



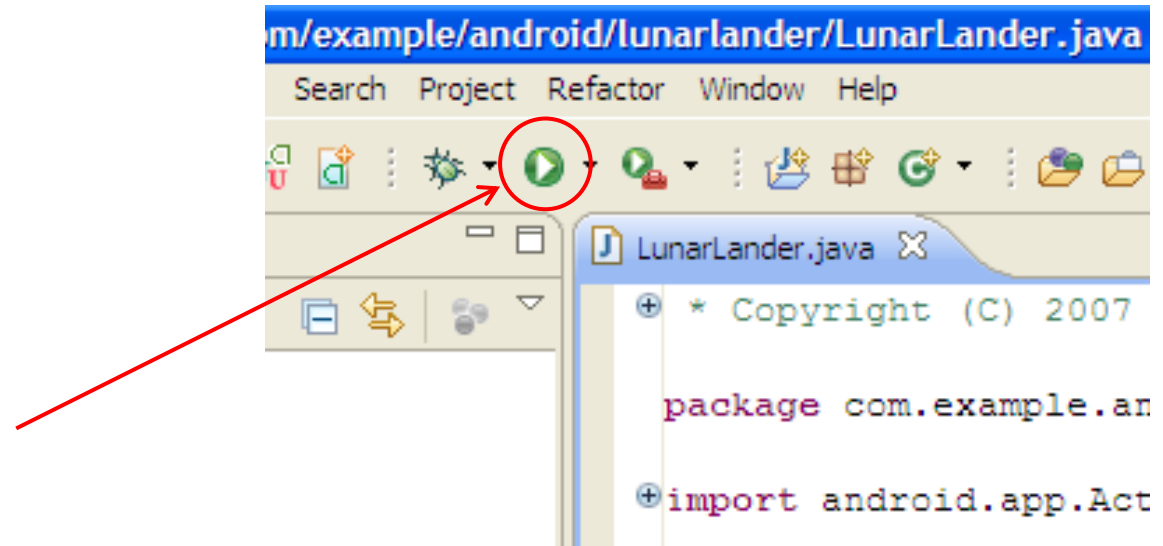
Introduction to Compilation, Just-in-Time Compilers and Virtual Machines

Erven ROHOU

INRIA Rennes – Bretagne Atlantique

ALF Project

About this Talk



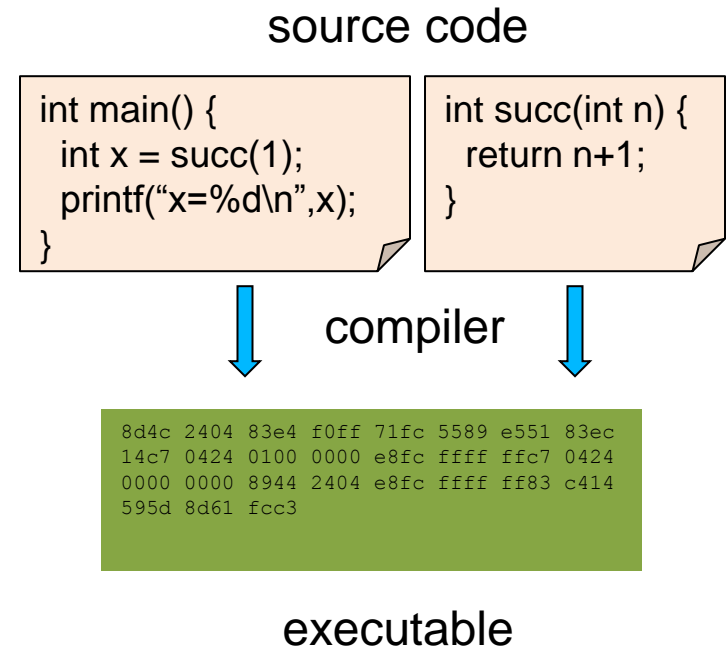
- This is an introduction
- Please, ask questions whenever unclear

Agenda

- Static Compilation
 - binutils
 - compilers
- Virtualization
- JIT Compilers
 - garbage collection
 - profiling

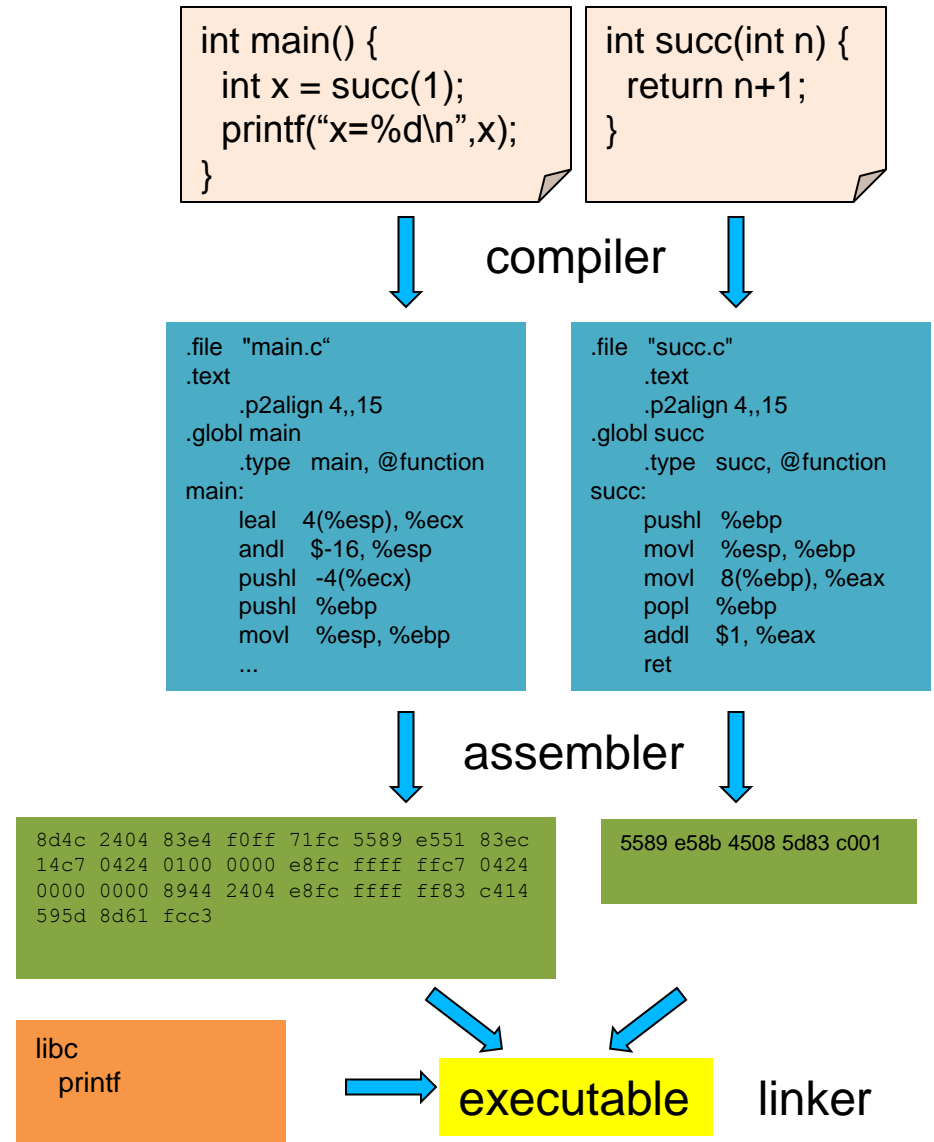
Static Compilation

- Translate from high-level programming language to machine code
- In addition
 - optimization
 - separate compilation
 - packaging: libraries
 - debug, profiling...
- Examples in C

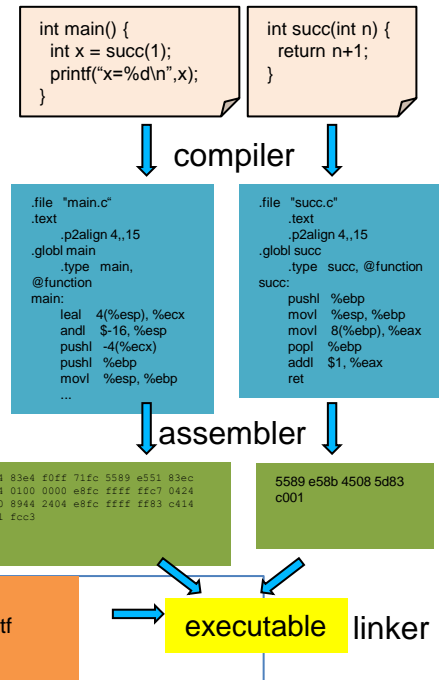


In More Details

- Precisely:
 - the *compiler* translates source to assembly
 - users invoke the *driver*
 - the *binutils* (binary utilities) deal with the rest
 - the driver invokes the tools as necessary



