Performance

Extract Dlls

1. .net Reflector

2. ILSPY

Visual Tree and Logical Tree

1.Snoop

2.WPF Inspector

3.XMLSPY

performanc tool

1.Microsoft performance Analyze



2.WPF Performance Suite

3**.**[**ANTS Performance Profiler Pro**](http://www.red-gate.com/products/dotnet-development/ants-performance-profiler/) **,** [**ANTS Memory Profiler**](http://www.red-gate.com/products/dotnet-development/ants-memory-profiler/)

1. For loop is faster than foreach loop

2. **Choose your data type before using it**

3.**Choose when to use a class and when to use a structure** (structure is faster)

4.**Always use Stringbuilder for String concatenation operations**

5.Before assigning a value to your class variable, I suggest you look at the following code and output screen at this point.

Public class Test{

public staticstring Name {get;set; }

public static String surname;

}

6. Managed Memory are Dispose in Garbage Collection of the destructor of that Calss.

For Unmanaged Memeory

Implement IDispose Method in Classs Remove unmanaged memeory from dispose method

eg. MemoryStream, com Connected

7. using(Textwriter obj=new TextWriter()) key

8.weakRefreence

9. use --> return age >= 21 ? "Congratulations! You are eligible to buy alcohol." :

"Sorry! You're not eligible to buy alcohol.";

10 use Convert.ToInt32(uservalue);

not use j = (int)money;

11. **Modern async await**

class ProgramB

{

public ProgramB() { }

public async void PrintAsync()

{

int a = await GetValueAsync();

Console.WriteLine("The Multiplication of one" +

" random number with itself is: {0}", a \* a);

}

private async Task<int> GetValueAsync()

{

int i = ReturnRandomNumber();

return await Task.FromResult<int>(i);

}

private int ReturnRandomNumber()

{

return new Random().Next(1, 1000);

}

}

12. use Task.Run() , not use Task.Factory.Run()

13.The **as** operator is similar to a cast operation; however, there are two advantages to using the as operator.

* It makes your code more readable.
* If a type mismatch occurs, the object will become null instead of throwing an exception.

XAML Performance

[Fix Binding Errors](http://pelebyte.net/blog/2011/07/11/twelve-ways-to-improve-wpf-performance/#FixBindingErrors)

I. [Hard-code widths and heights where possible](http://pelebyte.net/blog/2011/07/11/twelve-ways-to-improve-wpf-performance/#OptimizeLayout)

II. [Avoid CollectionView.Grouping](http://pelebyte.net/blog/2011/07/11/twelve-ways-to-improve-wpf-performance/#AvoidGrouping)

III. [Optimize bindings to collections that change](http://pelebyte.net/blog/2011/07/11/twelve-ways-to-improve-wpf-performance/#BetterObservableCollection)

IV. [Avoid DynamicResources](http://pelebyte.net/blog/2011/07/11/twelve-ways-to-improve-wpf-performance/#DynamicResource)

V. [Avoid ResourceDictionary](http://pelebyte.net/blog/2011/07/11/twelve-ways-to-improve-wpf-performance/#ResourceDictionary)

VI. [Simplify your visual tree](http://pelebyte.net/blog/2011/07/11/twelve-ways-to-improve-wpf-performance/#VisualTree)

VII. [Be wary of System.Windows.Interactivity.Behavior<T>.OnDetaching](http://pelebyte.net/blog/2011/07/11/twelve-ways-to-improve-wpf-performance/#BehaviorOnDetaching)

VIII. [Do not use DependencyPropertyDescriptor for any reason…ever](http://pelebyte.net/blog/2011/07/11/twelve-ways-to-improve-wpf-performance/#DependencyPropertyDescriptor)

IX. [Be careful of viewmodel events](http://pelebyte.net/blog/2011/07/11/twelve-ways-to-improve-wpf-performance/#ViewModelEvents)

