TPL Performance Improvements in .NET 4.5

Joseph E. Hoag

Microsoft Corporation

Overview

The Task Parallel Library (TPL) exists to enable .NET programmers to exploit parallelism and asynchrony in a relatively painless fashion. TPL comes in two main parts: (1) a low-level Task library that allows users great flexibility in setting up parallel/asynchronous scenarios, and (2) some higher level functionality (e.g. Parallel.For, Parallel.ForEach, Parallel.Invoke) built on Task that provides "out-of-the-box" logic to implement some fairly common parallel patterns.

TPL in .NET 4 was focused on fork-join style programming, which involves spawning Tasks and waiting for them to complete. For example:

```
Task A = Task.Factory.StartNew(() => { ParallelLogicA(); });
Task B = Task.Factory.StartNew(() => { ParallelLogicB(); });
Task.WaitAll(A, B);
AdditionalLogic();
```

As part of .NET 4, we also enabled TPL to support less structured forms of parallelism, including continuation support that enabled one or more tasks to be executed when one or more other tasks completed. Continuation-style programming eschews explicit waiting, instead specifying a directed acyclic graph of Tasks that are spawned in a specific order. The fork-join example above would look like this if converted to continuation-style programming:

```
Task A = Task.Factory.StartNew(() => { ParallelLogicA(); });
Task B = Task.Factory.StartNew(() => { ParallelLogicB(); });
Task.Factory.ContinueWhenAll(new Task[] { A, B }, _ =>
{
    AdditionalLogic();
});
```

While continuations were certainly important to us in .NET 4, our optimization efforts were focused more on structured parallelism and less on unstructured. In developing .NET 4.5, we acknowledged this continuation-style of programming to be as mainline a scenario as fork-join programming, largely due to the emergence of async/await support (which is built on top of continuations) in .NET 4.5. During .NET 4.5 development, we invested a lot of time in making continuations faster, while not hurting the performance of fork-join programming, and as you'll see, often actually helping.

In this document, we will examine some of the changes made to TPL in .NET 4.5, compare times for selected operations in .NET 4 vs. .NET 4.5, and make some general "best practices" recommendations for those interested in using TPL in .NET 4.5.

A Dive into the Major TPL Changes in .NET 4.5

(NOTE: This section divulges some implementation details in an effort to help the reader understand what we've changed in .NET 4.5. Please remember that these implementation details are very subject to change in future .NET releases – so please do not take any dependencies on them! We discussed similar details about .NET 4's implementation, and as you'll see, major modifications have been made to the internal workings of TPL since .NET 4.)

Restructuring Task

As mentioned in the Overview, our .NET 4 release was initially designed around fork-join scenarios – that is, launching one or more tasks and then waiting on them. The .NET 4.5 release evolves to make continuation-based programming perform much better, while at the same time maintaining the high levels of performance enabled for fork-join style programming.

In the initial .NET 4 TPL release, the Task class was divided into two parts: more commonly used fields were implemented as fields directly on the Task object, while less commonly used fields were put into a Task.ContingentProperties object that was only inflated if one of those less common fields needed to be written or read. This allowed Task to have a minimal memory footprint for common scenarios, while also supporting more rare scenarios (at the pay-for-play cost of a larger memory footprint).

In .NET 4.5, we maintained the idea of "more commonly used" and "less commonly used" fields, but shuffled them around a bit. Conceptually, Task was restructured from this in .NET 4:

```
public class Task : …
{
    // ...
    // The execution context to run the task within, if any.
    internal ExecutionContext m_capturedContext;
    // Lazily created if waiting is required.
    internal ManualResetEventSlim m_completionEvent;
    // Extra data instantiated only on-demand
    internal ContingentProperties m_contingentProperties;
    // ...
    internal class ContingentProperties
    {
        internal List<TaskContinuation> m_continuationList;
        // ...
    }
}
```

to this in .NET 4.5:

```
public class Task : …
{
    // Continuations storage
    internal object m_continuationObject;
    // Extra data instantiated only on-demand
    internal ContingentProperties m_contingentProperties;
    // ...
    internal class ContingentProperties
    {
```

```
        // The execution context to run the task within, if any.
        internal ExecutionContext m_capturedContext;
        // Lazily created if waiting is required.
        internal volatile ManualResetEventSlim m_completionEvent;
        // ...
    }
}
```

On the whole, these changes moved two object-sized fields out of Task and into Task.ContingentProperties, and one object-sized field out of Task.ContingentProperties and into Task. This resulted in an overall reduction in the memory footprint of Task of one object-sized field (4 bytes on x86, 8 bytes on x64) for common scenarios.

Let us now consider these refactorings individually. It was an easy choice to move the **m_capturedContext** field from Task to Task.ContingentProperties. If the captured ExecutionContext is null or "default," that information can be stored in a single bit elsewhere in the Task (Task maintains a bit field with such flags, and there were unused bits we were able to apply for this and other purposes). If, on the other hand, a non-null, non-default ExecutionContext needs to be instantiated, the time/memory taken to do so far outweighs the time/memory needed to allocate the ContingentProperties object. So the inflation of the ContingentProperties object is relatively painless in this case.

