Sistemas, Virtualización y Seguridad

Jesús Durán Hernández
Rebeca Bárcena Orero

Curso 2016-2017

Índice

1. Práctica 1: Caracterización del rendimiento **4**
1.1. Problemas iniciales 4
1.2. Tarea 1 4
1.3. Tarea 2 8

2. Práctica 2: Xen **13**
2.1. Tarea 1 13
2.2. Tarea 2 13
2.3. Tarea 3 13

3. Práctica 3: Optimización del rendimiento **15**
3.1. Tarea 1 15
3.2. Tarea 2 15
3.3. Tarea 3 16

4. Práctica 4: Containers & Public Clouds **18**
4.1. Tarea 1 18
4.2. Tarea 2 20
4.3. Tarea 3 21

5. Herramientas Utilizadas **23**

Índice de figuras

1. Resultados con 32 y 64 bits 6
2. Comparación del predictor de saltos 7
3. Comparación IPC 8
4. JBB con OpenJDK 9
5. JBB con Java de Oracle 10
6. Comparación OpenJDK y Java de Oracle 10
7. Resultados de SPECweb 12
8. Comparación CPU nativo y HVM 16
9. SPECjbb en nativo y HVM 17
10. SPECjbb en LXC 18
11. Comparación JBB entre nativo y LXC 19
12. Comparación SPECweb entre nativo y Docker 21
13. Comparación *Docker* y *Google* 22

1. Práctica 1: Caracterización del rendimiento

1.1. Problemas iniciales

Al comenzar con esta práctica, se encontraron algunos problemas con el sistema operativo Debian. El primer problema era que no se sabía cuál era la contraseña del usuario *root*, por lo que se tenía que cambiar a una conocida. Para ello, se entró al cargador GRUB, se añadió el texto *"init=/bin/bash"* y se reinició el sistema. Al reiniciar, ya se podía introducir la nueva contraseña de *root* que se quisiese.

También había problemas con los repositorios desde los cuáles actualiza el software, por lo que se tuvieron que introducir las URLs correctas modificando el archivo */etc/atp/sources.list*. Una vez actualizados los repositorios, se actualizó con *apt-get update* y se instalaron algunos paquetes con *apt-get install xfce4* y *apt-get install Linux-tools*.

1.2. Tarea 1

En la tarea 1 se trabajará con un *benchmark*, el SPECcpu. Lo primero que se hará será bajarlo de Internet y descomprimirlo con el comando *tar -xzvf*. Posteriormente, desde la carpeta descomprimida, habrá que ejecutar el *install* y *shrc* porque se trabaja desde en una terminal de tipo **bash** (comandos *./install.sh* y *. ./shrc* respectivamente).

Debido a la carpeta donde se descomprimió el directorio de SPECcpu, hubo problemas para ejecutar correctamente ambos ficheros, por lo que fue necesario modificar sus propietarios (*chown -R usuario:grupo **) y sus permisos (*chmod -R u+w **). Después, hay que configurar el entorno para *C, C++*y *Fortran*. Lo primero es instalar los compiladores para estos lenguajes. Para el correcto funcionamiento de ellos, hay que instalar, con *apt-get install*, los paquetes *libc6-dev, gcc, g++, dpkg-dev, make* y *gfortran*, y ejecutar el *shrc* (*. ./shrc*).

Una vez hecho esto, se creará una configuración propia. Para ello, dentro de la carpeta *config*, se buscará el archivo que más se asimile a la arquitectura con la que se trabaja, en nuestro caso *linux64-amd64-gcc42.cfg*, se copiará el archivo y se modificarán las líneas donde aparecen los *paths* de los compiladores de *C, C++* y *Fortran*. Para ver los *paths* donde están instalados los compiladores se puede ejecutar el comando *whereis <nombrePaqueteCompilador>*. A continuación, procederemos a la compilación. Usaremos el comando *runspec* con la opción *build* para sólo compilar y el nombre del fichero que hemos modificado para que sea nuestra propia configuración (*runspec –config= linux64-amd64-gcc42.cfg –action=build –tune=base –ignore_errors –noreportable –size=train*).

El problema encontrado es que al haber guardado fuera de la carpeta *config* la nueva configuración, el comando no la encuentra y sale un error de compila-

ción en *401.bzip2* (para ver mejor los errores, se puede entrar en el *log* correspondiente dentro de la carpeta *result*). Con mover el archivo de configuración a la carpeta *config* y darle otro nombre (*my-config* por ejemplo) para que no haya problemas con el ya existente y volver a ejecutar el comando con la opción *–config=my-config* se soluciona el problema.

Una vez solucionado esto, se procede a la compilación de todos los *benchmarks*, ejecutando *runspec –config=my-config.cfg –action=build –tune=base –ignore_errors –noreportable –size=train all*. Una vez acabada, se notifica que hay errores en *447.dealII(base)* y en *483.xalancbmk(base)*. Se pueden ver los errores con más detalle accediendo al *log* que se crea al compilar.

Para que el *xalancbmk* funcione, en el archivo de la configuración, tenemos que tener *"CXXPORTABILITY=-DSPEC_CPU_LINUX -include cstdlib -include cstring"*. También es necesario comentar una línea, *wrf_data_header_size=8*, para que *481.wrf* funcione correctamente. El *dealII*, como tiene un fallo más difícil de solucionar, no se ha arreglado.

Para ejecutar el *benchmark* en 32 bits, es necesario recompilar los archivos. Se ha modificado el fichero de configuración y se añade el *flag "-m32"* para especificar al compilador que lo haga para 32 bits. Una vez hecho esto, se ejecuta de nuevo con el comando *runspec*.

