COMPSCI 590S Systems for Data Science **Fall 2017**

Lecture 9: Compiled Languages: Java & C++

Lecturer: Emery Berger *Scribe(s): Karla Villata, Edward Pantridge*

9.1 Lock Recap

In the previous lecture locks were discussed in depth but there are a couple of points that were left out of the discussion:

- **Two Phase Locking (2PL)** is the idea that you acquire all locks then you release all them at once. This guarantees atomicity. When dealing with deadlocks, there are two main approaches optimistic and pessimistic. The first one hopes for the "best", no deadlocks will occur while the latter one assumes the worst case scenario which is that a deadlock is bound to occur therefore to avoid it you acquire all locks.
- **Avoiding deadlocks** If there is a cycle in locking graph, this is known as deadlock. One of the typical solutions for this is **canonical lock ordering** which means that you acquire the locks in increasing order, e.g. $lock_1$, $lock_2$, $lock_3$ and release then in decreasing order. This ensures that no cycles are form and hence is deadlock-free.

If you would like to read a thread of canonical lock ordering on the kernel, check out the following link. However be advised of the presence of profanity: `https://lkml.org/lkml/2017/10/4/580`

Going back to databases (DB), think of how would a database handle the case of getting the same commit from two different sources. Conflicts of this matter means that one of the transactions must be avoided.

9.2 Correctness Vs. Performance

The tradeoff between high and low level programming languages is **correctness vs performance**. A high level language, is correct and easy to program at the expense of its performance. Example of high level programmig languages are:

- SQL
- Haskell
- Scala

The tradeoff of correctness is low error bound.

Segmentation fault SEGV - Happens when a program write to an address that is outside of the scope of the memory that has been allocated to it. This problem is specific to low-level programs since they have access to memory. You cannot do this is Java or Scala for example.

Java is considered to be a *safe language*. Defining safety in this sense is tricky but the idea is that it is considered this way because it provides safe data types and has no memory access.

A note about SQL: SQL is so high level that is not a complete programming language. More formaly, it is not **"turing complete"**, which means that it cannot express all computations. It is common for SQL databases to extend the language to support some kind of user defined function (Java stored procedures in Orcacle, C# user defined operations in Microsoft Server).

9.3 Java

Java was initially an interpreted language designed to safely run in the browser. It was initially developed by Sun Microsystems [1]. The language allowed for people to write Java "Applets" which could be embedded in a web page. The applets were safe, but slow. Part of getting Applets to work correctly was achieving a portable language (aka. write once, run anywhere (WORA)). This property of Java resulted in its widespread use in servers.

Eventually Oracle bought Sun Microsystems because they wanted to add support for "stored procedures" in their databases. These are arbitrary functions written in Java that can safety be run to manipulate the database.[2] Ever since then Oracle has been a been trying to find new ways to profit on the widespread use of Java. This includes a lawsuit with Google over the use of Java in the android operating system. To get around these legal problems, Google created their own Java Virtual Machine called Dalvik which runs a different format of Java bytecode. (I think they recently made it faster and renamed Dalvik to Android Runtime). The main difference between the JVM and Dalvik is that Dalvik is register oriented and the Oracle JVM is stack oriented.

Somewhere along the way Professor Berger got hit with two bogus cease and desist orders. One patent infringement for a memory allocator called Hoard, and one trademark infringement for another memory allocator called DieHard by the *"household name"* DieHard batterys.

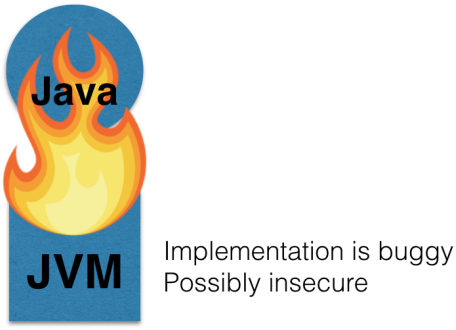
9.4 Java vs C++

Despite C++ being a lower level abstraction that has the potential for better performance, the JVM is still home to many high performance projects, such as Hadoop, Spark, Clojure, Scala, and DynamoDB.

Java	C/C++
JIT Compilation	AOT Compilation
Indirect access to memory	Direct access to memory (possibly unsafe)
Bounds checks	Explicit memory management
No explicit access to memory	
Code executes on write and read	
new (*delete* is handled by garbage collection.	

[1] I once heard a story that the reason the famous Java IDE is named Eclipse is because they block out the sun. I am eager to hear Professor Berger confirm or deny such folklore

[2] This is similar to User Defined Operations written in C# for Microsoft's SQL Server.