Moving **m_completionEvent** from Task to Task.ContingentProperties was a little trickier. In .NET 4, the completion event was inflated the first time that Wait() was called on a Task when that Task had not yet completed. Certainly, we didn't want to inflate the ContingentProperties for a Task every time that we waited on one. Accordingly, in .NET 4.5, Task.Wait functionality was transformed from this (conceptually):

```
bool Wait(…)
{
        ...
        if(!IsComplete)
        {
                if (m_completionEvent == null)
                    m_completionEvent = new ManualResetEventSlim();
                return m_completionEvent.Wait();
        }
}
```

to this (conceptually):

```
bool Wait(…)
{
        ...
        if(!IsComplete)
        {
                var event = new ManualResetEventSlim(false);
                this.ContinueWith( (antecedent, state) =>
                {
                        var myEvent = (ManualResetEventSlim) state;
```

```
                    myEvent.Set(true);
                }, event);
                return event.Wait();
            }
        }
```

The upshot here is that, in most cases, there is no need to store a ManualResetEventSlim on the Task at all; it can be "pay for play" for those wishing to use Wait() and the fork-join model. There is one scenario, though, in which the **m_completionEvent** field is still needed – synchronously waiting on an IAsyncResult, as in this example:

```
Task t1 = Task.Factory.StartNew( () => {});
((IAsyncResult)t1).AsyncWaitHandle.WaitOne();
```

In the case above, we need to instantiate/inflate the internal **m_completionEvent**, which in .NET 4.5 means inflating the Task's ContingentProperties. However, the IAsyncResult.AsyncWaitHandle.WaitOne() scenario is relatively rare, even amongst Asynchronous Programming Model (APM) users, where callbacks are highly preferred to synchronous blocking. Further, the cost of allocating a kernel synchronization primitive (as is done for ManualResetEvent) is a much more significant cost than inflating contingent properties.

Finally, we moved Task.ContingentProperties.**m_continuationList** to Task.**m_continuationObject** in order to optimize the performance of attaching continuations. This will be discussed in more detail in the next section.

Continuation streamlining

In .NET 4, continuations were optimized as a secondary use case for Tasks, so the Task's continuation list was stored in Task.ContingentProperties. In .NET 4.5, continuations were optimized as a primary use case for Tasks, due in no small part to the emergence of the async/await functionality which depends heavily upon continuations.

Attaching the initial continuation for a Task in .NET 4 meant (1) inflating the Task's ContingentProperties, (2) allocating a continuation list, and (3) inserting the continuation into that list. This was a pretty high price to pay (in terms of both time and memory) for adding a single continuation to a Task, especially considering that analysis of common scenarios indicates that having at most one continuation per Task is by far the most common case.

In .NET 4.5, we removed **m_continuationList** from Task.ContingentProperties, and replaced it with Task.**m_continuationObject**. The **m_continuationObject** field can be null (meaning "no continuations"), a single continuation object, a list of continuation objects, or a sentinel value meaning "this Task has completed." Attaching the initial continuation for a Task now requires a simple Interlocked.CompareExchange operation, which is **much** faster than what we did in .NET 4. Attaching a

second continuation to a Task is a bit more expensive, requiring the allocation of a List<object> and a couple of Interlocked operations. However, attaching two (or more) continuations to a Task is still faster in .NET 4.5 than it was in .NET 4, as we will see in our benchmark results later.

Task<TResult> optimizations

In .NET 4, the fields in the Task<TResult> class looked like this:

```
public class Task<TResult> : Task
{
    private object m_valueSelector; // The function which produces a value.
    private TResult m_result; // The result value itself, if set.
    internal bool m_resultWasSet; // Whether the result value was set.
    private object m_futureState;
}
```

It turned out that with some smart restructuring, only the **m_result** field was truly necessary. By repurposing fields already on the base Task class, **m_valueSelector** and **m_futureState** could be made obsolete, and the information stored by **m_resultWasSet** could instead be stored in the base type's aforementioned state flags.

In .NET 4.5, we removed the unnecessary fields. In addition, our changes eliminated the need for some delegate allocations during Task<TResult> construction. These changes resulted in a significant reduction in Task<TResult>'s memory footprint, and also sped up the construction of Task<TResult> objects.

Timing Comparisons

In the previous section, we discussed some of the TPL performance improvements in .NET 4.5. Let's run some benchmarks and put those improvements to the test.

Note that the figures given in this section are informal measurements that we are sharing to highlight the effectiveness of the improvements that we've made in .NET 4.5. Your exact results may vary, depending upon the specific architecture of your system, but the general trends shown by the numbers in this section should be valid across all architectures.

