Skalierbare .NET Apps mit async await

Expertenwissen und Tipps aus der Praxis

Kenny Pflug, Thinktecture

Today's menu

- Why is async await important for scalability?
- What happens during async I/O and async compute?
- How does the C# compiler transform async methods?
- What happens in memory when an async method executes?
- With examples in ASP.NET Core and Avalonia

think tecture

Kenny Pflug

Consultant Software Architect @ Thinktecture AG

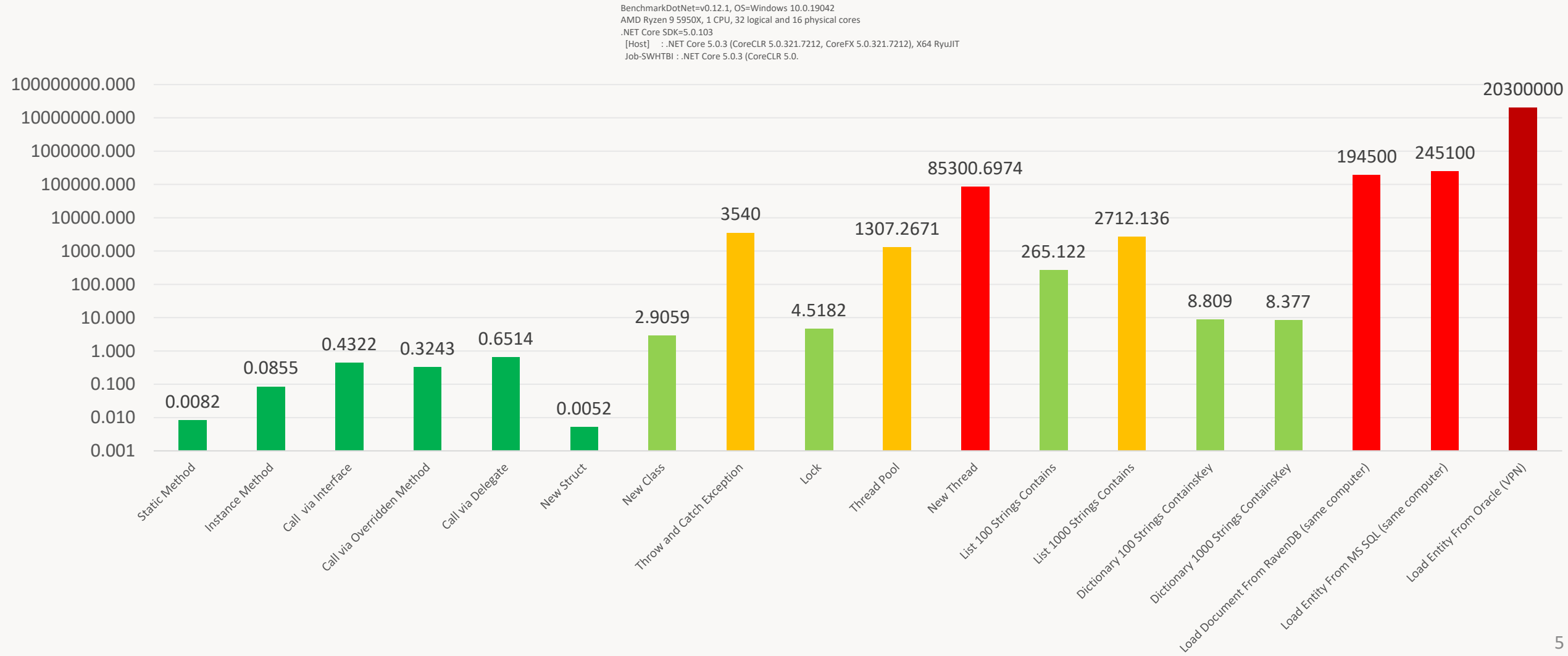
- Distributed Systems with ASP.NET Core
- .NET internals
- Cloud-native

kenny.pflug@thinktecture.com @feO2x https://www.thinktecture.com

Demo Time

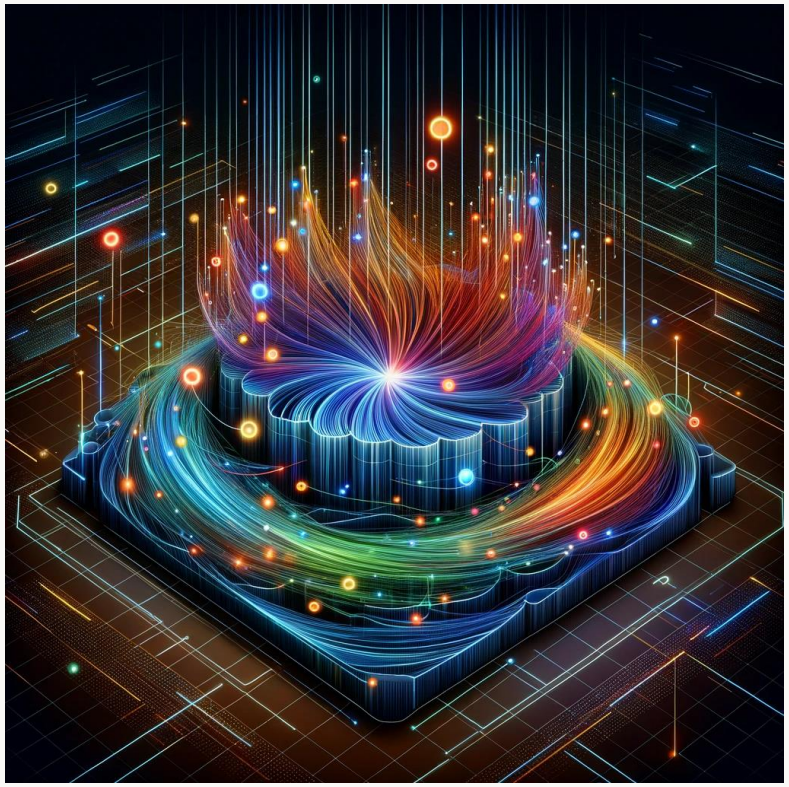
async await und service scalability

Performance of everyday things



Threads and performance in .NET

- A thread is one of the objects with the largest impact on memory and performance
- Thread stack default size: 1MB in 32 Bit mode, 4MB in 64 Bit mode
- .NET Threads are associated with OS threads, which have their own kernel objects to save registers during Context Switches
- Context Switches:
  - every ~15ms, the OS interrupts running threads and chooses new ones to run on the CPU cores
  - a thread starting to run likely needs other data than the previous thread (CPU Cache Misses)
- On Windows: every loaded DLL is notified when a thread is instantiated and destroyed



Avoid instantiating threads, reuse those of the thread pool

How many threads?

