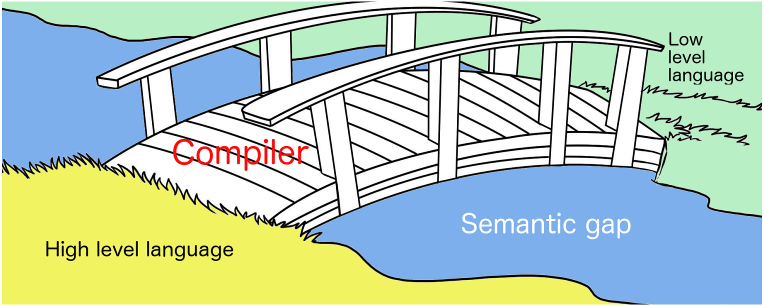
Compilers and computer architecture: Just-in-time compilation

Martin Berger [1]

December 2019

[1] Email: `M.F.Berger@sussex.ac.uk`, Office hours: Wed 12-13 in Chi-2R312

Recall the function of compilers



Welcome to the cutting edge

Welcome to the cutting edge

Compilers are used to translate from programming languages humans can understand to machine code executable by computers. Compilers come in two forms:

- Conventional **ahead-of-time** compilers where translation is done once, long before program execution.
- **Just-in-time** (JIT) compilers where translation of program fragments happens at the last possible moment and is interleaved with program execution.

Welcome to the cutting edge

Compilers are used to translate from programming languages humans can understand to machine code executable by computers. Compilers come in two forms:

- Conventional **ahead-of-time** compilers where translation is done once, long before program execution.
- **Just-in-time** (JIT) compilers where translation of program fragments happens at the last possible moment and is interleaved with program execution.

We spend the whole term learning about the former. Today I want to give you a **brief** introduction to the latter.

Why learn about JIT compilers?

Why learn about JIT compilers?

In the past, dynamically typed languages (e.g. Python, Javascript) were **much more slow** than statically typed languages (factor of 10 or worse). Even OO languages (e.g. Java) were a lot slower than procedural languages like C.

Why learn about JIT compilers?

In the past, dynamically typed languages (e.g. Python, Javascript) were **much more slow** than statically typed languages (factor of 10 or worse). Even OO languages (e.g. Java) were a lot slower than procedural languages like C.

In the last couple of years, this gap has been narrowed considerably. JIT compilers where the main cause of this performance revolution.

Why learn about JIT compilers?

In the past, dynamically typed languages (e.g. Python, Javascript) were **much more slow** than statically typed languages (factor of 10 or worse). Even OO languages (e.g. Java) were a lot slower than procedural languages like C.

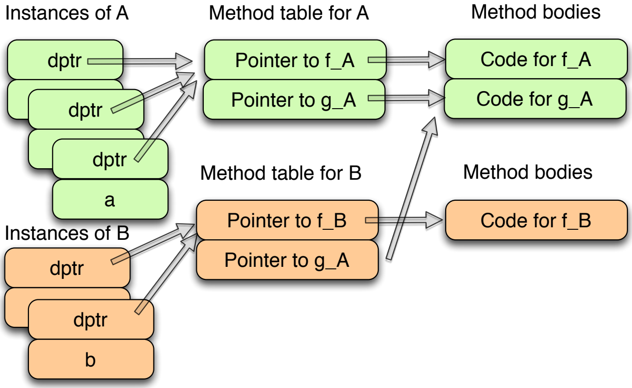
In the last couple of years, this gap has been narrowed considerably. JIT compilers where the main cause of this performance revolution.

JIT compilers are cutting (bleeding) edge technology and considerably more complex than normal compilers, which are already non-trivial. Hence the presentation today will be massively simplifying.

If JIT compilers are the answer ... what is the problem?

If JIT compilers are the answer ... what is the problem?

Let's look at two examples. Remember the compilation of objects and classes?



To deal with inheritance of methods, invoking a method is indirect via the method table. Each invocation has to follow two pointers. Without inheritance, no need for indirection.

If JIT compilers are the answer ... what is the problem?

Of course an individual indirection takes < 1 nano-second on a modern CPU. So why worry?

If JIT compilers are the answer ... what is the problem?

Of course an individual indirection takes < 1 nano-second on a modern CPU. So why worry? Answer: loops!

```
interface I {
  int f ( int n ); }

class A implements I {
  public int f ( int n ) { return n; } }

class B implements I {
  public int f ( int n ) { return 2*n; } }

class Main {
  public static void main ( String [] args ) {
  I o = new A ();
  for ( int i = 0; i < 1000000; i++ ) {
    for ( int j = 0; i < 1000000; j++ ) {
      o.f ( i+j ); } } } }
```

Performance penalties add up.

If JIT compilers are the answer ... what is the problem?

If JIT compilers are the answer ... what is the problem?

But, I hear you say, it's obvious, even at compile time, that the object `o` is of class `A`. A good optimising compiler should be able to work this out, and replace the indirect invocation of `f` with a cheaper direct jump.

```
class Main {
  public static void main ( String [] args ) {
  I o = new A ();
  for ( int i = 0; i < 1000000; i++ ) {
    for ( int j = 0; i < 1000000; j++ ) {
      o.f ( i+j ); } } } }
```

If JIT compilers are the answer ... what is the problem?

But, I hear you say, it's obvious, even at compile time, that the object `o` is of class `A`. A good optimising compiler should be able to work this out, and replace the indirect invocation of `f` with a cheaper direct jump.

```
class Main {
  public static void main ( String [] args ) {
  I o = new A ();
  for ( int i = 0; i < 1000000; i++ ) {
    for ( int j = 0; i < 1000000; j++ ) {
      o.f ( i+j ); } } } }
```

Yes, in this simple example, a good optimising compiler can do this. But what about the following?

If JIT compilers are the answer ... what is the problem?

```
public static void main ( String [] args ) {
  I o = null;
  if ( args [ 0 ] == "hello" )
   new A ();
 else
   new B ();
 for ( int i = 0; i < 1000000; i++ ) {
   for ( int j = 0; i < 1000000; j++ ) {
     o.f ( i+j ); } } } }
```

If JIT compilers are the answer ... what is the problem?

```
public static void main ( String [] args ) {
  I o = null;
  if ( args [ 0 ] == "hello" )
   new A ();
 else
   new B ();
 for ( int i = 0; i < 1000000; i++ ) {
   for ( int j = 0; i < 1000000; j++ ) {
     o.f ( i+j ); } } } }
```

Now the type of `o` is determined only at run-time.

If JIT compilers are the answer ... what is the problem?

```
public static void main ( String [] args ) {
  I o = null;
  if ( args [ 0 ] == "hello" )
   new A ();
 else
   new B ();
 for ( int i = 0; i < 1000000; i++ ) {
   for ( int j = 0; i < 1000000; j++ ) {
     o.f ( i+j ); } } } }
```

Now the type of `o` is determined only at run-time. What is the problem?

If JIT compilers are the answer ... what is the problem?

```
public static void main ( String [] args ) {
  I o = null;
  if ( args [ 0 ] == "hello" )
   new A ();
 else
   new B ();
 for ( int i = 0; i < 1000000; i++ ) {
   for ( int j = 0; i < 1000000; j++ ) {
     o.f ( i+j ); } } } }
```

Now the type of `o` is determined only at run-time. What is the problem? **Not enough information at compile-time to carry out optimisation!**

If JIT compilers are the answer ... what is the problem?

```
public static void main ( String [] args ) {
  I o = null;
  if ( args [ 0 ] == "hello" )
   new A ();
 else
   new B ();
 for ( int i = 0; i < 1000000; i++ ) {
   for ( int j = 0; i < 1000000; j++ ) {
     o.f ( i+j ); } } } }
```

Now the type of `o` is determined only at run-time. What is the problem? **Not enough information at compile-time to carry out optimisation!** At run-time we do have this information, but that's too late (for normal compilers).

