Java Acceleration Compilers

Java Software Platform

Java VM is a popular embedded S/W platform

- Why VM in embedded systems?
  - Diverse H/W platforms (CPU, OS, display…)
  - Provide consistent runtime environment
- Why Java?
  - Security issues
    - Hard to kill the whole system with malicious Java code
  - Easy to develop S/W contents
    - Mature API
    - Robust language features: garbage collection, exception handling

Java Performance

- One critical problem: performance
  - Due to its "write once, run everywhere" portability
  - Compiled into bytecode, not native machine code
  - Software layer (JVM) to interpret bytecode is really slow

- Solution: Java acceleration
  - S/W solution: translation into machine code
    - Just-in-Time compilation (JITC)
    - Ahead-of-Time compilation (AOTC)
    - Client-AOTC (c-AOTC), install-time compilation (ITC)
  - H/W solution: direct hardware execution of bytecode
    - ARM Jazelle, Nazomi JSTAR, …

S/W Solution: Translation

- Translate bytecode method into native method
  - JITC: on-line, just before it is executed
    - Translated methods are cached for later use
    - Hotspot, J9, Jikes RVM, …
  - AOTC: off-line, even before the programs starts
    - Dynamically loaded methods should be interpreted
    - WIPI, Jbuild, …

- Allow Java programs run as native executables

A Translation Example

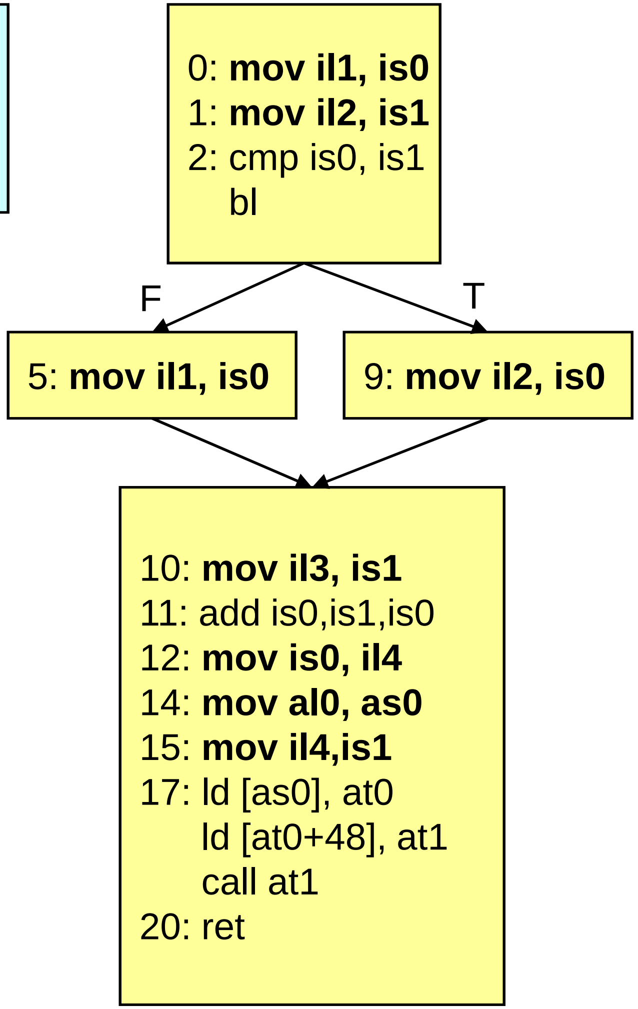
Java source code

```
int work_on_max(int x, int y, int tip) {
       int val = ( (x>=y) ? x : y ) + tip;
       return work (val);
}
```

Bytecode

```
0:   iload_1
1:   iload_2
2:   if_icmplt 9
5:   iload_1
6:   goto 10
9:   iload_2
10: iload_3
11: iadd
12: istore 4
14: aload_0
15: iload 4
17: invokevirtual
      <int work(int)>
20: ireturn
```