- Depends on the number of CPU cores in your target system
- Ideally, you have one thread per logical core: AMD Simultaneous Multi-Threading (SMT), Intel Hyper Threading
- The .NET Thread Pool manages an “optimal” number of worker threads
- It increases (and decreases) the number of threads dynamically based on the amount of enqueued work
(HTTP requests in ASP.NET Core)
- Work can be handed over, for example via **ThreadPool.QueueUserWorkItem** or **Task.Start**

The big issue with the .NET thread pool

Do not block its worker threads!
The thread pool will create more if work items need to be handled by it.

What blocks threads?
Usually, Thread.Sleep and sync I/O!

The big issue in more detail

- If the .NET Thread Pool has work items in its queue and the OS signals that the pool's worker threads are blocked, the pool will create additional worker threads
- Depending on the amount of work items and the block duration, this can spiral out of control (Thread Pool Starvation)
- Since .NET 7, the .NET Thread Pool is no longer implemented in C++, but in C# (ThreadPool.Portable in dotnet/runtime repo)
- This comes with a new Hill-Climbing algorithm for worker thread allocation which creates less worker threads and decreases its number earlier towards goal of the CPU's logical core count

How do we avoid synchronous I/O?

```
dbContext
    .Posts
    .Where(p => p.PublishDate >= from &&
                p.PublishDate <= to)
    .OrderBy(p => p.PublishDate)
    .ToList();
```

```
dbContext
    .Posts
    .Where(p => p.PublishDate >= from &&
                p.PublishDate <= to)
    .OrderBy(p => p.PublishDate)
    .ToListAsync();
```

But only if your data access library plays along…

Targeting different third-party systems with async I/O

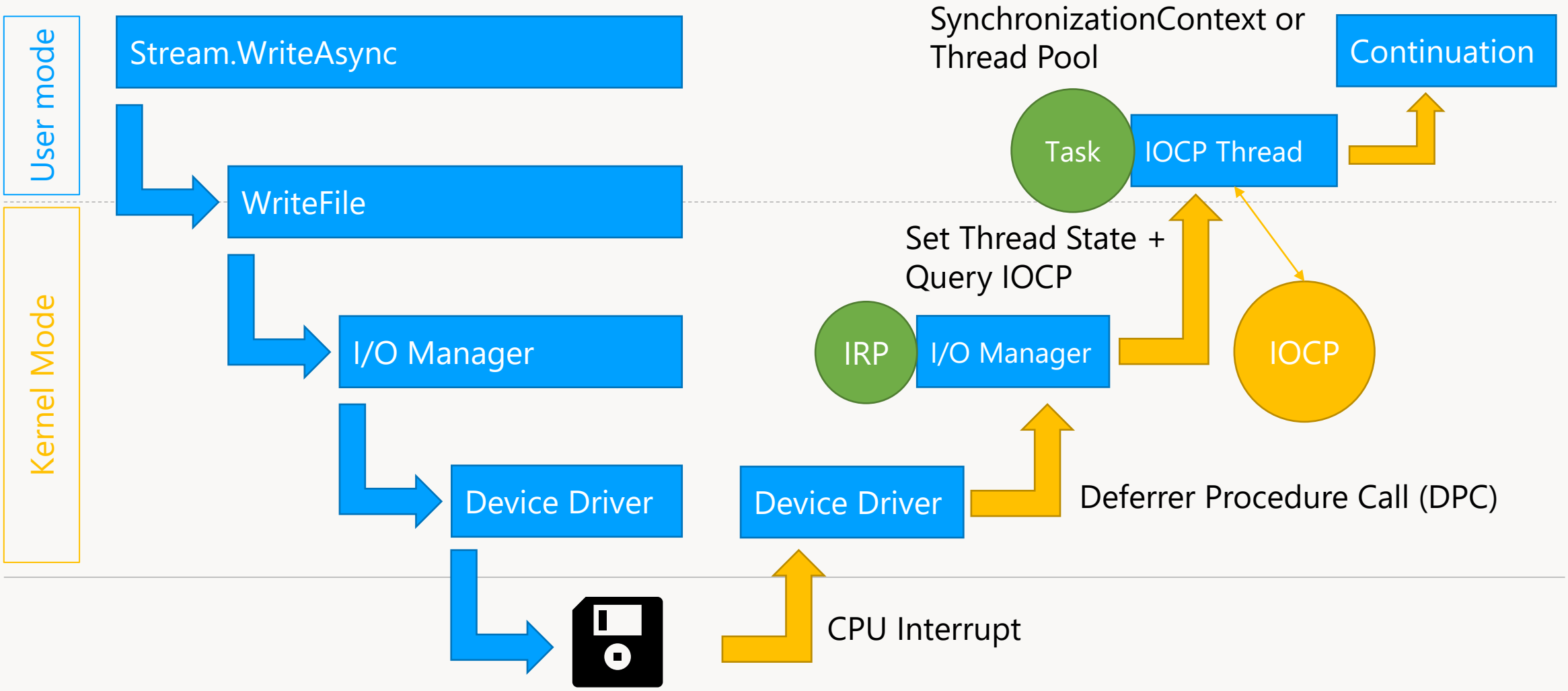
- HttpClient, GRPC Services, and Sockets are async by default
- FileStream has great async support since .NET 6 without performance degradation
- ADO.NET abstractions have async methods, but their default implementation executes synchronously (providers must override these methods and implement proper async I/O)
- Object/Relational Mappers (ORMs) usually build on top of ADO.NET – they won't fix the problems in underlying providers
- What about SAP, ERP systems, or other third-party systems with proprietary protocols?

Does your third-party access library support async I/O?

The situation with third-party access libraries

- System.Data.SqlClient and Microsoft.Data.SqlClient support async I/O except transactions (might be coming to Microsoft.Data.SqlClient)
- Npgsql: full async support
- RavenDB.Client: full async support
- MySql.Data: no support at all, you should use MySqlConnector instead
- Oracle.ManagedDataAccess.Core: async support since version 23.4 (which was published last week, be sure to update)
- SQLite: no async support (but usually local-file-based)
- RabbitMQ.Client: async support comes in version 7, but this version is only in alpha
- SAP Connector for .NET: no async support

How does I/O work under the covers (on Windows)?



What are these IOCP threads of the Thread Pool?

- IOCP stands for I/O Completion Port and is the central mechanism of the Windows I/O manager for notifying callers about async I/O events
- IOCP threads of the .NET Thread Pool bind to one IOCP and then block
- You can bind as many threads as you want to one IOCP port
- If an IOCP event is ready, the Windows I/O manager will choose one bound thread and set its state to “Ready to run”, so that it can dequeue the event from the IOCP
- After dequeuing, the IOCP thread will update the corresponding task and enqueue the continuation either on the Thread Pool or on the original caller’s SynchronizationContext
- If you run on Linux or MacOS X, there are similar mechanisms like epoll and kqueue

The difference between async I/O and async compute

Async IO

- We start an I/O operation on the calling thread, once the I/O target is working (usually a network or disk controller), we return to the caller
- The caller can do something else in the meantime (handle other HTTP requests, run the UI loop)
- When the I/O operation finishes, the continuation will be executed on the SynchronizationContext's thread or on a worker of the .NET thread pool.