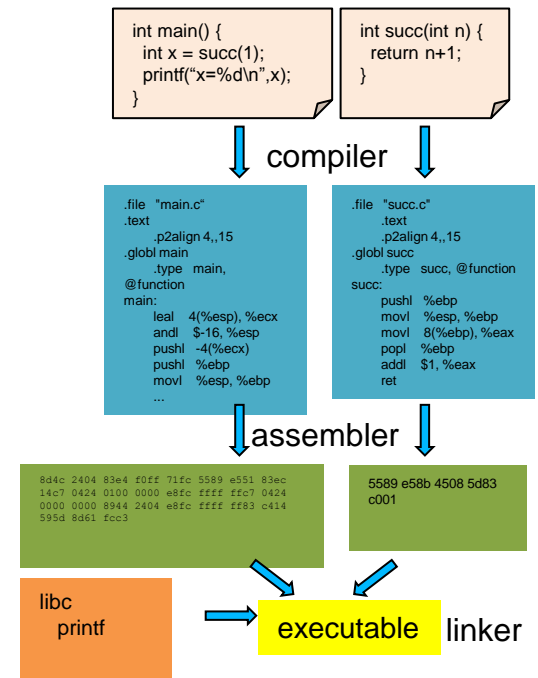
The Driver is your Friend



```
% gcc -v -o succ main.c succ.c
cc1 ... main.c -mtune=generic -march=x86-64 -o /tmp/ccyCWXJg.s
as --64 -o /tmp/ccu4EbpF.o /tmp/ccyCWXJg.s
cc1 ... succ.c -mtune=generic -march=x86-64 -o /tmp/ccyCWXJg.s
as --64 -o /tmp/ccS8HB63.o /tmp/ccyCWXJg.s
collect2 ...-dynamic-linker /lib64/ld-linux-x86-64.so.2 -o succ
/usr/lib64/crt1.o /usr/lib64/crti.o crtbegin.o
-L/udd/alf/rohou/gcc-4.6.1/lib/gcc/x86_64-unknown-linux-gnu/4.6.1
-L/udd/alf/rohou/gcc-4.6.1/lib64 -L/lib64 -L/usr/lib64 -L/udd/alf/rohou/gcc-4.6.1/lib
/tmp/ccu4EbpF.o /tmp/ccS8HB63.o
-lgcc_s -lc -lgcc
/udd/alf/rohou/gcc-4.6.1/lib/gcc/x86_64-unknown-linux-gnu/4.6.1/crtend.o
/usr/lib64/crtn.o
```

Compilation Toolchain

- Bottom-up
 - i.e. more or less chronologically
- Most useful binutils
- Compiler proper



as – assembler

- Mostly bi-univoque relation
 - syntax is more user-friendly
- Compute labels
- Deal with file format

```
.file "main.c"
.text
.p2align 4,,15
.globl main
.type main, @function
main:
    leal 4(%esp), %ecx
    andl $-16, %esp
    pushl -4(%ecx)
    pushl %ebp
    movl %esp, %ebp
    ...
```



```
8d4c 2404 83e4 f0ff 71fc 5589 e551 83ec
14c7 0424 0100 0000 e8fc ffff ffc7 0424
0000 0000 8944 2404 e8fc ffff ff83 c414
595d 8d61 fcc3
```

Libraries

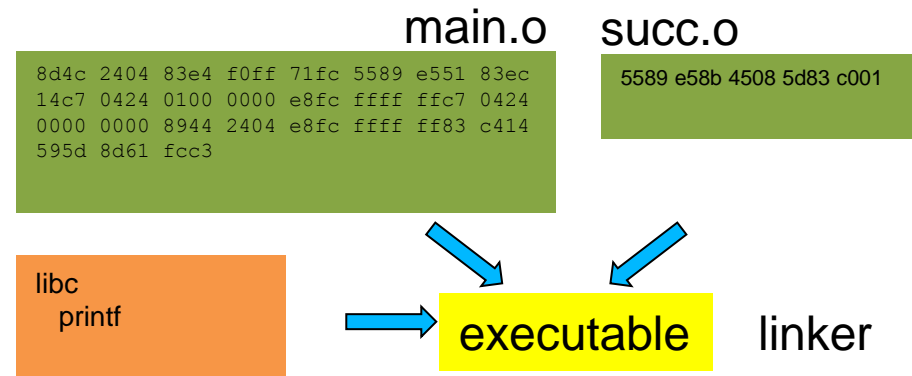
- Factorize commonly used functions/features
 - in Java: `import java.xxx`
- Code reuse, sharing
 - avoid rewrite
- In C: static and dynamic libraries
 - dynamic libraries reduce size of executable

ld – linker

- Combines object files
- Resolve visible symbols
- Relocates data

```
int main() {  
    int x = succ(1);  
    printf("x=%d\n",x);  
}
```

```
int succ(int n) {  
    return n+1;  
}
```



GNU Binutils

- **as** – the GNU assembler
- **ld** – the GNU linker
- **addr2line** - Converts addresses into filenames and line numbers.
- **ar** - A utility for creating, modifying and extracting from archives.
- **c++filt** - Filter to demangle encoded C++ symbols.
- **dlltool** - Creates files for building and using DLLs.
- **gold** - A new, faster, ELF only linker, still in beta test.
- **gprof** - Displays profiling information.
- **nlmconv** - Converts object code into an NLM.
- **nm** - Lists symbols from object files.
- **objcopy** - Copies and translates object files.
- **objdump** - Displays information from object files.
- **ranlib** - Generates an index to the contents of an archive.
- **readelf** - Displays information from any ELF format object file.
- **size** - Lists the section sizes of an object or archive file.
- **strings** - Lists printable strings from files.
- **strip** - Discards symbols.
- **windmc** - A Windows compatible message compiler.
- **windres** - A compiler for Windows resource files.

objdump

- Display information from object files
 - ELF sections
 - code disassembly
 - ...

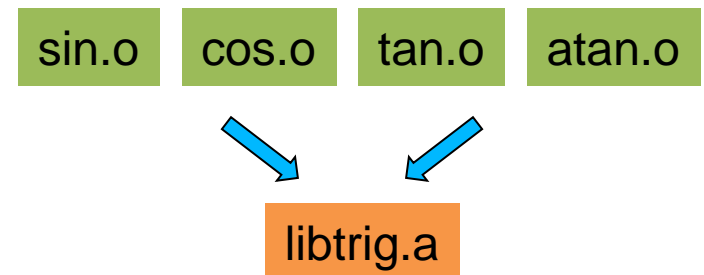
```
0000000004004a4 <succ>:
4004a4:  55                push   %rbp
4004a5:  48 89 e5          mov    %rsp,%rbp
4004a8:  89 7d fc          mov    %edi,-0x4(%rbp)
4004ab:  8b 45 fc          mov    -0x4(%rbp),%eax
4004ae:  83 c0 01          add    $0x1,%eax
4004b1:  5d                pop    %rbp
4004b2:  c3                retq
4004b3:  90                nop

0000000004004b4 <main>:
4004b4:  55                push   %rbp
4004b5:  48 89 e5          mov    %rsp,%rbp
4004b8:  bf bc 05 40 00    mov    $0x4005bc,%edi
4004bd:  e8 de fe ff ff    callq 4003a0 <puts@plt>
4004c2:  b8 00 00 00 00    mov    $0x0,%eax
4004c7:  5d                pop    %rbp
4004c8:  c3                retq
```

```
% objdump -h succ
erat:      file format elf64-x86-64
Sections:
Idx Name   Size      VMA              LMA              File off  Algn
  9 .init   00000018  0000000000400470  0000000000400470  00000470  2**2
          CONTENTS, ALLOC, LOAD, READONLY, CODE
 11 .text   000002a8  0000000000400500  0000000000400500  00000500  2**4
          CONTENTS, ALLOC, LOAD, READONLY, CODE
 13 .rodata 00000030  00000000004007b8  00000000004007b8  000007b8  2**3
          CONTENTS, ALLOC, LOAD, READONLY, DATA
 22 .data   00000010  0000000000600aa8  0000000000600aa8  00000aa8  2**3
          CONTENTS, ALLOC, LOAD, DATA
```

ar, ranlib

- **ar** creates, modifies, and extracts from archives
 - combines object files
 - produces static library
- **ranlib** generates index to archive
 - faster linking



nm, strip, strings, size

- **nm** lists symbols from object files
- **strip** discards all symbols from object files
 - makes executable (slightly) smaller
 - cannot see symbols anymore (**nm** lists nothing)
- **strings** prints printable character sequences
 - at least 4 characters long by default
- **size** lists the section sizes

```
% strings succ
/lib64/ld-linux-x86-64.so.2
__gmon_start__
libc.so.6
printf
__libc_start_main
GLIBC_2.2.5
fff.
fffff.
l$ L
t$(L
|$(0H
x = %d
```

```
% nm succ
...
00000000004004c0 T main
                                U printf@@GLIBC_2.2.5
00000000004004b0 T succ
```

gprof

- Produces an execution profile
- Useful to pinpoint long routines
 - at compile time, link with special library
 - at runtime, produce profile
 - post-mortem, run **gprof**

indx	%time	self	children	called	name
		0.03	0.00	1/1	main [2]
[1]	100.0	0.03	0.00	1	compute_primes [1]