And please keep in mind that these timings reflect improvements in the *overheads* associated with various TPL operations. The amount of improvement that you will see in your TPL-based applications will vary. Applications that are sensitive to TPL overhead (which typically involve lots of short-running Tasks) should see non-negligible performance improvement when moving from .NET 4 to .NET 4.5. Applications that are not sensitive to TPL overhead (which typically involve a few long-running Tasks) will not see as much improvement.

Task Creation

Let's compare .NET 4 vs. .NET 4.5 performance with respect to Task creation. We'll use this simple benchmark to test both time and memory consumption associated with Task creation:

```
public static Tuple<long, long> CreateTasks(int ntasks)
{
    Task[] tasks = new Task[ntasks];
    Stopwatch sw = new Stopwatch();
    Action action = () => { };
    long startBytes = GC.GetTotalMemory(true);
    sw.Start();
    for (int i = 0; i < ntasks; i++) tasks[i] = new Task(action);
    sw.Stop();
    long endBytes = GC.GetTotalMemory(true);
    GC.KeepAlive(tasks);
    return Tuple.Create(sw.ElapsedMilliseconds, endBytes - startBytes);
}
```

The results on my test machine are as follows:

	x86 avg time	x86 Task size	x64 avg time	x64 Task size
.NET 4	10 ms	44 bytes	12 ms	80 bytes
.NET 4.5	8 ms	40 bytes	10 ms	72 bytes
Improvement	20%	9%	17%	10%

Table 1: CreateTasks performance comparison, ntasks=100000, iterations=25

The benchmark results do indeed show the smaller footprint of a Task in .NET 4.5, in addition to the decreased amount of time that it takes to create Tasks.

Task<TResult> Creation

Let's compare the relative performance of .NET 4 and .NET 4.5 with respect to creating Task<TResult>s (which are also sometimes called "futures"). We'll us this benchmark to perform the comparison:

```
public static Tuple<long, long> CreateFutures(int ntasks)
{
    Task<int>[] tasks = new Task<int>[ntasks];
    Stopwatch sw = new Stopwatch();
    Func<int> function = () => 42;
    long startBytes = GC.GetTotalMemory(true);
    sw.Start();
    for (int i = 0; i < ntasks; i++) tasks[i] = new Task<int>(function);
    sw.Stop();
    long endBytes = GC.GetTotalMemory(true);
    GC.KeepAlive(tasks);
    return Tuple.Create(sw.ElapsedMilliseconds, endBytes - startBytes);
}
```

The results are as follows:

	x86 avg time	x86 Task<int> size	x64 avg time	x64 Task<int> size
.NET 4	15.2 ms	92 bytes	22.3 ms	168 bytes
.NET 4.5	7.7 ms	44 bytes	10 ms	80 bytes
Improvement	49%	52%	55%	52%

Table 2: CreateFutures performance comparison, ntasks=100000, iterations=25

These are pretty significant time and memory reductions with respect to Task<TResult> creation. The reduction in the number of fields and in the number of delegates created internally during Task<TResult> construction made a big difference.

TaskCompletionSource<TResult> Creation

Let's compare the creation time of a TaskCompletionSource<TResult> (using TaskCompletionSource<int>) relative to .NET 4. We'll use this benchmark to make the comparison:

```
public static Tuple<long, long> CreateTCSs(int ntasks)
{
    TaskCompletionSource<int>[] tasks = new TaskCompletionSource<int>[ntasks];
    Stopwatch sw = new Stopwatch();
    long startBytes = GC.GetTotalMemory(true);
    sw.Start();
    for (int i = 0; i < ntasks; i++) tasks[i] = new TaskCompletionSource<int>();
    sw.Stop();
    long endBytes = GC.GetTotalMemory(true);
    GC.KeepAlive(tasks);
    return Tuple.Create(sw.ElapsedMilliseconds, endBytes - startBytes);
}
```

The results are as follows:

	x86 avg time	x86 TCS<int> size	x64 avg time	x64 TCS<int> size
.NET 4	8 ms	72 bytes	16 ms	128 bytes
.NET 4.5	7.3 ms	56 bytes	8 ms	104 bytes
Improvement	9%	22%	50%	19%

Table 3: CreateTCSs performance comparison, ntasks=100000, iterations=25

You might ask, "How could a TaskCompletionSource<TResult> have a smaller memory footprint in .NET 4 than Task<TResult>, upon which it is built?" The answer is that TaskCompletionSource<TResult> passed a null delegate to Task<TResult>'s constructor, which eliminated the necessity of Task<TResult> creating some costly delegates. Now, in .NET 4.5, TaskCompletionSource<TResult>'s memory footprint is greater that Task<TResult>'s memory footprint (by an amount equal to the size of the smallest object possible in .NET), which is intuitive.