If JIT compilers are the answer ... what is the problem?

```
public static void main ( String [] args ) {
   I o = null;
   if ( args [ 0 ] == "hello" )
    new A ();
  else
    new B ();
  for ( int i = 0; i < 1000000; i++ ) {
    for ( int j = 0; i < 1000000; j++ ) {
      o.f ( i+j ); } } } }
```

Now the type of `o` is determined only at run-time. What is the problem? **Not enough information at compile-time to carry out optimisation!** At run-time we do have this information, but that's too late (for normal compilers).

(Aside, can you see a hack to deal with this problem in an AOT compiler?)

If JIT compilers are the answer ... what is the problem?

If JIT compilers are the answer ... what is the problem?

Dynamically typed languages have a worse problem.

If JIT compilers are the answer ... what is the problem?

Dynamically typed languages have a worse problem.

Simplifying a little, variables in dynamically typed languages store not just the usual value, e.g. `3`, but also the type of the value, e.g. `Int`, and sometimes even more.

If JIT compilers are the answer ... what is the problem?

Dynamically typed languages have a worse problem.

Simplifying a little, variables in dynamically typed languages store not just the usual value, e.g. `3`, but also the type of the value, e.g. `Int`, and sometimes even more. Whenever you carry an innocent operation like

```
x = x + y
```

under the hood something like the following happens.

```
let tx = typeof ( x )
let ty = typeof ( y )
if ( tx == Int && ty == Int )
   let vx = value ( x )
   let vy = value ( y )
   let res = integer_addition ( vx, vy )
   x_result_part = res
   x_type_part = Int
else
   ... // even more complicated.
```

If JIT compilers are the answer ... what is the problem?

Imagine this in a nested loop!

```
for ( int i = 0; i < 1000000; i++ ) {
   for ( int j = 0; i < 1000000; j++ ) {
      let tx = typeof ( x )
      let ty = typeof ( y )
      if ( tx == Int && ty == Int )
         let vx = value ( x )
         let vy = value ( y )
         let res = integer_addition ( vx, vy )
         x_result_part = res
         x_type_part = Int
      ...
```

If JIT compilers are the answer ... what is the problem?

Imagine this in a nested loop!

```
for ( int i = 0; i < 1000000; i++ ) {
   for ( int j = 0; i < 1000000; j++ ) {
      let tx = typeof ( x )
      let ty = typeof ( y )
      if ( tx == Int && ty == Int )
         let vx = value ( x )
         let vy = value ( y )
         let res = integer_addition ( vx, vy )
         x_result_part = res
         x_type_part = Int
      ...
```

This is painful. This is why dynamically typed languages are slow(er).

If JIT compilers are the answer ... what is the problem?

But ...

If JIT compilers are the answer ... what is the problem?

But ... in practise, variables **usually** do not change their types in inner loops.

If JIT compilers are the answer ... what is the problem?

But ... in practise, variables **usually** do not change their types in inner loops.

Why?

If JIT compilers are the answer ... what is the problem?

But ... in practise, variables **usually** do not change their types in inner loops.

Why?

Because typically innermost loops work on big and uniform data structures (usually big arrays).

If JIT compilers are the answer ... what is the problem?

But ... in practise, variables **usually** do not change their types in inner loops.

Why?

Because typically innermost loops work on big and uniform data structures (usually big arrays).

So the compiler should move the type-checks outside the loops.

If JIT compilers are the answer ... what is the problem?

Recall that in dynamically typed languages

```
for ( int i = 0; i < 1000000; i++ ) {
  for ( int j = 0; i < 1000000; j++ ) {
    a [i, j] = a[i,j] + 1 } }
```

If JIT compilers are the answer ... what is the problem?

Recall that in dynamically typed languages

```
for ( int i = 0; i < 1000000; i++ ) {
  for ( int j = 0; i < 1000000; j++ ) {
    a [i, j] = a[i,j] + 1 } }
```

Is really

```
for ( int i = 0; i < 1000000; i++ ) {
  for ( int j = 0; i < 1000000; j++ ) {
      let ta = typeof ( a[i, j] ) // always same
      let t1 = typeof ( 1 )       // always same
      if ( ta == Int && t1 == Int ) {
         let va = value ( a[i, j] )
         let v1 = value ( 1 ) // simplifying
         let res = integer_addition ( va, v1 )
         a[ i, j ]_result_part = res
         a[ i, j ] _type_part = Int }
      else { ... } } }
```

If JIT compilers are the answer ... what is the problem?

So program from last slide can be

```
let ta = typeof ( a )
let t1 = typeof ( 1 )
if ( ta == Array [...] of Int && t1 == Int ) {
for ( int i = 0; i < 1000000; i++ ) {
  for ( int j = 0; i < 1000000; j++ ) {
         let va = value ( a[i, j] )
         let v1 = value ( 1 ) // simplifying
         let res = integer_addition ( va, v1 )
         a[ i, j ]_result_part = res } } }
else { ... }
```

If JIT compilers are the answer ... what is the problem?

So program from last slide can be

```
let ta = typeof ( a )
let t1 = typeof ( 1 )
if ( ta == Array [...] of Int && t1 == Int ) {
for ( int i = 0; i < 1000000; i++ ) {
  for ( int j = 0; i < 1000000; j++ ) {
         let va = value ( a[i, j] )
         let v1 = value ( 1 ) // simplifying
         let res = integer_addition ( va, v1 )
         a[ i, j ]_result_part = res } } }
else { ... }
```

Alas, at compile-time, the compiler does not have enough information to make this optimisation safely.

If JIT compilers are the answer ... what is the problem?

If JIT compilers are the answer ... what is the problem?

Let's summarise the situation.

- Certain powerful optimisations cannot be done at compile-time, because the compiler has not got enough information to know they are safe.
- At run-time we have enough information to carry out these optimisations.

If JIT compilers are the answer ... what is the problem?

Let's summarise the situation.

- Certain powerful optimisations cannot be done at compile-time, because the compiler has not got enough information to know they are safe.
- At run-time we have enough information to carry out these optimisations.

Hmmm, what could we do ...

How about we compile and optimise only at run-time?

How about we compile and optimise only at run-time?

But there is no run-time if we don't have a compilation process, right?

How about we compile and optimise only at run-time?

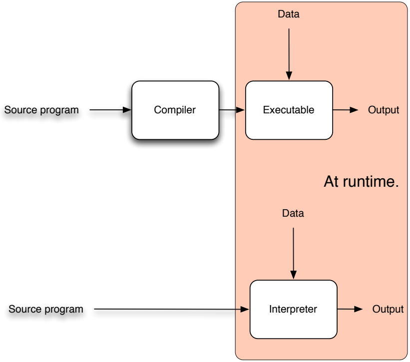
But there is no run-time if we don't have a compilation process, right?

Enter **interpreters**!

Interpreters

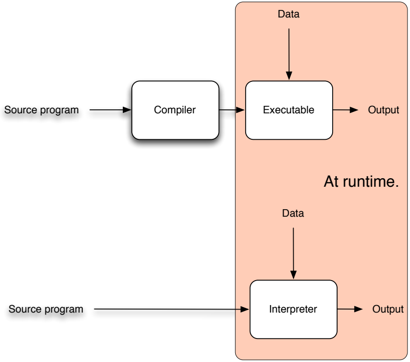
Interpreters

Recall from the beginning of the course, that interpreters are a second way to run programs.