				<spontaneous>	
[2]	100.0	0.00	0.03		main [2]
		0.03	0.00	1/1	compute_primes [1]
		0.00	0.00	921500/921500	print_other [3]
		0.00	0.00	78498/78498	print_prime [4]

		0.00	0.00	921500/921500	main [2]
[3]	0.0	0.00	0.00	921500	print_other [3]

		0.00	0.00	78498/78498	main [2]
[4]	0.0	0.00	0.00	78498	print_prime [4]

```
#include <stdio.h>
#define N 100

void compute_primes(int n,
                    char* sieve)
{
    int i, j;
    for(i=0; i<n; i++)
        sieve[i]=1;
    for(i=2; i<n; i++) {
        for(j=2*i; j<n; j+=i)
            sieve[j] = 0;
    }
}

void print_prime(int n) {
    printf("%d prime\n",n);
}

void print_other(int n) {
    printf("%d NOT prime\n",n);
}

int main() {
    char sieve[N];
    int i;
    compute_primes(N, sieve);
    for(i=2; i<N; i++)
        if (sieve[i])
            print_prime(i);
        else print_other(i);
}
```

Dynamic Loader

- Not a *visible* tool
 - invoked by the system to start process
- fork/exec (seen already)
- Load libraries
- Load executable
- Patch relocations
- Execute

The Compiler

- Translate from high-level programming language to assembly
- In addition
 - optimization

```
int main() {  
    int x = succ(1);  
    printf("x=%d\n",x);  
}
```



```
.file "main.c"  
.text  
    .p2align 4,,15  
.globl main  
    .type main, @function  
main:  
    leal 4(%esp), %ecx  
    andl $-16, %esp  
    pushl -4(%ecx)  
    pushl %ebp  
    movl %esp, %ebp  
    ...
```

cpp

- Macro processor
 - #define
 - #include
 - #if #endif
- Invoked automatically by the C compiler

```
gcc -DDEBUG -o foo foo.c
```

```
#include <stdio.h>

int foo(int n) {
    #if DEBUG
        printf("n = %d\n", n);
    #endif
    ...
}
```

Lexical Analysis

- Read source code as text file
- Produce tokens
- Recognize key language elements
 - reserved keywords
 - numbers
 - ...

i	f		(c	o	n	d)		{
		x	=	4	2	;				
}										
e	l	s	e		{					
		y	=	2	.	4	;			
}										

if		(cond)		{				
		x	=	42	;					
									INT	
}										
else					{					
		y	=	2.4	;					
									FLOAT	
}										

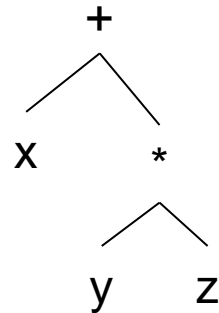
Parsing

- Start organizing the tokens
 - grammar
- Eclipse does it all the time to show errors
- Many tools to generate code
 - lex, yacc, ANTLR

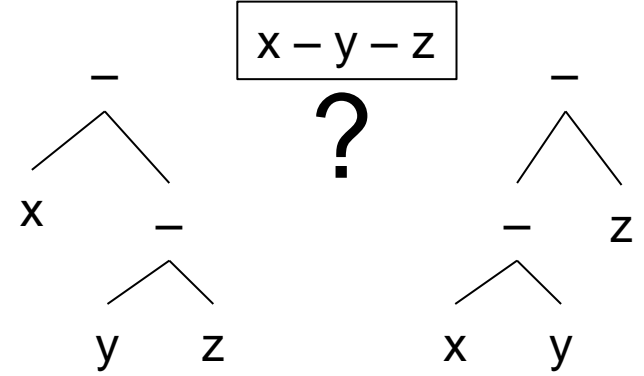
```
S → S; S | ε
S → if E then S
S → if E then S else S
...
E → E + T | E - T | T
T → T * F | T / F | F
F → num
F → id
F → (E)
```

```
E → E + E | E - E | T
T → T * F | T / F | F
F → id
```

x + y * z

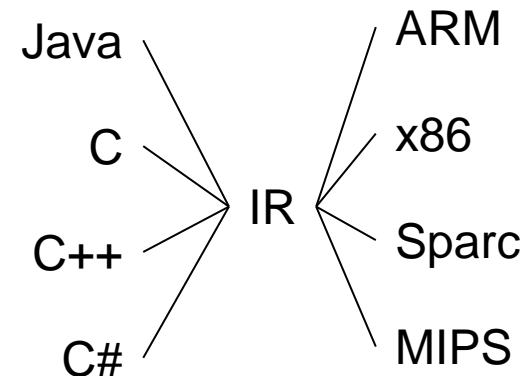
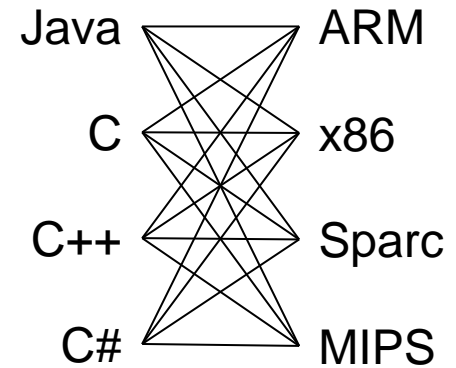


x - y - z



Intermediate Code

- Intermediate Representation
 - abstract machine language
 - independent of details of source language
- Key idea: for N languages, M machines
 - $N + M < N.M$
- Many examples
 - GCC: GIMPLE, RTL
 - Open64: WHIRL
 - recent: MinIR, Tires, ...



Optimizations

- Take advantage of
 - algebraic properties
 - instruction set characteristics
 - ...
- Hundreds of optimizations
 - Common subexpression elimination
 - Tree balancing
 - Strength reduction
 - Loop unswitching
 - Dead code elimination

```
x = a * b + c
y = a * b + d
```



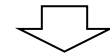
```
tmp = a * b
x = tmp + c
y = tmp + d
```

```
a = Math.pow(b, 2)
x = y * 16
```



```
a = b * b
x = y << 4
```

```
x = ((a + b) + c) + d
```



```
x = (a + b) + (c + d)
```

```
for(i=0; i < n; i++) {
  if (cond) {
    ...
  }
}
```



```
if (cond) {
  for(i=0; i < n; i++) {
    ...
  }
}
```

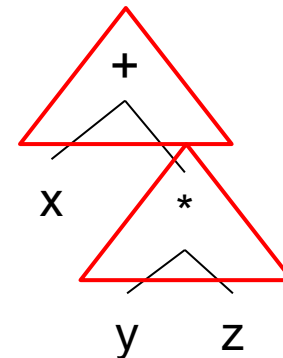
Instruction Selection

- From IR to actual instructions
- Cover the IR tree with tiles
 - tiles represent the instruction set
 - the tree is the optimized IR
- Minimize cost

$+$
add

$-$
sub

$*$
mul

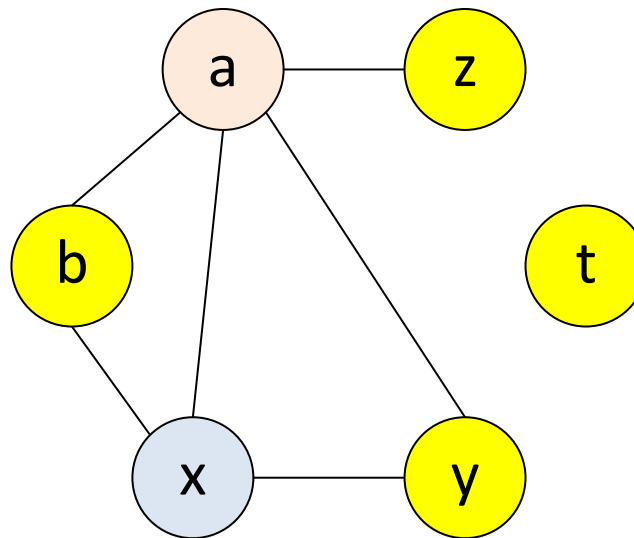
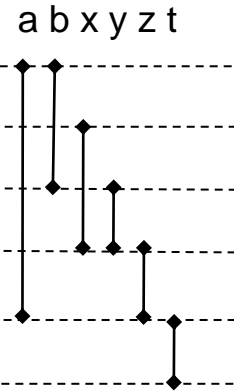


```
mul tmp ← y, z  
add res ← x, tmp
```

Register Allocation

- Map large number of values to (small number of) machine registers
- Sub tasks
 - Liveness analysis
 - Interference graph
 - Graph coloring
 - Spilling
- but also
 - Precolored nodes
 - Specific constraints

```
int f(int a, int b) {  
  x = a + b;  
  y = a - b;  
  z = x * y;  
  t = z + a;  
  return t;  
}
```