Continuation Attachment – Single

Let's examine the relative performance of single continuation attachment in .NET 4 and .NET 4.5. We made single continuation attachment a "sweet spot" in .NET 4.5, because, as previously mentioned, the most common uses of continuations involve attaching a single continuation to a Task. Attaching the initial continuation to a Task is accomplished in .NET 4.5 via a single Interlocked.CompareExchange operation. We'll measure the performance of single continuation attachment with the following benchmark, which sets up a continuation chain of **ntasks** Tasks:

```
public static Tuple<long, long> LinearContinuationChain(int ntasks)
{
    Task[] tasks = new Task[ntasks];
    Action<Task> action = _ => { };
```

```
    Stopwatch sw = new Stopwatch();
    long startBytes = GC.GetTotalMemory(true);
    sw.Start();
    tasks[0] = new Task(() => { });
    for (int i = 1; i < ntasks; i++)
            tasks[i] = tasks[i - 1].ContinueWith(action);
    sw.Stop();
    long endBytes = GC.GetTotalMemory(true);
    GC.KeepAlive(tasks);
    return Tuple.Create(sw.ElapsedMilliseconds, endBytes - startBytes);
}
```

The results are as follows:

	x86 avg time	x86 size-per-"link"	x64 avg time	x64 size-per-"link"
.NET 4	76.5 ms	212 bytes	81.8 ms	400 bytes
.NET 4.5	17 ms	64 bytes	18.7 ms	120 bytes
Improvement	78%	70%	77%	70%

Table 4: Single continuation attachment performance comparison, ntasks=100000, iterations=25

The speedup improvement and memory reduction are really good, and the results are not surprising because, as mentioned above, we optimized for single continuation attachment in .NET 4.5. Whereas in .NET 4 the first continuation on a Task involved the allocation of a ContingentProperties object and a List<TaskContinuation> object, in .NET 4.5 the first continuation on a Task requires only a single Interlocked.CompareExchange operation.

Continuation Attachment – Multiple

Let's examine the relative performance of multiple continuation attachment in .NET 4 and .NET 4.5. For our test, we'll attach two continuations per Task, as is it the second continuation attachment that incurs the most cost for our new .NET 4.5 continuation attachment scheme (subsequent continuations are added into the list that was allocated for the second continuation). We'll measure performance with the following benchmark:

```
public static Tuple<long, long> DualContinuationChain(int ntasks)
{
    Task[] tasks = new Task[ntasks];
    Stopwatch sw = new Stopwatch();
    Action<Task> action = _ => { };
    long startBytes = GC.GetTotalMemory(true);
    sw.Start();
    tasks[0] = new Task(() => { });
    for (int i = 1; i < ntasks; i++)
    {
        tasks[i] = tasks[i - 1].ContinueWith(action); // from prev link
        tasks[i - 1].ContinueWith(action); // "Continuation to nowhere"
    }
    sw.Stop();
    long endBytes = GC.GetTotalMemory(true);
    GC.KeepAlive(tasks);
    return Tuple.Create(sw.ElapsedMilliseconds, endBytes - startBytes);
}
```

The results are as follows:

	x86 avg time	x86 size-per-"link"	x64 avg time	x64 size-per-"link"
.NET 4	136 ms	304 bytes	134.9 ms	576 bytes
.NET 4.5	84 ms	184 bytes	88.2 ms	344 bytes
Improvement	38%	39%	35%	40%

Table 5: Dual continuation attachment performance comparison, ntasks=100000, iterations=25

The speedup and reduction of memory footprint here is not quite as dramatic as they were for single continuation attachment; in .NET 4.5, the second continuation added to a Task requires the allocation of a List<Object>. But there is a significant speedup and reduction of memory footprint in .NET 4.5 nonetheless.

Task.WaitAll and Task.WaitAny

As alluded to earlier, we've altered Task's waiting logic in .NET 4.5. The performance gain for this change is most apparent when waiting on multiple Tasks, such as when using Task.WaitAll and Task.WaitAny. Let's explore the extent of this performance boost with this benchmark code for Task.WaitAll:

```
public static Tuple<long, long> TestWaitAll(int ntasks)
{
    Task[] tasks = new Task[ntasks];
    Action action = () => { };
    for (int i = 0; i < ntasks; i++) tasks[i] = new Task(action);
    Stopwatch sw = new Stopwatch();
    long startBytes = GC.GetTotalMemory(true);
    sw.Start();
    Task.WaitAll(tasks, 1);
    sw.Stop();
    long endBytes = GC.GetTotalMemory(true);
    GC.KeepAlive(tasks);
    return Tuple.Create(sw.ElapsedMilliseconds, endBytes - startBytes);
}
```

