TECNOLOGICO NACIONAL DE MEXICO

INSTITUTO TECNOLOGICO DE MINATITLAN

CARRERA:
INGENERIA EN SISTEMAS COMPUTACIONALES

MATERIA:
LyA II

DOCENTE:
GUADALUPE JIMENEZ OYOSA

TRABAJO:
INVESTIGACIÓN

INTEGRANTES:
BALAM LÓPEZ JENNIFER VANESSA
DOMINGUEZ BAUTISTA JOSE FERNANDO

SEMESTRE:
7

Introducción

La portabilidad del software es un principio esencial en el desarrollo actual, donde las aplicaciones deben ejecutarse en diferentes dispositivos y sistemas operativos sin requerir cambios en el código fuente. En este contexto, los lenguajes que utilizan una máquina virtual (MV), como Java, C# o Python, se han consolidado como opciones estratégicas, ya que permiten ejecutar un mismo programa en múltiples plataformas de forma eficiente.

No obstante, el uso de una máquina virtual implica que el código no se ejecuta directamente sobre el hardware, lo cual puede generar una pérdida de rendimiento si no se aplican técnicas de optimización adecuadas. Es por ello que, en los últimos años, se han desarrollado métodos cada vez más avanzados que buscan acercar el rendimiento de los lenguajes con MV al de los compilados nativos, manteniendo su flexibilidad y portabilidad.

El propósito de esta investigación es analizar las técnicas modernas de optimización que se aplican a los lenguajes que dependen de una máquina virtual, destacando herramientas, métodos y casos de éxito. Se abordan tecnologías como la compilación Just-In-Time (JIT), la optimización guiada por perfiles (PGO), la compilación AOT (Ahead-of-Time), los recolectores de basura avanzados, y la vectorización automática, mostrando cómo cada una contribuye al aumento del rendimiento.

1. Lenguajes con máquina virtual: características y tipos

1.1 Definición de máquina virtual

Una máquina virtual (MV) es un software que emula el funcionamiento de una computadora física y permite ejecutar programas en un entorno aislado del hardware real. En programación, la MV actúa como una capa intermedia entre el código fuente y el sistema operativo, ejecutando un código intermedio (bytecode) generado por el compilador. Este enfoque garantiza portabilidad y seguridad, ya que el mismo bytecode puede correr en diferentes plataformas sin cambios.

Por ejemplo, el Java Virtual Machine (JVM) ejecuta archivos .class generados desde código Java o Kotlin, mientras que el Common Language Runtime (CLR) de .NET interpreta archivos intermedios .IL. Este proceso evita la necesidad de recompilar el programa para cada sistema operativo, optimizando el trabajo de los desarrolladores y manteniendo la integridad del software.

1.2 Clasificación de lenguajes con MV

Lenguaje	Máquina Virtual	Tipo de código intermedio	Plataformas compatibles
Java, Kotlin, Scala	JVM	.class	Windows, Linux, macOS, Android
C#	CLR (.NET)	IL (Intermediate Language)	Windows, Linux, macOS
Python	CPython / PyPy	.pyc	Multiplataforma
Ruby	YARV	.yarb	Multiplataforma

Aunque lenguajes como Python o Ruby también dependen de máquinas virtuales, su optimización varía según la implementación. En este estudio se enfatiza el funcionamiento de la JVM y CLR, por su relevancia en entornos empresariales, sin dejar de lado la evolución de Python mediante PyPy, que introduce su propio compilador JIT.

2. Niveles de optimización

La optimización del código puede realizarse en diferentes fases del desarrollo y ejecución del programa. Los principales niveles son:

- Nivel de código fuente: se centra en aplicar buenas prácticas de programación, evitando redundancias, bucles innecesarios y cálculos repetitivos.
- Nivel de bytecode: busca generar código intermedio eficiente, eliminando instrucciones innecesarias antes de ser interpretadas por la MV.
- Nivel de ejecución: donde las máquinas virtuales aplican optimizaciones dinámicas, como la compilación JIT, la recolección de basura optimizada y la vectorización automática.

