UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO
FACULTAD DE CIENCIAS

Proyecto 02

Compiladores Just-in-Time (JIT) y optimización adaptativa

Compiladores

Integrantes:

Maryam Michelle Del Monte Ortega 320083527
Sahara Mariel Monroy Romero 320206391

Profesor: Manuel Soto Romero
Ayudantes: Diego Méndez Medina
Fausto David Hérnandez Jasso
José Alejandro Pérez Márquez
Jose Manuel Evangelista Tiburcio

1. Resumen

En este proyecto nos propusimos investigar cómo funcionan realmente los compiladores modernos, específicamente los del tipo *Just-in-Time* (JIT). La duda principal que queríamos resolver era cómo es posible que estos sistemas mejoren el código mientras el programa ya se está ejecutando. A diferencia de lo que solíamos pensar que un programa se compila una sola vez y ya, descubrimos que el JIT actúa más como un copiloto que va ajustando la ruta según el tráfico. Para entender esto a fondo, investigamos conceptos clave como el perfilado dinámico (*profiling*) y las trazas de ejecución. Además, para no quedarnos solo en la teoría, diseñamos una pequeña prueba que simula este comportamiento: un intérprete básico que detecta funciones muy usadas y cambia su ejecución por una versión optimizada, lo que nos permitió ver en la práctica cómo la frecuencia de uso es el detonante para la optimización.

2. Introducción

Cuando uno empieza a programar casi siempre nos presentan una división muy clara: por un lado están los lenguajes compilados (como C o C++), que son rápidos pero requieren un paso previo de "traducción"; y por otro, los interpretados (como Python o las versiones viejas de JavaScript), que son flexibles pero generalmente más lentos. Sin embargo, al investigar para este proyecto, nos dimos cuenta de que esa visión se ha quedado un poco obsoletas para explicar cómo funciona el software hoy en día.

La realidad actual es que necesitamos lo mejor de los dos mundos: queremos que el desarrollo sea rápido y dinámico, pero no queremos que la aplicación se arrastre al ejecutarse. Aquí es donde entran los compiladores JIT. La idea central es que el compilador no desaparece cuando termina de generar el ejecutable, sino que se queda "vivo"dentro de la máquina virtual observando qué pasa.

El objetivo de este ensayo es explicar cómo el JIT logra ajustar la optimización del código basándose en el comportamiento real del programa. No se trata de adivinar qué hará el código (como hace el análisis estático), sino de ver qué está haciendo realmente. A lo largo del texto explicaremos el proceso de "perfilado", cómo se toman decisiones de optimización y describiremos el experimento práctico que realizamos para visualizar este fenómeno.

3. £Qué es realmente el JIT?

Para entender el JIT, primero hay que quitarse la idea de que la compilación es un evento único. En nuestra investigación encontramos un artículo muy claro en *freeCodeCamp* que lo explica de una forma muy sencilla: el JIT es básicamente una compilación que ocurre en el último momento posible", justo antes de que el código necesite ejecutarse [1].

El articulo señala algo interesante: el código fuente (o el bytecode, en el caso de Java) se traduce a instrucciones de máquina en tiempo real. Esto permite que el programa arranque rápido (porque no compila todo de golpe al principio) y que luego, poco a poco, vaya ganando velocidad. Es una estrategia híbrida.

Lo que nos pareció más relevante de esta fuente es la distinción que hace sobre el contexto. Un compilador estático (AOT) no sabe si el usuario va a usar una función una vez o un millón de veces. El JIT, al estar ahí mientras todo ocurre, tiene esa información privilegiada. Esto cambia por completo las reglas del juego, porque permite optimizaciones que serían arriesgadas o imposibles de hacer antes de ejecutar el programa.

4. El "Monitor": Perfilado en tiempo de ejecución

La pregunta clave de nuestra investigación era: £Cómo sabe el compilador qué partes optimizar? La respuesta está en el perfilado o *profiling*.

Imaginemos que el entorno de ejecución tiene un contador interno. Cada vez que se llama a un método o se da una vuelta en un bucle, ese contador sube. Según la documentación técnica de IBM [2], el compilador JIT utiliza estos contadores para identificar las zonas *calientes* del código, conocidas como *hot spots*.

Al principio, el programa corre en modo interpretado (o con una compilación muy básica y rápida). El sistema no gasta recursos en optimizar nada porque no sabe qué vale la pena. Pero, cuando una parte del código

supera cierto umbral de ejecuciones (digamos, se ha llamado a una función mil veces), el monitor alerta al compilador JIT. En ese momento, el sistema decide que vale la pena invertir tiempo de CPU en compilar esa función específica a código máquina altamente optimizado.

Este proceso es fascinante porque implica que el rendimiento del programa no es constante; mejora con el tiempo. Es lo que se llama *calentamiento* (*warm-up*). Cuanto más tiempo corre el programa, más inteligente se vuelve el JIT sobre cómo ejecutarlo eficientemente.

5. Optimización Adaptativa y Especulación

Aquí es donde la cosa se pone más compleja y técnica. Una vez que el JIT decide optimizar, no solo traduce el código tal cual. Hace algo llamado ”optimización especulativa”

Básicamente, el compilador hace apuestas. Si observa que una variable en una función siempre ha sido un número entero durante las últimas 500 llamadas, asume (especula) que siempre será un entero. Entonces, genera código máquina específico para sumar enteros, eliminando todas las comprobaciones de tipo que suelen hacer lenguajes dinámicos como JavaScript. Esto hace que el código sea notablemente rápido.

Pero, £qué pasa si la apuesta falla? Si de repente llega un texto en lugar de un número, el código optimizado fallaría. Los sistemas JIT tienen una "salida de emergencia"(llamada *deoptimization* o *bailout*). Si la suposición deja de ser cierta, el sistema tira a la basura el código optimizado y vuelve a la versión interpretada lenta pero segura. Esta capacidad de ir y venir entre versiones rápidas y lentas es lo que llamamos optimización adaptativa [3].

