Improving Scrolling Performance in Windows Presentation Foundation

Cedric Dussud  
Microsoft Corporation

**Summary:** There are several ways to improve the scrolling performance of large data sets in Microsoft Windows Presentation Foundation (WPF). In order of their increasing performance, they are: using UI virtualization, recycling UI containers, and implementing a data-binding proxy object. (9 printed pages)

# Introduction

This article tries to help answer the following question: “I'm displaying many items in an **ItemsControl**. How can I make scrolling/panning faster?”

Although Microsoft Windows Presentation Foundation (WPF) does its best to be as fast as possible, it is necessary, in some cases, to write some things yourself. This article represents a guide to how to do that; it describes the performance bottlenecks that you'll see, and suggests ways to get around them.

This article is divided into sections, each of which briefly describes an implementation, along with its performance issues. Each subsequent implementation builds on the last and is faster, but requires more work on the part of the user (that is, you). The suggestions here apply equally well to 1-D (**StackPanel**) and 2-D layouts.

There is also a sample application that provides an implementation of each performance optimization that is discussed (see the "Sample Application" section).

There are several ways to improve the scrolling performance of large data sets in Microsoft Windows Presentation Foundation (WPF). In order of their increasing performance, they are:

* Using UI virtualization.
  + On by default in **ListView** and **ListBox**.
  + Trivial to enable in **ComboBox**.
  + Not implemented in **TreeView**. User can write a **ListView** that acts like a **TreeView** (see the "Resources" section for more info).
* Recycling UI containers.
  + See the **ContainerList** class in the provided sample application for a reference implementation.
* Implementing a data-binding proxy object.
  + See the **ContainerListFastBinding** class in the provided sample application for a reference implementation.

# Non-Virtualized UI

Because this article discusses a series of more performant ways to display items, it is best to start with the simplest implementation as a reference point. This is mostly theoretical, because WPF supports UI virtualization (see the next section) by default in **ListBox** and **ListView**.

### Background

A *panel* in WPF is an element that hosts UI children and lays them out. That's a panel's only job; someone gives it a list of children, and it decides how to size and position them properly. A **StackPanel**, for example, stacks children either from top to bottom or from left to right.

**ItemsControl**s in WPF contain a set of items for display; typically, this is a list of data items that are specified by the **ItemsSource** property. The **ItemsControl** is (effectively) responsible for creating visuals to display each data item and for placing those visuals in a panel.

The simplest way for an **ItemsControl** to display children is to create one container for each data item and to give the entire set to a panel. The panel then has everything that it needs to handle the sizing, positioning, and scrolling of items. For example, by measuring all of the containers, it knows the scroll extent.

### Implementation

Although WPF by default virtualizes most of its **ItemsControl**s, one could implement this by creating an **ItemsControl** that has its **ItemsPanel** set to a **StackPanel**.

### Performance Bottlenecks

The problem with the standard implementation is that, for large data sets, the **StackPanel** will generate and lay-out containers that are likely off-screen. This is costly in two ways:

1. The application will instantiate containers that aren't visible, which wastes memory.
2. Each layout pass computes the size and position of all of the containers, which makes scrolling expensive.

Because both of these costs are proportional to the number of data items, this just doesn't scale for large data sets. The best way to resolve this issue is with UI virtualization.

# UI Virtualization

*UI virtualization* means deferring container generation until it is needed. For example, say that we have a **ListBox** that shows a list of 2,000 items in which only 20 are visible at a given time. **ListView** (technically, the **VirtualizingStackPanel** that is underneath) is able to generate containers the moment that they are scrolled into view. The panel also removes containers that no longer are needed, so that it will use around 20 containers. For large data collections, the scrolling-performance benefits are significant.

### Suggested Implementation

Both **ListBox** and **ListView** already implement UI virtualization. Because most large data sets in WPF are displayed in **ListBox**es or **ListView**s, this is the effective starting point for most applications.

**ComboBox** has it off, by default, but it is fairly trivial to enable it. Just replace its default **ItemsPanel** with a **VirtualizingStackPanel**, and set the panel's **IsVirtualizing** property to **true**.

**TreeView** does not implement UI virtualization. Currently, the best workaround is to create a **ListView** that behaves like a **TreeView** (that is, it knows how to add and remove children, and indent them properly). For a great discussion of **TreeView** performance issues and ways in which to solve them, see Beatriz Costa's blog post: <http://www.beacosta.com/blog/?p=42>

### Performance Bottlenecks

**VirtualizingStackPanel** in WPF creates items as they're scrolled into view, and cleans them up when they are moved off-screen. The problem is that expanding data templates is very expensive; and, now that layout is doing far less work, a significant portion of the scrolling time is now spent creating containers.

|  |  |
| --- | --- |
| **Contribution (percent)** | **Operation** |
| 30 | Creating containers (expanding data templates and item templates) |
| 18 | Measuring |
| 15 | Arranging |
| 10 | Removing old containers from the panel |
| 10 | Rendering containers |
| 10 | Data binding |
| 6 | Adding new containers to the panel |

Table . Performance of ListBox

The most obvious way to improve the cost of container creation is to get rid of it wherever possible. This is known as container recycling.

