Just In Time Compilation

Slides Adapted from Prof Oscar Niersrasz
Overview

> What is Just-In-Time Compilation (JIT)?
> History of JIT
> JIT Overhead
> Optimization Techniques in JIT
## Compilation vs Interpretation

<table>
<thead>
<tr>
<th>Compilation</th>
<th>Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Pros</strong></td>
<td><strong>Pros</strong></td>
</tr>
<tr>
<td>&gt; Programs run faster</td>
<td>&gt; Programs are typically smaller</td>
</tr>
<tr>
<td><strong>Cons</strong></td>
<td><strong>Cons</strong></td>
</tr>
<tr>
<td>&gt; Compilation overhead</td>
<td>&gt; Programs tend to be more portable</td>
</tr>
<tr>
<td>&gt; Programs are typically bigger</td>
<td>&gt; Access to run-time information</td>
</tr>
<tr>
<td>&gt; Programs are not portable</td>
<td>&gt; Programs run slower</td>
</tr>
<tr>
<td>&gt; No run-time information</td>
<td></td>
</tr>
</tbody>
</table>
JVM and Common Language Runtime

Virtual Machine: a software execution engine for a program written in a machine-independent language – Ex., Java bytecodes, CLI, Pascal p-code, Smalltalk v-code

Step 1: syntax analysis and generate intermediate code, e.g., bytecode

Step 2: Interpret/compile the code on the VM (managed runtime) for portability, better safety checks

Microsoft Common Language Runtime: The immediate language can be shared by multiple source languages (e.g., C# and managed C++)

CIL is a CPU- and platform-independent instruction set.
The Common Language Runtime (CLR) manages the execution of code.

CLR uses Just-In-Time (JIT) compiler to compile the CIL code to the native code for device used.

Through the runtime compilation process CIL code is verified for safety during runtime, providing better security and reliability than natively compiled binaries.
What is Just-In-Time Compilation?

**Dynamic Translation:** Compilation done during execution of a program – *at run time* – rather than prior to execution.
A Typical JIT Compilation Model?

- First time method is called the IL is compiled and optimized
- Compiled machine code is cached in transient memory
- Cached copy used for subsequent calls
What is Just-In-Time Compilation?

- Code generation component of a virtual machine
- Compiles bytecodes to in-memory binary machine code
  - Simpler front-end and back-end than traditional compiler
    - Not responsible for source-language error reporting
    - Doesn't have to generate object files or relocatable code
- Compilation is interspersed with program execution
  - Compilation time and space consumption are very important
- Compile program incrementally; unit of compilation is a method
  - JIT may never see the entire program
  - Must modify traditional notions of IPA (Interprocedural Analysis)
Why Just-In-Time Compilation?

Preserve Safety and Reusability
Translation Time of JIT Compiler + Execution Time of Native Code <= Running Time on JVM (Improve time and space efficiency of programs utilizing)

> portable and space-efficient byte-code
> run-time information → feedback directed optimizations
> Speculative optimization
Why Just-In-Time Compilation?
Outline

> What is Just-In-Time Compilation (JIT)?
> History of JIT
> JIT Overhead
> Optimization Techniques in JIT
History of Just-In-time

First Just-In-Time

> 1960
> McCarthy's LISP paper about dynamic compilation

Fortran

> 1974
> Optimization of “hot spots”

Smalltalk

> 1980 – 1984
> Bytecode to native code translation
> First modern VM
History of Just-In-time

Self
> 1986 – 1994
> New Advanced VM techniques

Java
> 1995 – present
> First VM with mainstream market penetration

Android RunTime (ART)
> 2014
> No JIT ;-)
History of Just-In-time

1st generation JVM
  • Purely interpreting
  • 30 - 50 times slower than C++

2nd generation JVM
  • JIT compilers
  • 3 - 10 times slower than C++

Static compilers
  • Better performance than JIT’s
Outline

- What is Just-In-Time Compilation (JIT)?
- History of JIT
- JIT Overhead
- Optimization Techniques in JIT
Just-In-Time Overhead