Translated RISC (SPARC) code



0: **mov il1, is0**
1: **mov il2, is1**
2: cmp is0, is1
bl

F

T

5: **mov il1, is0**

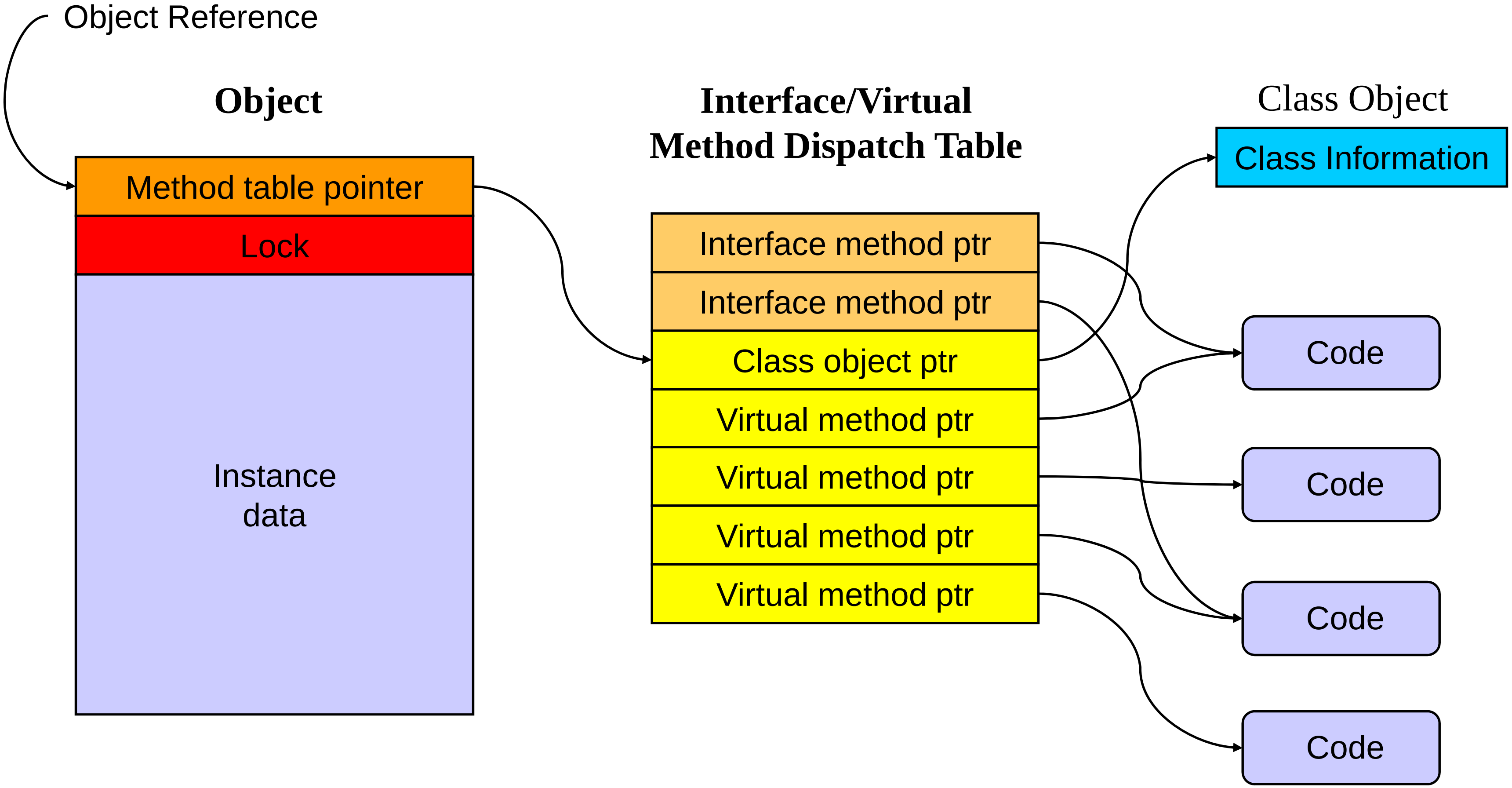
9: **mov il2, is0**

10: **mov il3, is1**
11: add is0,is1,is0
12: **mov is0, il4**
14: **mov al0, as0**
15: **mov il4,is1**
17: ld [as0], at0
ld [at0+48], at1
call at1
20: ret

