Bytecode Execution

We return to bytecode: there are several different ways bytecode can get executed

Bytecode Execution

We return to bytecode: there are several different ways bytecode can get executed

Java is bytecode, and has separate compile and runtime systems (`javac` vs. `java`)

Bytecode Execution

We return to bytecode: there are several different ways bytecode can get executed

Java is bytecode, and has separate compile and runtime systems (`javac` vs. `java`)

Its main objective is machine-independent (byte)code

Bytecode Execution

We return to bytecode: there are several different ways bytecode can get executed

Java is bytecode, and has separate compile and runtime systems (`javac` vs. `java`)

Its main objective is machine-independent (byte)code

Java compilers are quite heavyweight (i.e., large and relatively slow), but the code they produce is only moderately well optimised

Bytecode Execution

We return to bytecode: there are several different ways bytecode can get executed

Java is bytecode, and has separate compile and runtime systems (`javac` vs. `java`)

Its main objective is machine-independent (byte)code

Java compilers are quite heavyweight (i.e., large and relatively slow), but the code they produce is only moderately well optimised

We have a compile $\rightarrow$ run $\rightarrow$ debug $\rightarrow$ compile $\rightarrow \ldots$ cycle of development

Bits and Pieces

Bytecode Execution

Python is bytecode and has an integrated compile and runtime system

Bits and Pieces

Bytecode Execution

Python is bytecode and has an integrated compile and runtime system

Each time a Python program (the source text) is run it is first compiled, then executed: this helps rapid development, as above

Bits and Pieces

Bytecode Execution

Python is bytecode and has an integrated compile and runtime system

Each time a Python program (the source text) is run it is first compiled, then executed: this helps rapid development, as above

So total run time $=$ compile time $+$ execution time

Bits and Pieces

Bytecode Execution

Python is bytecode and has an integrated compile and runtime system

Each time a Python program (the source text) is run it is first compiled, then executed: this helps rapid development, as above

So total run time $=$ compile time $+$ execution time

It helps that the Python compiler is really fast!

Bits and Pieces

Bytecode Execution

But this does mean the Python bytecode produced is not optimised

Bits and Pieces

Bytecode Execution

But this does mean the Python bytecode produced is not optimised

So Java code should run faster than Python code

Bits and Pieces

Bytecode Execution

But this does mean the Python bytecode produced is not optimised

So Java code should run faster than Python code

But I've seen the same program written in Java and in Python, and the Python compiled and ran to completion in less time than the `java` runtime managed to even start and load the code

Bits and Pieces

Bytecode Execution

But this does mean the Python bytecode produced is not optimised

So Java code should run faster than Python code

But I've seen the same program written in Java and in Python, and the Python compiled and ran to completion in less time than the `java` runtime managed to even start and load the code

And that's ignoring the time the Java compiler took beforehand

Bits and Pieces

Bytecode Execution

We have an (apparent) run $\rightarrow$ debug $\rightarrow$ run $\rightarrow \ldots$ cycle of development

Bits and Pieces

Bytecode Execution

We have an (apparent) run $\rightarrow$ debug $\rightarrow$ run $\rightarrow \ldots$ cycle of development

Python is about development time, not execution time

Bits and Pieces

Bytecode Execution

We have an (apparent) run $\rightarrow$ debug $\rightarrow$ run $\rightarrow \ldots$ cycle of development

Python is about development time, not execution time

The compiled form of the Python program may or may not be kept around for the next run

Bits and Pieces

Bytecode Execution

We have an (apparent) run $\rightarrow$ debug $\rightarrow$ run $\rightarrow \ldots$ cycle of development

Python is about development time, not execution time

The compiled form of the Python program may or may not be kept around for the next run

Perl and Lua are similar to Python in this execution

Bits and Pieces

Bytecode Execution

We have an (apparent) run $\rightarrow$ debug $\rightarrow$ run $\rightarrow \ldots$ cycle of development