# Recycling Containers

*Container recycling* is what it sounds like: Just reuse the off-screen containers, instead of generating new ones. As of Microsoft .NET Framework, 3.5, WPF does not implement container recycling in any of its panels.

### Expected Performance Benefits

Although the performance benefits will differ substantially (depending on the complexity of the container, the implementation of the recycling algorithm, the number of items that are data-bound inside the container, and so forth), implementing this should show a significant improvement.

A good container-recycling algorithm should be able to remove the cost of creating (30 percent), removing (10 percent), and adding (6 percent) containers to the panel. Removing 46 percent of the bottlenecks will almost double the scrolling speed.

### Suggested Implementation

The simplest way to benefit from container recycling is to write your own **FrameworkElement** that knows how to scroll and display your data items.

The element can be responsible for:

* Holding on to a list of data items (via a dependency property such as **ItemsSource**).
* Instantiating containers, and assigning data items to them.
* Creating its own UI:
  + Creating a panel to hold the containers.
  + Creating a scroll bar, as necessary.
  + Creating exactly as many containers as will fit in the view.

Upon scrolling, the element will basically shift the data items to new containers, as appropriate. This way, the data moves, but the containers do not. This entails:

* Figuring out where each data item is now located in the viewport.
* Updating each container to point to the proper data item.

As before, the "look" of your data can still be defined by **DataTemplate**s. If you use a **ContentPresenter** (or if there is one in the visual tree of your container), it will continue to work.

For an example, see the **ContainerList** class in the sample application.

### Performance Bottlenecks

The performance bottlenecks for our sample application now break down as follows:

|  |  |
| --- | --- |
| **Contribution (percent)** | **Operation** |
| 27 | Arranging (16 percent is **TextBlock.OnRender**) |
| 25 | Data binding (**BindingExpression.Activate**) |
| 20 | Measuring (almost all **TextBlock**) |
| 13 | Rendering |

Table . Performance of recycling containers

The main thing to note here is that the performance is now tightly connected with the specific scenario. Most of the time now is spent laying out and rendering **TextBlock**s. The arranging, measuring, and rendering costs are necessary; this is good, in that it means that we're starting to cut to the bone.

The fact that data binding takes up almost 25 percent of the time is a result of our having six bindings per container. On every scroll, we're invalidating all of those bindings for each container, so that this shouldn't come as much of a surprise. If we still need to make scrolling faster, getting rid of this cost is our next step.

# Faster Bindings

Data binding is immensely useful and, when used properly, fairly fast. Before taking the time to work around using bindings, be sure to profile your application (see the "Resources" section) to ensure that it is indeed the bottleneck.

In this section, we'll assume that, on each scroll, your application is reevaluating a series of bindings as described in the preceding section. There are roughly two ways to get rid of this cost: Don't use data binding at all, or give the data-binding engine a little help.

### Expected Performance Benefits

The expected benefits depend on which implementation you choose. It's possible to remove entirely the cost (25 percent) of data binding by just not using the feature. You can also choose to "help" the data-binding engine, which will remove about half of the data-binding cost.

### Suggested Implementation

**Removing Bindings**

Getting rid of data binding is a sure way to get rid of that 25 percent bottleneck, but it isn't likely worth the cost; if you've chosen to use data binding, it is probably for a reason. You'll lose tons of great features, such as the use of **DataTemplate**s, and you'll have to manage yourself all of the updates to data.

Because this implementation is highly application-dependent and fairly straightforward (although work-intensive), we won't discuss it further here.

**Helping the Data-Binding Engine**

The problem with our sample application is that, each time we scroll, we set the **Content** property on each container in the view to a new data item. This internally causes the **ContentPresenter** to update its **DataContext** property, which means that all of its bindings are now invalid. The data-binding engine here has no choice but to reactivate every **BindingExpression**; that is, it needs to attach it to the common language runtime (CLR) property in question, and transfer the value to the target **DependencyProperty**.

Both of these operations split the time roughly equally. There's nothing that we can do about transferring the value; if it changes, that has to happen. We can, however, avoid attaching the binding to the data object.