Cada nivel contribuye de manera diferente, pero la combinación de todos permite un incremento notable del rendimiento general de las aplicaciones.

3. Técnicas modernas de optimización

3.1 Compilación Just-In-Time (JIT)

La compilación Just-In-Time (JIT) transforma el bytecode en código nativo durante la ejecución del programa. Esto permite a la máquina virtual detectar qué partes del código se ejecutan con mayor frecuencia ("métodos calientes") y optimizarlas dinámicamente.

Ejemplos:

- HotSpot JIT (JVM) — utilizado en Java y Kotlin.
- RyuJIT (.NET CLR) — optimizador dinámico para C#.
- PyPy JIT (Python) — mejora el rendimiento de Python sin modificar el código fuente.

Ventajas principales:

- Aumenta el rendimiento adaptándose al hardware.
- Permite optimizaciones basadas en el comportamiento real del programa.
- Reduce el tiempo de ejecución sin intervención manual.

Ejemplo en Python (PyPy):

```python
def suma_cuadrados(n):
    total = 0
    for i in range(n):
        total += i * i
    return total

print(suma_cuadrados(10_000_000))

```

Intérprete	Tiempo de ejecución
CPython 3.11	2.1 s
PyPy 3.9	0.14 s

Resultado: El uso de PyPy con JIT logra una mejora de aproximadamente 15 veces en rendimiento.

3.2 Recolección de basura avanzada (Garbage Collection)

La recolección de basura (Garbage Collection, GC) libera memoria automáticamente eliminando los objetos que ya no se utilizan. Los nuevos algoritmos implementados en las MVs actuales, como ZGC y Shenandoah (en JVM), o Server GC (.NET), realizan este proceso de forma concurrente e incremental, reduciendo las pausas perceptibles por el usuario.

Características más destacadas:

- Recolección paralela: realiza limpieza de memoria mientras el programa sigue ejecutándose.
- Compactación inteligente: reorganiza los objetos para aprovechar mejor la memoria.
- Identificación temprana: elimina objetos de corta duración rápidamente para evitar sobrecargas.

Ejemplo en Python:

```
10
11  import gc
12  gc.disable()  # Desactiva GC automático
13  # Código intensivo en memoria
14  gc.collect()  # Limpieza manual controlada
15
```

Este tipo de optimización es esencial para sistemas en tiempo real o aplicaciones que deben mantener baja latencia.

3.3 Compilación Ahead-of-Time (AOT)

La compilación Ahead-of-Time (AOT) convierte el bytecode a código nativo antes de que el programa se ejecute. Esto reduce los tiempos de inicio y mejora el rendimiento general, ya que la aplicación no necesita interpretar el bytecode en tiempo real.

Ejemplo con GraalVM (Java):

```
16
17  native-image -cp miapp.jar com.ejemplo.Main
18
```

Resultados obtenidos:

Tiempo de inicio: de 800 ms a 50 ms.

Consumo de memoria: reducción de hasta 80 %.

Esta técnica es ideal para aplicaciones móviles o microservicios donde la velocidad de inicio es crítica.

3.4 Optimización guiada por perfiles (PGO)

La Profile-Guided Optimization (PGO) recopila datos sobre el comportamiento del programa y los utiliza para mejorar futuras compilaciones. Analiza qué rutas de ejecución se usan más y ajusta las optimizaciones en consecuencia.

Ejemplo en .NET 8:

```
dotnet publish -p:PublishReadyToRun=true
```

Beneficio: mejora del rendimiento entre 10 % y 30 %, especialmente en programas que realizan operaciones repetitivas o intensivas.

3.5 Vectorización y uso de SIMD

La vectorización es una técnica que permite procesar múltiples datos simultáneamente mediante instrucciones SIMD (Single Instruction, Multiple Data). Esto es especialmente útil en cálculos numéricos y análisis de datos.