En cuanto a FDO, se tiene que modificar el fichero de configuración para que admita la opción *peak* cuando se ejecuta *runspec*. Para ello, se puede buscar un ejemplo en la carpeta *config* del SPECcpu (dentro del directorio *config*, ejecutar *grep 'FDO' **) y copiar la parte referente a *peak* en el archivo de configuración, dentro de la sección de optimización. Se debería de desactivar el *turbo boost* para que no interfiera en los resultados, pero finalmente se han hecho las pruebas sin hacerlo ya que no se podía. Se ha ejecutado el *runspec* con opción *run* para los *benchmarks 456.Hmmer (INT)* y *433.Milc (FP)*, tanto para base como para *peak*.

Tras realizar todos estos pasos, se han obtenido los siguientes resultados del benchmark:

Benchmark	Tiempo de ejecución en 32 bits (s)	Tiempo de ejecución en 64 bits (s)
456.Hmmer	68.2	37.3
433.Milc	13.9	11.9

Tabla 1: Tiempos de ejecución en 32 y 64 bits

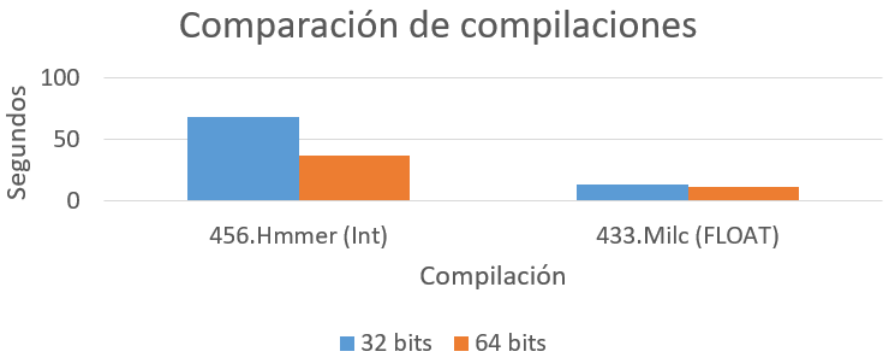

Figura 1: Resultados con 32 y 64 bits

El *milc* no tiene tanta diferencia en tiempo de ejecución entre sus versiones (32 y 64 bits) porque es *memory bound* y la mayor parte del tiempo de ejecución la pasa realizando operaciones en memoria, así que las instrucciones que ejecuta la CPU no representan una gran parte del tiempo de ejecución. En cambio, el *Hmmer*, como es *CPU bound*, pasa la mayor parte de su tiempo de ejecución en la CPU. Por eso el cambio es tan significativo (casi el doble de tiempo) de ejecutarlo compilado en 32 o en 64 bits. Sin embargo, como se ha ejecutado el *runspec* con *train* en lugar de *ref* los resultados no son tan significativos.

En cuanto al *peak*, para el *Hmmer* nos daba 360, pero con el *Milc* no se ha conseguido ejecutar, por lo que no se han obtenido conclusiones.

Por último se ha usado el comando *perf* para analizar algunos parámetros de rendimiento del sistema. Para ello se ha usado el manual de Desarrolladores de Software para las arquitecturas Intel 64 e IA-32, en el cual aparecen los códigos que hay que pasar como parámetro al *perf* para que devuelva los datos que se deseen. En este caso se ha calculado cuál es la eficiencia del predictor de saltos y la *memory speculation*. Los resultados obtenidos se ven en las siguientes tablas:

Benchmark / medida	Branch Instruction Retired	Branch misses retired	Eficiencia predictor de saltos
456.Hmmer	46438447671	577872199	98.755 %
433.Milc	11089511490	125849648	98.865 %

Tabla 2: Medidas para los *benchmarks*

Benchmark / medida	Unhalted Core Cycles	Instructions retired	Instrucciones por ciclo
456.Hmmer	417441814142	836635219133	2 inst/ciclo
433.Milc	141568376365	139261640848	1 inst/ciclo

Tabla 3: Continuación de las medidas para los *benchmarks*

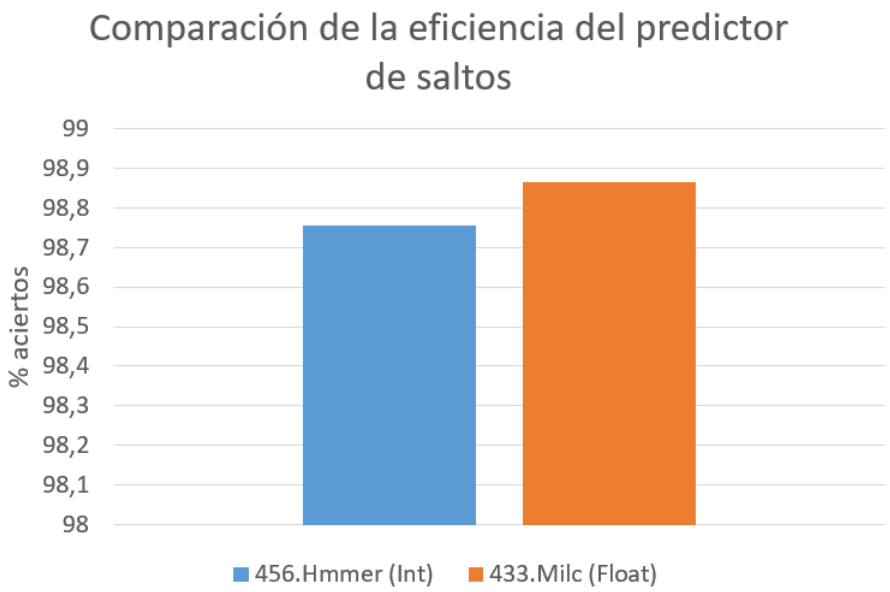

Figura 2: Comparación del predictor de saltos

Comparando uno con otro vemos que el predictor es prácticamente igual; sin embargo, ninguno de los dos son muy buenos, porque más de un 1 % de errores cuando se tienen cientos de instrucciones en vuelo se convierte en una tasa de fallo mayor, que puede influenciar negativamente en el rendimiento.