X. [Batch up Dispatcher.BeginInvoke](http://pelebyte.net/blog/2011/07/11/twelve-ways-to-improve-wpf-performance/#Dispatcher)

XI. [In general, beware of memory leaks](http://pelebyte.net/blog/2011/07/11/twelve-ways-to-improve-wpf-performance/#MemoryLeaks)

I. Fix Binding Errors and Exceptions

Every time a binding error occurs, your app hangs for just a split second as it writes outerrors to the trace log. If you have a lot of binding errors, then those split seconds start to add up. Make sure to go through your bindings, especially those on ItemsControls (ListViews, custom grids, etc.) and verify that there are no binding errors.

Open up your app in the debugger and play around, especially where there is slowness. Make sure all bindings resolve without errors.

[RelativeSource](http://msdn.microsoft.com/en-us/library/system.windows.data.relativesource.aspx) in DataTemplates may also result in bindings that break initially, but then later resolve properly. Be wary of them, and try to use inherited attached properties instead of relying on RelativeSource in DataTemplates.

**Viewmodel bindings**

1. Make sure that your views and view models are in sync. Use [ReSharper 6](http://www.jetbrains.com/resharper/features/xaml_editor.html) to help you find broken bindings.

2. If you’re binding to a collection of objects with mixed types, add different DataTemplates so that none of them refer to non-existent properties.

3. Make sure that your converters aren’t throwing exceptions. These have a cost too.

**View-based RelativeSource bindings**

1. When using ListBoxes and ListViews, it’s a common problem to have [this problem](http://stackoverflow.com/questions/160391/listbox-with-grid-as-itemspaneltemplate-produces-weird-binding-errors). Avoid RelativeSource.FindAncestor expressions at all cost here, because the deferred behavior of templates cause the object and its bindings to be created (and resolved) before the ListBoxItem/ListViewItem is added to the visual tree.

2. An alternative is to define an attached dependency property on the ListBoxItem/ListViewItem, and use property inheritance to give your child items the necessary property values. This essentially pushes property values down the visual tree instead of searching up.

II. Hard-code sizes where possible

This may not always be a practical or desirable solution, but layout passes perform faster when widths and heights do not have to be recalculated. They may also help stop a layout pass from rippling through an entire visual tree.

And always set specific widths on columns in a grid (be it a ListView + GridView or any third-party control), because these tend to be very expensive, especially with larger data sets.

III. Avoid CollectionView.Grouping

Grouping in WPF doesn’t perform terribly well, especially with ListViews and GridViews. Create a collection with mixed viewmodel types–your original collection, and one that represents the “group”. Use DataTemplates to change the appearance of your “group” objects.

For example, if you have a PersonViewModel class with a property that you want to group by (let’s say Region), it is faster to create a mixed collection of MyGroupViewModel and PersonViewModel objects, ordered correctly by group, with different DataTemplates, than it is to bind to a grouped collection. Unfortunately, it’s a lot more work.

IV. Optimize bindings to collections that change

Repeatedly calling [ObservableCollection<T>.Add](http://msdn.microsoft.com/en-us/library/ms132404.aspx) when the collection is data-bound can be a prohibitively expensive operation, especially with thousands of rows. Unfortunately, the framework provides no easy, satisfactory fix.

**Fix 1: Use ObservableCollection as-is, but break bindings**

1. Break the binding to the collection.

2. Update the collection while not data-bound.

3. Re-bind.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37 | *// some methods removed for brevity*  **public** **partial** **class** MyViewModel : INotifyPropertyChanged  {  **private** ObservableCollection<T> \_people;    **public** IList People  {  get { **return** \_people; }  **private** set  {  **if** (\_people != value)  {  \_people = value;  OnPropertyChanged("People");  }  }  }    **void** BatchAddPeople(IEnumerable<Person> newPeople)  {  var currentPeople = \_people;    *// stop WPF from listening to the changes that we're about*  *// to perform*  **this**.People = **null**;    *// change*  **foreach** (var person **in** newPeople)  {  currentPeople.Add(person);  }    *// cause WPF to rebind--but only once instead of once for*  *// each person*  **this**.People = currentPeople;  }  } |

**Fix 2: Use the older .NET 2.0-era collections instead**

1. Use [System.ComponentModel.BindingList<T>](http://msdn.microsoft.com/en-us/library/ms132679.aspx) (from the old days) instead; it has an API for suppressing change notifications.