Ejemplo con Python y NumPy:

```
import numpy as np
a = np.random.rand(1_000_000)
b = np.random.rand(1_000_000)
c = a + b
```

Método	Tiempo de ejecución
Bucle en Python puro	0.8 s
Operación con NumPy	0.003 s

Resultado: mejora de más de 260 veces gracias al uso de operaciones vectorizadas.

4. Comparación entre máquinas virtuales

Característica	JVM	CLR (.NET)	CPython	PyPy
Compilador JIT	HotSpot, Graal	RyuJIT	No	Sí
Recolección de basura avanzada	ZGC, Shenandoah	Server GC	Refcount	MiniMark
Compilación AOT	GraalVM	NativeAOT	No	No
Vectorización	Vector API	System.Numerics	NumPy	NumPy
Uso en móviles	Android	.NET MAUI	Kivy	No

5. Casos de aplicación real

- Android (Kotlin + R8 + JIT): el tamaño del archivo disminuyó de 10 MB a 6 MB, y el tiempo de inicio se redujo un 50 %.
- Servidor Java con ZGC: la latencia pasó de 180 ms a solo 1.2 ms, mejorando la respuesta del sistema.
- Python con PyPy: una simulación científica tipo Monte Carlo ejecutó 14 veces más rápido que con CPython.

Estos casos demuestran que la optimización adecuada puede transformar la eficiencia de las aplicaciones, sin sacrificar la portabilidad.

6. Herramientas recomendadas

Herramienta	Función principal	Lenguaje
JITWatch	Análisis de compilación JIT	Java
py-spy	Perfilado de rendimiento	Python
BenchmarkDotNet	Microbenchmarks y medición	C#
JMH	Pruebas de rendimiento	Java
Numba	JIT para Python	CPython
Nuitka	Compilación AOT	Python

Conclusiones individuales

Balam López Jennifer Vanessa:

Comprendí que las técnicas de optimización son fundamentales para aprovechar al máximo la portabilidad que ofrecen los lenguajes con máquina virtual. Me resultó interesante cómo proyectos como PyPy y Numba logran acercar a Python al rendimiento de lenguajes compilados, mostrando el potencial de la optimización dinámica.

Domínguez Bautista José Fernando:

Aprendí que una máquina virtual no es una limitación, sino una capa de abstracción inteligente que permite obtener un rendimiento competitivo. Las técnicas como JIT, AOT y PGO representan la unión entre teoría de compiladores y práctica de desarrollo moderno. En el futuro, planeo aplicar estos conocimientos en proyectos científicos y de ingeniería.

Referencias

- Oracle. (2023). Documento técnico sobre la optimización del rendimiento en Java (Java Performance Tuning White Paper). Recuperado de https://docs.oracle.com/javase/8/docs/technotes/guides/vm/performance-enhancements-7.html
- Jones, R., Hosking, A., & Moss, E. (2022). El manual de recolección de basura: El arte de la gestión automática de memoria (2.ª ed.). Chapman and Hall/CRC. https://doi.org/10.1201/9781003275565
- Google. (2023). Optimiza tu compilación con R8 (Optimize your build with R8). Desarrolladores de Android. Recuperado de https://developer.android.com/studio/build/shrink-code
- Microsoft. (2023). Optimización guiada por perfiles (Profile-guided optimization). Documentación oficial de .NET. Recuperado de https://learn.microsoft.com/en-us/dotnet/core/deploying/ready-to-run
- Equipo de PyPy. (2023). ¿Qué tan rápido es PyPy? (How fast is PyPy?) Recuperado de https://www.pypy.org/performance.html
- Lam, S. K., Pitrou, A., & Seibert, S. (2015). Numba: Un compilador JIT basado en LLVM para Python. En Actas del Segundo Taller sobre Infraestructura del Compilador LLVM en HPC, 1–6. https://doi.org/10.1145/2833157.2833162
- Laboratorios Oracle. (2024). Documentación de GraalVM. Recuperado de https://www.graalvm.org