Estudio de la portabilidad de la JVM: ZERO/SHARK con LLVM 12

JVM portability case study: ZERO/SHARK with LLVM 12

Trabajo de fin de Grado en Ingeniería Informática

Facultad de Informática
Universidad Complutense de Madrid

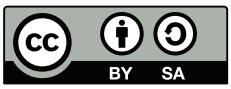
AUTOR
Mario Daniel Gallardo Cruzado
DIRECTOR
José Luis Sierra Rodríguez

2020-2021

Documento maquetado con TEXiS v.1.0., TEXIS se atribuye las siguientes licencias:

El codigo fuente correspondiente al paquete `TeXiS` se distribuye bajo la *LaTeX Project Public License* (Licencia Pública del Proyecto LaTeX).

Los ficheros y scripts de apoyo para la generación del documento, presentan licencia GPLv3.



El el resto del documento se distribuye bajo la licencia Creative Commons (CC-BY-SA).

Índice

Resumen

En este trabajo realizamos un estudio sobre la virtualización, teoría, características y ejemplos, para posteriormente abordar el caso concreto de la *jvm* y su portabilidad.

El trabajo culmina resucitando un proyecto de código abierto llamado *Shark*, que aborda del problema de la portabilidad de la *jvm* a través del marco *LLVM* de construcción de compiladores. Para ello, modernizamos y adaptamos el proyecto para que utilice la última versión de *LLVM* y la última versión de la *jvm*. Finalmente verificamos esta *jvm* con tests de estrés, y la ejecución de algunos programas de prueba.

Palabras clave— jvm, jit, zero, shark, llvm, compilación, openjdk

Abstract

In this work we study the topic of Virtualization, the theory behind it, characteristics and examples, to later address the specific case of the jvm and its portability.

The work culminates by resurrecting an open-source project called Shark, which addresses the jvm portability problem through the LLVM compiler-construction framework. For this purpose, we modernize and adapte Shark to use the latest version of LLVM and the latest version of the jvm. Finally we verify this jvm with stress tests, and the execution of some test programs.

Palabras clave— jvm, jit, zero, shark, llvm, compilación, openjdk

Capítulo 1

Introducción

La *jvm* (*Java Virtual Machine*) es una máquina virtual que es capaz de ejecutar múltiples lenguajes de programación, y está diseñada para funcionar en una amplia variedad de dispositivos, arquitecturas y sistemas operativos. El principal lenguaje de programación soportado por la *jvm* es *Java*, un lenguaje orientado a objetos e imperativo, que compila al *bytecode* de la *jvm* como código intermedio, y que permite la portabilidad entre sistemas operativos que soportan una implementación de la *jvm*.

En realidad existen múltiples implementaciones de la *jvm* [8], y la máquina en sí ha pasado por muchas versiones. La última versión de la *jvm* da soporte a la última versión de *Java*, y ambas comparten la versión 17. La variante de código abierto de la *jvm* es el proyecto *OpenJDK*, un proyecto que se encuentra bajo la licencia GPLv2. *OpenJDK* es objeto de estudio de una amplia variedad de universidades y desarrolladores independientes, y ha impulsado en gran medida el desarrollo de la *jvm*. El proyecto cuenta entre sus patrocinadores con Oracle, la principal compañía impulsora del lenguaje *Java*.

Resulta pues atractivo aprovechar el potencial de la *jvm* en nuevas plataformas, no sólo por el potencial de ejecutar programas portables, sino también como caso de estudio, ya que dicho potencial es enriquecedor para cualquier programador. La *jvm* es un claro ejemplo del estado del arte en el área de las máquinas virtuales a nivel de proceso, es decir máquinas virtuales que proporcionan un formato de código intermedio o *bytecode* al que los programas escritos en lenguajes de alto nivel pueden traducirse, y que luego la máquina es capaz de ejecutar, ya sea mediante interpretación, o mediante compilación.

La interpretación es un sistema de ejecución muy conocido y usado en, por ejemplo, navegadores web, clientes de terminal como *bash*, o incluso videojuegos. Sin embargo la interpretación de código no destaca precisamente por su eficiencia, llegando incluso a ser entre 10 y 30 veces menos eficiente que la ejecución de código nativo.

Esto conlleva a desarrollar otras alternativas como la compilación *Just In Time (JIT)*. La *jvm* utiliza esta técnica para compilar los métodos que más convienen para incrementar el rendimiento, y es la técnica que se abarca en este proyecto mediante el uso de una biblioteca llamada *LLVM*.

Introduction

The jvm (Java Virtual Machine) is a virtual machine that is capable of running multiple programming languages, and is designed to run on a wide variety of devices, architectures and operating systems. The main programming language supported by the jvm is Java, an object-oriented, imperative language, which compiles to the jvm bytecode as intermediate code, and allows portability between operating systems that support a jvm implementation.

There are actually multiple implementations of the jvm [8], and the machine itself has gone through many versions. The latest version of jvm supports the latest version of Java, and both share version 17. The open source variant of the jvm is the OpenJDK project, a project that is licensed under the GPLv2. OpenJDK is the subject of study by a wide variety of universities and independent developers, and has greatly driven the development of the jvm. The project's sponsors include Oracle, the main company behind the Java language.

It is therefore attractive to take advantage of the potential of the jvm in new platformforms, not only for the potential of running portable programs, but also as a case study, since such potential is enriching for any programmer. The jvm is a clear example of the state of the art in the area of process-level virtual machines, i.e. virtual machines that provide an intermediate code format or bytecode into which programs written in high-level languages can be translated, and which the machine is then able to execute, either by interpretation or by compilation.

Interpretation is a well-known execution system used in, for example, web browsers, terminal clients such as bash, or even video games. However, code interpretation does not exactly stand out for its efficiency, being between 10 and 30 times less efficient than native code execution.

This leads to the development of alternatives such as Just InTime Compilation (JIT). Jvm uses this technique to compile the most suitable methods to increase performance, and it is the technique covered in this project through the use of a library called LLVM.

1.1. Presentación del problema

OpenJDK 17 soporta las siguientes plataformas[1]: Linux x86_x64, Linux aarch64, Mac OS X x86_64, Windows x86_64. Portar el *OpenJDK* a otras plataformas no es un problema trivial, debido a las fuertes dependencias de sus distintos componentes de cada plataforma. Esto es especialmente cierto para la implementación de la *jvm*.

Efectivamente, la *jvm* tiene múltiples dependencias de la plataforma física concreta. Una de estas dependencias se debe al componente *JIT*, ya que dicho componente debe generar código objeto para la plataforma concreta. Por tanto, cuando se porta la *jvm* a otra plataforma, puede ser necesario re-implementar completamente dicho componente, lo que puede suponer un esfuerzo considerable. Por tanto, es interesante encontrar mecanismos que faciliten dicha portabilidad. Este proyecto se centra en estos mecanismos.

1.1 Problem Introduction

OpenJDK17 supports the following platforms[1]: Linux x86_x64, Linux aarch64, Mac OS X x86_64, Windows x86_64. Porting the OpenJDK to other platforms is not a trivial problem, due to the strong dependencies of its various components on each platform. This is especially true for the jvm implementation.

Indeed, jvm has multiple dependencies on the particular physical platform. One of these dependencies is due to the JIT component, since that component must generate object code for the concrete platform.Therefore, when porting the jvm to another platform, it may be necessary to completely re-implement that component, which can involve considerable effort. Therefore, it is interesting to find mechanisms that facilitate such portability. This project focuses on such mechanisms.

1.2. Objetivos del proyecto

Los objetivos generales de este proyecto son los siguientes:

1. Analizar el dominio de la virtualización y de las máquinas virtuales, con énfasis en la *jvm*.
2. Abordar el problema de la portabilidad de la *jvm*, y, en particular, del componente *JIT* de la misma.
3. Validar y evaluar los mecanismos propuestos para facilitar dicha portabilidad.

[1] https://wiki.openjdk.java.net/display/Build/Supported+Build+Platforms

Como objetivo complementario se persigue también aprovechar y ampliar los conocimientos y competencias adquiridos en las siguientes asignaturas de la carrera: Procesadores de lenguajes (PL), Sistemas operativos (SO), Ampliación de sistemas operativos (ASOR), y Tecnología de la programación (TP).

1.2 Main Objectives

The general objectives of this project are the following:

1. to analyze the domain of virtualization and virtual machines,with emphasis on the jvm.
2. To address the problem of jvm portability and, in particular, of the JIT component of the same.
3. Validate and evaluate the proposed mechanisms to facilitate such portability.

As a complementary objective, the aim is also to take advantage of, and extend, the knowledge and skills acquired in the following subjects of the degree: Language Processors (PL), Operating Systems (OS), Operating Systems Extension (ASOR), and Programming Technology (TP).

1.3. Desarrollo del Proyecto

Durante el periodo de preacuerdo de TFG y contando con la asignatura Procesadores de lenguajes realizada, se presentó al profesor de la asignatura, y actual director de este TFG, la idea de portar la máquina virtual de *Java* a otra plataforma. La idea original consistía en portar la *jvm* a una consola de videojuegos o alguna plataforma nueva como podría ser la *Raspeberry Pi*. Sin embargo, ya existían versiones de la *jvm* para muchas de estas plataformas, lo que hacía menos atractiva la idea como TFG. Por su parte el profesor hizo notar la posibilidad de que, por la complejidad y el límite del tiempo requerido, la realización del proyecto podría no ser factible al menos en su totalidad. La alternativa propuesta consistía en reducir la cobertura del proyecto acotándola al componente *JIT* de la *jvm*, haciendo notar además lo beneficioso que sería LLVM con respecto a la portabilidad. Esta propuesta fue aceptada por ambas partes, y ya que en un principio tendría carácter exploratorio, el proyecto tomó por título: *Portabilidad de Máquinas Virtuales: Caso de estudio de la JVM.*

Para orquestar el trabajo del proyecto se optó por SCRUM, una metodología de desarrollo ágil que permite gran flexibilidad y equipos reducidos. Por la situación de la pandemia del COVID-19, las reuniones o *scrum meetings* se realizaron por

videollamadas, normalmente reuniones de 60 minutos cada 2 semanas. La comunicación diaria se llevó a cabo mediante email, y el rol de *scrum master* fue asumido por el profesor, influyendo en la realización de cada *sprint* de desarrollo, ya fuera de documentación o programación. Asimismo, aunque no se tenía una hoja de ruta definida a priori, las etapas fueron surgiendo dinámicamente hasta completar el desarrollo del proyecto. Con estas consideraciones, podemos dividir el desarrollo en las siguientes etapas:

1. La primera etapa fue el estudio del campo de la virtualización, máquinas virtuales, y *jvm*. Para ello resultó de gran utilidad el libro *Virtual Machines*, [1], en el que se aborda todo lo relacionado a la virtualización y máquinas virtuales. También sirvió como referencia el libro *Java Virtual Machine Specification*, [3], en el que se detalla a nivel técnico la especificación de todas las partes que tiene la *jvm*. Esta etapa sentó las bases de los contenidos que se discuten en el Capítulo 2 de la memoria.

2. La siguiente etapa consistió en la profundización en los aspectos específicos del desarrollo. La elección de una estrategia sobre la que abordar la materialización del proyecto, empezó por tomar la decisión de implementar una máquina virtual reducida o utilizar una existente. A este respecto:

   i) El desarrollo de un intérprete para una máquina virtual reducida resulta relativamente sencillo y se podría tomar como punto de inicio para abordar el proyecto. Sin embargo, debido a la complejidad de la *jvm*, pensamos que la mayor parte del proyecto se invertiría en el desarrollo de un emulador para un subconjunto de la *jvm*, quedando reducido los aspectos de portabilidad, en el mejor de los casos, a un plano secundario.

   ii) Por tanto, seguidamente estudiamos la posibilidad de desarrollar el trabajo en el contexto de un proyecto real de código abierto ya existente. A este respecto consideramos la elección de *OpenJDK* como el marco más conveniente para abordar proyectos exploratorios, ya que la documentación es mayor en comparación a otras alternativas (*véase Sección 2.3*), el código suele estar anotado y, en caso de necesitar de más información sobre los *commits*, siempre se puede recurrir a las discusiones que se llevan a cabo en las listas de correo. Además, los programadores del *OpenJDK* suelen publicar entradas de blogs o dar conferencias como es el caso de FOSDEM[2]. Todos estos factores nos hicieron decantarnos por desarrollar sobre la *OpenJDK*.

3. A continuación se analizó cómo se había abordado el problema de la portabilidad en *OpenJDK*. Se descubrió que, de hecho, dicho problema se había abordado ya parcialmente usando LLVM. La idea de la portabilidad no es un tema nuevo dentro del *OpenJDK*. Efectivamente, en este contexto se han desarrollado iniciativas como ZERO, que minimiza el uso de ensamblador en la *jvm* para facilitar la portabilidad. A estos efectos el *JIT* quedaba fuera de

[2]https://fosdem.org/2021/archives/

ZERO, reduciéndose esta alternativa a un intérprete convencional. Para remediar este problema surgió *Shark*. con la finalidad de añadir a ZERO un *JIT* portable mediante el uso de LLVM. Estos dos proyectos (ZERO y *Shark*) han funcionado juntos hasta la versión 9 de la *jvm*, versión en la que se prescindió de *Shark*. Cuando fue discontinuado, *Shark* utilizaba la versión 3.4 de LLVM, una versión que ha quedado obsoleta (la versión actual de LLVM es la 12).