Async compute

- We have a long calculation that we don't want to execute on the current thread - we use Task.Start to enqueue the operation on a worker of the .NET thread pool
- The caller can do something else in the meantime (run the UI loop)
- When the task finishes, the continuation will be executed on the SynchronizationContext's thread or on a worker of the .NET thread pool.

Don’t mix async I/O and async compute (unless necessary)

```
await Task.Run(() =>
dbContext
    .Posts
    .Where(p => p.PublishDate >= from &&
                p.PublishDate <= to)
    .OrderBy(p => p.PublishDate)
    .ToListAsync()
);
```

```
await dbContext
    .Posts
    .Where(p => p.PublishDate >= from &&
                p.PublishDate <= to)
    .OrderBy(p => p.PublishDate)
    .ToListAsync();
```

Demo Time

async await decompiled

async Methods

- The C# compiler will transform any method that is marked with the async modifier
- The resulting state machine has a MoveNext method that consists of your actual code and generated code for await expressions (depends heavily on Control Flow)
- In Debug mode, the state machine is a class instead of a struct
- The fields of this struct consist of your original method's parameters, variables, the "this" reference (if the source method was not static), task awaiters, and a method builder (the reusable part of the state machine)
- MoveNext will likely return to the caller before the task completes
- MoveNext uses labels and goto statements to jump to the correct continuation for the previous task
- You can analyze your own methods by using an IL Viewer (lowered C# code)

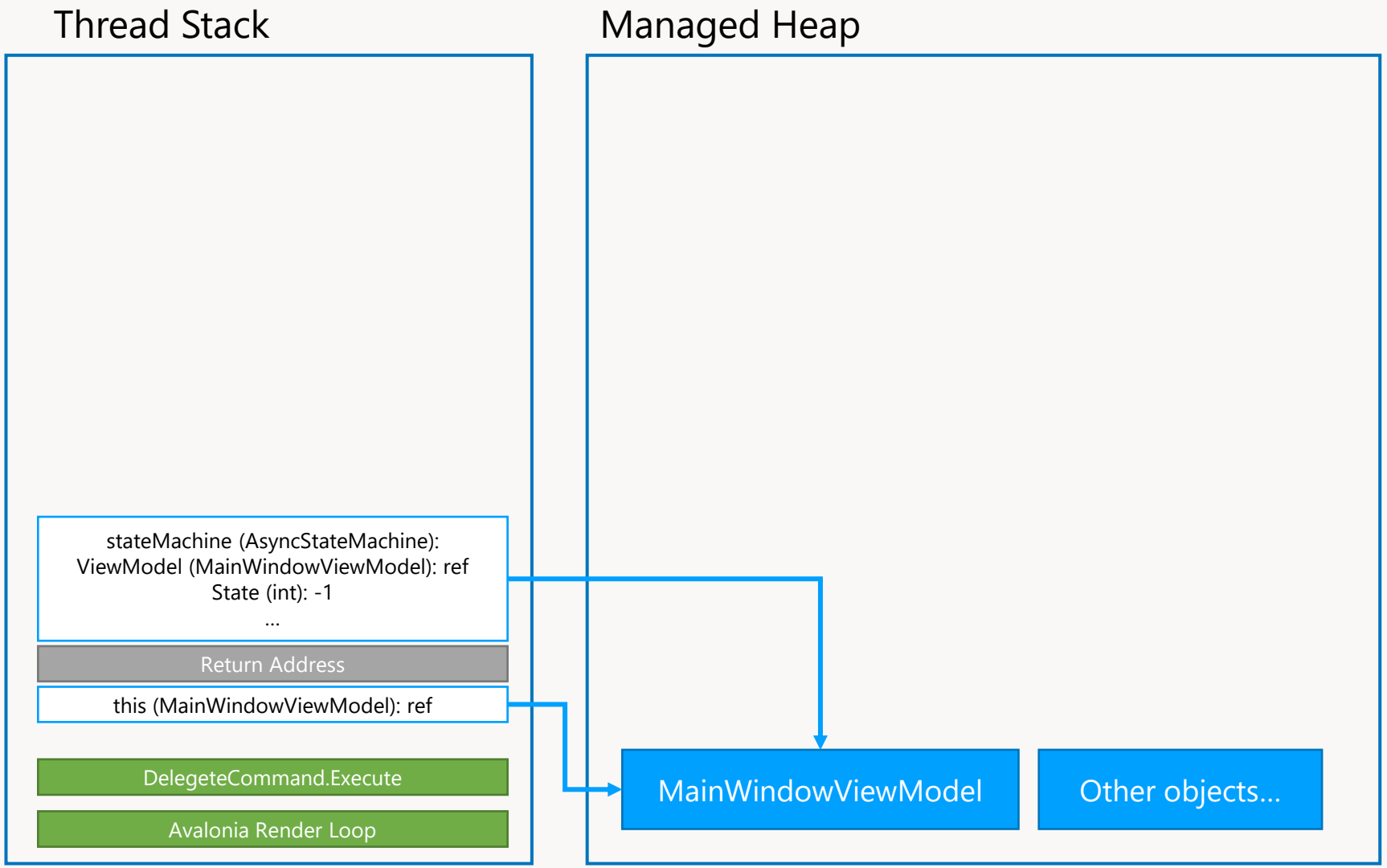
async await – what you need to be aware of

- async await has a performance overhead: initializing the state machine, boxing and moving it to the Managed Heap, etc. -> distinguish between sync and async calls!
- Be aware that each async method gets its own state machine when it is called – generally avoid nesting to many calls of async methods
- If the last statement in your async method is returning a task, you can skip the async keyword to reduce the number of state machines created (useful for Humble Objects)
- Be aware what the last statement is (e.g. using keyword for disposal)
- async await has a drastic impact on our Object-Oriented design: asynchronous method signatures are usually required across the whole call chain, avoid sync-over-async scenarios (but sometimes, this is useful)