The trick here is always to keep the **ContentPresenter** pointed at the same object. To do so:

* Create a proxy object that has the same properties as the data item.
  + It should hold on to an instance of the data item, and forward calls to the data item to its properties.
  + It should implement **INotifyPropertyChanged**.
  + Modify your **DataTemplate**s to reference the proxy object, instead of the data item.
* Have each container point to a proxy object, instead of a data item.
* When scrolling, find each container's proxy object, and change its data item. If the proxy fires its **PropertyChanged** event, the bindings will be updated properly.

The ContainerListFastBinding.cs in the code sample shows how this can be done.

### Performance Bottlenecks

As mentioned earlier, the binding optimization is expected to cut the cost of data binding in half for our target scenario. The numbers now look like the following:

|  |  |
| --- | --- |
| **Contribution (percent)** | **Operation** |
| 34 | Arranging (22 percent is **TextBlock.OnRender**) |
| 26 | Measuring (almost all **TextBlock**) |
| 17 | Rendering |
| 11 | Data binding (**BindingExpression.TransferValue**) |

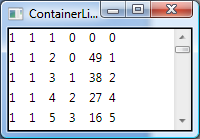
Table . Performance of recycling containers, and optimizing data binding

These are all mandatory operations. Although they look like large bottlenecks, it means that this implementation is about is fast is it will get. The sample application literally only displays **TextBlock**s; it should come as no surprise that almost all of the time is spent preparing and rendering them.

There is also a savings that is not captured in this table. By avoiding changing the **Content** property, we're also avoiding a tree walk on each container to update dependent children (because the property is inheritable). It's a relatively small cost, but it is still there.

# Sample Application

The application consists of a **ListBox** with its **ItemsSource** set to a collection of 2,000 data items. Each data item is a CLR object that has six string properties. To display each item, we used a **DataTemplate** that binds six text boxes to each of the six properties.



**Organization**

The point of the sample application is to show a reference implementation for each performance optimization that was described earlier. It has one class for each section of the document.

|  |  |
| --- | --- |
| **Section** | **Class** |
| Non-Virtualized UI | none |
| UI Virtualization | **ListBox** |
| Recycling Containers | **ContainerList** |
| Faster Bindings | **ContainerListFastBinding** |

Table . Sample application overview

**ContainerList** is a **FrameworkElement** that looks somewhat like **ListBox** and implements a simple container-recycling scheme.

**ContainerListFastBinding** is a subclass of **ContainerList** that implements the data-binding optimizations that are described in the "Faster Bindings" section.

**"Real" Performance Numbers**

**Disclaimer:** The numbers here are illustrative only. The improvements of each method are largely dependent on your data set, the number of items that are in it, and so on.

|  |  |
| --- | --- |
| **Control** | **Time to scroll 150 pages (sec)** |
| **ListBox** | 2.1 |
| **ContainerList** | 1.5 |
| **ContainerListFastBinding** | 1.1 |

Table . Sample application performance

Modifying the sample application to display images instead of text boxes shows a smaller improvement. This is so, because each image is decoded as it is brought into view (a comparatively expensive operation, compared to container instantiation). The sample application has only one image with one binding, so that the data-binding optimization had minimal impact.

|  |  |
| --- | --- |
| **Control** | **Time to scroll 150 pages (sec)** |
| **ListBox** | 5.2 |
| **ContainerList** | 4.6 |

Table . Sample application performance with images

This article focused mostly on describing the bottlenecks in WPF. As Table 6 shows, if your application also does significant work while scrolling, you'll see less of a benefit. Thus, it is important to profile your application, to understand where the most important bottlenecks are.

# Resources

Good background information on this subject can be found on the Web:

* Beatriz Costa's Blog (part three of a three-part blog post, and thorough discussion of **TreeView** performance for large data sets):
  + <http://www.beacosta.com/blog/?p=45>
* "**VirtualizingStackPanel** Class" (overview):
  + <http://msdn2.microsoft.com/en-us/library/system.windows.controls.virtualizingstackpanel.aspx>
* Dan Crevier's Blog (blog posts on implementing a virtualizing tile panel, which could be helpful for readers who hope to apply the concepts in this article to 2-D layouts):
  + <http://blogs.msdn.com/dancre/archive/tags/VirtualizingTilePanel/default.aspx>
* "Analyzing Application Performance" (using the profiling tools that are included in Microsoft Visual Studio 2005 Team System):
  + <http://msdn2.microsoft.com/en-us/library/z9z62c29(VS.80).aspx>
* "Optimizing WPF Application Performance" (general performance tips):
  + <http://msdn2.microsoft.com/en-us/library/aa970683.aspx>

# Acknowledgements

The performance suggestions and sample application in this article are based largely on code by Ben Carter, Atanas Koralski, and Kevin Moore of the WPF team. Jossef Goldberg modified the sample application to display images instead of text.

# About the Author

**Cedric Dussud** is a Software Development Engineer at Microsoft Corporation.