The code above times the overhead of setting up a WaitAll for **ntasks** uncompleted Tasks, plus a one millisecond timeout. This test is admittedly less than perfectly precise, as the actual time before the WaitAll call times out could be anywhere from 1 millisecond to the scheduler quantum of the underlying operating system. Nevertheless, the test results still shed some light on the performance differences between .NET 4 and .NET 4.5 for this scenario:

	x86 time-per-WaitAll	x86 memory-per-WaitAll	x64 time-per-WaitAll	x64 memory-per-WaitAll
.NET 4	105.3 ms	6400000 bytes	83 ms	12000000 bytes
.NET 4.5	5 ms	36 bytes	5 ms	64 bytes
Improvement	95%	~100%	94%	~100%

Table 6: TestWaitAll performance, ntasks=100000, iterations=25

Even taking into account the inaccuracy around the one-millisecond timeout in the benchmark, it's clear to see that .NET 4.5 time and memory consumption for this benchmark show a vast improvement over .NET 4. The benchmark may seem slightly contrived – after all, how often do people wait for 100,000 Tasks to complete? However, the main point rings true regardless of the number of Tasks waited upon: In .NET 4, Task.WaitAll required the creation of a kernel synchronization primitive for each waited-upon Task, whereas in .NET 4.5, Task.WaitAll only has to allocate a single kernel synchronization primitive, which is then manipulated by continuations off of each waited-upon Task.

Similar tests can be done with Task.WaitAny, for which the benchmark code is exactly the same as it was for Task.WaitAll, except that Task.WaitAny is substituted for Task.WaitAll:

```
public static Tuple<long, long> TestWaitAny(int ntasks)
{
    Task[] tasks = new Task[ntasks];
    Action action = () => { };
    for (int i = 0; i < ntasks; i++) tasks[i] = new Task(action);
    Stopwatch sw = new Stopwatch();
    long startBytes = GC.GetTotalMemory(true);
    sw.Start();
    Task.WaitAny(tasks, 1);
    sw.Stop();
    long endBytes = GC.GetTotalMemory(true);
    GC.KeepAlive(tasks);
    return Tuple.Create(sw.ElapsedMilliseconds, endBytes - startBytes);
}
```

The results for Task.WaitAny are as follows:

	x86 time-per-WaitAny	x86 memory-per-WaitAny	x64 time-per-WaitAny	x64 memory-per-WaitAny
.NET 4	54.8 ms	12400000 bytes	36.8 ms	23200000 bytes
.NET 4.5	2.2 ms	80 bytes	3 ms	152 bytes
Improvement	96%	~100%	92%	~100%

Table 7: TestWaitAny performance, ntasks=100000, iterations=25

We can see that Task.WaitAny exhibits a performance boost in .NET 4.5 similar to that of Task.WaitAll, and for the same reason: WaitAny now just needs to allocate a single kernel synchronization primitive, whereas in .NET 4 it had to allocate a kernel synchronization primitive per waited-upon Task.

Best Practices

This section enumerates some of the performance lessons that we learned during the development of .NET 4.5.

Less is more – minimize memory allocations

As can be seen in the results of our benchmarks, there is a direct correlation between the amount of memory allocated in a test and the time taken for that test to complete. When viewed individually, memory allocations are not very expensive. The pain comes when the memory system occasionally cleans up unused memory, and this happens at a rate proportional to the amount of memory being allocated. So the more memory that you allocate, the more frequently the memory is garbage collected, and thus the worse your code's performance becomes.

Encode closure information into a Task's state object

One way to cut down on memory allocations is to avoid closure allocations by encoding closure information into a Task's state object. To understand what this means, consider the following code, where a Task adds items to a ConcurrentQueue:

```
ConcurrentQueue<int> cq = new ConcurrentQueue<int>();

...

Task t1 = Task.Factory.StartNew( () =>
{
    for (int i = 0; i < 10; i++) cq.Enqueue(i);
});
```

The use of **cq** inside of **t1**'s delegate means that the compiler will need to allocate a "closure" object to capture **cq**. The compiler will generate code that actually looks something like this:

```
private class Locals
{
    public ConcurrentQueue<int> _cq;
    public void Anonymous()
    {
        for (int i = 0; i < 10; i++) _cq.Enqueue(i);
    }
}

...

Locals _locals = new Locals();
_locals._cq = new ConcurrentQueue<int>();
Task t1 = Task.Factory.StartNew(new Action(_locals.Anonymous));
```

That's an extra allocation that you don't need! If you want to avoid the allocation of the compiler-generated closure class, then eliminate the use of captured variables in your Task delegate. This can be done via the use of the "state object" overload of Task.Factory.StartNew, like this:

```
ConcurrentQueue<int> cq = new ConcurrentQueue<int>();

...

Task t1 = Task.Factory.StartNew( state =>
{
    var mycq = (ConcurrentQueue<int>)state;
```

```
        for (int i = 0; i < 10; i++) mycq.Enqueue(i);
    }, cq);
```

In the example above, there are no variables captured by **t1**'s delegate, so no closure allocation is necessary[1].