6. Tracing JIT: Siguiendo el rastro

Investigando un poco más sobre cómo se implementa esto, encontramos el concepto de *Tracing JIT*, que es muy popular en implementaciones como PyPy para Python. En lugar de mirar funciones enteras, el compilador mira ”caminos” (*traces*).

Como explican en el artículo de *Kipp.ly*, los programas suelen pasar mucho tiempo en bucles. El Tracing JIT graba exactamente qué camino toma el código dentro de un bucle (por ejemplo, entró al ‘if‘, luego sumó, luego volvió al inicio") [4]. Luego, compila esa secuencia lineal de instrucciones. Esto es muy efectivo porque ignora todo el código que no se usa en ese momento, enfocándose puramente en lo que el programa está haciendo en la realidad.

7. Descripción del Caso Práctico

La implementación que hicimos su objetivo es mostrar de manera clara y directa como un compilador Just-In-Time puede ajustar la forma de ejecutar un programa basandose solo en lo que observa durante su ejecución. El sistema comienza ejecutando todas las funciones de manera sencilla sin asumir nada sobre su importancia o frecuencia de uso.

Durante la ejecución, el sistema mantiene un contador asociado a cada función. Cada vez que una función es llamada, su contador aumenta, permitiendo que el sistema identifique cuáles partes del programa se utilizan con mayor frecuencia. Cuando una función alcanza un número determinado de ejecuciones, se considera que esa parte del programa es relevante y que vale la pena cambiar su forma de ejecución. Esta decisión no se toma antes de correr el programa, sino como resultado directo de su comportamiento observado.

Una vez alcanzado el umbral de uso, el sistema reemplaza la versión original de la función por una version mas directa y eficiente. Este cambio ocurre mientras el programa sigue en ejecucion y no requiere reiniciar ni recompilar el programa completo. De esta forma, se simula el comportamiento de la optimización adaptativa, donde el sistema aprende y se ajusta conforme avanza la ejecución.

Las capturas siguientes muestrans el momento en que el sistema decide realizar este ajuste. En la salida se observa claramente un mensaje que indica cuándo se aplica el ajuste dinámico, seguido de la continuación normal del programa. Esta evidencia visual confirma que la optimización depende del uso real de la función y no de una decisión tomada de antemano.

```
maryam@skibidi:~/compiladores/Proyecto2-Compiladores/src$ runhaskell Main.hs
Iteracion 0:
  Resultado: 6
  Funcion ''suma'' ejecutada 1 veces
  Estado: no optimizada

Iteracion 1:
  Resultado: 6
  Funcion ''suma'' ejecutada 2 veces
  Estado: no optimizada

Iteracion 2:
  Resultado: 6
  Funcion ''suma'' ejecutada 3 veces
  Estado: no optimizada

Iteracion 3:
  Resultado: 6
  Funcion ''suma'' ejecutada 4 veces
  Estado: no optimizada

Iteracion 4:
  Resultado: 6
  Funcion ''suma'' ejecutada 5 veces
  Estado: optimizada

Iteracion 5:
  Resultado: 6
  Funcion ''suma'' ejecutada 6 veces
  Estado: optimizada

Iteracion 6:
  Resultado: 6
  Funcion ''suma'' ejecutada 7 veces
  Estado: optimizada

Iteracion 7:
  Resultado: 6
  Funcion ''suma'' ejecutada 8 veces
  Estado: optimizada
```

```
Iteracion 8:
  Resultado: 6
  Funcion ''suma'' ejecutada 9 veces
  Estado: optimizada

Iteracion 9:
  Resultado: 6
  Funcion ''suma'' ejecutada 10 veces
  Estado: optimizada

Ejecucion terminada con 10 iteraciones completadas
```

8. Conclusiones

Realizar este proyecto nos ha permitido cambiar nuestra perspectiva sobre cómo se ejecuta el software. Antes veíamos la compilación como un paso estático, pero ahora entendemos que es un proceso vivo y dinámico.

Podemos concluir que:

- El JIT es necesario en el mundo actual porque los lenguajes dinámicos (como JS o Python) son muy difíciles de optimizar estáticamente. Se necesita verlos correr para saber cómo hacerlos rápidos.
- La clave de todo es el *profiling*. Sin datos en tiempo real, el compilador estaría "ciego".
- La optimización adaptativa permite escribir código limpio y legible, confiando en que el compilador hará el trabajo sucio de optimizar las partes críticas.

En resumen, los compiladores JIT aprovechan el comportamiento del programa para tomar decisiones que un humano o un compilador estático no podrían anticipar.

Referencias

[1] C. Nokes, "Just-In-Time Compilation Explained," *freeCodeCamp.org*, 13-feb-2018. [En línea]. Disponible en: `https://www.freecodecamp.org/news/just-in-time-compilation-explained/`. Accedido: 13-dic-2025.

[2] IBM, "JIT compiler," *IBM SDK, Java Technology Documentation*. [En línea]. Disponible en: `https://www.ibm.com/docs/es/sdk-java-technology/8?topic=reference-jit-compiler`. Accedido: 13-dic-2025.

[3] Mozilla Developer Network, "Just-In-Time Compilation," *MDN Web Docs*. [En línea]. Disponible en: `https://developer.mozilla.org/en-US/docs/Glossary/Just_In_Time_Compilation`. Accedido: 13-dic-2025.

[4] Kipp.ly, "An Introduction to JITs," *Kipp.ly Engineering Blog*. [En línea]. Disponible en: `https://kipp.ly/jits-intro/`. Accedido: 13-dic-2025.