Compiladores

Aula 26: Tópicos Avançados - JIT

Prof. Aléssio Miranda Júnior
alessio@cefetmg.br

CEFET-MG - Campus Timoteo
Dep. Engenharia de Computação

Fevereiro de 2026

Sumário

1. Introdução: AOT vs Interpretação
2. Arquitetura e Profiling
3. A Mágica da Otimização Especulativa
4. Bailout e On-Stack Replacement
5. Conclusão

O Espectro Clássico de Execução

Compilação AOT (Ahead-of-Time)

C, C++, Rust, Go

O compilador gasta muito tempo analisando o código inteiro antes da execução para gerar um binário nativo hiper-otimizado.

+ Desempenho máximo.
- Arquitetura engessada (precisa compilar para Windows, Mac, Linux separadamente).

Interpretação Pura

Python (CPython), Ruby, PHP

Não há binário nativo. Um programa (o Interpretador) lê o código fonte ou bytecode instrução por instrução durante a execução.

+ Início instantâneo (*Fast Startup*), portabilidade extrema.
- Muito lento! Gigantesco *overhead* no laço de interpretação.

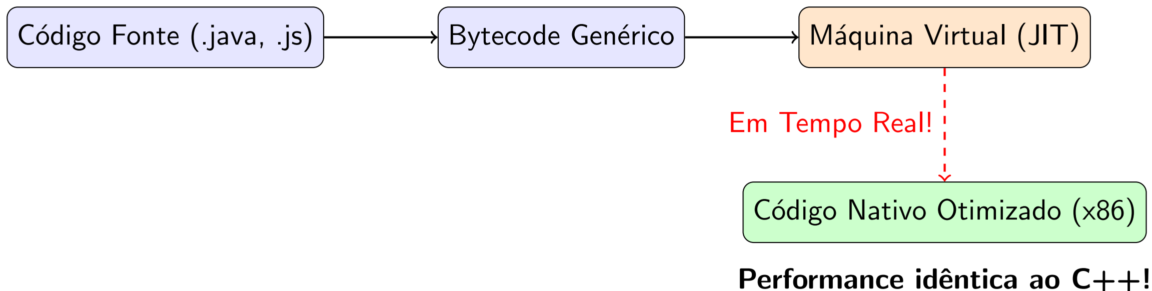
O Paradigma JIT (Just-In-Time)

Objetivo: Combinar a portabilidade do Interpretador com o desempenho feroz do AOT.

Executa na hora, portável, mas lento.

O Paradigma JIT (Just-In-Time)

Objetivo: Combinar a portabilidade do Interpretador com o desempenho feroz do AOT.



Como o JIT decide o que compilar?

Compilar código de verdade gasta RAM, tempo de CPU e bateria. Compilar um script enorme de Javascript do zero **congelaria** a página web por vários segundos!

A Regra do 90/10

Apenas 10% do código de um programa executa em 90% do tempo de CPU (geralmente laços de repetição). O restante do código só roda uma vez (inicialização) ou nunca roda (tratamento de erros).

Como o JIT decide o que compilar?

Compilar código de verdade gasta RAM, tempo de CPU e bateria. Compilar um script enorme de Javascript do zero **congelaria** a página web por vários segundos!

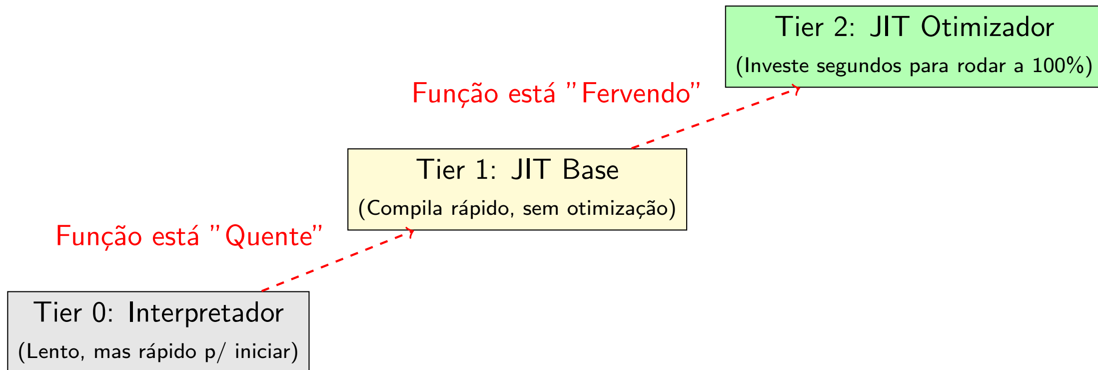
A Regra do 90/10

Apenas 10% do código de um programa executa em 90% do tempo de CPU (geralmente laços de repetição). O restante do código só roda uma vez (inicialização) ou nunca roda (tratamento de erros).

O JIT atua de forma **preguiçosa** (*lazy*): Ele joga o código inicial para rodar velozmente no Interpretador e embute um "espião" (**Profiler**) para contar quais funções são as mais usadas.

Tiered Compilation (A Escada de Níveis)

Sistemas como o **JVM HotSpot** e o **Chrome V8** não compilam tudo de uma vez, eles escalam o código em níveis de agressividade:



Por que o JIT pode bater o C++ nativo?

Muitos *benchmarks* modernos mostram o Java rodando mais rápido que binários C++ estáticos. Como um compilador "apressado" em tempo real bate um compilador que analisou o código por horas?

Por que o JIT pode bater o C++ nativo?