4. Aunque haya sido discontinuada del flujo principal de desarrollo de *OpenJDK*, creemos que la idea que *Shark* intentó realizar sigue siendo totalmente válida en la actualidad. Es por ello que se tomó la decisión de "revivir" *Shark* desde su última versión operativa conocida, la versión presente en la distribución openjdk9-b06 de OpenJDK, que hacía uso de LLVM3.4, e integrarlo en las versiones más actuales del *OpenJDK*, haciendo uso de la versión más actual de LLVM: es decir, integrar *Shark* en la distribución openjdk 17 del *OpenJDK*, modernizándolo para utilizar LLVM 12. Como resultado, conseguimos delimitar claramente el alcance del proyecto exploratorio planteado, delimitación que queda patente en el título definitivo escogido para el mismo: *Estudio de la portabilidad de la JVM:ZERO/SHARK con LLVM 12*. Para ello:

   i) Primero se abordó el estudio de la versión funcional de *Shark* y la operación conjunta con ZERO, se crearon algunos programas de prueba, y se generó documentación con *Doxygen*. Además, se logro visualizar la interacción de ZERO con *Shark* mediante *FlameGraphs* y un agente de la *jvm* (*véase Capítulo 4*). Se intentaron también otras iniciativas para capturar las llamadas a *Shark* desde la *jvm*, como la visualización con *ftrace* y *TraceCompass*[3] pero fueron dejándose de lado debido a que no eran consideradas vitales para el desarrollo del proyecto.

   ii) Una vez familiarizados con el código de *Shark*, lo siguiente consistió en retocar las piezas que llevaban a cabo la configuración de compilación, así como modificar el código fuente de la *jvm* para "revivir" a *Shark*. Para lo primero, abordamos la modificación de los ficheros *Make* del proyecto ZERO (*véase Capítulo 4*). Para lo segundo utilizamos la nueva API *JIT* de LLVM para reemplazar la obsoleta *MCJIT*. Esta nueva API se denomina ORC JIT (*On Request Compilation*), ya que soporta, entre otras funcionalidades, compilación bajo demanda (*véase Sección 2.6*), algo que ya hace la *jvm* con su política de compilación (`compilationPolicy.hpp`), modernizando el código de *Shark* para volver a hacerlo funcionar.

   iii) Finalmente, evaluamos la modernización realizada, para identificar los problemas que presenta, y poder definir el trabajo a realizar en futuros desarrollos.

De esta forma, podemos considerar ambos cuatrimestres como puntos de consecución de dos hitos independientes en el desarrollo del proyecto:

[3] https://www.eclipse.org/tracecompass/

1. El primer cuatrimestre fue, fundamentalmente, de naturaleza exploratoria, y dio como resultado la definición clara del alcance final del proyecto. Efectivamente, durante este cuatrimestre se abordaron las tres primeras etapas descritas anteriormente. Tras llevar a cabo el estudio de las máquinas virtuales, cuales son los tipos y las técnicas de virtualización, las técnicas de optimización, y además como encajan los lenguajes de alto nivel dentro de la virtualización, decidimos focalizar nuestros esfuerzos en *OpenJDK*, descubrimos los proyectos ZERO/*Shark*, y planteamos la modernización de *Shark*. Como requisito previo para lograr dicha modernización vimos que era necesario contar con una distribución funcional que soportara ZERO/*Shark*. Como resultado, al final del cuatrimestre ya contábamos con dicha distribución.

2. De esta forma, el segundo cuatrimestre ha consistido en la manipulación del código del *OpenJDK*. La familiarización con el código se ha realizado con la ayuda de la herramienta *Doxygen* para la visualización de las relaciones entre las rutinas del código del *OpenJDK*. Como resultado, al final de este segundo cuatrimestre se ha conseguido modernizar el proyecto *Shark* adaptado a las últimas versiones del *OpenJDK* y LLVM, y se ha llevado a cabo una evaluación de dicha modernización.

El código del proyecto se encuentra disponible en el siguiente repositorio *github*: https://github.com/danisilver/jdk/

1.3 Project Development

During the end-of-degree project pre-agreement period and with the subject Language Processors completed, the idea of porting the Java virtual machine to another platform was presented to the professor of the subject, and current supervisor of this project. The original idea was to port the jvm to a video game console or some new platform such as the Raspeberry Pi. However, versions of the jvm already existed for many of these platforms, which made the idea less attractive as an end-of-degree project. For his part, the professor noted the possibility that, due to the complexity and time constraints required, the project might not be feasible, at least in its entirety. The alternative proposed was to reduce the coverage of the project by limiting it to the JIT component of the jvm, noting also how beneficial LLVM would be with respect to portability. This proposal was accepted by both parties, and since it would initially be exploratory, the project took the title: *Virtual Machine Portability: JVM Case Study.*

SCRUM, an agile development methodology that allows great flexibility and small teams, was chosen to orchestrate the project work. Due to the COVID-19 pandemic situation, the meetings or scrum meetings were conducted by video calls, usually 60-minute meetings every 2 weeks. Daily communication was carried out by email, and the role of scrum master was assumed by the teacher, influencing

the execution of each development sprint, whether it was documentation or programming, and although there was no a priori defined roadmap, the stages emerged dynamically until the project development was completed. With these considerations, we can divide the development into the following stages:

1. The first stage was the study of the field of virtualization, virtual machines, and jvm. For this purpose, the book *Virtual Machines*, Smith and Nair (2005), which deals with everything related to virtualization and virtual machines, was very useful. The book *Java Virtual Machine Specification*, Lindholm et al. (2014), which details at a technical level the specification of all the parts of the jvm, also served as a reference. This stage laid the foundation for the contents discussed in Chapter 2 of the report.

2. The next stage consisted of delving deeper into the specific aspects of the development. The choice of a strategy on which to approach the materialization of the project, started by making the decision to implement a scaled-down virtual machine or to use an existing one. In this regard:

   i) The development of an interpreter for a reduced virtual machine is relatively straightforward and could be taken as a starting point for approaching the project. However, due to the complexity of the jvm, we believe that most of the project would be spent on developing an emulator for a subset of the jvm, with portability issues being reduced to a secondary plane at best.

   ii) Therefore, we next considered the possibility of developing the work in the context of a real existing open source project. In this respect we considered the choice of OpenJDK as the most convenient framework to approach exploratory projects, since the documentation is more extensive compared to other alternatives (see Section 2.3), the code is usually annotated and, in case of needing more information about commits, one can always resort to the discussions on the mailing lists. In addition, OpenJDK programmers often publish blog posts or give lectures as in the case of FOSDEM[4]. All these factors made us decide to develop on the OpenJDK.

3. We then analyzed how the portability problem had been addressed in OpenJDK. It was found that, in fact, this problem had already been partially addressed using LLVM. The idea of portability is not a new issue within the OpenJDK. Indeed, initiatives such as ZERO, which minimizes the use of assembler in the jvm to facilitate portability, have been developed in this context. For this purpose, JIT was left out of ZERO, reducing this alternative to a conventional interpreter. To remedy this problem, Shark was created to add a portable JIT to ZERO using LLVM. These two projects (ZERO and Shark) worked together until version 9 of the jvm, when Shark was discontinued, Shark was using LLVM version 3.4, a version that is now obsolete (the current version of LLVM is 12).

[4]https://fosdem.org/2021/archives/

4. Although it has been discontinued from the main OpenJDK development stream, we believe that the idea that Shark tried to realize is still fully valid today. That is why the decision was made to "revive" Shark from its last known working version, the version present in the openjdk9-b06 distribution of OpenJDK, which made use of LLVM3.4, and to integrate it into the most current versions of OpenJDK, making use of the most current version of LLVM: that is, to integrate Shark into the openjdk 17 distribution of OpenJDK, modernizing it to use LLVM 12. As a result, we managed to clearly delimit the scope of the proposed exploratory project, a delimitation that is evident in the final title chosen for it: *Study of the portability of JVM:ZERO/SHARK with LLVM 12*.

   i) First, the study of the functional version of Shark and the joint operation with ZERO was addressed, some test programs were created, and documentation was generated with Doxygen. In addition, the interaction of ZERO with Shark was visualized using FlameGraphs and a jvm agent (see Chapter 4). Other initiatives to capture the calls to Shark from the jvm, such as visualization with ftrace and TraceCompass 3, were also tried but were dropped because they were not considered vital for the development of the project.

   ii) Once we were familiar with the Shark code, the next step was to tinker with the parts that carried out the decompilation configuration, as well as to modify the jvm source code to revive Shark. For the former, we tackled the modification of the Make files of the ZERO project (see Chapter 4). For the second, we used the new LLVM JIT API to replace the obsoleteMCJIT. This new API is called ORC JIT (OnRequest Compilation), since it supports, among other functionalities, on-demand compilation (see Section 2.6), something that the jvm already does with its compilation policy (compilationPolicy.hpp), modernizing the Shark code to make it work again.

   iii) Finally, we evaluated the modernization carried out, in order to identify the problems it presents, and to be able to define the work to be done in future developments.

In this way, we can consider both quarters as points of achievement of two independent milestones in the development of the project:

1. The first four-month period was essentially exploratory in nature and resulted in a clear definition of the final scope of the project. Indeed, the first three stages described above were addressed during this term. After studying virtual machines, the types and techniques of virtualization, optimization techniques, and how high-level languages fit into virtualization, we decided to focus our efforts on OpenJDK, discovered the ZERO/Shark projects, and proposed the modernization of Shark. As a prerequisite to achieve this modernization, we saw that it was necessary to have a functional distribution that supported

ZERO/Shark. As a result, by the end of the quarter we had such a distribution in place.

2. Thus, the second quarter consisted of manipulating the OpenJDK code. The familiarization with the code was done with the help of the Doxygen tool for the visualization of the relationships between the routines of the OpenJDK code. As a result, at the end of the second quarter, the Shark project had been modernized to the latest versions of OpenJDK and LLVM, and an evaluation of this modernization had been carried out.

The project code is available at the following *github* repository:
https://github.com/danisilver/jdk/

1.4. Estructura de la memoria

El documento está estructurado en 6 capítulos que abarcan los diferentes aspectos del proyecto, como se muestra a continuación:

- En el **Capítulo 2** presentamos un estudio general de las máquinas virtuales, de la *jvm* y los aspectos de portabilidad asociados, del marco LLVM, y del soporte *JIT* en dicho marco .
- En el **Capítulo 3** describimos el problema de modernización de *Shark* abordado en este proyecto.
- En el **Capítulo 4** describimos cómo hemos llevado a cabo dicha modernización, para actualizar *Shark* a las versiones más recientes del *OpenJDK*.
- En el **Capítulo 5** evaluamos la modernización, identificando los problemas que presenta.
- Finalmente en el **Capítulo 6** presentamos las conclusiones obtenidas, y analizamos las posibles extensiones o mejoras que se pueden llevar a cabo en esta línea en el futuro.

1.4 Contents Structure

The document is structured in 6 chapters covering the different aspects of the project, as shown below:

- In Chapter 2 we present an overview of virtual machines, of jvm and associated portability aspects, of theLLVM framework, and of JIT support in that framework .

- In Chapter 3 we describe the Shark modernization problem addressed in this project.
- In Chapter 4 we describe how we have carried out the modernization, to upgrade Shark to the latest versions of the OpenJDK.
- In Chapter 5 we evaluate the modernization, identifying the problems it presents.
- Finally in Chapter 6 we present our conclusions, and discuss possible extensions or improvements that can be carried out along these lines in the future.

Capítulo 2

Estudio del dominio

En este capítulo se resumen los aspectos más relevantes estudiados en la primera etapa del proyecto. La **Sección 2.1** hace un estudio previo que pone en relieve la posición de la *jvm* dentro de esta área. La **Sección 2.2** ahonda en la estructura interna de la *jvm*. La **Sección 2.3** muestra algunas implementaciones relevantes de la *jvm*. La **Sección 2.4** discute la cuestión de la portabilidad de la *jvm* a otra plataforma. La **Sección 2.5** describe la estructura de ZERO y *Shark*. La **Sección 2.6** presenta el marco LLVM, e introduce algunos términos propios de este marco. Por útimo, la **Sección 2.7** introduce la API *JIT* de LLVM.

2.1. Virtualización y máquinas virtuales

Frecuentemente escuchamos a personas referirse a aspectos tales como "disco duro virtual", "impresora virtual", "red virtual", etc. Todos estos aspectos tienen que ver con el concepto de *virtualización*. La virtualización tiene sus inicios en los ordenadores tipo *mainframe*, unidades de procesamiento a gran escala, en las que se requería compartir los recursos del sistema entre gran cantidad de usuarios. En la actualidad la virtualización no se restringe al *hardware*, ya que los servicios en la nube también requieren de la virtualización del *software*.

Para facilitar la virtualización, hacemos uso de los niveles de abstracción, es decir las capas independientes sobre la que están diseñadas las partes del sistema. Así por ejemplo, un procesador diseñado por Intel y otro diseñado por AMD, que tienen el mismo repertorio de instrucciones IA-32, son compatibles a nivel de ISA (*Instruction Set Architecture*), o un disco duro de una determinada marca es indistinguible, a nivel de sistema operativo, de otro hecho por otro fabricante, ya que dicho sistema operativo brinda la misma interfaz para ambos.

Podríamos considerar que en un sistema operativo, ya encontramos virtualización. Esto es debido a la multiprogramación o paralelismo simulado, ya que el reper-

torio de instrucciones utilizado por cada proceso es el mismo que el de la máquina real, y las peticiones de recursos se realizan a través de llamadas al sistema o *syscalls* y el resto de la *Application Binary Interface (ABI)* de la máquina real. Otro ejemplo famoso es el de *docker*, en el que la virtualización se da a través de *containers*, y en la que a cada recurso se le asigna un *namespace*.

La virtualización se puede ampliar al conjunto del *hardware* de un sistema, o al conjunto *hardware/software*, también denominado *Plataforma*. Esto es lo que se conoce como una *Máquina virtual (MV)*, Dependiendo del punto de vista, podemos distinguir dos tipos de máquina virtual:

- Desde el punto de vista de un proceso, en el que tenemos disponible el repertorio de instrucciones de nivel de usuario y disponemos del sistema operativo, la MV toma el nombre de máquina virtual a *nivel de proceso*.
- Desde el punto de vista del sistema, en el que tenemos acceso a todo el repertorio de instrucciones, a todo el *hardware*, y a procesos de múltiples usuarios, la MV se denomina una máquina virtual a *nivel de sistema*.