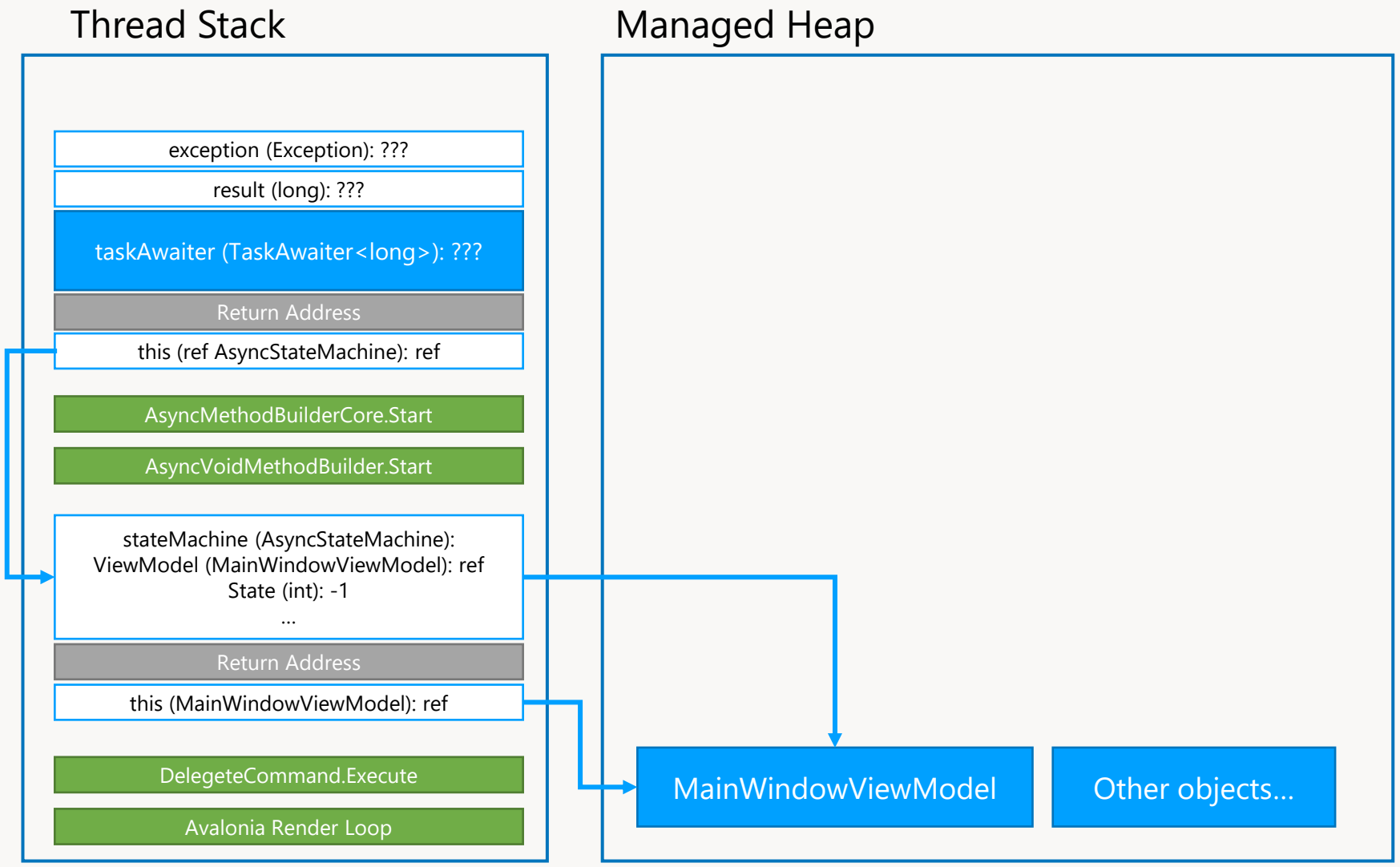
What happens in memory when an async method is called?

The memory management basics:

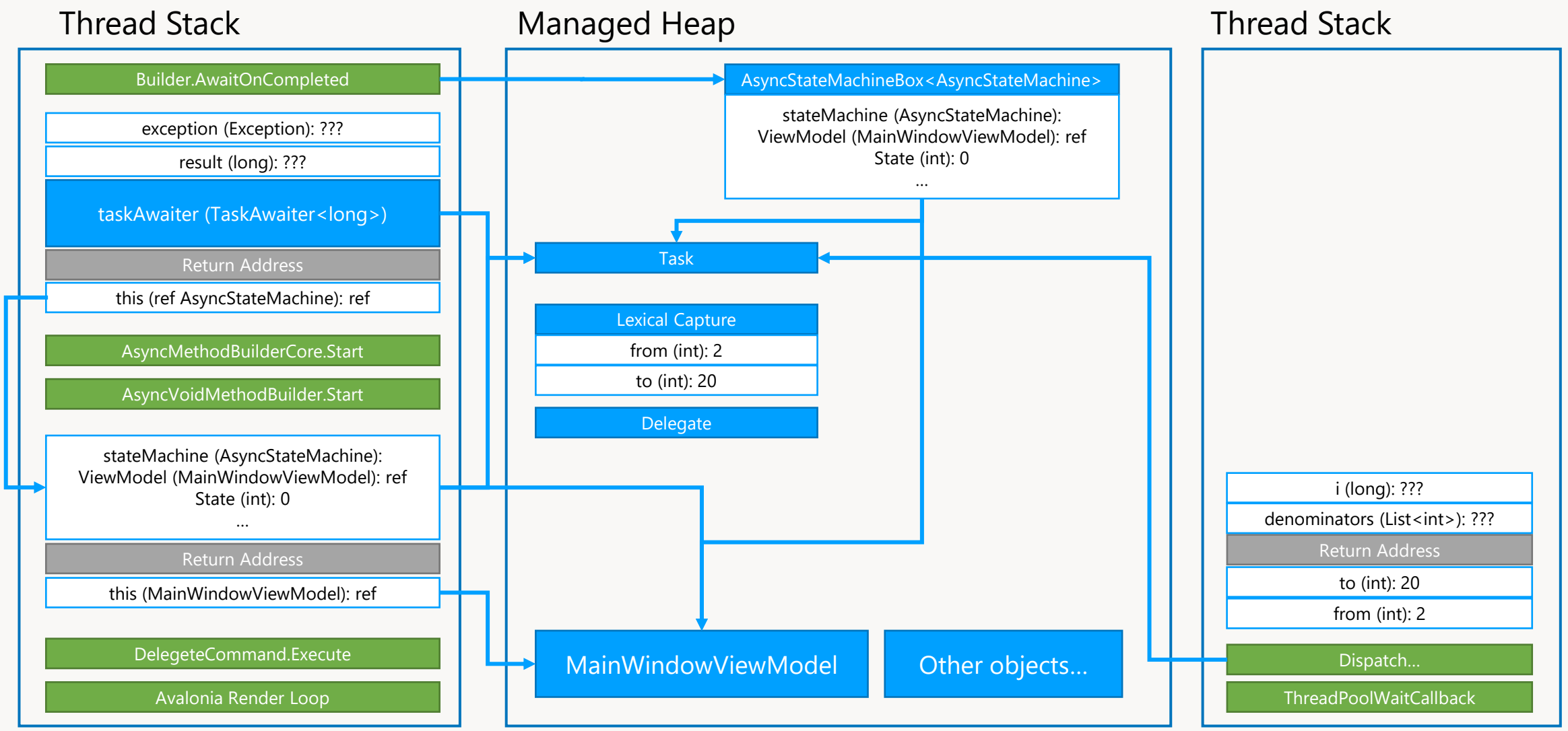
- We distinguish between thread stacks for parameters/variables and the managed heap (for objects).
- When a method is called, an activation frame is pushed on the thread stack.
- Each activation frame consists of parameters, return address, variables.
- When a method returns, its activation frame will be deallocated (by simply removing pointers).
- Value Types reside directly in a parameter/variable, or as part of an object. They can be boxed when addressed as a Reference Type!
- Reference types always point to an object in the heap.