Interpreters

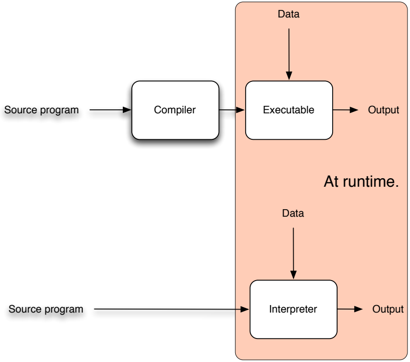
Recall from the beginning of the course, that interpreters are a second way to run programs.



- Compilers generate a program that has an effect on the world.

Interpreters

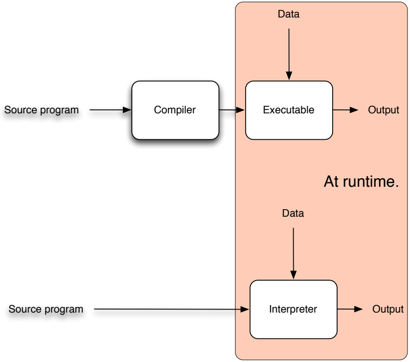
Recall from the beginning of the course, that interpreters are a second way to run programs.



- Compilers generate a program that has an effect on the world.
- Interpreters effect the world directly.

Interpreters

Recall from the beginning of the course, that interpreters are a second way to run programs.



- The advantage of compilers is that generated code is **faster**, because a lot of work has to be done only once (e.g. lexing, parsing, type-checking, optimisation). And the results of this work are shared in every execution. The interpreter has to redo this work every time.
- The advantage of interpreters is that they are much **simpler** than compilers.

JIT compiler, key idea

JIT compiler, key idea

Interpret the program, and compile (parts of) the program at run-time.

JIT compiler, key idea

Interpret the program, and compile (parts of) the program at run-time. This suggests the following questions.

JIT compiler, key idea

Interpret the program, and compile (parts of) the program at run-time. This suggests the following questions.

- When shall we compile, and which parts of the program?

JIT compiler, key idea

Interpret the program, and compile (parts of) the program at run-time. This suggests the following questions.

- When shall we compile, and which parts of the program?
- How do interpreter and compiled program interact?

JIT compiler, key idea

Interpret the program, and compile (parts of) the program at run-time. This suggests the following questions.

- When shall we compile, and which parts of the program?
- How do interpreter and compiled program interact?
- But most of all: compilation is **really slow**, especially optimising compilation. Don't we make performance worse if we slow an already slow interpreter down with a lengthy compilation process?

JIT compiler, key idea

Interpret the program, and compile (parts of) the program at run-time. This suggests the following questions.

- When shall we compile, and which parts of the program?
- How do interpreter and compiled program interact?
- But most of all: compilation is **really slow**, especially optimising compilation. Don't we make performance worse if we slow an already slow interpreter down with a lengthy compilation process?

In other words, we are facing the following conundrum:

JIT compiler, key idea

Interpret the program, and compile (parts of) the program at run-time. This suggests the following questions.

- When shall we compile, and which parts of the program?
- How do interpreter and compiled program interact?
- But most of all: compilation is **really slow**, especially optimising compilation. Don't we make performance worse if we slow an already slow interpreter down with a lengthy compilation process?

In other words, we are facing the following conundrum:

- We want to optimise as much as possible, because optimised programs run faster.

JIT compiler, key idea

Interpret the program, and compile (parts of) the program at run-time. This suggests the following questions.

- When shall we compile, and which parts of the program?
- How do interpreter and compiled program interact?
- But most of all: compilation is **really slow**, especially optimising compilation. Don't we make performance worse if we slow an already slow interpreter down with a lengthy compilation process?

In other words, we are facing the following conundrum:

- We want to optimise as much as possible, because optimised programs run faster.
- We want to optimises as little as possible, because running the optimisers is really slow.

JIT compiler, key idea

Interpret the program, and compile (parts of) the program at run-time. This suggests the following questions.

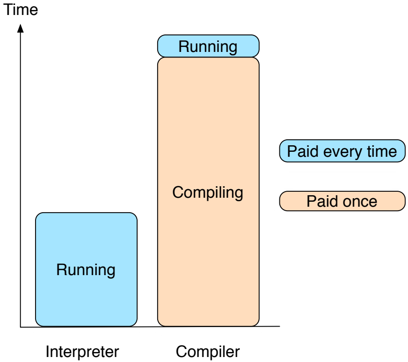
- When shall we compile, and which parts of the program?
- How do interpreter and compiled program interact?
- But most of all: compilation is **really slow**, especially optimising compilation. Don't we make performance worse if we slow an already slow interpreter down with a lengthy compilation process?

In other words, we are facing the following conundrum:

- We want to optimise as much as possible, because optimised programs run faster.
- We want to optimises as little as possible, because running the optimisers is really slow.

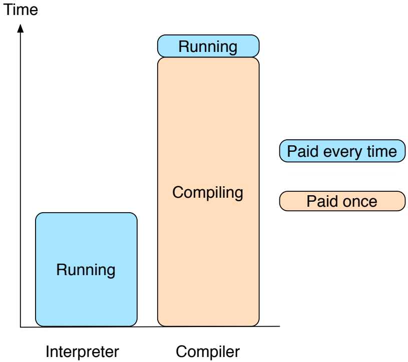
Hmmmm ...

Pareto principle and compiler/interpreter ∆ to our rescue



Interpretation is much faster than (optimising) compilation. But a compiled program is much faster than interpretation. And we have to compile only once.

Pareto principle and compiler/interpreter Δ to our rescue



Interpretation is much faster than (optimising) compilation. But a compiled program is much faster than interpretation. And we have to compile only once.

Combine this with the Pareto principle, and you have a potent weapon at hand.

Pareto principle, aka 80-20 rule

Pareto principle, aka 80-20 rule

Vilfredo Pareto, late 19th, early 20th century Italian economist. Noticed:

- 80% of land in Italy was owned by 20% of the population.
- 20% of the pea pods in his garden contained 80% of the peas.

Pareto principle, aka 80-20 rule

Vilfredo Pareto, late 19th, early 20th century Italian economist. Noticed:

- 80% of land in Italy was owned by 20% of the population.
- 20% of the pea pods in his garden contained 80% of the peas.

This principle applies in many other areas of life, including program execution:

Pareto principle, aka 80-20 rule

Vilfredo Pareto, late 19th, early 20th century Italian economist. Noticed:

- 80% of land in Italy was owned by 20% of the population.
- 20% of the pea pods in his garden contained 80% of the peas.

This principle applies in many other areas of life, including program execution:

The great majority of a program's execution time is spent running in a tiny fragment of the code.

Pareto principle, aka 80-20 rule

Vilfredo Pareto, late 19th, early 20th century Italian economist. Noticed:

- 80% of land in Italy was owned by 20% of the population.
- 20% of the pea pods in his garden contained 80% of the peas.

This principle applies in many other areas of life, including program execution:

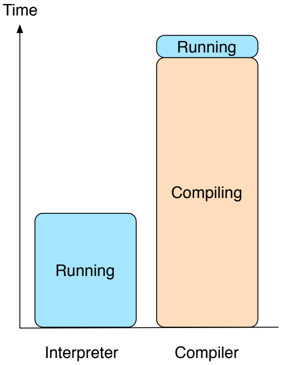
The great majority of a program's execution time is spent running in a tiny fragment of the code.