Así pues las MVs a nivel de proceso pueden hacer uso de la ISA a nivel de usuario y del sistema operativo para dar soporte a alguna de las partes que difieran de las de la máquina real. Esta capa de *software* recibe el nombre de *runtime*. En las MVs a nivel de sistema disponemos de todo el repertorio de instrucciones y tenemos que dar soporte a un sistema operativo en su totalidad. En este caso esta capa se denomina *VM monitor*, o *Hypervisor*.

Un posible uso de las MVs a nivel de proceso es el dar soporte a aplicaciones precompiladas para otro repertorio de instrucciones, pero que se ejecutan sobre el mismo sistema operativo y ABI. Así por ejemplo, tenemos a la capa de traducción *Rosetta* de *Apple* para dar soporte a aplicaciones compiladas para x86 en los nuevos procesadores ARM con el mismo sistema operativo de *Apple*, *MacOs*.

Otro método dentro de las MVs a nivel de proceso consiste en interpretar la ISA de la MV. En este caso hay que decodificar cada instrucción y traducirla en una o mas instrucciones equivalentes en la máquina real. Esto es ineficiente, pero existen varios métodos para mejorar el rendimiento. Además estas MVs no necesitan tener una implementación real. En este caso el repertorio de instrucciones es también virtual dando lugar a lo que se denomina *Virtual Instruction Set Architecture (VISA)*.

Un ejemplo de MV con repertorio de instrucciones virtual es la diseñada para el lenguaje de programación Pascal. La *máquina-p* de Pascal soporta una VISA denominada *código-p*, la cual es una ISA orientada a pila, es decir, elimina la dependencia de los registros y las direcciones absolutas, en favor de las instrucciones de tipo pila "push", "pop". De esta forma se evita el problema del agotamiento de los registros.

Otro ejemplo, más popular en la actualidad, es el de la ya citada máquina virtual de *Java* (*jvm*), que como ya hemos comentado en la introducción soporta una plé-

tora de lenguajes, siendo *Java* el principal, y que compila a un lenguaje intermedio denominado *bytecode*. *Java* además incluye otras características[5] más modernas que Pascal, adecuadas a las necesidades de los lenguajes de alto nivel (HLLs: *High Level Languages*).

Los recursos de estas MVs orientadas a HLLs suelen exponerse, además, como parte de la biblioteca estándar que acompaña al lenguaje, lo que permite por ejemplo, la capacidad de introspección, obtener información de las estructuras en tiempo de ejecución, la posibilidad de ejecutar código remoto (RMI) o ejecutar *Applets* (obsoletos), asegurar la confianza de las fuentes de código, firmar digitalmente los paquetes, cargar clases por medio de la red o del sistema de ficheros, ejecución de código nativo (como, por ejemplo, la *Java Native Interface (JNI)*), etc.

2.2. La máquina virtual de Java™

La *jvm* involucra tres elementos:

- El *ClassLoader* es el encargado de cargar los *classfiles* en la *jvm*. Los *classfiles* son archivos que contienen el código generado en *bytecode* para una clase, el *constant pool*, que son las constantes del programa, y las referencias a las demás clases de las que dicha clase hace uso.
- El *Área de datos* de la *jvm*, como su nombre indica, es el área donde se alojan todos los datos, manipulados por la máquina. Este área incluye los siguientes segmentos:
  - El *Heap*. En el *Heap* encontramos los objetos que se han ido creando en todos los hilos de ejecución. Cada vez que se hace un *new* se reserva espacio aquí para alojar una nueva instancia de una clase.
  - El *Method Area*. Este segmento es donde se guarda el código de los métodos, así como el *constant pool* en tiempo de ejecución.
  - El *Stack*. Este segmento es el espacio que sirve para almacenar las variables y los resultados de operar con los distintos valores. Cada vez que se invoca un método, se crea un *frame* y se apila en el *Stack* para el hilo apropiado.
- Por último, la *jvm* también incluye un *Execution Engine*. Este es el componente de la *jvm* encargado de ejecutar el código. Dicho componente incluye un intérprete, que ejecuta cada instrucción apuntada por el *Program Counter (PC)*. También puede incluir un compilador *JIT* que, como ya hemos indicado, se encarga de compilar el *bytecode* a código nativo para mejorar el rendimiento. Por último, incluye un recolector de basura que se encarga de reciclar el espacio del *Heap* que se puede liberar.

La especificación de la *jvm* permite la existencia de múltiples *ClassLoaders*. Esto permite una carga mas personalizada para aplicaciones específicas. El principal es el *Bootstrap Classloader* que provee un entorno de ejecución para una carga segura de los *classfiles*. Este *ClassLoader* se encuentra disponible desde *Java* mediante la clase `java.lang.ClassLoader`. El proceso de carga consta de las siguientes fases:

- *Carga*: sigue la estrategia especificada por el *ClassLoader* elegido para traer la clase a memoria.
- *Enlazado*: consiste en verificar la consistencia de los tipos, inicialización de las variables estáticas con valores por defecto, y la resolución de los símbolos de las clases externas.
- *Inicialización*: carga de las clases externas y la ejecución de los bloques estáticos de la clase cargada.

Obsérvese que, de esta forma, la carga de una clase puede desencadenar recursivamente la carga de las clases externas de las que esta depende. Dicha carga puede seguir, asimismo, distintas estrategias, dependiendo del *ClassLoader* utilizado (por ejemplo, cargar las clases externas de manera perezosa cuando se necesiten, o cargarlas impacientemente, justo antes de proceder con la inicialización).

La unión de la *jvm* con el conjunto de APIs de la biblioteca estándar de *Java* toma el nombre de *Java Runtime Environment (JRE)*.

Una vez disponibles el código y los segmento de datos de los *classfiles*, la *jvm* procede a la ejecución del código. Dicha ejecución puede proceder en modo "intérprete" o en modo *JIT*. Asimismo, el *ExecutionEngine* cuenta con varios hilos:

- *Main Thread*: hilo principal que ejecuta operaciones de coordinación.
- *GC Threads*: hilos destinados a la recolección de basura.
- *Compiler Threads*: hilos de compilación.

La mayoría del tiempo el hilo principal espera algún evento. Asimismo, para proceder con la recolección de basura espera a alcanzar algún *safepoint*, que es una instrucción que indica una posición segura donde llamar al recolector de basura. Para ello, antes de invocar al recolector, se espera a que todos los hilos de ejecución alcancen un *safepoint*.

La *jvm* soporta 5 niveles de ejecución:

- level 0 - interpreter
- level 1 - C1 with full optimization (no profiling)

- level 2 - C1 with invocation and backedge counters
- level 3 - C1 with full profiling (level 2 + MDO)
- level 4 - C2

Para entender estos niveles, debe tenerse en cuenta que la *jvm* utiliza dos compiladores: C1, que es el compilador llamado cliente, y C2, el compilador llamado servidor. C1 es un compilador orientado a programas de baja latencia como interfaces gráficas, en comparación a C2 que es un compilador orientado a optimizaciones de cara al servidor. Desde java 7 estos compiladores se usan conjuntamente, dando lugar a lo que se denomina compilación *tiered*. Mientras que C1 es rápido, C2 se beneficia de los datos recopilados por C1, para luego entrar en acción.

Los niveles 0, 2, 3 periódicamente notifican al *runtime* información relativa a los contadores de invocación de los métodos y los bucles (*backedge*). La frecuencia de las notificaciones cambia a cada nivel. Estas notificaciones se utilizan en la política de compilación. Dicha política consiste en comenzar la ejecución en el nivel 0. Luego se puede elegir entre la compilación al nivel 2 o 3, dependiendo de los siguientes factores:

- La longitud de la cola de C2. La observación es que el nivel 2 es generalmente 30 % mas rápido que el nivel 3. Por tanto se minimiza el tiempo que un método está en el nivel 3. Sólo se debe permanecer en el nivel 3 para recolectar suficiente información. Entonces, si la cola de C2 esta relativamente llena, es mejor ir primero al nivel 2.
- El nivel de llenado C1 sirve como filtro adicional, porque en el momento en el que un método está listo para ser compilado, este podría no ser necesario.

Cuando se ha conseguido suficiente información en el nivel 3, pasamos al nivel 4. Nuevamente la longitud de la cola de C2 se utiliza como indicador. También se puede conseguir información desde el nivel 0. Si la recolección de información termina antes antes de llegar al nivel 3, se puede pasar al nivel 4. Asimismo, después de la compilación C1 se tienen datos suficientes como el numero de bucles y bloques, así que se puede asumir que el código no necesita recompilación C2.

En cuanto al proceso de interpretación, existen dos[7] implementaciones del interprete. La elección de una de estas dos se efectúa al seleccionar la variante de la *jvm*, por lo que sólo uno se selecciona para ejecutar.

- *TemplateInterpreter*. Es el interprete por defecto y tiene segmentos de ensamblador por cada instrucción, que se almacenan en una tabla dinámica. Estos segmentos se inicializan al inicio de la *jvm*.

- *BytecodeInterpreter*. Este es el interprete que usa ZERO, y está implementado mediante operaciones en lenguaje de alto nivel en lugar de en ensamblador. La racionalidad es el soportar la portabilidad de ZERO.

Hasta la versión 11, la *jvm* usaba el recolector de basura G1. Desde la versión 12 de la *jvm* se introdujo el recolector de basura *Shenandoah*, un recolector con latencias muy bajas, que opera de manera concurrente. G1 utiliza pausas y es capaz de desfragmentar el espacio sólo durante la pausa. *Shenandoah* es capaz de hacer ambas cosas simultaneamente.

Para lograr esto utiliza lo que se conoce como *Brooks pointer*, un campo adicional en los objetos del *Heap*, un puntero a sí mismo. Esto facilita el movimiento de los objetos de manera concurrente, ya que cuando un objeto ha sido trasladado a una nueva posición, el puntero se queda apuntando a la nueva posición del objeto. De esta forma, con el desacoplo de la limpieza y la compactación se puede lograr la concurrencia.

Para otro material complementario a este capítulo se puede acudir a [5] o a la referencia oficial de la jvm [4].

2.3. Ejemplos de implementación de la JVM

A continuación describimos brevemente algunos ejemplos de implementación de la *jvm*:

- La implementación de código abierto de la *jvm* lleva el nombre de `Hotspot`, que heredó de los tiempo de *Sun*, la compañía que creó originalmente Java y que, posteriormente, fue absorbida por *Oracle* . Esta implementación era código propietario, pero finalmente se liberó el código fuente. *Oracle* ofrece además versiones cerradas con más características.
- `HaikuVM`, es una implementación para micro controladores como *atmega8* o *Arduino*, que soporta todos los *bytecodes* menos *invokedynamic*. Esta máquina funciona traduciendo cada *bytecode* a estructuras de datos e instrucciones en lenguaje C, para luego compilarlos a la plataforma destino. No soporta reflexión ni serialización. La última versión de java que soporta es la 1.8 .
- `Jikes RVM`[1], es una máquina virtual escrita completamente en *Java*, y que ejecuta *bytecode* sin necesidad de otra máquina virtual. Su objetivo principal es la investigación.
- `OpenJ9` es una máquina virtual de altas prestaciones, originalmente creada por IBM para acompañar a muchos de sus servicios, con los que ganaron mucho

[1] Jikes Research Virtual Machine

reconocimiento. En la actualidad forma parte de la *Eclipse Foundation* y es de código libre. El desarrollo de esta MV continúa activo en la actualidad.

- `GraalVM`, es una máquina virtual basada en el *OpenJDK* que soporta más modos de ejecución como la compilación AOT (*Ahead of Time*), y la interoperatibilidad entre varios lenguajes bajo un mismo proyecto multilenguaje, por lo que se la denomina una máquina virtual *políglota*. Esta máquina incluye un compilador *JIT* denominado *Graal*, que está implementado con Java mediante la interfaz *JVMCI*, también incluida en el *OpenJdk*.

Aparte de estos ejemplos, cabe destacar también la MV ZERO, que será descrita en la Sección 2.5 .

2.4. Portabilidad de la JVM

Las implementaciones de la *jvm* tienen fuertes dependencias con cada plataforma concreta, lo que puede dificultar su portabilidad. Estas dependencias pueden apreciarse a través de *OpenJDK*, ya que su estructura refleja los aspectos a abordar para portar la *jvm* a una nueva plataforma. Efectivamente, para ello necesitamos (ver Figura 2.1):

- Primero crear un subdirectorio dentro de `cpu` para dar soporte a nuestro repertorio de instrucciones. En este directorio se alojan las clases para generar código ensamblador de nuestra cpu. Hay que agregar ciertas instrucciones para añadir la funcionalidad de la *jvm* (estas instrucciones de relleno se llaman `VM stubs`). Estas instrucciones son, por ejemplo, saltos a ciertas partes de la *jvm* o del interprete, creación de arrays u objetos, y resto de funcionalidad de la *jvm*. También hay que definir otros aspectos de la arquitectura del procesador como los registros, formato de almacenamiento de datos (*endianess*) y el modelo de memoria.
- A continuación crear otro subdirectorio debajo de `os` para dar soporte a nuestro sistema operativo. Para dar ello tendremos que dar soporte a la creación de procesos, reserva de memoria y paginación, e hilos.
- Y, finalmente, crear otro subdirectorio dentro de `os_cpu` para las dependencias entre el sistema operativo con el repertorio. Aquí se encuentra la abstracción de las funcionalidades de la *jvm* en las que se hace uso del `os` y de la `cpu`.