JVM and SPARC Calling Convention

- JVM is a stack machine where an activation record is pushed/popped when a method is invoked/returned
  - Activation record has local variables and operand stack
  - Parameters become local variables of callee method
  - Return value is pushed on top of caller’s operand stack

An Example Object Model



Object Reference

Object

Method table pointer

Lock

Instance data

Interface/Virtual Method Dispatch Table

Interface method ptr

Interface method ptr

Class object ptr

Virtual method ptr

Virtual method ptr

Virtual method ptr

Virtual method ptr

Class Object

Class Information

Code

Code

Code

Code

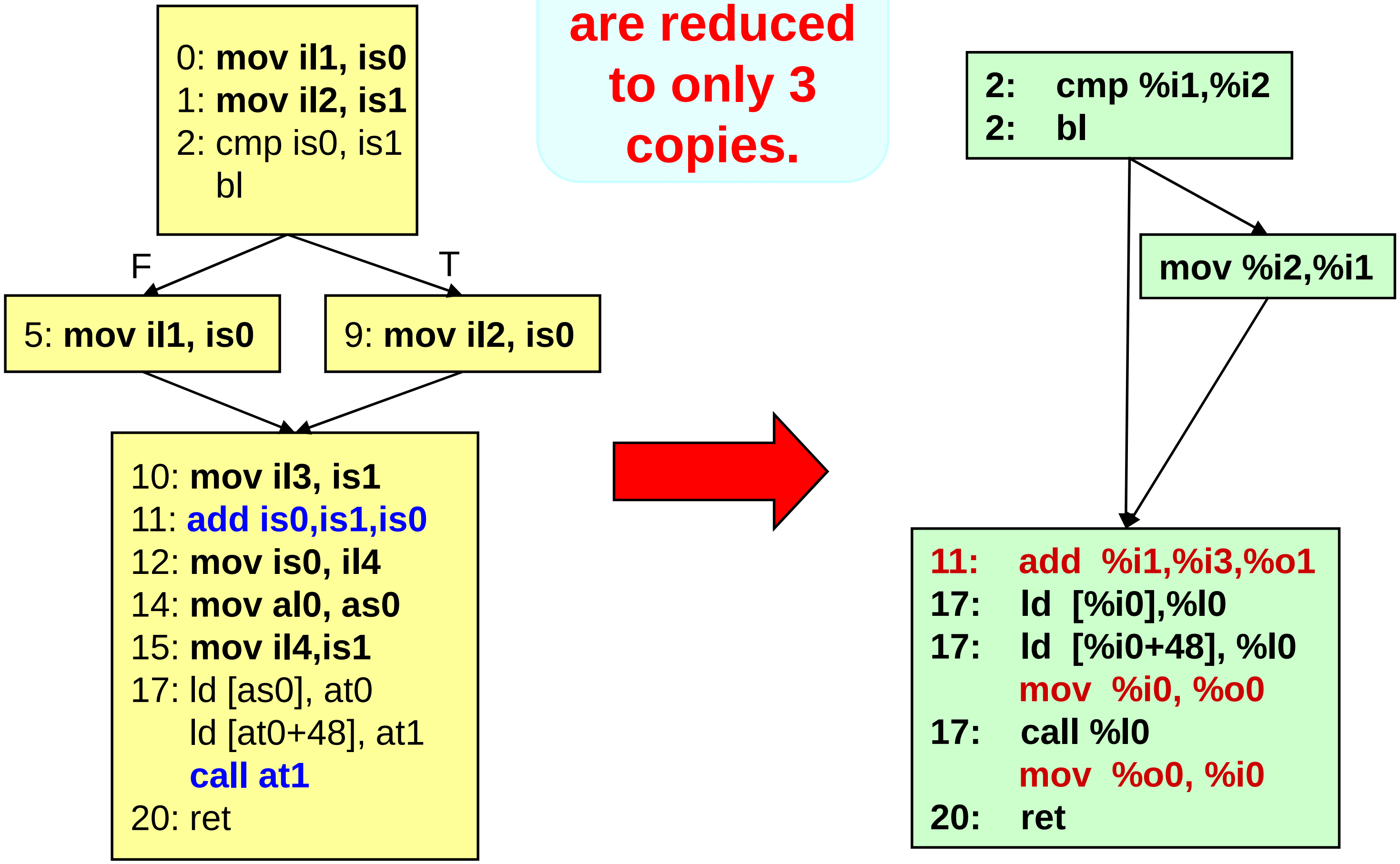
Issues in Translation: Optimization

- Naïve translation is not enough
  - Likely to generate inefficient, low-performance code
    - Mapping variables/stack locations to memory generates slow code
    - Naïve register allocation makes too many copies for pushes & pops
    - Null check for each reference? Bound check for each array access?
    - Method call overheads?

- Solution: code optimization
  - Quality of optimizations
  - Optimization overhead

An Optimization Example

8 copies are reduced to only 3 copies.



0: **mov il1, is0**
1: **mov il2, is1**
2: cmp is0, is1
bl

5: **mov il1, is0**

9: **mov il2, is0**

10: **mov il3, is1**
11: **add is0,is1,is0**
12: **mov is0, il4**
14: **mov al0, as0**
15: **mov il4,is1**
17: ld [as0], at0
ld [at0+48], at1
call at1
20: ret

2: cmp %i1,%i2
2: bl

mov %i2,%i1

11: add %i1,%i3,%o1
17: ld [%i0],%l0
17: ld [%i0+48], %l0
mov %i0, %o0
17: call %l0
mov %o0, %i0
20: ret