Such code is referred to as **hot**.

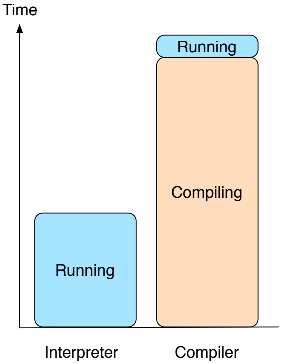
Putting the pieces together

Putting the pieces together

Clearly compiling at run-time code that's executed infrequently will slow down execution. Trade-offs are different for hot code.



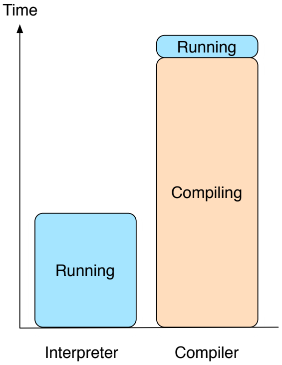
Putting the pieces together



Clearly compiling at run-time code that's executed infrequently will slow down execution. Trade-offs are different for hot code.

An innermost loop may be executed billions of times. The more often, the more optimising compilation pays off.

Putting the pieces together

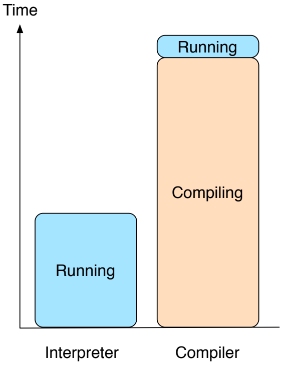


Clearly compiling at run-time code that's executed infrequently will slow down execution. Trade-offs are different for hot code.

An innermost loop may be executed billions of times. The more often, the more optimising compilation pays off.

Pareto's principle tells us that (typically) a program contains some hot code.

Putting the pieces together



Clearly compiling at run-time code that's executed infrequently will slow down execution. Trade-offs are different for hot code.

An innermost loop may be executed billions of times. The more often, the more optimising compilation pays off.

Pareto's principle tells us that (typically) a program contains some hot code.

With the information available at run-time, we can aggressively optimise such hot code, and get a massive speed-up. The rest is interpreted. Sluggishness of interpretation doesn't matter, because it's only a fraction of program execution time.

There is just one problem ... how do we find hot code?

There is just one problem ... how do we find hot code?

Remember, at compiler time, the optimiser couldn't work it out (reliably).

There is just one problem ... how do we find hot code?

Remember, at compiler time, the optimiser couldn't work it out (reliably).

There is just one problem ... how do we find hot code?

Remember, at compiler time, the optimiser couldn't work it out (reliably).

Let's use counters at run-time!

There is just one problem ... how do we find hot code?

Remember, at compiler time, the optimiser couldn't work it out (reliably).

Let's use counters at run-time!

We instrument the interpreter with counters, that increment every time a method is called, or every time we go round a loop.

There is just one problem ... how do we find hot code?

Remember, at compiler time, the optimiser couldn't work it out (reliably).

Let's use counters at run-time!

We instrument the interpreter with counters, that increment every time a method is called, or every time we go round a loop.

Whenever these counters reach a threshold, we assume that the associated code is hot. We compile that hot code, and jump to the compiled code.

There is just one problem ... how do we find hot code?

Remember, at compiler time, the optimiser couldn't work it out (reliably).

Let's use counters at run-time!

We instrument the interpreter with counters, that increment every time a method is called, or every time we go round a loop.

Whenever these counters reach a threshold, we assume that the associated code is hot. We compile that hot code, and jump to the compiled code.

Making this play nice with garbage collection, exceptions, concurrency, debugging isn't easy ...

There is just one problem ... how do we find hot code?

Remember, at compiler time, the optimiser couldn't work it out (reliably).

Let's use counters at run-time!

We instrument the interpreter with counters, that increment every time a method is called, or every time we go round a loop.

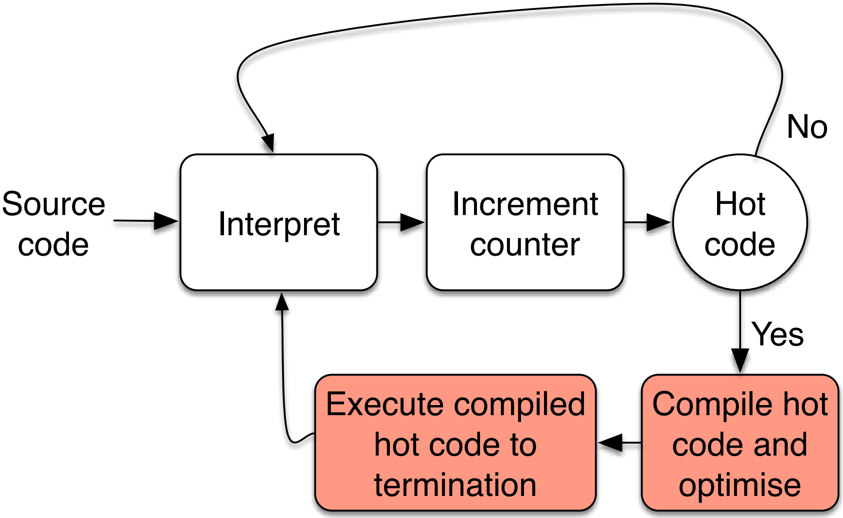
Whenever these counters reach a threshold, we assume that the associated code is hot. We compile that hot code, and jump to the compiled code.

Making this play nice with garbage collection, exceptions, concurrency, debugging isn't easy ...

When the compiled code terminates, we switch back to interpretation.

In a picture

In a picture


Source code
Interpret
Increment counter
Hot code
No
Yes
Compile hot code and optimise
Execute compiled hot code to termination

Aside

Aside

Have you noticed that Java programs start up quite slowly?

Aside

Have you noticed that Java programs start up quite slowly?

This is because at the beginning, everything is interpreted, hence slow. Then JIT compilation starts, also slow.

Aside

Have you noticed that Java programs start up quite slowly?

This is because at the beginning, everything is interpreted, hence slow. Then JIT compilation starts, also slow.

Eventually, the hot code is detected and compiled with a great deal of optimisation. Then execution gets really fast.

The devil is in the details

The devil is in the details

This picture omits many subtleties.

The devil is in the details

This picture omits many subtleties.

Chief among those is that the handover of control from interpreter to compiler and back works seamlessly.

The devil is in the details

This picture omits many subtleties.

Chief among those is that the handover of control from interpreter to compiler and back works seamlessly.

Also, we don't want to recompile code, typically use cache of already compiled code.

The devil is in the details

This picture omits many subtleties.

Chief among those is that the handover of control from interpreter to compiler and back works seamlessly.

Also, we don't want to recompile code, typically use cache of already compiled code.

How actually to do the optimisations, taking information available at run-time into account.

The devil is in the details

This picture omits many subtleties.

Chief among those is that the handover of control from interpreter to compiler and back works seamlessly.

Also, we don't want to recompile code, typically use cache of already compiled code.

How actually to do the optimisations, taking information available at run-time into account.

Etc etc.

JIT compilers summary

JIT compilers summary

JIT compilers are the cutting edge of compiler technology. They were first conceived (in rudimentary form) in the 1960s, but came to life in the last 10 years or so.

JIT compilers summary

JIT compilers are the cutting edge of compiler technology. They were first conceived (in rudimentary form) in the 1960s, but came to life in the last 10 years or so.

