CSE 504: Compiler Design

Runtime Environments

Pradipta De
pradipta.de@sunykorea.ac.kr
Current Topic

- Procedure Abstractions
  - Mechanisms to manage procedures and procedure calls from compiler’s perspective

- Runtime Environment
  - Choices in converting source code to machine code
Runtime Environment

• Source language provides various abstractions
  – Compiler implements strategies to convert each construct into machine code
  – Runtime algorithms and data structures together implement the abstractions

All the code for runtime environments is \textit{generated at compile time}, but \textit{accessed at runtime}
Procedure Abstractions

• Procedure call
  – Standard techniques to invoke procedures
  – Standard linkage conventions
    • Allows invoking code written and compiled by other programmers (library routines, system services)

• Name space
  – Each procedure creates its own name space
    • New names can be used within this space

• External Interface
  – Technique to map names to values

Procedures create named variables and map them to virtual addresses
OS maps virtual addresses to physical addresses
Procedure Calls

- Typical procedural language
  - Procedure call transfers control from call site (caller) to the start of callee
  - On exit from callee, control returns to point of invocation
  - A call to a procedure leads to activation of the procedure

- The behavior can be modeled with a stack
  - Can handle recursive calls
Procedure calls in OOL

- Calling procedure is similar to procedural languages

- Key differences
  - How to name the callee
  - Mechanisms to locate the callee at runtime
Name Spaces

• Nested Lexical Scoping
  – In a given scope, each name refers to its lexically closest declaration
  – Scoping rules vary across languages

• Static Coordinates for names
  – For a name $x$ declared in scope $s$, its static coordinate is a pair $<l,o>$ where $l$ is the lexical nesting level of $s$ and $o$ is the offset where $x$ is stored in the scope’s data area.
program Main(input, output):
  var x1, y1, z1: integer;
procedure Fee1:
  var x2: integer:
  begin { Fee1 }
    x2 := 1;
    y1 := x2 * 2 + 1
  end:
procedure Fie1:
  var y2: real;
procedure Foe2:
  var z3: real:
  procedure Fum3
    var y4: real:
    begin { Fum3 }
      x1 := 1.25 * z3:
      Fee1:
      writeln('x = ', x1)
    end:
    begin { Foe2 }
      z3 := 1:
      Fee1:
      Fum3
    end:
    begin { Fie1 }
      Foe2:
      writeln('x = ', x1)
    end:
    begin { Main }
      x1 := 0:
      Fie1
    end.
<table>
<thead>
<tr>
<th>Scope</th>
<th>x</th>
<th>y</th>
<th>z</th>
</tr>
</thead>
<tbody>
<tr>
<td>Main</td>
<td>⟨1,0⟩</td>
<td>⟨1,4⟩</td>
<td>⟨1,8⟩</td>
</tr>
<tr>
<td>Fee</td>
<td>⟨2,0⟩</td>
<td>⟨1,4⟩</td>
<td>⟨1,8⟩</td>
</tr>
<tr>
<td>Fie</td>
<td>⟨1,0⟩</td>
<td>⟨2,0⟩</td>
<td>⟨1,8⟩</td>
</tr>
<tr>
<td>Foe</td>
<td>⟨1,0⟩</td>
<td>⟨2,0⟩</td>
<td>⟨3,0⟩</td>
</tr>
<tr>
<td>Fum</td>
<td>⟨1,0⟩</td>
<td>⟨4,0⟩</td>
<td>⟨3,0⟩</td>
</tr>
</tbody>
</table>
Activation Record

- Activation record supports the abstractions of procedure calls and scoped name spaces
- Each procedure call creates a new activation record
- Compiler’s tasks:
  - Store return address
  - Map actual parameters to formal parameters
  - Create storage space for variables declared in the callee’s local scope
  - Other information required by callee ➞ manage callee’s activation record
Activation Record

ARP points to the data area of the activation record.

Using positive and negative offsets with respect to ARP, one can locate various fields.

ARP is often stored in a designated register.
Initializing variables

- Local variable initialization
  - Compiler must ensure the assignment \( \rightarrow \) it is initialized at runtime

- Static variables
  - Lifetime of the variable is independent of the procedure scope
  - Store static variables outside the AR
  - Use separate static area
    - One per procedure or one for entire program
Allocating ARs

• Stack based allocation
  – Contents of AR are useful only during lifetime of the procedure
    • Variables cannot outlive the procedure lifetime

• Heap allocation
  – If a procedure can outlive its caller
  – If there is a reference to a local variable, implicitly or explicitly, which will be used after the lifetime of the procedure

• Coalescing ARs
  – If the compiler discovers a set of procedures that are always invoked in a fixed sequence, it may be able to combine their activation records
Name space of OOL

- Some OOLs have features that create name spaces which cannot be understood until runtime
  - Use runtime interpretation to runtime compilation
  - Example: Java

- Name resolution can depend on details of the code definitions and class structures
  - inheritance relations
  - Visibility modifiers

- At runtime, class structure can change → naming becomes more complex
  - Import class feature in Java
  - Open class hierarchy: can change class hierarchy at runtime (Java)
  - Closed class hierarchy: class structure fixed at runtime (C++)
Runtime Structures for OOL

• Object lifetimes need not match the invocation of any particular method, so their persistent state cannot be stored in some Activation Record.

