Chapter 1

Tópicos Avançados: Compilação JIT

1.1 Objetivos

⋆ title=Objetivos da Aula

Ao final deste capítulo, o estudante deverá ser capaz de:

- Compreender o espectro de execução de programas, diferenciando Compilação *Ahead-of-Time* (AOT) e Interpretação.
- Explicar o paradigma da compilação *Just-In-Time* (JIT) e como ele combina vantagens de ambas as abordagens.
- Descrever a arquitetura de múltiplos níveis (*Tiered Compilation*) presente em máquinas virtuais modernas (JVM, V8).
- Entender o conceito de Otimização Especulativa e os benefícios do *Polymorphic Inline Caching* (PIC).
- Explicar os mecanismos de Desotimização (*Bailout*) e *On-Stack Replacement* (OSR).

1.2 Introdução: O Espectro de Execução

Historicamente, as linguagens de programação foram divididas em duas grandes categorias baseadas na forma como seu código fonte era executado pelas máquinas: **Compiladas AOT** e **Interpretadas**.

Na **Compilação AOT (Ahead-of-Time)**, vista em linguagens como C, C++ e Rust, o compilador recebe o código fonte e gasta todo o tempo necessário para traduzi-lo em um binário executável nativo. O usuário final roda o binário diretamente no hardware. Essa

abordagem garante desempenho máximo durante a execução, pois todas as otimizações, alocação de registradores e verificações de tipo foram resolvidas previamente. Contudo, essa compilação é estática e gera um binário engessado a uma arquitetura específica de processador e sistema operacional.

Por outro lado, a **Interpretação Pura**, muito utilizada em linguagens de *script* antigas como as primeiras versões de Python, Ruby e PHP, não gera nenhum binário nativo. O programa (interpretador) lê o código fonte (ou uma forma intermediária, o *bytecode*) e executa as ações em tempo real. A grande vantagem é a portabilidade extrema e o início instantâneo de execução. A desvantagem fatal é o desempenho: o *overhead* do laço central do interpretador para decodificar instruções repetidamente torna a execução matematicamente ordens de grandeza mais lenta do que o código de máquina nativo.

1.3 O Paradigma Just-In-Time (JIT)

A compilação **Just-In-Time** surgiu com o objetivo ambicioso de unir a portabilidade da interpretação ao alto desempenho da compilação AOT.

Em uma arquitetura JIT, o código fonte é inicialmente traduzido para um formato independente de máquina (*Bytecode*). Quando o programa é executado, a Máquina Virtual (como a JVM do Java) invoca um interpretador veloz para começar a rodar o código instantaneamente. No entanto, o sistema trabalha paralelamente como um observador.

A compilação completa para máquina ocorre estritamente *em tempo de execução*. A regra empírica do 90/10 afirma que 90% do tempo de CPU de um programa é gasto em 10% do código (geralmente os laços de repetição densos). Compilar o código inteiro atrasaria a execução, portanto o JIT se concentra exclusivamente nesses "pontos quentes" (*Hot spots*), compilando-os para código de máquina nativo e substituindo-os na memória de forma transparente.

1.3.1 Arquitetura e Profiling (Tiered Compilation)

Para encontrar os *Hot spots*, o ambiente JIT incorpora um módulo chamado **Profiler**. O Profiler conta quantas vezes uma instrução, laço ou método é executado.

Sistemas modernos como o JVM HotSpot e o motor V8 do Google Chrome (JavaScript) não utilizam um único compilador JIT, mas escalam o esforço em um sistema conhecido como **Tiered Compilation** (Compilação em Níveis):

- **Tier 0 (Interpretador):** Todo código começa aqui. O tempo de inicialização é zero, mas a execução é lenta. O Profiler acopla contadores de execução.
- **Tier 1 (JIT Base):** Se uma função for executada milhares de vezes (por exemplo, 1.000 chamadas), ela atinge um limiar (*threshold*). O JIT Base (como o C1 no Java) é acionado. Ele compila o Bytecode rapidamente para Assembly, aplicando o mínimo de otimizações. O objetivo é remover o custo da interpretação o mais rápido possível.
- **Tier 2 (JIT Otimizador):** Se a função continuar a ser chamada repetidamente e atingir dezenas de milhares de chamadas (estiver "fervendo"), o ambiente aciona o JIT Otimizador (como o C2 no Java ou o TurboFan no V8). Este compilador gasta preciosos segundos rodando pesadas heurísticas (Análise de Fluxo de Dados, Escalonamento de

Registradores) para gerar um Assembly altamente eficiente, rivalizando com o C++ mais veloz.

1.4 Otimização Especulativa e Polimorfismo

É comum questionar como um JIT rodando em tempo real pode vencer um compilador C++ AOT que analisa o código por minutos. A resposta reside na **Otimização Especulativa**.

O compilador AOT possui apenas informações estáticas do código; ele precisa garantir corretude absoluta para qualquer entrada teórica concebível. O compilador JIT, no entanto, *assiste* a execução do programa e percebe os **padrões reais** do usuário através do Profiler.

O maior desafio das linguagens modernas Orientadas a Objetos é o Polimorfismo (chamada virtual de métodos). O C++ precisa olhar em uma Tabela Virtual (*vtable*) na RAM a cada invocação de objeto, bloqueando otimizações críticas como o *Inlining*.

```
1 Animal a = obterAnimal();
2 a.fazerSom(); // Pode ser Cachorro, Gato ou Pato.
```

Listing 1.1: O Paradoxo do Inlining em O.O.

O JIT, por sua vez, observa o comportamento. Se, após minutos de execução, o método `obterAnimal()` só retornou objetos do tipo `Cachorro`, o JIT **especula** que o programa sempre se comportará assim. Ele realiza um **Polymorphic Inline Cache (PIC)**, injetando agressivamente o código de máquina do `Cachorro` de forma direta (*Inlining*), precedido apenas de uma rápida condicional (`if (a.tipo == CACHORRO)`), ignorando a pesada tabela virtual.

1.4.1 Desotimização (Bailout) e OSR

Se a especulação for excessiva, o JIT prevê o que fazer caso o padrão mude. Suponha que o usuário gere um objeto `Gato`.

A instrução condicional inicial no código nativo falhará. Essa falha é chamada de **Guard Failure**. Num ambiente estático, tal falha resultaria em corrupção de memória. No ambiente JIT, ocorre a **Desotimização** (*Deoptimization* ou *Bailout*). A máquina virtual intercepta a execução nativa, invalida e destrói o código de máquina Otimizado do Tier 2, e "rebaixa" a função para rodar seguramente no lento Interpretador (Tier 0). Posteriormente, ela compilará uma nova versão que compreenda Gatos.

Por fim, há situações extremas onde a função pesada é um gigantesco laço `while` infinito que nunca retorna. O JIT não pode esperar o laço acabar para trocar a execução do interpretador para o código nativo recém-gerado. Para resolver isso, utiliza-se a técnica de **On-Stack Replacement (OSR)**. O JIT pausa a thread, captura todas as variáveis espalhadas pela memória do Interpretador, traduz e injeta esses valores de estado *diretamente* nos registradores da arquitetura da CPU, realizando um salto (*Jump*) preciso para o meio da versão nativa do código compilado, ocorrendo a substituição "com o avião em pleno voo".