Por tanto, portar la *jvm* a una nueva plataforma supone un esfuerzo sustancial, así como un profundo conocimiento de las características del *hardware* y del *sistema operativo* de la plataforma. Es interesante, por tanto, disponer de mecanismos que faciliten este proceso, independizando la implementación de plataformas concretas, y manteniendo, al mismo tiempo, la eficiencia. ZERO y *Shark* son un primer paso en esta dirección.

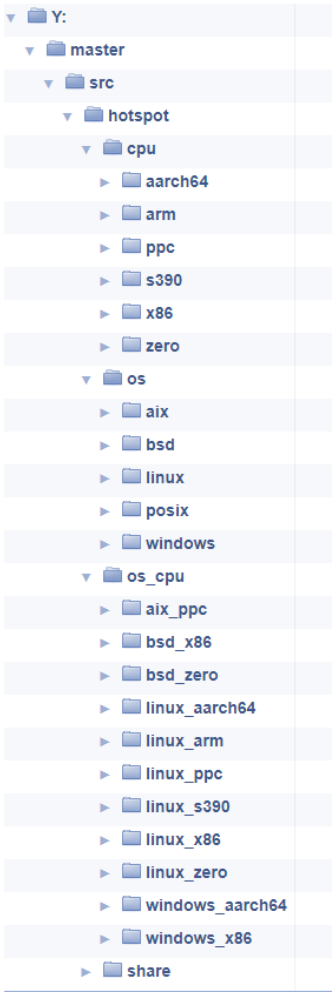


Figura 2.1: Estructura de directorios del jdk

2.5. ZERO y Shark

El proyecto *Zero-Assembler* es un proyecto del *OpenJDK*, cuya finalidad es facilitar la portabilidad de la *jvm*. Su objetivo es, como ya se ha indicado, proporcionar una implementación de la *jvm*, denominada ZERO, que no depende de código o ensamblador especifico a ninguna plataforma. Esto quiere decir que no incluye ninguno de los compiladores C1 o C2. El proyecto fue completado para *Linux* y *gcc* en marzo del 2008, siendo muy activo su desarrollo en la versión *IcedTea* de la *jvm*. Su integración en el *OpenJDK* se produjo en octubre del 2009. Posteriormente fue integrado en el openjdk6 b112, y ha seguido presente hasta la actualidad en la versión 17 del *OpenJDK*.

A continuación se resumen las principales clases de ZERO:

- `ZeroInterpreter`. Esta clase implementa el emulador de la *jvm*. Para ello,

utiliza a `BytecodeInterpreter` para ejecutar cada *bytecode* hasta finalizar la ejecución. Consta de un bucle principal `ZeroInterpreter::main_loop`. Cuando se vuelve de un método se invocan a los métodos de entrada al intérprete, con `ZeroInterpreter::normal_entry` y `ZeroInterpreter::native_entry` en caso de código nativo.

- `ZeroFrame`. Un *frame* es un lugar donde se guardan los resultados, se lanzan excepciones, y se enlazan los símbolos sin resolver. Se crea cada vez que se invoca un método. Los *frames* específicos del interprete de ZERO se representan mediante esta clase.
- `ZeroStack`. Esta clase representa la pila de ejecución, en la que se apilan las variables y los resultados intermedios de la ejecución de *bytecodes* de la *jvm*.
- `ZerEntry`. Esta clase representa entradas al código ejecutable. Hay dos tipos de entradas: `NormalEntryFunction` y `OSREntryFunc` para soportar *on-stack replacement*.

La principal desventaja de ZERO frente a otras implementaciones de la *jvm* es la eficiencia, ya que carece, como hemos indicado, de compiladores JIT. Para ello se propuso la creación de *Shark*, un proyecto experimental liderado por Gary Benson, que ha sido discontinuado y retirado de *OpenJDK*. *Shark* consiste en la sustitución de los compiladores C1/C2 por un JIT creado con la biblioteca LLVM. Esto hace que la portabilidad a otras plataformas se amplié a las plataformas soportadas por LLVM. La última versión funcional de *Shark* se encuentra en el openjdk9u64.

La interfaz de *Shark* consta de las siguientes clases:

- `SharkBlock`. Esta clase representa bloques de código dentro de un método. Cada bloque tiene un estado asociado.
- `SharkToplevelBlock`. Esta clase representa un `ciBlock`, es decir un bloque de código de la *jvm*. Este tipo de bloques analiza el *bytecode Java* y lo descompone en bloques del tipo `SharkBlock`.
- `SharkBuilder`. Esta clase sirve para construir un bloque de código a partir de su representación en *bytecode*. Extiende a la clase `IRBuilder`. Aquí están declarados los intrínsecos que *Shark* soporta, y que se expresan en la IR (*Intermediate Representation*) de LLVM.
- `SharkCompiler`. Es el compilador JIT de *Shark*. Hereda de `AbstractCompiler`. Tiene un `SharkContext` para código de *Shark* y otro para código JNI, que es envuelto en un `SharkNativeWrapper`.
- `SharkEntry`. Es una subclase de `ZeroEntry` que representa una entrada de código. Sobre-escribe otros métodos, tales como donde empieza y termina el tamaño del código generado. Además guarda un contexto y una instancia de `SharkFunction`.

- `SharkIntrinsics`. Esta clase contiene rutinas que apilan sobre la pila del bloque actual el resultado de funciones intrínsecas, como por ejemplo `System.currentTimeMillis()`.
- `SharkMemoryManager`. Clase no implementada pero que más adelante podría servir para administrar la reserva y liberación de memoria para el código compilado.
- `SharkRuntime`. Partes del *runtime* de la *jvm* de bajo nivel, como la creación de instancias de objetos, arrays o excepciones.
- `SharkFrame`. Es un *frame* con propósito idéntico a los *frames* de ZERO.
- `SharkStack`. Representa una pila análoga a la de ZERO, pero que maneja valores de LLVM. Tiene dos subclases: `SharkStackWithNormalFrame` para *frames* con *bitcode* de LLVM, y `SharkStackWithNativeFrame` para *frames* de código nativo.
- `SharkState`. Esta clase sirve para guardar el estado de un `SharkBlock`. Aquí hay también una pila de operandos llamada *xstack*.
- `SharkFunction`. Es la representación de un método en *bitcode* de LLVM antes de ser compilado.
- `SharkContext`. Es una subclase de `LLVMContext` que sirve de capa de traducción entre LLVM y la *jvm* de manera concurrente. Hay un contexto por cada hilo. Además cuenta con un módulo de LLVM al que se le van insertando funciones para compilar.
- `SharkType`. Contiene varias rutinas para convertir los tipos de datos de la *jvm* a los de LLVM.
- `SharkValue`. Este es el tipo de datos que se apilan en la pila de *Shark*. Tiene rutinas para convertir valores de la *jvm* a los de LLVM y viceversa.
- `SharkCompileInvariants`. Es una clase creada para contener variables que acompañan a todos los métodos de alto nivel durante la compilación.
- `SharkTargetInvariants` Es una subclase de `SharkCompileInvariants` creada para los métodos cuyos valores pueden cambiar a lo largo de la compilación.
- `SharkNativeWrapper` Es una subclase de `SharkCompileInvariants` que, como adaptador del código nativo, tiene los argumentos, el tipo de retorno, una pila y la representación del método en la *jvm*.

De esta forma, *Shark* puede concebirse como un adaptador entre la *jvm* y LLVM que, además, aprovecha las características de compilación JIT de este último marco. Las siguientes secciones proporcionan detalles sobre LLVM y sus capacidades JIT.

2.6. LLVM

LLVM *toolchain* es un conjunto de APIs sobre interfaces bien definidas para construir compiladores. Fue creado como un proyecto de investigación en la Universidad de Illinois, y en la actualidad ha adquirido popularidad entre la comunidad de código abierto.

En el pasado los compiladores eran monolíticos y era difícil reusar sus componentes. Incluso hoy seria complicado reusar el compilador *gcc* para usarlo, por ejemplo, como compilador JIT. Es por eso que LLVM fue diseñado para ser modular. De esta forma, y considerando la abstracción convencional de un compilador en tres 3 capas (Figura 2.2):

- LLVM permite soportar distintos *front ends*. Desde el punto de vista LLVM, un *front end* es la capa de traducción del lenguaje de origen a la IR, una representación intermedia común al resto de lenguajes. De esta forma, con el *front end* el diseñador del lenguaje proporciona una correspondencia semántica entre el lenguaje fuente y la IR. Es posible, asímismo, disponer de *front ends* genéricos sobre los que puede mapearse múltiples lenguajes (por ejemplo, los *front end llvm-gcc* o *Clang* en la Figura 2.2).
- LLVM permite reutilizar y adaptar *middle ends*. El *middle end* es la capa de optimizaciones. En esta capa se aplican análisis y transformaciones a la representación intermedia.
- LLVM soporta, asímismo, múltiples *backends*, que definen la traducción del código intermedio en la IR a código nativo. Algunos *backends* soportados por LLVM son: alpha, arm, bfin, cellspu, mblaze, mips, mipsel, msp430, ppc32, ppc64, ptx32, ptx64, sparc, sparcv9, systemz, thumb, x86, x86-64.

El lenguaje intermedio de LLVM, la IR, es un lenguaje fuertemente tipado basado en registros de asignación única (SSA[9]): es decir, en la IR un registro sólo puede ser asignado una vez. Además en comparación con los lenguajes ensamblador, el IR de LLVM soporta una cantidad ilimitada de registros. Cada instrucción se agrupa en bloques básicos y la relación que se forma entre los bloques es un grafo dirigido acíclico (DAG), en el que los bloques están unidos por relación de llamada o salto. De esta forma, no es posible saltar a mitad de un bloque. Además existe un tipo de instrucción especial llamada *Phi* que permite representar bifurcaciones.

LLVM cuenta además con procedimientos para el procesamiento del IR. Estos procedimientos pueden ser:

- Procesamientos de *análisis* tales como *análisis de dependencia*, *lint*, *análisis de bucles*, *dependencia de memoria*, etc. que recaban información útil para la realización de posteriores optimizaciones.

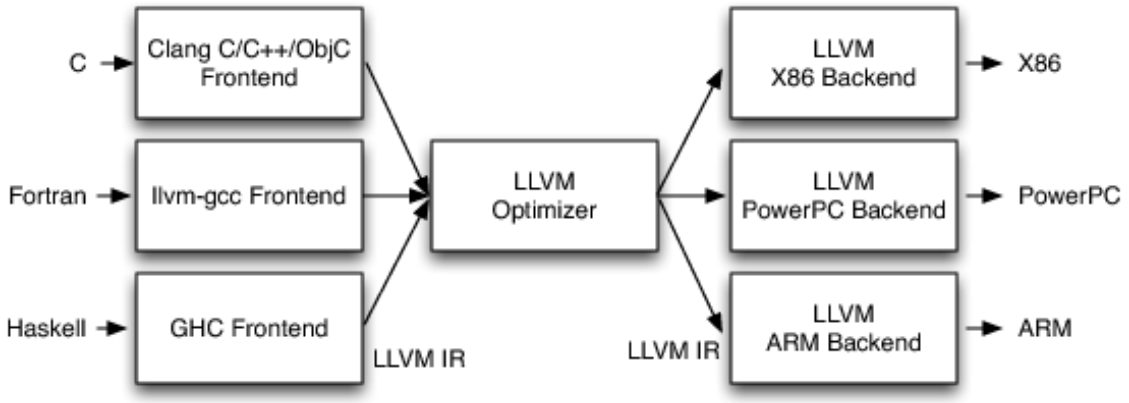


Figura 2.2: Papel de LLVM en la organización de un compilador (fuente: `http://www.aosabook.org/en/llvm.html`)

- Procesamientos de *transformación*, que llevan a cabo las optimizaciones pertinentes, procesando una IR y devolviendo otra IR equivalente optimizada. Algunos ejemplos son el *loop unrolling*, *eliminación de código muerto*, *inlining* o *jump threading*.

LLVM se usa extensivamente en la implementación de compiladores para multitud de lenguajes. Por nombrar algunos de ellos:

- Swift el lenguaje de programación de Apple para *apps* iOS.
- Clang, un compilador de C/C++ capaz de compilar el kernel de linux.
- Julia, un lenguaje de programación mutiparadigma, hecho con LLVM.
- Nvidia CUDA, un lenguaje similar a C que funciona sobre los procesadores de nvidia.
- Linux eBPF. LLVM permite compilar C a *bytecode extended Berkley Packet Filter (eBPF)*.
- CheerpJ, un compilador de java a *webassembly* que permite ejecutar aplicaciones *Java* en el navegador.

Las versiones actuales de LLVM, a diferencia de sus primeras versiones, se han modernizado y hacen un uso intensivo de *templates*, *move semantics* y los punteros inteligentes de la biblioteca *std*.

2.7. Soporte JIT en LLVM

A lo largo de las distintas versiones LLVM ha contado con 3 APIs o interfaces para crear compiladores JIT:

- *Legacy* JIT. Esta API fue la primera interfaz que pertenecía probablemente a versiones anteriores a la 3 de LLVM. De esta API tenemos poca documentación y está actualmente obsoleta.
- MCJIT. Esta fue la segunda interfaz y aún está presente en LLVM, aunque hay planes para su eliminación del proyecto, ya que la arquitectura de esta API es totalmente inflexible, sumando a esto que no soporta ejecución remota.
- ORC. Esta API debe su nombre a que soporta compilación bajo demanda (*On-Request Compilation*). Está destinada a reemplazar a la antigua MCJIT. Entre las nuevas características de ORC cabe destacar las siguientes: compilación remota, compilación perezosa, compilación concurrente, compilación especulativa.