JIT: 4x speedup, but 20x initial overhead

Short Running tasks

Long Running tasks

Matthew Arnold, Stephen Fink, David Grove, and Michael Hind, ACACES'06, 2006

Dynamic Compilation and Adaptive Optimization in Virtual Machines
Selective Optimization

> Start program in interpreted mode
> Find “hot spots”
> compile only hot spots
Selective Optimization

- **JIT1, JIT2 and JIT3**: the better startup, the worse steady state performance.
- **Selective optimization with JIT3**: reaches best startup and best steady state performance

Matthew Arnold, Stephen Fink, David Grove, and Michael Hind, ACACES'06, 2006
Java Virtual Machine

> HotSpot
> server mode (-server)
  — aggressive and complex optimizations
  — slow startup
  — fast execution
> client mode (-client)
  — less optimizations
  — fast startup
  — slower execution
What To Optimize

> Method Counters
> Call Stack Sampling
What To Optimize: Method Counters

```java
public void foo() {
    fooCounter++;
    if (fooCounter > threshold) {
        recompile();
    }
}
```

> Approximation of time spent in each method
> Popular
> Might have significant overhead
What To Optimize: Call Stack Sampling

- Call stack inspected in regular intervals as the program is running
- Approximation of time spent in each method
- Not deterministic
Outline

- What is Just-In-Time Compilation (JIT)?
- History of JIT
- JIT Overhead
- Optimization Techniques in JIT
Optimization Techniques

- Loop Unrolling
- Register Allocation
- Global Code Motion
- Machine Code Generation
- Inlining
- Code Positioning
- Multi-Versioning
- Dynamic Class Hierarchy Mutation
Standard Techniques Revised

- Loop Unrolling
  - unroll “hot” loops only

- Register Allocation
  - assign register to “hot path” variables first

- Global Code Motion
  - move code from “hot” block

- Machine Code Generation
  - generate code for the particular architecture
Inlining (Pros & Cons)

> **Pros**
  — removes cost of a function call and return instruction
  — improves locality of code
  — once performed, additional optimizations can become possible

> **Cons**
  — increases code size
  — may degrade performance (code size overflows cache)
Speculative Inlining

```java
for (Shape shape : shapes) {
    shape.computeArea();
}
```

> Inline `Circle.computeArea()`
> Monitor class hierarchy
> Recompile if `Shape` has more subclasses
On Stack Replacement (OSR)

- Multiple implementation of the same function (optimized version, unoptimized version, interpreted version)

- When OSR occurs, the VM is paused, and the stack frame for the target function is replaced by an equivalent frame which may have variables in different locations.

```java
for (Shape shape : shapes) {
    area = ((Circle)shape).r() * pi^2;
}
```

Square appears in the shapes.
We cannot wait for loop to finish.
On Stack Replacement Applications

- Invalidation of speculative optimization
- De-optimization for debugging
- Runtime optimization of long-running activations
Multiversioning

- Multiple implementations of a code
- The best implementation is chosen at runtime

```java
for (Shape shape : shapes) {
    area = shape.area();
}
```
Code Positioning

- Linearizes the most common path
- Improves code locality
- Eliminates jumps
- Improves cache performance
Inline Caches (ILC)

> speed up runtime method binding

> based on the empirical observation that the objects that occur at a particular call site are often of the same type

> Improves performance by remembering the result of previous method lookup at the call site.
Inline Caches (ILC)

> Monomorphmic inline cache: worst case scenario

```javascript
var values = [1, "a", 2, "b", 3, "c", 4, "d"];
for (var i in values) {
    document.write(values[i].toString());
}
```

> Polymorphic inline cache:

- better deal with call sites that frequently see a limited number of different types
- multiple method lookup results can be recorded at the same call site
- a jump table which consists of a preamble that derives the type of the receiver and a series of constant compares and conditional jumps
Instruction Scheduling

> Improves Performance with instruction pipelines
> Heavily dependent on underlying architecture

```plaintext
load r1
load r2
add r3
```