Python is about development time, not execution time

The compiled form of the Python program may or may not be kept around for the next run

Perl and Lua are similar to Python in this execution

Exercise Reflect on reasons why it might not be a good idea to keep the compiled version

Bits and Pieces

Bytecode Execution

We have an (apparent) run $\rightarrow$ debug $\rightarrow$ run $\rightarrow \ldots$ cycle of development

Python is about development time, not execution time

The compiled form of the Python program may or may not be kept around for the next run

Perl and Lua are similar to Python in this execution

Exercise Reflect on reasons why it might not be a good idea to keep the compiled version

Exercise There is currently a drive in the Python world to produce better optimised bytecode. Read about this

Bits and Pieces

Interpreters

As regards bytecode execution, some systems initially interpret the bytecode but keep note of those parts of code that are used frequently, e.g., bodies of loops

Bits and Pieces

Interpreters

As regards bytecode execution, some systems initially interpret the bytecode but keep note of those parts of code that are used frequently, e.g., bodies of loops

They then dynamically (during runtime) compile just those parts to machine code so they will subsequently execute more quickly

Bits and Pieces

Interpreters

As regards bytecode execution, some systems initially interpret the bytecode but keep note of those parts of code that are used frequently, e.g., bodies of loops

They then dynamically (during runtime) compile just those parts to machine code so they will subsequently execute more quickly

That is, they take time away from running your code to compile bits of your code

Bits and Pieces

Interpreters

As regards bytecode execution, some systems initially interpret the bytecode but keep note of those parts of code that are used frequently, e.g., bodies of loops

They then dynamically (during runtime) compile just those parts to machine code so they will subsequently execute more quickly

That is, they take time away from running your code to compile bits of your code

They are taking the gamble this will improve overall runtime

Bits and Pieces

Interpreters

As regards bytecode execution, some systems initially interpret the bytecode but keep note of those parts of code that are used frequently, e.g., bodies of loops

They then dynamically (during runtime) compile just those parts to machine code so they will subsequently execute more quickly

That is, they take time away from running your code to compile bits of your code

They are taking the gamble this will improve overall runtime

This is *Just in Time* (JIT) compilation

Bits and Pieces

Interpreters

As regards bytecode execution, some systems initially interpret the bytecode but keep note of those parts of code that are used frequently, e.g., bodies of loops

They then dynamically (during runtime) compile just those parts to machine code so they will subsequently execute more quickly

That is, they take time away from running your code to compile bits of your code

They are taking the gamble this will improve overall runtime

This is *Just in Time* (JIT) compilation

Examples include Java and JavaScript VMs

Bits and Pieces

Bytecode Execution

Occasionally JIT can produce faster running code than simple static compilation as the compilation process can be informed by the profile information gained from running the program, e.g., which methods are actually being chosen and called

Bits and Pieces

Bytecode Execution

Though this does incur some runtime overhead: compilation is not cheap, and unless you are careful it can dominate the running time in a short-lived program

Bits and Pieces

Bytecode Execution

Though this does incur some runtime overhead: compilation is not cheap, and unless you are careful it can dominate the running time in a short-lived program

You might argue that it doesn't matter in a short-lived program as it will soon be finished anyway

Bits and Pieces

Bytecode Execution

Though this does incur some runtime overhead: compilation is not cheap, and unless you are careful it can dominate the running time in a short-lived program

You might argue that it doesn’t matter in a short-lived program as it will soon be finished anyway

However if you run that program many times it does add up to a lot of extra CPU cycles (i.e., energy) as the same JIT compilations are done and re-done every run time

Long-running programs benefit a lot, though

Bits and Pieces

Bytecode Execution

Long-running programs benefit a lot, though

Despite the overheads of monitoring the execution of the code to determine which parts to compile and actually doing the compilation

Bits and Pieces

Bytecode Execution

Long-running programs benefit a lot, though

Despite the overheads of monitoring the execution of the code to determine which parts to compile and actually doing the compilation