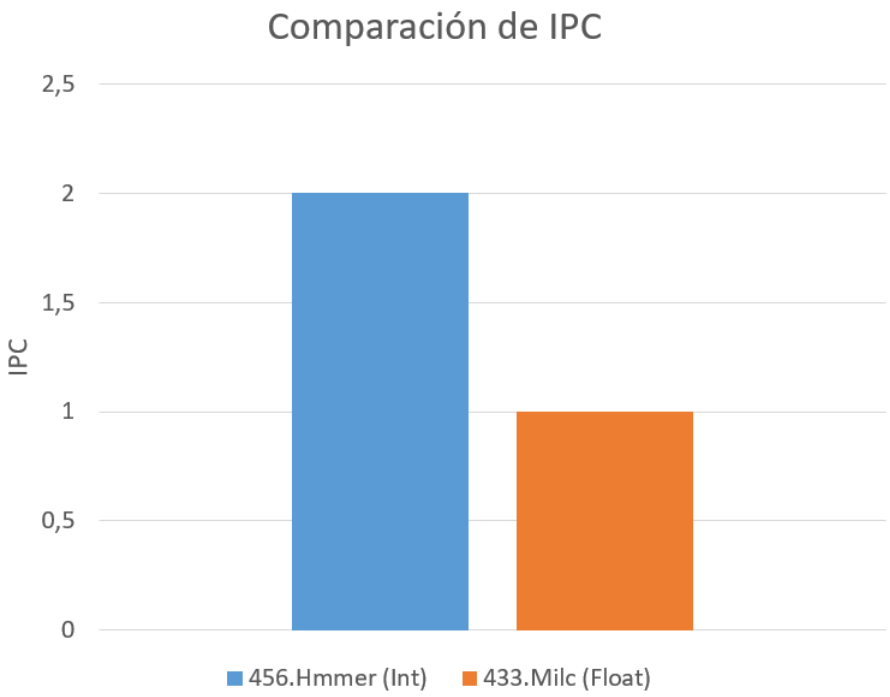

Figura 3: Comparación IPC

Si comparamos sus IPCs, vemos que el *Milc*, que pasa más tiempo en memoria, hace una instrucción por ciclo, ya que las instrucciones de acceso a memoria son más lentas y costosas. Por el contrario, el *Hmmer*, que pasa más tiempo en la CPU, hace el doble de instrucciones por ciclo.

Otra medida que se podría haber obtenido para ver qué *benchmark* depende más de la memoria y cuál de la CPU sería comprobar, por ejemplo, el número de instrucciones que entran al *pipeline* y el que se retira, calculando así el número de instrucciones falladas, que será mayor en el *benchmark* que depende más de la CPU.

1.3. Tarea 2

En esta tarea trabajaremos con los *benchmarks* SPECjbb y SPECweb. El primero sirve para evaluar el rendimiento de Java en el servidor, mientras que SPECweb evalúa el rendimiento de los servidores destinados a conexiones de red.

En cuanto al SPECjbb, compararemos su rendimiento al ejecutarlo con el Java de OpenJDK y el Java de Oracle. Primero, instalamos el Java de

OpenJDK con el comando, y ejecutamos el *benchmark*. Inicialmente, tras buscar documentación en Internet para configurar Java, se utilizó el comando *update-alternatives*, para conformar la configuración del compilador de Java. Sin embargo, no funcionaba correctamente usando este método, por lo que se buscó otro. Finalmente se encontró un método para configurarlo a través de variables de entorno. Para ello se establecen las variables de entorno $JAVA_HOME y $PATH. En la primera hay que especificar la ruta a la carpeta donde hayamos descomprimido el fichero de Java, y en la segunda la carpeta con los fichero binarios (misma ruta añadiendo */bin* al final).

Tras descomprimir el archivo con el *benchmark*, se configura el archivo *SPECjbb.props* para configurar el número de *warehouses* con los que se realizará la prueba. Configuramos que comience con uno, termine con seis, y con saltos de uno en uno. En el fichero *SPECjbb_config.props* configuramos el espacio de memoria que le asignamos a la máquina virtual de Java, y lo establecemos en 2048MB, ya que al ejecutarlo inicialmente con 1024MB no conseguía terminar la ejecución porque necesitaba más memoria. Los resultados obtenidos se muestran en la siguiente imagen:

Warehouses	bops	Incl. in metric
1	66388	
2	129975	
3	167900	
4	170817	*
5	167384	*
6	113612	*
SPECjbb2005	(from 4 to 8)	90363 bops

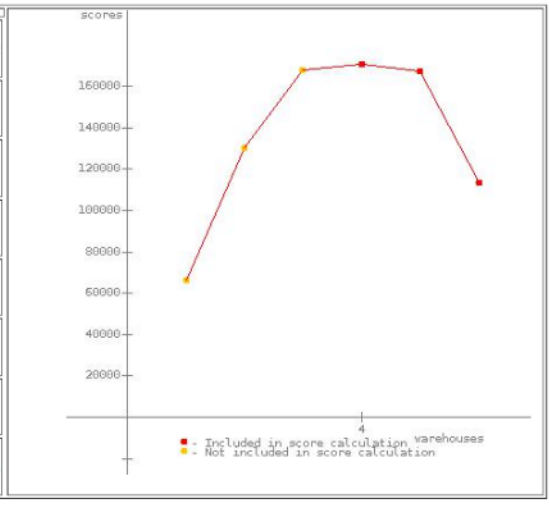

Figura 4: JBB con OpenJDK

Para ejecutarlo con el Java de Oracle, desinstalamos el OpenJDK para que no hubiera conflictos, configuramos las variables de entorno y ejecutamos de nuevo el *benchmark*. Obtenemos estos resultados:

