TIL (Things I Learned)
Performance Analysis of CoreCLR Interpreter

Yuchong Pan

Microsoft

August 6, 2020

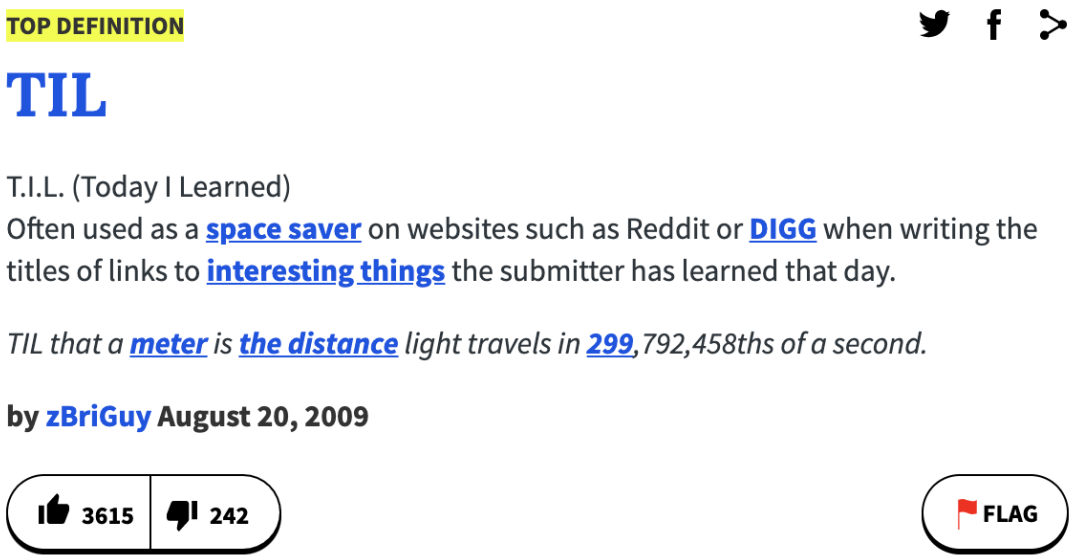
TIL: What's TIL?

TIL: What's TIL?

- TIL = **T**oday **I** **L**earned

TIL: What's TIL?

- TIL = **T**oday **I** **L**earned



TOP DEFINITION

TIL

T.I.L. (Today I Learned)
Often used as a **space saver** on websites such as Reddit or **DIGG** when writing the titles of links to **interesting things** the submitter has learned that day.

*TIL that a **meter** is **the distance** light travels in **299**,792,458ths of a second.*

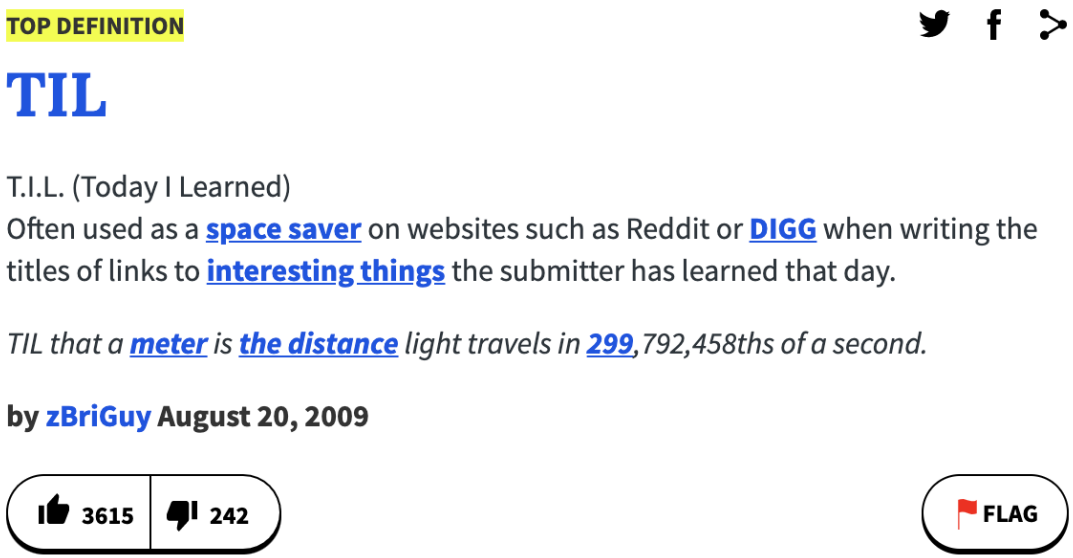
by zBriGuy August 20, 2009

3615 242

FLAG

TIL: What's TIL?

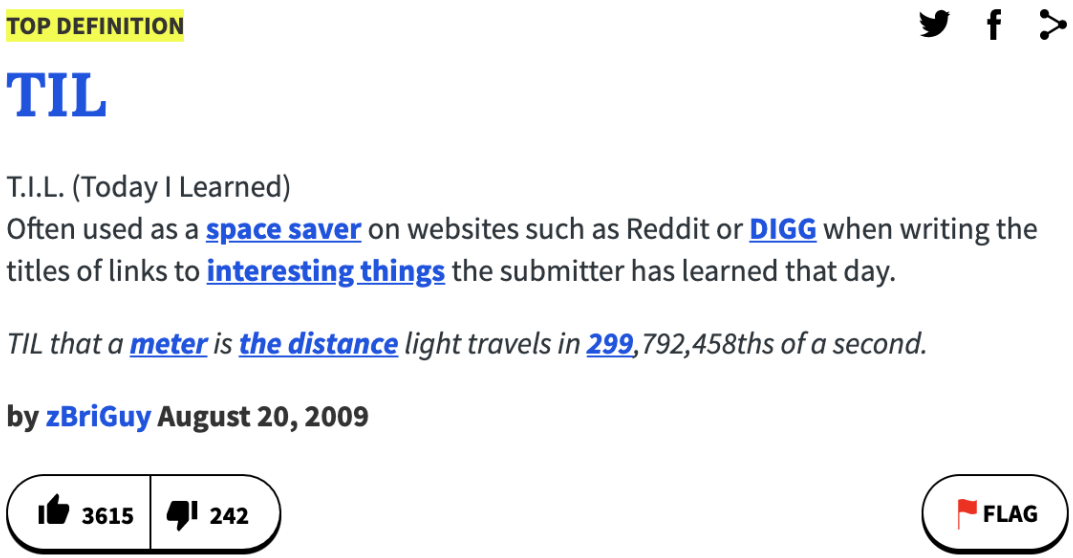
- TIL = **T**oday **I** **L**earned



- TIL = **T**hings **I** **L**earned

TIL: What's TIL?

- TIL = **T**oday **I** **L**earned



- TIL = **T**hings **I** **L**earned
- TIL = **T**hings **I** **L**earned about the CoreCLR interpreter and its performance during my internship in the summer of 2020

TIL: Who Am I?

TIL: Who Am I?

- Yuchong Pan 潘宇冲

TIL: Who Am I?

- Yuchong Pan 潘宇冲
- `http://ypan.me`

TIL: Who Am I?

- Yuchong Pan 潘宇冲
- `http://ypan.me`
- UBC Combined Honours Computer Science and Mathematics

TIL: Who Am I?

- Yuchong Pan 潘宇冲
- `http://ypan.me`
- UBC Combined Honours Computer Science and Mathematics
- It rains a lot in Vancouver, too.

TIL: Who Am I?

- Yuchong Pan 潘宇冲
- `http://ypan.me`
- UBC Combined Honours Computer Science and Mathematics
- It rains a lot in Vancouver, too.
- Compilers, programming languages, and algorithms!

TIL: Who Am I?

- Yuchong Pan 潘宇冲
- `http://ypan.me`
- UBC Combined Honours Computer Science and Mathematics
- It rains a lot in Vancouver, too.
- Compilers, programming languages, and algorithms!
- Third internship at Microsoft

TIL: Who Am I?

- Yuchong Pan 潘宇冲
- `http://ypan.me`
- UBC Combined Honours Computer Science and Mathematics
- It rains a lot in Vancouver, too.
- Compilers, programming languages, and algorithms!
- Third internship at Microsoft
- Last year: .NET Core Uninstall Tool

.NET Core Uninstall Tool

05/27/2020 • 11 minutes to read •

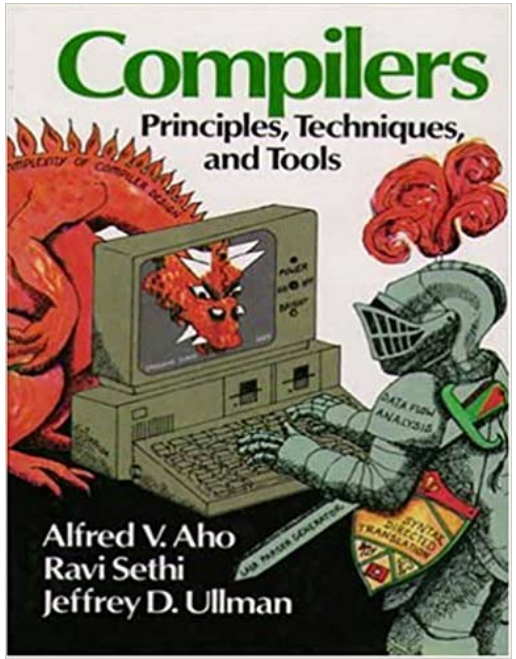
The .NET Core Uninstall Tool (`dotnet-core-uninstall`) lets you remove .NET Core SDKs and Runtimes from a system. A collection of options is available to specify which versions you want to uninstall.

TIL: Who Am I?

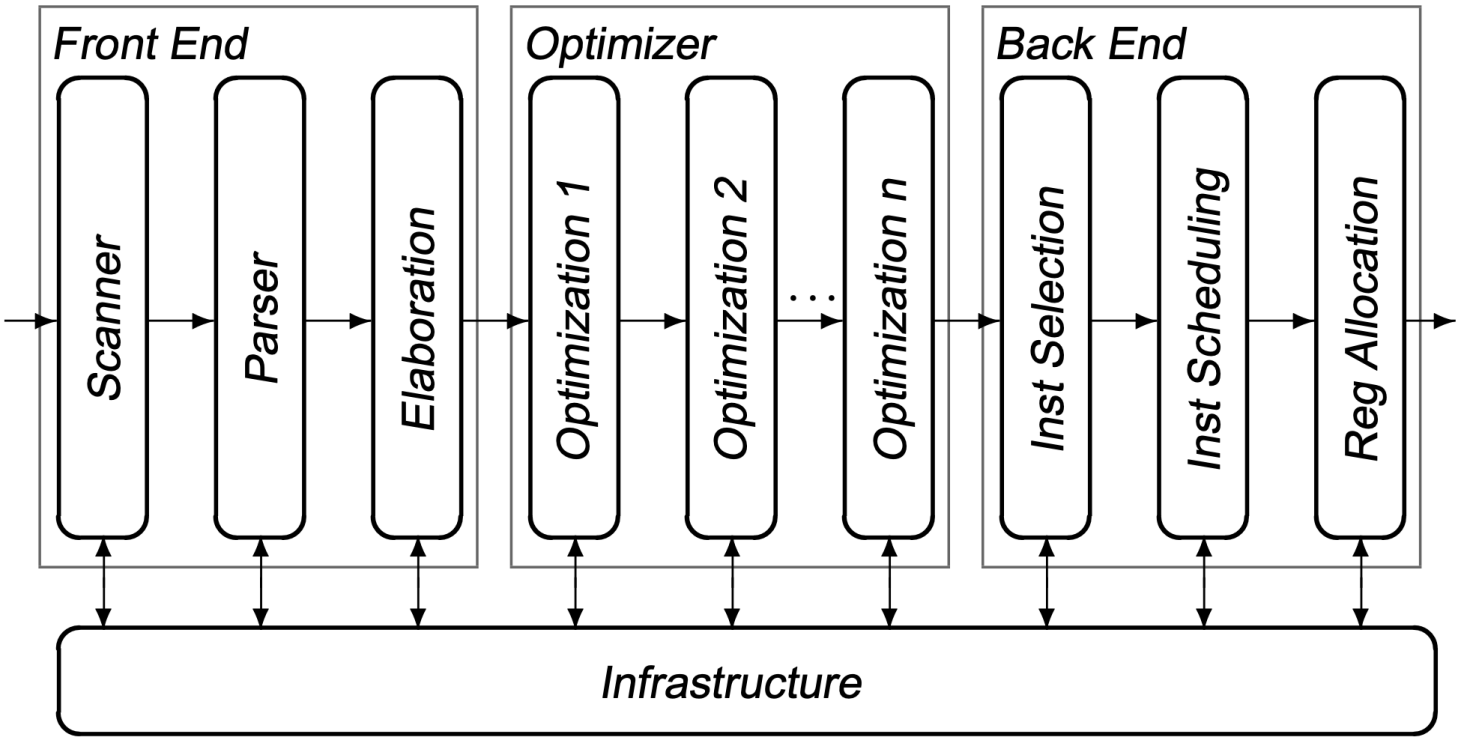
TIL: Why Is the Interpreter So Interesting?

TIL: Why Is the Interpreter So Interesting?

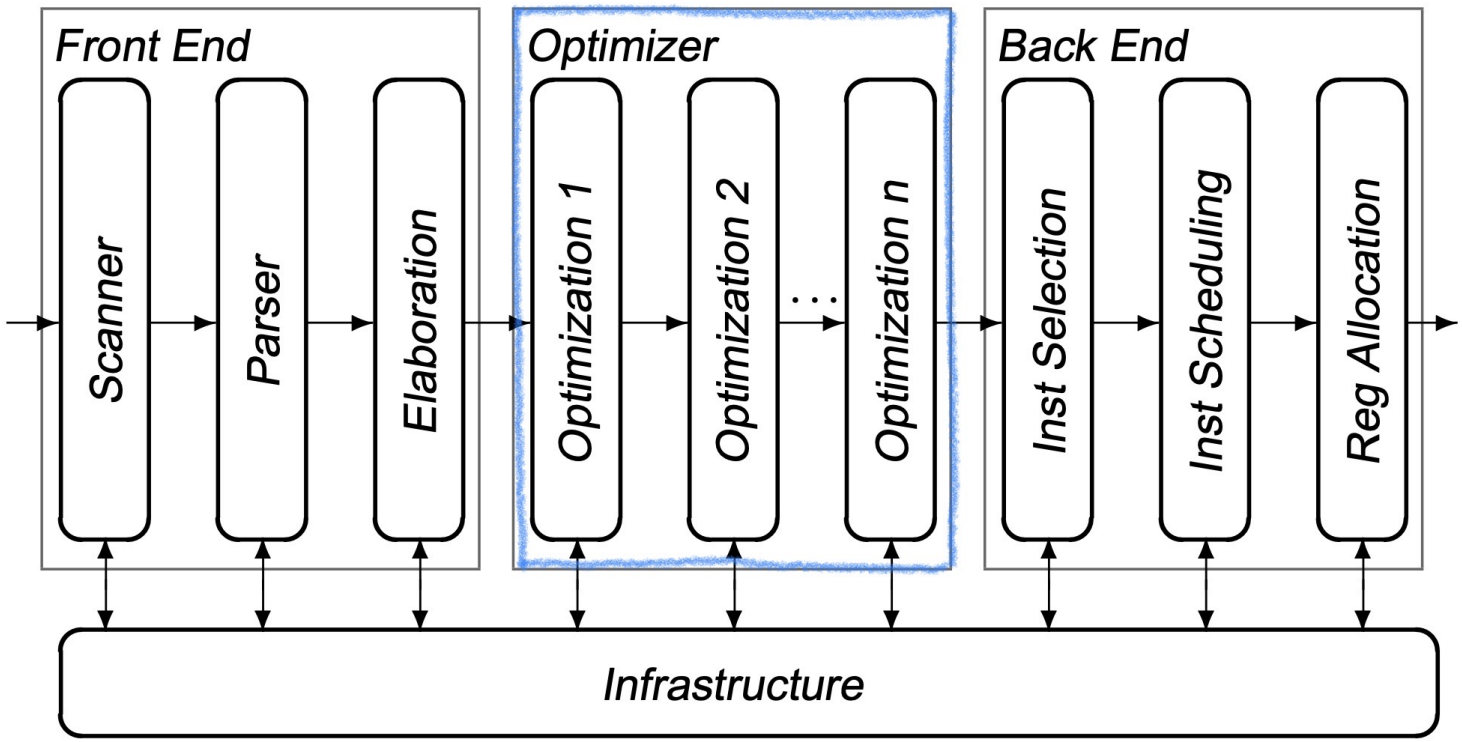
- In a typical, traditional undergraduate compiler course, each method in your program is compiled **once**.



TIL: Why Is the Interpreter So Interesting?



TIL: Why Is the Interpreter So Interesting?



TIL: Why Is the Interpreter So Interesting?

- Execution Time = **Jitting Time** + Assembly Execution Time

TIL: Why Is the Interpreter So Interesting?

- Execution Time = **Jitting Time** + Assembly Execution Time
- Trade-offs!

TIL: Why Is the Interpreter So Interesting?

- Execution Time = **Jitting Time** + Assembly Execution Time
- Trade-offs!

- **More** optimization means **faster** steady-state performance but **slower** start-up time.

TIL: Why Is the Interpreter So Interesting?

- Execution Time = **Jitting Time** + Assembly Execution Time
- Trade-offs!

- **More** optimization means **faster** steady-state performance but **slower** start-up time.
- Alternatively, **less** optimization means **faster** start-up time but **slower** steady-state performance.

TIL: Why Is the Interpreter So Interesting?

TIL: Why Is the Interpreter So Interesting?

- **Tiered Compilation!**

TIL: Why Is the Interpreter So Interesting?

- **Tiered Compilation!**
- Introduced in .NET Core 2.1

TIL: Why Is the Interpreter So Interesting?