**Fix 3: Reimplement ObservableCollection.**

1. Create your own collection that implements INotifyCollectionChanged.

 Raise INotifyCollectionChanged as sparingly as you can.

 Raise the event with a NotifyCollectionChangedAction.Reset event for anything more trivial than a simple single-item add, remove, change, or move. Do not take advantage of the NotifyCollectionChangedEventArgs constructors that take collections of items; you will find support for it spotty at best.

2. Implement System.Collections.IList on your collection. WPF does not use the generic System.Collections.Generic.IList<T> interface; it is completely ignored. If you don’t implement the interface, WPF will perform almost all operations (including accessing rows by number!) strictly by the IEnumerable implementation, and it won’t be very optimal or fast about it.

3. (Probably) implement System.Collections.Generic.IList<T> as well. WPF doesn’t use it, but you probably will (through LINQ, Rx, etc.)

V. Avoid DynamicResources

Even in .NET 4.0, DynamicResource access is still slower than StaticResource access. And worse, once you start nesting DynamicResources (for example, a ListView whose Style contains a ControlTemplate that references objects through DynamicResources), you’re likely to run into situations where you leak controls.

VI. Avoid ResourceDictionary

This advice is practically impossible to follow, but do your best. There is a huge cost in constructing ResourceDictionaries, and depending on where you place them, you are probably constructing many more objects than you realize.

A common, sensible, and logical pattern is to keep usages of elements as close to where you use them as possible. Many people place resources in UserControl.Resources, or break up their themes into multiple ResourceDictionaries for clarity and separation. Although this is arguably good programming practice, it also tends to be insanely slow. If your windows/controls or your ListBoxItem/ListViewItems in a ListBox/ListView are coming up more slowly than you would like, it’s probably a combination of too much ResourceDictionary construction and/or DynamicResources. (Yes, even in .NET 4.0.) Collapse ResourceDictionaries as much as you can, and try to ensure that the contents of these dictionaries is only loaded once. The easiest and surest way is to include it in the resources of your System.Windows.Application object, almost always difficult or infeasible for composite applications.

I have also frequently taken to creating static classes that contain nothing but highly reusable resources (think static framework classes like the [Brushes](http://msdn.microsoft.com/en-us/library/system.windows.media.brushes.aspx) class) because it’s easier to guarantee that objects are only being created once, and hopefully at app startup instead of triggered by the user interacting with the application and forcing a lazy load at an undesirable time. Not necessary the healthiest design, but the performance is quite a bit better.

Using [implicit ControlTemplate/DataTemplate styles](http://msdn.microsoft.com/en-us/library/ms750613.aspx#stylesimplicitkeys) will also help keep your code and XAML organized without the need for either StaticResource or DynamicResource.

VII. Simplify your visual tree

Shallow visual trees are better than deeper visual trees. Again, ItemsControls will usually exacerbate performance problems with deep visual trees because if they’re not being virtualized, they’re being destroyed and recreated; if they are being virtualized, changing DataContext in a deeper tree takes more time than changing DataContext in a shallower one.

VIII. Be wary of System.Windows.Interactivity.Behavior<T>.OnDetaching

Sadly, [System.Windows.Interactivity.Behavior<T>.OnDetaching](http://msdn.microsoft.com/en-us/library/system.windows.interactivity.behavior.ondetaching(v=expression.40).aspx) will generally not get called. Put a breakpoint and see for yourself.