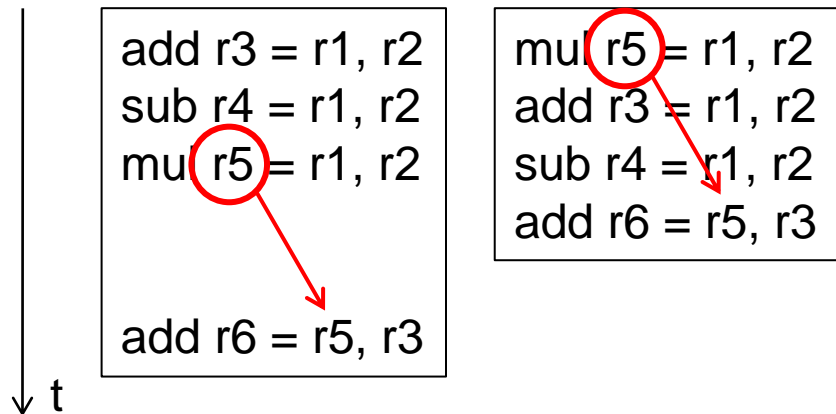


```
add r3 = r1, r2  
sub r2 = r1, r2  
mul r2 = r2, r3  
add r2 = r2, r1  
mov r1 = r2  
ret
```

Scheduling

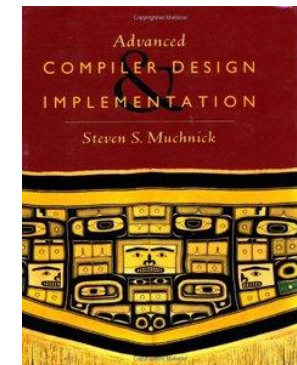
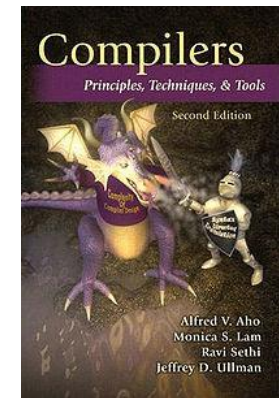
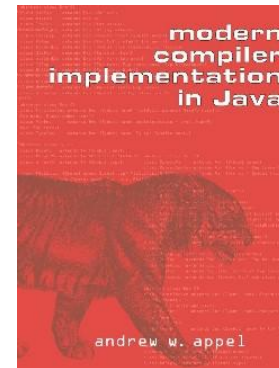
- Optimize order of instructions
 - better usage of processor pipeline
 - respect dependences
- Very important for VLIW and in-order processors
 - less so for out-of-order processors (x86)

Example:
add takes 1 cycle
mul takes 3 cycles



Some References

- “Compilers: Principles, Techniques, & Tools”, Aho, Lam, Sethi, Ullman. Dragon Book.
- “Modern Compiler Implementation in Java”, Andrew W. Appel.
- “Advanced Compiler Design and Implementation”, Steven Muchnick.
- “Basics of Compiler Design”, Torben Ægidius Mogensen, <http://www.diku.dk/~torbenm/Basics>



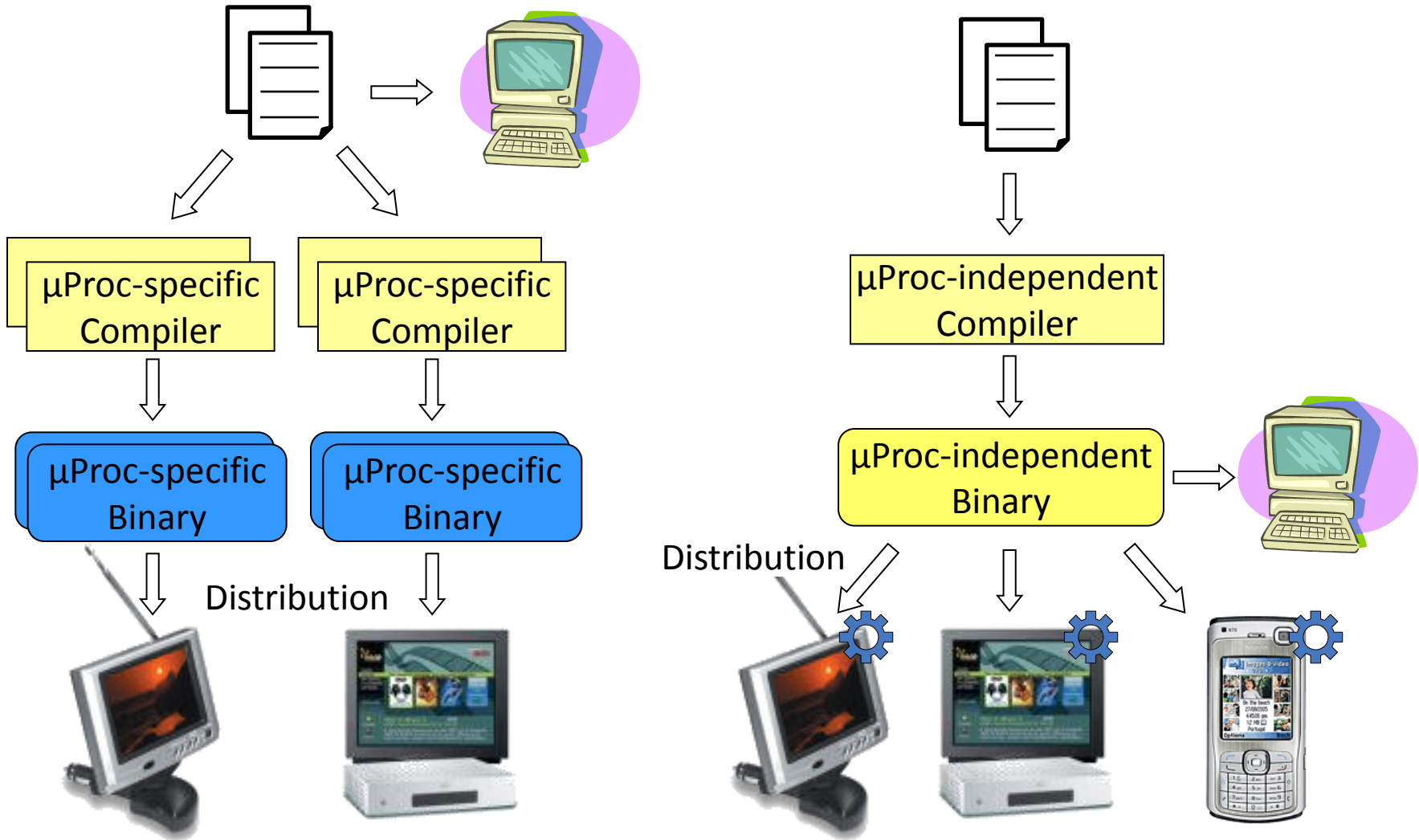
Developing software is difficult

- Programming languages provide abstractions...
 - dynamic memory allocation
 - object-oriented programming
 - strong typing
 - software components, etc.
- ...to help programmers
 - reduced efforts
 - better error detection
 - better reuse (libraries, components)

The catch

- Abstractions have a performance cost
 - dynamic memory allocation: false dependencies
 - object-oriented programming: virtual dispatch, extra indirection, small methods
 - automatic memory management: need efficient garbage collection
 - late binding: need to deal with unknown information
 - reflection: deal with changing code
- (Good) support is provided in some kind of runtime
 - the Virtual Machine (VM)
- Not all aspects derive from the JIT/VM
 - for example: GC, object oriented language

Bytecodes and Just-in-time Compilers



Why?

- Deployment
 - simplify software engineering
 - addresses legacy problems
- Security, sandboxing
- Observability
 - for the application, the VM is OS, hardware, runtime, ...
- Performance
 - because of additional information
 - compile only what is needed

Deployment

- Toolchain burden
 - maintain, upgrade
- Debug, validate
 - run what you validate
- Access to remote parts of the system
 - ISV
- Ship to future versions of the system
 - no worry about binary compatibility
 - no (less) worry about performance

Security, sandboxing

- Typical of VM (even though not a consequence of JIT)
- Limit access to physical resources
 - I/O, network
 - by policy
 - cf. `AndroidManifest.xml` and `<uses-permission>`
- Static analysis
 - array bounds
 - stack overflow, underflow

Obtain Even More Performance

- More information is available
 - OS
 - actual hardware
 - Nehalem vs. Core vs. Atom vs. ARM vs. ???
 - generate most efficient instructions (e.g. SSE4.2 if available)
 - whole program optimization
 - inline library functions
 - runtime constants
 - program input

