Just-in-time compilation

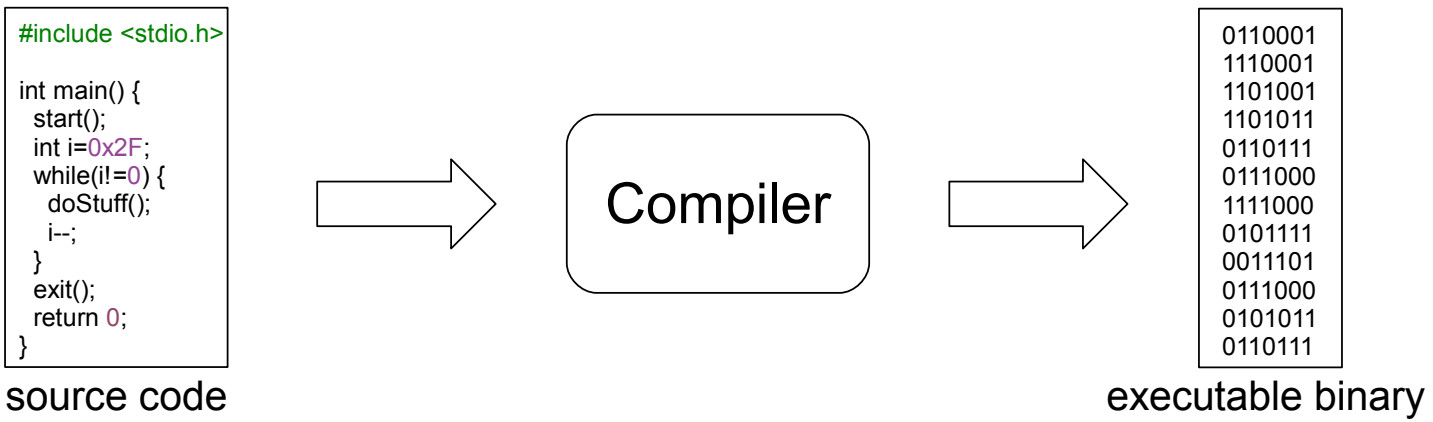
Kai Frerich

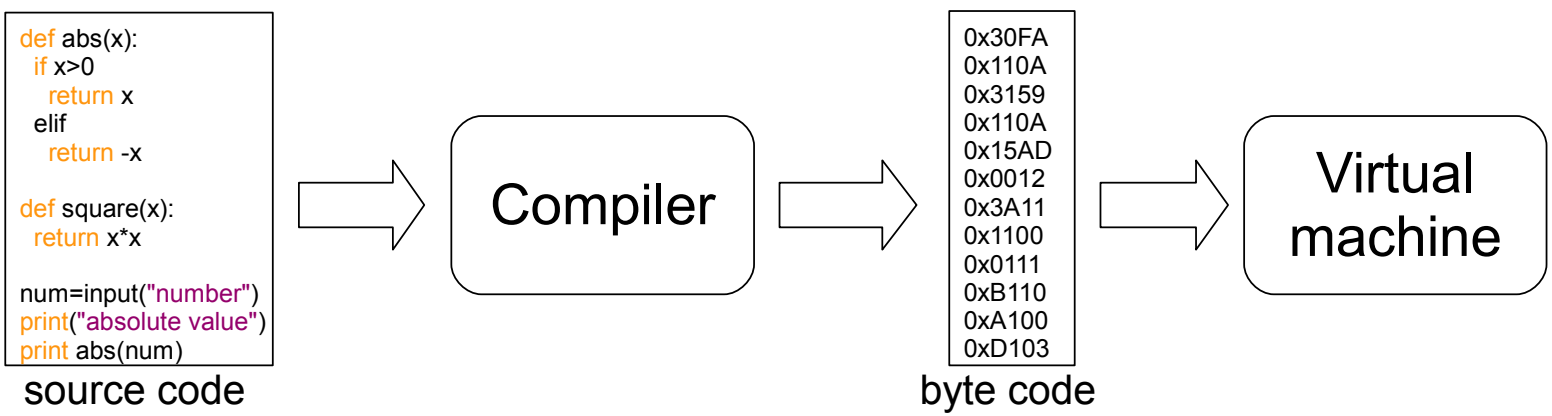
Seminar on Languages for Scientific Computing
Rheinisch-Westfälische Technische Hochschule Aachen