- **Tiered Compilation!**
- Introduced in .NET Core 2.1
- Multiple compilations for the same method

TIL: Why Is the Interpreter So Interesting?

- **Tiered Compilation!**
- Introduced in .NET Core 2.1
- Multiple compilations for the same method
- Hot-swapped at runtime

TIL: Why Is the Interpreter So Interesting?

- **Tiered Compilation!**
- Introduced in .NET Core 2.1
- Multiple compilations for the same method
- Hot-swapped at runtime
- **Faster start-up time:** JIT generates the initial compilation quickly, sacrificing code optimization. **(Tier 0, QuickJIT)**

TIL: Why Is the Interpreter So Interesting?

- **Tiered Compilation!**
- Introduced in .NET Core 2.1
- Multiple compilations for the same method
- Hot-swapped at runtime
- **Faster start-up time:** JIT generates the initial compilation quickly, sacrificing code optimization. **(Tier 0, QuickJIT)**
- **Faster steady-state performance:** If a method is called frequently, more optimized code is generated **asynchronously**, which then replaces the initial code. **(Tier 1)**

TIL: Why Is the Interpreter So Interesting?

- **Tiered Compilation!**
- Introduced in .NET Core 2.1
- Multiple compilations for the same method
- Hot-swapped at runtime
- **Faster start-up time:** JIT generates the initial compilation quickly, sacrificing code optimization. **(Tier 0, QuickJIT)**
- **Faster steady-state performance:** If a method is called frequently, more optimized code is generated **asynchronously**, which then replaces the initial code. **(Tier 1)**
- **ReadyToRun** images have several constraints which prohibit some types of optimization. TC generates optimized code that replaces the pre-compiled version.

TIL: Why Is the Interpreter So Interesting?

- Where do we go from here?

TIL: Why Is the Interpreter So Interesting?

- Where do we go from here?
- Execution Time = Jitting Time + Assembly Execution Time

TIL: Why Is the Interpreter So Interesting?

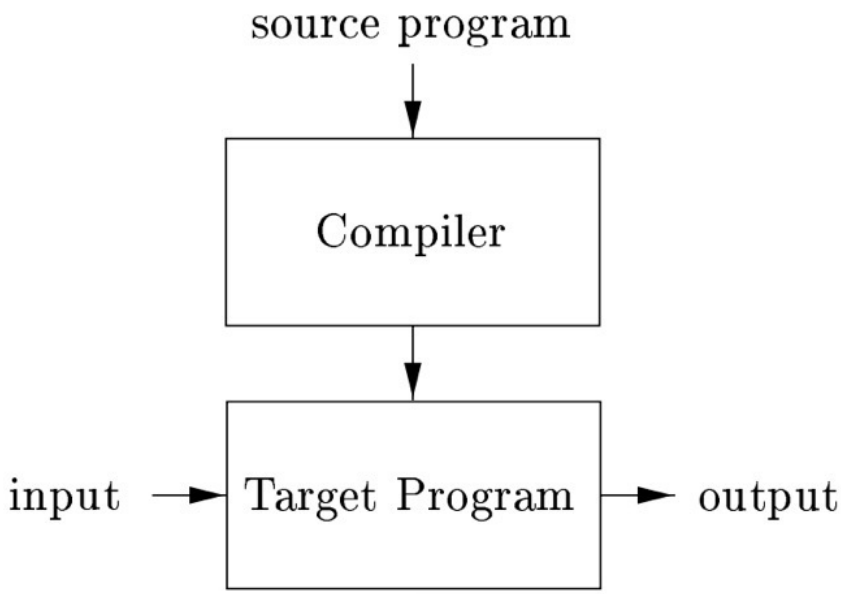
- Where do we go from here?
- Execution Time = Jitting Time + Assembly Execution Time
- Execution Time = ~~Jitting Time~~ + Assembly Execution Time?

TIL: Why Is the Interpreter So Interesting?

- Where do we go from here?
- Execution Time = Jitting Time + Assembly Execution Time
- Execution Time = ~~Jitting Time~~ + Assembly Execution Time?

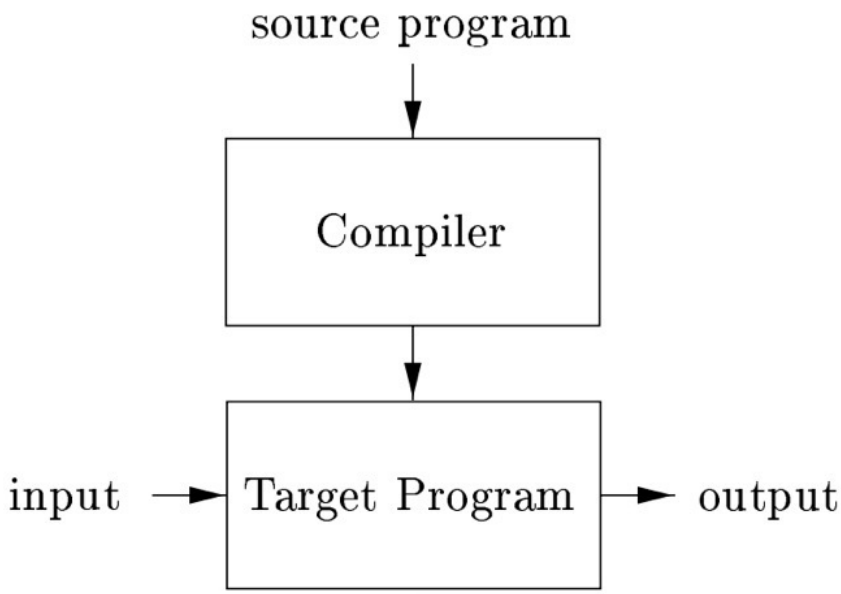
TIL: Why Is the Interpreter So Interesting?

- This is a **compiler**:

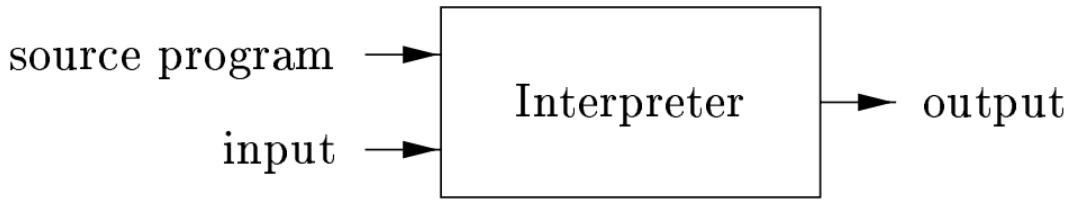


TIL: Why Is the Interpreter So Interesting?

- This is a **compiler**:



- This is an **interpreter**:



TIL: Why Is the Interpreter So Interesting?

- An **interpreter** does not need to JIT assembly code before it is able to execute methods.

TIL: Why Is the Interpreter So Interesting?

- An **interpreter** does not need to JIT assembly code before it is able to execute methods.
- Hence, an **interpreter** executes IL code **immediately**.

TIL: Why Is the Interpreter So Interesting?

- An **interpreter** does not need to JIT assembly code before it is able to execute methods.
- Hence, an **interpreter** executes IL code **immediately**.
- Execution Time $=$ ~~Jitting Time~~ $+$ Interpretation Time

TIL: Why Is the Interpreter So Interesting?

- An **interpreter** does not need to JIT assembly code before it is able to execute methods.
- Hence, an **interpreter** executes IL code **immediately**.
- Execution Time $= \cancel{\text{Jitting Time}} + \text{Interpretation Time}$

Conjecture

Replacing the current Tier 0 JIT with an interpreter in Tiered Compilation would achieve faster start-up time, while preserving the fast steady-state performance from the Tier 1 JIT, and not deteriorating the total execution time by too much.

Where Are We?

Where Are We?

- Why Is the Interpreter So Interesting?

Where Are We?

- Why Is the Interpreter So Interesting?
- ... and Fixes, and Fixes, and Fixes, ...

Where Are We?

- Why Is the Interpreter So Interesting?
- ... and Fixes, and Fixes, and Fixes, ...
- TIL from Beautiful Plots

Where Are We?

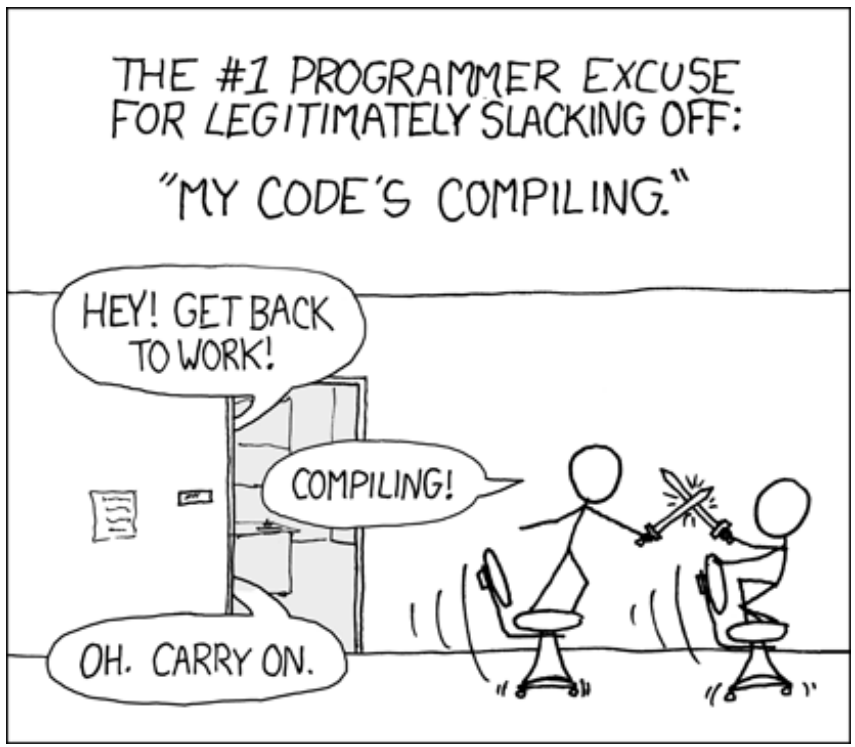
- Why Is the Interpreter So Interesting?
- ... and Fixes, and Fixes, and Fixes, ...
- TIL from Beautiful Plots
- Oh My! Bottlenecks!

Where Are We?

- Why Is the Interpreter So Interesting?
- ... and Fixes, and Fixes, and Fixes, ...
- TIL from Beautiful Plots
- Oh My! Bottlenecks!
- Where Do We Go from Here?

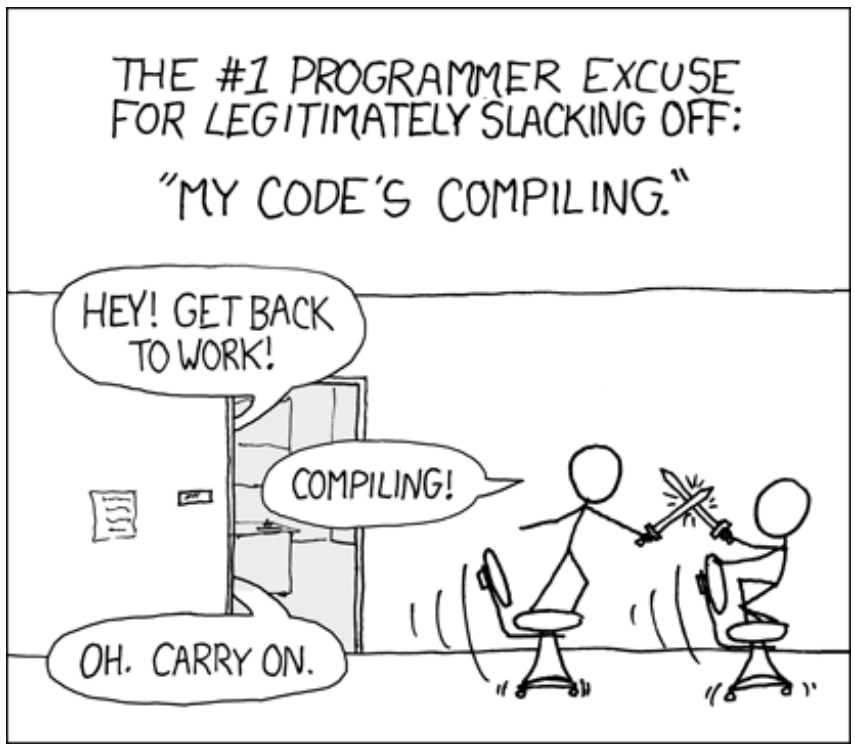
Where Are We?

- Why Is the Interpreter So Interesting?
- ... and Fixes, and Fixes, and Fixes, ...
- TIL from Beautiful Plots
- Oh My! Bottlenecks!
- Where Do We Go from Here?



Where Are We?

- Why Is the Interpreter So Interesting?
- ... and Fixes, and Fixes, and Fixes, ...
- TIL from Beautiful Plots
- Oh My! Bottlenecks!
- Where Do We Go from Here?



TIL: ... and Fixes, and Fixes, and Fixes, ...

TIL: ... and Fixes, and Fixes, and Fixes, ...

- The interpreter did not run on `HelloWorld` in its initial state.

TIL: ... and Fixes, and Fixes, and Fixes, ...

- The interpreter did not run on `HelloWorld` in its initial state.

```
static bool Not(bool p)
{
    bool q = p;
    q = !q;
    return q;
}

static void Main(string[] args)
{
    Console.WriteLine("Hello World!");
    bool b = true;
    bool bb = Not(b);

    bb = Not(bb);
    // ... 30 additional lines of `bb = Not(bb);`

    Console.WriteLine(bb);
}
```

TIL: ... and Fixes, and Fixes, and Fixes, ...

- The interpreter did not run on `HelloWorld` in its initial state.

```
static bool Not(bool p)
{
    bool q = p;
    q = !q;
    return q;
}

static void Main(string[] args)
{
    Console.WriteLine("Hello World!");
    bool b = true;
    bool bb = Not(b);

    bb = Not(bb);
    // ... 30 additional lines of `bb = Not(bb);`

    Console.WriteLine(bb);
}
```

- The number of calls to `Not` is used to trigger the promotion to Tier 1 in Tiered Compilation.

TIL: ... and Fixes, and Fixes, and Fixes, ...

- **Fix #1:** Fixed the type cast compilation errors.

TIL: ... and Fixes, and Fixes, and Fixes, ...

- **Fix #1:** Fixed the type cast compilation errors.
- **Fix #2:** Fixed several bugs in the implementation of the asynchronous tiering-up in Tiered Compilation.

TIL: ... and Fixes, and Fixes, and Fixes, ...

- **Fix #1:** Fixed the type cast compilation errors.
- **Fix #2:** Fixed several bugs in the implementation of the asynchronous tiering-up in Tiered Compilation.
- **Fix #3:** Fixed the call info of virtual function calls to avoid the "Bad IL Format" exception.

TIL: ... and Fixes, and Fixes, and Fixes, ...

- **Fix #1:** Fixed the type cast compilation errors.
- **Fix #2:** Fixed several bugs in the implementation of the asynchronous tiering-up in Tiered Compilation.
- **Fix #3:** Fixed the call info of virtual function calls to avoid the "Bad IL Format" exception.
- **Fix #4:** Provided a workaround for the handling of hardware intrinsics in the interpreter.

TIL: ... and Fixes, and Fixes, and Fixes, ...

- **Fix #1:** Fixed the type cast compilation errors.
- **Fix #2:** Fixed several bugs in the implementation of the asynchronous tiering-up in Tiered Compilation.
- **Fix #3:** Fixed the call info of virtual function calls to avoid the "Bad IL Format" exception.
- **Fix #4:** Provided a workaround for the handling of hardware intrinsics in the interpreter.
- **Fix #5:** Two new COM+ environment variables:

TIL: ... and Fixes, and Fixes, and Fixes, ...

- **Fix #1:** Fixed the type cast compilation errors.
- **Fix #2:** Fixed several bugs in the implementation of the asynchronous tiering-up in Tiered Compilation.
- **Fix #3:** Fixed the call info of virtual function calls to avoid the "Bad IL Format" exception.
- **Fix #4:** Provided a workaround for the handling of hardware intrinsics in the interpreter.
- **Fix #5:** Two new COM+ environment variables:
  - `ForceInterpreter`: Force the interpreter always to be used when executing a program.

TIL: ... and Fixes, and Fixes, and Fixes, ...

- **Fix #1:** Fixed the type cast compilation errors.
- **Fix #2:** Fixed several bugs in the implementation of the asynchronous tiering-up in Tiered Compilation.
- **Fix #3:** Fixed the call info of virtual function calls to avoid the "Bad IL Format" exception.
- **Fix #4:** Provided a workaround for the handling of hardware intrinsics in the interpreter.
- **Fix #5:** Two new COM+ environment variables:
  - `ForceInterpreter`: Force the interpreter always to be used when executing a program.
  - `InterpreterHWIntrinsicsIsSupportedFalse`: Control the workaround in **Fix #4** since it is not production-level.

TIL: ... and Fixes, and Fixes, and Fixes, ...

- **Fix #1:** Fixed the type cast compilation errors.
- **Fix #2:** Fixed several bugs in the implementation of the asynchronous tiering-up in Tiered Compilation.
- **Fix #3:** Fixed the call info of virtual function calls to avoid the "Bad IL Format" exception.
- **Fix #4:** Provided a workaround for the handling of hardware intrinsics in the interpreter.
- **Fix #5:** Two new COM+ environment variables:
  - `ForceInterpreter`: Force the interpreter always to be used when executing a program.
  - `InterpreterHWIntrinsicsIsSupportedFalse`: Control the workaround in **Fix #4** since it is not production-level.

TIL: ... and Fixes, and Fixes, and Fixes, ...

- **Hardware Intrinsics**

TIL: ... and Fixes, and Fixes, and Fixes, ...