• Each object needs its Object Record to hold its state
Single inheritance and open class structure

class Point {
    public int x, y;
    public void draw() {...};
    public void move() {...};
}

class ColorPoint extends Point {
    Color c; // local field of ColorPoint
    public void draw() {...}; // hide Point's draw()
    public void test() {...; draw();}; // local method
}

class C {
    int x, y; // local fields
    public void m() {
        int y; // local variable of m
        Point p = new ColorPoint(); // uses ColorPoint and, by
        y = p.x // inheritance, Point
        p.draw()
    }
}
Runtime structures that can result from instantiating three objects
Issues in OOL

• **Method Invocation**
  – Generating code to invoke a method in an object
  – Check visibility at the point of the call
  ➔ The compiler first looks in the method’s lexical hierarchy, then in the class hierarchy, and, finally, in the global scope

• **Object record layout**
  – Implementation must maintain consistent offsets, by name, up and down the class hierarchy
  – With closed structure, object layout possible at compile time
• Static versus Dynamic dispatch
  – Every method call requires one or more load operations to locate the method’s implementation
  – Closed structure: compiler can avoid the overhead by tying a concrete implementation to a method name
  – Exception: virtual method cannot be resolved \(\Rightarrow\) depends on the receiver’s class \(\Rightarrow\) dynamic dispatch
Issues in OOL

- **Multiple Inheritance**
  - A new class may inherit from several superclasses with inconsistent object layouts
  - Different superclass may assign conflicting offsets to their fields

Impose an ordering among the superclasses from which the child class inherits
Parameter Passing

• Call by value
  – Caller copies the value of the actual parameter to appropriate location

• Call by reference
  – Caller stores a pointer in the AR slot for each parameter

• Passing large values, like arrays, records or structures
  – Call by value is expensive
  – Preferred is call by reference
Procedure Calls: Return Value

• Return value needs space outside callee’s Activation Record
  – return value is used after callee has terminated

• Mechanism
  – Caller reserves space for return value in its AR
  – Callee uses the pointer to the space allocated for return value
  – If caller cannot determine size of the return value, then callee must allocate space (in heap)
  – If return value is smaller than the space allocated in the caller’s AR, callee can directly store it (eliminates reference based allocation)
Procedure Linking

- Procedure linkage ensures interoperability of compiled user code, and code from other sources (system libs, application libs)

**Precall**: starts preparing callee’s environment
**Postreturn**: Undo the actions of precall sequence
**Prologue**: Completes the task of creating callee’s environment
**Epilogue**: dismantles callee’s environment, and reconstructs caller’s environment
Code Shape

How does compiler choose among many alternative ways to implement each construct in machine language
Example of alternative choices

<table>
<thead>
<tr>
<th>Source Code</th>
<th>Low-Level, Three-Address Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>x+y+z</td>
<td>r1 ← rx+ry, r2 ← r1+rz, r1 ← ry+rz</td>
</tr>
</tbody>
</table>

Compiler must generate code to evaluate x+y+z
- Three possible ways to evaluate the expression using a sequence of binary operations
- Choice of left associative or right associative computation
  - What if y and z are constant values?
- Without interpreting the context, compiler cannot choose the best alternative
  - Use the context from surrounding code
Assigning Storage Locations

• Compiler must assign storage location to each variable

• Memory models
  – Memory-to-memory model: values are read from memory to register during computation, and stored back into memory
  – Register-to-register model: the number of registers is assumed infinite → compiler generates a virtual register name for computation
Compiler writer assumes a flat address space
- how to lay out, manipulate, and manage a program’s address is addressed by OS
Alignment

- Within each data area, compiler assigns an offset to each name.
  - Machine architecture determines layout

- Machine architecture can be byte-aligned or word-aligned
  - Byte aligned: Each word begins at byte boundary
  - Word aligned: Each word begins at word boundary
    - On 32-bit arch, word begins at 4 byte boundary
    - On 64-bit arch, word begins at 8 byte boundary
If a is in memory, it is loaded into register using the offset into the AR
Same for each variable

Will the demand for register change with the choice of generated code?