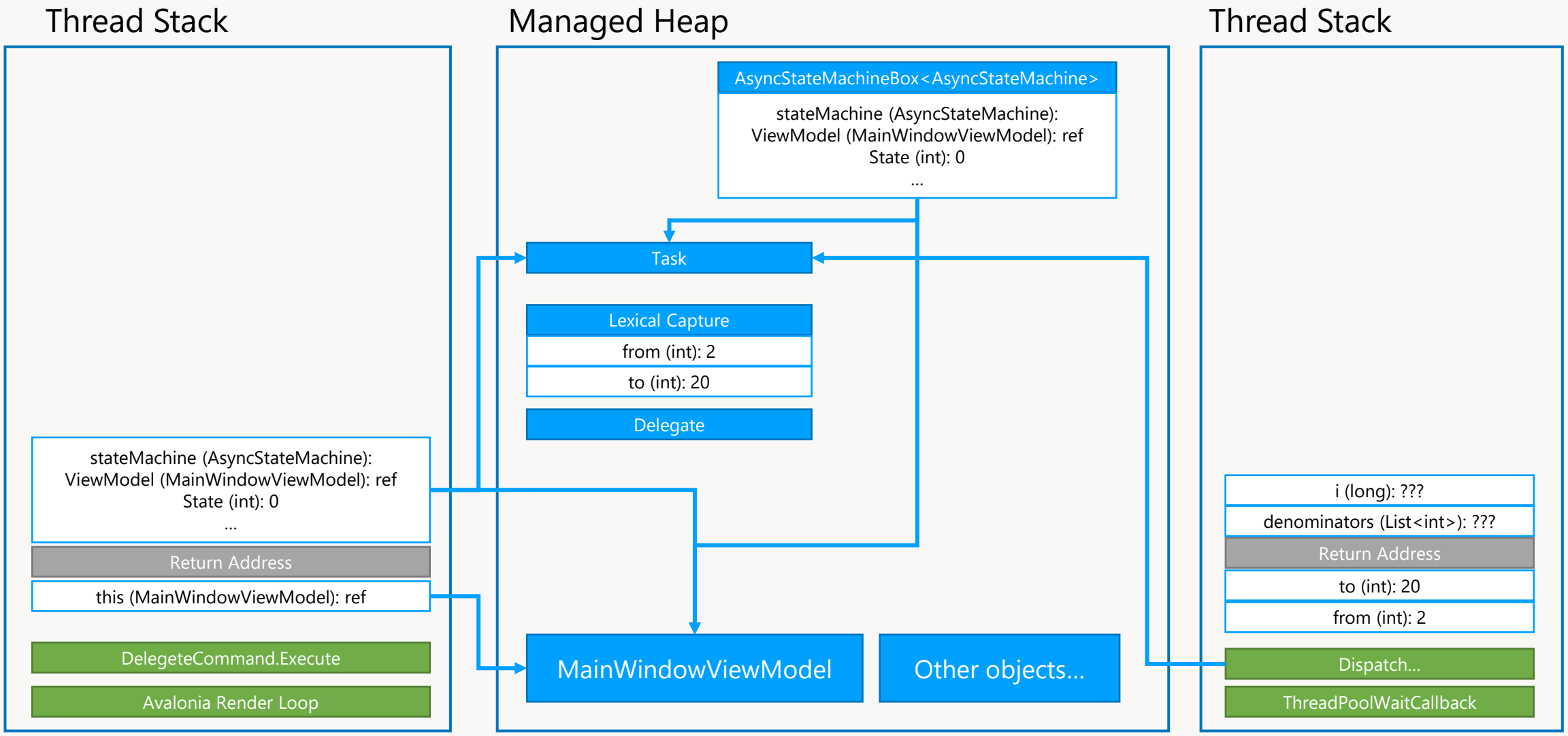
Before calling stateMachine.Builder.Start



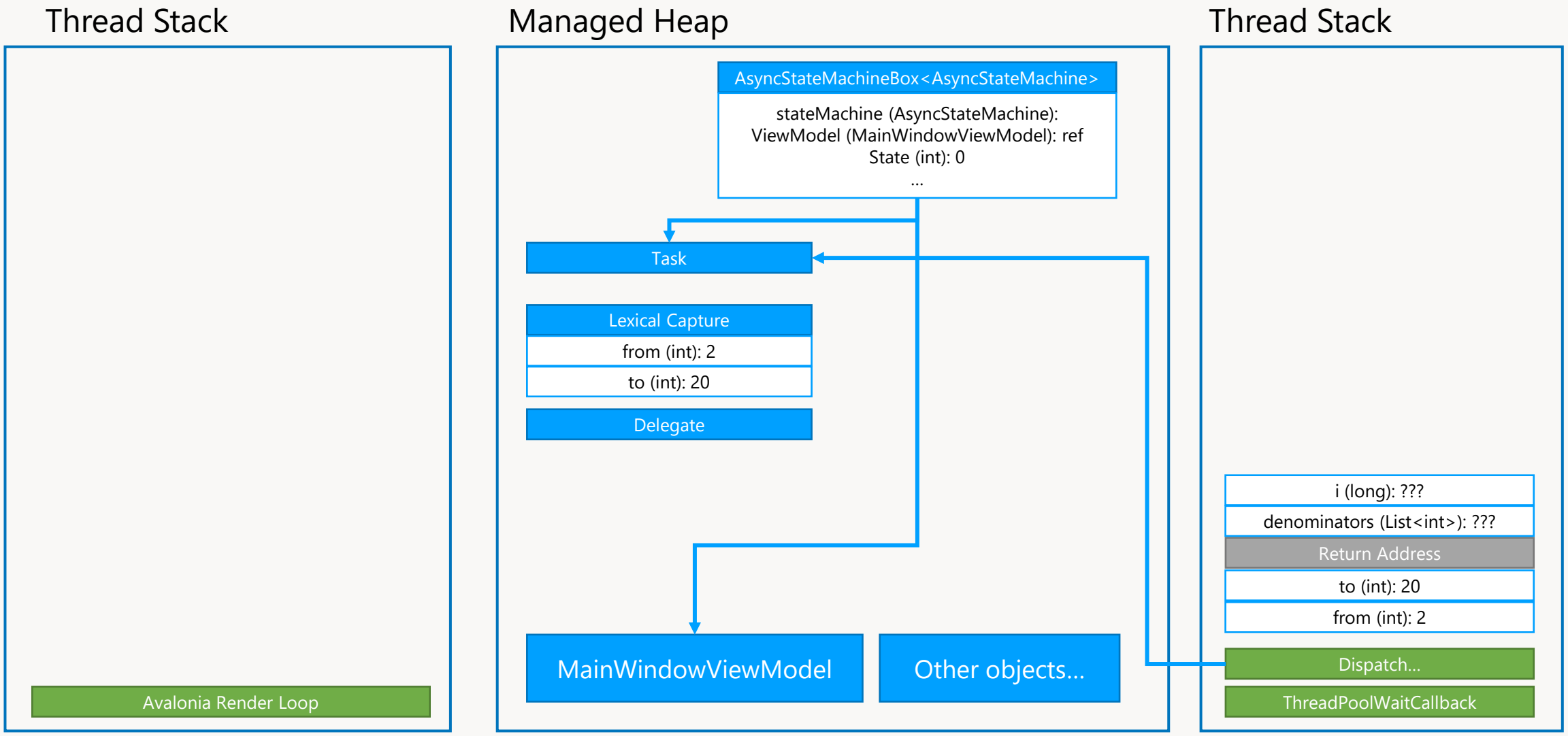
At the beginning of the first MoveNext call