Warehouses	bops	Incl. in metric
1	66371	
2	132187	
3	166667	
4	170513	*
5	167705	*
6	38000	*
SPECjbb2005	(from 4 to 8)	75243 bops

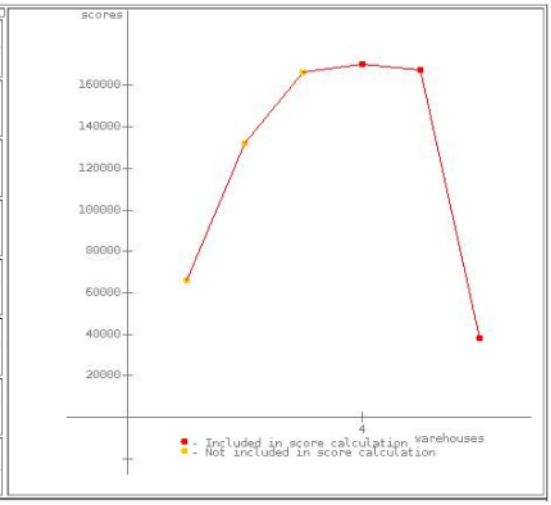

Figura 5: JBB con Java de Oracle

Se puede ver mejor la comparación en la siguiente gráfica:

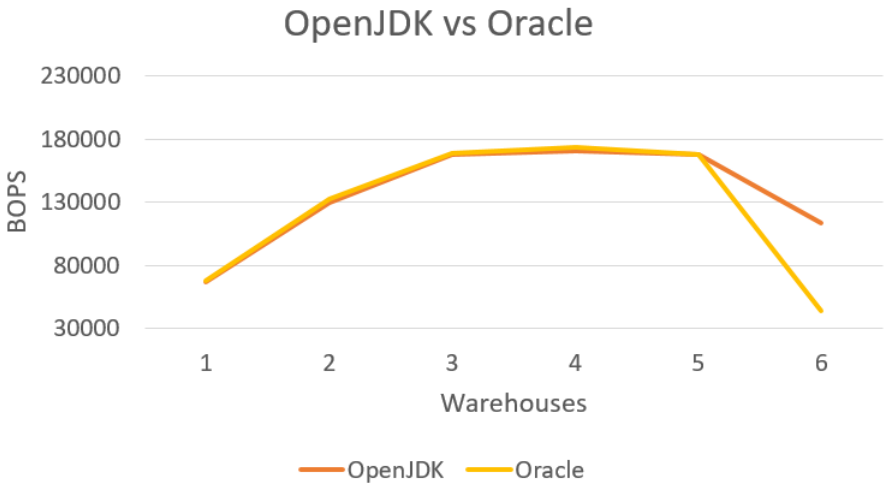

Figura 6: Comparación OpenJDK y Java de Oracle

Como se puede ver, los resultados de ambos tipos de Java desde un *warehouse* hasta cinco son muy similares, teniendo ambos el pico más alto con n=4 *warehouses*. Esto se debe a que nuestra máquina tiene cuatro núcleos. Por

esa razón los resultados a partir de n=4 *warehouses* comienzan a descender. Además, se percibe una caída mucho mayor para n=6 cuando utilizamos el Java de Oracle (38000 bops) que cuando lo ejecutamos con el OpenJDK (113612 bops).

Respecto a SPECweb, se ha bajado y descomprimido, y se han seguido las instrucciones dadas por el profesor para instalarlo. Ha sido necesario descargar también Apache2, PHP5 y fastcgi (un protocolo para interconectar programas interactivos con un servidor web). Dentro de SPECweb hay tres *benchmarks*:

- *Support*: es el más fácil, no requiere SSL y es simple HTML estático.
- *Ecommerce*: simula un comercio electrónico y tiene contenido estático y dinámico.
- *Banking*: simula un banco y requiere SSL. Todo el contenido es dinámico.

Primero se ha configurado y probado el *Support*, por ser el más fácil, para lo que ha sido necesario crear los archivos estáticos e ir haciendo diferentes pruebas para poder ver hasta con qué número de sesiones se puede ejecutar antes de que falle.

Para solucionar algunos de los fallos ha sido necesario copiar los scripts que se encontraban en la carpeta de SPECweb al directorio */var/www*, ya que sino no funcionaba correctamente, dar permisos al usuario *apache*, crear el archivo *init_vars.php* (el *benchmark* usa este fichero para escribir alguna variable) y comprobar que el *Wafgen* se ha generado acorde al número de clientes. Dentro de este *benchmark*, a su vez, hay tres módulos:

- Clientes: clientes que acceden a la web (se pueden modificar a través del número de sesiones).
- *BeSim*: simulador de *back-end* ligero y eficiente para representar la parte del servidor de aplicaciones y bases de datos de la arquitectura.
- *Web server*: servidor de la web.

