Software Assurance Tips

A product of the Software Assurance Tips Team[1]

Generated Wednesday 28th April, 2021

Jon Hood

Monday 26th April, 2021

1 Polymorphic Catch Performance in C#

In a recent code review, one of our customers were marked for being sloppy with catch-all exceptions (CWE-396), which is sometimes an indicator of poorly designed program flow. To fix this, the developers changed the catch blocks to something like the one in Figure 1.

```
1  try
2  { DoSomethingWith("C:\\File.txt"); }
3  catch (Exception e) when (e is FileNotFoundException)
4  { DoSomethingElse(e); }
```

Figure 1: When-Conditioned Exception

The intent of a "when" clause in an exception is for situations where the exception may have an error code. For example, a SqlException may return different codes that can be used to differentiate their catch blocks and clean up a failed state. So how does the code in Figure 1 differ from the code in Figure 2?

```
1  try
2  { DoSomethingWith("C:\\File.txt"); }
3  catch (FileNotFoundException e)
4  { DoSomethingElse(e); }
```

Figure 2: Classically-Conditioned Exception

While both blocks of code achieve the same result, the generated code to catch the exception in Figure 1 has twelve times as many instructions to execute. The .NET Intermediate Language (IL) is very different between the two implementations. The "when" clause creates what the IL defines as a filter block and instantiates a structure similar to a truth stack. This additional structure and the extra operations on it reduce performance and add unnecessary bloat to the software. The generated IL can be compared in Figures 3 and 4. The IL was created in optimized Release mode.

```
1  filter {
2  IL_000d:  isinst [System.Runtime]System.Exception
3  IL_0012:  dup
4  IL_0013:  brtrue.s IL_0019
5  IL_0015:  pop
6  IL_0016:  ldc.i4.0
7  IL_0017:  br.s IL_0024
8  IL_0019:  isinst [System.Runtime]System.IO.FileNotFoundException
9  IL_001e:  ldnull
10 IL_001f:  cgt.un
11 IL_0021:  ldc.i4.0
12 IL_0022:  cgt.un
13 IL_0024:  endfilter
14 }
```

Figure 3: When-Conditioned Exception IL

This leads to another question: what's the most optimal way to catch several types of exceptions? The most optimal way is to catch the general exception, then use polymorphism to operate on it (code in Figure 5). The IL code adds a single comparison for each conditional in an if statement and

```
catch class [mscorlib]System.IO.FileNotFoundException
```

Figure 4: Classically-Conditioned Exception IL

avoids the overhead incurred by the when clause's filter block. Note that most scanning tools and style checkers will identify Figure 5 as a catch for the generic exception, but this should be marked as a false positive if the handler appropriately determines the failure scenarios for the appropriate exception types. The IL code for Figure 5 up to the first exception type is shown in Figure 6. Please also realize that such minimal performance gains should not be interpreted as a reason to violate a consistent standard already established in a code base.

```
1 try
2 { DoSomethingWith("C:\\File.txt"); }
3 catch (Exception e)
4 {
5         if ((e is FileNotFoundException) || (e is SecurityException))
6                 DoSomethingElse(e);
7         else
8                 Log(e);
9 }
```

Figure 5: If-Formatted Exception Chain

```
catch class [mscorlib]System.Exception {
        IL_000d:  isinst [System.Runtime]System.IO.FileNotFoundException
```

Figure 6: If-Formatted Exception Chain IL

When performance is a primary concern, developers should include comments that indicate their consideration of exceptional cases in generic catch blocks. Validators should watch for these comments to help them mark the generic exception catches as false positives.

References

[1] Jon Hood, ed. *SwATips*. https://www.SwATips.com/.