

VM Compilation Strategies

Richard Warburton, Warwick University Computer Science
Department

June 9, 2008

Introduction

Definition

Coverage

Architectures

Limits

Architectures

High Level

Low Level

JIT Compilers

Definition

A software program that executes programs like a real machine.

- ▶ System Virtual Machines
- ▶ Process/Application virtual machine

Coverage

▶ Architecture Examples

Coverage

- ▶ Architecture Examples
- ▶ **Intermediate Representation**

Coverage

- ▶ Architecture Examples
- ▶ Intermediate Representation
- ▶ **Just In Time compilation**

Architectures

- ▶ High Level - JVM/CLR

Architectures

- ▶ High Level - JVM/CLR
- ▶ Low Level - LLVM

Architectures

- ▶ High Level - JVM/CLR
- ▶ Low Level - LLVM
- ▶ **Specialist - WAM/Valgrind**

Architectures

- ▶ High Level - JVM/CLR
- ▶ Low Level - LLVM
- ▶ Specialist - WAM/Valgrind
- ▶ PyPy

I won't be talking about ...

▶ Garbage Collection

I won't be talking about ...

- ▶ Garbage Collection
- ▶ **Specialist VMs**

I won't be talking about ...

- ▶ Garbage Collection
- ▶ Specialist VMs
- ▶ Focus on JVM/CLR

High Level

- ▶ High Level Languages (Java/C#)

High Level

- ▶ High Level Languages (Java/C#)
- ▶ simple AOT compiler

High Level

- ▶ High Level Languages (Java/C#)
- ▶ simple AOT compiler
- ▶ **Introspection (Reflection)**

High Level

- ▶ High Level Languages (Java/C#)
- ▶ simple AOT compiler
- ▶ Introspection (Reflection)
- ▶ Tool Apis (JVMTI)

High Level

- ▶ High Level Languages (Java/C#)
- ▶ simple AOT compiler
- ▶ Introspection (Reflection)
- ▶ Tool Apis (JVMTI)
- ▶ **Mature GC/JIT**

Intermediate Representation

▶ Binary format

see example

Intermediate Representation

- ▶ Binary format
- ▶ Strongly Typed Assembly (Bytecode verification)

see example

Intermediate Representation

- ▶ Binary format
- ▶ Strongly Typed Assembly (Bytecode verification)
- ▶ **Explicit Class Hierachy**

see example

Intermediate Representation

- ▶ Binary format
- ▶ Strongly Typed Assembly (Bytecode verification)
- ▶ Explicit Class Hierachy
- ▶ No expression/Statement seperation

see example

Intermediate Representation

- ▶ Binary format
- ▶ Strongly Typed Assembly (Bytecode verification)
- ▶ Explicit Class Hierachy
- ▶ No expression/Statement seperation
- ▶ Exception handling

see example

Intermediate Representation

- ▶ Binary format
- ▶ Strongly Typed Assembly (Bytecode verification)
- ▶ Explicit Class Hierachy
- ▶ No expression/Statement seperation
- ▶ Exception handling
- ▶ **Stack based expressions**

see example

LLVM

Disclaimer: LLVM is not just a VM

- ▶ lifetime optimisation approach

LLVM

Disclaimer: LLVM is not just a VM

- ▶ lifetime optimisation approach
 - ▶ Allows offline optimisation

LLVM

Disclaimer: LLVM is not just a VM

- ▶ lifetime optimisation approach
 - ▶ Allows offline optimisation
 - ▶ link time optimisation

LLVM

Disclaimer: LLVM is not just a VM

- ▶ lifetime optimisation approach
 - ▶ Allows offline optimisation
 - ▶ link time optimisation
 - ▶ runtime profiling

LLVM

Disclaimer: LLVM is not just a VM

- ▶ lifetime optimisation approach
 - ▶ Allows offline optimisation
 - ▶ link time optimisation
 - ▶ runtime profiling
 - ▶ sound with incomplete information

LLVM

Disclaimer: LLVM is not just a VM

- ▶ lifetime optimisation approach
 - ▶ Allows offline optimisation
 - ▶ link time optimisation
 - ▶ runtime profiling
 - ▶ sound with incomplete information
- ▶ Tiered VM architecture - VMkit

Intermediate Representation

- ▶ 3 isomorphic formats - disk text, bytecode, in-memory

Intermediate Representation

- ▶ 3 isomorphic formats - disk text, bytecode, in-memory
- ▶ **3-Address Representation**

Intermediate Representation

- ▶ 3 isomorphic formats - disk text,bytecode,in-memory
- ▶ 3-Address Representation
- ▶ SSA. with explicit phi nodes

