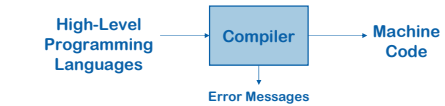


Dynamic Compilation and Optimization

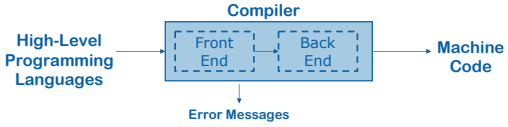
CS 471
December 3, 2007



So Far... Static Compilation



Digging Deeper...



1 CS 471 – Fall 2007

Alternatives to the Traditional Model

Static Compilation
All work is done “ahead-of-time”

Just-in-Time Compilation
Postpone some compilation tasks

Multiversioning and Dynamic Feedback
Include multiple options in binary

Dynamic Binary Optimization
Traditional compilation model
Executables can adapt

2 CS 471 – Fall 2007

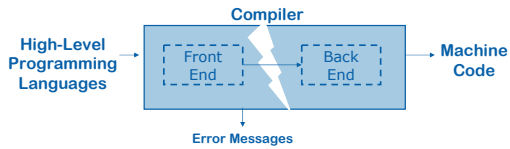
Move More of Compilation to Run Time

Execution environment may be quite different from the assumptions made at compile time

- Dynamically loaded libraries
- User inputs
- Hardware configurations
- Dependence on software vendors
- Apps on tap
- Incorporate profiling

3 CS 471 – Fall 2007

Just-in-Time Compilation



Ship bytecodes (think IR) rather than binaries

- Binaries execute on machines
- Bytecodes execute on *virtual* machines

4 CS 471 – Fall 2007

Just-in-Time Compilation

`source` → `javac` → `bytecode`

javac the Java bytecode compiler

`bytecode` → `java` → `execute`

java the Java virtual machine

Bytecode: machine independent, portable

Step One: “Compile” `Circle.java`
`% javac Circle.java -> Circle.class`

Step Two: “Execute”
`% java Circle.class`

5 CS 471 – Fall 2007

Bytecodes

Each frame contains *local variables* and an *operand stack*

Instruction set

- Load/store between locals and operand stack
- Arithmetic on operand stack
- Object creation and method invocation
- Array/field accesses
- Control transfers and exceptions

The **type of the operand stack** at each program point is known at compile time

6

CS 471 – Fall 2007

Bytecodes (cont.)

Example:

```
iconst 2
iload a
iload b
iadd
imul
istore c
```

Computes: $c := 2 * (a + b)$

7

CS 471 – Fall 2007

Bytecodes (cont.)

Example:

```
iconst 2
iload a
iload b
iadd
imul
istore c
```

a	42
b	7
c	0

Computes: $c := 2 * (a + b)$

8

CS 471 – Fall 2007

Bytecodes (cont.)

Example:

```
iconst 2
iload a
iload b
iadd
imul
istore c
```

a	42
b	7
c	0

Computes: $c := 2 * (a + b)$

9

CS 471 – Fall 2007

Bytecodes (cont.)

Example:

```
iconst 2
iload a
iload b
iadd
imul
istore c
```

a	42
b	7
c	0

Computes: $c := 2 * (a + b)$

10

CS 471 – Fall 2007

Bytecodes (cont.)

Example:

```
iconst 2
iload a
iload b
iadd
imul
istore c
```

a	42
b	7
c	0

Computes: $c := 2 * (a + b)$

11

CS 471 – Fall 2007

Bytecodes (cont.)

Example:

```

iconst 2
iload a
iload b
iadd
imul
istore c

```

a	42	
b	7	
c	0	49
		2

Computes: $c := 2 * (a + b)$

12 CS 471 - Fall 2007

Bytecodes (cont.)

Example:

```

iconst 2
iload a
iload b
iadd
imul
istore c

```

a	42	
b	7	
c	0	98

Computes: $c := 2 * (a + b)$

13 CS 471 - Fall 2007

Bytecodes (cont.)

Example:

```

iconst 2
iload a
iload b
iadd
imul
istore c

```

a	42	
b	7	
c	98	

Computes: $c := 2 * (a + b)$

14 CS 471 - Fall 2007

Executing Bytecode

`java Circle.class` - What happens?

Interpreting

- map each bytecode to a machine code sequence,
- for each bytecode, execute the sequence

Translation to machine code

- map all the bytecodes to machine code (or a higher level intermediate representation)
- massage them (e.g., remove redundancies)
- execute the machine code

15 CS 471 - Fall 2007

Hotspot Compilation

A hybrid approach

- Initially interpret
- Find the "hot" (frequently executed) methods
- Translate only hot methods to machine code

16 CS 471 - Fall 2007

The Virtual Machine

An extreme version of an old idea

Previously:

Now:

17 CS 471 - Fall 2007

Compile-Time Multiversioning

- Multiple **versions** of code sections are generated at compile-time
- Most appropriate variant is **selected** at runtime based upon characteristics of the **input data** and/or machine **environment**
- Multiple variants can cause code explosion
 - Thus typically only a few versions are created

18 CS 471 – Fall 2007

Another Alternative

Optimize a *traditional* application as it executes

binary → [???] → optimized binary

Why?