Exercise Look at the optimisations that modern implementations of Java and JavaScript use

Bits and Pieces

Bytecode Execution

Another approach is *ahead of time* (AOT) compilation

Bits and Pieces

Bytecode Execution

Another approach is *ahead of time* (AOT) compilation

This takes bytecode and further compiles it for the specific OS and hardware at *installation time*

Bits and Pieces

Bytecode Execution

Another approach is *ahead of time* (AOT) compilation

This takes bytecode and further compiles it for the specific OS and hardware at *installation time*

Devised mostly for users (not developers!) of apps for low-energy devices (phones), where the repeated runtime interpretation or JIT compilation every time the app is run is wasted energy

Bits and Pieces

Bytecode Execution

Another approach is *ahead of time* (AOT) compilation

This takes bytecode and further compiles it for the specific OS and hardware at *installation time*

Devised mostly for users (not developers!) of apps for low-energy devices (phones), where the repeated runtime interpretation or JIT compilation every time the app is run is wasted energy

Suitable compilation and optimisation is done just once, when the app is installed: “delivery time compilation”

Bits and Pieces

Bytecode Execution

AOT gives us

Bits and Pieces

Bytecode Execution

AOT gives us

- a faster running app, as there is reduced run-time overhead of interpretation or compilation

Bits and Pieces

Bytecode Execution

AOT gives us

- a faster running app, as there is reduced run-time overhead of interpretation or compilation
- less energy used, as we don’t repeatedly use energy in doing the same compilation every time the app is run

Bits and Pieces

Bytecode Execution

Downsides include

- you lose the run-time information of a JIT that could possibly produce better optimised code. However, this loss appears to be outweighed by the gains from being able to optimise globally the whole app, rather than JIT's local optimisations

Bits and Pieces

Bytecode Execution

Downsides include

- you lose the run-time information of a JIT that could possibly produce better optimised code. However, this loss appears to be outweighed by the gains from being able to optimise globally the whole app, rather than JIT's local optimisations
- installing the app will take a lot longer if a thorough optimising compiler is used. A user would do this just once, though

Bits and Pieces

Bytecode Execution

Downsides include

- you lose the run-time information of a JIT that could possibly produce better optimised code. However, this loss appears to be outweighed by the gains from being able to optimise globally the whole app, rather than JIT's local optimisations
- installing the app will take a lot longer if a thorough optimising compiler is used. A user would do this just once, though
- the compiled code takes up more space. Becoming less of an issue as memory capacity on small devices improves

Bits and Pieces

Bytecode Execution

You can also use a mixture of AOT and JIT

Bits and Pieces

Bytecode Execution

You can also use a mixture of AOT and JIT

Later version of Android do not use AOT when installing an app

Bits and Pieces

Bytecode Execution

You can also use a mixture of AOT and JIT

Later version of Android do not use AOT when installing an app

When your phone is idle it then sneakily uses AOT while you are not looking

Bits and Pieces

Bytecode Execution

You can also use a mixture of AOT and JIT

Later version of Android do not use AOT when installing an app

When your phone is idle it then sneakily uses AOT while you are not looking

And it also uses JIT to tune apps as they run

Bytecode Execution

You get the advantages of fast installation and AOT and JIT

Bytecode Execution

You get the advantages of fast installation and AOT and JIT

But this makes the Android runtime very complicated!

You get the advantages of fast installation and AOT and JIT

But this makes the Android runtime very complicated!

Exercise What does Apple do?

