Chapter 1

JIT

- Also the vm instructions are portable
- Compiler does not have many information like ... A JIT compiler runs after the program has started and compiles the code (usually bytecode or some kind of VM instructions) on the fly (or just-in-time, as it's called) into a form that's usually faster, typically the host CPU's native instruction set. A JIT has access to dynamic runtime information whereas a standard compiler doesn't and can make better optimizations like inlining functions that are used frequently.
- In a stack based machine, opeartions are done by pshing operands onto the stack and then performing the opeartions on the opearnds on the top of the stack. Stack based architecture got revived with the intro of JVM whih is a software interpretter for java bytecodes (an intermedate lang prducrd by java compiler) An interpretter provides software compatibility across multiple platforms. TO overcome the high perf pnalty of interpretation JIT compilers are creted.
- for dynamically typed language interpretation is important and to overcome the limitations of interpretation.
- (HOW)In practice, methods are not compiled the first time they are called. For each method, the JVM maintains a call count, which is incremented every time the method is called. The JVM interprets a method until its call count exceeds a JIT compilation threshold. Therefore, often-used methods are compiled soon after the JVM has started, and less-used methods are compiled much later, or not at all. The JIT compilation threshold helps the JVM start quickly and still have improved performance. The threshold has been carefully selected to obtain an optimal balance between startup times and long term performance.

  After a method is compiled, its call count is reset to zero and subsequent calls to the method continue to increment its count. When the call count

of a method reaches a JIT recompilation threshold, the JIT compiler compiles it a second time, applying a larger selection of optimizations than on the previous compilation. This process is repeated until the maximum optimization level is reached. The busiest methods of a Java program are always optimized most aggressively, maximizing the performance benefits of using the JIT compiler. The JIT compiler can also measure operational data at run time, and use that data to improve the quality of further recompilations

This is in contrast to a traditional compiler that compiles all the code to machine language before the program is first run.

Chapter 2

SLIDE 1

Dynamic languages like javascripts are popular: expressie, accessible to non experts and make deployment easy as dstributing a source file. BUT more difficult to compile than statically typed languages. Even static type inferencing is too expesive to help for highly interactive environment

- Compilers for statically typed languages rely on type information to generate efficient machine code. In a dynamically typed programming language such as JavaScript, the types of expressions may vary at runtime. This means that the compiler can no longer easily transform operations into machine instructions that operate on one specific type. Without exact type information, the compiler must emit slower generalized machine code that can deal with all potential type combinations. While compile-time static type inference might be able to gather type information to generate optimized machine code,

2.1 static vs dynamic type

- In a **statically typed language**, every variable name is bound to a type (at compile time, by means of a data declaration) (The binding to an object is optional if a name is not bound to an object, the name is said to be null.)

  Once a variable name has been bound to a type (that is, declared) it can be bound (at run time) (via an assignment statement) only to objects of that type; it cannot ever be bound to an object of a different type. An attempt to bind the name to an object of the wrong type will raise a type exception. Also a type specified what operation that u are permitted to do on that varible

  In a **dynamically typed language**, every variable name are bound to objects at execution time by means of assignment statements, and it is

possible to bind a name to objects of different types during the execution of the program.

Python is a dynamically-typed language. Java is a statically-typed language.

- In a weakly typed language, variables can be implicitly coerced to unrelated types (like ints and strings), whereas in a strongly typed language they cannot, and an explicit conversion is required.

  (Note that I said unrelated types. Most languages will allow implicit coercions between related types for example, the addition of an integer and a float. By unrelated types I mean things like numbers and strings.)

  In a typical weakly typed language, the number 9 and the string 9 are interchangeable, and the following sequence of statements is legal.

  a = 9 b = "9" c = concatenate(a, b) // produces "99" d = add(a, b) // produces 18 In a strongly typed language, on the other hand, the last two statements would raise type exceptions. To avoid these exceptions, some kind of explicit type conversion would be necessary, like this.

  a = 9 b = "9" c = concatenate(str(a), b) d = add(a, int(b))

  Python, java is a strongly typed language. perl.Javascript is a weakly-typed language.

- A third distinction may be made between manifestly typed languages in which variable names must have explicit type declarations, and implicitly typed languages in which this is not required. Implicitly-typed languages use type inferencing rather than data declarations to determine the types of variables.

  Most static languages, like Java, are also manifestly typed. Haskell can infer the type of any variable based on the operations performed on it, with only occasional help from an explicit type.

static type inferencing may give conservative results ... many integers adds and one fp add.... static inferencing generalizes the type to fp.. which imples more runtine cost compared to the frequent case

Chapter 3

Basic approach

Mixed mode execution approach:

- first compiled into Bytecode
- starts running a BC interpretter (Spider Monkey)
- As program runs hot bytecode sequesnce (traces) is identified, which are then recorded and compliled to fast naive code.
- after compiling traces, native code is called instead of interpretation of the bytecode

Unlike method based dynamic compilers, their compilr operates at loop level granularity, because

- interpreter spends most of time in loops.
- "hot" loops are expected to be mostly type-stable
- thus loops can be type specialized The above guarentees small number of type specialized complied traces. Else /* If control flw is different or a value of diff type is generated */ use guards to maintain program integrity.

trace recording and patching for non loop and nested loop case

Aborting the recording (in any case trace recorder is disengaged and regular interpretion resumes) ========== exception , trace recorder is disengaged and regular interpretion resumes. small memory limiting the recording of traces