22. November 2012

Overview

Ways of running programs on a computer

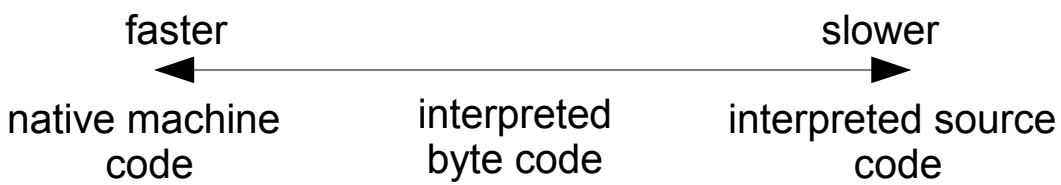




JIT compilation

- translation of code after program has started
- mostly referring to compilation to native code

Reason to do it: gaining a performance boost



What code parts to compile?

First idea: compile whole program at startup
problems:

- much CPU time needed to compile at start up
- huge memory usage
- doing possibly unnecessary work

Observation on many programs: most time is spent executing small parts of the code(hot parts)
Task for JIT compilation: find these hot parts

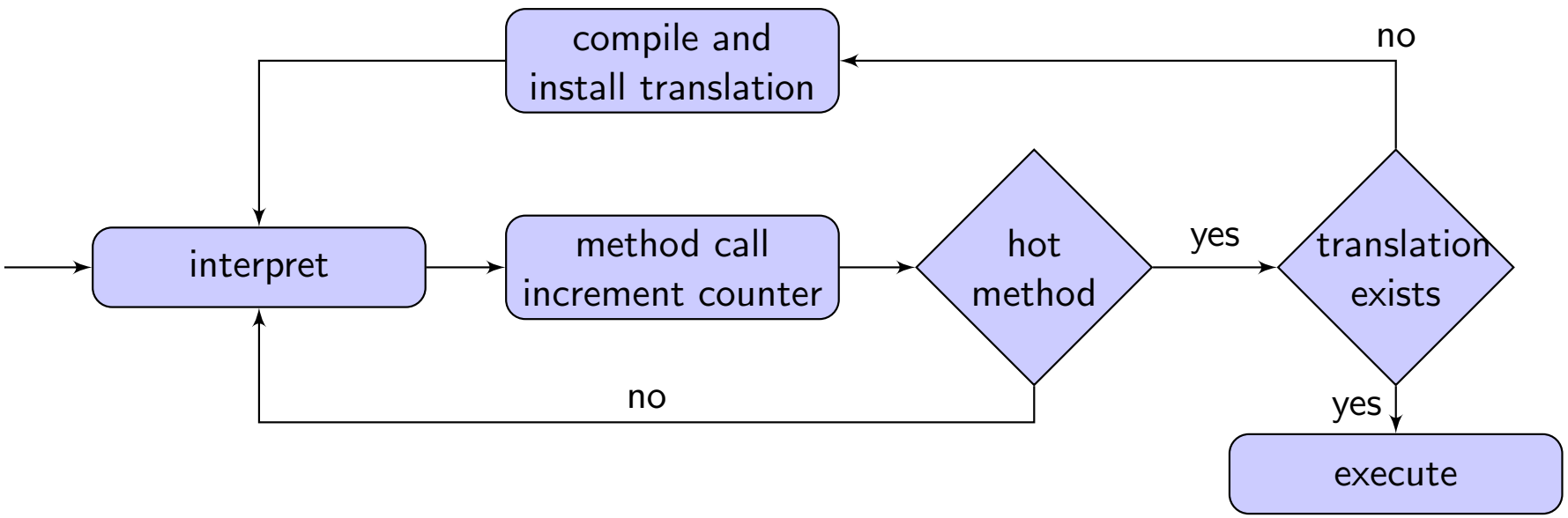
Find good trade off between compilation time and runtime improvements

Byte or source code often consists of numerous methods.