- **Hardware Intrinsics**
- **Issue:** The IL implementations of `get_IsSupported()` methods for hardware intrinsics are non-terminating recursions and will be handled by jitted assembly code. The interpreter is unaware of this special case.

TIL: ... and Fixes, and Fixes, and Fixes, ...

- **Hardware Intrinsics**
- **Issue:** The IL implementations of `get_IsSupported()` methods for hardware intrinsics are non-terminating recursions and will be handled by jitted assembly code. The interpreter is unaware of this special case.

```
.method public hidebysig specialname static
        bool  get_IsSupported() cil managed
{
  // Code size       6 (0x6)
  .maxstack  8
  IL_0000:  call       bool System.Runtime.Intrinsics.X86.Sse2::get_IsSupported()
  IL_0005:  ret
} // end of method Sse2::get_IsSupported
```

TIL: ... and Fixes, and Fixes, and Fixes, ...

- **Hardware Intrinsics**
- **Issue:** The IL implementations of `get_IsSupported()` methods for hardware intrinsics are non-terminating recursions and will be handled by jitted assembly code. The interpreter is unaware of this special case.

```
.method public hidebysig specialname static
        bool  get_IsSupported() cil managed
{
  // Code size       6 (0x6)
  .maxstack  8
  IL_0000:  call       bool System.Runtime.Intrinsics.X86.Sse2::get_IsSupported()
  IL_0005:  ret
} // end of method Sse2::get_IsSupported
```

- **"Fix" (a.k.a. Workaround):** Force the interpreter to return `false` for `get_IsSupported()` to disable hardware intrinsics.

TIL: ... and Fixes, and Fixes, and Fixes, ...

- **Hardware Intrinsics**
- **Issue:** The IL implementations of `get_IsSupported()` methods for hardware intrinsics are non-terminating recursions and will be handled by jitted assembly code. The interpreter is unaware of this special case.

```
.method public hidebysig specialname static
        bool  get_IsSupported() cil managed
{
  // Code size       6 (0x6)
  .maxstack  8
  IL_0000:  call       bool System.Runtime.Intrinsics.X86.Sse2::get_IsSupported()
  IL_0005:  ret
} // end of method Sse2::get_IsSupported
```

- **"Fix" (a.k.a. Workaround):** Force the interpreter to return `false` for `get_IsSupported()` to disable hardware intrinsics.
- Controlled by the COM+ environment variable `InterpreterHWIntrinsicsIsSupportedFalse`

TIL: ... and Fixes, and Fixes, and Fixes, ...

- **Issue of "Fix":** Arbitrary combinations of different code generators (JIT, CrossGen, interpreter, etc.) in the same process might cause **inconsistent** values of `IsSupported`. (@jkotas)

TIL: ... and Fixes, and Fixes, and Fixes, ...

- **Issue of "Fix":** Arbitrary combinations of different code generators (JIT, CrossGen, interpreter, etc.) in the same process might cause **inconsistent** values of `IsSupported`. (@jkotas)

```
readonly static RenderingEngine s_engine = Avx.IsSupported ? new AvxRenderingEngine() : new GenericRenderingEngine();

class GenericRenderingEngine : RenderingEngine
{
    virtual void Render()
    {
        Debug.Assert(!Avx.IsSupported);
        // ...
    }

    // ...
}
```

TIL: ... and Fixes, and Fixes, and Fixes, ...

- **Issue of "Fix":** Arbitrary combinations of different code generators (JIT, CrossGen, interpreter, etc.) in the same process might cause **inconsistent** values of `IsSupported`. (@jkotas)

```
readonly static RenderingEngine s_engine = Avx.IsSupported ? new AvxRenderingEngine() : new GenericRenderingEngine();

class GenericRenderingEngine : RenderingEngine
{
    virtual void Render()
    {
        Debug.Assert(!Avx.IsSupported);
        // ...
    }

    // ...
}
```

- **Fix #1:** Depend on Fallback to JIT for hardware intrinsics methods and let JIT generate valid method bodies.

TIL: ... and Fixes, and Fixes, and Fixes, ...

- **Issue of "Fix":** Arbitrary combinations of different code generators (JIT, CrossGen, interpreter, etc.) in the same process might cause **inconsistent** values of `IsSupported`. (@jkotas)

```
readonly static RenderingEngine s_engine = Avx.IsSupported ? new AvxRenderingEngine() : new GenericRenderingEngine();

class GenericRenderingEngine : RenderingEngine
{
    virtual void Render()
    {
        Debug.Assert(!Avx.IsSupported);
        // ...
    }

    // ...
}
```

- **Fix #1:** Depend on Fallback to JIT for hardware intrinsics methods and let JIT generate valid method bodies.
- **Fix #2:** Implement hardware intrinsics methods in the interpreter.

TIL: ... and Fixes, and Fixes, and Fixes, ...

TIL: ... and Fixes, and Fixes, and Fixes, ...

- **Issue of "Fix":** Arbitrary combinations of different code generators (JIT, CrossGen, interpreter, etc.) in the same process might cause **inconsistent** values of `IsSupported`. (@jkotas)

```
readonly static RenderingEngine s_engine = Avx.IsSupported ? new AvxRenderingEngine() : new GenericRenderingEngine();

class GenericRenderingEngine : RenderingEngine
{
    virtual void Render()
    {
        Debug.Assert(!Avx.IsSupported);
        // ...
    }

    // ...
}
```

- **Fix #1:** Depend on Fallback to JIT for hardware intrinsics methods and let JIT generate valid method bodies.
- **Fix #2:** Implement hardware intrinsics methods in the interpreter.

TIL: ... and Fixes, and Fixes, and Fixes, ...

TIL: ... and Fixes, and Fixes, and Fixes, ...

- **TIL:** There are too many arbitrary bugs of the interpreter functionalities to be fixed within a 12-week internship.

TIL: ... and Fixes, and Fixes, and Fixes, ...

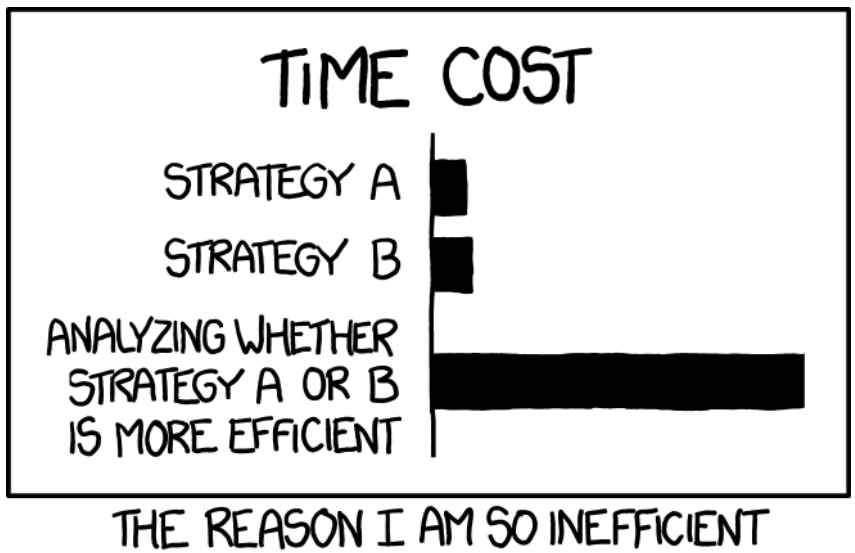
- **TIL:** There are too many arbitrary bugs of the interpreter functionalities to be fixed within a 12-week internship.
- **TIL:** The performance analysis of the interpreter might be more interesting than fixing functionality bugs.

TIL: ... and Fixes, and Fixes, and Fixes, ...

- **TIL:** There are too many arbitrary bugs of the interpreter functionalities to be fixed within a 12-week internship.
- **TIL:** The performance analysis of the interpreter might be more interesting than fixing functionality bugs.
- **Priority 0 tests:**
  Total: 2822
  Failed: 184
  Passing Percentage: 93.48%

Where Are We?

- Why Is the Interpreter So Interesting?
- ... and Fixes, and Fixes, and Fixes, ...
- TIL from Beautiful Plots
- Oh My! Bottlenecks!
- Where Do We Go from Here?



TIL from Beautiful Plots

TIL from Beautiful Plots

Conjecture

Replacing the current Tier 0 JIT with an interpreter in Tiered Compilation would achieve faster start-up time, while preserving the fast steady-state performance from the Tier 1 JIT, and not deteriorating the total execution time by too much.

TIL from Beautiful Plots

Conjecture

Replacing the current Tier 0 JIT with an interpreter in Tiered Compilation would achieve faster start-up time, while preserving the fast steady-state performance from the Tier 1 JIT, and not deteriorating the total execution time by too much.

- **Steady-state performance:** `SciMark 2.0` with `BenchmarkDotNet`, including benchmarks of scientific and numerical computing, e.g. Fast Fourier Transform (FFT), Successive Over-Relaxation (SOR), Monte Carlo Integration, Sparse Matrix Multiply, and LU Matrix Factorization.

TIL from Beautiful Plots

Conjecture

Replacing the current Tier 0 JIT with an interpreter in Tiered Compilation would achieve faster start-up time, while preserving the fast steady-state performance from the Tier 1 JIT, and not deteriorating the total execution time by too much.

- **Steady-state performance:** `SciMark 2.0` with `BenchmarkDotNet`, including benchmarks of scientific and numerical computing, e.g. Fast Fourier Transform (FFT), Successive Over-Relaxation (SOR), Monte Carlo Integration, Sparse Matrix Multiply, and LU Matrix Factorization.
- **Question #1:** How much is the interpreter slower than the Tier 0 and Tier 1 JIT, respectively?

TIL from Beautiful Plots

Conjecture

Replacing the current Tier 0 JIT with an interpreter in Tiered Compilation would achieve faster start-up time, while preserving the fast steady-state performance from the Tier 1 JIT, and not deteriorating the total execution time by too much.

- **Steady-state performance:** `SciMark 2.0` with `BenchmarkDotNet`, including benchmarks of scientific and numerical computing, e.g. Fast Fourier Transform (FFT), Successive Over-Relaxation (SOR), Monte Carlo Integration, Sparse Matrix Multiply, and LU Matrix Factorization.
- **Question #1:** How much is the interpreter slower than the Tier 0 and Tier 1 JIT, respectively?
- **Question #2:** Is the steady-state performance after several warm-up iterations close to the Tier 1 JIT performance?

TIL from Beautiful Plots

- Five configurations:

TIL from Beautiful Plots

- Five configurations:
- `default`: the current default configuration of Tiered Compilation, with a tiering-up threshold of 30

TIL from Beautiful Plots

- Five configurations:
- `default`: the current default configuration of Tiered Compilation, with a tiering-up threshold of 30
- `jit-min-opts`: always JIT and execute the minimally optimized assembly code (Tier 0)

TIL from Beautiful Plots

- Five configurations:
- `default`: the current default configuration of Tiered Compilation, with a tiering-up threshold of 30
- `jit-min-opts`: always JIT and execute the minimally optimized assembly code (Tier 0)
- `tier1-only`: always JIT and execute the highly optimized assembly code (Tier 1)

TIL from Beautiful Plots

- Five configurations:
- `default`: the current default configuration of Tiered Compilation, with a tiering-up threshold of 30
- `jit-min-opts`: always JIT and execute the minimally optimized assembly code (Tier 0)
- `tier1-only`: always JIT and execute the highly optimized assembly code (Tier 1)
- `interp-only`: always interpret IL code with the interpreter

TIL from Beautiful Plots

- Five configurations:
- `default`: the current default configuration of Tiered Compilation, with a tiering-up threshold of 30
- `jit-min-opts`: always JIT and execute the minimally optimized assembly code (Tier 0)
- `tier1-only`: always JIT and execute the highly optimized assembly code (Tier 1)
- `interp-only`: always interpret IL code with the interpreter
- `interp-tc`: Tiered Compilation with Tier 0 substituted with the interpreter, with a tiering-up threshold of 0

TIL from Beautiful Plots

- **Question #1:** How much is the interpreter slower than the Tier 0 and Tier 1 JIT, respectively?

TIL from Beautiful Plots

- **Question #1:** How much is the interpreter slower than the Tier 0 and Tier 1 JIT, respectively?
- Computed the average execution time over 15-20 iterations for each benchmark, after throwing away warm-up iterations

TIL from Beautiful Plots

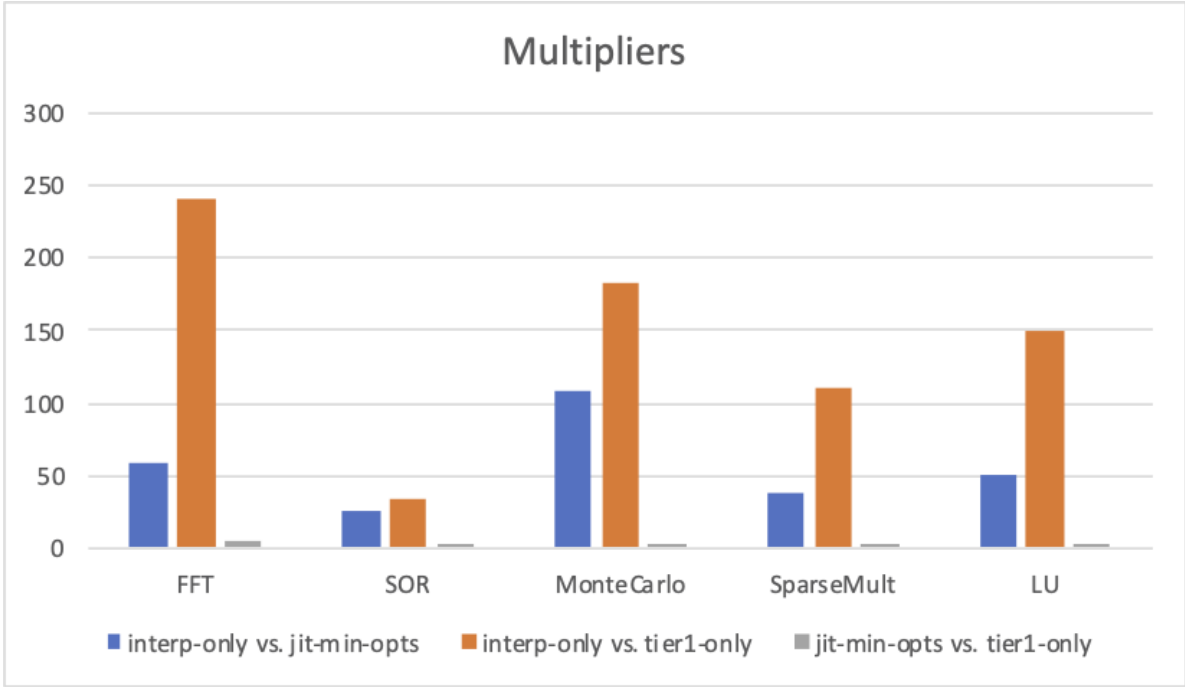
- **Question #1:** How much is the interpreter slower than the Tier 0 and Tier 1 JIT, respectively?
- Computed the average execution time over 15-20 iterations for each benchmark, after throwing away warm-up iterations

	interp-only vs. jit-min-opts	interp-only vs. tier1-only	jit-min-opts vs. tier1-only
FFT	58.51598622	241.0419967	4.119250349
SOR	25.35198702	33.61917818	1.326096379
MonteCarlo	107.8597921	182.8229512	1.695005596
SparseMult	37.65475619	109.7832527	2.915521539
LU	49.93685214	148.7842465	2.979447844

TIL from Beautiful Plots

- **Question #1:** How much is the interpreter slower than the Tier 0 and Tier 1 JIT, respectively?
- Computed the average execution time over 15-20 iterations for each benchmark, after throwing away warm-up iterations

	interp-only vs. jit-min-opts	interp-only vs. tier1-only	jit-min-opts vs. tier1-only
FFT	58.51598622	241.0419967	4.119250349
SOR	25.35198702	33.61917818	1.326096379
MonteCarlo	107.8597921	182.8229512	1.695005596
SparseMult	37.65475619	109.7832527	2.915521539
LU	49.93685214	148.7842465	2.979447844



TIL from Beautiful Plots

	interp-only vs. jit-min-opts	interp-only vs. tier1-only	jit-min-opts vs. tier1-only
FFT	58.51598622	241.0419967	4.119250349
SOR	25.35198702	33.61917818	1.326096379
MonteCarlo	107.8597921	182.8229512	1.695005596
SparseMult	37.65475619	109.7832527	2.915521539
LU	49.93685214	148.7842465	2.979447844

TIL from Beautiful Plots

	interp-only vs. jit-min-opts	interp-only vs. tier1-only	jit-min-opts vs. tier1-only
FFT	58.51598622	241.0419967	4.119250349
SOR	25.35198702	33.61917818	1.326096379
MonteCarlo	107.8597921	182.8229512	1.695005596
SparseMult	37.65475619	109.7832527	2.915521539
LU	49.93685214	148.7842465	2.979447844

- Compared to QuickJIT, the interpreter is 25-times slower in the best case (`SOR`), and 108-times slower in the worst case (`MonteCarlo`).

TIL from Beautiful Plots

	interp-only vs. jit-min-opts	interp-only vs. tier1-only	jit-min-opts vs. tier1-only
FFT	58.51598622	241.0419967	4.119250349
SOR	25.35198702	33.61917818	1.326096379
MonteCarlo	107.8597921	182.8229512	1.695005596
SparseMult	37.65475619	109.7832527	2.915521539
LU	49.93685214	148.7842465	2.979447844

- Compared to QuickJIT, the interpreter is 25-times slower in the best case (`SOR`), and 108-times slower in the worst case (`MonteCarlo`).
- Compared to Tier 1, the interpreter is 34-times slower in the best case (`SOR`), and 241-times slower in the worst case (`FFT`).