JIT compilers are very complicated. The JVM, probably the best known JIT compiler, probably took 1000+ person years to build.

JIT compilers summary

JIT compilers are the cutting edge of compiler technology. They were first conceived (in rudimentary form) in the 1960s, but came to life in the last 10 years or so.

JIT compilers are very complicated. The JVM, probably the best known JIT compiler, probably took 1000+ person years to build.

So what's next in compiler technology?

JIT compilers summary

JIT compilers are the cutting edge of compiler technology. They were first conceived (in rudimentary form) in the 1960s, but came to life in the last 10 years or so.

JIT compilers are very complicated. The JVM, probably the best known JIT compiler, probably took 1000+ person years to build.

So what's next in compiler technology? Let me introduce you to ...

Tracing JIT compilers

Tracing JIT compilers

Tracing JIT compilers are a form of JIT compilation where optimisation is especially aggressive.

Tracing JIT compilers

Tracing JIT compilers

Hot code can contain code that is not used (much).

Tracing JIT compilers

Hot code can contain code that is not used (much). Imagine the compilation of:

```
for ( x = 1 to 1000000 )
   for ( y = 1 to 1000000 )
      try
         a[ x ][ y ] = a[ x+1 ][ a [ y-1 ][ y+1 ] ]
      catch ...  // error handling
```

Tracing JIT compilers

Hot code can contain code that is not used (much). Imagine the compilation of:

```
for ( x = 1 to 1000000 )
   for ( y = 1 to 1000000 )
      try
         a[ x ][ y ] = a[ x+1 ][ a [ y-1 ][ y+1 ] ]
      catch ...  // error handling
```

Clearly the `try-catch` block is an innermost loop, so potentially hot code. But if the programmer does a good job, the exception handling will never be triggered. Yet we have all this exception handling code (tends to be large) in the hot loop. This causes all manner of problems, e.g. cache locality is destroyed.

Tracing JIT compilers

Hot code can contain code that is not used (much). Imagine the compilation of:

```
for ( x = 1 to 1000000 )
   for ( y = 1 to 1000000 )
      try
         a[ x ][ y ] = a[ x+1 ][ a [ y-1 ][ y+1 ] ]
      catch ...  // error handling
```

It is difficult to figure out, even at run-time (!) to find such parts.

Tracing JIT compilers

Hot code can contain code that is not used (much). Imagine the compilation of:

```
for ( x = 1 to 1000000 )
   for ( y = 1 to 1000000 )
      try
         a[ x ][ y ] = a[ x+1 ][ a [ y-1 ][ y+1 ] ]
      catch ...  // error handling
```

It is difficult to figure out, even at run-time (!) to find such parts.

Why can't we use counters?

Tracing JIT compilers

Hot code can contain code that is not used (much). Imagine the compilation of:

```
for ( x = 1 to 1000000 )
   for ( y = 1 to 1000000 )
      try
         a[ x ][ y ] = a[ x+1 ][ a [ y-1 ][ y+1 ] ]
      catch ...  // error handling
```

It is difficult to figure out, even at run-time (!) to find such parts.

Why can't we use counters? Yes but ... counters only give us some relevant information ... for good optimisation we need more information. **Traces** give us this information. What are traces?

Tracing JIT compilers

Tracing JIT compilers

Tracing JIT compilers have not one, but several compilers (or interpreters) inside (simplifying greatly).

Tracing JIT compilers

Tracing JIT compilers have not one, but several compilers (or interpreters) inside (simplifying greatly).

After the interpreter has found hot code, the hot code is compiled and executed once (called tracing execution).

Tracing JIT compilers

Tracing JIT compilers have not one, but several compilers (or interpreters) inside (simplifying greatly).

After the interpreter has found hot code, the hot code is compiled and executed once (called tracing execution).

In the tracing execution, the machine code actually executed is **recorded**, yielding the trace of the hot code.

Tracing JIT compilers

Tracing JIT compilers have not one, but several compilers (or interpreters) inside (simplifying greatly).

After the interpreter has found hot code, the hot code is compiled and executed once (called tracing execution).

In the tracing execution, the machine code actually executed is **recorded**, yielding the trace of the hot code.

Note that if the machine code to be traced is branching, only the branch taken is in the trace. **Traces are linear, no branching**. This makes optimisation algorithms much simpler and faster.

Tracing JIT compilers

Tracing JIT compilers have not one, but several compilers (or interpreters) inside (simplifying greatly).

After the interpreter has found hot code, the hot code is compiled and executed once (called tracing execution).

In the tracing execution, the machine code actually executed is **recorded**, yielding the trace of the hot code.

Note that if the machine code to be traced is branching, only the branch taken is in the trace. **Traces are linear, no branching**. This makes optimisation algorithms much simpler and faster.

Once tracing has finished, e.g. the body of the hot loop has been executed once: then analyse and optimise trace.

Tracing JIT compilers

Tracing JIT compilers have not one, but several compilers (or interpreters) inside (simplifying greatly).

After the interpreter has found hot code, the hot code is compiled and executed once (called tracing execution).

In the tracing execution, the machine code actually executed is **recorded**, yielding the trace of the hot code.

Note that if the machine code to be traced is branching, only the branch taken is in the trace. **Traces are linear, no branching**. This makes optimisation algorithms much simpler and faster.

Once tracing has finished, e.g. the body of the hot loop has been executed once: then analyse and optimise trace.

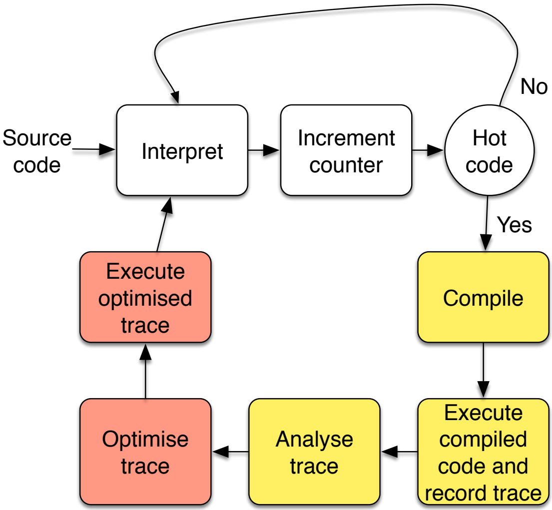
Based on the analysis another compiler generates another (highly optimised) executable, which is then run to termination, then control goes back to interpreter.

Tracing JIT compilers

Analysing and optimising the trace:

- Find out if variables change type in the loop, if not, move type-checking out of the loop. (For dynamically typed languages.)
- Find out if object change type in the loop, if not, use short-cut method invocations, no need to go via method table.
- Let the interpreter handle the rarely used parts of the hot loop (e.g. error handling).
- ...
- Finally, enter the third phase, the ’normal’ execution of the optimised trace.

A tracing JIT compiler in a picture



Difficulties

As with normal JIT compilers, we have to orchestrate the interplay of all these compiler phases, e.g.: Handover of control from interpreter to compiler, to tracing, to execution of optimised trace, and back. Garbage collection, exceptions, concurrency etc must all also work.

Difficulties

As with normal JIT compilers, we have to orchestrate the interplay of all these compiler phases, e.g.: Handover of control from interpreter to compiler, to tracing, to execution of optimised trace, and back. Garbage collection, exceptions, concurrency etc must all also work.

Typical optimisations: type-specialisation, bypassing method invocation, function inlining, register allocation, dead code elimination.

Difficulties

As with normal JIT compilers, we have to orchestrate the interplay of all these compiler phases, e.g.: Handover of control from interpreter to compiler, to tracing, to execution of optimised trace, and back. Garbage collection, exceptions, concurrency etc must all also work.