Intermediate Representation

- ▶ 3 isomorphic formats - disk text,bytecode,in-memory
- ▶ 3-Address Representation
- ▶ SSA. with explicit phi nodes
- ▶ Exception handling support

Intermediate Representation

- ▶ 3 isomorphic formats - disk text, bytecode, in-memory
- ▶ 3-Address Representation
- ▶ SSA. with explicit phi nodes
- ▶ Exception handling support
- ▶ **Explicit Type, CFG and allocation information**

Intermediate Representation

- ▶ 3 isomorphic formats - disk text,bytecode,in-memory
- ▶ 3-Address Representation
- ▶ SSA. with explicit phi nodes
- ▶ Exception handling support
- ▶ Explicit Type, CFG and allocation information
- ▶ Allows arbitrary casting (ie weak typing to support unsafe languages)

Intermediate Representation

- ▶ 3 isomorphic formats - disk text,bytecode,in-memory
- ▶ 3-Address Representation
- ▶ SSA. with explicit phi nodes
- ▶ Exception handling support
- ▶ Explicit Type, CFG and allocation information
- ▶ Allows arbitrary casting (ie weak typing to support unsafe languages)
- ▶ Registers (Albeit infinite number)

Motivations

Architectural Motivations

- ▶ Interpreters = slower program execution than compilers

Language Motive

Motivations

Architectural Motivations

- ▶ Interpreters = slower program execution than compilers
- ▶ **compiler optimisations benefit from profiling**

Language Motive

Motivations

Architectural Motivations

- ▶ Interpreters = slower program execution than compilers
- ▶ compiler optimisations benefit from profiling

Language Motive

- ▶ dynamic properties don't suit static compilers

Motivations

Architectural Motivations

- ▶ Interpreters = slower program execution than compilers
- ▶ compiler optimisations benefit from profiling

Language Motive

- ▶ dynamic properties don't suit static compilers
- ▶ call graphs, aliasing, etc.

Motivations

Architectural Motivations

- ▶ Interpreters = slower program execution than compilers
- ▶ compiler optimisations benefit from profiling

Language Motive

- ▶ dynamic properties don't suit static compilers
- ▶ call graphs, aliasing, etc.
- ▶ **see example**

Just In Time?

Solution:

Just In Time (JIT) = Compilation during program execution

Comparison with AOT

Opportunities:

- ▶ use of runtime information

Comparison with AOT

Opportunities:

- ▶ use of runtime information
- ▶ bytecode IR \therefore no need for complex parsers (C++) or type inference (ML)

Comparison with AOT

Opportunities:

- ▶ use of runtime information
- ▶ bytecode IR \therefore no need for complex parsers (C++) or type inference (ML)

Comparison with AOT

Opportunities:

- ▶ use of runtime information
- ▶ bytecode IR \therefore no need for complex parsers (C++) or type inference (ML)

A Problem ! ...

Comparison with AOT

Opportunities:

- ▶ use of runtime information
- ▶ bytecode IR \therefore no need for complex parsers (C++) or type inference (ML)

A Problem ! ... Computational Cost

Early JITs

Idea: JIT @ load time

- ▶ Long startup times

Early JITs

Idea: JIT @ load time

- ▶ Long startup times
- ▶ Poor runtime performance (little runtime information used)

Early JITs

Idea: JIT @ load time

- ▶ Long startup times
- ▶ Poor runtime performance (little runtime information used)
- ▶ Much code occasionally used \therefore wasted compilation time

Mature JITs

Idea: JIT only after profiling

- ▶ Method Invocation counters

Mature JITs

Idea: JIT only after profiling

- ▶ Method Invocation counters
- ▶ track caller/callee relationships

Mature JITs

Idea: JIT only after profiling

- ▶ Method Invocation counters
- ▶ track caller/callee relationships
- ▶ Identify Hotspots (90/10)

Mature JITs

Idea: JIT only after profiling

- ▶ Method Invocation counters
- ▶ track caller/callee relationships
- ▶ Identify Hotspots (90/10)
- ▶ enable optimisation opportunities - inlining, specialisation

Mature JITs

Idea: JIT only after profiling

- ▶ Method Invocation counters
- ▶ track caller/callee relationships
- ▶ Identify Hotspots (90/10)
- ▶ enable optimisation opportunities - inlining, specialisation
- ▶ **Deoptimization - "the process of changing an optimized stack frame to an unoptimized one"**

Optimizations

- ▶ Discuss specific Hotspot Optimisations

The End

Any Questions?