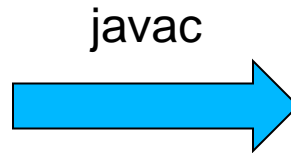
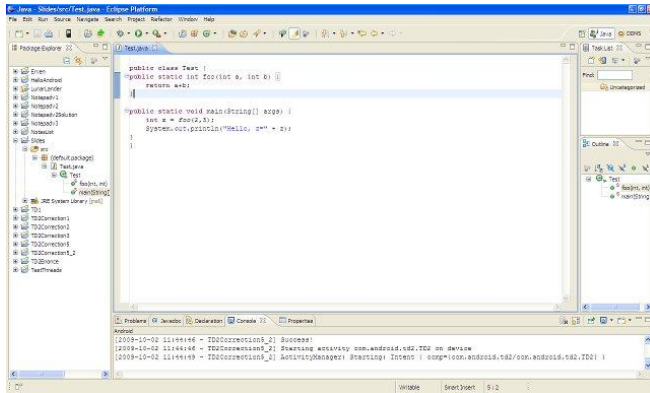
Challenges

- Compile time is part of runtime
- Memory shared between compiler and user
- Reverse engineering
 - obfuscation

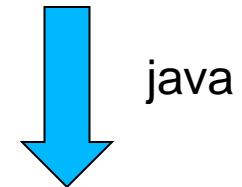
History

- UCSD Pascal p-System mid-1970
- Smalltalk 1980
- CLisp 1987
- Sun's Self 1990
- Java 1995
- Transmeta's Crusoe and Code Morphing 2000 (x86 -> VLIW)
- LLVM initiated 2000
 - goes into Apple's Mac OS 10.5 'Leopard' OpenGL stack 2006
- CLI
 - .NET 1.0 2002, .NET 3.5 2007
 - ECMA standard 2001, ISO standard 2003

Java Compilation Flow

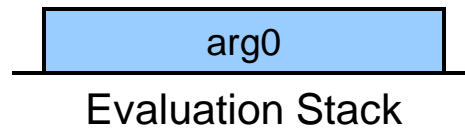


Test.class



Evaluation Stack

`iload_0`

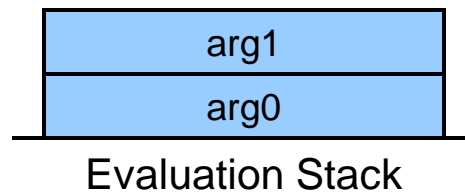


- Potential Benefits:
 - Machine independence
 - Managed environment and security (against both malicious code and application faults)
 - Cross-language interoperability

Evaluation Stack

`iload_0`

`iload_1`



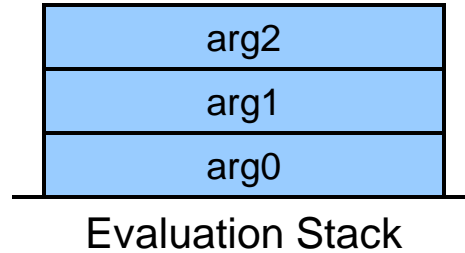
- Potential Benefits:
 - Machine independence
 - Managed environment and security (against both malicious code and application faults)
 - Cross-language interoperability

Evaluation Stack

`iload_0`

`iload_1`

`iload_2`



- Potential Benefits:
 - Machine independence
 - Managed environment and security (against both malicious code and application faults)
 - Cross-language interoperability

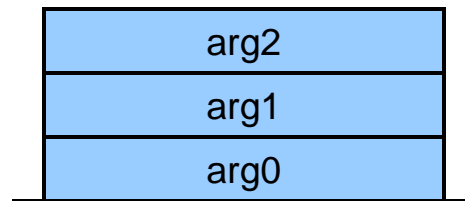
Evaluation Stack

`iload_0`

`iload_1`

`iload_2`

`iadd`



Evaluation Stack

- Potential Benefits:
 - Machine independence
 - Managed environment and security (against both malicious code and application faults)
 - Cross-language interoperability

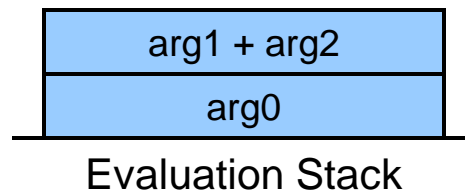
Evaluation Stack

`iload_0`

`iload_1`

`iload_2`

`iadd`



- Potential Benefits:
 - Machine independence
 - Managed environment and security (against both malicious code and application faults)
 - Cross-language interoperability

Evaluation Stack

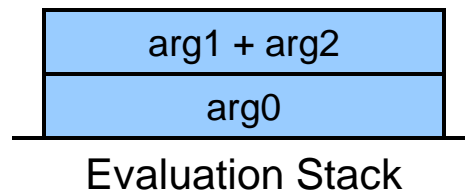
`iload_0`

`iload_1`

`iload_2`

`iadd`

`imul`



- Potential Benefits:
 - Machine independence
 - Managed environment and security (against both malicious code and application faults)
 - Cross-language interoperability

Evaluation Stack

`iload_0`

`iload_1`

`iload_2`

`iadd`

`imul`

`arg0 * (arg1 + arg2)`

Evaluation Stack

- Potential Benefits:
 - Machine independence
 - Managed environment and security (against both malicious code and application faults)
 - Cross-language interoperability

Evaluation Stack

`iload_0`

`iload_1`

`iload_2`

`iadd`

`imul`

`ireturn`

`arg0 * (arg1 + arg2)`

Evaluation Stack

- Potential Benefits:
 - Machine independence
 - Managed environment and security (against both malicious code and application faults)
 - Cross-language interoperability

Evaluation Stack

`iload_0`

`iload_1`

`iload_2`

`iadd`

`imul`

`ireturn`

Evaluation Stack

- Potential Benefits:
 - Machine independence
 - Managed environment and security (against both malicious code and application faults)
 - Cross-language interoperability

Getting more concrete

- Example of Java bytecode
- Notice
 - evaluation stack
 - local variables
 - strongly typed

```
public class Test {  
    public static int foo(int a, int b) {  
        return a+b;  
    }  
  
    public static void main(String[] args) {  
        int z = foo(2,3);  
        System.out.println("Hello, z=" + z);  
    }  
}
```

```
public static int foo(int, int);  
    0: iload_0  
    1: iload_1  
    2: iadd  
    3: ireturn  
public static void main(java.lang.String[]);  
    0: iconst_2  
    1: iconst_3  
    2: invokestatic #2;    //foo:(II)I  
    5: istore_1  
    6: getstatic #3;        //Field System.out:Ljava/io/PrintStream;  
    9: new #4;              //class java/lang/StringBuilder  
    12: dup  
    13: invokespecial #5;    // java/lang/StringBuilder."<init>":()V  
    16: ldc #6;              //String Hello, z=  
    18: invokevirtual #7;    //StringBuilder.append;  
    21: iload_1  
    22: invokevirtual #8;    //StringBuilder.append;  
    25: invokevirtual #9;    //StringBuilder.toString;  
    28: invokevirtual #10;   //java/io/PrintStream.println:()V  
    31: return
```

Interpreting

- Easy to implement
 - basically, a huge switch statement
 - Java implementations in cell phones
- No startup
- Slow execution

```
while (bytecode = read()) {
    switch(OPCODE(bytecode)) {
        case IADD:
            x = pop();
            y = pop();
            push(x+y);
            break;

        case ICONST_2:
            push(2);
            break;

        case INVOKESTATIC:
            ...
            break;

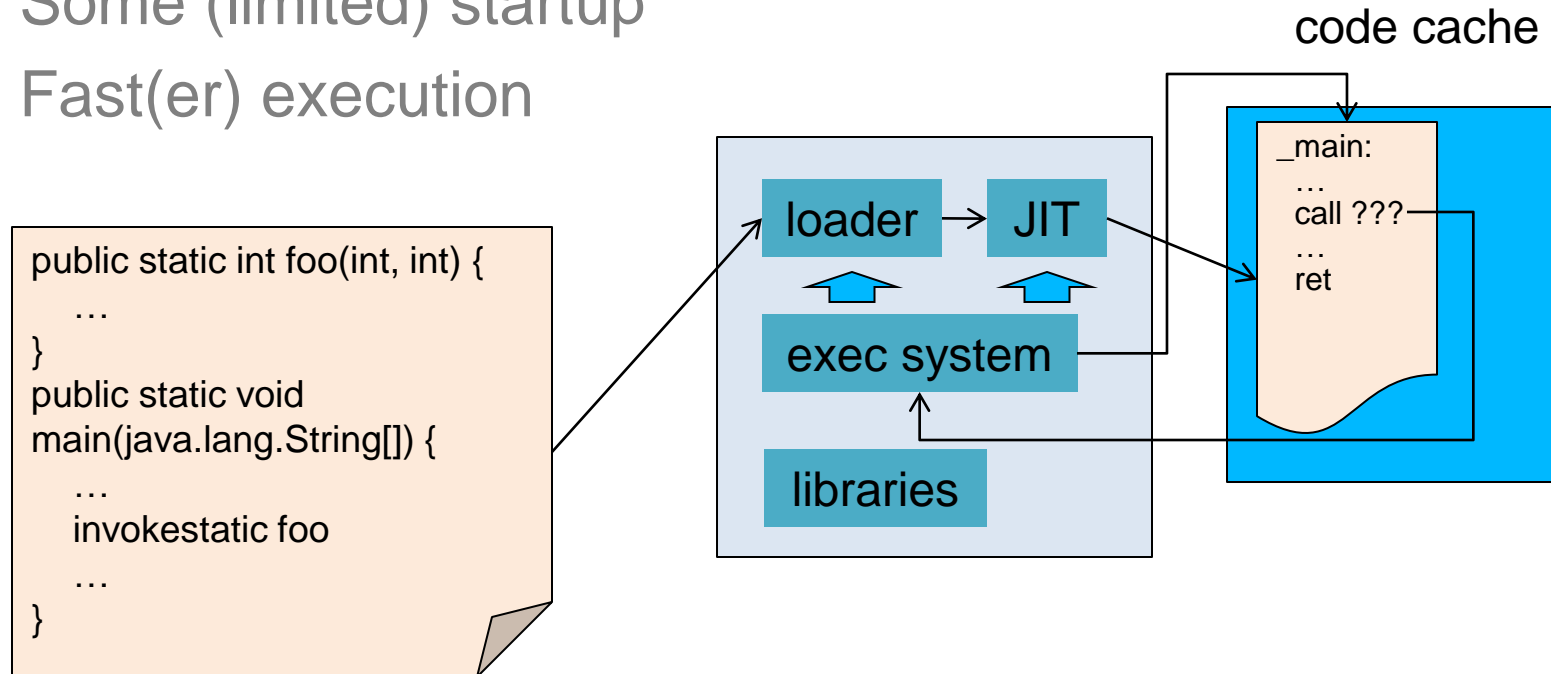
        ...
    }
}
```