This leads to a basic technique:

- find often executed methods
- translate these methods to native machine code and execute it instead of interpreting the methods again

Compile the most executed methods



problem: hot methods can contain rarely used code

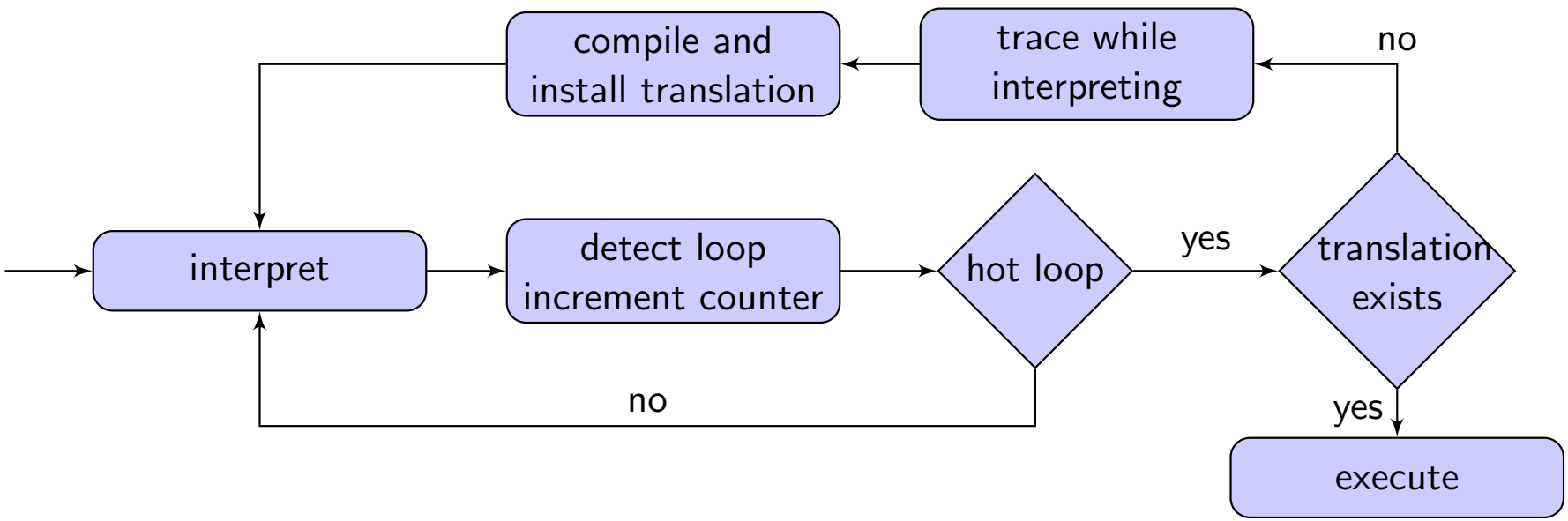
```
1  public void doStuff(String arg) {
2    if(arg=="") { //first error case
3    ..error solving code..
4    }
5    else if(arg.length()>BUFFER_SIZE) { //second error case
6    ..error solving code..
7    }
8    ..actually make something..
9  }
```

Assumptions:

- programs spend most of the time in loops
- in most iterations similar sequences of code(trace) are executed