A través de los archivos de configuración se pueden modificar el número de sesiones, es decir, de clientes, que acceden a la web, y con los *scripts wafgen* se generan los datos en función de ellas. En este caso se han realizado pruebas para varios números de clientes:

Clientes	*% Good*	*% Tolerable*	*% Fail*
100	100	100	0
145	100	100	0
150	99.8	100	0
160	31.8	51.5	48.5
200	6.5	6.7	93.3

Tabla 4: Resultados de SPECweb

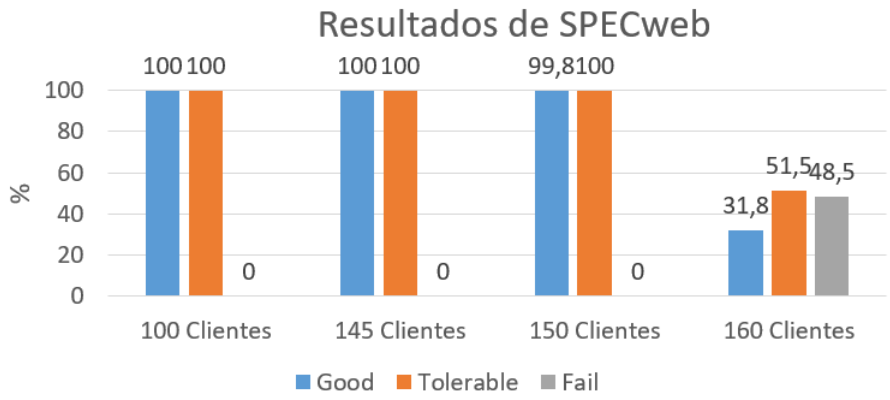

Figura 7: Resultados de SPECweb

Como se ve en la gráfica, hasta 150 clientes todo funciona correctamente, pero al pasar de este número de clientes el sistema comienza a funcionar mal. En cuanto al SPECweb *Ecommerce*, se ha intentado configurar y ejecutar, pero no se ha conseguido que funcionase correctamente.

2. Práctica 2: Xen

2.1. Tarea 1

Esta tarea se ha centrado en la preparación de Xen, distinguiendo una pre-instalación, una instalación y una post-instalación.

En cuanto a la instalación, hay que comprobar si VT-x está o no activado, y en caso de que no sea así, activarlo en la BIOS. En nuestro caso, no se podía cambiar porque no se sabía la contraseña para acceder a la BIOS. Sin embargo, se comprobó que estaba activado utilizando el comando *grep –color vmx /proc/cpuinfo*. Posteriormente, se creó una partición de 40 GB, que es con la que se trabajará a partir de ahora, y un volumen lógico dentro de ella, que es donde se creará la primera máquina virtual.

Respecto a la instalación, se descargó Xen 4.1 desde repositorio y se configuró Xen como entrada por defecto en el GRUB, modificando el archivo de configuración de GRUB y especificando cuál es la entrada por defecto en el *GRUB_DEFAULT*. Para configurar el máximo y mínimo de memoria del dom0 configuramos el dom0 con una CPU y 2 GB.

Finalmente, tras la instalación, se ha configurado la red para Xen, descomentando la línea *network-script network-bridge* del archivo de configuración de Xen (se hace un bridge entre la red de la máquina real y Xen y se puede usar DHCP para asignar las IPs a los domUs), *xen-config*, y se han activado *X11 forwarding* en el dom0 para tener interfaz gráfica.

2.2. Tarea 2

Inicialmente se ha creado una instalación de Debian 7 (wheezy) en un domU, sobre un volumen lógico de 5 GB. A partir de un archivo de configuración de un domU y modificándolo, se creó el domU llamado nuevaVM y el volumen lógico, llamado base, se creó de forma similar al del apartado anterior. Posteriormente, se descargó SPECcpu2006 y se creó un clon de este domU usando un *snapshot*, por lo que se tuvieron que crear dos volúmenes lógicos más, ya que más adelante se pide que se cree otro clon.

2.3. Tarea 3

En cuanto a las tareas de gestión de domUs, se han añadido domUs al *xend-store*. Para ello se ha instalado *phyton-lxml* y se han añadido con el comando *xm new* y el archivo de configuración de la máquina.

Para instalar *kernels* desde *back-ports* es necesario añadirlo a */etc/apt/-sources.list* y, si no existe, crear */etc/apt/preferences*. Después se descarga el *back-ports* y ya está todo listo para descargar el *kernel* que se desee. Por último,

para eliminar los *snapshots* y clones, se eliminan los volúmenes lógicos y los archivos de configuración respectivos.

3. Práctica 3: Optimización del rendimiento

3.1. Tarea 1

En esta tarea se crea una instalación HVM en un domU, por lo que se ha creado un nuevo fichero de configuración para dicha instalación y un nuevo volumen lógico de 15 GB, sobre el que se ha creado. Dentro de este domU se han descargado los *benchmarks* SPECcpu2006, SPECjbb2005 y SPECweb2005, ya que posteriormente se trabajará con algunos de ellos.

En un principio se usaba el *front-end* gráfico del dom0 que viene por defecto para acceder al domU (*vncviewer*), pero después se configuró la consola para poder usar la de Xen, ya que es mejor. Al utilizar esta consola se tuvieron problemas, ya que la disposición del teclado no era la española. Se consiguió solucionar con el comando *dkpg-reconfigure keyboard-configuration*, luego se reinicia el servicio con *service keyboard-setup restart*. Para hacerlo permanente se modifica el fichero */etc/default/keyboard*.

A continuación se configuró la máquina HVM y el volumen lógico en el que estaba para que soportase PV. Para ello se usa la imagen de HVM, se añade la consola *hvc0* al */etc/inittab* y se habilitan los *drivers* PV. Tras hacer esto, ya se puede usar HVM y PV con la misma imagen, pero no arrancarlas simultáneamente. Sin embargo, no se ha conseguido que funcione PV, ya que arrancaba, pero al iniciar sesión, entraba en el sistema y después volvía a salirse para pedir el usuario y contraseña otra vez. Para intentar solucionarlo se ha buscado información en Internet, pero no se ha conseguido solucionar el fallo tras probar los ejemplos que se daban. Es por ello que en los apartados en los que había que hacer pruebas con PV se han realizado con HVM.

