COMP 520 - Compilers

Lecture 17 (Tue Apr 19, 2022)

Virtual Machines

- Reading
  - Skim Chapter 8
Topics

• review a simple code generation example

• Virtual machines
  – why target a virtual machine?
  – some virtual machines
    • mJAM (miniJava Abstract Machine)
    • JVM (Java Virtual Machine)
    • .NET and MSIL (Microsoft intermediate language)

• Nomenclature
  – TAM and mJAM are Abstract Machines
  – the more commonly used name is a Virtual Machines
• miniJava program

```
class PA4Tiny {
    public static void main(String[] args) {
        /* 1: simple literal */
        int x = 1;
        System.out.println(x);
    }
}
```

• mJAM assembler code

```
0   LOADL        0
1   CALL         newarr
2   CALL         L10
3   HALT (0)     
4   L10: LOADL    1
5   LOAD         3[LB]
6   CALL         println
7   RETURN (0)   1
```
What is a virtual machine

- A software interpreter $M$ for a (low-level) “machine” language
  - $M$ typically
    - interprets instruction codes in binary form
    - implements stack based execution of the operations
  - $M$ is written in some language (e.g. C) and compiled into code for physical machine $M'$
    - … which is interpreted by $M'$ hardware

- Cost of a virtual machine vs direct translation to hardware instructions
  a) program $p$ translated into $M$ and interpreted using $M'$
    - simple translation (stack model)
  b) program $p$ translated directly into $M'$ and interpreted using $M'$
    - complex translation (register model)

but … strategy (b) typically runs a factor of 2x - 20x faster than (a)
Why target a virtual machine? (1)

• Simplicity of code generation
  – stack machine
  – “convenient” and “appropriate” operations

• Portability
  – n languages, m target machines
    • using a virtual machine as intermediate target, need n translators plus m interpreters
  – write once, run anywhere
    • but performance is an issue

• Compactness
  – virtual machines can have very compact object code
    • amenable to caching (data cache)
Why target a virtual machine? (2)

- Type/memory safety (only for appropriate VM designs)
  - Can verify type correctness of code even when not generated by a compiler
    - Why useful
      - Assures memory safety
      - Better security

- Runtime flexibility and interoperability
  - dynamic loading of classes
  - standard libraries
  - native libraries

- Guard intellectual property
  - does not require distribution of source code
    - however, not very resistant to reverse engineering
Some virtual machines

- All of these execute stack-oriented binary code

<table>
<thead>
<tr>
<th>Virtual Machine</th>
<th>dynamic loading?</th>
<th>type checking?</th>
<th>types</th>
</tr>
</thead>
<tbody>
<tr>
<td>mJAM</td>
<td>no</td>
<td>some</td>
<td>boolean, int, classtypes, linear arrays of int or classtypes</td>
</tr>
<tr>
<td>JVM (1995)</td>
<td>yes</td>
<td>yes</td>
<td>boolean, byte, short, char, int, long, float, double class types, array types, interface types</td>
</tr>
<tr>
<td>.NET CLI (2000)</td>
<td>yes</td>
<td>yes</td>
<td>bool, char, string, object (unsigned) int{8,16,32,64}, float{32,64} tuple types $\alpha \times \beta \times \cdots$ typeref $\alpha$, functionref $\alpha \rightarrow \beta$, multidimensional array $\alpha$</td>
</tr>
</tbody>
</table>
JVM – Java Virtual Machine

• **Some properties of the JVM**
  – Input: classfiles
    • binary .class files – one per class, loaded on use
      – byte code for static initialization, constructors, and methods
      – complete names of external classes and methods used
      – types and names of public fields and methods provided by the class
      – types of parameters and local variables in each method
  
  – **Data memory**
    • all storage locations are 4 bytes wide
      – long and double values written into two consecutive locations (64 bit architectures)
    • expression evaluation stack and call stack are logically distinct
      – all object references have one level of indirection to simplify use of a compacting and generational garbage collector

  – **Instructions**
    • byte code has variable length operands
      – space efficient, but introduces overheads in decoding
    • types of arguments are encoded in the instruction name
      – iadd, ladd, fadd, dadd

COMP 520: Compilers
JVM: Dynamic loading

- Essentially the ability to execute separately compiled components using a load-on-use strategy
  - JVM response to a method invocation in a class that is not loaded
    - Run the class loader on the appropriate class file
      - load code, constants, and type information into JVM
      - add class descriptor verify loaded code (option)
    - Link invocation(s) to methods
      - connect symbolic references to actual code location
      - or through dynamic dispatch table
    - Initialization
      - run static initialization for loaded class
  - Execution
    - run requested method in loaded class

- Dynamic loading may invalidate assumptions made in previously loaded methods
  - ex: all invocations are monomorphic (always the same dynamic type)
JVM Architecture
Microsoft .NET Common Language Interface

• .NET can be described as several things:
  – An application development framework with tons of windows-centric capabilities
  – A comprehensive communication and interoperation layer, involving XML, SOAP, COM, and other technologies
  – Microsoft’s answer to Sun’s Oracle’s Java / JVM
  – A cool virtual machine
  – A marketing term applied to things that Microsoft creates

• These are all correct!
Components of the .NET Framework

Microsoft .NET Framework

- Common Language Specification
  - System.Web
    - Web Services
    - Web Forms
    - ASP.NET Application Services
  - System.WinForms
    - Controls
    - Drawing
    - Windows Application Services

- Data and System Base Classes
  - ADO.NET
  - XML
  - SQL
  - Threading
  - IO
  - Net
  - Security
  - Service Process