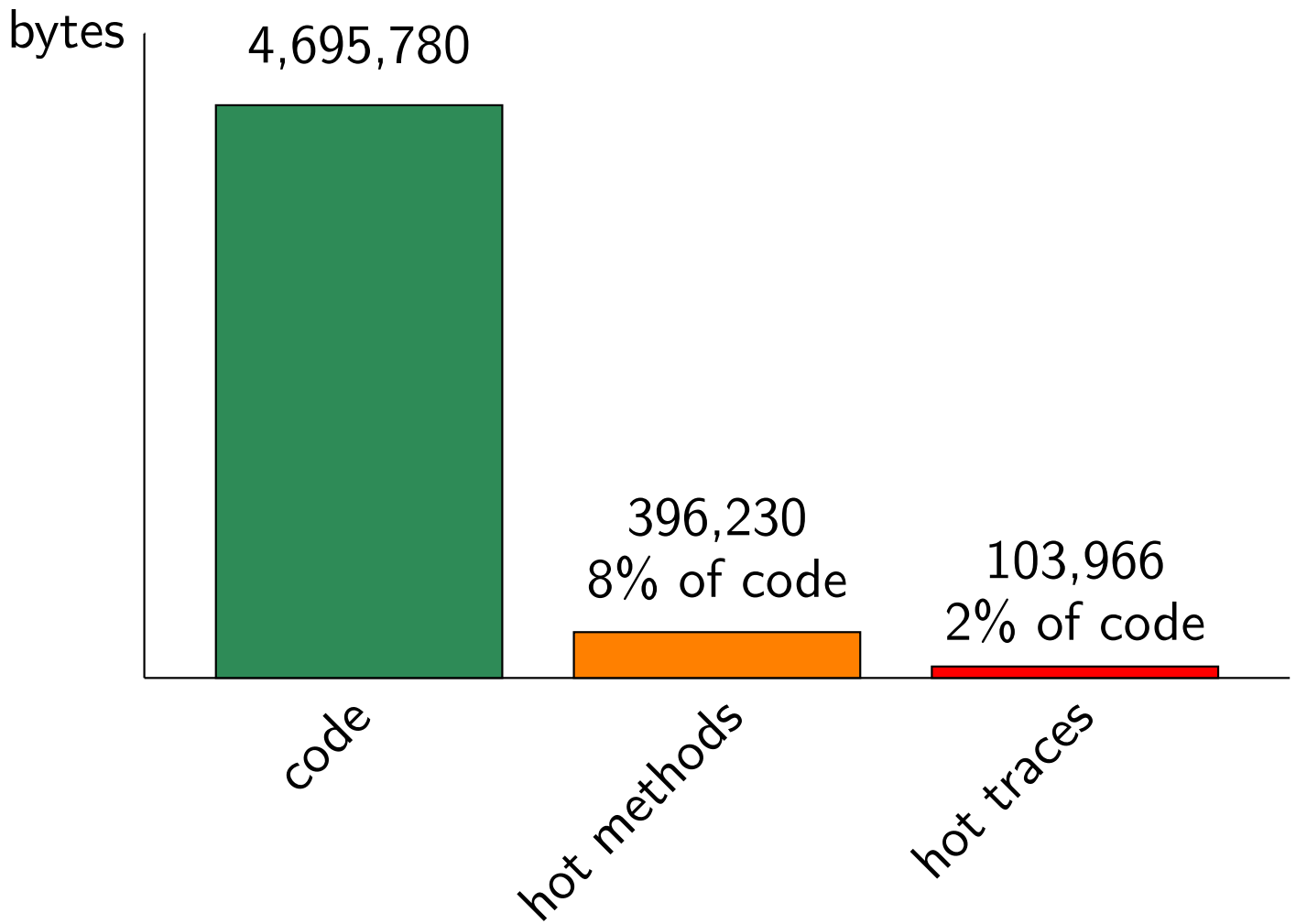
Find traces and leave executing more rarely used code to the interpreter

Find and compile hot traces



Amount of hot code

Profiling system_server program:



Unoptimized native code not that fast but we need fast code!
This leads to a difficult situation:

- optimizing the emitted native code benefits the programs performance
- applying optimizations needs valuable CPU time during the runtime

Find good trade off between run time improvement, compilation and optimization time

Questions of applying optimizations often resolved like this:

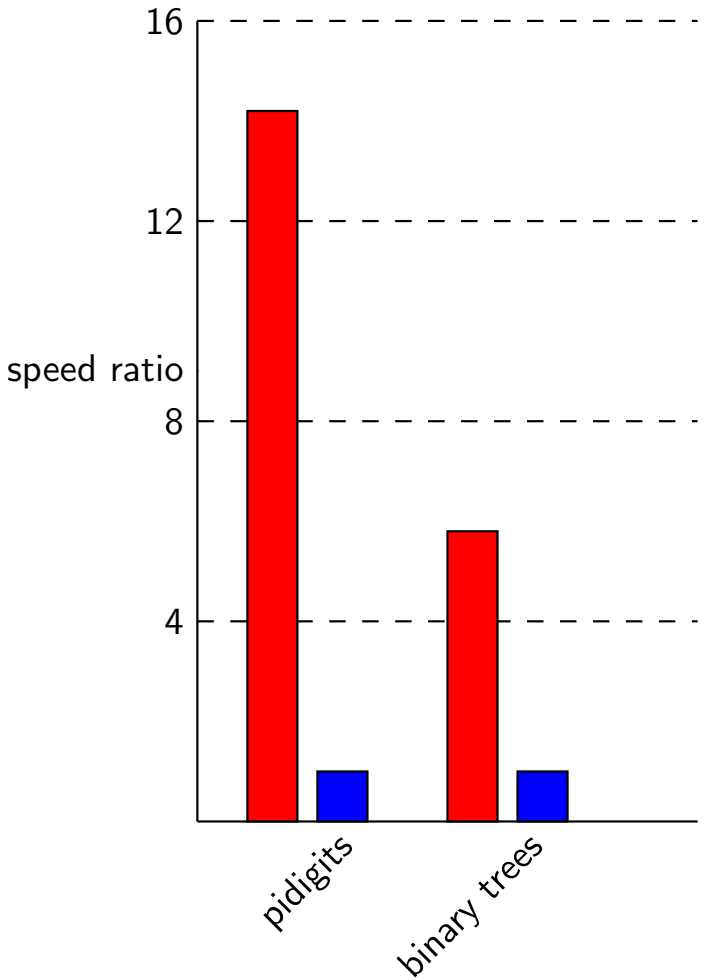
- if program needs much CPU time only do few optimizations and schedule further optimization to later time
- in periods where CPU is not busy scheduled code optimization can be done

By this proceeding delays at runtime are avoided and heavily optimized code can be gained after some time.