This same technique can also be applied to continuations, as (beginning with .NET 4.5) there are also "state object" overloads of Task.ContinueWith. Consider this method:

```
public static Task<int> ResultOrZero(Task<int> task)
{
    TaskCompletionSource<int> tcs = new TaskCompletionSource<int>();

    task.ContinueWith( _ =>
    {
        if(task.Status == TaskStatus.RanToCompletion)
            tcs.SetResult(task.Result);
        else
            tcs.SetResult(0);
    });

    return tcs.Task;
}
```

The **ResultOrZero** method accepts a source Task<int> (**task**) and will return a proxy (**tcs.Task**) for that Task<int>. The proxy will complete when the source completes; if the source runs to completion, the proxy's result will be the source's result, otherwise the proxy's result will be zero. In the **ResultOrZero** method, the continuation delegate actually has *two* captured variables: **task** and **tcs**. We can remedy this by putting ContinueWith's antecedent and state variables to use:

```
public static Task<int> ResultOrZero(Task<int> task)
{
    TaskCompletionSource<int> tcs = new TaskCompletionSource<int>();

    task.ContinueWith( (antecedent, state) =>
    {
        var myTcs = (TaskCompletionSource<int>)state;
        if (antecedent.Status == TaskStatus.RanToCompletion)
            myTcs.SetResult(antecedent.Result);
        else
            myTcs.SetResult(0);
    }, tcs);

    return tcs.Task;
}
```

The continuation's delegate in the code above now has no captured variables, thus eliminating the need for closure allocation.

Use cached Tasks when possible

[1] It is also the case that eliminating variable capture in the delegate typically allows the compiler to cache the delegate, saving you yet another allocation. But I won't go into that too heavily here.

While we have strived to slim down Task and eliminate unnecessary allocations in .NET 4.5, the optimal allocation is one that never happens. Sometimes it is possible to cache a single instance of a frequently used Task, eliminating the need to continually re-allocate the Task. This is frequently useful in treating edge cases. Consider the following method that sums the results of an array of Task<int> once they've completed:

```
public static Task<int> WaitAndSum(Task<int>[] tasks)
{
    if ((tasks == null) || (tasks.Length == 0))
    {
        return Task.FromResult(0);
    }

    Task.WaitAll(tasks);
    int sum = 0;
    foreach (var task in tasks) sum += task.Result;
    return Task.FromResult(sum);
}
```

Note that, if the **tasks** array is null or empty, the method allocates a Task<int> with a result of 0 and returns it (Task.FromResult was added in .NET 4.5). Why not just cache a single copy of that Task as follows?

```
private static Task<int> s_zeroTask = Task.FromResult(0);

public static Task<int> WaitAndSum(Task<int>[] tasks)
{
    if ((tasks == null) || (tasks.Length == 0))
    {
        return s_zeroTask;
    }

    Task.WaitAll(tasks);
    int sum = 0;
    foreach (var task in tasks) sum += task.Result;
    return Task.FromResult(sum);
}
```

We've done some work in .NET 4.5 to enable Task caching. The main reason that you couldn't really do it in .NET 4 was that a consumer could Dispose() of the cached Task that you handed out, making it impossible for future consumers to do anything further with it (spawning a continuation off of it, waiting on it, etc.). In .NET 4.5, we've redesigned Task.Dispose such that even if Dispose is called on a cached Task, the Task can still be used for continuations, waits, and the like.

Know the causes of Task inflation

The ContingentProperties fields in Task will be allocated (thus "inflating" the Task) if any of the following occur:

- The Task is created with a CancellationToken

- The Task is created from a non-default ExecutionContext
- The Task is participating in “structured parallelism” as a parent Task
- The Task ends in the Faulted state
- The Task is waited on via ((IAsyncResult)Task).AsyncWaitHandle.Wait()

Now, note that the heading to this subsection was “Know the causes of Task inflation”, as opposed to “Avoid Task inflation at all costs”. There are definitely situations that call for the use of CancellationTokens, or non-default ExecutionContexts, or structured parallelism; these are valuable mechanisms that should definitely be employed when the scenario is benefited by them. The purpose of the list above is merely to give you a better understanding of the causes of Task inflation.

As to the final item in the list, it should rarely be necessary to use IAsyncResult.WaitHandle; it is much more efficient to register a callback with an asynchronous operation than it is to synchronously wait for one to complete.

Avoid concurrency measures in constructor paths

When writing thread-safe code, it is often the case that some sort of concurrency measure, say a lock or an Interlocked operation, is needed to protect access to a critical resource within the object. However, if the access to the critical resource is on the constructor path, then there is no possibility of contention for the resource[2], and therefore there is no need for expensive measures to protect against contention.