- Don't have source code!

19 CS 471 – Fall 2007

What is a Dynamic Optimization System?

Transforms* an application at run time

* { translate, optimize, extend }

20 CS 471 – Fall 2007

Classification

Dynamic binary optimizers (x86 → x86opt)

- Complement the static compiler
 - User inputs, phases, DLLs, hardware features
 - Examples: DynamoRIO, Mojo, Strata

Dynamic translators (x86 → PPC)

- Convert applications to run on a new architecture
 - Examples: Rosetta, Transmeta CMS, DAISY

Binary instrumentation (x86 → x86instr)

- Inspect and/or add features to existing applications
 - Examples: Pin, Valgrind

JITs + adaptive systems (Java bytecode → x86)

21 CS 471 – Fall 2007

Dynamic Instrumentation Demo

Pin

- Four architectures – IA32, EM64T, IPF, XScale
- Four OSes – Linux, FreeBSD, MacOS, Windows

22 CS 471 – Fall 2007

What are the Challenges?

Performance!

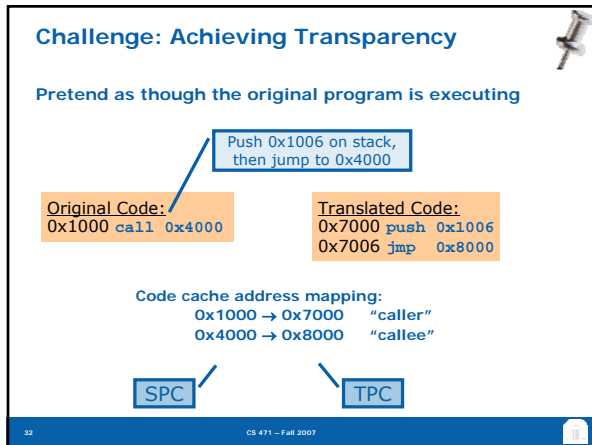
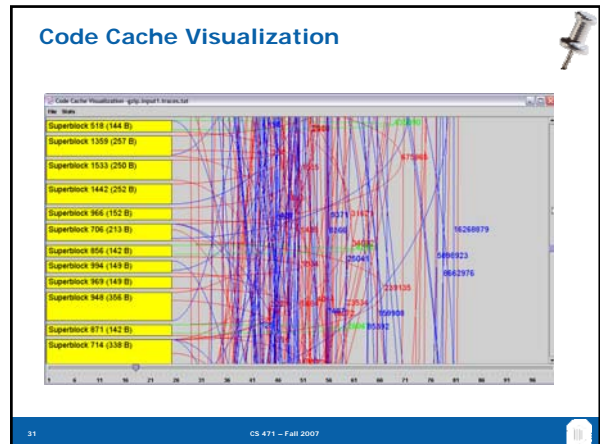
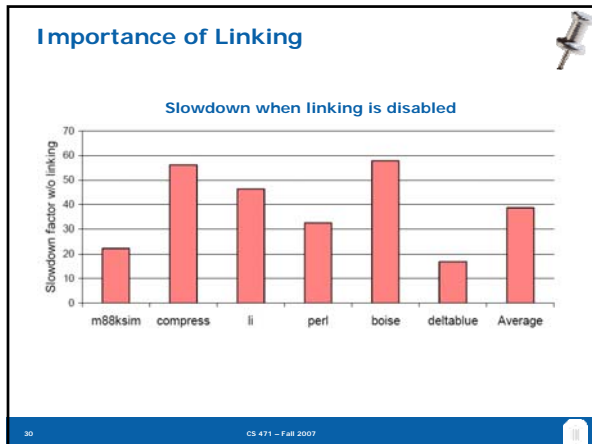
Solutions:

- Code caches – only transform code once
- Trace selection – focus on hot paths
- Branch linking – only perform cache lookup once
- Indirect branch hash tables / chaining
- Memory “management”

Correctness – self-modifying code, munmaps

Transparency – context switching, eflags

23 CS 471 – Fall 2007



Challenge: Self-Modifying Code

The problem
 Code cache must detect SMC and invalidate corresponding cached traces

Solutions
 Many proposed ... but without HW support, they are very expensive!

- Changing page protection
- Memory diff prior to execution
- On ARM, there is an explicit instruction for SMC!

33 CS 471 - Fall 2007

False Self-Modifying Code

The problem
 On some architectures (x86) code may be mixed with data

- Write to data - OK
- Write to code - need to synch

Solution?
 No great solution! (Yet...)

34 CS 471 - Fall 2007

Dynamic Optimization Summary

Complement the static compiler

- Shouldn't compete with static compilers
- Observe execution pattern
- Optimize frequently executed code
 - Optimization overhead could degrade performance

Exploits opportunities

- Arise only at runtime
 - DLLs
 - Runtime constants
 - Hardware features, user patterns, etc.
- Too expensive to fully exploit statically
 - Path-sensitive optimizations

35 CS 471 - Fall 2007

Next Time...

**Course Summary and Wrap-Up
Preparation for Compiler Wars**