Typical optimisations: type-specialisation, bypassing method invocation, function inlining, register allocation, dead code elimination.

Etc etc.

Example compilers

Example compilers

The **JVM** (from Oracle). It is a method based JIT compiler, meaning that methods are the units of compilation. It is not tracing.

Example compilers

The **JVM** (from Oracle). It is a method based JIT compiler, meaning that methods are the units of compilation. It is not tracing.

The first implementation of a tracing JIT was HPs **Dynamo**. It does not compile from a high-level language to a low-level language. Instead it optimises machine-code.

Example compilers

The **JVM** (from Oracle). It is a method based JIT compiler, meaning that methods are the units of compilation. It is not tracing.

The first implementation of a tracing JIT was HPs **Dynamo**. It does not compile from a high-level language to a low-level language. Instead it optimises machine-code.

HotpathVM was the first tracing JIT for a high-level language (Java).

Example compilers

The **JVM** (from Oracle). It is a method based JIT compiler, meaning that methods are the units of compilation. It is not tracing.

The first implementation of a tracing JIT was HPs **Dynamo**. It does not compile from a high-level language to a low-level language. Instead it optimises machine-code.

HotpathVM was the first tracing JIT for a high-level language (Java).

TraceMonkey, one of Firefox's JavaScript implementations was first JIT compiler for Javascript. (NB: Current Firefox's SpiderMonkey is not tracing.)

Example compilers

The **JVM** (from Oracle). It is a method based JIT compiler, meaning that methods are the units of compilation. It is not tracing.

The first implementation of a tracing JIT was HPs **Dynamo**. It does not compile from a high-level language to a low-level language. Instead it optimises machine-code.

HotpathVM was the first tracing JIT for a high-level language (Java).

TraceMonkey, one of Firefox's JavaScript implementations was first JIT compiler for Javascript. (NB: Current Firefox's SpiderMonkey is not tracing.)

Hard to say exactly who uses what (e.g. Apple Safari) since companies rarely say what they're using. They can use more than one. Trade secrets.

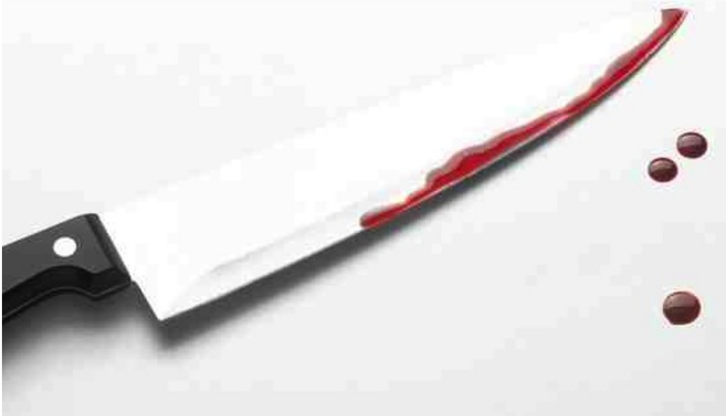
Example compilers

Open source: **PyPy**, a **meta-tracing** framework for Python.

Example compilers

Open source: **PyPy**, a **meta-tracing** framework for Python.
Meta-tracing, what's that?

Meta-tracing?



Meta-tracing

Background:

Meta-tracing

Background: Writing compilers is hard, writing optimising compilers is harder, writing JIT compilers is harder still, but writing tracing JIT compilers is the hardest.

Meta-tracing

Background: Writing compilers is hard, writing optimising compilers is harder, writing JIT compilers is harder still, but writing tracing JIT compilers is the hardest.

Designers of new programming languages cannot really produce a good code generator for a new language. Typically language designers write interpreters for new languages. But that means the new language is hampered. This impedes progress in programming languages.

Meta-tracing

Background: Writing compilers is hard, writing optimising compilers is harder, writing JIT compilers is harder still, but writing tracing JIT compilers is the hardest.

Designers of new programming languages cannot really produce a good code generator for a new language. Typically language designers write interpreters for new languages. But that means the new language is hampered. This impedes progress in programming languages.

Great idea: how about using a JIT compiler to compile the interpreter, hoping that JITing will speed up interpreter, hence new PL.

Meta-tracing

Background: Writing compilers is hard, writing optimising compilers is harder, writing JIT compilers is harder still, but writing tracing JIT compilers is the hardest.

Designers of new programming languages cannot really produce a good code generator for a new language. Typically language designers write interpreters for new languages. But that means the new language is hampered. This impedes progress in programming languages.

Great idea: how about using a JIT compiler to compile the interpreter, hoping that JITing will speed up interpreter, hence new PL.

This idea is ingenious, simple, old and ...

Meta-tracing

Background: Writing compilers is hard, writing optimising compilers is harder, writing JIT compilers is harder still, but writing tracing JIT compilers is the hardest.

Designers of new programming languages cannot really produce a good code generator for a new language. Typically language designers write interpreters for new languages. But that means the new language is hampered. This impedes progress in programming languages.

Great idea: how about using a JIT compiler to compile the interpreter, hoping that JITing will speed up interpreter, hence new PL.

This idea is ingenious, simple, old and ... wrong!

Meta-tracing

Background: Writing compilers is hard, writing optimising compilers is harder, writing JIT compilers is harder still, but writing tracing JIT compilers is the hardest.

Designers of new programming languages cannot really produce a good code generator for a new language. Typically language designers write interpreters for new languages. But that means the new language is hampered. This impedes progress in programming languages.

Great idea: how about using a JIT compiler to compile the interpreter, hoping that JITing will speed up interpreter, hence new PL.

This idea is ingenious, simple, old and ... wrong!

The problem is that interpreter loops are the kinds of loops that JITers do not optimise well. Let's explain this in detail.

Why JIT compilers can't optimise interpreter loops

Why JIT compilers can't optimise interpreter loops

An interpreter is a big loop that gets the next command and acts on it, e.g.

Why JIT compilers can't optimise interpreter loops

An interpreter is a big loop that gets the next command and acts on it, e.g.

```
while true do:
   cmd = getNextCommand
   if cmd is:
      "x := E" then ...
      "if C then M else N" then ...
      "while C do M" then ...
      "repeat M until C" then ...
      "print(M)" then ...
      ...
```

Why JIT compilers can't optimise interpreter loops

An interpreter is a big loop that gets the next command and acts on it, e.g.

```
while true do:
    cmd = getNextCommand
    if cmd is:
        "x := E" then ...
        "if C then M else N" then ...
        "while C do M" then ...
        "repeat M until C" then ...
        "print(M)" then ...
        ...
```

Now JIT compilers are really good at optimising loops, why do they fail with interpreter loops?

Key requirements for good JIT optimisation of loops

The essence of JIT compilation are tight inner loops that are executed a large number of times. This insight can be split into separate parts.

Key requirements for good JIT optimisation of loops

The essence of JIT compilation are tight inner loops that are executed a large number of times. This insight can be split into separate parts.

- Because they are executed a large number of times the effect of the optimisation is magnified. ✓

Key requirements for good JIT optimisation of loops

The essence of JIT compilation are tight inner loops that are executed a large number of times. This insight can be split into separate parts.

- Because they are executed a large number of times the effect of the optimisation is magnified. ✓
- Optimising these inner loops heavily gives substantial performace benefits. ✓

Key requirements for good JIT optimisation of loops

The essence of JIT compilation are tight inner loops that are executed a large number of times. This insight can be split into separate parts.