TIL from Beautiful Plots

	interp-only vs. jit-min-opts	interp-only vs. tier1-only	jit-min-opts vs. tier1-only
FFT	58.51598622	241.0419967	4.119250349
SOR	25.35198702	33.61917818	1.326096379
MonteCarlo	107.8597921	182.8229512	1.695005596
SparseMult	37.65475619	109.7832527	2.915521539
LU	49.93685214	148.7842465	2.979447844

- Compared to QuickJIT, the interpreter is 25-times slower in the best case (`SOR`), and 108-times slower in the worst case (`MonteCarlo`).
- Compared to Tier 1, the interpreter is 34-times slower in the best case (`SOR`), and 241-times slower in the worst case (`FFT`).

TIL from Beautiful Plots

- **Accidentally**, also measured the steady-state performance for each benchmark with a **Checked** build.

TIL from Beautiful Plots

- **Accidentally**, also measured the steady-state performance for each benchmark with a **Checked** build.

	interp-only vs. jit-min-opts	interp-only vs. tier1-only	jit-min-opts vs. tier1-only
FFT	473.5579762	1806.993473	3.815780884
SOR	333.4660434	440.5411904	1.321097602
MonteCarlo	2306.499301	3937.286102	1.707039798
SparseMult	410.4487699	1213.298956	2.956030191
LU	380.695905	1595.84787	4.191922868

TIL from Beautiful Plots

- **Accidentally**, also measured the steady-state performance for each benchmark with a **Checked** build.

	interp-only vs. jit-min-opts	interp-only vs. tier1-only	jit-min-opts vs. tier1-only
FFT	473.5579762	1806.993473	3.815780884
SOR	333.4660434	440.5411904	1.321097602
MonteCarlo	2306.499301	3937.286102	1.707039798
SparseMult	410.4487699	1213.298956	2.956030191
LU	380.695905	1595.84787	4.191922868

Observation

The multipliers with a **Checked** build is between 333–2306 compared to `jit-min-opts`, and between 441–3937 compared to `tier1-only`, much greater than a **Release** build.

TIL from Beautiful Plots

- **Accidentally**, also measured the steady-state performance for each benchmark with a **Checked** build.

	interp-only vs. jit-min-opts	interp-only vs. tier1-only	jit-min-opts vs. tier1-only
FFT	473.5579762	1806.993473	3.815780884
SOR	333.4660434	440.5411904	1.321097602
MonteCarlo	2306.499301	3937.286102	1.707039798
SparseMult	410.4487699	1213.298956	2.956030191
LU	380.695905	1595.84787	4.191922868

Observation

The multipliers with a **Checked** build is between 333–2306 compared to `jit-min-opts`, and between 441–3937 compared to `tier1-only`, much greater than a **Release** build.

Conjecture

The interpretation of IL code is the execution of native C++ code. A **Release** build disables many assertions and enables many optimizations of the native code.

TIL from Beautiful Plots

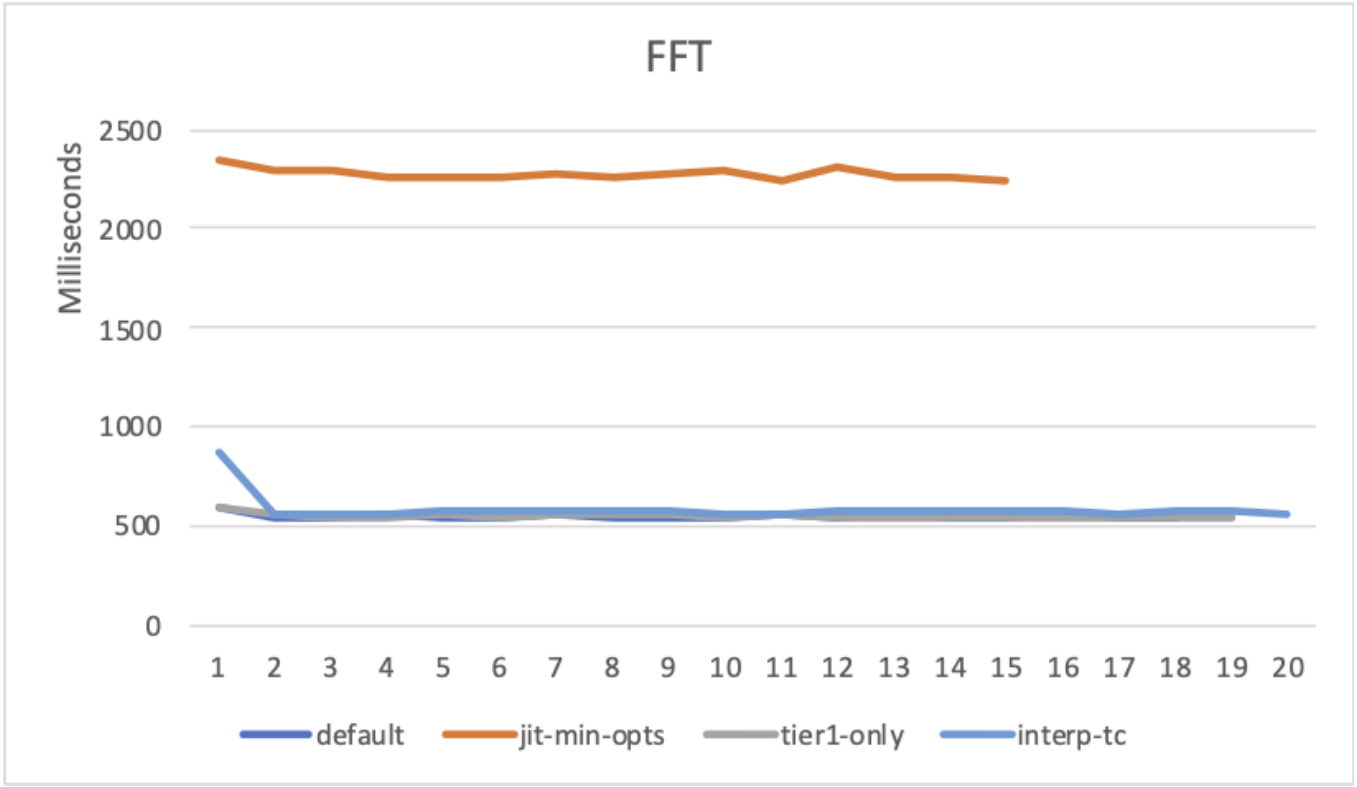
- **Question #2:** Is the steady-state performance after several warm-up iterations close to the Tier 1 JIT performance?

TIL from Beautiful Plots

- **Question #2:** Is the steady-state performance after several warm-up iterations close to the Tier 1 JIT performance?
- Yes.

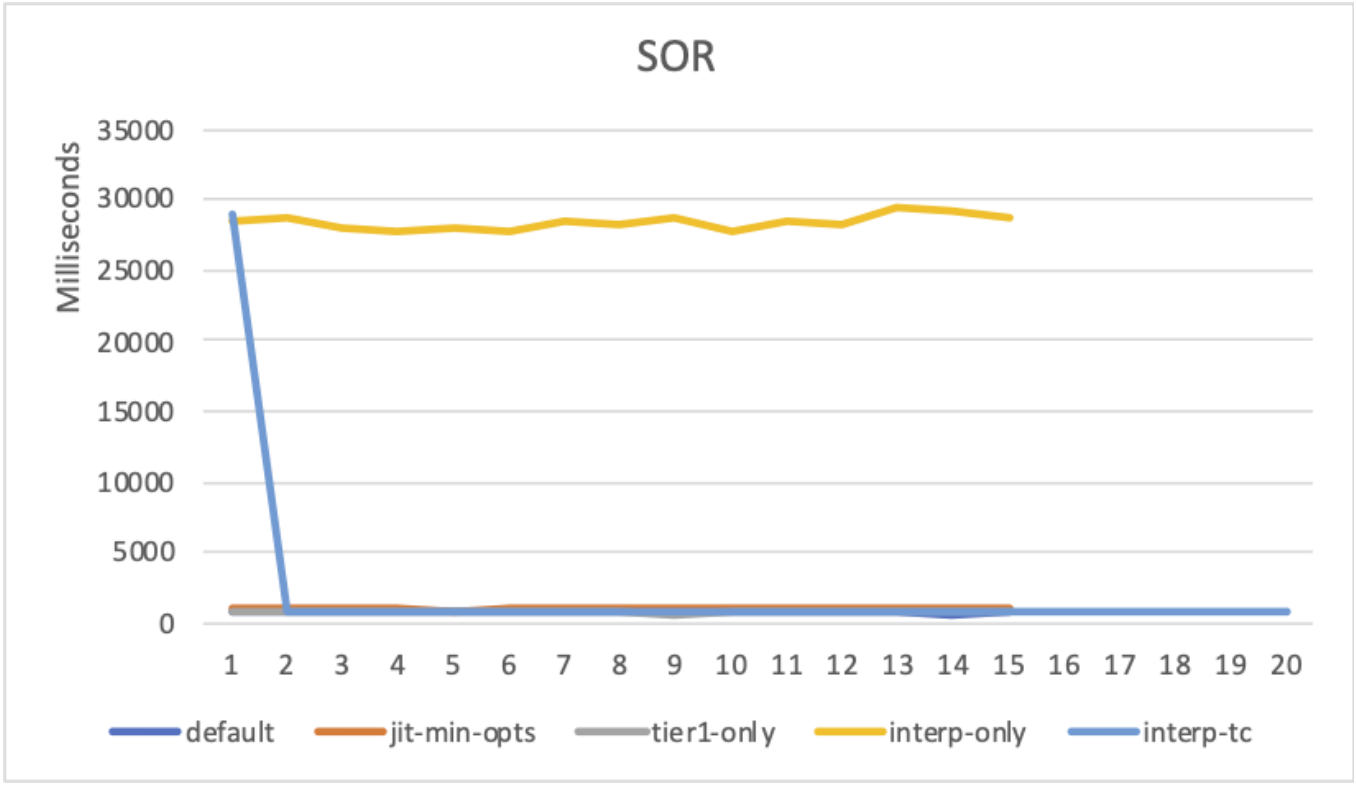
TIL from Beautiful Plots

- **Question #2:** Is the steady-state performance after several warm-up iterations close to the Tier 1 JIT performance?
- Yes.



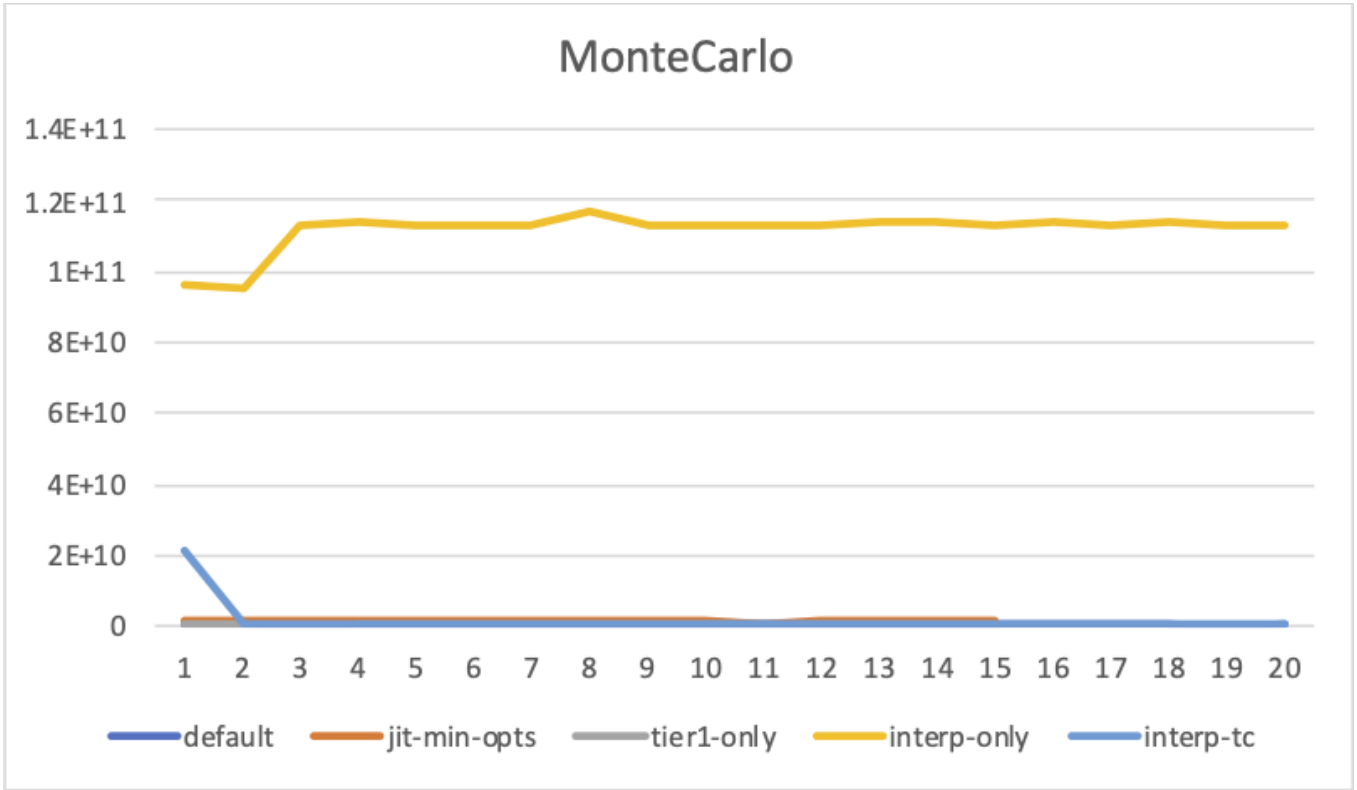
TIL from Beautiful Plots

- **Question #2:** Is the steady-state performance after several warm-up iterations close to the Tier 1 JIT performance?
- Yes.



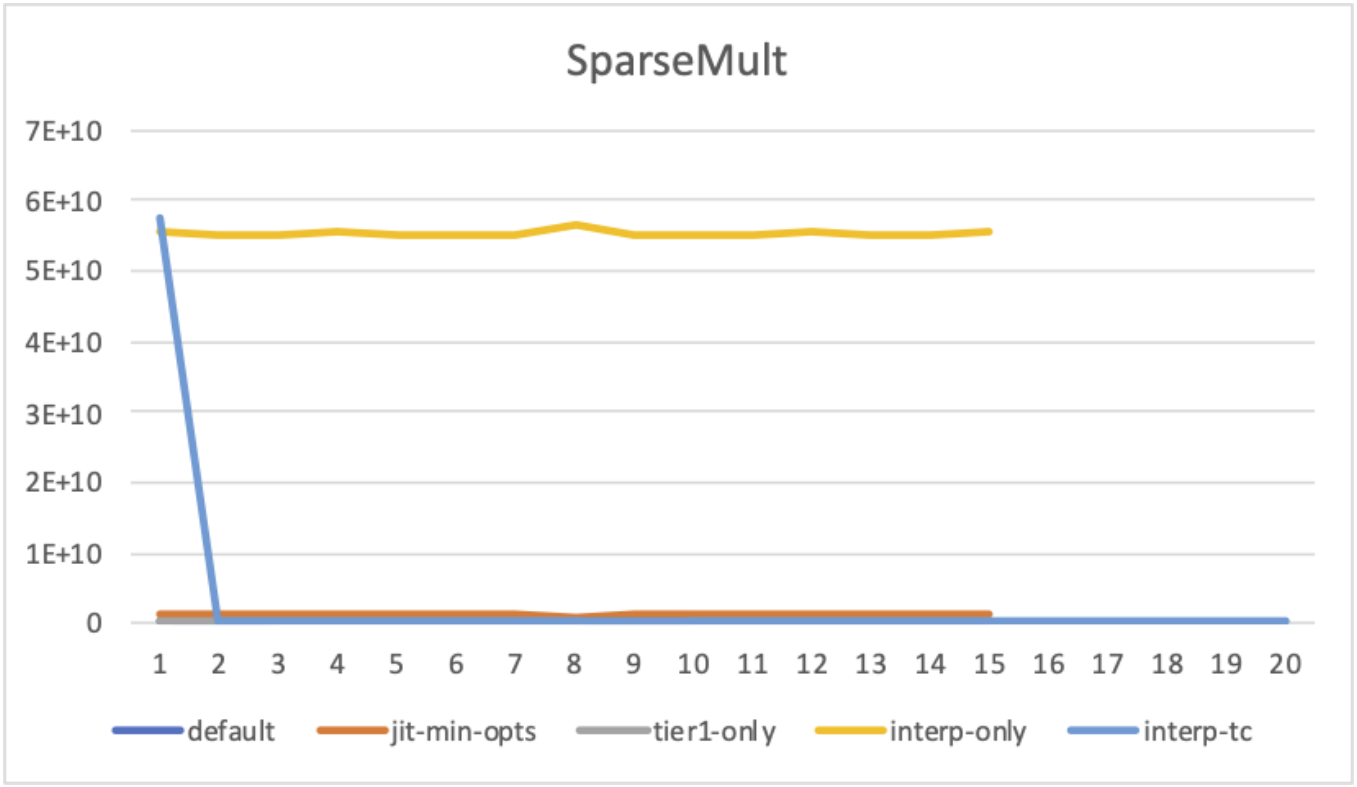
TIL from Beautiful Plots

- **Question #2:** Is the steady-state performance after several warm-up iterations close to the Tier 1 JIT performance?
- Yes.