AOT (Ahead Of Time) Compiling

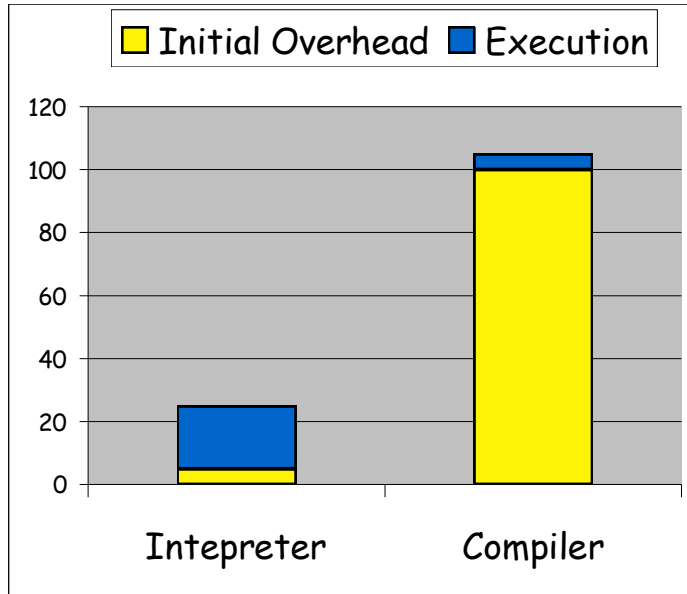
- Standard compilation, but happens on target platform
- Large startup
 - time to compile the bytecode to native code
- Fast execution
 - same as standard compilation

JIT (Just In Time) Compiling

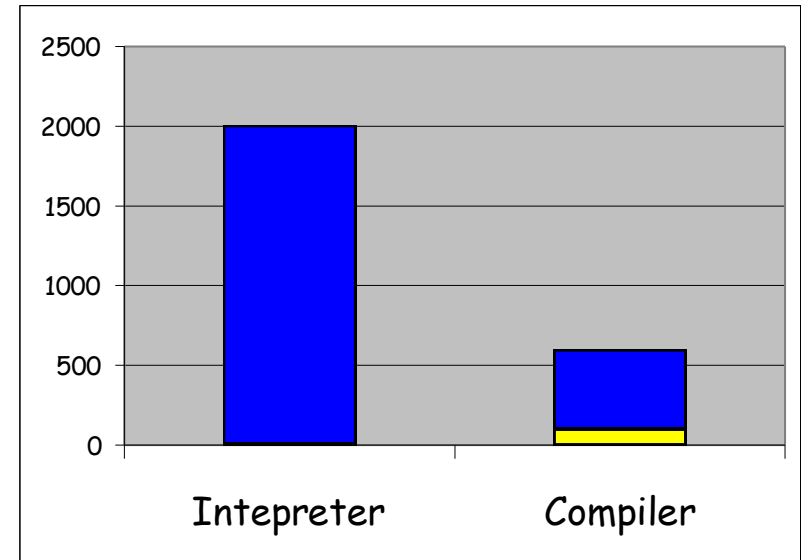
- Compile code Just-in-Time in memory
 - usually one function at a time
- Some (limited) startup
- Fast(er) execution



Interpretation vs JIT



Execution: 20 time units

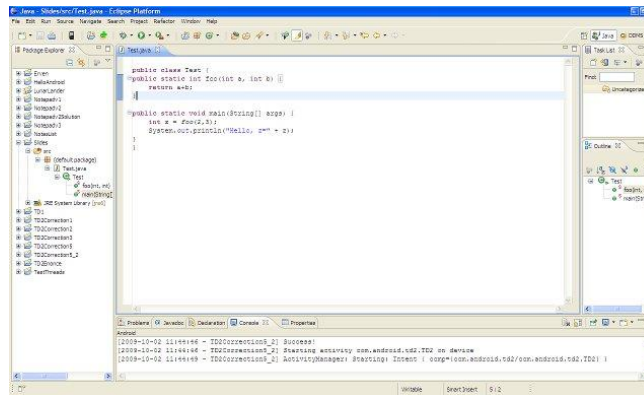


Execution: 2000 time units

[From Kathryn S McKinley, UT]

Android Compilation Flow

- Based on standard Java flow, but:
 - own bytecode, translated from Java
 - own libraries (not Java)
 - registers instead of evaluation stack



javac



Test.class

dx



Test.dex



adb install



Portions of this page are reproduced from work created and [shared by Google](#) and used according to terms described in the [Creative Commons 3.0 Attribution License](#).

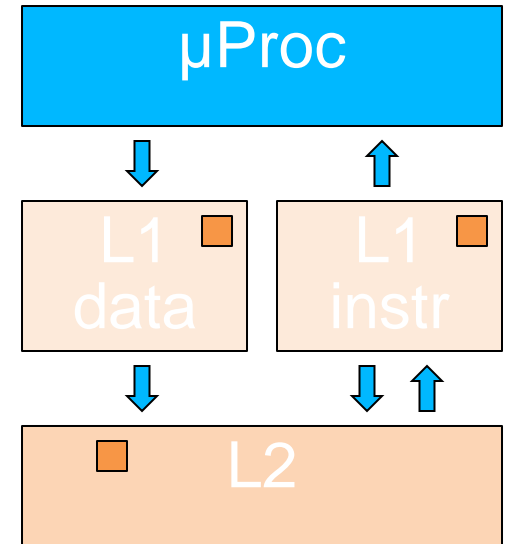
Android Runtime

- User code (.dex) used to be interpreted
 - Now JIT compiled
- Large set of libraries
- native code



Code Cache

- Store binary code (as if it was data)
- Need to handle hardware I/D cache
 - automatic on x86
 - careful on others
- New security mechanisms in Linux
 - protect against buffer overflow attacks
 - ...but also against JIT compilers