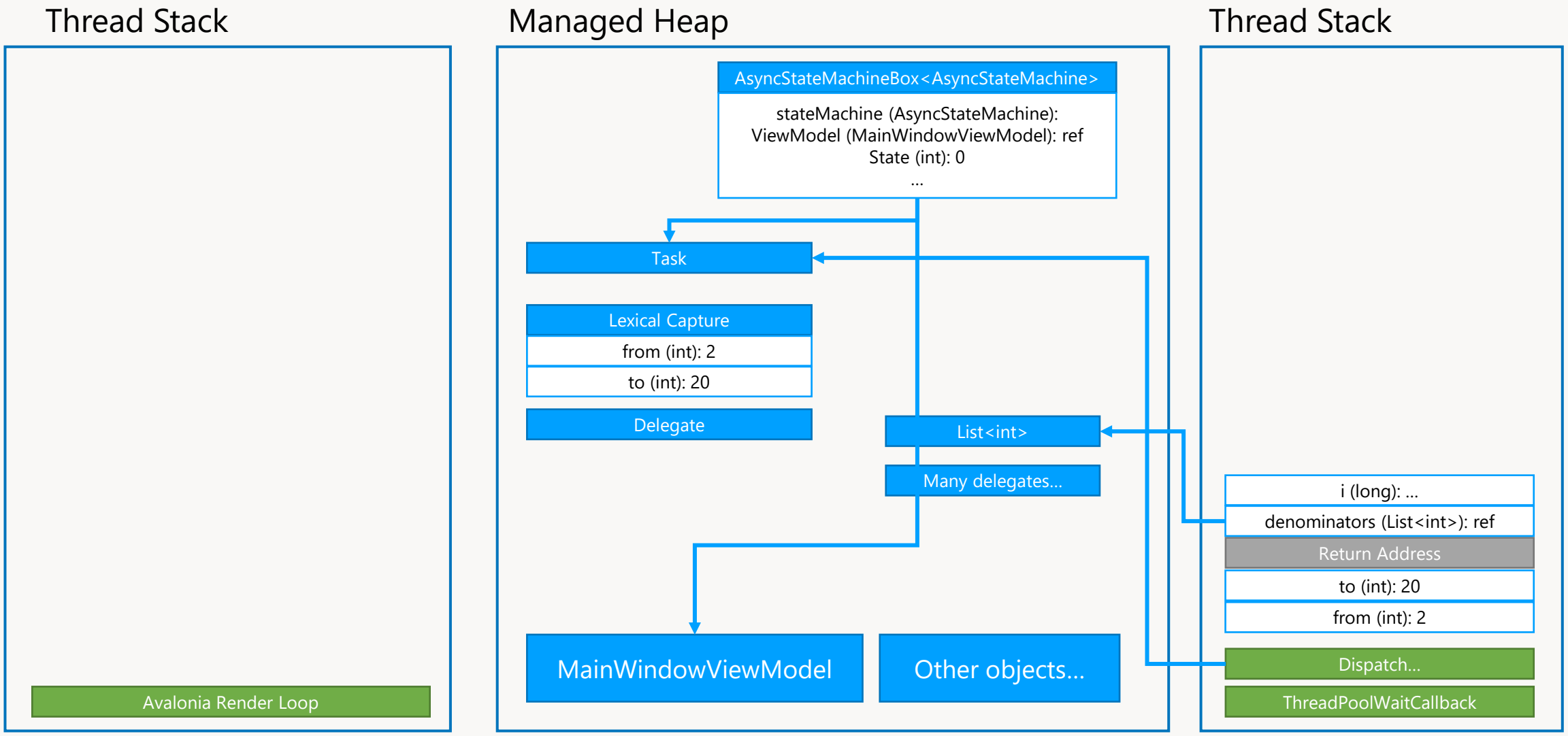
At the end of Builder.AwaitOnCompleted



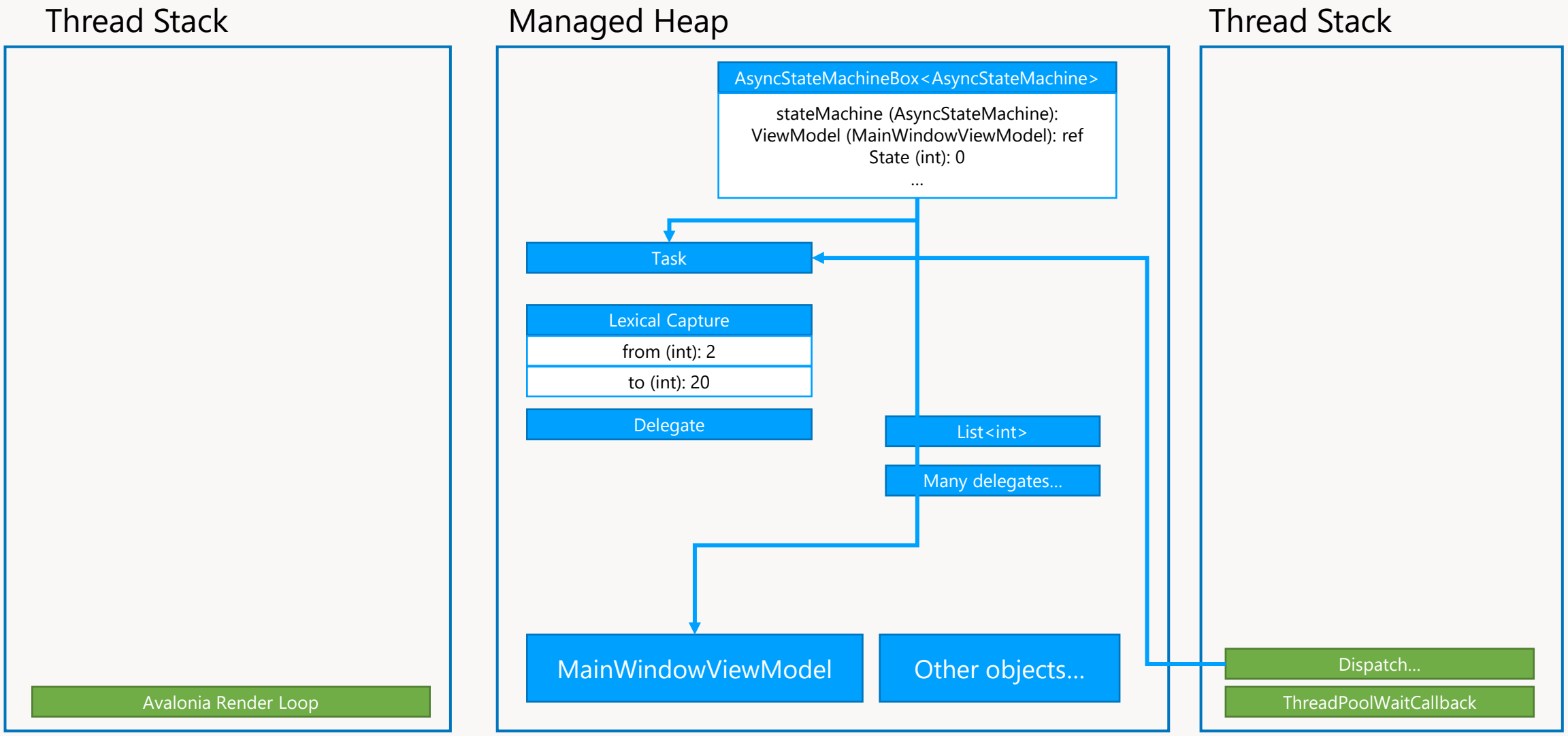
Return to CalculateOnBackgroundThreadDecompiled