- Because they are executed a large number of times the effect of the optimisation is magnified. ✓
- Optimising these inner loops heavily gives substantial performace benefits. ✓
- Each iteration (or at least most of them) do the same thing.

Key requirements for good JIT optimisation of loops

The essence of JIT compilation are tight inner loops that are executed a large number of times. This insight can be split into separate parts.

- Because they are executed a large number of times the effect of the optimisation is magnified. ✓
- Optimising these inner loops heavily gives substantial performace benefits. ✓
- Each iteration (or at least most of them) do the same thing.

Last requirement is violated in interpreter loops.

Why can’t interpreter loops be JITed?

The problem is that the source language to be interpreted has loops too.

Why can't interpreter loops be JITed?

The problem is that the source language to be interpreted has loops too.

Let's assume this is the programm we are interpreting.

```
while i > 0:
      j = j+i
      i = i-1
```

Why can’t interpreter loops be JITed?

The problem is that the source language to be interpreted has loops too.

Let’s assume this is the programm we are interpreting.

```
while i > 0:
      j = j+i
      i = i-1
```

This gives rise to something like the following bytecode

```
loop:
   br r17 exit
   add r21 r33 r21
   subabs r33 1 r33
   jump loop
exit:
   ...
```

Why can't interpreter loops be JITed?

Let's have bytecode and bytecode interpreter side-by-side:

```
loop:
   br r17 exit
   add r21 r33 r21
   subabs r33 1 r33
   jump loop
exit:
   ...
```

```
while true:
   op = mem [ pc ]
   pc = pc+1
   case op = br:
      r = mem [ pc ]
      pc = pc+1
      if mem [ r ] == 0:
         pc := mem [ pc ]
   case op = add:
      r1 = mem [ pc ]
      pc = pc+1
   ...
```

Why can't interpreter loops be JITed?

Let's have bytecode and bytecode interpreter side-by-side:

```
loop:
   br r17 exit
   add r21 r33 r21
   subabs r33 1 r33
   jump loop
exit:
   ...
```

```
while true:
   op = mem [ pc ]
   pc = pc+1
   case op = br:
      r = mem [ pc ]
      pc = pc+1
      if mem [ r ] == 0:
         pc := mem [ pc ]
   case op = add:
      r1 = mem [ pc ]
      pc = pc+1
   ...
```

Now every round of the interpreter takes a different branch.

Why can't interpreter loops be JITed?

Let's have bytecode and bytecode interpreter side-by-side:

```
loop:
   br r17 exit
   add r21 r33 r21
   subabs r33 1 r33
   jump loop
exit:
   ...
```

```
while true:
   op = mem [ pc ]
   pc = pc+1
   case op = br:
      r = mem [ pc ]
      pc = pc+1
      if mem [ r ] == 0:
         pc := mem [ pc ]
   case op = add:
      r1 = mem [ pc ]
      pc = pc+1
   ...
```

Now every round of the interpreter takes a different branch.
The tracing JIT can just optimise one branch through the loop.

Why can't interpreter loops be JITed?

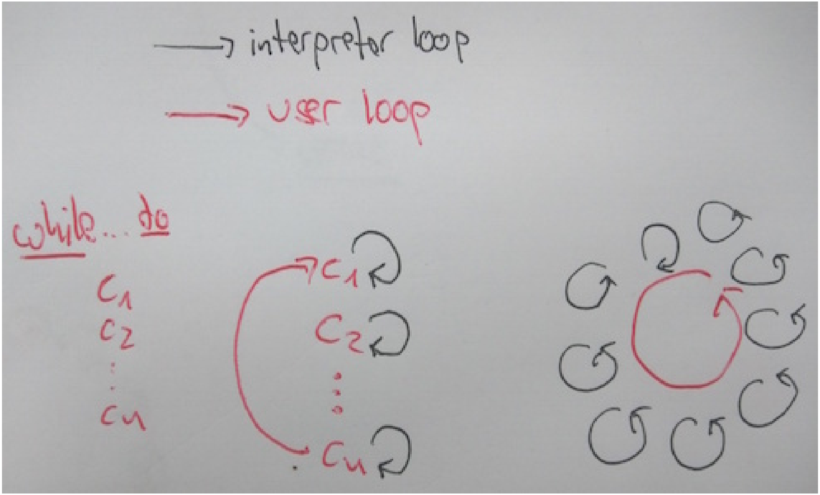
Let's have bytecode and bytecode interpreter side-by-side:

```
loop:
   br r17 exit
   add r21 r33 r21
   subabs r33 1 r33
   jump loop
exit:
   ...
```

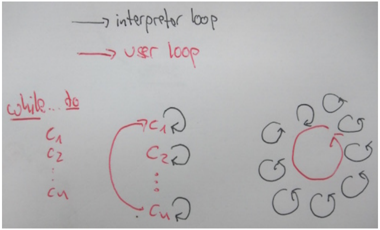
```
while true:
   op = mem [ pc ]
   pc = pc+1
   case op = br:
      r = mem [ pc ]
      pc = pc+1
      if mem [ r ] == 0:
         pc := mem [ pc ]
   case op = add:
      r1 = mem [ pc ]
      pc = pc+1
   ...
```

Now every round of the interpreter takes a different branch.
The tracing JIT can just optimise one branch through the loop.
This is the worst case scenario: we pay the price of tracing, optimisation (since loop is executed a lot), only to throw away the optimisation and go back to interpretation.

Why can't interpreter loops be JITed?

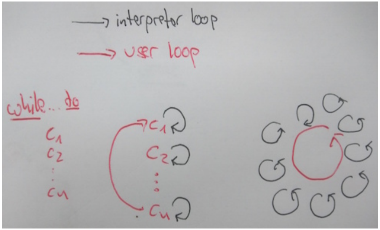


Why can't interpreter loops be JITed?



Profiling detects the **wrong loop** as hot code!

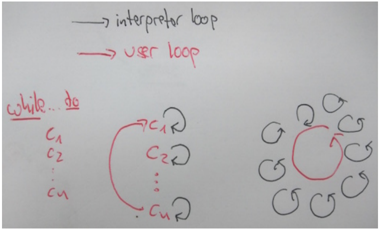
Why can't interpreter loops be JITed?



Profiling detects the **wrong loop** as hot code!

We want profiling to detect the (code corresponding to the) **user loop**, not the interpreter loop.

Why can't interpreter loops be JITed?

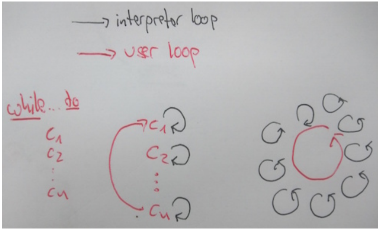


Profiling detects the **wrong loop** as hot code!

We want profiling to detect the (code corresponding to the) **user loop**, not the interpreter loop.

Note that the (code corresponding to the) user loop consists of **several rounds** of the interpreter loop.

Why can't interpreter loops be JITed?



Profiling detects the **wrong loop** as hot code!

We want profiling to detect the (code corresponding to the) **user loop**, not the interpreter loop.

Note that the (code corresponding to the) user loop consists of **several rounds** of the interpreter loop.

This is too difficult to detect for profiling, since user programs can vary greatly.

Why can't interpreter loops be JITed?

The interpreter **writer** knows what the user loops are like:

```
while true do:
    cmd = getNextCommand
    if cmd is:
        "x := E" then ...
        "if C then M else M" then ...
        "while C do M" then ...
        "repeat M until C" then ...
        "print(M)" then ...
        ...
```

Why can't interpreter loops be JITed?

