Self-Study

1. When did Microsoft do to overcome the Jitting overhead ?

1. Overview of JIT Compilation in .NET

In the .NET framework, source code (e.g., C#) is compiled into Common Intermediate Language (CIL), a platform-agnostic intermediate representation. At runtime, the .NET Common Language Runtime (CLR) uses a JIT compiler to translate CIL into native machine code specific to the operating system and hardware architecture. This process introduces overhead because it requires runtime compilation, but Microsoft employs several strategies to minimize this.

2. Is the JIT Compiler 64-bit or 32-bit?

The JIT compiler in .NET is designed to generate code that matches the architecture of the process running the application:

- **32-bit vs. 64-bit**: The JIT compiler itself is part of the CLR, which runs in the same bitness as the application. For example, if a .NET application runs as a 32-bit process on a 64-bit OS, the JIT compiler generates 32-bit machine code. Conversely, for a 64-bit process, it generates 64-bit machine code. Since .NET Framework 4.0 and .NET Core, the default for most applications on 64-bit systems is to run as 64-bit processes, leveraging the 64-bit JIT compiler (RyuJIT in newer versions) for better performance on modern hardware.
- **Impact of Bitness**: A 64-bit JIT compiler can take advantage of larger memory addressing and more registers, which can lead to more efficient code generation compared to 32-bit. For instance, 64-bit code can use additional general-purpose registers, reducing memory access and improving performance. However, the choice of bitness depends on the application's configuration (e.g., AnyCPU,

x86, or x64 in the project settings). Running a 32-bit process on a 64-bit system may incur slight overhead due to compatibility layers, but the JIT itself is optimized for the target architecture.

3. Does Jitting Happen for All Functions at Once or Only for Called Functions?

The JIT compiler in .NET operates **on-demand**, compiling only the methods that are called during execution:

- **Method-by-Method Compilation**: When a method is invoked for the first time, the CLR passes its CIL to the JIT compiler, which generates native machine code for that specific method. This approach reduces startup time and memory usage because only the code that is actually executed is compiled. Uncalled methods remain in CIL form, avoiding unnecessary compilation overhead.
- **Pre-JIT Scenarios**: There are exceptions where more code is compiled upfront. For example, in .NET's **ReadyToRun** (RTR) compilation (formerly called NGen), a precompilation step generates native code for an entire assembly before execution, reducing JIT overhead at runtime. ReadyToRun is particularly useful for improving startup performance in large applications.

4. Does Jitting Happen Every Time the Program Runs?

By default, JIT compilation occurs **every time a program runs**, as the generated native code is not persisted to disk in the standard JIT process. However, Microsoft introduced mechanisms to mitigate this repeated overhead:

- **NGEN (Native Image Generator)**: In the .NET Framework, NGen allows precompilation of CIL into native machine code, which is stored in the Native Image Cache on disk. When the application runs, the CLR loads the precompiled native code instead of jitting, significantly reducing startup time. However, NGen has limitations,

such as requiring installation on the target machine and potential invalidation if dependencies change.

- **ReadyToRun (RTR)**: Introduced in .NET Core and later .NET versions, ReadyToRun is a more modern approach to precompilation. It generates native code during the build process, embedding it into the assembly. This reduces JIT overhead at runtime while being more portable than NGen, as the precompiled code is included in the application package.
- **Tiered Compilation**: Starting with .NET Core 2.1 and .NET 5, Microsoft introduced **tiered compilation**, which further optimizes the JIT process:
    - **Tier 0 (Quick JIT)**: Initially, methods are compiled quickly with minimal optimizations to reduce startup time. This produces less efficient code but allows the application to start executing faster.
    - **Tier 1 (Optimized JIT)**: As the application runs, frequently executed methods are recompiled in the background with full optimizations (e.g., inlining, loop unrolling) to improve performance. This happens asynchronously, so it doesn't block execution.
    - Tiered compilation balances startup speed and runtime performance by deferring heavy optimization to hot paths (frequently executed code).
- **Caching in Memory**: Within a single run of an application, the JIT-compiled native code is cached in memory. If a method is called multiple times during the same session, the CLR reuses the cached native code, avoiding redundant compilation.

5. Additional Optimizations to Reduce JIT Overhead

- **Inlining**: The JIT compiler can inline small methods (i.e., embed their code directly at the call site) to reduce function call overhead. This is particularly effective in Tier 1 compilation.
- **Profile-Guided Optimization (PGO)**: In newer .NET versions, dynamic PGO collects runtime data to guide JIT optimizations, focusing on frequently executed code paths.
- **RyuJIT**: Introduced in .NET Framework 4.6 and used in .NET Core/.NET 5+, RyuJIT is a high-performance JIT compiler optimized for modern hardware. It generates more efficient code than the older JIT32 compiler and supports advanced optimizations like SIMD (Single Instruction, Multiple Data) for parallel processing.
- **AOT (Ahead-of-Time) Compilation**: In .NET 7 and later, Microsoft enhanced support for AOT compilation, where the entire application is compiled to native code at build time, eliminating JIT compilation at runtime. This is particularly useful for scenarios requiring minimal startup time, such as serverless applications or mobile apps (e.g., via .NET MAUI).
- **Code Sharing**: The CLR shares JIT-compiled code across AppDomains or processes when possible, reducing redundant compilation for shared libraries like mscorlib.