3.2. Tarea 2

Como se ha explicado anteriormente, las pruebas se han realizado comparando HVM y PV, ejecutando SPECcpu (se ha descargado y configurado como se explicó en la Tarea 1.2 de la Práctica 1: Caracterización del rendimiento). Si se hubiese conseguido hacer funcionar PV sólo habría que haberlo ejecutado en la máquina PV normal, ya que como no usa VT-x no hace falta hacer dos pruebas, ya que su activación o desactivación no influye.

Los resultados obtenidos se ven a continuación:

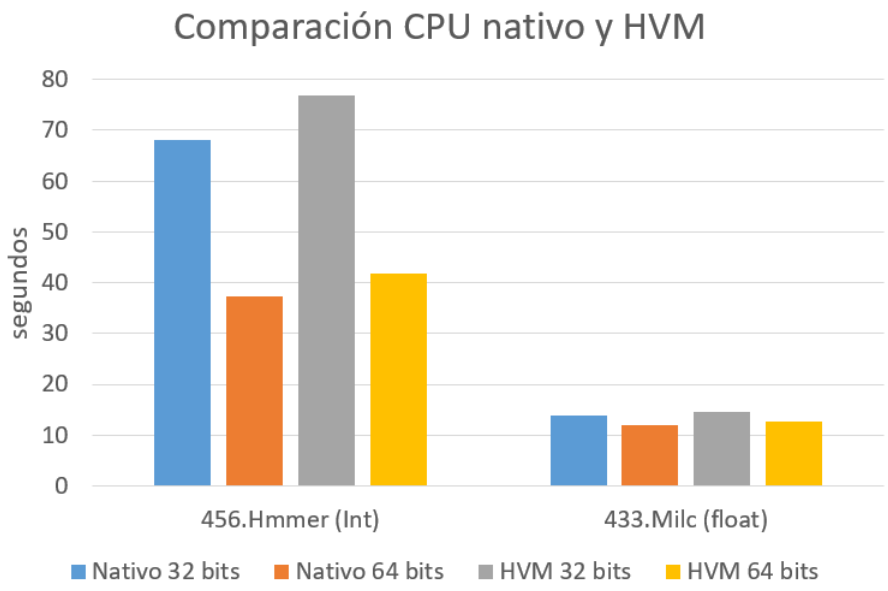

Figura 8: Comparación CPU nativo y HVM

Como se admira en la gráfica, en nativo se tiene un mejor rendimiento, tanto en 64 como en 32 bits. En el *Milc* no se aprecia tanto, ya que se ejecuta en pocos segundos, pero en el *Hmmer* se aprecia mejor que la máquina virtual HVM pierde rendimiento.

3.3. Tarea 3

Se han configurado dos domUs con HVM, ya que no se ha conseguido una virtualización mejor, con SPECjbb y el JDK de Oracle (la versión 1.8). Uno, el que se muestra en la gráfica, tiene 4 CPUs, y el otro 1, que comparte con el primero. Además, se le asigna un peso mayor (1024) con *xm sched-credit -d <domName> -w 1024*. Por defecto este peso tiene un valor de 256.

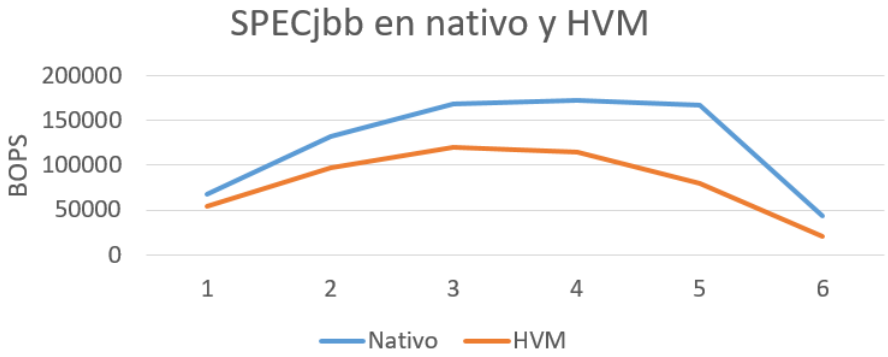

Figura 9: SPECjbb en nativo y HVM

Como se ve el pico en HVM se alcanza en el tercer *warehouse*, ya que cuenta con 3 CPUs exclusivas. Por el contrario, en nativo sigue creciendo hasta el cuarto *warehouse*, aunque entre el tercero y el cuarto apenas se nota la diferencia. Luego los BOPS de ambos caen, ya que se comparten las CPUs entre los *warehouses* y se pierde rendimiento con los cambios de contexto. También se ve que el rendimiento de HVM es siempre inferior que el de nativo, al tratarse de una máquina virtual.

4. Práctica 4: Containers & Public Clouds

4.1. Tarea 1

Para poder usar LXC, se instala *wheezy-backports* y se añade *backports* a los repositorios en */etc/atp/sources.list*. Se crea un contenedor LXC *my_jessie* con la última versión de debian y con el SPECjbb que se utilizará en esta tarea. También se pone red al contenedor, modificando el fichero */var/lib/lxc/my_jessie*, y se descarga el SPECjbb en él. Después, se hace un clon del contenedor para crear *clon1* con el mismo contenido que hay en *my_jessie*.