A continuación resumimos las clases presentes en la API ORC:

- `LLJIT`. Esta clase proporciona un JIT prefabricado que ofrece todas las ventajas del API ORC de LLVM. La colaboración con el resto de componentes de ORC se ilustra en la Figura 2.3.
- `LLazyJIT`. Una especialización de LLJIT (Figura 2.4) que ofrece, además compilación perezosa.
- `JITDylib`. Representa una tabla símbolos que puede recibir consultas asíncronas. LLJIT crea una por defecto, que puede ser accedida mediante `LLJIT::getMainJITDylib`.
- `MaterializationUnit`. Representa un conjunto de símbolos que pueden ser materializados o eliminados en grupo.
- `ExecutionSession`. Representa un programa JIT en ejecución.
- `IRTransformLayer`. Representa una capa que procesa los módulos que serán emitidos por el compilador
- `IRBuilder`. Proporciona un API uniforme con el que crear la representación del código intermedio de LLVM.
- `IRCompileLayer`. Una capa que dice cómo, cuándo y desde donde se emiten las unidades de compilación.
- `LLJITBuilder`. Clase que permite la creación de LLJIT.

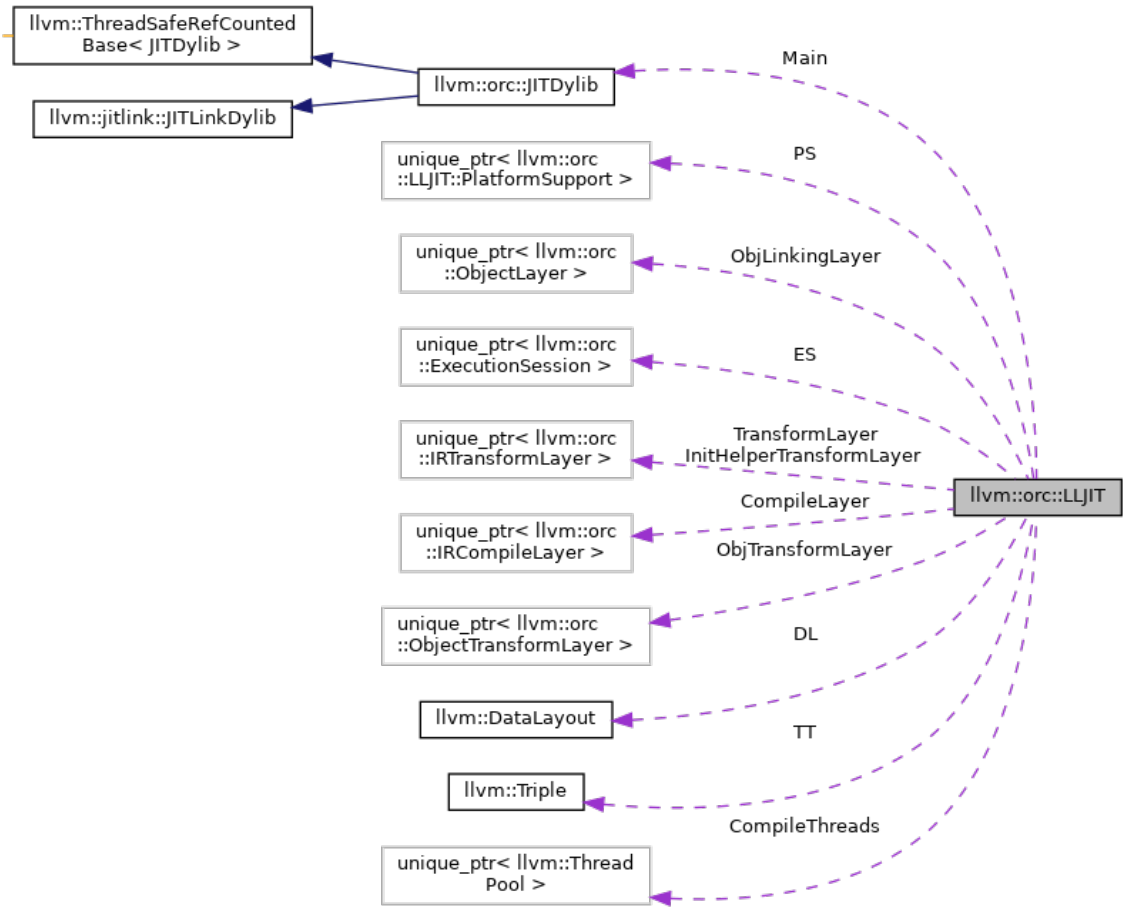


Figura 2.3: diagrama de colaboración de LLJIT con otros componentes de ORC

- `ThreadSafeModule`. Módulo de IR adaptado para el funcionamiento con LLJIT. Para modificar el módulo desde varios hilos hace falta bloquear el módulo con el metodo `ThreadSafeModule::withModuleDo()`.
- `ThreadSafeContext`. Contexto adaptado para el funcionamiento con `LLJIT`. Cada hilo de ejecución debe tener una contexto propio.

La última versión funcional de *Shark* se basaba en MCJIT. En este trabajo la hemos modernizado para funcionar sobre ORC.

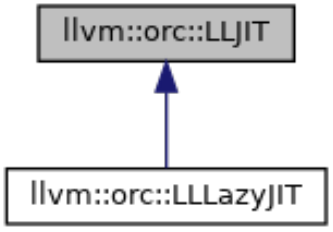


Figura 2.4: LLJIT y LLazyJIT

Capítulo 3

Identificación del problema: Modernización de Shark

3.1. Nacimiento, evolución y muerte de Shark

Los principales contribuyentes al proyecto Shark fueron dos ingenieros de Redhat, Gary Benson, el creador, y Roman Kennke. El proyecto se anunció[1] en el 2008 en las listas de correo porters-dev del *OpenJDK*. El proyecto pasó por dos grandes reconstrucciones: la primera[2] en el 2012 (debido a algunas modificaciones del *OpenJDK* como el desacoplamiento de operaciones y metadatos, así como a cambios en la IR de LLVM tales como la inclusión de *memory barriers* y operaciones atómicas), y la segunda[3] en el 2015 (para solucionar problemas de compilación en el openjdk9).

Posteriormente Roman Kennke, en el año 2017 publico, un JEP (*Java Enhancement Proposal*)[4], en el que propuso eliminar *Shark* de la distribución de *OpenJDK* con los siguientes argumentos:

- La comunidad no hacía mucho uso del proyecto.
- El proyecto no había recibido mantenimiento prácticamene en los dos dos últimos años.
- La interfaz JIT de LLVM utilizada por la implementación de *Shark* estaba obsoleta: el proyecto funcionaba con LLVM 3, y la versión más actual en 2017 era LLVM 5. Asimismo, el API JIT cambiaba con frecuencia con cada versión de LLVM.

[1] http://mail.openjdk.java.net/pipermail/porters-dev/2008-July/000160.html
[2] http://mail.openjdk.java.net/pipermail/hotspot-dev/2012-November/007294.html
[3] http://mail.openjdk.java.net/pipermail/hotspot-dev/2015-January/016714.html
[4] http://openjdk.java.net/jeps/8189173

- El *OpenJDK* aun no soportaba C++11 y LLVM 3.5 ya utilizaba las características de C++11.
- La interacción entre *Shark* y el recolector de basura afectaba al rendimiento.
- Las plataformas a las que daba soporte en un principio, ahora contaban con soporte oficial.

Aunque el código permaneció durante un tiempo, no funcionaba ya con el openjdk10. Efectivamente, la última versión funcional de *Shark* se encuentra en el openjdk9-b06. El proyecto fue eliminado definitivamente de *OpenJDK* en 2019.

3.2. Resurrección de Shark

A pesar de las razones pragmáticas que llevaron a la eliminación de *Shark* de la distribución de *OpenJDK*, desde nuestro punto de vista el enfoque del proyecto es extremadamente interesante desde el punto de vista de la portabilidad de la *jvm*. Efectivamente, el proyecto permite desvincular la *jvm* de arquitecturas concretas, delegando dicha responsabilidad en LLVM, un marco que, por otra parte, está precisamente diseñado para abordar dicha problemática. Por supuesto que queda aún un largo camino para conseguir madurar dicha tecnología, pero ello abre también prometedoras línea de investigación e innovación en el futuro. Es por ello que, como ya hemos comentado, tras nuestra exploración inicial nos propusimos, como concreción y culminación de este TFG, el *resucitar Shark*, modernizándolo para funcionar con las versiones más modernas de LLVM, y abordando su reintegro en la distribución actual de *OpenJDK*.

Reintegrar *Shark*k en la versión mas actual requiere lo siguiente:

- Conocer como funciona el sistema de compilación del openjdk.
- Recuperar los *macros* que seleccionan el código de SHARK y definen las variables globales con los valores por defecto de shark.
- Arreglar los conflictos con la API de LLVM, usar la nueva interfaz ORC de LLVM.

Todos estos aspectos se abordarán en el capítulo siguiente.

Capítulo 4

Abordaje del problema: Shark sobre LLVM 12

En este capítulo se resumen las principales actividades realizadas para "revivir" *Shark* en la distribución openjdk 17, modernizado para operar con LLVM 12.

4.1. Preparación del entorno de desarrollo

Para abordar el proyecto, nuestro primer paso fue elegir una distribución adecuada de Linux. Aunque con cualquier distribución de Linux es posible compilar todo, algunas distribuciones cuentan con bibliotecas compiladas disponibles mediante su administrador de paquetes, lo que facilita el subsecuente desarrollo. Para ello, se optó por una máquina virtual basada en Centos 7. El principal motivo, frente a la elección, por ejemplo, de la última versión de Ubuntu, es que Centos 7 es una distribución desarrollada por Redhat, que es un contribuyente principal al desarrollo del openjdk, por lo que se consideró una buena elección. En particular, reciclamos la máquina virtual de la asignatura Ampliación de Sistemas Operativos y Redes.

Seguidamente abordamos la preparación de un sistema de compilación adecuado. A este respecto, el openjdk17 requiere C++11, por lo que es necesario tener una versión mayor a la 4.7 del *toolchain* de gcc. En Centos 7 es posible conseguir la versión 8 del compilador GCC mediante el repositorio SCL. Este se instala en la carpeta `/opt/rh/devtoolset-8/root/bin` por lo que al compilar el *jdk* es necesario especificar que vamos a hacer uso de este toolchain.

4.2. Obtención de openjdk17

Normalmente la compilación del openjdk necesita de una instalación funcional del propio openjdk con una versión anterior a la que se quiere compilar. En particular, con openjdk17 necesitamos como mínimo el openjdk15. Para Centos 7 tenemos disponible binarios precompilados para Linux en `https://jdk.java.net/archive/`. En concreto, el openjdk15 puede obtenerse como:

```
wget https://download.java.net/java/GA/jdk15/779bf45e88a44cbd9ea
↪  6621d33e33db1/36/GPL/openjdk-15_linux-x64_bin.tar.gz
tar -xzf openjdk-15_linux-x64_bin.tar.gz
```

El siguiente paso fue descargar el openjdk17 desde un repositorio oficial: github (recomendado) o mercurial. En nuestro caso, la descarga desde github se lleva a cabo como:

```
git clone https://github.com/openjdk/jdk
# hg clone https://hg.openjdk.java.net/jdk/jdk/
```

Una vez disponible el openjdk17, los siguientes paquetes son necesarios para compilarlo en linux:

- Para *Swing* y el conjunto de interfaces gráficas necesitamos los componentes de Xorg, se pueden conseguir con el comando `sudo yum install libXtst-devel libXt-devel libXrender-devel libXrandr-devel libXi-devel`
- Para la interfaz de sonido es necesario ALSA, que se instala como `sudo yum install alsa-lib-devel`.
- Para Zero es necesario libffi (Portable foreign function interface) (`sudo yum install libffi-devel`), así como Cups (Comon Unix Printing system) (`sudo yum install cups-devel`).

Una vez hecho esto, los archivos de compilación de openjdk17 pueden obtenerse como:

```
bash configure                                \
    CC=/opt/rh/devtoolset-8/root/bin/gcc      \
    CXX=/opt/rh/devtoolset-8/root/bin/g++     \
    --with-jvm-variants=zero                  \
    --with-boot-jdk="../jdk-15.0.2/"
```

4.3. Obtención de LLVM

Desafortunadamente Centos no tiene paquetes disponibles para LLVM. Por tanto, para trabajar sobre el proyecto Shark se han compilado dos versiones distintas de LLVM, la versión 3.4 y la versión 12.0.

Para la versión 3.4 el proceso es similar a la compilación del openjdk, mediante configure y make. Sin embargo para la versión 12, el procedimiento de compilación se ha complicado: esta vez la compilación se hace mediante CMake, y Centos tampoco dispone de la versión necesaria de CMake, lo que nos obligó a actualizar dicha herramienta.

Una vez disponibles las herramientas adecuadas, lo primero fue conseguir el código de LLVM desde su repositorio en github:

```
# descargar todo el repositorio completo
git clone https://github.com/llvm/llvm-project/
# descargar solo la última versión
# git clone --depth=1 https://github.com/llvm/llvm-project/
cd llvm-project
```

Para compilar LLVM 12 con Cmake debemos crear una carpeta donde queremos guardar el resultado de la compilación y además tenemos en cuenta los siguientes parámetros, suponiendo que los binarios de CMake están en la carpeta `cmake-3.19.2/bin`:

```
mkdir llvm-build-dir
cd llvm-build-dir
../cmake-3.19.2/bin/cmake                                    \
        -DCMAKE_C_COMPILER=/opt/rh/devtoolset-8/root/bin/gcc  \
        -DCMAKE_BUILD_TYPE=MinSizeRel                        \
        -DCMAKE_CXX_COMPILER=/opt/rh/devtoolset-8/root/bin/g++\
        ../llvm-project/llvm/                                \
        -DLLVM_TARGETS_TO_BUILD="X86"                        \
        -DLLVM_BUILD_TOOLS=On                                \
        -DLLVM_INCLUDE_TOOLS=On
```

donde:

- `-DCMAKE_C_COMPILER` es la ubicación del compilador GCC 8.2
- `-DCMAKE_BUILD_TYPE` especifica que queremos una compilación sin información de depuración.