For example, consider the following class:

```
public class Foo
{
    private List<int> m_items;

    // Lazily initializes m_items, if necessary, in a thread-safe fashion
    private List<int> EnsureListInitialized()
    {
        var list = m_items;
        return (list != null) ?
            list :
            LazyInitializer.EnsureInitialized<List<int>>(
                ref m_items,
                () => new List<int>()
            );
    }

    public Foo(bool someCondition)
    {
        // ...
        if (someCondition) AddItem(0);
        // ...
    }

    public void AddItem(int item)
```

[2] Of course this is not true if the constructor publishes a reference to itself (e.g., “s_someStaticField = this;”), but such practices are error-prone and highly discouraged.

```
        {
            var list = EnsureListInitialized();
            lock (list) list.Add(item);
        }

    }
```

The code above has a nice abstraction (`AddItem`) for lazily allocating `m_items` and adding an item in a thread-safe fashion. Under certain conditions, the constructor takes advantage of this abstraction to add an item to the list. However, since there is no possibility of contending accesses to `m_items` during construction, why force the constructor to jump through concurrency hoops? Instead, the code can be refactored as follows:

```
public class Foo
{
    private List<int> m_items;

    // Lazily initializes m_items, if necessary, in a thread-safe fashion
    private List<int> EnsureListInitialized()
    {
        var list = m_items;
        return (list != null) ?
            list :
            LazyInitializer.EnsureInitialized<List<int>>(
                ref m_items,
                () => new List<int>()
            );
    }

    public Foo(bool someCondition)
    {
        // ...
        if (someCondition) AddItemInternal(0, false);
        // ...
    }

    public void AddItem(int item)
    {
        AddItemInternal(item, true);
    }

    private void AddItemInternal(int item, bool needsProtection)
    {
        if (needsProtection)
        {
            var list = EnsureListInitialized();
            lock (list) list.Add(item);
        }
        else
        {
            if (m_items == null) m_items = new List<int>();
            m_items.Add(item);
        }
    }
}
```

Such changes can yield much better performance for your constructor. In this case, when the Foo constructor is called and `someCondition` is true, the constructor no longer needs to employ any special concurrency protection when creating the `m_items` list and adding an element to it.

Provide fast (inline-able) versions of methods on the hot path

It is often the case that methods can be split into "simple hot path logic" and "longer cold path logic". In such cases, the hot path logic should be condensed to something sufficiently small so that it will be inlined by the JIT-compiler. For example, consider the following class:

```
class ThreadSafeObjectHolder
{
    private volatile object m_objectHolder;

    public ThreadSafeObjectHolder() { }

    public void AddObject(object obj)
    {
        // For the initial object, just store directly
        if ((m_objectHolder == null) &&
            (Interlocked.CompareExchange(ref m_objectHolder, obj, null) == null)
        )
        {
            return;
        }

        // At this point, m_objectHolder is guaranteed to be non-null
        object temp = m_objectHolder;
        if (!(temp is List<object>))
        {
            var newList = new List<object>(); // Allocate a list
            newList.Add(temp); // Add the previously "single" element

            // Convert m_objectHolder to list
            Interlocked.CompareExchange(ref m_objectHolder, newList, temp);
        }

        // At this point, m_objectHolder is guaranteed to be a list
        var list = m_objectHolder as List<object>; // resample and cast
        lock (list)
        {
            list.Add(obj);
        }
    }
}
```

This ThreadSafeObjectHolder class can hold no objects, a single object, or a list of objects. (In the real world, there would also need to be a method to retrieve the object(s) being held, but we'll omit that here for brevity. And, yes, I'm assuming that only non-null objects are being stored.) ThreadSafeObjectHolder allows for a very efficient way to store a single object, but has the flexibility to store multiple objects.

Suppose that the most common use case for ThreadSafeObjectHolder was to store a single object. The "hot path" would then be the block of code at the top of the `AddObject` method:

```
    // For the initial object, just store directly
    if ((m_objectHolder == null) &&
        (Interlocked.CompareExchange(ref m_objectHolder, obj, null) == null)
    )
    {
        return;
    }
```

Just by inspection, it's easy to see that `AddObject` is not going to be inlined by the JIT – it's just too complex. However, we could refactor the method into two methods: an inline-able method for the "hot" path, and a less-likely-to-be-inlined method for the "cold" path.

```
public void AddObject(object obj)
{
    // For the initial object, just store directly
    if ((m_objectHolder != null) ||
        (Interlocked.CompareExchange(ref m_objectHolder, obj, null) != null)
    )
    {
        AddObjectComplex(obj); // cold path
    }
}

private void AddObjectComplex(object obj)
{
    // At this point, m_objectHolder is guaranteed to be non-null
    object temp = m_objectHolder;
    if (!(temp is List<object>))
    {
        var newList = new List<object>(); // Allocate a list
        newList.Add(temp); // Add the previously "single" element

        // Convert m_objectHolder to list
        Interlocked.CompareExchange(ref m_objectHolder, newList, temp);
    }

    // At this point, m_objectHolder is guaranteed to be a list
    var list = m_objectHolder as List<object>; // resample and cast
    lock (list)
    {
        list.Add(obj);
    }
}
```

