 de octubre de 2009].

14. *High level emulation* [en línea]. <http://www.worldlingo.com /ma/enwiki/en/High-level_emulation>. [Consulta: 7 de septiembre de 2009.

15. HOOOKWAY, Herdeg. *Combining emulation and binary translation*, Digital Tech. Journal. Enero 1997, p. 3.

16. *iBCS* [en línea]. <http://www.everything2.com/index.pl?node=iBCS>. [Consulta: 3 de septiembre de 2009].

17. *Instruction Level Paralelism* [en línea]. <http://en.wikipedia.org/wiki/ File:Fivestagespipeline.png>. [Consulta: 6 de noviembre de 2009].

18. *Java Threads, Java 1.4 API documentation* [en línea]. <http://java.sun.com/j2se/1.4.2/docs/api/java/lang/Thread.html>. [Consulta: 6 de noviembre de 2009]

19. *Jit Overview* [en línea]. <http://publib.boulder.ibm.com/ infocenter/javasdk/v6r0/index.jsp?topic=/com.ibm.java.doc .diagnostics.60/html/jit_overview.html>. [Consulta: 9 de diciembre de 2009].

20. *La Tesis de Church-Turing* [en línea]. <http://es.wikibooks.org/ wiki/La_tesis_de_Church-Turing/TextoCompleto>. [Consulta: 2 de noviembre de 2009].

21. *Ley de Amhdal* [en línea]. <http://www.tomechangosubanana.com/tesis/escrito-1-split/node18.html>. [Consulta: 1 de noviembre de 2009].

22. *Linux Internals* [en línea]. <http://learnlinux.tsf.org.za/courses/build/internals/internals-all.html>. [Consulta: 26 de agosto de 2009].

23. MANDAL, Abhijit. *Design and Implementation of Java Virtual Machine*, Mumbay: Indian Institute of Science, 2005. 132 p.

24. NARENDAR B. Sahgal. *Understanding Intel Virtualization Technology (VT)* [en línea]. <http://www.docstoc.com/docs/45758154/Understanding-Intel-%C2%AE-Virtualization-Technology-%28VT%29>. [Consulta: 7 de enero de 2010].

25. *PowerPC Programming by the Book, a Developer's guide to PowerPC Architecture* [en línea]. <http://www.devx.com/ibm/Article/20943>. [Consulta: 22 de diciembre de 2009].

26. *QEMU internal Documentation* [en línea]. <http://www.qemu.org/qemu-tech.html#SEC9>. [Consulta: 15 de enero de 2010].

27. SESTOFT, Peter. *Runtime code generation with JVM and CLR.* Copenhagen: Royal Veterinarian and Agricultural University of Copenhagen, 2004. 28 p.

28. SINGH, Arindama. *Elements of computational theory.* Londres: Springer, 2009. ISBN 18680941.

29. SMITH, James Edward; RAVI, Nair. *Virtual machines: versatile platforms for systems and processes.* San Francisco: Elsevier, 2005. ISBN 1558609105.

30. *Super NES programming guide* [en línea]. <http://en.wikibooks.org/wiki/Super_NES_Programming/SNES_Specs>. [Consulta: 4 de enero de 2010].

31. *TCP/IP stack* [en línea]. <http://www.tcpipguide.com/free/t_TCPIPArchitectureandtheTCPIPModel-2.htm>. [Consulta: 25 de agosto de 2009].

32. *The Java virtual machine specification* [en línea]. <http://java.sun.com/docs/books/jvms/second_edition/html/VMSpecTOC.doc.html>, [Consulta: 7 diciembre de 2009].

33. VAN DOORN LEENDERT, T. *Hardware Virtualization Trends.* Palo Alto, California: Watson Research Center, 2006. 44 p.

34. *Zsnes Emulator* [en línea]. <http://en.wikipedia.org/wiki/Super_Nintendo_Entertainment_System#Technical_specifications>. [Consulta: 03 de enero de 2010].

35. *ZSNes Readme* [en línea]: <http://zsnes-docs.sourceforge.net/html/readme.htm>. [Consulta: 25 de agosto de 2009].

APÉNDICE A. SCRIPT EN PYTHON DEL ALGORITMO EVALUADO

```
import threading
import math

def primefactors(x):
        factorlist=[]
        loop=2
        while loop <= x:
                if x%loop == 0:
                        x /= loop
                        factorlist.append(loop)
                else:
                        loop += 1
        return factorlist

class MyThread ( threading.Thread):
        def run(self):
                for i in range (10000):
                        print primefactors(i)

MyThread().start()
```

APÉNDICE B. SCRIPT EN BASH PARA LA EJECUCIÓN DEL SCRIPT EVALUADO

```
echo "Iniciando evaluacion"
for i in {1..100}; do
        echo iteration: $i && \
        touch time.txt && \
        /usr/bin/time -o time.txt -a -f $i,%E,%S,%U python2
primefactor.py > /dev/null;
done
echo "Fin de evaluacion"
```