- `-DLLVM_TARGETS_TO_BUILD` especifica que la única plataforma que vamos a soportar el x86, ya que compilar todos los demás tardaría demasiado.
- `-DLLVM_BUILD_TOOLS` especifica que queremos las herramientas del toolchain.
- `-DLLVM_INCLUDE_TOOLS` especifica que cree targets de compilación para cada herramienta externa.

4.4. Inclusión de Shark en openjdk

Una vez teniendo disponible todos los requisitos pudimos continuar con la compilación del openjdk para integrar de nuevo Shark. La jvm del openjdk compila de forma estática: el resultado es un fichero .so con el cual enlazan distintos componentes del toolchain de java. Para ello, lo primero es ejecutar `configure`. Esto verifica la existencia de todas las dependencias y en caso de faltar alguna nos da un comando para instalarla desde nuestro gestor de paquetes. Además recibe parámetros para seleccionar el tipo de *jvm* que queremos compilar. En nuestro caso, elegimos la versión *zero*

Aquí distinguimos que para inspeccionar la última versión funcional de la jvm con Shark tuvimos que remitirnos a la openjdk9-b06, por lo que tuvimos que cambiar de commit. Afortunadamente la última versión de git nos permite tener copias de otro commit o rama en otra carpeta separada, por lo que no hace falta volver a clonar el jdk. Por el contrario, podemos usar worktree de la siguiente manera:

```
git worktree add ../jdk-shark jdk9-b06
```

```
bash configure
# para compilar la versión jdk9-b06 en su lugar tenemos que usar
#               --with-jvm-variants=zeroshark
#               --with-boot-jdk="../jdk-8/"
                --with-jvm-variants=zero
                --with-boot-jdk="../jdk-15.0.2/"
                CC=/opt/rh/devtoolset-8/root/bin/gcc
                CXX=/opt/rh/devtoolset-8/root/bin/g++
```

Como la última versión del jdk no tiene la variante shark, debemos reconstruir el sistema de compilación, regenerando el script `configure` mediante las herramientas `AUTOTOOLS` [1] de `gnu`, o cambiar la variante `Zero` para que incluya la compilación del proyecto shark. Esto último es más sencillo, por lo que fue la opción seguida.

[1] https://en.wikipedia.org/wiki/GNU_Autotools

Uno de los objetivos de configure es crear las variables con las que Make decida las plantillas que ha de usar para compilar, por lo que al invocar Make, se ha creado la variable variante Zero, en un bloque condicional podemos encontrar que otras características se incluyen cuando make detecta dicha variante, por ejemplo gcc debe activar los bloques de código indicados con la definición del macro `-DZERO -DSHARK` así como las definiciones de macros que necesiten LLVM y la ubicación de los headers de este: modificamos la variante zero de la siguiente manera:

```
#make/hotspot/lib/JvmFeatures.gmk
ifeq ($(call check-jvm-feature, zero), true)
  JVM_EXCLUDES += opto libadt
  JVM_EXCLUDE_PATTERNS += c1_ c1/ c2_ runtime_ /c2/
  JVM_EXCLUDE_FILES += templateInterpreter.cpp
  ↪  templateInterpreterGenerator.cpp \
                       bcEscapeAnalyzer.cpp ciTypeFlow.cpp
  JVM_CFLAGS_FEATURES += -DZERO
  ↪  -DZERO_LIBARCH='"$(OPENJDK_TARGET_CPU_LEGACY_LIB)"'
  ↪  $(LIBFFI_CFLAGS)
  JVM_CFLAGS_FEATURES += -DSHARK -DSHARK_LLVM_VERSION=34
  ↪  -I/usr/local/include -D_GNU_SOURCE -D__STDC_CONSTANT_MACROS
  ↪  -D__STDC_FORMAT_MACROS -D__STDC_LIMIT_MACROS
  JVM_LDFLAGS_FEATURES += $(shell llvm-config --ldflags) -lz
  JVM_LIBS_FEATURES += $(shell llvm-config --libs)
  JVM_LIBS_FEATURES += $(LIBFFI_LIBS)
  ifeq ($(ENABLE_LIBFFI_BUNDLING), true)
    JVM_LDFLAGS_FEATURES += $(call SET_EXECUTABLE_ORIGIN,/..)
  endif
else
  JVM_EXCLUDE_PATTERNS += /zero/
endif
```

Para finalizar este apartado copiamos los ficheros de la antigua versión de Shark con las modificaciones de la siguiente sección.

4.5. Reviviendo Shark

Para incluir la compilación de todo el código de *Shark* tenemos que modificar `JvmFeatures.gmk`. Los parámetros importantes dentro del target zero son los siguientes: `JVM_CFLAGS_FEATURES`, `JVM_LDFLAGS_FEATURES`, `JVM_LIBS_FEATURES`. Es necesario añadir los flags de LLVM y las opciones de enlazado con el uso de `llvm-config`

El primer error que nos encontramos al incluir *Shark* es que el *openjdk* ha restringido el uso del operador `new`. Todo el código dentro del *openjdk* gestiona la memoria

con herencia, lo que dificulta la adaptación de las antiguas clases dependientes de LLVM. Sin embargo se puede crear un alojador de memoria para LLVM instanciando `LLVMUser`. No obstante, nuestra motivación fue usar punteros inteligentes y *move semantics* para evitar fugas de memoria, así que desactivamos el mecanismo de esta detección. Esto afecta tanto a la compilación y en momento de ejecución, por lo que modificamos `CompileJvm.gmk` y `allocation.hpp`.

Entrando al código fuente pero siguiendo en el tema de compilación, tendremos que actualizar la parte en la que se inicializa *Shark* como producto. Por ejemplo el compilador C2 tiene inicializaciones distintas a las del C1, y lo lógico es que *Shark* también lo tenga. Por ello hay que actualizar
`src/hotspot/share/shark/shark_globals.hpp`
`src/hotspot/share/shark/shark_globals_zero.hpp`
`src/hotspot/share/utilities/macros.hpp` para que se adapte a la nueva declaración de flags.

Echando un vistazo a `allFlags.hpp` encontramos la definición del macro `ALL_FLAGS`. Esta macro hace uso de otras tales como `COMPILER1_PRESENT` o `COMPILER2_PRESENT` por lo que tenemos que crear un macro `SHARK_PRESENT` para que los macros de *Shark* se expandan cuando se invoque `ALL_FLAGS` en `globals.cpp`.

La expansión de `SHARK_FLAGS` produce que se creen variables globales de tipo `product, develop, diagnostic`. Estas son opciones que se pueden pasar al iniciar la *jvm*. Algunos ejemplos se muestran en ❶, ❷.

```
//hotspot/share/utilities/macros.hpp
// SHARK COMPILER
#ifdef SHARK
#define SHARK_PRESENT(code) code
#define NOT_SHARK(code)
#else // COMPILER2
#define SHARK_PRESENT(code)
#define NOT_SHARK(code) code
#endif // SHARK COMPILER

//src/hotspot/share/runtime/flags/allFlags.hpp
#define ALL_FLAGS(                                \
  /*...más declaraciones*/                        \
  COMPILER2_PRESENT(C2_FLAGS(                     \
    develop,                                      \
    develop_pd,                                   \
    product,                                      \
    product_pd,                                   \
    notproduct,                                   \
    range,                                        \
    constraint))                                  \
                                                  \
```

```
  SHARK_PRESENT(SHARK_FLAGS(                      \
    develop,                                       \
    develop_pd,                                    \
    product,                                       \
    product_pd,                                    \
    notproduct,                                    \
    range,                                         \
    constraint))                                   \
  /*...más declaraciones*/
↪   \

//src/hotspot/share/shark/shark_globals.hpp
#define SHARK_FLAGS(develop,                        \
                    develop_pd,                     \
                    product,                        \
                    product_pd,                     \
                    notproduct,                     \
                    range,                          \
                    constraint)                     \
                                                    \
  product(intx, MaxNodeLimit, 65000,                \
          "Maximum number of nodes")                \
                                                    \
  product(intx, SharkMaxInlineSize❶, 32,            \
    "Max bc size of methods to inline when using Shark")\
                                                    \
  product(bool, EliminateNestedLocks❷, true,        \
    "Elim nested locks of the same object when possible")\
  ...
```

Tras realizar estos ajustes podemos contar con que el código de *Shark* se va a compilar. Para reintegrar *Shark* como compilador, lo declaramos como un nuevo tipo de compilador añadiendo *Shark* como compilador, y su tipo de *stackframe*:

```
//src/hotspot/share/compiler/compilerDefinitions.hpp
enum CompilerType {
  compiler_none,
  compiler_c1,
  compiler_c2,
  compiler_jvmci,
  compiler_shark,
  compiler_number_of_types
};

//src/hotspot/cpu/zero/stack_zero.hpp
```

```
enum FrameType {
  ENTRY_FRAME = 1,
  INTERPRETER_FRAME,
  SHARK_FRAME,
  FAKE_STUB_FRAME
};

//src/hotspot/share/shark/sharkCompiler.cpp
SharkCompiler::SharkCompiler()
        : AbstractCompiler(compiler_shark)
{/*...*/}

//src/hotspot/share/compiler/abstractCompiler.hpp
bool is_shark() const { return _type == compiler_shark; }
```

Seguidamente agregamos una instancia de nuestro compilador al vector de compiladores disponibles al `CompilerBroker`, ya que es este el que utiliza los distintos compiladores presentes en la *jvm* cuando se invoca al método `CompilerBroker::compile_method`:

```
//src/hotspot/share/compiler/compileBroker.cpp
void CompileBroker::compilation_init_phase1(Thread* THREAD) {
  /*...*/
#ifdef SHARK
  if(_c2_count <= 0) _c2_count = 1;
  _compilers[1] = new SharkCompiler();
#endif
  /*...*/
}
```

En una fase posterior la *jvm* decide liberar el código compilado. Esta llama al método `flush` de una instancia de `nmethod` y esta a su vez necesita del compilador *Shark*, por lo que tenemos que conseguir la instancia del compilador, de la siguiente manera:

```
//src/hotspot/share/code/nmethod.cpp
void nmethod::flush() {
  /*...*/
#ifdef SHARK
  SharkCompiler *compiler = (SharkCompiler *)
   ↪  CompileBroker::compiler(_comp_level);
  compiler->free_compiled_method(insts_begin());
#endif // SHARK
  /*...*/
}
```

4.6. Actualizando Shark a LLVM 12

Para actualizar *Shark* a la versión 12 de LLVM comenzamos utilizando la clase `LLJIT` para reemplazar al `ExecutionEngine` e inicializamos dos contextos, uno para el código de java y otro para el código jni. Seguidamente tenemos que añadir módulos a cada contexto. Esto supone un cambio con respecto a MCJIT, ya que ahora se usa `ThreadSafeModule` para dar soporte a la compilación concurrente.

```
//src/hotspot/share/shark/sharkCompiler.cpp
SharkCompiler::SharkCompiler(): AbstractCompiler(compiler_shark) {
  // Create the lock to protect the memory manager and execution
  ↪  engine
  // Initialize the native target
  // initialize a native AsmPrinter

  auto nptr = std::make_unique<SharkContext>("normal");
  _normal_context = nptr.get();
  auto sptr = std::make_unique<SharkContext>("native");
  _native_context = sptr.get();

  auto J = llvm::orc::LLJITBuilder().create();
  //abortar si no se puede crear
  _execution_engine = std::move(*J);
  std::unique_ptr<llvm::Module>
  ↪  uptr_modnormal(_normal_context->module());
  std::unique_ptr<llvm::Module>
  ↪  uptr_modnative(_native_context->module());
  auto normalE = execution_engine()->addIRModule(
    llvm::orc::ThreadSafeModule(std::move(uptr_modnormal),
    ↪  std::move(nptr)));
  auto nativeE = execution_engine()->addIRModule(
    llvm::orc::ThreadSafeModule(std::move(uptr_modnative),
    ↪  std::move(sptr)));

  //abortar si (normalE || nativeE)
  // All done
}
```

Por ser ahora de tipo `unique_ptr` (1) se ha de utilizar el método `get` para conseguir el puntero a la instancia de `LLJIT` y también es necesario corregir el método `compile_method` para que acepte 2 parámetros nuevos (2) de `AbstractCompiler`

```
//src/hotspot/share/shark/sharkCompiler.cpp
llvm::orc::LLJIT* execution_engine() const {
```

```
  assert(execution_engine_lock()->owned_by_self());
  return _execution_engine.get();(1)
}

//src/hotspot/share/shark/sharkCompiler.hpp
void compile_method(ciEnv* env, ciMethod* target, int entry_bci);
void compile_method(ciEnv* env, ciMethod* target, int entry_bci, bool
↪  install_code, DirectiveSet* directive(2));
```

Otro aspecto que ha cambiado respecto a LLVM 3.4 es que no podemos acceder a la representación la IR de un método mediante el método `dump`. En su lugar se utiliza un `stringstream` para obtener dicha IR.

```
void SharkCompiler::generate_native_code(SharkEntry* entry,
                                         Function*   function,
                                         const char* name) {
  // Print the LLVM bitcode, if requested
  if (SharkPrintBitcodeOf != NULL) {
    if (!fnmatch(SharkPrintBitcodeOf, name, 0)){
      std::string strIR;
      llvm::raw_string_ostream OS(strIR);
      OS << *function;
      OS.flush();
      tty->print_cr("%s", strIR.c_str());
    }
  }
  // ... continuar con la generación del código nativo
}
```