Other issue:
- How many instructions while accessing parameter values
- Generating code for mixed type expressions
  - Compiler may have to generate machine dependent code
Evaluate b*c first
Arrays

• How does a compiler translate a reference to an array element V[6]?

loadI @V \Rightarrow r@V \quad // \text{get } V\text{'s address}
subI r_1,3 \Rightarrow r_1 \quad // (offset - lower bound)
multi r_1,4 \Rightarrow r_2 \quad // x \text{ element length (4)}
add r@V, r_2 \Rightarrow r_3 \quad // \text{address of } V[i]
load r_3 \Rightarrow r_V \quad // \text{value of } V[i]

There are 3 arithmetic operations

Can the sequence of operations be improved?
Array indexing: alternatives

Replacing multiply with shift operations, when word length is power of 2

```
loadI   @V   ⇒ r@V   // get V’s address
subI    r_i, 3 ⇒ r_1  // (offset - lower bound)
lsiftI  r_1, 2 ⇒ r_2  // x element length (4)
loadAO  r@V, r_2 ⇒ r_V // value of V[i]
```

Assume a lower bound of 0 for the array ⇒ eliminates the sub operation

```
loadI   @V_0  ⇒ r@V_0 // adjusted address for V
lsiftI  r_i, 2 ⇒ r_1  // x element length (4)
loadAO  r@V_0, r_1 ⇒ r_V // value of V[i]
```
Storing Arrays

A matrix is expressed as,

\[
\begin{array}{cccc}
1,1 & 1,2 & 1,3 & 1,4 \\
2,1 & 2,2 & 2,3 & 2,4 \\
\end{array}
\]

Row major representation

\[
1,1 \ 1,2 \ 1,3 \ 1,4 \ 2,1 \ 2,2 \ 2,3 \ 2,4
\]

Column major representation

\[
1,1 \ 2,1 \ 1,2 \ 2,2 \ 1,3 \ 2,3 \ 1,4 \ 2,4
\]

Indirection vectors
Control Flow Constructs

- Control flow constructs are variants of “if-then-else” constructs
  - Arbitrary code blocks in then and else parts
- For simple then-else parts
  - Goal is efficient expression evaluation
- For complex then-else parts,
  - The evaluation order matters
Example

Target machine can issue 2 instructions per cycle

<table>
<thead>
<tr>
<th>Unit 1</th>
<th>Unit 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>comparison ⇒ r₁</td>
<td></td>
</tr>
<tr>
<td>(r₁) op₁ (¬r₁) op₁₁</td>
<td></td>
</tr>
<tr>
<td>(r₁) op₂ (¬r₁) op₁₂</td>
<td></td>
</tr>
<tr>
<td>(r₁) op₃ (¬r₁) op₁₃</td>
<td></td>
</tr>
<tr>
<td>(r₁) op₄ (¬r₁) op₁₄</td>
<td></td>
</tr>
<tr>
<td>(r₁) op₅ (¬r₁) op₁₅</td>
<td></td>
</tr>
<tr>
<td>(r₁) op₆ (¬r₁) op₁₆</td>
<td></td>
</tr>
<tr>
<td>(r₁) op₇ (¬r₁) op₁₇</td>
<td></td>
</tr>
<tr>
<td>(r₁) op₈ (¬r₁) op₁₈</td>
<td></td>
</tr>
<tr>
<td>(r₁) op₉ (¬r₁) op₁₉</td>
<td></td>
</tr>
<tr>
<td>(r₁) op₁₀ (¬r₁) op₂₀</td>
<td></td>
</tr>
</tbody>
</table>

No Branching
10 instructions

Using Branching
5 instructions

<table>
<thead>
<tr>
<th>Unit 1</th>
<th>Unit 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>compare &amp; branch</td>
<td></td>
</tr>
<tr>
<td>L₁: op₁ op₂</td>
<td></td>
</tr>
<tr>
<td>op₃ op₄</td>
<td></td>
</tr>
<tr>
<td>op₅ op₆</td>
<td></td>
</tr>
<tr>
<td>op₇ op₈</td>
<td></td>
</tr>
<tr>
<td>op₉ op₁₀</td>
<td></td>
</tr>
<tr>
<td>jumpI → L₃</td>
<td></td>
</tr>
<tr>
<td>L₂: op₁₁ op₁₂</td>
<td></td>
</tr>
<tr>
<td>op₁₃ op₁₄</td>
<td></td>
</tr>
<tr>
<td>op₁₅ op₁₆</td>
<td></td>
</tr>
<tr>
<td>op₁₇ op₁₈</td>
<td></td>
</tr>
<tr>
<td>op₁₉ op₂₀</td>
<td></td>
</tr>
<tr>
<td>jumpI → L₃</td>
<td></td>
</tr>
<tr>
<td>L₃: nop</td>
<td></td>
</tr>
</tbody>
</table>
Summary

• Compiler has different alternatives to generate code for various programming language constructs

• Choices are often dictated by
  – Machine architectures
  – Information required by the optimization step