Optimizations in JITC & AOTC

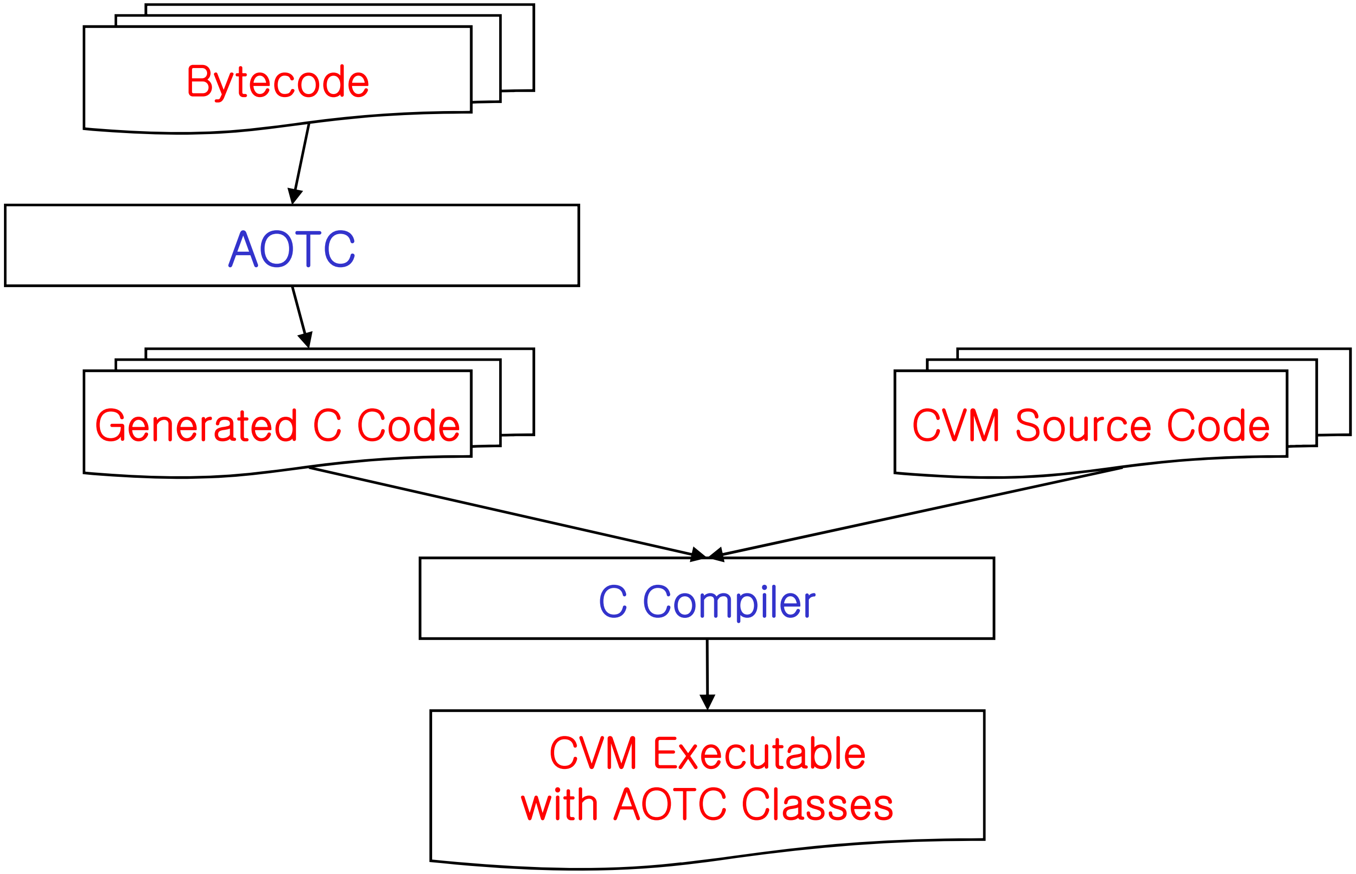
- AOTC
  - + No runtime translation/optimization overhead
  - + Full, off-line optimizations

- JITC
  - + Optimizations can exploit runtime information
    - e.g, Inlining of hot spot methods
  - + Transparent

Bytecode-to-C AOTC

- Translate bytecode to C code, which is then compiled by gcc optimizing compiler
  - Simpler to implement, better portability
  - Resort to gcc for code optimization
- AOTC performs Java-specific optimizations
  - To cope with the quality of bytecode-to-native code
    - check eliminations, OO optimziations, …
- Integration of VM components (GC, EH, ..)

Structure of the AOTC



An AOTC Example

Java source

```
package java.lang;
public class Math {
    public int max(int a, int b) {
        return (a >= b) ? a : b;
    }
}
```

Bytecode

```
 0:   iload_0      // push a
 1:   iload_1      // push b
 2:   if_icmplt 9  // if(a<b) goto9
 5:   iload_0      // push a
 6:   goto      10 // goto 10
 9:   iload_1      // push b
10:   ireturn      // return
```

Generated C file

```
int Java_java_lang_Math_max__II
    (CVMExecEnv *ee, int l0_int,
     int l1_int)
{
   int s0_int;
   int s1_int;

   s0_int = l0_int;       // 0:
   s1_int = l1_int;       // 1:
   if (s0_int < s1_int) { // 2:
      goto L9;
   }
   s0_int = l0_int;       // 5:
   goto L10;              // 6:
 L9:
   s0_int = l1_int;       // 9:
 L10:
   return s0_int;         // 10:
}
```