También ha cambiado la forma de crear llamadas a rutinas que existen dentro de la máquina virtual a su equivalente en código intermedio. En la versión antigua, mediante `IRBuilder` se creaban instrucciones de tipo `InstCall` mediante el método `createCall` pero ahora necesita el tipo de la función, es decir, el tipo de retorno, el tipo de cada argumento y un booleano que indica si es una rutina con argumentos variables. Esto ha sido adaptado con el método `make_ftype` de la clase `SharkBuilder`, la cual se ha expuesto con visibilidad pública:

```
//src/hotspot/share/shark/sharkBuilder.hpp
public:
    static llvm::FunctionType* make_ftype(const char* params,
                                          const char* ret);
/*...*/

//src/hotspot/share/shark/sharkTopLevelBlock.hpp
llvm::CallInst* call_vm(llvm::Type*   retType,
```

```
                        llvm::Value*  callee,
                        llvm::Value** args_start,
                        llvm::Value** args_end,
                        int           exception_action) {
  /*juntar los argumentos con los ptr doblemente enlazados*/
  stack()->CreateSetLastJavaFrame();
  // crear una instruccion de llamada
  llvm::CallInst *res = builder()->CreateCall(
    llvm::FunctionType::get(retType, llvm::makeArrayRef(typesVect),
    ↪  false),
    callee,
    llvm::makeArrayRef(args_start, args_end));
  stack()->CreateResetLastJavaFrame();
  /*...*/
  return res;
}
```

De esta forma las llamadas se hacen conjuntamente con `make_ftype`. Por ejemplo, para conseguir el IR del operador `new` hacemos la siguiente llamada (obsérvese que `IRBuilderBase` va insertando las instrucciones en un bloque que se puede obtener mediante `IRBuilderBase::getInsertBlock()` por lo que no hace falta guardar el resultado en el código mostrado):

```
void SharkTopLevelBlock::do_new() {
  /*...*/
  call_vm(
    builder()->make_ftype("Ti", "v"),
    builder()->new_instance(),
    LLVMValue::jint_constant(iter()->get_klass_index()),
    EX_CHECK_FULL);
  /*...*/
}
```

En `SharkBuilder` también debemos actualizar las constantes que construyen instrucciones atómicas, para invocar adecuadamente a los recursos LLVM actuales:

```
//src/hotspot/share/shark/sharkBuilder.hpp
class SharkBuilder : public llvm::IRBuilder<> {
/*...*/
public:
  llvm::LoadInst* CreateAtomicLoad(llvm::Value* ptr,
        unsigned align = HeapWordSize,
        llvm::AtomicOrdering ordering = llvm::SequentiallyConsistent,
        llvm::AtomicOrdering ordering =
        ↪  llvm::AtomicOrdering::SequentiallyConsistent,
```

```
        llvm::SynchronizationScope synchScope = llvm::CrossThread,
        llvm::SyncScope::ID synchScope = llvm::SyncScope::System,
        bool isVolatile = true,
        const char *name = "");
  llvm::StoreInst* CreateAtomicStore(/*similar*/);
/*...*/
}
```

Otro cambio de la *jvm* actual es que `CppInterpreter` se ha eliminado y la entrada principal al interprete para los métodos no compilados será el análogo en `ZeroInterpreter`. Estos se introducen en los `SharkStack` de la siguiente manera (lo mismo sucede en `SharkBuilder` en la rutina `SharkBuilder::deoptimized_entry_point()` pero se utiliza en cambio la entrada al bucle principal ⓪):

```
//src/hotspot/share/shark/sharkStack.cpp
address SharkStackWithNormalFrame::interpreter_entry_point() const {
  return (address) ZeroInterpreter::normal_entry;
}
address SharkStackWithNativeFrame::interpreter_entry_point() const {
  return (address) ZeroInterpreter::native_entry;
}

Value* SharkBuilder::deoptimized_entry_point() {
  return make_function((address) ZeroInterpreter::main_loop⓪, "iT",
   ↪  "v");
}
```

Con C++11 se han creado sobrecargas a algunas funciones que se usaban en funciones intrínsecas de la *jvm*. La compilación se detiene porque se detecta ambigüedad de las funciones, la solución consiste en crear punteros como ① y ② a las funciones ambiguas y con ellas crear el IR de una función a código de la *jvm*:

```
//src/hotspot/share/shark/sharkBuilder.cpp
double (*tan_ref)(double) = &::tan;①
llvm::Value* SharkBuilder::tan() {
  return make_function((address) tan_ref, "d", "d");
}

double (*atan2_ref)(double, double) = &::atan2;②
llvm::Value* SharkBuilder::atan2() {
  return make_function((address) atan2_ref, "dd", "d");
}
```

Otro tipo de IR que se crea son los flags *safepoints*, puntos donde los hilos se detienen para que la jvm pueda hacer operaciones sin que la consistencia de la

ejecución de todos los hilos se vea afectada. En la última versión encontramos que se ha restringido la visibilidad del método `SafepointSynchronize::block` para poder crear estos flags con `SharkBuilder`:

```
//src/hotspot/share/runtime/safepoint.hpp
class SafepointSynchronize : AllStatic {
/*...*/
public:
  // Called when a thread voluntarily blocks
  static void block(JavaThread *thread);(0)
/*...*/
}

//src/hotspot/share/shark/sharkBuilder.cpp
llvm::Value* SharkBuilder::safepoint() {
  /*utilizamos*/
  return make_function((address)SafepointSynchronize::block(0), "T",
   ↪  "v");
}
```

Otro código afectado por la modernización es que las listas de inicialización deben ser inicializadas en el orden en el que se declaran para evitar ambigüedad. Esto implica en refactorizar cada constructor sin esta restricción.

Para añadir fases de optimización de LLVM, se ha utilizado un `IRTransform` que llama a una función `optimize` en la que se ejecuta un `PassManager` por cada `MaterializationUnit`. Finalmente se añade al execution_engine con `execution_engine->addIRTransform(optimize_transform)`:

```
//hotspot/share/shark/sharkCompiler.cpp
SharkCompiler::SharkCompiler()
  : AbstractCompiler(compiler_shark) {
  /*...*/
  IRTransform optimizeTransform(J->getExecutionSession(), optimize);
  J->addIRTransform(optimizeTransform);
  /*...*/
}

//hotspot/share/shark/sharkCompiler.hpp
class SharkCompiler{
/*...*/
public:
  static void optimize(MaterializationUnit mu){
    PassManager pm;
    pm.addPass(JumpThreadingPass());
```

```
    pm.addPass(LoopDeletion());
    pm.addPass(LoopExtractor());
    pm.addPass(LoopUnrollPass());
    /*more passes*/
    mu.run(pm);
  }
  /*...*/
}
```

La creación de cada método sería ineficiente si hubiera que crear un módulo por cada método, por lo que es mejor utilizar `MaterializationUnits` y de esta forma podemos manejar la eliminación en caso de que sea necesario liberar memoria. Esto se puede hacer así:

```
//hotspot/share/shark/sharkCompiler.cpp
void SharkCompiler::generate_native_code(SharkEntry* entry,
                                         Function*   function,
                                         const char* name) {
  /*...*/
  ResourceTracker rt;
  MaterializatinoUnit mu(entry, name);
  function.setRT(mu, rt);
  /*...*/
}


//hotspot/share/shark/sharkFunction.hpp
class SharkFunction{
  /*...*/
  ~SharkFunction(){//Destructor
    _rt.remove();
  }
  /*...*/
}
```

Por último, cuando la *jvm* requiera liberar espacio, invocará al método `free_compiled_method` y la liberación será automática, tal y como se muestra en el siguiente pequeño extracto de `SharkCompiler`:

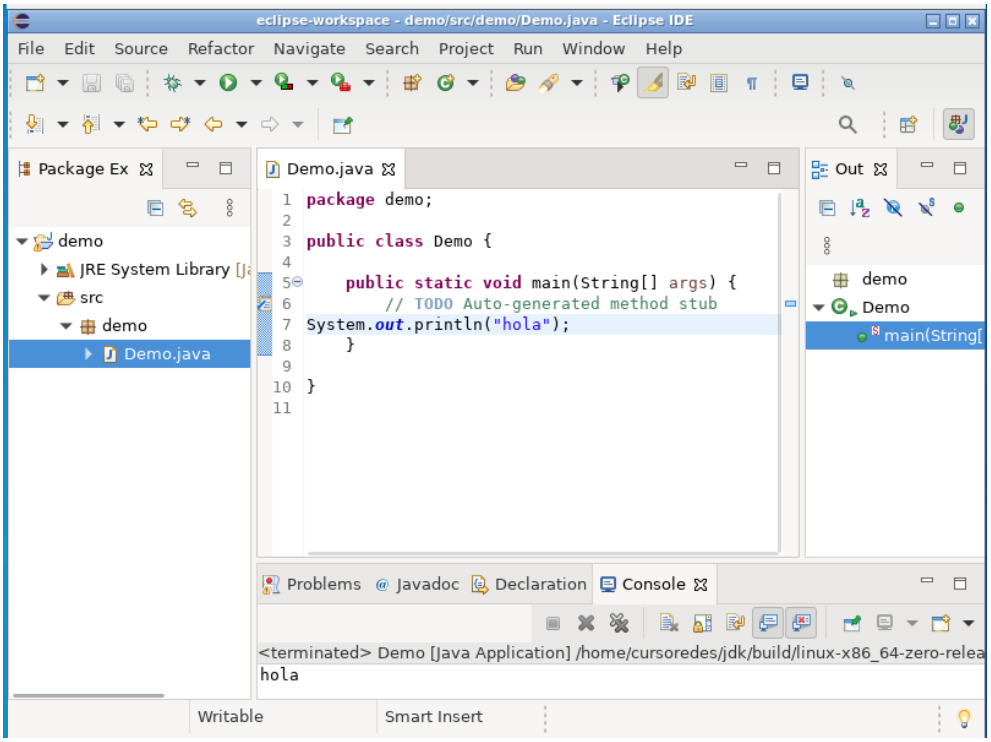
```
void SharkCompiler::free_compiled_method(address code) {
  SharkEntry *entry = (SharkEntry *) code;
  entry->context()->push_to_free_queue(entry->function());
}
```

Capítulo 5

Validación y Evaluación de la solución

5.1. Integración en Eclipse

El proyecto Zero/Shark revivido y modernizado para funcionar con las últimas versiones de *openjdk* y LLVM ha sido probado con la última versión de de Eclipse como se puede ver en la figura siguiente.



5.2. Pruebas preliminares

La variante del jdk obtenida se ha demostrado funcional con varios algoritmos que se incluyen en el repositorio público *TheAlgorithms*. Entre ellos se encuentran los algoritmos de Kruskal (Figura 5.1), A*, FloydWarshall, BinarySearch, BucketSort, BubbleSort, entre otros.

El repositorio se puede obtener mediante el comando:
`git clone https://github.com/TheAlgorithms/Java-git`

A modo de ejemplo, una vez clonado el repositorio, podemos compilar el algoritmo de Kruskal y compilarlo con https://github.com/TheAlgorithms/ con la siguiente dinámica:

```
//Users/cursoredes/Downloads/
jdk/build/linux-x86_64-zero-release/jdk/bin/javac -cp Java
Java/DataStructures/Graphs/Kruskal.java

jdk/build/linux-x86_64-zero-release/jdk/bin/java -cp
Java/DataStructures /Graphs Kruskal

Initial Graph:
0 <-- weight 3 --> 2
0 <-- weight 2 --> 1
0 <-- weight 3 --> 3
1 <-- weight 3 --> 4
1 <-- weight 4 --> 2
2 <-- weight 5 --> 3
2 <-- weight 1 --> 4
3 <-- weight 7 --> 5
4 <-- weight 8 --> 5
5 <-- weight 9 --> 6

Minimal Graph:
0 <-- weight 3 --> 3
0 <-- weight 2 --> 1
0 <-- weight 3 --> 2
2 <-- weight 1 --> 4
3 <-- weight 7 --> 5
5 <-- weight 9 --> 6
```

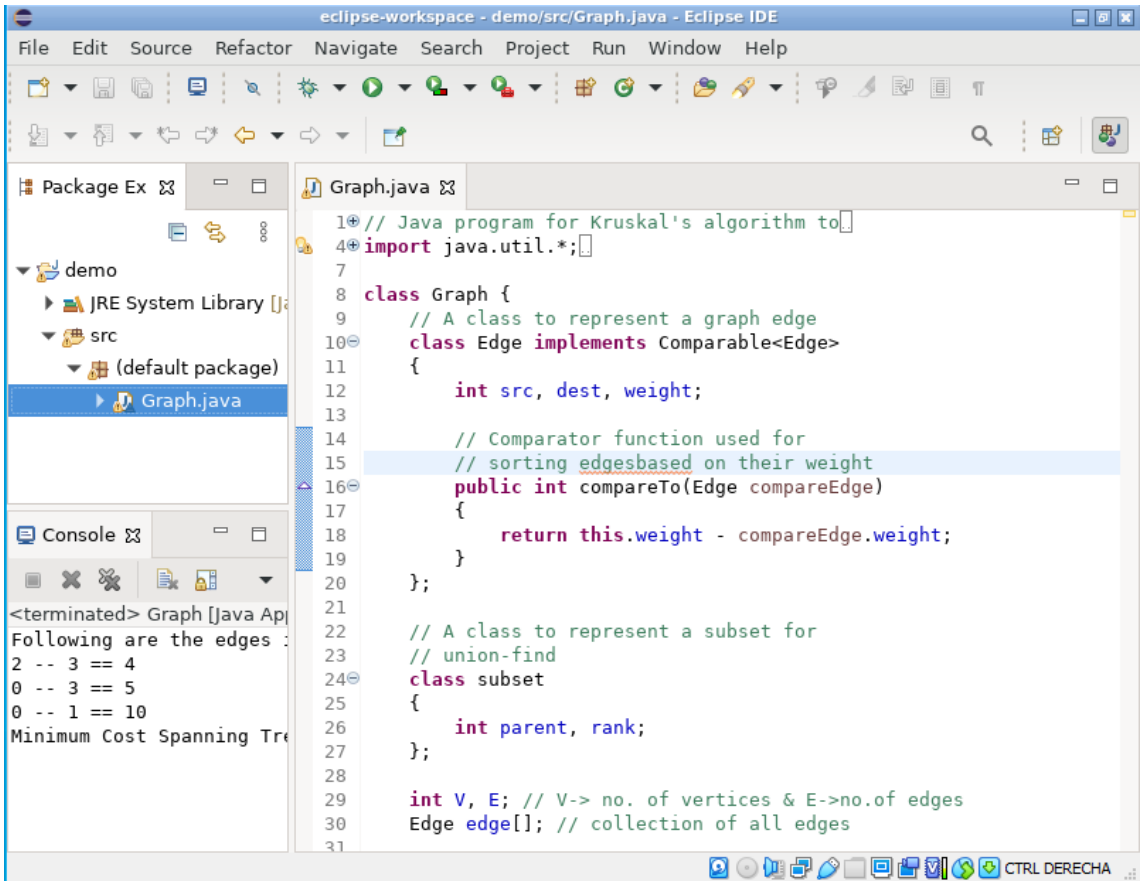


Figura 5.1: Detalles internos de Eclipse.