How do we know that `AddObject` is now being successfully inlined? That is admittedly somewhat of a dark art! The most robust way to test this is via ETW events output by the JIT compiler, as these will inform you as to which methods were inlined, which weren't, and why they weren't. However, I chose a simpler approach; I added this line at the end of `AddObjectComplex`:

```
    Console.WriteLine("Stack trace: \n{0}", Environment.StackTrace);
```

I then called `AddObject` two times on the same ThreadSafeObjectHolder object. The stack trace printed out from the second call was this:

> Stack trace:
>
> at System.Environment.GetStackTrace(Exception e, Boolean needFileInfo)
>
> at System.Environment.get_StackTrace()
>
> at Samples.ObjectHolder2.AddObjectComplex(Object obj) in c:\Users\joehoag\Documents\Visual Studio Dev11\Projects\Sample\Sample\Program.cs:line 567
>
> at Samples.Program.Main(String[] args) in c:\Users\joehoag\Documents\Visual Studio Dev11\Projects\Sample\Sample\Program.cs:line 648

Hmmm… I see Main(), from which I called `AddObject`, and I see AddObjectComplex() – what happened to the `AddObject` call? It was inlined, so it never showed up in the stack trace. Mission accomplished!

Making your hot paths inline-able in this fashion can have a measurably positive effect on your code's performance. I wrote the following code to test the performance of adding a single object to a ThreadSafeObjectHolder:

```
object storedObject = new object();
ThreadSafeObjectHolder[] objectHolders =
     new ThreadSafeObjectHolder[500000];
for (int j = 0; j < 500000; j++)
     objectHolders[j] = new ThreadSafeObjectHolder();
Stopwatch sw = Stopwatch.StartNew();
for (int j = 0; j < 500000; j++)
     objectHolders[j].AddObject(storedObject);
sw.Stop();
```

On my machine, this test took an average of about 25799 ticks before the inlining, while the inlined version took an average of about 16798 ticks. Obviously, the inlining made a difference.

Consider using await instead of ContinueWith/Unwrap

Support for the new "await" keyword is built on top of Task continuations, but the await support is highly optimized and generally much more efficient than an "off the shelf" continuation. So those of you already familiar with Task continuations should consider using await, where applicable, instead of ContinueWith and Unwrap. And "where applicable" can generally be interpreted as "in any linear chain of async methods where you don't need the returned Task instance for each continuation".

Consider a method that needs to call three asynchronous methods – AsyncMethod1, AsyncMethod2 and AsyncMethod3. Using .NET 4, before async/await support became available, you would code this up with ContinueWith and Unwrap as follows:

```
public static Task AsyncMethod1() { ... }
public static Task AsyncMethod2() { ... }
```

```
public static Task AsyncMethod3() { ... }

public static Task AsyncCallingMethod()
{
    Task returnValue = AsyncMethod1()
    .ContinueWith(_ => AsyncMethod2()).Unwrap()
    .ContinueWith(_ => AsyncMethod3()).Unwrap();

    return returnValue;
}
```

With the advent of async/await support in .NET 4.5, you can now code this up much more simply:

```
public static async Task AsyncCallingMethod()
{
    await AsyncMethod1();
    await AsyncMethod2();
    await AsyncMethod3();
}
```

Not only is the second version of `AsyncCallingMethod` easier to read and reason about, but it is also implemented more efficiently behind the scenes than is the first version. How much more efficiently? I used this code for measuring the respective performance of await and ContinueWith/Unwrap:

```
// A "fake" async method, measures system overhead
internal static Task AsyncMethod()
{
    return Task.Factory.StartNew(() => { });
}

// Set up an await chain ntasks long
internal static async Task DoAwaits(int ntasks)
{
    for (int i = 0; i < ntasks; i++) await AsyncMethod();
}

// Set up a ContinueWith/Unwrap chain ntasks long
internal static Task DoContinuations(int ntasks)
{
    Task curr = AsyncMethod();
    for (int i = 1; i < ntasks; i++)
            curr = curr.ContinueWith(_ => AsyncMethod()).Unwrap();
    return curr;
}
```

And the results were as follows on my system using .NET 4.5:

	x86 avg. time	x86 avg. memory	x64 avg. time	x64 avg. memory
DoContinuations	256.3 ms	14.5 MB	280.1 ms	26.7 MB
DoAwaits	143.9 ms	3.3 MB	149.2 ms	3.6 MB
Improvement	44%	77%	47%	87%

Table 8: Performance results for async method chaining, ntasks=100000, 25 iterations

From these results, it is clear that use of the “await” keyword to handle async method call chains is much more efficient than trying to do the same thing with ContinueWith and Unwrap.

Closing Thoughts

I hope that you’ve enjoyed this tour of the performance improvements that we’ve made to TPL in .NET 4.5, as well as some of the lessons that we’ve learned about writing efficient .NET code while developing TPL. Please take .NET 4.5 for a spin and let us know what you think!