Code Transformation and Optimization - Notes

260236

March 2024

Preface

Every theory section in these notes has been taken from two sources:

- None

About:

- GitHub repository

Contents

1 Introduction

1.1 Why, When, What, Where?

The introduction to this course begins with some basic questions.

Why compiling?

Although it's a trivial question, the reason is not so clear. You could choose interpreters in spite of compilers. Well, actually the **goal of an interpreter is to read one statement/line of code at a time and perform the specified actions**.

⚠ Disadvantage

There is a **short delay before execution starts**.

✓ Advantages

We can do **interactive execution** (in other words, we can execute partially written code). Also, the interpreter **avoids compilation overhead if the code is not executed**.

In contrast, the **goal of a compiler is to translate a compilation unit into machine code**.

✓ Advantages

Optimizes across different statements; **faster execution** once the code is compiled; finally, the **compilation needs to be performed only once**.

When to compile?

There are several types of compilers, each type depending on when to compile:

- Before the execution - Static Compiler. **Translates source code into machine code well before execution, possibly even on a different machine**. There is no compilation overhead at run-time. However, there is no way to adapt the code at runtime, and a version of the code must be generated for each target platform.

- **During the execution** - **Just-In-Time (JIT) Compiler**. **Translates each function the first time it is called**. Only code that is actually executed is translated. It also targets the code for the specific architecture, possibly taking into account runtime constraints (e.g. availability of resources). Finally, the code may be optimized taking into account runtime information (e.g. runtime constants).

- Mix - **Ahead-Of-Time (AOT) Compiler**. **Translates each function before it is first invoked**. It attempts to mix the two above styles to reap the benefits of both. It's popular with virtual machines (e.g. DotNet, Java).

What to compile?

It depends on the **compilation units**. They can be:

- A **statement** or a **line of code**. In other words, small regions of code.
- A **function**/**procedure**. Medium size code snippets that can be reused by calling a function.
- **Module**/**source file**. Large pieces of code.

There is also a **trade-off** with compilation units. In fact, a **smaller** compilation unit has a faster translation and can be used to provide interactive compilation and execution. But the **larger** compilation units may require more optimization and have more complexity.

Where to compile?

It depends on the resources available. Exists in two ways:

- **Cross-Compilation**. Compile on machine H and producing code for machine T. This is generally useful when machine T has insufficient resources for handling compilation task (e.g. embedded micro-controllers) or is not suited for general purpose processing (e.g. GPGPU). Also useful for distributing to different platforms while having a single build machine.
- **Split-Compilation**. Perform part of the compilation on machine H, producing an intermediate, portable code (e.g. bytecode), and part on machine T. Allows target-specific optimization to be performed more easily. Simplifies distribution of code on wide ranges of different target platforms.

References

Index