[OnAttached](http://msdn.microsoft.com/en-us/library/system.windows.interactivity.behavior.onattached(v=expression.40).aspx) signals the addition of a behavior to a control (generally at instantiation of your XAML); OnDetaching signals the removal of a behavior from a control (generally never, as behaviors don’t get removed from controls). Don’t put sensitive disposing behavior in OnDetaching. The Unloaded event is a better place for that, but be aware that it will get raised every time the control is removed from the visual tree.

IX. Do not use DependencyPropertyDescriptor for any reason…ever

[DependencyPropertyDescriptor.AddValueChanged](http://msdn.microsoft.com/en-us/library/system.componentmodel.dependencypropertydescriptor.addvaluechanged.aspx) classes cause the WPF framework to take a strong reference to the source of the event that isn’t removed until you call[DependencyPropertyDescriptor.RemoveValueChanged](http://msdn.microsoft.com/en-us/library/system.componentmodel.dependencypropertydescriptor.removevaluechanged.aspx). This class is frequently used in conjunction with Behaviors, so if you have RemoveValueChanged in OnDetaching, you’re likely leaking memory. Because of WPF’s references to your objects, it is not just enough to drop references to your view and view model.

A better alternative is to rely on data binding where you can; create a DependencyProperty for the sole purpose of listening to changes on your target property, and use the change notifications in DependencyProperty in order to listen to changes on the target property. It keeps the observed object (generally, your view model) from accidentally holding a strong reference to your view.

X. Be careful of view model events

If your views or behaviors rely on events being raised from a viewmodel (as innocuous as[INotifyPropertyChanged.PropertyChanged](http://msdn.microsoft.com/en-us/library/system.componentmodel.inotifypropertychanged.aspx) or[INotifyCollectionChanged.CollectionChanged](http://msdn.microsoft.com/en-us/library/system.collections.specialized.inotifycollectionchanged.aspx)), subscribe to them weakly. Viewmodels tend to have a longer lifetime than views (consider a virtualized ItemsControl), so it’s possible that your view model will inadvertently gather references to views. Use classes like[PropertyChangedEventManager](http://msdn.microsoft.com/en-us/library/system.componentmodel.propertychangedeventmanager.aspx) or [CollectionChangedEventManager](http://msdn.microsoft.com/en-us/library/system.collections.specialized.collectionchangedeventmanager.aspx), or (painfully) use the[WeakEventManager](http://msdn.microsoft.com/en-us/library/system.windows.weakeventmanager.aspx) to create your own event manager for your custom events. It’s painful, but usually necessary in order to prevent view models from taking references to views.

XI. Batch up Dispatcher.BeginInvoke

If your application displays data from a network, you’re probably using background threads to accomplish the task (which is good). However, you’re probably not consciously counting your [Dispatcher.BeginInvoke](http://msdn.microsoft.com/en-us/library/system.windows.threading.dispatcher.begininvoke.aspx) calls (which is not as good). The more you’re able to coalesce multiple Dispatcher.BeginInvokes into a single call, the more likely WPF will be able to help you out by making single subsequent layout and render passes, and the lower your overall CPU usage.

To be fair, WPF is much better at trying to help you here than VB6/WinForms—you won’t often see apps that dance and flicker incessantly any more—but there is still a non-zero cost to updating the screen, and if you have a particularly large application, this could be a problem for you.

XII. In general, beware of memory leaks

This is a bit of a generalization of the last few points, but memory leaks make apps behave worse over time. Fixing memory leaks goes a long way in fixing the performance of an application. And since most developers are constantly restarting WPF apps as they work on them, they often go undetected until the software is delivered.

**Lower The Bitmapscalingmode to Render Images Faster**

You can lower the consumption of resources on a machine when you have certain animations beingprocessed by your WPF application. To do this, you need to use the BitmapScalingMode property of theRenderOptions object.

You would need to use the "LowQuality" option from the BitMapScalingMode enum to ensure that the image is processed using the speed algorithm instead of the default high-quality image re-sampling algorithm.