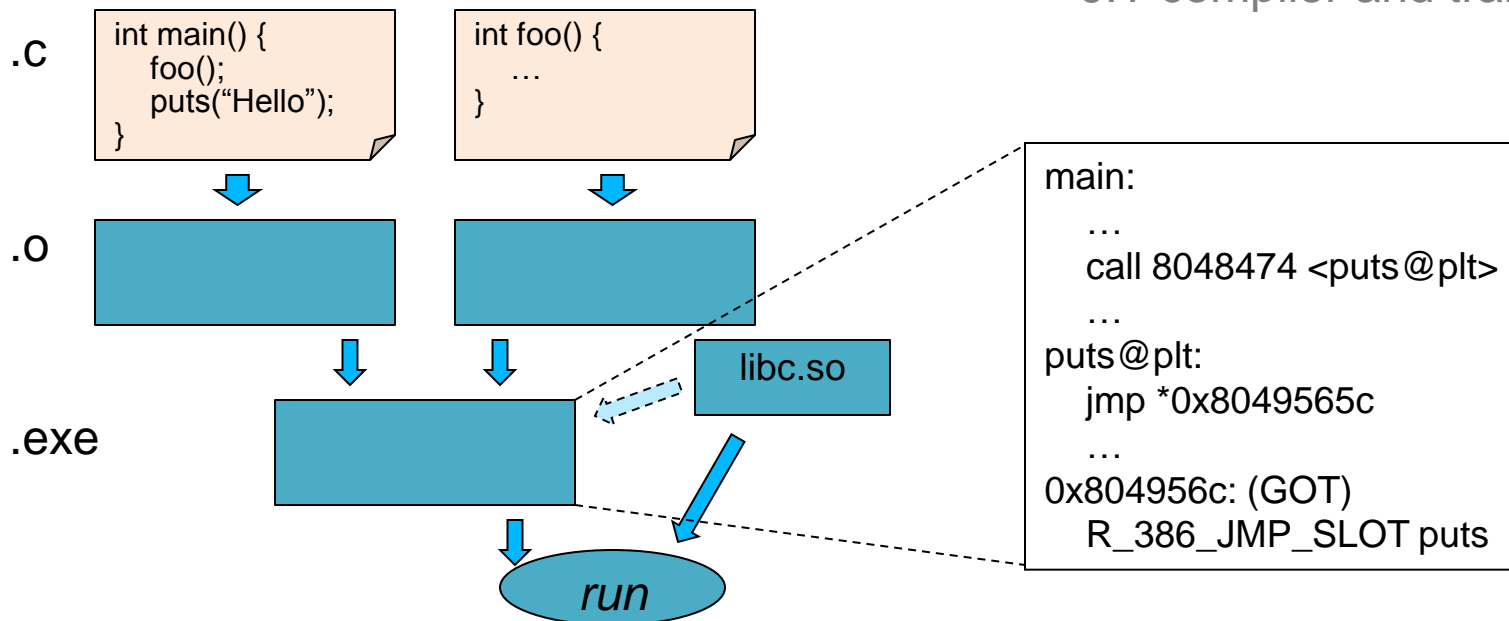
Tool chain: when JIT is easier

Standard compilation

- Deal with unknown addresses
 - relocations
 - assembler, linker, loader

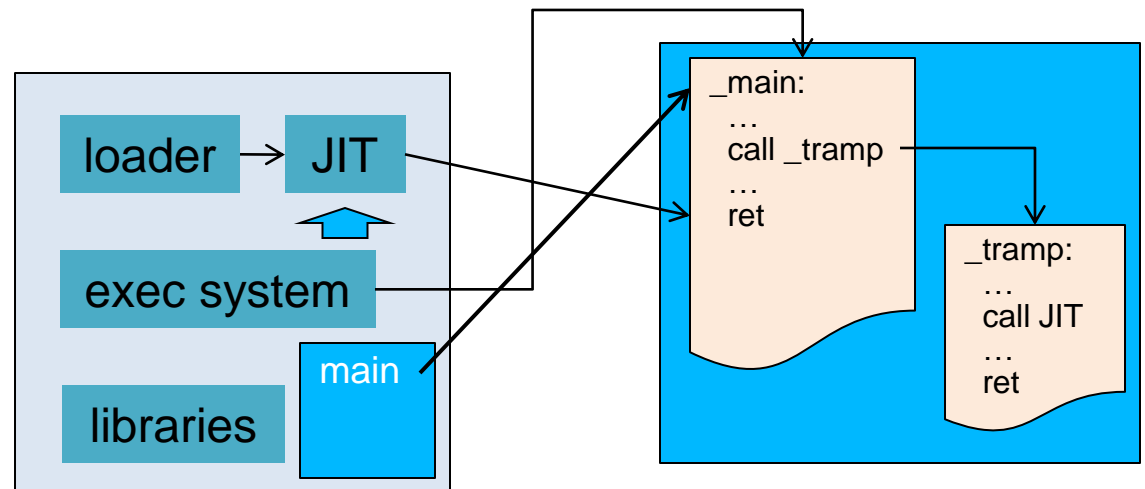
JIT compilation

- Code is generated when needed
 - addresses are known
 - JIT compiler and trampolines



Trampolines

- Hook to call back JIT when function is missing
- Can be used for additional services
 - e.g. profiling
- Remove when code becomes available



Runtime Opportunities

- Profiling, tracing
 - knowledge of application behavior
 - knowledge of environment (OS, actual target)
- Specialization
 - generate code that takes advantage of this knowledge
- Re-optimization
 - *hot* functions
 - phase changes

Selective Optimization

- Leverage the 90/10 rule (aka 80/20)
 - 90% of the time is spent in 10% of the code
 - optimize only the important part
- Strategy
 - initial unoptimized version
 - profile to detect *hot* functions
 - optimize them

Garbage Collection (GC)

- Simplify memory management
 - allocate
 - leave deallocation to the system
- Already present in Lisp, Smalltalk
- Today in Java, C#, Perl, Javascript,...

```
ptr = malloc(sizeof(*ptr));  
...  
free(ptr);
```

```
t = new Thread();  
...
```

Garbage Collection: Principles

- Garbage: any object that will never be referenced again
 - problem: cannot be computed
- Garbage: any object that cannot be reached
 - ok: approximate liveness with reachability
- Basic idea
 - start from known live objects
 - follow all “links” and mark reachable objects as alive
 - reclaim unmarked objects

```
b = new Button();  
...  
b = new Button(); /* kills previous */
```

Garbage Collection: Pros

- Simpler code
 - easier to understand
 - less error prone
 - faster
- Helps avoid
 - access to non-allocated memory
 - free already freed memory
 - memory leak
- Can make code faster
 - simpler code, easier to optimize
 - better locality with compacting GC

```
while (cond1) {  
    if (cond2)  
        ptr = malloc(sizeof(*ptr));  
    foo(ptr);  
}  
...  
if (???)  
    free(ptr);
```

```
while (cond1) {  
    if (cond2)  
        a = new A();  
    foo(a);  
}  
...
```

Garbage Collection: Cons

- Pause-time
- Difficult to control collection time
 - inappropriate for real-time applications
- Might require more memory
- Memory leaks more difficult to detect

Profiling

- What do you collect?
- When do you collect?
- How do you collect?
- What do you do with the collected data?

What data do you collect?

- Executed functions
- Executed paths, branch outcome
- Parameter values
- Loads and stores
- ...

- Profile or trace?
- And consider overhead!

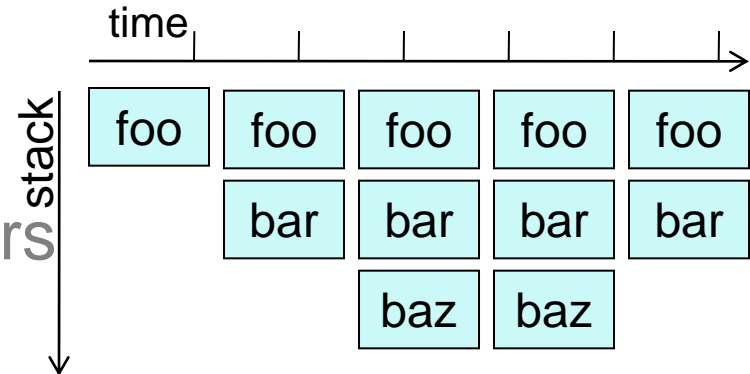
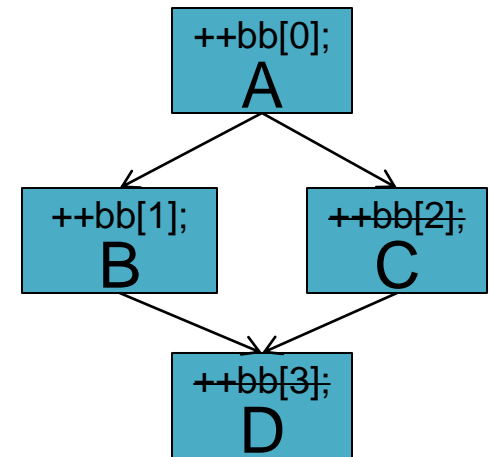
When do you collect the data?

- Interpretation vs. JIT
- Continuous vs. intermittent
 - install and uninstall profiling code
- Phase based: early, steady state
- Tradeoff accuracy vs. overhead

How do you collect the data?

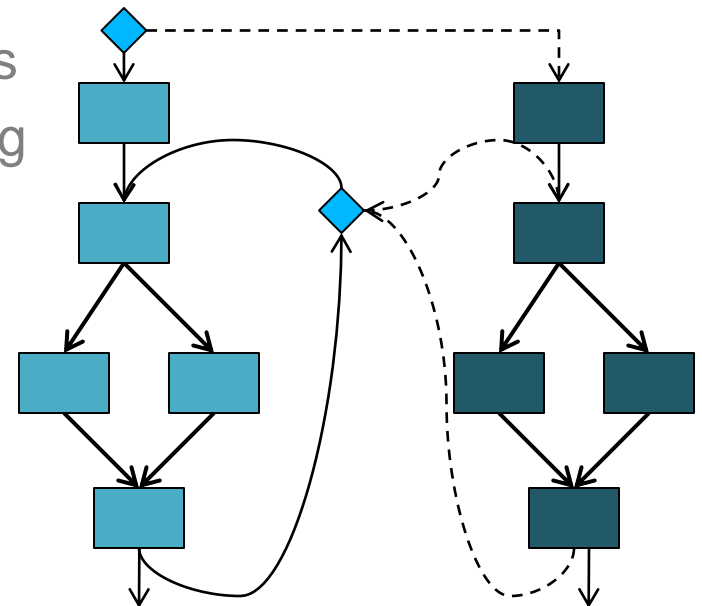
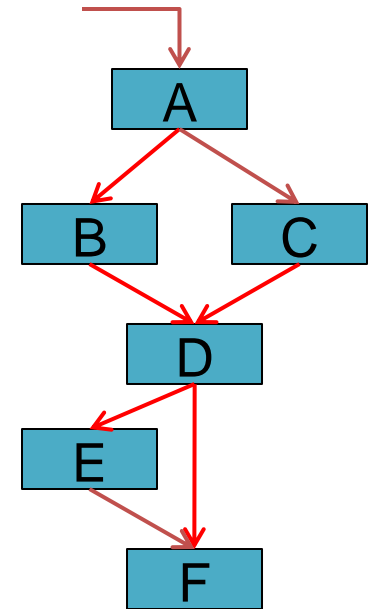
- Program instrumentation
 - function, basic block, edges, value
 - potential optimization (e.g. spanning tree)
- Sampling
 - running method, call stack
 - not deterministic, less accurate
- Hybrid
- Provided by VM
 - GC, class hierarchy
- Hardware performance counters
 - locality estimate, IPC, ...

```
public int foo() {  
    ++freq[FOO_ID];  
    ...  
}
```