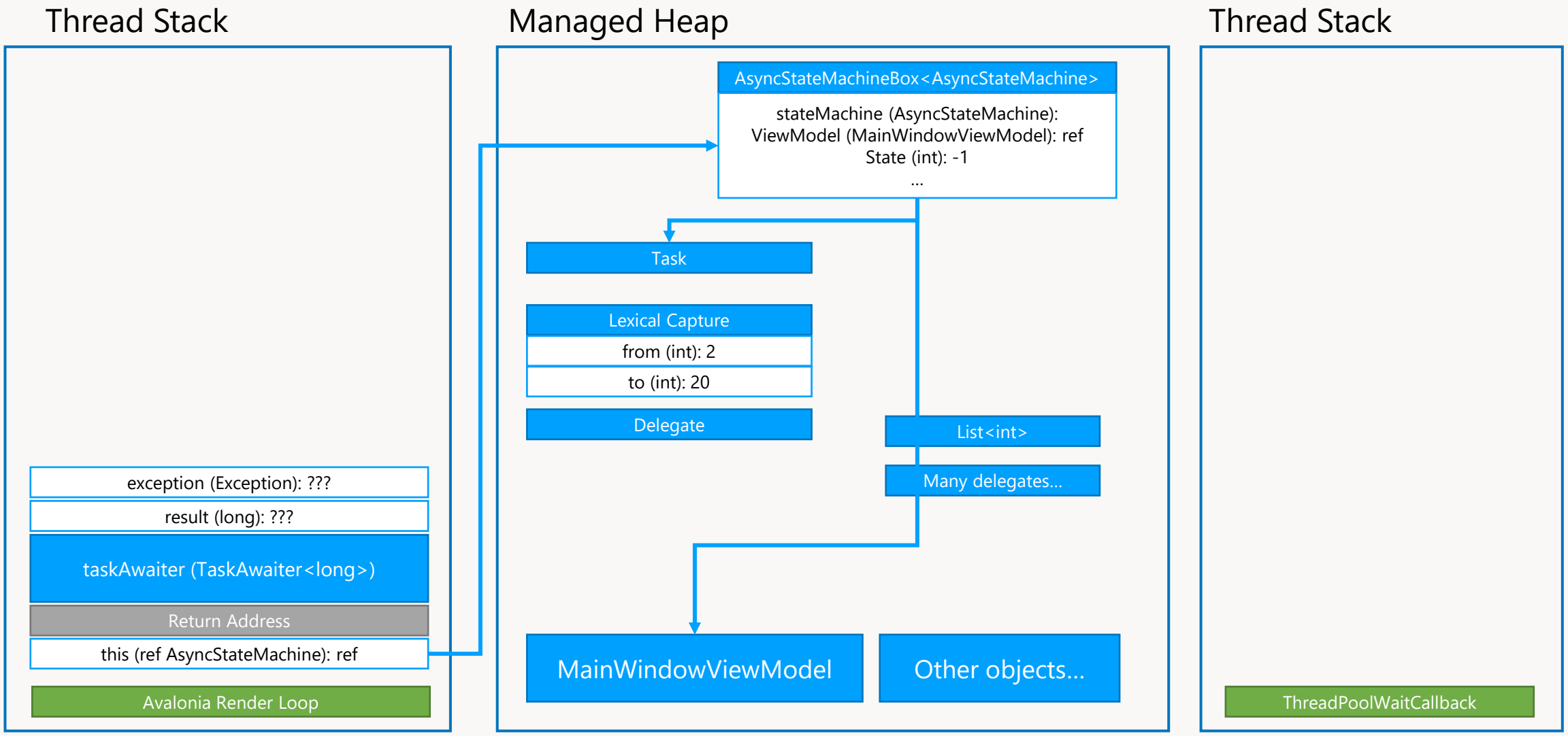
Return to the Avalonia render loop



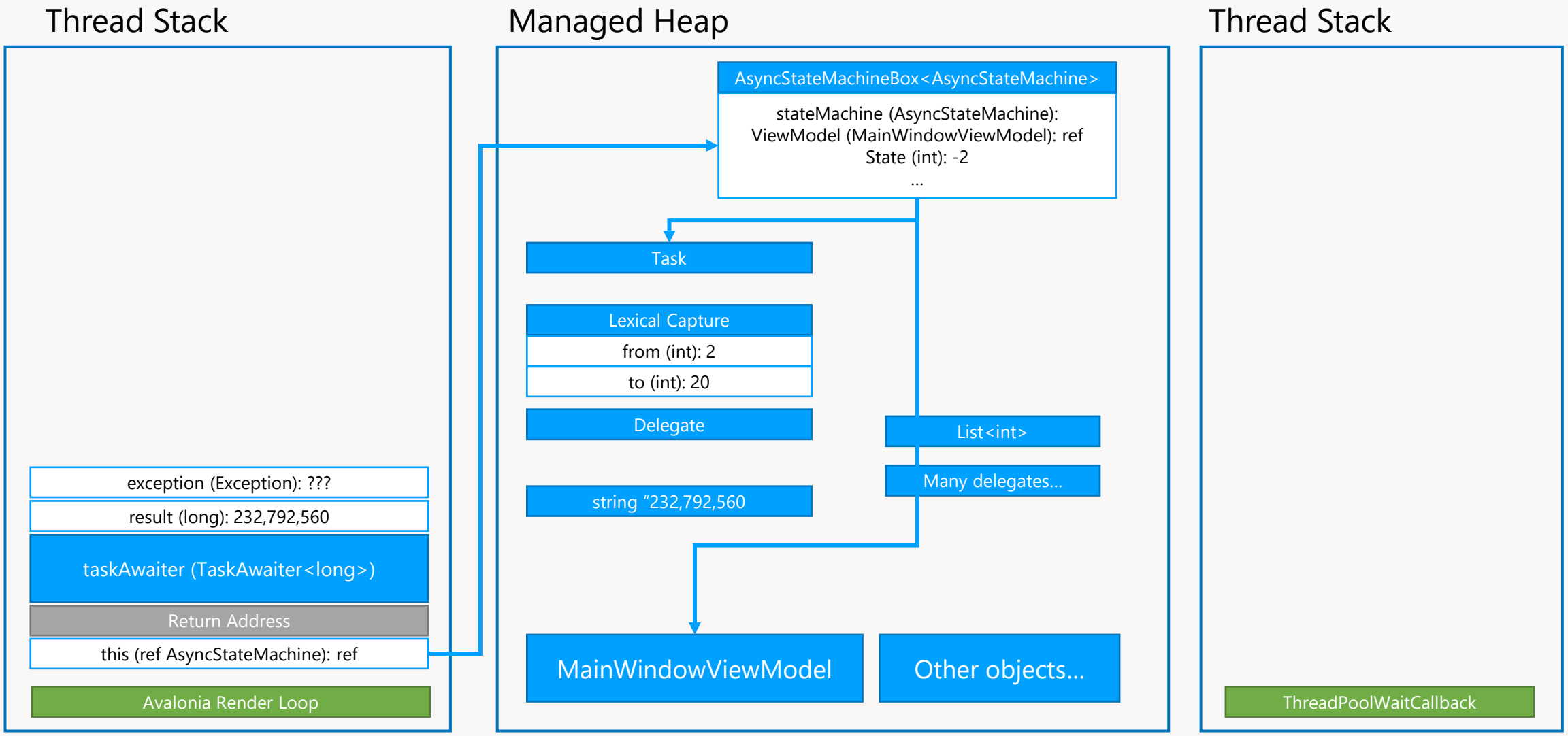
Progress on the background thread



Background task is completed



Continuation starts on the UI thread



At the end of the second MoveNext call

What can we learn from all this?

- Async methods are rewritten, parameters and variables become fields in a state machine
- The state machine can return early while the async operation is still ongoing
- This is why ref/out parameters cannot be used in async methods
- When returning early, the state machine is boxed on the managed heap – this way, it survives when the activation frames of the initiating thread are released
- When the async operation signals completion via the corresponding task, the async state machine's MoveNext method is executed on the original caller's thread (via SynchronizationContext) or on a worker thread of the .NET thread pool

We need new principles, especially for young devs

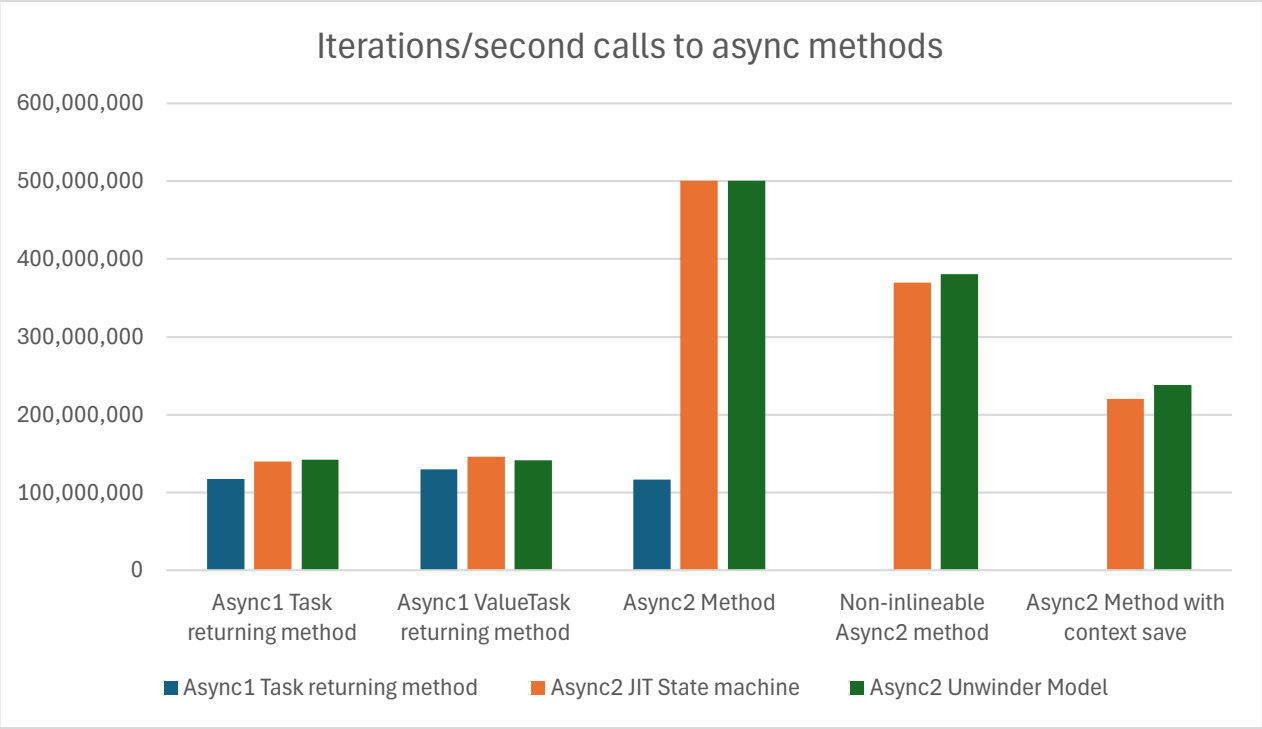
Learn the Internals (LTI)

Study the internals of the runtimes, tools, and frameworks/libraries that you use and understand how they solve recurring problems. Examine which call patterns will result in problems / undesirable outcome and ensure that these patterns are avoided in your code base.

Respect the Process Boundary (RPB)

Make a clear distinction between in-memory and I/O operations as the latter usually have way longer execution times. Consider performing I/O asynchronously so that the calling threads can perform other work while an I/O operation is ongoing. I/O calls should be abstracted so that in-memory logic can run independently from third-party systems.

What might change in .NET 9



In the current implementation, the async state machine is always generated as IL code. For .NET 9, the .NET team explores the possibility to move large parts of the state machine to the Common Language Runtime (async2). This could reduce the generated state machine and increase performance.

https://github.com/dotnet/runtime/issues/94620

Sources

- Jeffrey Richter: Advanced Threading in .NET
- Joseph Albahari: Threading in C#
- John Skeet: Asynchronous C# 5.0
- Stephen Toub: How async await really works
- Stephen Cleary: There is no thread
- Konrad Kokosa: Pro .NET Memory Management

Vielen Dank! Ich freue mich auf Feedback!