Advantages

- speed up for nearly every type of program (compared to pure interpreting)
- can even produce native code exactly fitting to the CPU (e.g. using instruction set extensions)
- most JIT implementations don't require activation by the user and run in the background

Lua vs. LuaJIT

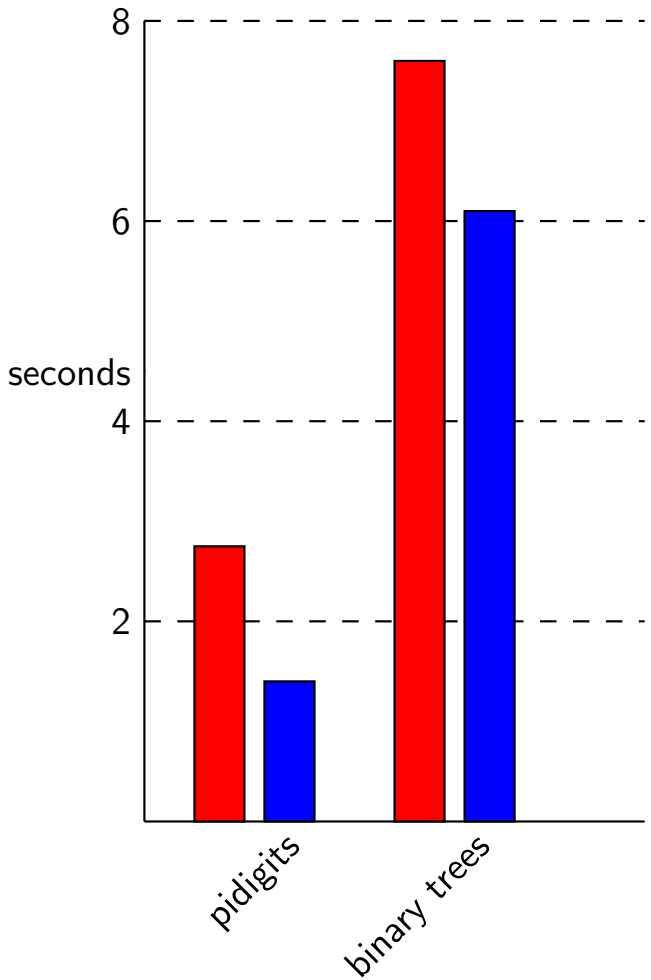


LuaJIT Lua source: luajit.org/performance_x86.html

Disadvantages

- interpreter and compiler parts both have to be developed and maintained
- security issues could allow executing of arbitrary code
- can cause short delays on start up of the program
- after all statically compiled programs use to be faster in the normal case

Java vs. C



Java C

source: shootout.alioth.debian.org/

HotSpot by Oracle

- most common Java virtual machine on desktop and server
- method based JIT compilation of Java byte code
- highly optimized
- historically developed because execution of Java programs was too slow

Dalvik by Google

- common virtual machine for Android systems
- tracing based JIT compilation
- optimized for mobile devices with small memory

Pypy

- faster than standard Python implementation CPython
- tracing based JIT compilation
- optimized for speed and compatibility with CPython
- written in Python

- interpreted or byte code interpreted languages nearly always benefit of using just-in-time compilation regarding execution speed
- if pure speed is needed and memory usage has to be low a statically compiled language should be used

references


J. Aycock: *A Brief History of Just-In-Time*, ACM Computing Surveys, Vol. 35, No. 2 (2003)


C. Rohlf and Y. Ivnitskiy *Attacking Clientside JIT Compilers*


B. Cheng and B. Buzbee *A JIT Compiler for Android's Dalvik VM* Google Tech Talk, San Francisco(2003)


Java vs. C *shootout.alioth.debian.org/*


Lua vs. LuaJIT *luajit.org/*

Thanks for the attention