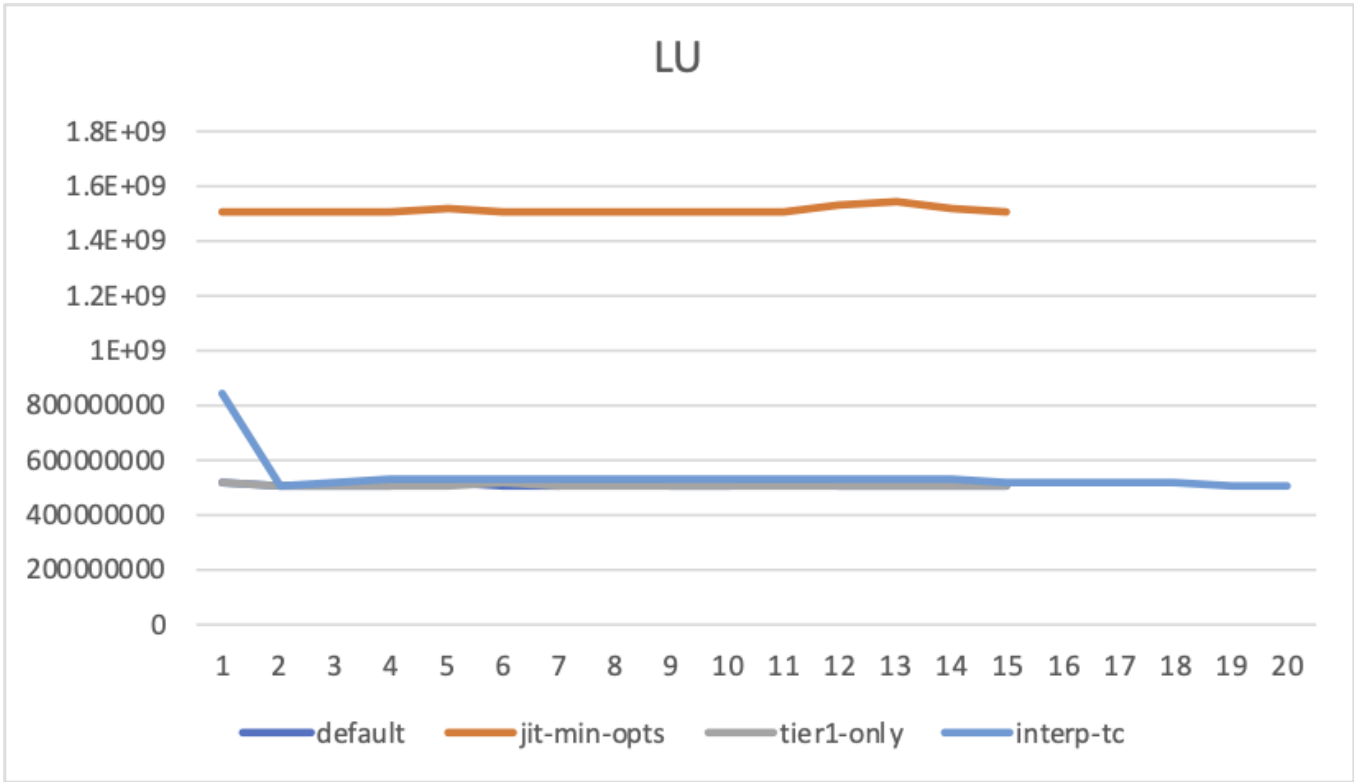
TIL from Beautiful Plots

- **Question #2:** Is the steady-state performance after several warm-up iterations close to the Tier 1 JIT performance?
- Yes.



TIL from Beautiful Plots

- **Question #2:** Is the steady-state performance after several warm-up iterations close to the Tier 1 JIT performance?
- Yes.



TIL from Beautiful Plots

- **Start-up time**

TIL from Beautiful Plots

- **Start-up time**
- A computational loop, where the number of iterations taken from an exponential sequence $1, 2, 4, 8, 16, \ldots$. Plot the total execution time of the computational loop against the number of iterations. (@BruceForstall @AndyAyersMS)

TIL from Beautiful Plots

- **Start-up time**
- A computational loop, where the number of iterations taken from an exponential sequence $1, 2, 4, 8, 16, \ldots$. Plot the total execution time of the computational loop against the number of iterations. (@BruceForstall @AndyAyersMS)
- Two additional configurations:

TIL from Beautiful Plots

- **Start-up time**
- A computational loop, where the number of iterations taken from an exponential sequence $1, 2, 4, 8, 16, \ldots$. Plot the total execution time of the computational loop against the number of iterations. (@BruceForstall @AndyAyersMS)
- Two additional configurations:
- `interp-without-loops`: interpret IL code with the interpreter, except that methods containing loops are JIT'ed

TIL from Beautiful Plots

- **Start-up time**
- A computational loop, where the number of iterations taken from an exponential sequence $1, 2, 4, 8, 16, \ldots$. Plot the total execution time of the computational loop against the number of iterations. (@BruceForstall @AndyAyersMS)
- Two additional configurations:
- `interp-without-loops`: interpret IL code with the interpreter, except that methods containing loops are JIT'ed
- `interp-tc-without-loops`: Tiered Compilation with Tier 0 substituted with the interpreter with a tiering-up threshold of 0, except that methods containing loops are JIT'ed

TIL from Beautiful Plots

- **Start-up time**
- A computational loop, where the number of iterations taken from an exponential sequence $1, 2, 4, 8, 16, \ldots$. Plot the total execution time of the computational loop against the number of iterations. (@BruceForstall @AndyAyersMS)
- Two additional configurations:
- `interp-without-loops`: interpret IL code with the interpreter, except that methods containing loops are JIT'ed
- `interp-tc-without-loops`: Tiered Compilation with Tier 0 substituted with the interpreter with a tiering-up threshold of 0, except that methods containing loops are JIT'ed
- **Question #1:** Does the interpreter easily beat QuickJIT and Tier 1 JIT for small iteration counts?

TIL from Beautiful Plots

- **Start-up time**
- A computational loop, where the number of iterations taken from an exponential sequence $1, 2, 4, 8, 16, \ldots$. Plot the total execution time of the computational loop against the number of iterations. (@BruceForstall @AndyAyersMS)
- Two additional configurations:
- `interp-without-loops`: interpret IL code with the interpreter, except that methods containing loops are JIT'ed
- `interp-tc-without-loops`: Tiered Compilation with Tier 0 substituted with the interpreter with a tiering-up threshold of 0, except that methods containing loops are JIT'ed
- **Question #1:** Does the interpreter easily beat QuickJIT and Tier 1 JIT for small iteration counts?
- **Question #2:** What are the **crossover** times when QuickJIT beats the interpreter, and when Tier 1 JIT beats QuickJIT?

TIL from Beautiful Plots

```
public static int execute(int num_iterations)
{
    int i;
    int a = 0, b = 1;

    for (i = 0; i < num_iterations; i++)
    {
        update(ref a, ref b);
    }

    return a;
}
```

```
[MethodImplAttribute(MethodImplOptions.NoInlining)]
private static void update(ref int a, ref int b)
{
    int c = a + b;

    if (c >= MOD)
    {
        c -= MOD;
    }

    a = b;
    b = c;
}
```

TIL from Beautiful Plots

```
public static int execute(int num_iterations)
{
    int i;
    int a = 0, b = 1;

    for (i = 0; i < num_iterations; i++)
    {
        update(ref a, ref b);
    }

    return a;
}
```

```
[MethodImplAttribute(MethodImplOptions.NoInlining)]
private static void update(ref int a, ref int b)
{
    int c = a + b;

    if (c >= MOD)
    {
        c -= MOD;
    }

    a = b;
    b = c;
}
```

- `execute` is only invocated **once** for each benchmark, and hence will not be tiered up.

TIL from Beautiful Plots

```
public static int execute(int num_iterations)
{
    int i;
    int a = 0, b = 1;

    for (i = 0; i < num_iterations; i++)
    {
        update(ref a, ref b);
    }

    return a;
}
```

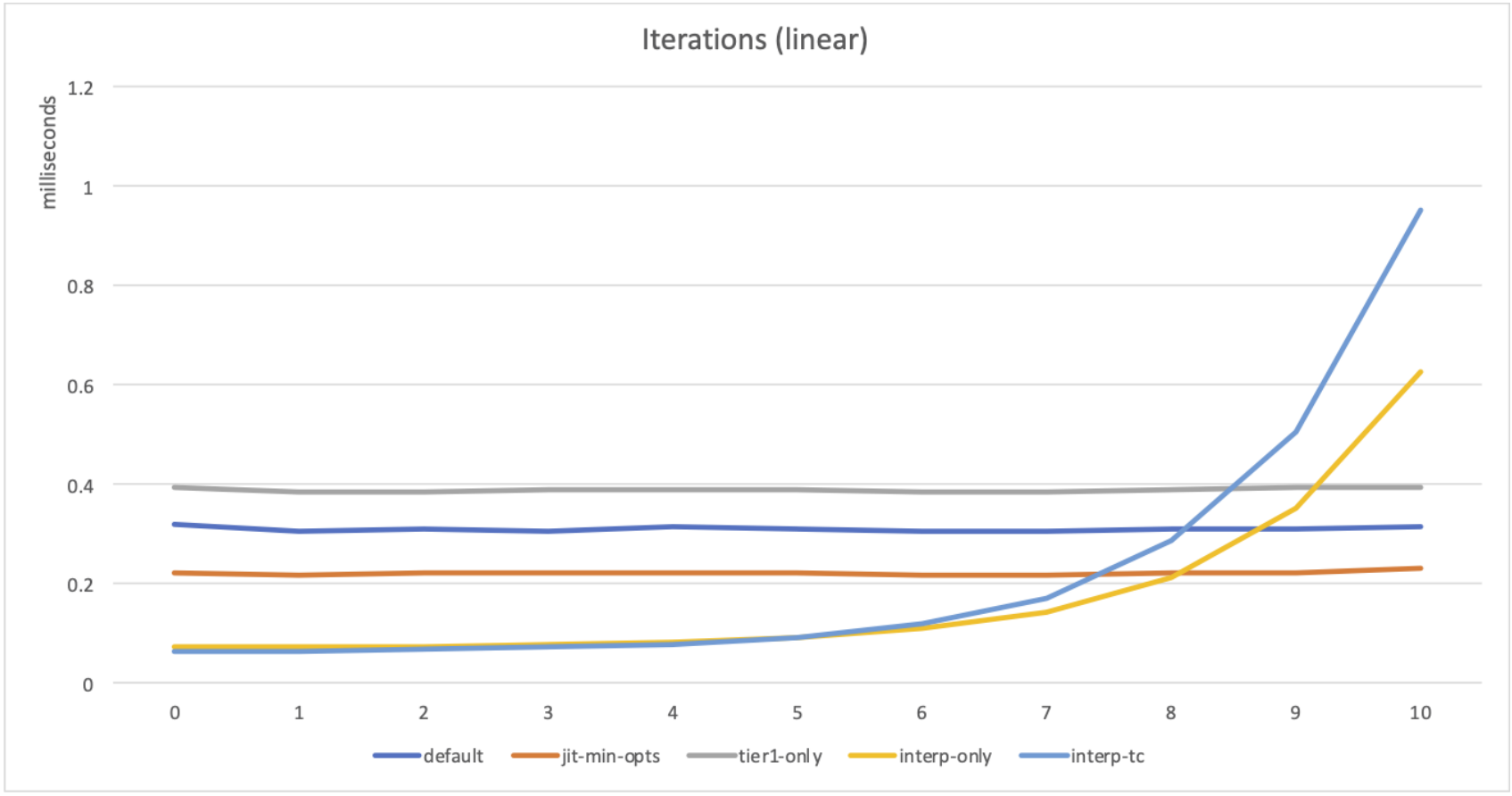
```
[MethodImplAttribute(MethodImplOptions.NoInlining)]
private static void update(ref int a, ref int b)
{
    int c = a + b;

    if (c >= MOD)
    {
        c -= MOD;
    }

    a = b;
    b = c;
}
```

- `execute` is only invocated **once** for each benchmark, and hence will not be tiered up.
- Set `COMPlus_InterpreterDoLoopMethods=0` to JIT `execute` while interpreting `update`.

TIL from Beautiful Plots



TIL from Beautiful Plots

- The JIT curves are **flat** for small iteration counts. The y-values of the flat curves correspond to the jitting time of different configurations.

TIL from Beautiful Plots

- The JIT curves are **flat** for small iteration counts. The y-values of the flat curves correspond to the jitting time of different configurations.
- The execution time of a single iteration approximately equals the start-up time of the interpreter.

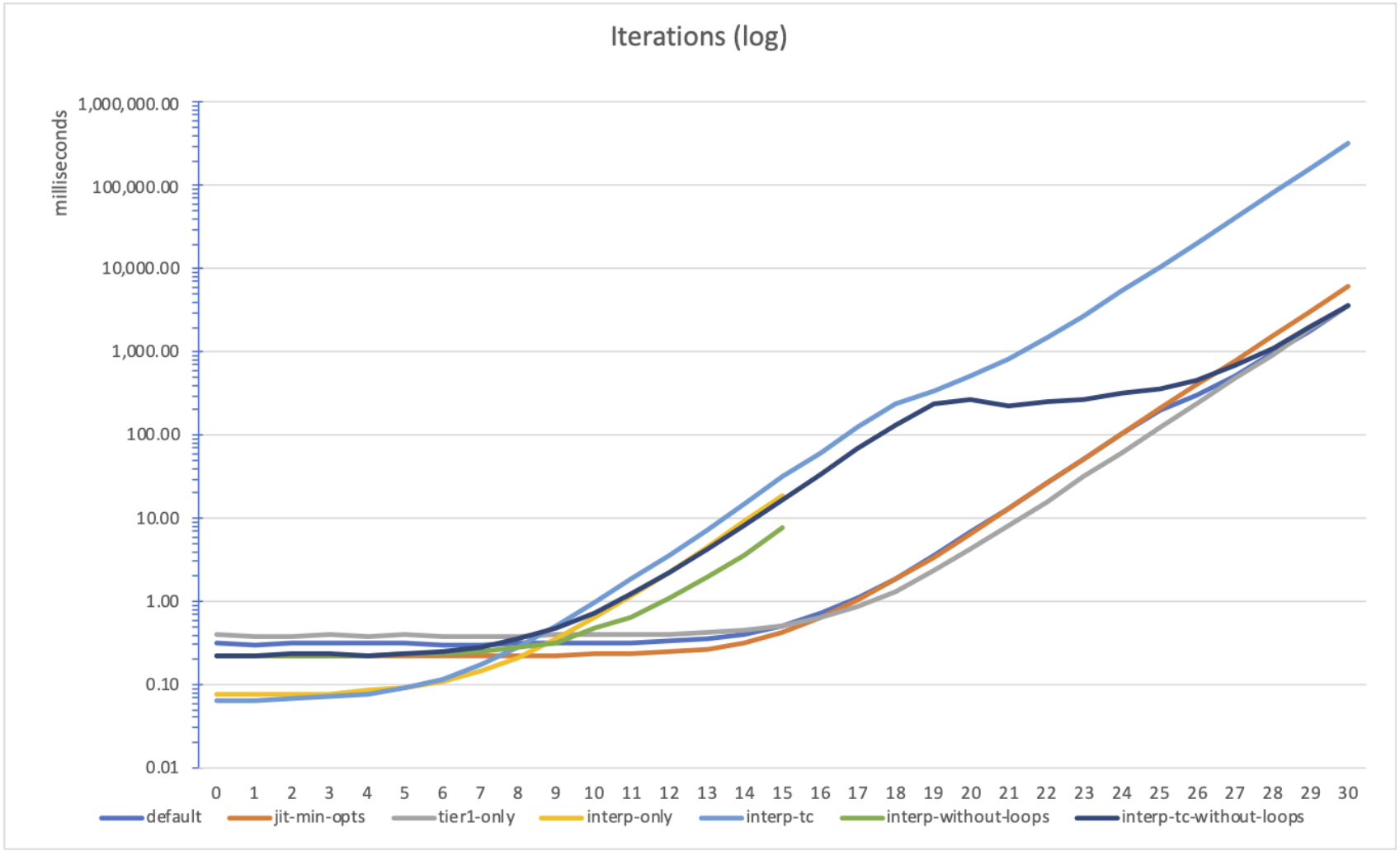
TIL from Beautiful Plots

- The JIT curves are **flat** for small iteration counts. The y-values of the flat curves correspond to the jitting time of different configurations.
- The execution time of a single iteration approximately equals the start-up time of the interpreter.
- **TIL:** The start-up performance of the interpreter (≈ 0.075 ms) is approximately 3 times that of QuickJIT (≈ 0.22 ms), and 5 times that of Tier 1 JIT (≈ 0.39 ms).

TIL from Beautiful Plots

- The JIT curves are **flat** for small iteration counts. The y-values of the flat curves correspond to the jitting time of different configurations.
- The execution time of a single iteration approximately equals the start-up time of the interpreter.
- **TIL:** The start-up performance of the interpreter (≈ 0.075 ms) is approximately 3 times that of QuickJIT (≈ 0.22 ms), and 5 times that of Tier 1 JIT (≈ 0.39 ms).
- **TIL:** The interpreter beats QuickJIT for the first 2^8 iterations, and beats Tier 1 JIT for the first 2^9 iteration.

TIL from Beautiful Plots



TIL from Beautiful Plots

- `default` gets tiered up at the 2^{25}th iteration. (threshold: 30)

TIL from Beautiful Plots

- `default` gets tiered up at the 2^{25}th iteration. (threshold: 30)
- `interp-tc` and `interp-tc-without-loops` get tiered up at the 2^{18}th and 2^{19}th iterations, respectively. (thresholds: 0)

TIL from Beautiful Plots

- `default` gets tiered up at the 2^{25}th iteration. (threshold: 30)
- `interp-tc` and `interp-tc-without-loops` get tiered up at the 2^{18}th and 2^{19}th iterations, respectively. (thresholds: 0)
- **TIL:** The **actual**, **asynchronous** tiering-up occurs much later after it is queued in the background.

TIL from Beautiful Plots

- `default` gets tiered up at the 2^{25}th iteration. (threshold: 30)
- `interp-tc` and `interp-tc-without-loops` get tiered up at the 2^{18}th and 2^{19}th iterations, respectively. (thresholds: 0)
- **TIL:** The **actual**, **asynchronous** tiering-up occurs much later after it is queued in the background.
- **Question:** Would **synchronous** tiering-up have better overall performance if Tier 0 is the interpreter?