The interpreter **writer** knows what the user loops are like:

```
while true do:
    cmd = getNextCommand
    if cmd is:
        "x := E" then ...
        "if C then M else M" then ...
        "while C do M" then ...
        "repeat M until C" then ...
        "print(M)" then ...
        ...
```

The idea of meta-tracing is to let the interpreter writer **annotate** the interpreter code with 'hooks' that tell the tracing JIT compiler where user loops start and end. The profiler can then identify the hot loops in (the interpretation of) user code.

Why can't interpreter loops be JITed?

```
while true do:
   beginInterpreterLoop
   cmd = getNextCommand
   if cmd is:
      "x := E" then ...
      "while C do M" then
         beginUserLoop
         ...
         endUserLoop
      "repeat M until C" then
         beginUserLoop
         ...
         endUserLoop
      ...
   endInterpreterLoop
```

Annotations are used by profiler for finding hot user loops.
Then user loops are traced & optimised.

Why can't interpreter loops be JITed?

```
while true do:
   beginInterpreterLoop
   cmd = getNextCommand
   if cmd is:
      "x := E" then ...
      "while C do M" then
         beginUserLoop
         ...
         endUserLoop
      "repeat M until C" then
         beginUserLoop
         ...
         endUserLoop
      ...
   endInterpreterLoop
```

Annotations are used by profiler for finding hot user loops.
Then user loops are traced & optimised. **Result: Speedup!**

Why can't interpreter loops be JITed?

```
while true do:
    beginInterpreterLoop
    cmd = getNextCommand
    if cmd is:
        "x := E" then ...
        "while C do M" then
            beginUserLoop
            ...
            endUserLoop
        "repeat M until C" then
            beginUserLoop
            ...
            endUserLoop
        ...
    endInterpreterLoop
```

Annotations are used by profiler for finding hot user loops.
Then user loops are traced & optimised. **Result: Speedup!**
It is simple to annotate an interpreter.

Meta-tracing as game changer in PL development

The real advantage of this is that it divides the problem of developing high-performance JIT compilers for a language into several parts, each of which separately is much more mangable:

Meta-tracing as game changer in PL development

The real advantage of this is that it divides the problem of developing high-performance JIT compilers for a language into several parts, each of which separately is much more mangable:

1. Develop a (meta-)tracing JIT compiler. **Hard**, but needs to be **done only once.**

Meta-tracing as game changer in PL development

The real advantage of this is that it divides the problem of developing high-performance JIT compilers for a language into several parts, each of which separately is much more mangable:

1. Develop a (meta-)tracing JIT compiler. **Hard**, but needs to be **done only once**.
2. Develop an interpreter for the given source language. **Easy!**

Meta-tracing as game changer in PL development

The real advantage of this is that it divides the problem of developing high-performance JIT compilers for a language into several parts, each of which separately is much more mangable:

1. Develop a (meta-)tracing JIT compiler. **Hard**, but needs to be **done only once**.
2. Develop an interpreter for the given source language. **Easy!**
3. Add annotations in the interpreter to expose user loops. **Easy!**

Meta-tracing as game changer in PL development

The real advantage of this is that it divides the problem of developing high-performance JIT compilers for a language into several parts, each of which separately is much more mangable:

1. Develop a (meta-)tracing JIT compiler. **Hard**, but needs to be **done only once**.
2. Develop an interpreter for the given source language. **Easy!**
3. Add annotations in the interpreter to expose user loops. **Easy!**
4. Run the interpreter using the tracing JIT from (1). **Easy!**

Meta-tracing as game changer in PL development

The real advantage of this is that it divides the problem of developing high-performance JIT compilers for a language into several parts, each of which separately is much more mangable:

1. Develop a (meta-)tracing JIT compiler. **Hard**, but needs to be **done only once**.
2. Develop an interpreter for the given source language. **Easy!**
3. Add annotations in the interpreter to expose user loops. **Easy!**
4. Run the interpreter using the tracing JIT from (1). **Easy!**

The tracing JIT from (1) can be reused for an unlimited number of language interpreters.

Meta-tracing as game changer in PL development

The real advantage of this is that it divides the problem of developing high-performance JIT compilers for a language into several parts, each of which separately is much more mangable:

1. Develop a (meta-)tracing JIT compiler. **Hard**, but needs to be **done only once**.
2. Develop an interpreter for the given source language. **Easy!**
3. Add annotations in the interpreter to expose user loops. **Easy!**
4. Run the interpreter using the tracing JIT from (1). **Easy!**

The tracing JIT from (1) can be reused for an unlimited number of language interpreters. Once a meta-tracing JIT is available, we can easily develop new languages and have high-performance compilers for them (almost) for free.

Meta-tracing as game changer in PL development

The real advantage of this is that it divides the problem of developing high-performance JIT compilers for a language into several parts, each of which separately is much more mangable:

1. Develop a (meta-)tracing JIT compiler. **Hard**, but needs to be **done only once**.
2. Develop an interpreter for the given source language. **Easy!**
3. Add annotations in the interpreter to expose user loops. **Easy!**
4. Run the interpreter using the tracing JIT from (1). **Easy!**

The tracing JIT from (1) can be reused for an unlimited number of language interpreters. Once a meta-tracing JIT is available, we can easily develop new languages and have high-performance compilers for them (almost) for free.

The PyPy meta-tracing framework runs Python substantially faster than e.g. the CPython framework.

Brief remarks on performance

Brief remarks on performance

JIT compilers are built upon many trade-offs.

Brief remarks on performance

JIT compilers are built upon many trade-offs.

Although JIT compilers can give lightning fast execution on typical programs, their worst-case execution time can be dreadful.

Brief remarks on performance

JIT compilers are built upon many trade-offs.

Although JIT compilers can give lightning fast execution on typical programs, their worst-case execution time can be dreadful.

JIT compilers work best for languages that do a lot of stuff at run-time (e.g. type-checking).

Brief remarks on performance

JIT compilers are built upon many trade-offs.

Although JIT compilers can give lightning fast execution on typical programs, their worst-case execution time can be dreadful.

JIT compilers work best for languages that do a lot of stuff at run-time (e.g. type-checking).

For bare-bones languages like C, there is little to optimise at run-time, and code generated by a conventional C compiler with heavy (hence slow) optimisation will almost always beat a modern JIT compiler.

Compiler development in industry

Compiler development in industry

Lot's of research going on into compilers, both conventional and (tracing) JITs. It's super high-tech.

Compiler development in industry

Lot's of research going on into compilers, both conventional and (tracing) JITs. It's super high-tech.

Big companies (Google, Microsoft, Oracle, Intel, Arm, Apple) compete heavily on quality (e.g. speed, energy usage) of their compilers. They have large teams working on this. Find it difficult to hire, because advanced compiler knowledge is rare.

Compiler development in industry

Lot's of research going on into compilers, both conventional and (tracing) JITs. It's super high-tech.

Big companies (Google, Microsoft, Oracle, Intel, Arm, Apple) compete heavily on quality (e.g. speed, energy usage) of their compilers. They have large teams working on this. Find it difficult to hire, because advanced compiler knowledge is rare.

Much work left to be done.

Interested?

Interested?

Compilers (and related subjects) great subject for final year projects.

Interested?

Compilers (and related subjects) great subject for final year projects.

JRA (Junior Research Assistant) in the summer 2019.

Interested?

Compilers (and related subjects) great subject for final year projects.

JRA (Junior Research Assistant) in the summer 2019.

Feel free to talk to me about this.


Merry
Christmas

Happy New year