9.4.1 JIT Compilation

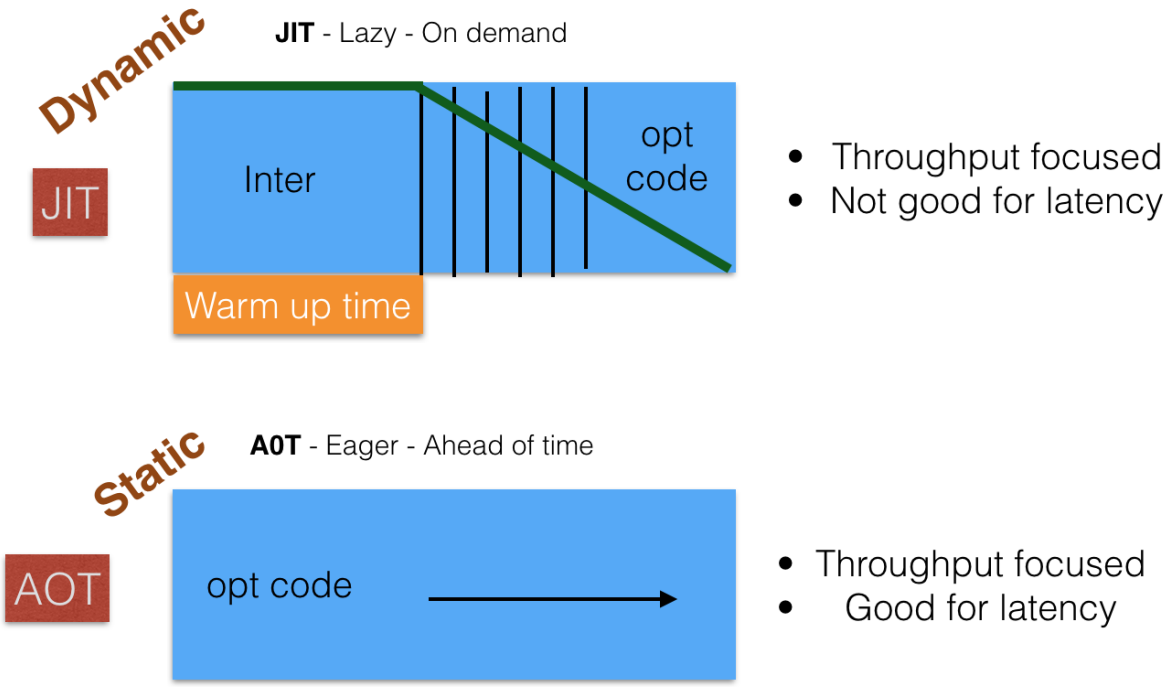
Just-In-Time. Lazy. On-demand. Designed for good throughput, but can have high latency. The reason for the high latency is a "warm-up" period where the JIT compiler is determine which code is used frequently enough to merit compilation.

This is the approach used by the JVM to compile Java bytecode to machine instructions. The JVM uses "hotspots" to find and compile code that is being used frequently.

9.4.2 AOT Compilation

Ahead-of-Time. Eager. Designed for good throughput and low latency.

This type of compilation is not possible in Java because Java is a dynamic language. In particular, Java supports things like class loaders and reflections which make AOT compilation impossible.



9.5 Compiler Optimizations

There are different optimizations performed by compiler:

9.5.1 Hoisting

Hoisting refers to the moving of operations found in iterative control structures (ie. loops) that do not change each iteration so they are outside of the control structure. Consider the following example.

```
for (int i = 0; i < 10; i++) {
    int z = 5 + 10;
    // more logic here ...
}
```

In the above code block, the variable z is the same at each iteration. The compiler can "hoist" the computation of z to be above the loop to avoid computing z more times than necessary. The resulting computation would be equivalent to the following.

```
int z = 5 + 10;
for (int i = 0; i < 10; i++) {
    // more logic here ...
}
```

9.5.2 Inlining

This refers to the task of merging in a single function functions contained within other functions. For example:

```
f() {       g() {       h() {
  A           B           C
  g();        h();      }
}           }
```

This can all be combined into a single one:

```
f(){
  A
  B
  C
  D
  E
}
```

The above combines all computation in a single, more efficient step. Inlining exposes optimization opportunities. If done too aggressively, it can slow down the code rather than speeding it up.

9.5.3 Constant Propagation

Constant propagation refers to the substitution of successive assignments to the same variable with a single condensed assignment. Consider the following example of a series of statements inside a function (possibly after inlining):

```
f(){
  x = 1;        // assignment1
  x = 2;        // assignment2
  x = x + 2;    // assignment3
  x = x * x;    // assignment4
}
```

It is clear that *assignment1* is not necessary given that it is followed by *assignment2*. Also, we can propagate the value of x through assignments 3 and 4 to get a single equivalent statement. After constant propagation, the compiled definition of $f()$ would be equivalent to the following.

```
f(){
  x = 16;
}
```

We also mentioned "copy propagation" as another type of compiler optimization, although we did not discuss it in depth yet.

9.6 Issues with Java

Java has a few of sources of overhead which are easier to avoid in c/c++. On average, Java is 5% to 15% slower than c/c++. In addition, for matrix operations Java is vastly slower than c/c++ due to Java's poor array implementation.

9.6.1 Bounds Checks

When indexing an array, the Java Virtual Machine or JVM must check that the index is within the range $[0, x)$ where x is the length of the array. If this is not the case, then an exception is triggered.

9.6.2 Straggler Problem

When a set of tasks is running in parallel, they must be synced at the end. If one task runs significantly longer than the others, it is considered a "straggler". We can think back to the Master - Worker framework. The workers refer back to the master once they all have completed their task but if one of the workers hits the garbage collector, the rest of the workers must wait for this "straggler" befored they all get synced. This indeed slows down the pipeline since there is a high probability that at least one worker will hit the GC.

9.6.3 Space

The garbage collection in Java has considerable memory requirements.