TIL from Beautiful Plots

- `default` gets tiered up at the 2^{25}th iteration. (threshold: 30)
- `interp-tc` and `interp-tc-without-loops` get tiered up at the 2^{18}th and 2^{19}th iterations, respectively. (thresholds: 0)
- **TIL:** The **actual**, **asynchronous** tiering-up occurs much later after it is queued in the background.
- **Question:** Would **synchronous** tiering-up have better overall performance if Tier 0 is the interpreter? What about a combination of synchronous and asynchronous tiering-up?

TIL from Beautiful Plots

- `default` gets tiered up at the 2^{25}th iteration. (threshold: 30)
- `interp-tc` and `interp-tc-without-loops` get tiered up at the 2^{18}th and 2^{19}th iterations, respectively. (thresholds: 0)
- **TIL:** The **actual**, **asynchronous** tiering-up occurs much later after it is queued in the background.
- **Question:** Would **synchronous** tiering-up have better overall performance if Tier 0 is the interpreter? What about a combination of synchronous and asynchronous tiering-up?
- **Question:** Background requests go into a queue, building up a big backlog. Can this be turned into a priority queue? What criteria to determine the priority?

TIL from Beautiful Plots

- `default` gets tiered up at the 2^{25}th iteration. (threshold: 30)
- `interp-tc` and `interp-tc-without-loops` get tiered up at the 2^{18}th and 2^{19}th iterations, respectively. (thresholds: 0)
- **TIL:** The **actual**, **asynchronous** tiering-up occurs much later after it is queued in the background.
- **Question:** Would **synchronous** tiering-up have better overall performance if Tier 0 is the interpreter? What about a combination of synchronous and asynchronous tiering-up?
- **Question:** Background requests go into a queue, building up a big backlog. Can this be turned into a priority queue? What criteria to determine the priority?
- A "turning point" at the 2^{18}th iteration on `interp-tc`.

TIL from Beautiful Plots

- `default` gets tiered up at the 2^{25}th iteration. (threshold: 30)
- `interp-tc` and `interp-tc-without-loops` get tiered up at the 2^{18}th and 2^{19}th iterations, respectively. (thresholds: 0)
- **TIL:** The **actual**, **asynchronous** tiering-up occurs much later after it is queued in the background.
- **Question:** Would **synchronous** tiering-up have better overall performance if Tier 0 is the interpreter? What about a combination of synchronous and asynchronous tiering-up?
- **Question:** Background requests go into a queue, building up a big backlog. Can this be turned into a priority queue? What criteria to determine the priority?
- A "turning point" at the 2^{18}th iteration on `interp-tc`.
- **TIL:** `update` gets tiered up. However, the total execution time is dominated by the interpretation time of the loop, which cannot be tiered up.

TIL from Beautiful Plots

- `default` gets tiered up at the 2^{25}th iteration. (threshold: 30)
- `interp-tc` and `interp-tc-without-loops` get tiered up at the 2^{18}th and 2^{19}th iterations, respectively. (thresholds: 0)
- **TIL:** The **actual**, **asynchronous** tiering-up occurs much later after it is queued in the background.
- **Question:** Would **synchronous** tiering-up have better overall performance if Tier 0 is the interpreter? What about a combination of synchronous and asynchronous tiering-up?
- **Question:** Background requests go into a queue, building up a big backlog. Can this be turned into a priority queue? What criteria to determine the priority?
- A "turning point" at the 2^{18}th iteration on `interp-tc`.
- **TIL:** `update` gets tiered up. However, the total execution time is dominated by the interpretation time of the loop, which cannot be tiered up.

TIL from Beautiful Plots

TIL: On Stack Replacement

TIL: On Stack Replacement

- **Tiered Compilation:** If a method is invocated frequently, optimize it.

TIL: On Stack Replacement

- **Tiered Compilation:** If a method is invocated frequently, optimize it.

```
public static int execute(int num_iterations)
{
    int i;
    int a = 0, b = 1;

    for (i = 0; i < num_iterations; i++)
    {
        update(ref a, ref b);
    }

    return a;
}
```

TIL: On Stack Replacement

- **Tiered Compilation:** If a method is invocated frequently, optimize it.

```
public static int execute(int num_iterations)
{
    int i;
    int a = 0, b = 1;

    for (i = 0; i < num_iterations; i++)
    {
        update(ref a, ref b);
    }

    return a;
}
```

- If a method is invocated very few times but its execution takes long time, we can never optimize it with Tiered Compilation **only**.

TIL: On Stack Replacement

- **Tiered Compilation:** If a method is invocated frequently, optimize it.

```
public static int execute(int num_iterations)
{
    int i;
    int a = 0, b = 1;

    for (i = 0; i < num_iterations; i++)
    {
        update(ref a, ref b);
    }

    return a;
}
```

- If a method is invocated very few times but its execution takes long time, we can never optimize it with Tiered Compilation **only**.
- `COMPlus_InterpreterDoLoopMethods=0` to avoid interpreting methods with loops.

TIL: On Stack Replacement

- **On Stack Replacement (OSR):** If some part of a method is called frequently, optimize that part.

TIL: On Stack Replacement

- **On Stack Replacement (OSR):** If some part of a method is called frequently, optimize that part.
- Formally, OSR allows the code executed by currently running methods to be changed in the middle of method execution, while those methods are active "on stack."

TIL: On Stack Replacement

- **On Stack Replacement (OSR):** If some part of a method is called frequently, optimize that part.
- Formally, OSR allows the code executed by currently running methods to be changed in the middle of method execution, while those methods are active "on stack."
- With OSR, we can safely start with interpreting a method with a loop. If the loop body is executed frequently, OSR allows the loop to be optimized.

TIL: On Stack Replacement

- **On Stack Replacement (OSR):** If some part of a method is called frequently, optimize that part.
- Formally, OSR allows the code executed by currently running methods to be changed in the middle of method execution, while those methods are active "on stack."
- With OSR, we can safely start with interpreting a method with a loop. If the loop body is executed frequently, OSR allows the loop to be optimized.
- A **patchpoint** is a particular location in a method that supports OSR transitions.

TIL: On Stack Replacement

- **On Stack Replacement (OSR):** If some part of a method is called frequently, optimize that part.
- Formally, OSR allows the code executed by currently running methods to be changed in the middle of method execution, while those methods are active "on stack."
- With OSR, we can safely start with interpreting a method with a loop. If the loop body is executed frequently, OSR allows the loop to be optimized.
- A **patchpoint** is a particular location in a method that supports OSR transitions.
- At a patchpoint, the **live state** of the ongoing computation must be identifiable (e.g. registers, stack slots, return address, implicit arguments).

TIL: On Stack Replacement

- Commonly, targets of **lexical** loop back edges or backward branches are chosen to be patchpoints.

TIL: On Stack Replacement

- Commonly, targets of **lexical** loop back edges or backward branches are chosen to be patchpoints.
- Identified by a single scan of the IL code of a method

TIL: On Stack Replacement

- Commonly, targets of **lexical** loop back edges or backward branches are chosen to be patchpoints.
- Identified by a single scan of the IL code of a method
- **Partially interruptible**

TIL: On Stack Replacement

- Commonly, targets of **lexical** loop back edges or backward branches are chosen to be patchpoints.
- Identified by a single scan of the IL code of a method
- **Partially interruptible**
- With such choices, a method cannot execute indefinitely between patchpoints.

TIL: On Stack Replacement

- Commonly, targets of **lexical** loop back edges or backward branches are chosen to be patchpoints.
- Identified by a single scan of the IL code of a method
- **Partially interruptible**
- With such choices, a method cannot execute indefinitely between patchpoints.
- Loops that execute with **non-empty evaluation stacks** are likely rare, hence minimizing and regularizing the live state.

TIL: On Stack Replacement

- How does OSR work if the interpreter is used as Tier 0?

TIL: On Stack Replacement

- How does OSR work if the interpreter is used as Tier 0?
- **Challenge:** The current approach of capturing the live state is limited to JIT'ed code. Local variables and parameters are stored in a custom structure in the interpreter.

TIL: On Stack Replacement

- How does OSR work if the interpreter is used as Tier 0?
- **Challenge:** The current approach of capturing the live state is limited to JIT'ed code. Local variables and parameters are stored in a custom structure in the interpreter.
- **Option #1:** Pointers to the piece of memory that contains the live state

TIL: On Stack Replacement

- How does OSR work if the interpreter is used as Tier 0?
- **Challenge:** The current approach of capturing the live state is limited to JIT'ed code. Local variables and parameters are stored in a custom structure in the interpreter.
- **Option #1:** Pointers to the piece of memory that contains the live state
- **Option #2:** Copy the live state and create a new patchpoint information structure

TIL: On Stack Replacement

- How does OSR work if the interpreter is used as Tier 0?
- **Challenge:** The current approach of capturing the live state is limited to JIT'ed code. Local variables and parameters are stored in a custom structure in the interpreter.
- **Option #1:** Pointers to the piece of memory that contains the live state
- **Option #2:** Copy the live state and create a new patchpoint information structure
- QuickJIT currently uses **Option #1**.

TIL: On Stack Replacement

- How does OSR work if the interpreter is used as Tier 0?
- **Challenge:** The current approach of capturing the live state is limited to JIT'ed code. Local variables and parameters are stored in a custom structure in the interpreter.
- **Option #1:** Pointers to the piece of memory that contains the live state
- **Option #2:** Copy the live state and create a new patchpoint information structure
- QuickJIT currently uses **Option #1**.
- In the interpreter, the local variables and parameters are stored in a custom structure.

TIL: On Stack Replacement

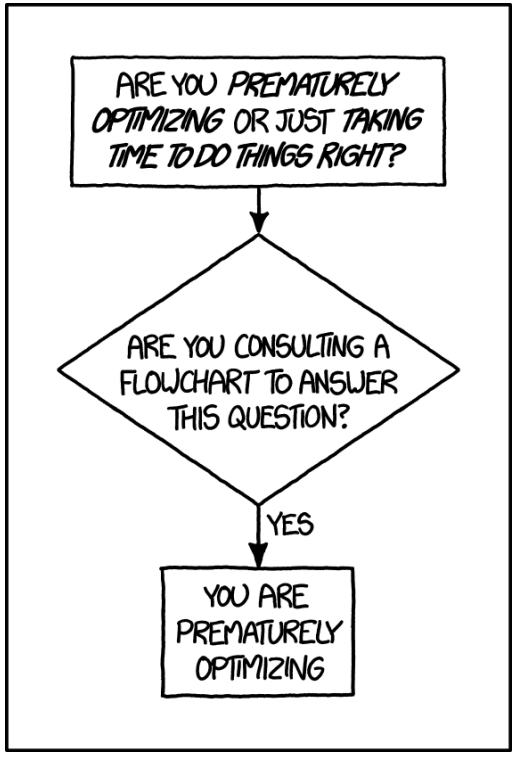
- How does OSR work if the interpreter is used as Tier 0?
- **Challenge:** The current approach of capturing the live state is limited to JIT'ed code. Local variables and parameters are stored in a custom structure in the interpreter.
- **Option #1:** Pointers to the piece of memory that contains the live state
- **Option #2:** Copy the live state and create a new patchpoint information structure
- QuickJIT currently uses **Option #1**.
- In the interpreter, the local variables and parameters are stored in a custom structure.
- Optimized jitted code needs to access these pieces of memory.

TIL: On Stack Replacement

- How does OSR work if the interpreter is used as Tier 0?
- **Challenge:** The current approach of capturing the live state is limited to JIT'ed code. Local variables and parameters are stored in a custom structure in the interpreter.
- **Option #1:** Pointers to the piece of memory that contains the live state
- **Option #2:** Copy the live state and create a new patchpoint information structure
- QuickJIT currently uses **Option #1**.
- In the interpreter, the local variables and parameters are stored in a custom structure.
- Optimized jitted code needs to access these pieces of memory.
- **Option #2** might be easier to implement without updates to the original pieces of memory.

Where Are We?

- Why Is the Interpreter So Interesting?
- ... and Fixes, and Fixes, and Fixes, ...
- TIL from Beautiful Plots
- Oh My! Bottlenecks!
- Where Do We Go from Here?



TIL: Oh My! Bottlenecks!

TIL: Oh My! Bottlenecks!

- **Question:** What makes the interpreter slow?

TIL: Oh My! Bottlenecks!

- **Question:** What makes the interpreter slow?
- Performance analysis on the ETW profiling data with PerfView and speedscope

TIL: Oh My! Bottlenecks!

- **Question:** What makes the interpreter slow?
- Performance analysis on the ETW profiling data with PerfView and speedscope
- A very preliminary analysis

TIL: Oh My! Bottlenecks!

- **Question:** What makes the interpreter slow?
- Performance analysis on the ETW profiling data with PerfView and speedscope
- A very preliminary analysis
- **inclusive time:** the execution time including its callees

TIL: Oh My! Bottlenecks!

- **Question:** What makes the interpreter slow?
- Performance analysis on the ETW profiling data with PerfView and speedscope
- A very preliminary analysis
- **inclusive time:** the execution time including its callees
- **exclusive time:** the execution time excluding its callees

TIL: Oh My! Bottlenecks!

- **Question:** What makes the interpreter slow?
- Performance analysis on the ETW profiling data with PerfView and speedscope
- A very preliminary analysis
- **inclusive time:** the execution time including its callees
- **exclusive time:** the execution time excluding its callees
- Two approaches:

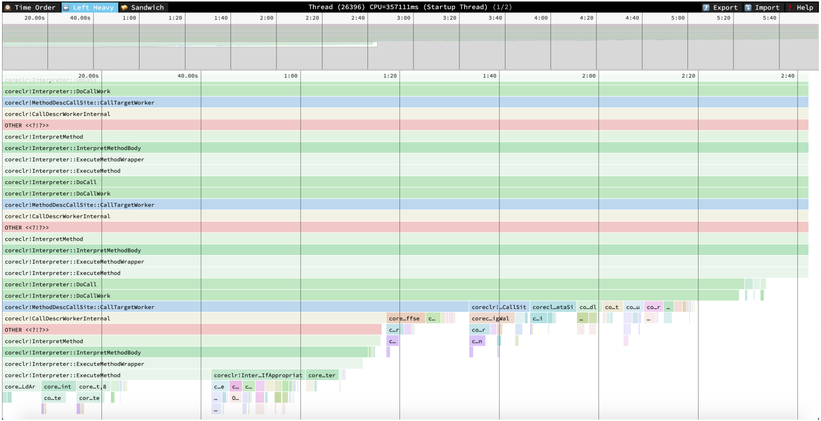
TIL: Oh My! Bottlenecks!

- **Question:** What makes the interpreter slow?
- Performance analysis on the ETW profiling data with PerfView and speedscope
- A very preliminary analysis
- **inclusive time:** the execution time including its callees
- **exclusive time:** the execution time excluding its callees
- Two approaches:
- **Bottom-up:** Start with functions of high exclusive time, and analyze its callers.

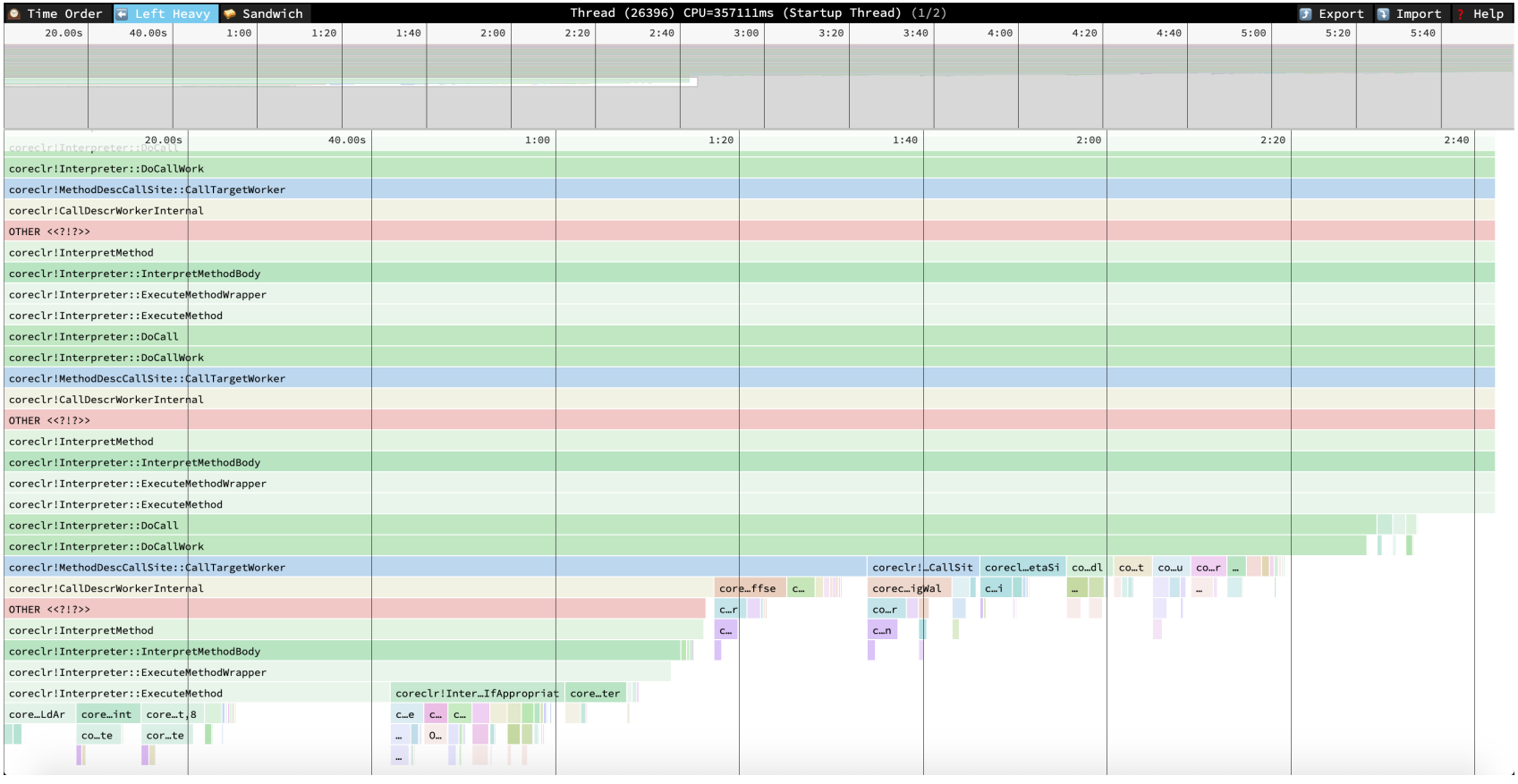
TIL: Oh My! Bottlenecks!