The following code snippet shows you how to do that:

RenderOptions.SetBitmapScalingMode(imageObject,BitmapScalingMode.LowQuality);

**Use The Right Elements in the Right Places**

We need to use the right elements in the right places. Avoid UIElements as child or nested controls when you build the tree. The best example is the Flow Document. We often use the TextBlock element inside theFlowDocument.

<FlowDocument>  <Paragraph>    <TextBlock>some text</TextBlock>  </Paragraph> </FlowDocument>

Instead of doing above, we can rewrite the XAML content as shown below. The Run element is not aUIElement and involves lesser overhead while rendering.

<FlowDocument>  <Paragraph>    <Run>some text</Run>  </Paragraph> </FlowDocument>

A similar example is the usage of Content property of the Label Control. If this content is updated more than once in its lifetime and is a string, this databinding procedure can hinder the application's performance. Since the Content is a string, it will be discarded and recreated during the databinding. Use a TextBlock in such cases and data bind to its Text property.

The unnecessary elements in the visual tree also contribute to the slowness of WPF applications. You should ideally combine the layout and optimize the default control templates.

**Increase Usage of Static Resources**

Static resources are pre-defined resources that can be hooked to XAML properties. It is similar to Compile-time tie up's and does not have a performance impact. The dynamic resources on the other hand involve a run-time seek and also construction of such objects which lead to performance impacts. Doing it this way also enables you to share common resources like brushes.

Also note that the static resources need to be present at compile time.

Read the article on XAML Static Resources at DevX ([Simplay XAML with Static Resources](http://www.devx.com/dotnet/Article/42304) for a deeper insight into usage of Static Resources. The following XAML snippet is from this article.

A Static resource can be referenced in the following manner:

<Button Template="{StaticResource RoundButtonWithThickEdge}"   x:Name="button1" Content="Button 1" > </Button>

The following code snippet shows definition of the Static resource: RoundButtonWithThickEdge

<ControlTemplate    x:Key="RoundButtonWithThickEdge"    TargetType="{x:Type Button}">    <Grid>      <Ellipse Fill="{TemplateBinding Background}"        Stroke="{x:Null}"        HorizontalAlignment="Stretch" x:Name="ellipse"/>       <ContentPresenter HorizontalAlignment="Center" VerticalAlignment="Center"/>       <Ellipse Stroke="{x:Null}" Margin="2,3,4,5">         <Ellipse.Fill>           <LinearGradientBrush EndPoint="0.5,1" StartPoint="0.5,0">             <GradientStop Color="#FFFBFAFA" Offset="0"/>            <GradientStop Color="#1DFFFFFF" Offset="1"/>          </LinearGradientBrush>        </Ellipse.Fill>      </Ellipse>    </Grid>  </ControlTemplate>

**Use Controls with UI Virtualization When You Display Large Data**

Imagine binding a combo box with large number of rows. Would it not make the rendering of the items in the combo box pretty slow? The slowness is caused by the time taken to compute the position of each item in such a situation. With WPF, you can defer this behavior.

This is called as UI Virtualization and it simply refers to the item container generation only on its visibility.

To achieve this, you need to set the IsVirtualizing property to true for such controls. For example, Listbox is a control which often is bound with a large set of data and is a primary candidate for UI virtualization. Other examples include Combobox, ListView, and TreeView.

**Use Deferred Scrolling to Enrich User Experience**

If you remember scrolling the datagrid or a listbox, it often slows down the entire application because of the continuous updates that are invoked forcibly due to the scrolling. This is the behavior by default. In such cases, we can use the "Deferred Scrolling" property of the controls to enrich the user experience.

All you have to do is set the IsDeferredScrollingEnabled attached property to true.

**Use Font Cache Service to Improve the Start-up Time**

The WPF applications can share the font data among them. This can be achieved through the Windows service named PresentationFontCache Service. This is automatically started with windows.

You can find this service under the Services panel (Type "Services.msc" in your run menu with the double quotes, and check for this service) and ensure that is started.