Muitos *benchmarks* modernos mostram o Java rodando mais rápido que binários C++ estáticos. Como um compilador "apressado" em tempo real bate um compilador que analisou o código por horas?

Resposta: O compilador C++ tem que adivinhar o comportamento do usuário e garantir segurança absoluta para todos os casos teóricos. O compilador JIT está **assistindo** o programa enquanto ele roda!

Otimização Especulativa

Baseado nos dados do *Profiler*, o JIT assume que o estado atual do programa será eterno. Ele cria "atalhos" agressivos que o AOT jamais teria coragem de fazer por medo de quebrar o programa num caso atípico.

O Desafio do Polimorfismo e Inlining

Em linguagens modernas OO (Java, C#), a chamada de métodos é sempre virtual (resolução dinâmica).

```
// Código AOT (C++) não
// tem coragem de fazer Inlining
Animal a = obterAnimal();
a.fazerSom();
// A variável 'a' pode ser Cachorro,
// Gato, Pato... A CPU precisa
// travar, buscar na Tabela Virtual
// da memória e pular -> Lento.
```

O Desafio do Polimorfismo e Inlining

Em linguagens modernas OO (Java, C#), a chamada de métodos é sempre virtual (resolução dinâmica).

```
// Código AOT (C++) não
// tem coragem de fazer Inlining
Animal a = obterAnimal();
a.fazerSom();
// A variável 'a' pode ser Cachorro,
// Gato, Pato... A CPU precisa
// travar, buscar na Tabela Virtual
// da memória e pular -> Lento.
```

```
// O JIT assiste e o Profiler anota:
// "Há 5 minutos o usuário só
// retorna Cachorro daqui".

// JIT especula e altera o Assembly:
if (a.tipo == CACHORRO) {
    print("Au au"); // INLINING DIRETO!
} else {
    // E se não for? (Bailout)
}
```

Polymorphic Inline Cache (PIC): O JIT elimina a terrível busca dinâmica na memória em troca de um rápido `if` de CPU e injeta as instruções diretamente na veia!

Desotimização (Deoptimization / Bailout)

O que acontece se a especulação hiper-agressiva do JIT falhar? Se, após 1 milhão de iterações certinhas, o usuário de repente instanciar um `Gato`?

- O bloco `if (a.tipo == CACHORRO)` emitirá **Falso**. Isso aciona uma **Guard Failure** (Falha da Guarda de Segurança) injetada pelo JIT.

Desotimização (Deoptimization / Bailout)

O que acontece se a especulação hiper-agressiva do JIT falhar? Se, após 1 milhão de iterações certinhas, o usuário de repente instanciar um `Gato`?

- O bloco `if (a.tipo == CACHORRO)` emitirá **Falso**. Isso aciona uma **Guard Failure** (Falha da Guarda de Segurança) injetada pelo JIT.
- Se isso fosse C++, a Otimização Incorreta causaria corrupção de memória ou *Segfault*.

Desotimização (Deoptimization / Bailout)

O que acontece se a especulação hiper-agressiva do JIT falhar? Se, após 1 milhão de iterações certinhas, o usuário de repente instanciar um `Gato`?

- O bloco `if (a.tipo == CACHORRO)` emitirá **Falso**. Isso aciona uma **Guard Failure** (Falha da Guarda de Segurança) injetada pelo JIT.
- Se isso fosse C++, a Otimização Incorreta causaria corrupção de memória ou *Segfault*.
- No JIT, a falha dispara um mecanismo de pânico: o sistema pausa o programa, **joga o código nativo no lixo** e rebaixa a função de volta para o lento **Interpretador** (Tier 0).

Desotimização (Deoptimization / Bailout)

O que acontece se a especulação hiper-agressiva do JIT falhar? Se, após 1 milhão de iterações certinhas, o usuário de repente instanciar um `Gato`?

- O bloco `if (a.tipo == CACHORRO)` emitirá **Falso**. Isso aciona uma **Guard Failure** (Falha da Guarda de Segurança) injetada pelo JIT.
- Se isso fosse C++, a Otimização Incorreta causaria corrupção de memória ou *Segfault*.
- No JIT, a falha dispara um mecanismo de pânico: o sistema pausa o programa, **joga o código nativo no lixo** e rebaixa a função de volta para o lento **Interpretador** (Tier 0).
- O processo continua perfeitamente seguro e sem *crash*. Após um tempo, o JIT recompilará a função anotando a presença rara do Gato.

On-Stack Replacement (OSR)

Há um problema extremo: e se o código que o JIT está observando for um **laço infinito** gigantesco e a função nunca retornar? Como substituir a execução lenta pelo código de máquina se estamos presos *dentro* do laço?

- Ele usa a técnica insana chamada **On-Stack Replacement (OSR)**.
- O JIT "pausa" a thread no meio da instrução, mapeia as dezenas de variáveis locais soltas da RAM do Interpretador, encaixa-as matematicamente nos registradores nativos (`EAX`, `EBX`) e **dá um pulo (jump)** diretamente para o meio do código de máquina recém-compilado!

Conclusão e Legado

- O JIT é a culminação de todo o estudo de compiladores. Ele é um monstro que engloba Analisador Léxico, Sintático, geração de IR, Análise de Fluxo e Grafo de Otimizações operando e mutando **em milissegundos enquanto o programa interage com o usuário**.
- A web moderna só existe por causa da Guerra dos Browsers em 2008, onde o Google Chrome lançou o motor V8 introduzindo o JIT para Javascript, acelerando-o em 100x e permitindo aplicações pesadas como o Gmail ou Figma rodarem via browser.