- **Question:** What makes the interpreter slow?
- Performance analysis on the ETW profiling data with PerfView and speedscope
- A very preliminary analysis
- **inclusive time:** the execution time including its callees
- **exclusive time:** the execution time excluding its callees
- Two approaches:
- **Bottom-up:** Start with functions of high exclusive time, and analyze its callers.
- **Top-down:** Start from functions of high inclusive time, and analyze its callees to see where the interpreter spends time

TIL: Oh My! Bottlenecks!

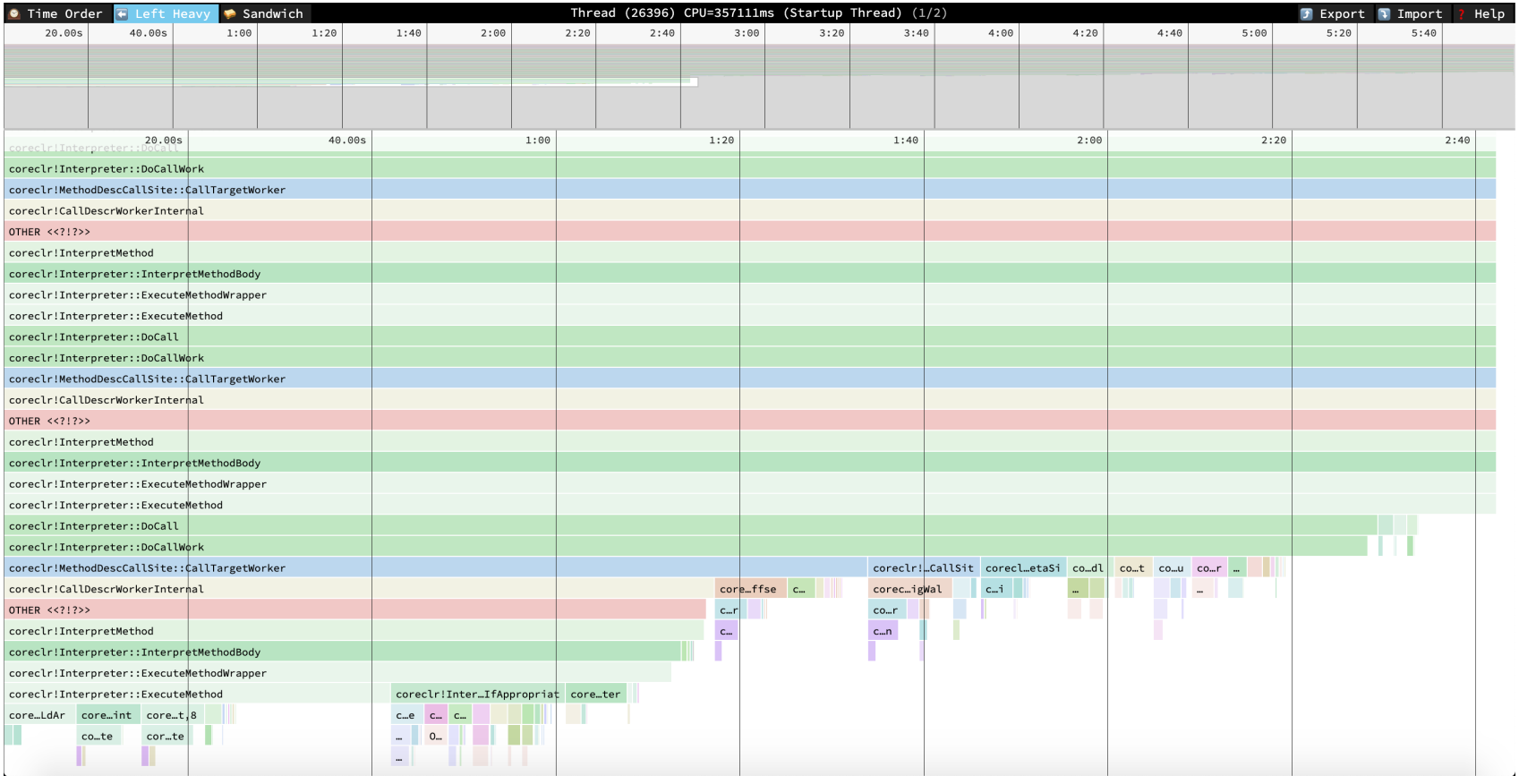


TIL: Oh My! Bottlenecks!



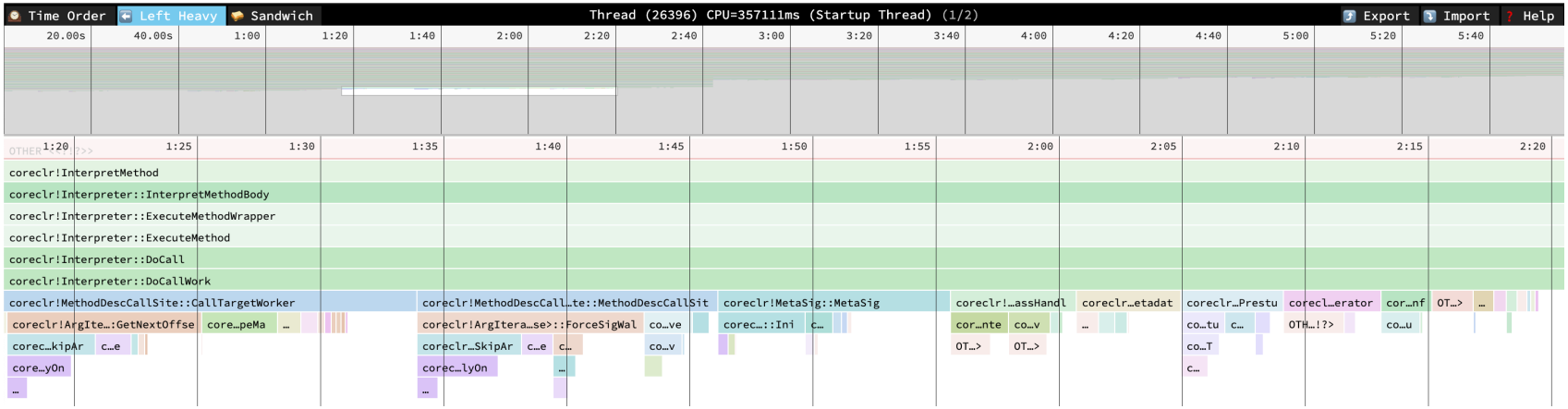
- The pattern of the call stack suggests that `DoCall` and `DoCallWork` take a large portion of inclusive time.

TIL: Oh My! Bottlenecks!

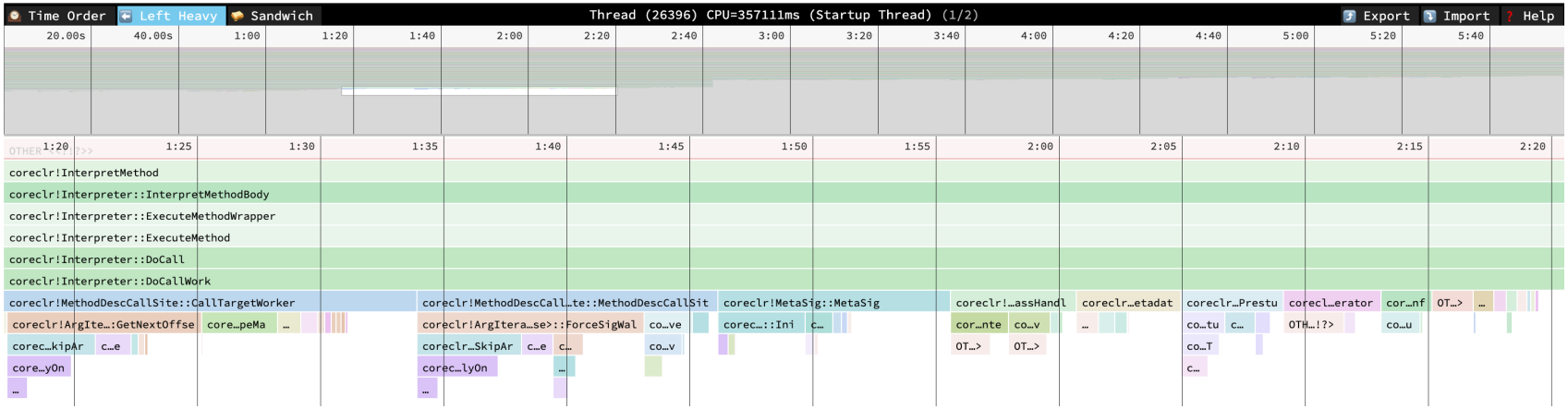


- The pattern of the call stack suggests that `DoCall` and `DoCallWork` take a large portion of inclusive time.
- **Question:** Which callees (direct or indirect) of `DoCallWork` have large inclusive and exclusive time?

TIL: Oh My! Bottlenecks!



TIL: Oh My! Bottlenecks!



Name	Exc %	Exc
coreclr!Interpreter::ExecuteMethod	16.4	26,605
OTHER <<ntdll!?>>	7.2	11,658
coreclr!Interpreter::ThrowOnInvalidPointer	6.6	10,724
coreclr!Interpreter::DoCallWork	6.4	10,339
coreclr!Interpreter::LdArg	5.0	8,157
coreclr!MetaSig::MetaSig	4.0	6,492
coreclr!SigParser::SkipExactlyOne	4.0	6,420
coreclr!Interpreter::JitMethodIfAppropriate	3.4	5,569
coreclr!MethodDescCallSite::CallTargetWorker	3.4	5,546
coreclr!Interpreter::ExecuteMethodWrapper	3.0	4,790
coreclr!Interpreter::Interpreter	2.8	4,489
coreclr!ArgIteratorTemplate<ArgIteratorBase>::ForceSigWalk	2.3	3,713
coreclr!CEEInfo::getMethodNameFromMetadata	2.1	3,414
coreclr!Interpreter::BrOnComparison<3,0,1>	2.0	3,172
coreclr!MetaSig::Init	1.9	3,161
coreclr!FillInRegTypeMap	1.9	3,110
coreclr!ArgIteratorTemplate<ArgIteratorBase>::GetNextOffset	1.8	2,888
coreclr!SigPointer::PeekElemTypeNormalized	1.7	2,773
coreclr!ILCodeVersion::GetActiveNativeCodeVersion	1.6	2,620
coreclr!InterpretMethod	1.6	2,565
coreclr!InterpreterType::IsLargeStruct	1.5	2,477
coreclr!MethodDesc::IsPointingToPrestub	1.5	2,396
coreclr!Interpreter::StInd<int>	1.3	2,118
coreclr!MapSHash<CORINFO_METHOD_STRUCT_*,Interpreter::MethInfo,NoRemoveSHashTraits<MapSHashTraits<CORINFO_METHOD_STRUCT_*,Interpreter::MethInfo> > >::Lookup	1.3	2,104
coreclr!PEDecoder::CheckILMethod	1.3	2,080
coreclr!MethodDesc::EnsureActive	1.2	1,996
coreclr!MetaSig::SkipArg	1.2	1,936
coreclr!COR_ILMETHOD_DECODER::COR_ILMETHOD_DECODER	1.1	1,812
coreclr!Precode::IsPointingToPrestub	1.1	1,806
coreclr!CEEJitInfo::CEEJitInfo	1.1	1,719
OTHER <<ucrtbase!?>>	1.1	1,707

TIL: Oh My! Bottlenecks!

- **TIL:** The flamegraph and the exclusive time data suggest that the interpretation of function calls spends much time on parsing the signature and the method descriptor of the callee.

TIL: Oh My! Bottlenecks!

- **TIL:** The flamegraph and the exclusive time data suggest that the interpretation of function calls spends much time on parsing the signature and the method descriptor of the callee.
- e.g. `MetaSig::MegaSig` (3.7%), `SigParse::SkipExactlyOnce` (3.6%), `ArgIteratorTemplate::ForceSigWalk` (2.2%) `MegaSig::Init` (1.8%), `FillInRegTypeMap` (1.8%) etc.

TIL: Oh My! Bottlenecks!

- **TIL:** The flamegraph and the exclusive time data suggest that the interpretation of function calls spends much time on parsing the signature and the method descriptor of the callee.
- e.g. `MetaSig::MegaSig` (3.7%), `SigParse::SkipExactlyOnce` (3.6%), `ArgIteratorTemplate::ForceSigWalk` (2.2%) `MegaSig::Init` (1.8%), `FillInRegTypeMap` (1.8%) etc.
- The source code of `DoCallWork` also suggests the parsing of the callee signature.

TIL: Oh My! Bottlenecks!

- **TIL:** The flamegraph and the exclusive time data suggest that the interpretation of function calls spends much time on parsing the signature and the method descriptor of the callee.
- e.g. `MetaSig::MegaSig` (3.7%), `SigParse::SkipExactlyOnce` (3.6%), `ArgIteratorTemplate::ForceSigWalk` (2.2%) `MegaSig::Init` (1.8%), `FillInRegTypeMap` (1.8%) etc.
- The source code of `DoCallWork` also suggests the parsing of the callee signature.
- `ForceSigWalk` handles calling conventions for callees, and enumerates each argument of a signature in turn.

TIL: Oh My! Bottlenecks!

- **TIL:** The flamegraph and the exclusive time data suggest that the interpretation of function calls spends much time on parsing the signature and the method descriptor of the callee.
- e.g. `MetaSig::MegaSig` (3.7%), `SigParse::SkipExactlyOnce` (3.6%), `ArgIteratorTemplate::ForceSigWalk` (2.2%) `MegaSig::Init` (1.8%), `FillInRegTypeMap` (1.8%) etc.
- The source code of `DoCallWork` also suggests the parsing of the callee signature.
- `ForceSigWalk` handles calling conventions for callees, and enumerates each argument of a signature in turn.
- **Question:** Can this information of mapping arguments to registers be cached somehow? If we repeatedly call a method, can we let the arguments stay in the registers?

TIL: Oh My! Bottlenecks!

- **TIL:** `DoCallWork` also has large exclusive time of 6.4%.

TIL: Oh My! Bottlenecks!

- **TIL:** `DoCallWork` also has large exclusive time of 6.4%.
- An inspection into the implementation of `DoCallWork` suggests that it re-processes arguments every time a function is called.

TIL: Oh My! Bottlenecks!

- **TIL:** `DoCallWork` also has large exclusive time of 6.4%.
- An inspection into the implementation of `DoCallWork` suggests that it re-processes arguments every time a function is called.
- **Question:** Can this information also be cached somehow?

TIL: Oh My! Bottlenecks!

```
// CYCLE PROFILE: BEFORE ARG PROCESSING.

if (sigInfo.hasThis())
{
    if (m_callThisArg != NULL)
    {
        if (size_t(m_callThisArg) == 0x1)
        {
            args[curArgSlot] = NULL;
        }
        else
        {
            args[curArgSlot] = PtrToArgSlot(m_callThisArg);
        }
        argTypes[curArgSlot] = InterpreterType(CORINFO_TYPE_BYREF);
    }
    else
    {
        args[curArgSlot] = PtrToArgSlot(OpStackGet<void*>(argsBase + arg));
        argTypes[curArgSlot] = OpStackTypeGet(argsBase + arg);
        arg++;
    }
    // AV -> NullRef translation is NYI for the interpreter,
    // so we should manually check and throw the correct exception.
    if (args[curArgSlot] == NULL)
    {
        // If we're calling a constructor, we bypass this check since the runtime
        // should have thrown OOM if it was unable to allocate an instance.
        if (m_callThisArg == NULL)
        {
            _ASSERTE(!methToCall->IsStatic());
            ThrowNullPointerException();
        }
        // ...except in the case of strings, which are both
        // allocated and initialized by their special constructor.
        else
        {
            _ASSERTE(methToCall->IsCtor() && methToCall->GetMethodTable()->IsString());
        }
    }
    curArgSlot++;
}
```

TIL: Oh My! Bottlenecks!

- We use the names from the metadata to recognize whether a called method is a hardware intrinsic or SIMD method.

TIL: Oh My! Bottlenecks!

- We use the names from the metadata to recognize whether a called method is a hardware intrinsic or SIMD method.

```
            // Hardware intrinsics are recognized by name.
            const char* namespaceName = NULL;
            const char* className = NULL;
            const char* methodName = m_interpCeeInfo.getMethodNameFromMetadata((CORINFO_METHOD_HANDLE)methToCall, &className, &namespaceName, NULL);
            if (
#if defined(TARGET_X86) || defined(TARGET_AMD64)
                strcmp(namespaceName, "System.Runtime.Intrinsics.X86") == 0 &&
#elif defined(TARGET_ARM64)
                strcmp(namespaceName, "System.Runtime.Intrinsics.Arm") == 0 &&
#endif // defined(TARGET_X86) || defined(TARGET_AMD64)
                strcmp(methodName, "get_IsSupported") == 0
            )
            {
                GCX_COOP();
                DoGetIsSupported();
                didIntrinsic = true;
            }
```

TIL: Oh My! Bottlenecks!