5.3. Tests de estrés

Por último, hemos sometido la variante a los *tests* de estrés de *OpenJdk* (en concreto, la batería de pruebas de regresión denominada *jtreg*). Aunque estas pruebas han evidenciado que hacen falta aún corregir ciertos *bugs* para que pueda pasar completamente los *tests*, la relación de aciertos y fallos ha sido mayormente positivos: se han superado satisfactoriamente 493 tests, habiendo fallado otros 150 (tasa de éxito por encima del 75 %).

Pueden consultarse todos los detalles de la ejecución de los *tests* en github[1].

Respecto al análisis de los que *tests* que han fallado, podemos destacar los siguientes:

- `OverflowCodeCacheTest.java` Este test prueba a desbordar el segmento de código compilado usando ambos compiladores. Como *Shark* solo usa 1 compilador y además no puede generar desborde, no produce el overflow esperado

[1] https://gist.github.com/danisilver/ff90a5b0fa6cb6435afb9046bc3a4826

que provoca la liberación de espacio en el segmento de código. Por tanto, el fallo de este test no se considera un problema en nuestro contexto.

- `TestCharVect2.d` y `TestUnswitchCloneSkeletonPredicates.d` Estos tests necesitan unas opciones muy específicas de la *jvm*. Por ejemplo, la segunda necesita la opción
  `PeelPartialLoop` que es una optimización en la que la primera iteración del bucle se extrae de este. Habría que implementar esta optimización manualmente mediante IR de LLVM.

- `IntrinsicAvailableTest.java`, `TestCRC32C.java` y
  `IntrinsicDisabledTest.java` . En estos test algunas funciones intrínsecas se dan por no encontradas. Por ejemplo, en el caso de CRC32 es un algoritmo de verificación de datos, que debería implementarse en el código del proyecto y exponerla como intrínseco de la *jvm*.

El análisis detallado de otros *tests* que han fallado se relega al trabajo futuro.

Capítulo 6

Conclusiones y Trabajo Futuro

6.1. Conclusiones

Hemos visto que LLVM es una librería muy versátil. Con cada versión se añaden mas características interesantes. Aunque esto pueda ser una desventaja se espera que ORC sea una API estable. En este trabajo hemos demostrado que su aplicación al problema de la portabilidad de la *jvm* es factible y abre una amplia gama de posibilidades.

A continuación, evaluamos el trabajo realizado en relación con los objetivos inicialmente planteados:

- Analizar el dominio de la virtualización y de las máquinas virtuales, con énfasis en la *jvm*, Este objetivo se ha abordado de manera satisfactoria, como evidencian los dos primeros capítulos de esta memoria, en los que se ha logrado introducir conceptos claves y describirlos en suficiente detalle para luego ahondar en las siguientes fases del proyecto.
- Abordar el problema de la portabilidad de la *jvm*, y, en particular, del componente *JIT* de la misma. Este objetivo ha sido el resultado de la exploración de capítulos previos y posteriormente de la programación sobre el código de la *OpenJdk*. La parte en la que se enfoca dicho esfuerzo en el *JIT Shark*. Como ya mencionamos hemos resucitado satisfactoriamente este proyecto por lo que este objetivo se da por satisfecho.
- Validar y evaluar los mecanismos propuestos para facilitar dicha portabilidad. Además de describir LLVM y Shark, sus APIs y funcionamiento, hemos logrado unir todo y con el resultado hemos ejecutado la batería de pruebas del *OpenJDK* (*jtreg*), por lo que consideramos también que este objetivo se ha completado satisfactoriamente.

6.1. Conclusions

We have seen that LLVM is a very versatile library. With each release more interesting features are added. Although this may be a disadvantage, ORC is expected to be a stable API. In this work we have shown that its application to the jvm portability problem is feasible and opens up a wide range of possibilities. In the following, we evaluate the work done in relation to the initially stated objectives:

- To analyze the domain of virtualization and virtual machines,with emphasis on jvm. This objective has been satisfactorily addressed as the firt two chapters of this report make apparent. As a consequence, it has been possible to introduce key concepts and describe these concepts in sufficient detail to then delve into the following phases of the project.
- To address the problem of jvm portability and, in particular, the JIT component of jvm. This objective has been the result of the exploration of previous chapters and of the subsquent programming with the OpenJdk code. This effort is focused on the Shark JIT. As already mentioned we have successfully resurrected this project so the deegree of achivement of this objective is considered satisfactory.
- To validate and evaluate the mechanisms proposed to facilitate such portability. In addition to describing LLVM and Shark, their APIs and operation, we have managed to integrate all the components and with the result we have executed the battery of tests of OpenJdk (jtreg). We therefore consider that this objective has also been satisfactorily completed.

6.2. Trabajo futuro

La primera línea de trabajo futuro es llevar a cabo un análisis detallado de los *tests* de estrés que han fallado durante la fase de evaluación del proyecto, para detectar los distintos aspectos de la modernización que no han sido adecuadamente manejados, y abordar dichos aspectos.

Otra línea clara de trabajo futuro es reintegrar *Shark* en la distribución principal de *openjdk*, así como convencer a la comunidad de su utilidad.

Otra línea sería abordar las limitaciones como la implementación de `SharkMemoryManager` ya que el código actual utiliza el reserva de memoria de LLVM, mientras que la *jvm* utiliza su propio sistema de reserva de memoria.

Otro posible salida de este trabajo es el estudio de la compilación hacia *webassembly*, como lo hace el proyecto *CheerpJ*, en la que se crea un sistema de ficheros virtual y se consigue una plataforma capaz de ejecutar programas java en el navegador.

Otra idea es crear una interfaz para cargar módulos de código en modo caliente dentro de la *jvm* mediante JITDYLib sin tener que utilizar código *jni* y poder interactuar mas libremente entre C++ y java.

Por último, como otra línea de trabajo futuro se estima interesante ampliar Zero/Shark para que pueda trabajar con las herramientas de profiling de linux como Perf, uprobes y eBPF. Un uso directo de Perf seria visualizar las partes del código que más tiempo tardan en ejecutarse mediante Flamegraphs como se muestra en la Figura 6.1.

6.2. Future Work

The first line of future work is to carry out a detailed analysis of the stress tests that have failed during the evaluation phase of the project, to detect the different aspects of the modernization that have not been adequately handled, and to address those aspects.

Another clear line of future work is to reintegrate Shark into the main openjdk distribution, as well as to convince the community of its usefulness. Another would be to address limitations such as the implementation of SharkMemoryManager since the current code uses LLVM's memory allocation, while jvm uses its own memory allocation system.

Another possible output of this work is the study of compilation to webassembly, as the CheerpJ project does, in which a virtual file system is created and a platform

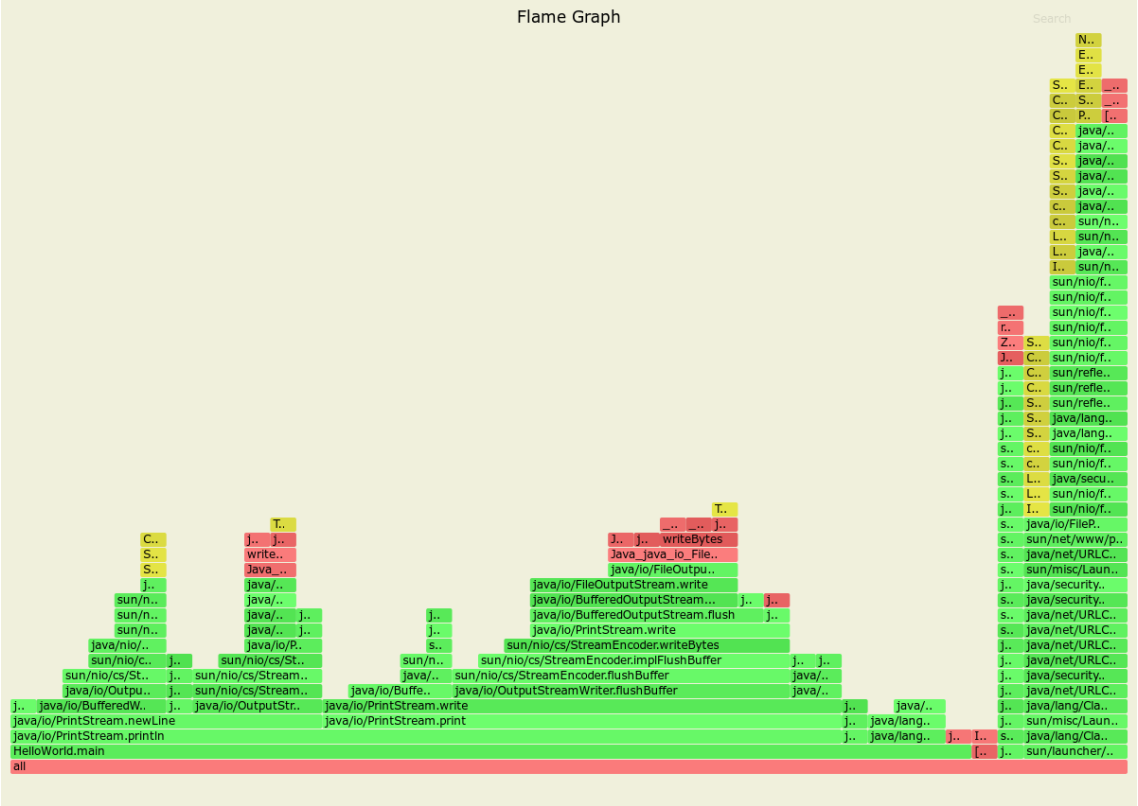


Figura 6.1: flamegraph

capable of running java programs in the browser is achieved.

Another idea is to create an interface to load code modules in hot mode inside the jvm through JITDYLib without having to use jni code and to be able to interact more freely between C++ and java.

Finally, as another line of future work it is considered interesting to extend Zero/Shark to work with linux profiling tools such as Perf, uprobes and eBPF. A direct use of Perf would be to visualize the parts of the code that take the longest time to execute using Flamegraphs as shown in Figure 6.1.

Bibliografía

[1] Jim Smith and Ravi Nair. *Virtual Machines: Versatile Platforms for Systems and Processes (The Morgan Kaufmann Series in Computer Architecture and Design)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2005. ISBN 1558609105.

[2] Reinhard Wilhelm and Helmut Seidl. *Compiler Design: Virtual Machines*. Springer Publishing Company, Incorporated, 1st edition, 2011. ISBN 3642149081.

[3] Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. *The Java Virtual Machine Specification, Java SE 8 Edition*. Addison-Wesley Professional, 1st edition, 2014. ISBN 013390590X.

[4] Bill Venners. *Inside the Java Virtual Machine*. McGraw-Hill, Inc., USA, 1996. ISBN 0079132480.

[5] V. van der Leun. *Introduction to JVM Languages*. Packt Publishing, 2017. ISBN 9781787126589. URL `https://books.google.es/books?id=7Hc5DwAAQBAJ`.

[6] Aarne Ranta. *Implementing Programming Languages, an introduction to compilers and interpreters*. College Publications, 2012. ISBN 1848900643.

[7] Mete Balci. Demystifying the jvm, interpretation, jit and aot compilation. `https://metebalci.com/blog/demystifying-the-jvm-interpretation-jit-and-aot-compilation/`, 2017.

[8] Wikipedia contributors. List of java virtual machines — Wikipedia, the free encyclopedia, 2021. URL `https://en.wikipedia.org/w/index.php?title=List_of_Java_virtual_machines&oldid=1039254350`. [Online; accessed 17-September-2021].

[9] Wikipedia contributors. Static single assignment form — Wikipedia, the free encyclopedia, 2021. URL `https://en.wikipedia.org/w/index.php?title=Static_single_assignment_form&oldid=1018849387`. [Online; accessed 17-September-2021].

Glosario

ABI Application Binary Interface. 15

API Application Programming Interface. 17

DAG Directed Aciclyc Graph. 24

eBPF extended Berkley Packet Filter. 25

HLL High Level Language. 16

ISA Instruction set arquitecture. 14

JIT Just In Time compilation. 4

JNI Java Native Interface. 16

JRE Java Runtime Environment. 17

JVM Java Virtual Machine. *Glosario:* jvm

jvm Máquina Virtual de java. 3, 15, 16

MV (también VM) Maquina virtual. 15

ORC On Request Compilation. 26

PC Program Counter. 16

plataforma Conjunto de las interfaces Hardware/Software. 15

RMI Remote Method Invocation. 16

SSA Single Static Assignment. 24

VISA Virtual Instruction Set Architecture. 15