- Common Language Runtime
  - Security Engine
  - Thread Support
  - Exception Handler
  - IL to Native Compiler
  - Type System
  - Garbage Collector

MSIL
.NET Application Development

• .NET is made up of two parts:
  – CLI (Common Language Interface)
    • CLS (Common Language Specification)
    • CTS (Common Type System)
    • MSIL (Microsoft Intermediate Language)
  – VES (Virtual Execution System)
    • CLR (Common Language Runtime)
    • JIT compiler

• Some of these have counterparts in Java/JVM, while some do not

• .NET languages supported
  – New with .NET
    • C# (enhanced Java with lots of windows libs)
    • F# (functional language for scientific computing)
  – Existing
    • C++ (sort of), VB, Managed Jscript (not JavaScript), IronPython, …
    • no Java in .NET
Some properties of the CLI

- Input: assemblies
  - roughly comparable to JAR files – collection of classes
  - binary or text form
    - Microsoft Intermediate Language (MSIL)

No public interpreter (except Xamarin Mono), but well-defined virtual machine

- Data memory
  - all storage locations have a known type and size
    - actual layout in memory can only be partially controlled
  - expression evaluation, method arguments, and locals are in logically distinct areas
    - each has a static size that is part of the method
    - no reuse of locals storage for locals with disjoint lifetimes

- Instructions
  - types of values and frame safety are fully specified and checked
    .method static int32 foo(int32 x) {
      .maxstack 1
      .locals (int32 z)
      ldarg x; stloc z; ret z
    }
Example MSIL code

All assemblies can be fully type checked

```csharp
.assembly PrintString {} /* Console.WriteLine("Hello, World") */
.method static public void main() il managed
{
    .entrypoint // this function is the application entry point
    .maxstack 8

    // load string onto stack
    ldstr "Hello, World"

    // call static System.Console.WriteLine function
    call void [mscorlib]System.Console::WriteLine(class System.String)

    ret
}
```
What would it take to type-check mJAM code?

- What are the types of values resulting from the following operations?
  - LOADL 12324 ; what type? int or address or bool?
  - LOAD 5[LB] ; what type? must know the types of the local vars
  - LOAD -3[LB] ; what type? must know the types of the parameters
  - CALL ADD ; int if top two elts on stack are always int, else type error
  - LOAD 4[LB] ; could be object address
  - LOADL 3 ; field index – could be arbitrary int expr
  - CALL ADD ; this is an uncontrolled address – type error
  - LOADI ; what is the result type?

- Requirements
  - all objects, methods, and local variables must be typed
    - what about different type locals with disjoint lifetimes at the same offset?
  - operations must be type specific
  - use fieldref and arrayref instead of arithmetic on addresses
.NET vs JVM

**.NET Advantages**
- Designed to work with many source languages, rather than just one
  - C#, C++ (kinda), VB, Haskell, …
  - Supports values of arbitrary size, function values, non-local reference, union types, integer arithmetic with overflow detection, etc.
- More extensive class library than Java
  - E.g. record types, function types
- Type system permits “escapes” for low-level programming

**.NET Disadvantages**
- Less emphasis on exception handling (perhaps advantage)
- Java is much more widely accepted and used, with a much larger installed user base
- The CLR is very Windows-centric
Dynamic Compilation

• What is it?
  – compilation of virtual machine code to native machine code at start of execution
    • Examples
      – Virtual Execution System in Microsoft CLI
      – Open Runtime Platform (Intel)

• Why useful?
  – avoids premature commitment to execution platform
  – compile only what is needed

• Managed Runtime Environment
  – key feature is typed virtual machine code
Just-in-time Compilation (JIT)

- **What is it?**
  - fast compilation of virtual machine code to native machine code
    - typically on a method-by-method basis
    - use simple optimization strategy to limit compilation overheads
  - used selectively within virtual machine interpreter
    - rule of thumb
      - >80% of the time is spent in <20% of the program
      - recall 10x - 20x penalty for interpretation
    - cost-effectiveness
      - JIT is worthwhile if
        » interpret-time(P) > compile-time(P,P') + native-execution(P')
      - HotSpot analysis: compile only “worthwhile” methods
        » interpreter tracks time spent in each method
        » invokes JIT compilation for the method when a threshold is reached
        » substitutes compiled code for interpreted code
JIT Advantages

• Some aspects of the environment are fixed when JIT compiler runs
  – Allows the compiler to make assumptions that cannot be made by static compilers
    • The dynamic type of some values may be known
      – enables method inlining
    • The sizes of some arrays may be known
    • Array bound / null pointer checks
    • Dynamic method inlining

• Compiler can make probabilistic optimizations based on profiling information
  – Which loops run many iterations
  – Which conditions rarely are true

• Methods that are never executed are never compiled, reducing compilation time and executable size in memory
Compiling Java for high performance is difficult

- Distinguishing features of Java
  - many dynamic checks
    - array indexing, type casting
  - many heap allocated values and garbage collection
    - requires maintenance of accessibility, possible indirection for compacting GC
- good software engineering practices lead to
  - small method bodies
    - hard to amortize method invocation overhead
  - virtual method invocation
    - makes it difficult to “inline” method bodies
- dynamic class loading
  - can change assumptions
    - e.g. method p() always appears to call a particular instance of method q()
    - after loading a new class that is also a client of p(), this may no longer be true
- Native implementation of low-level libraries can be a big help
Strategies for JIT compilation

• Shared execution stack for
  – JVM
  – JIT-compiled code
  – native methods

• Streamlined representation of values
  – no indirection for (some) allocated objects

• optimizations
  – register allocation
  – method inlining
  – array access optimization
  – bounds check optimization
  – common subexpression elimination