- We use the names from the metadata to recognize whether a called method is a hardware intrinsic or SIMD method.

```
            // Hardware intrinsics are recognized by name.
            const char* namespaceName = NULL;
            const char* className = NULL;
            const char* methodName = m_interpCeeInfo.getMethodNameFromMetadata((CORINFO_METHOD_HANDLE)methToCall, &className, &namespaceName, NULL);
            if (
#if defined(TARGET_X86) || defined(TARGET_AMD64)
                strcmp(namespaceName, "System.Runtime.Intrinsics.X86") == 0 &&
#elif defined(TARGET_ARM64)
                strcmp(namespaceName, "System.Runtime.Intrinsics.Arm") == 0 &&
#endif // defined(TARGET_X86) || defined(TARGET_AMD64)
                strcmp(methodName, "get_IsSupported") == 0
            )
            {
                GCX_COOP();
                DoGetIsSupported();
                didIntrinsic = true;
            }
```

- **TIL:** Obtaining names from metadata is **expensive**.

TIL: Oh My! Bottlenecks!

- We use the names from the metadata to recognize whether a called method is a hardware intrinsic or SIMD method.

```
            // Hardware intrinsics are recognized by name.
            const char* namespaceName = NULL;
            const char* className = NULL;
            const char* methodName = m_interpCeeInfo.getMethodNameFromMetadata((CORINFO_METHOD_HANDLE)methToCall, &className, &namespaceName, NULL);
            if (
#if defined(TARGET_X86) || defined(TARGET_AMD64)
                strcmp(namespaceName, "System.Runtime.Intrinsics.X86") == 0 &&
#elif defined(TARGET_ARM64)
                strcmp(namespaceName, "System.Runtime.Intrinsics.Arm") == 0 &&
#endif // defined(TARGET_X86) || defined(TARGET_AMD64)
                strcmp(methodName, "get_IsSupported") == 0
            )
            {
                GCX_COOP();
                DoGetIsSupported();
                didIntrinsic = true;
            }
```

- **TIL:** Obtaining names from metadata is **expensive**.
- `CEEInfo::getMethodNameFromMetadata` has exclusive time percentage of 2.0%.

TIL: Oh My! Bottlenecks!

- We use the names from the metadata to recognize whether a called method is a hardware intrinsic or SIMD method.

```
            // Hardware intrinsics are recognized by name.
            const char* namespaceName = NULL;
            const char* className = NULL;
            const char* methodName = m_interpCeeInfo.getMethodNameFromMetadata((CORINFO_METHOD_HANDLE)methToCall, &className, &namespaceName, NULL);
            if (
#if defined(TARGET_X86) || defined(TARGET_AMD64)
                strcmp(namespaceName, "System.Runtime.Intrinsics.X86") == 0 &&
#elif defined(TARGET_ARM64)
                strcmp(namespaceName, "System.Runtime.Intrinsics.Arm") == 0 &&
#endif // defined(TARGET_X86) || defined(TARGET_AMD64)
                strcmp(methodName, "get_IsSupported") == 0
            )
            {
                GCX_COOP();
                DoGetIsSupported();
                didIntrinsic = true;
            }
```

- **TIL:** Obtaining names from metadata is **expensive**.
- `CEEInfo::getMethodNameFromMetadata` has exclusive time percentage of 2.0%.
- **Proposal #1:** Call `getMethodNameFromMetadata` only once for all special cases.

TIL: Oh My! Bottlenecks!

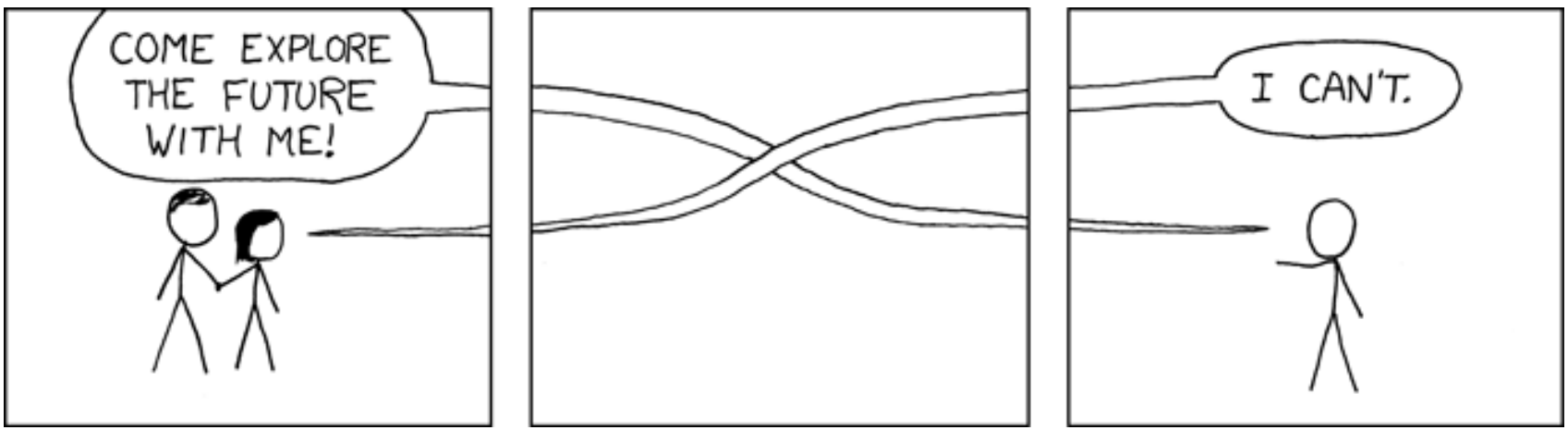
- We use the names from the metadata to recognize whether a called method is a hardware intrinsic or SIMD method.

```
            // Hardware intrinsics are recognized by name.
            const char* namespaceName = NULL;
            const char* className = NULL;
            const char* methodName = m_interpCeeInfo.getMethodNameFromMetadata((CORINFO_METHOD_HANDLE)methToCall, &className, &namespaceName, NULL);
            if (
#if defined(TARGET_X86) || defined(TARGET_AMD64)
                strcmp(namespaceName, "System.Runtime.Intrinsics.X86") == 0 &&
#elif defined(TARGET_ARM64)
                strcmp(namespaceName, "System.Runtime.Intrinsics.Arm") == 0 &&
#endif // defined(TARGET_X86) || defined(TARGET_AMD64)
                strcmp(methodName, "get_IsSupported") == 0
            )
            {
                GCX_COOP();
                DoGetIsSupported();
                didIntrinsic = true;
            }
```

- **TIL:** Obtaining names from metadata is **expensive**.
- `CEEInfo::getMethodNameFromMetadata` has exclusive time percentage of 2.0%.
- **Proposal #1:** Call `getMethodNameFromMetadata` only once for all special cases.
- **Proposal #2:** Cache the special casing information somewhere (e.g. `CORINFO_INTRINSIC_HWINTRINSIC`).

Where Are We?

- Why Is the Interpreter So Interesting?
- ... and Fixes, and Fixes, and Fixes, ...
- TIL from Beautiful Plots
- Oh My! Bottlenecks!
- Where Do We Go from Here?



TIL: Where Do We Go From Here?

TIL: Where Do We Go From Here?

- There are still many **functionality gaps** from the current state of the interpreter to a shippable shape.

TIL: Where Do We Go From Here?

- There are still many **functionality gaps** from the current state of the interpreter to a shippable shape.
- **Hardware intrinsics**

TIL: Where Do We Go From Here?

- There are still many **functionality gaps** from the current state of the interpreter to a shippable shape.
- **Hardware intrinsics**
- **ASP.NET:** Enables the performance measurements (throughput and start-up) of a web application as in the Tiered Compilation experiments

TIL: Where Do We Go From Here?

- Several broad, possibly interesting **performance** questions:

TIL: Where Do We Go From Here?

- Several broad, possibly interesting **performance** questions:
- Background tiering-up requests go into a queue, possibly building up a huge backlog with calls to various functions. Can we turn this into a **priority queue**? What determines the priority of an instance of tiering-up?

TIL: Where Do We Go From Here?

- Several broad, possibly interesting **performance** questions:
- Background tiering-up requests go into a queue, possibly building up a huge backlog with calls to various functions. Can we turn this into a **priority queue**? What determines the priority of an instance of tiering-up?
- **3-Level** Tiered Compilation (Interpreter-QuickJIT-Tier1)?

TIL: Where Do We Go From Here?

- Several broad, possibly interesting **performance** questions:
- Background tiering-up requests go into a queue, possibly building up a huge backlog with calls to various functions. Can we turn this into a **priority queue**? What determines the priority of an instance of tiering-up?
- **3-Level** Tiered Compilation (Interpreter-QuickJIT-Tier1)?
- **Synchronous** tiering-up?

TIL: Where Do We Go From Here?

- Several broad, possibly interesting **performance** questions:
- Background tiering-up requests go into a queue, possibly building up a huge backlog with calls to various functions. Can we turn this into a **priority queue**? What determines the priority of an instance of tiering-up?
- **3-Level** Tiered Compilation (Interpreter-QuickJIT-Tier1)?
- **Synchronous** tiering-up?
- **On Stack Replacement**?

TIL: This Internship Is Great! ♡

TIL: This Internship Is Great! ♡

Questions & Answers

Appendix

- **Issue #1:** The C++ compiler treats `reinterpret_cast` from a pointer to an integral type as an error.

Appendix

- **Issue #1:** The C++ compiler treats `reinterpret_cast` from a pointer to an integral type as an error.
- **Fix:**

```
reinterpret_cast<short>(ArgSlotEndianessFixup(directOffset, sizeof(void*)));
static_cast<short>(reinterpret_cast<intptr_t>(ArgSlotEndianessFixup(directOffset, sizeof(void*))));
```

Appendix

- **Issue #2:** The C++ compiler treats the conversion from a generic type `T` to an integral type as an error.

Appendix

- **Issue #2:** The C++ compiler treats the conversion from a generic type `T` to an integral type as an error.
- **Fix:** There are finitely many integral types as `CORINFO_TYPE`. Hence we can condition on `sizeof(T)` to determine which integral type to which `T` is equivalent.

```
// Widen narrow types.
int ires;
switch (sizeof(T))
{
case 1:
    ires = std::is_same<T, INT8>::value ?
           static_cast<int>(reinterpret_cast<INT8*>(res_ptr)[index]) :
           static_cast<int>(reinterpret_cast<UINT8*>(res_ptr)[index]);
    break;
case 2:
    ires = std::is_same<T, INT16>::value ?
           static_cast<int>(reinterpret_cast<INT16*>(res_ptr)[index]) :
           static_cast<int>(reinterpret_cast<UINT16*>(res_ptr)[index]);
    break;
case 4:
    ires = std::is_same<T, INT32>::value ?
           static_cast<int>(reinterpret_cast<INT32*>(res_ptr)[index]) :
           static_cast<int>(reinterpret_cast<UINT32*>(res_ptr)[index]);
    break;
default:
    _ASSERTE_MSG(false, "This should have exhausted all the possible sizes.");
    break;
}
```

Appendix

- **Issue #3:** The interpreter does not check the optimization tier before it triggers the **asynchronous** tiering-up. The tiering-up might be triggered again before the optimization tier is **actually** promoted.

Appendix

- **Issue #3:** The interpreter does not check the optimization tier before it triggers the **asynchronous** tiering-up. The tiering-up might be triggered again before the optimization tier is **actually** promoted.
- **Issue #4:** `LockHolder` is missing for synchronization.

Appendix

- **Issue #3:** The interpreter does not check the optimization tier before it triggers the **asynchronous** tiering-up. The tiering-up might be triggered again before the optimization tier is **actually** promoted.
- **Issue #4:** `LockHolder` is missing for synchronization.
- **Fix:**

```
#ifdef FEATURE_TIERED_COMPILATION
            CodeVersionManager::LockHolder _lockHolder;
            NativeCodeVersion activeCodeVersion = md->GetCodeVersionManager()->GetActiveILCodeVersion(md).GetActiveNativeCodeVersion(md);
            ILCodeVersion ilCodeVersion = activeCodeVersion.GetILCodeVersion();
            if (activeCodeVersion.GetOptimizationTier() == NativeCodeVersion::OptimizationTier0 &&
                !ilCodeVersion.HasAnyOptimizedNativeCodeVersion(activeCodeVersion))
            {
                tieredCompilationManager->AsyncPromoteToTier1(activeCodeVersion, &scheduleTieringBackgroundWork);
            }
#else
#error FEATURE_INTERPRETER depends on FEATURE_TIERED_COMPILATION now
#endif
```

Appendix

- **Issue #5:** "Bad IL Format" for virtual calls.
- **Fix:** Enable the `CORINFO_CALLINFO_CALLVIRT` flag in the call info for virtual calls.

```
combine(CORINFO_CALLINFO_SECURITYCHECKS,CORINFO_CALLINFO_LDFTN),
&callInfo);
  combine(CORINFO_CALLINFO_CALLVIRT,
          combine(CORINFO_CALLINFO_SECURITYCHECKS,
                  CORINFO_CALLINFO_LDFTN)),
  &callInfo);
```

Appendix

- **Issue #6:** The IL implementations of `get_IsSupported()` methods for hardware intrinsics are non-terminating recursions and will be handled by jitted assembly code. The interpreter is unaware of this special case.

Appendix

- **Issue #6:** The IL implementations of `get_IsSupported()` methods for hardware intrinsics are non-terminating recursions and will be handled by jitted assembly code. The interpreter is unaware of this special case.

```
.method public hidebysig specialname static
        bool  get_IsSupported() cil managed
{
  // Code size       6 (0x6)
  .maxstack  8
  IL_0000:  call       bool System.Runtime.Intrinsics.X86.Sse2::get_IsSupported()
  IL_0005:  ret
} // end of method Sse2::get_IsSupported
```

Appendix

- **Issue #6:** The IL implementations of `get_IsSupported()` methods for hardware intrinsics are non-terminating recursions and will be handled by jitted assembly code. The interpreter is unaware of this special case.

```
.method public hidebysig specialname static
        bool  get_IsSupported() cil managed
{
  // Code size       6 (0x6)
  .maxstack  8
  IL_0000:  call       bool System.Runtime.Intrinsics.X86.Sse2::get_IsSupported()
  IL_0005:  ret
} // end of method Sse2::get_IsSupported
```

- **“Fix” (a.k.a. Workaround):** Force the interpreter to return `false` for `get_IsSupported()` to disable hardware intrinsics.

Appendix

- **Issue #6:** The IL implementations of `get_IsSupported()` methods for hardware intrinsics are non-terminating recursions and will be handled by jitted assembly code. The interpreter is unaware of this special case.

```
.method public hidebysig specialname static
        bool  get_IsSupported() cil managed
{
  // Code size       6 (0x6)
  .maxstack  8
  IL_0000:  call       bool System.Runtime.Intrinsics.X86.Sse2::get_IsSupported()
  IL_0005:  ret
} // end of method Sse2::get_IsSupported
```

- **"Fix" (a.k.a. Workaround):** Force the interpreter to return `false` for `get_IsSupported()` to disable hardware intrinsics.
- Controlled by the COM+ environment variable `InterpreterHWIntrinsicsIsSupportedFalse`

Appendix

- **Issue of "Fix":** Arbitrary combinations of different code generators (JIT, CrossGen, interpreter, etc.) in the same process might cause **inconsistent** values of `IsSupported`. (@jkotas)

Appendix

- **Issue of "Fix":** Arbitrary combinations of different code generators (JIT, CrossGen, interpreter, etc.) in the same process might cause **inconsistent** values of `IsSupported`. (@jkotas)

```
readonly static RenderingEngine s_engine = Avx.IsSupported ? new AvxRenderingEngine() : new GenericRenderingEngine();

class GenericRenderingEngine : RenderingEngine
{
    virtual void Render()
    {
        Debug.Assert(!Avx.IsSupported);
        // ...
    }

    // ...
}
```

Appendix

- **Issue of "Fix":** Arbitrary combinations of different code generators (JIT, CrossGen, interpreter, etc.) in the same process might cause **inconsistent** values of `IsSupported`. (@jkotas)

```
readonly static RenderingEngine s_engine = Avx.IsSupported ? new AvxRenderingEngine() : new GenericRenderingEngine();

class GenericRenderingEngine : RenderingEngine
{
    virtual void Render()
    {
        Debug.Assert(!Avx.IsSupported);
        // ...
    }

    // ...
}
```

- **Fix #1:** Depend on Fallback to JIT for hardware intrinsics methods and let JIT generate valid method bodies.

Appendix

- **Issue of "Fix":** Arbitrary combinations of different code generators (JIT, CrossGen, interpreter, etc.) in the same process might cause **inconsistent** values of `IsSupported`. (@jkotas)

```
readonly static RenderingEngine s_engine = Avx.IsSupported ? new AvxRenderingEngine() : new GenericRenderingEngine();

class GenericRenderingEngine : RenderingEngine
{
    virtual void Render()
    {
        Debug.Assert(!Avx.IsSupported);
        // ...
    }

    // ...
}
```

- **Fix #1:** Depend on Fallback to JIT for hardware intrinsics methods and let JIT generate valid method bodies.
- **Fix #2:** Implement hardware intrinsics methods in the interpreter.

Appendix

- **Issue of "Fix":** Arbitrary combinations of different code generators (JIT, CrossGen, interpreter, etc.) in the same process might cause **inconsistent** values of `IsSupported`. (@jkotas)

```
readonly static RenderingEngine s_engine = Avx.IsSupported ? new AvxRenderingEngine() : new GenericRenderingEngine();

class GenericRenderingEngine : RenderingEngine
{
    virtual void Render()
    {
        Debug.Assert(!Avx.IsSupported);
        // ...
    }

    // ...
}
```

- **Fix #1:** Depend on Fallback to JIT for hardware intrinsics methods and let JIT generate valid method bodies.
- **Fix #2:** Implement hardware intrinsics methods in the interpreter.