**Use Unloaded Event to Unload Unnecessary Animations**

Animations definitely take a toll on the resource utilization, and would also increase if they are not disposed of in the right manner. You should dispose of them when you consider them useless. Failing to do so, will consume precious resources until Mr. Garbage Collector kicks in.

For example to remove a story board, use the Remove method of the Story board in the Unload Event. The following example from MSDN shows this:

<EventTrigger RoutedEvent="Page.Unloaded" >        <EventTrigger.Actions>          <RemoveStoryboard BeginStoryboardName="myBeginStoryboard" />        </EventTrigger.Actions> </EventTrigger>

**Use Container Recycling to Increase Performance**

You can increase the performance by recycling the containers that perform the Virtualization. The following code snippet sets the ViruatlizationMode to "Recycling" which allows you to gain more performance.

This forces the container objects to be reused when the user scrolls and reaches another item.

settingVirtualizingStackPanel.VirtualizationMode="Recycling"

**Predict the Graphics Capability and Provide Features**

Use the RenderCapability.Tier property to determine if the machine supports hardware acceleration, partial hardware acceleration, or no acceleration:

The sample code below shows you how to check the Tier:

int displayTier = (System.Windows.Media.RenderCapability.Tier > 16) if (displayTier == 0) {   //no hardware acceleration } else if (displayTier == 1) {   //partial hardware acceleration } else {   //supports hardware acceleration }

After you determine this, you can then selectively choose those capabilities that work best on the user's hardware.

**Use WPF Profiling Tools to Profile a WPF Applicaton**

Profiling a WPF application is an important step to understanding its behavior. There are lots of tools in the market for profiling WPF applications.

**Top 11 WPF Performance Tips**

Windows Presentation Foundation provides a very confortable way to develop rich user experiences. A drop shadow for example can added by inserting two simple lines of XML. But this simplicity can also mislead us to overuse them. This leads to performance issues. The following tipps may help you to avoid or fix them.

1. **Dispatch expensive calls** either within the UI thread with a lower DispatcherPriority by callingDispatcher.BeginInvoke() or to a background thread by using a BackgroundWorker to keep the UI responsive.

2. **Fix binding errors** because they consume a lot of time, trying to resolve the path error, including searching for attached properties. You can find them by looking for System.Windows.Data Error in the Visual Studio output log.

3. **Reduce the number of visuals** by removing unneeded elements, combining layout panels and simplifying templates. This keeps the memory footprint small and improves the rendering performance.

4. **Prevent Software Rendering**. The use of transparent windows by setting AllowsTransparency to true or using old BitmapEffects can cause WPF to render the UI in software on Windows XP, which is much slower.

5. **Load resources when needed**. Even thow it's the most comfortable way to merge all resources on application level it can also cost performance by loading all resources at startup. A better approach is to load only often used resources and load the other on view level.

6. **Virtualize lists and views** by using a VirtualizingStackPanel as ItemsPanel for lists. This only creates the visible elements at load time. All other elements are lazy created when they get visible. Be aware that grouping orCanContextScrol="True" prevents virtualization!

7. **Enable Container Recycling**. Virtualization brings a lot of performance improvements, but the containers will be disposed and re created, this is the default. But you can gain more performance by recycle containers by settingVirtualizingStackPanel.VirtualizationMode="Recycling"

8. **Freeze Freezables** by calling Freeze() in code or PresentationOptions:Freeze="true" in XAML. This reduces memory consumption and improves performance, because the system don't need to monitor for changes.

9. **Disable Assembly localization** if you don't need it. By using the [NeutralResourcesLanguageAttribute].This prevents an expensive lookup for satelite assemblies

10. **Lower the framerate of animations** by setting Storyboard.DesiredFrameRate to lower the CPU load. The default is 60 frames/second

11. **Use StreamGeometries instead of PathGeometries** if possible to draw complex 2D geometries, because they are much more efficient and consume less memory.