Con el objetivo de conseguir el 75 % del rendimiento de la ejecución nativa, se configuran los *cgroups*. Se asigna una sola CPU al contenedor *clon1*, y tres CPUs al contenedor *my_jessie*. Los resultados obtenidos en *my_jessie* son:

Warehouses	bops	Incl. in metric
1	65919	
2	115023	
3	139986	
4	126697	*
5	117716	*
6	27178	*
SPECjbb2005	(from 4 to 8)	54318 bops

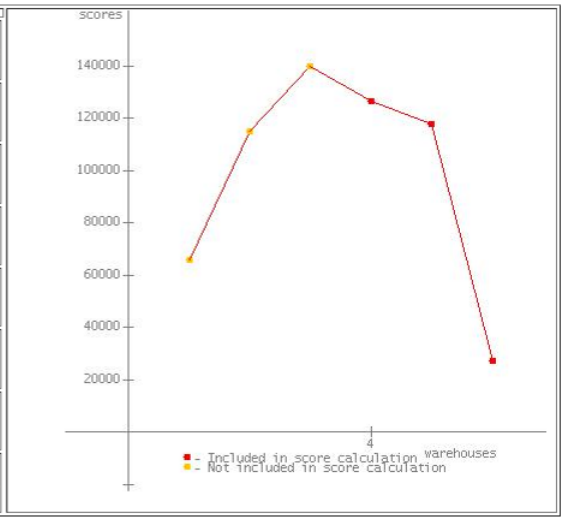

Figura 10: SPECjbb en LXC

A continuación se muestra una tabla para comparar este resultado con el nativo:

Warehouses	BOPS(container)	BOPS(nativo)	% BOPS (container/nativo)
1	65919	66371	99.32
2	115023	132187	87.02
3	139986	166667	83.99
4	126697	170513	74.30
5	117716	167705	70.19
6	27178	38000	71.52
Total(4+5+6)	271591	376218	72.19
Total	592519	741443	79.91

Tabla 5: Comparación JBB entre nativo y LXC

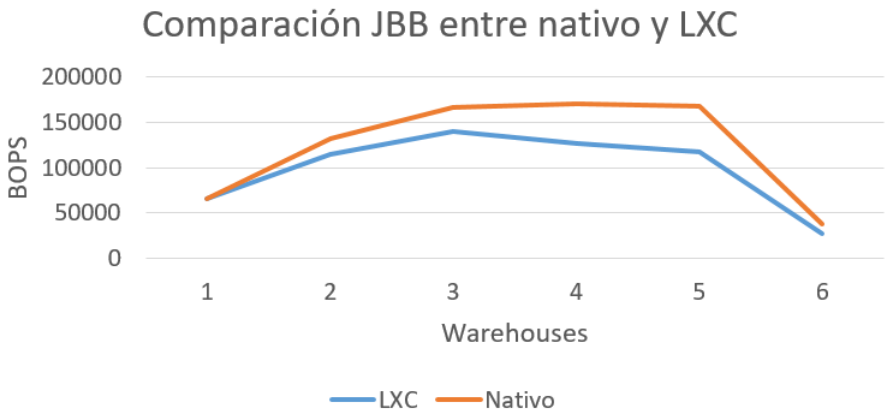

Figura 11: Comparación JBB entre nativo y LXC

Como se puede ver en la tabla y la gráfica, se consigue más de un 75 % del rendimiento en los tres primeros casos, pero después desciende y se queda entre el 70 y el 75 %. Esto se debe a que el contenedor tiene asignadas cuatro CPUs, pero una la comparte con el clon, por lo que el pico de rendimiento llega para n=3 *warehouses*. Sin embargo, en la ejecución en nativo se utilizan las cuatro CPUs exclusivamente para ello, por lo que el pico llega para n=4 *warehouses*.

Si tenemos en cuenta los tres últimos casos, el global reporta un rendimiento del 72.19 % en comparación con el nativo. Sin embargo, si tenemos en cuenta todos los casos, el rendimiento es de 79.91 %.

También se pregunta cómo se puede imitar la política de planificación en LXC; tras buscar en internet se han encontrado los siguientes pasos:

1. Habilitar *CAP_SYS_NICE* en el contenedor.

2. Esta capacidad permite la llamada *sched_setsheculer()* que es la necesaria para poner *SCHED_RR*.

3. Añadir en el archivo de configuración la llamada *lxc.cap.keep=sys_nice*.

4. Las aplicaciones tienen que tener también *CAP_SYS_NICE* o ser ejecutadas desde *root*, en cuyo caso ya lo tienen.

Otra opción más sencilla es jugar con las prioridades mediante la modificación de las propiedades de los *cgroups*.

4.2. Tarea 2

Con el fin de hacer más rápida la configuración de los aspectos básicos de los contenedores, se utiliza un *Dockerfile* para la creación de estos. Este obtiene la última versión estable de debian, y configura el contenedor con el SPECweb, descargando el *benchmark* y descomprimiendolo. Además, descarga e instala la versión de Java necesaria para su ejecución, y establece las variables de entorno.

Una vez que el contenedor está listo, se ejecuta el SPECweb varias veces, con distinto número de clientes (100, 145 y 150) para comparar los resultados con la ejecución nativa.

Después, se crean dos containers más a partir del *dockerfile*, con el objetivo de ejecutar el SPECweb de forma distribuida, teniendo en un container los clientes, en otro el *BeSim* y en el tercero el *Web Server*. Para que sea posible la comunicación entre ellos, se mapean los puertos (22, 80, 81, 1099) del contenedor a los de nuestra máquina cuando se crea, haciendo posible la comunicación. Además, en el fichero de configuración (*Test.config*) del *benchmark*, establecemos las IPs de cada container. Dado que no se ha conseguido que funcione correctamente este último caso, utilizaremos los resultados obtenidos al ejecutarlo en un solo container para compararlos con los resultados de la ejecución nativa.