`s0_int, s1_int`: stack entries

`l0_int, l1_int`: Java local variables

JITC

- Many JITCs employ adaptive compilation
  - A method is first executed by the interpreter
  - If the method is determined to be a hot spot, it is JITCed
  - We will discuss hot spot detection on 6/4
- Many optimizations are done by the JITC
  - Register allocation
  - Method inlining
    - Method call overhead is very high in JVM
  - Traditional optimizations, …

Another S/W Solution: client-AOTC

- AOTC at the client for downloaded applications
  - Using JITC
  - Translate bytecode into machine code at a client device and save it in a permanent storage there
  - When the saved machine code is needed later during execution, it is loaded directly and run
  - cf. server-AOTC where AOTC occurs at the server

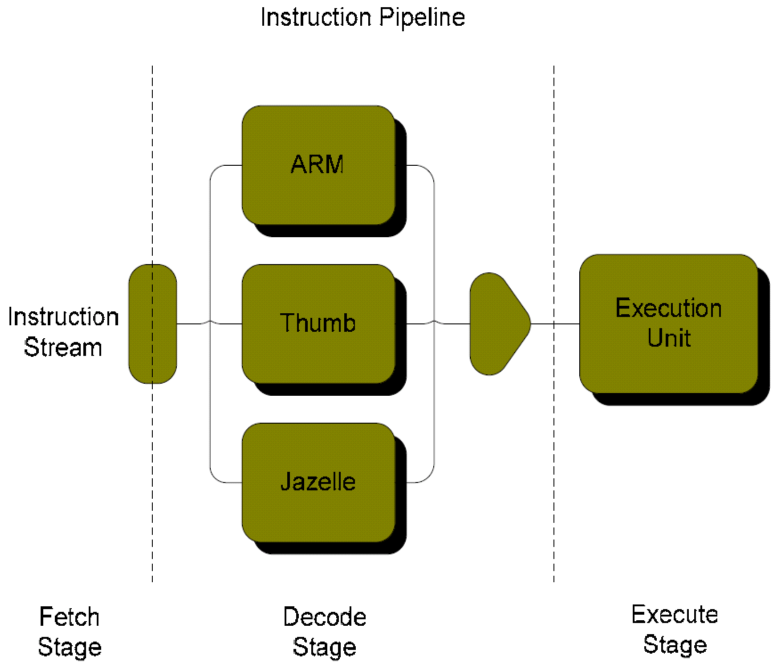
JITC-based Client-AOTC

- Translate using the JITC module at idle time
- We can save the JITC overhead when the machine code is loaded for execution
- Relocation is a major issue due to addresses that can change from run to run
  - Relocation information as well as machine code are saved

H/W Solution: Jazelle Approach

- Execute Java bytecode natively in hardware
  - Interoperate alongside existing ARM and Thumb modes
  - Fetch and decode bytecodes

  when branch-to-Java execute
  - Maintain Java operand stack
  - Assign six ARM registers
    - SP
    - Top 4 elements of stack
    - Local variable 0