Bits and Pieces

Compilation

You may wish to think about how compilation affects optimisation of your code

Bits and Pieces

Compilation

You may wish to think about how compilation affects optimisation of your code

“Normal” Compilation

A compiler is given a module/file at a time and compiles it, usually with some type information about the external functions called (e.g., `#include`, or `use` or equivalent)

You may wish to think about how compilation affects optimisation of your code

"Normal" Compilation

A compiler is given a module/file at a time and compiles it, usually with some type information about the external functions called (e.g., `#include`, or `use` or equivalent)

So if the code includes a call `k(x+1,y/2)`, where `k` is defined in another module, the compiler generally only has the type signature `int k(int a, int b)` so it knows enough to generate the correct code to pass the arguments to the function and get the return value

Bits and Pieces

Compilation

The code for `k` could be in a separate module, compiled at another time or place, so the compiler has no more information than the signature, and can make no assumptions on `k`

Bits and Pieces

Compilation

The code for `k` could be in a separate module, compiled at another time or place, so the compiler has no more information than the signature, and can make no assumptions on `k`

E.g., if it knew that `b` happened to be unused in `k`, it could optimise away the `y` and the division

Bits and Pieces

Compilation

The code for `k` could be in a separate module, compiled at another time or place, so the compiler has no more information than the signature, and can make no assumptions on `k`

E.g., if it knew that `b` happened to be unused in `k`, it could optimise away the `y` and the division

But without knowing more about `k`, it can't do anything clever like that

Total Compilation

Total Compilation

This is currently quite rare in practice, usually only for small programs

Total Compilation

This is currently quite rare in practice, usually only for small programs

The compiler is given the *whole* program code at once

Total Compilation

This is currently quite rare in practice, usually only for small programs

The compiler is given the *whole* program code at once

It can now look at every detail of every function and make optimisations such as the one above

Total Compilation

This is currently quite rare in practice, usually only for small programs

The compiler is given the *whole* program code at once

It can now look at every detail of every function and make optimisations such as the one above

Practically, this is clearly quite difficult for larger programs

Link Time Optimisation (LTO)

Link Time Optimisation (LTO)

With this, modules are compiled separately as normal, but in the link phase, when all the compiled parts are joined together, the linker can then make some optimisations

Link Time Optimisation (LTO)

With this, modules are compiled separately as normal, but in the link phase, when all the compiled parts are joined together, the linker can then make some optimisations

Again, technically difficult, but starting to be well supported by some languages and can make a big difference

Link Time Optimisation (LTO)

With this, modules are compiled separately as normal, but in the link phase, when all the compiled parts are joined together, the linker can then make some optimisations

Again, technically difficult, but starting to be well supported by some languages and can make a big difference

Note the *linker* could be doing some (re)compilation here!

Bits and Pieces

Compilation

Run Time Optimisation

The runtime system monitors the program as it is running, and make dynamic optimisations to the code using knowledge of what is actually happening in the code

Run Time Optimisation

The runtime system monitors the program as it is running, and make dynamic optimisations to the code using knowledge of what is actually happening in the code

This might involve moving bits of code or data around based on how often they are needed, to reduce memory pressure

Run Time Optimisation

The runtime system monitors the program as it is running, and make dynamic optimisations to the code using knowledge of what is actually happening in the code

This might involve moving bits of code or data around based on how often they are needed, to reduce memory pressure

Used to good effect in JIT compilers

Bits and Pieces

Compilation

In summary: running a program can be a very complex operation!

Bits and Pieces

Compilation

In summary: running a program can be a very complex operation!

Exercise Compare simple optimising-to-native compilers, e.g., C, with complex JIT runtimes, e.g., Java. Think about program speed, data size, complexity of supporting infrastructure, and so on

Classifications

There are a large number of ways we can look at languages, their features, their abilities

Classifications

There are a large number of ways we can look at languages, their features, their abilities

We have just touched on a few topics: there are many more things we could talk about

Classifications

There are a large number of ways we can look at languages, their features, their abilities

We have just touched on a few topics: there are many more things we could talk about

Exercise For example, read about *tail call optimisation*, *continuations*, *coroutines* and *generators*, all of which deal with manipulating the flow of control in a program

Classifications

Of course, it is important to know that these classifications exist so we can make informed choices amongst them

Classifications

Of course, it is important to know that these classifications exist so we can make informed choices amongst them

The right tool for the job