Control cost of profiling

- Do as much as possible “statically”
 - edge profiling with (minimum) spanning tree
- Duplicate method [ArnoldRyder2001]
 - one is profiled, one not
 - insert checks on entry and backedges
 - instrumented code returns to checking code



Control cost of profiling (cont'd)

- Ephemeral instrumentation [Traub2000]
 - patch/unpatch
 - example of conditional branches
 - patch at load-time, possibly at each page fault
 - unhook after some number of executions
 - rehook later to capture changing phases

```
...  
beq $r1,tgt  
br stub  
ft:  
...  
tgt:  
...
```

```
stub:  
    cmpeq $r1, 0, $t0  
    process_branch()  
    bne $t0, tgt  
    br ft  
  
[ total_count ]  
[ taken_count ]  
[ aggregate counters ]
```

```
process_branch:  
    total_count++  
    taken_count += $t0  
    if (total_count > unhook_constant)  
        unhook_branch()  
    return
```

What do you do with the data?

- Function specialization
- Inlining
- Code layout (hot/cold optimization)
- Multiversioning
- Loop unrolling
 - unroll *hot* loops
- Register allocation
 - spill in *cold* regions
- Stack allocate objects that escape only on cold paths

Specialization

- Take advantage of frequent values
 - identify pseudo constants
 - optimize for particular values
 - enable other optimizations

```
int mul(int a, int b)
{
    return a*b;
}
```



```
int mul(int a, int b) {
    return a*b;
}

int mul_2(int a) {
    return a + a;
}
```

```
int foo(int n)
{
    if (n==3) {
        complex_fun(n);
    }
}
```



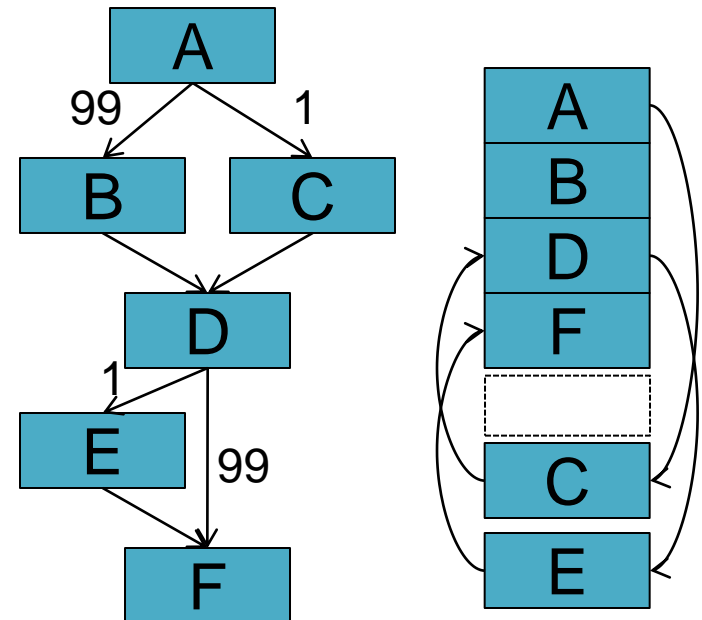
```
int foo1() {
    return;
}

int foo2(int n) {
    complex_fun(n);
}
```

Code Layout

- Profile paths
- Layout for most frequent path (hot/cold)
 - good for spatial locality, prefetch
 - good for branch prediction

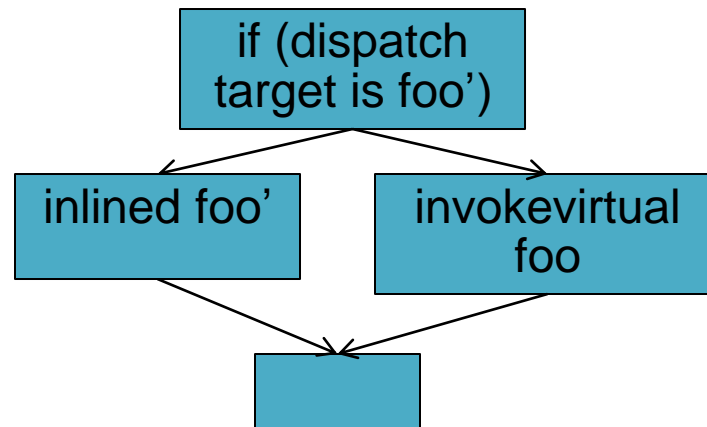
```
if (error) {  
    result = NULL;  
}  
else {  
    result =  
}  
if (error2) {  
    errorCode = 1;  
}
```



Multiversioning

- Static
 - emit multiple implementations
 - emit code to choose the best one at runtime
- Dynamic
 - generate ad hoc implementation on-the-fly
 - mostly deals with dispatch tables in OO languages

invokevirtual foo



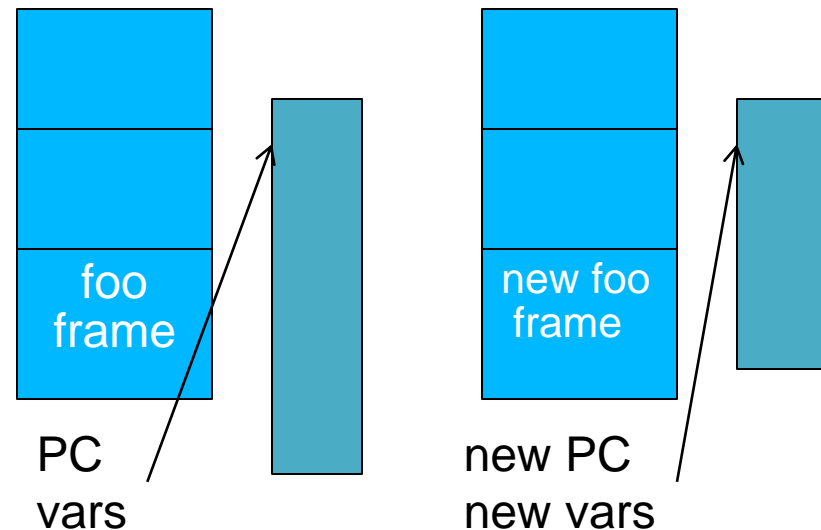
Re-optimization

- Keep monitoring for hot functions
 - or enable monitoring from time to time, sampling
- Re-optimize very hot ones
 - JIT again, using more aggressive optimizations
 - update in code cache
 - possibly use On-Stack-Replacement

On Stack Replacement

- Change code of running function
 - from interpreter to JIT
 - from low to high optimization level
 - adjust to changing behavior
- Generate program states
 - locations of local variables
 - program counter
- At replacement time
 - generate new stack frame
 - generate new code
 - restore state
 - transfer execution

```
public static int main(String[] args)
{
    for(int i=0; i<1000000; ++i) {
        /* do something */
    }
}
```



JIT/VM Interaction

- The JIT compiler is only a part of the VM
 - or the VM is only the support of the JIT?
- VM services
 - memory management (GC)
 - exception handling
 - type checking
 - interface to OS and hardware
 - dynamic linking
 - hardware counters, traps, signals
- Codesign is necessary

JIT support for VM

- For memory management
 - roots locations (registers, stack, etc.)
 - types
 - where GC can occur
- For exceptions
 - location of try/catch blocks in generated code
- For debugging
 - mapping bytecode - source code

VM support for JIT

- Memory allocation
 - inline frequent allocator
 - need to expose internal details
- NULL pointer check
 - rely on OS signals
- Generated code relies on runtime
 - “optimized” API
 - direct access to data structures

JIT/VM Tight Integration

Pros

- Performance of the runtime
- Allows optimized generated code
 - inlined code
 - direct access to data structures

Cons

- Software engineering
 - intricate software components
 - unspecified interfaces
 - maintenance, debug
 - no reuse of JIT in other VM