H/W Solution: Jazelle Approach

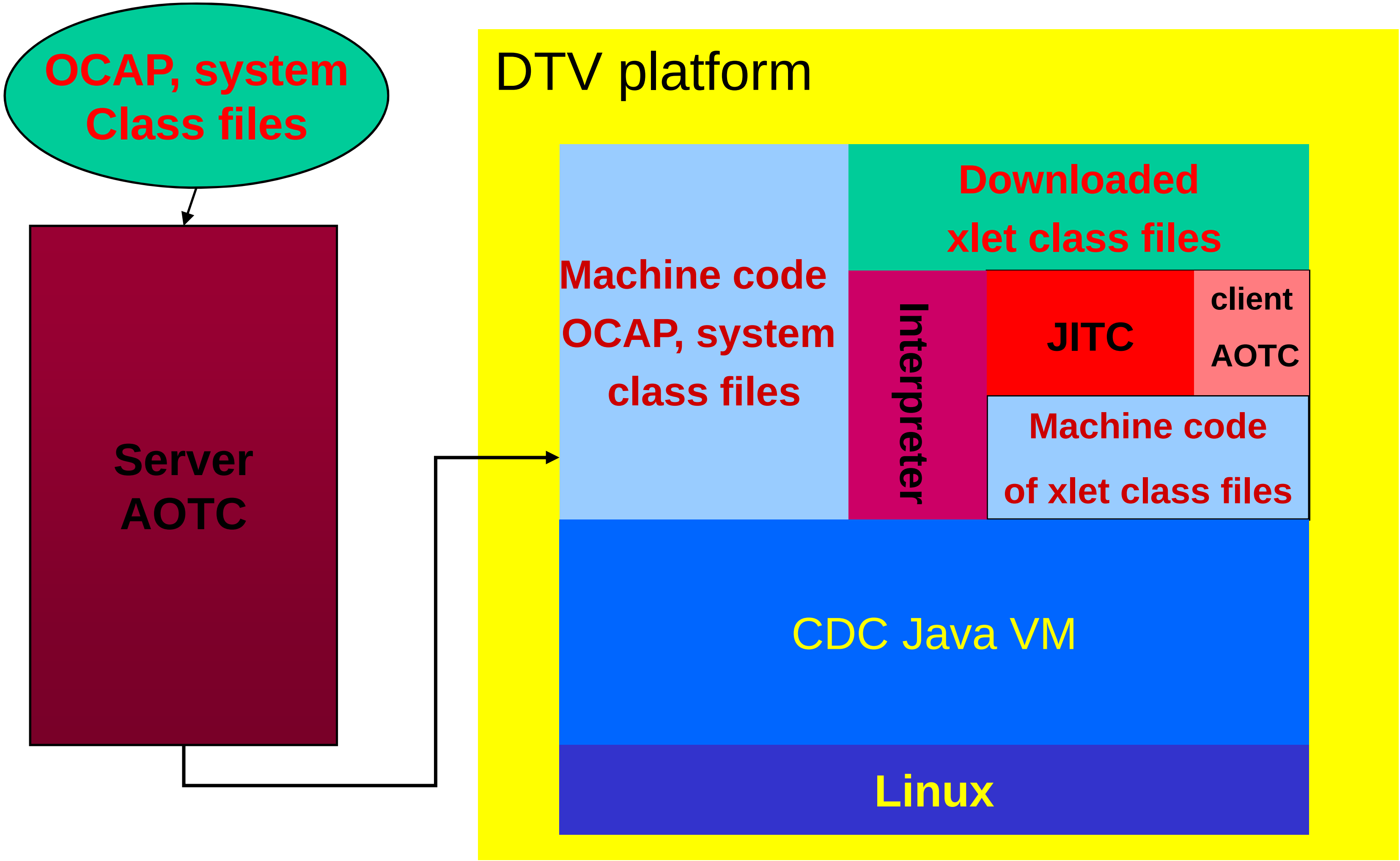
- ~60% of bytecode can be executed directly
  - Other complex bytecode must be emulated
- 2~4x performance compared to interpreter
  - For MIDP applications for cellular phones
  - Actual speedup is known to be less than this (max. ~2x)
  - Less performance advantage with faster CPUs

- Faster startup than JITC
- Less memory overhead than AOTC, JITC

Hybrid Acceleration Solution

- Hybrid solutions can also be useful – Why?
- Many embedded Java systems consist of
  - Java middleware installed statically at client devices
  - Java classes downloaded dynamically from service provider
  - e.g., OCAP (middleware) and xlet (dynamic classes) in DTV
  - e.g., MIDP (middleware) and midlet in cellular phones
  - e.g., BD-J (middleware) and xlet in Blu-ray disks
- AOTC for middleware and JITC/c-AOTC for dynamic classes would be a natural choice

Hybrid Solution Environment



OCAP, system
Class files

Server
AOTC

DTV platform

Machine code
OCAP, system
class files

Downloaded
xlet class files

Interpreter

JITC

client
AOTC

Machine code
of xlet class files

CDC Java VM

Linux