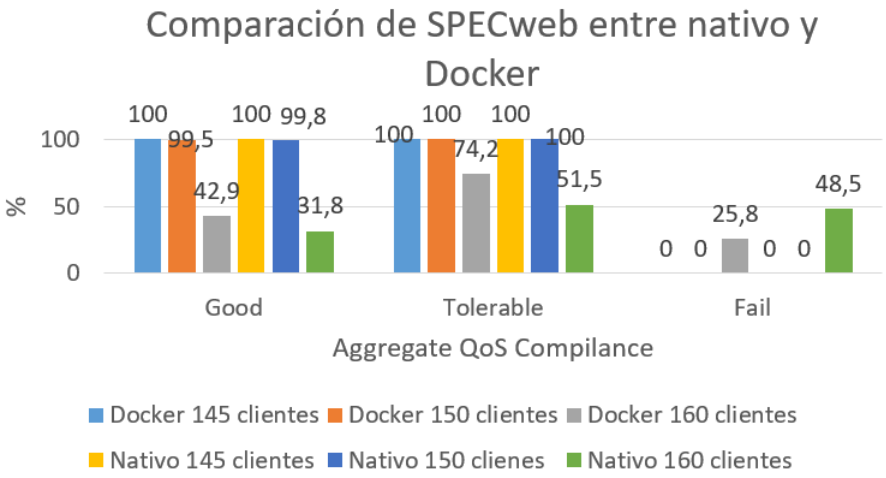

Figura 12: Comparación SPECweb entre nativo y Docker

Al observar los datos se ve que para 145 clientes se comportan de forma similar, aunque *Docker* tiene algunos fallos. Por otro lado, cuando se aumenta el número de clientes a 150, *Docker* empieza a bajar en el porcentaje de *Good*. Por último, cuando comprobamos 160, en los que ambos tienen una caída, la caída de *Docker* es menor que la de nativo. Sin embargo esto tampoco indica nada, porque una vez que empieza a fallar tampoco podemos garantizar que *Docker* siempre vaya a caer más o no.

Por último, se ha creado una cuenta de *Dockerhub* y un repositorio. Se ha añadido un *tag* (*docker tag*) y se ha subido el contenedor creado al repositorio.

4.3. Tarea 3

En *Google Cloud Platform*, se crea una máquina virtual de tipo *g1-small*, con una CPU y 1,7 GB de memoria. Se descarga *Docker* y el contenedor que estaba subido en *Dockerhub*, y ejecutamos de nuevo el SPECweb. Como ya se dijo en la tarea anterior, sólo se ha conseguido hacer funcionar el SPECweb en un contenedor, por lo que se ejecutará también en uno en *Google Cloud Platform*, en lugar de en tres.

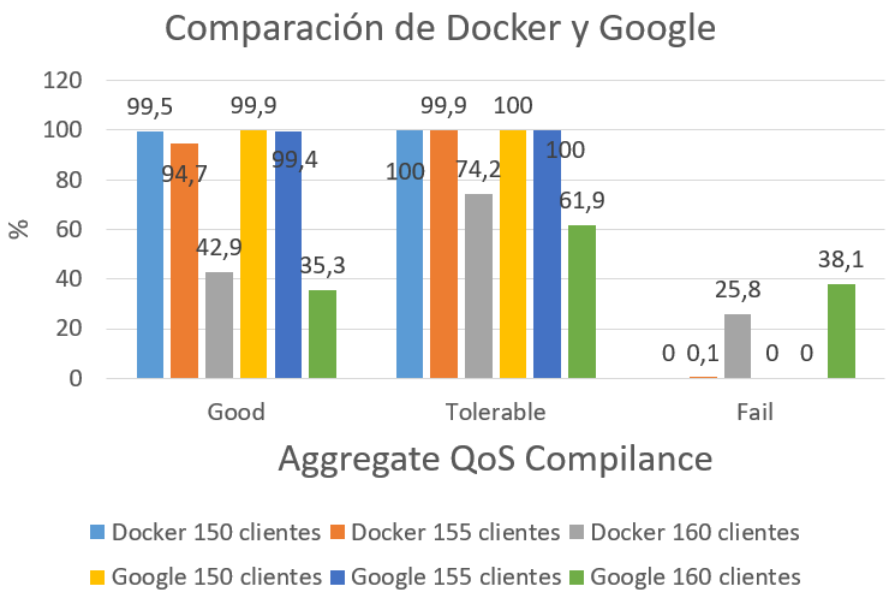

Figura 13: Comparación *Docker* y *Google*

Como se aprecia, con 150 clientes el comportamiento de ambos es igual. Al subir a 155 clientes, *Docker* empieza a tener un comportamiento peor a *Google Cloud*, pero al pasar a 160, el *Google Cloud* el que tiene un peor comportamiento, teniendo una caída más pronunciada que *Docker*. Esto puede deberse a la máquina que hemos elegido en Google, que es peor que la que tenemos en el laboratorio, puesto que tiene una CPU, y el rendimiento que se pierde al tener una VM y dentro un contenedor es menor en el laboratorio.

5. Herramientas Utilizadas

A lo largo de estas prácticas se han usado varias herramientas, entre las que destacan:

- Drive: usado para tomar notas en común de los pasos realizados, problemas, conclusiones,...
- GitHub: se ha creado un repositorio para subir archivos de configuración, con el fin de poder reutilizarlos si es necesario, y resultados de las pruebas llevadas a cabo.
  El enlace a dicho repositorio es: `https://github.com/rebecabarcena/SVS`
- Dockerhub: utilizado para subir los contenedores de la última práctica.
  El enlace al repositorio en Dockerhub es: `https://hub.docker.com/r/virtualizacionjr/svs/`