

ASP.NET Performance

Rob Howard

For more information...

- **Rob Howard**
 - Email: rhoward@telligent.com
 - Website: telligent.com
 - Blog: weblogs.asp.net/rhoward
- **Download Presentation**
 - Website: www.rob-howard.net

Agenda

- **Performance Overview**
 - How to think about web performance
- **Measuring Web Performance**
 - How to stress test and report web performance
- **Analyzing ASP.NET Performance**
 - How to understand ASP.NET under load
- **Web performance best practice tips**
 - Recommendations on techniques

Performance Overview

Performance Is A Feature

- Design up front with performance in mind
- Don't "add performance" as a post step!
- Measure & iterate throughout project

Quantifying web performance

- **Client Response Time**
 - Definition: How “fast” does web application appear to remote browser hitting the site
 - Measured via TTFB (time to first byte)
 - Measured via TTLB (time to last byte)
 - Impacts customer satisfaction with app
- **Machine Throughput**
 - Definition: How many client requests can a server handle under load
 - Measured in # of requests/sec
 - Impacts # of servers you need to buy

Measuring Web Performance

Measuring Web Performance

- Only way to measure web server performance is by stress testing the server
 - Automated stress tools only way to measure
 - Hitting refresh in the browser doesn't count...
- Collect performance data on multiple scenarios:
 - Simulate end to end scenario walkthrough of app
 - Measure individual page performance (hotspots)
- Performance metrics to measure:
 - Request/sec using different client loads
 - Identify maximum client load that fits into acceptable TTFB/TTLB response time range

Performance Test Tools

- **Microsoft Web Application Stress Tool**
 - Free 10Mb download for XP, 2000, 2003
 - <http://www.microsoft.com/technet/treeview/default.asp?url=/technet/itsolutions/intranet/downloads/webstres.asp>
- **Microsoft Application Center Test Tool**
 - Ships as part of VS.NET Enterprise
 - Enables richer scripting and reporting

Performance Tool Notes

- Always run stress tools on separate machine from web server
 - Otherwise tool will max out server CPU
 - Use multiple client machines for heavy load
- Configure tests to simulate different client bandwidth levels
 - Specifically measure 56k dialup

Analyzing ASP.NET Performance

Basics of ASP.NET

- **Implemented as an ISAPI Extension**
 - Runs on IIS5 and IIS6
- **Application code runs in worker process**
 - aspnet_wp.exe on IIS5 (Win 2000, Win XP)
 - w3wp.exe on IIS6 (Windows 2003)
- **Incoming Requests are Queued**
 - Executed in-order using a thread-pool
 - Typically 4-6 worker threads/processor
- **CLR manages overall code execution**
 - Memory Management, JIT Compilation

Analyzing ASP.NET PerfMon

- Processor, CPU % Utilization
 - More than 70% = more hardware or performance optimizations
 - Low numbers can be a sign of blocking or lock contention
- ASP.NET Applications, Requests/Sec
 - Dynamic throughput (should be consistent, not jagged)
- ASP.NET, Requests Rejected
 - Indicates server is overwhelmed (should be 0)
 - You need to have multiple stress client threads to hit this
- ASP.NET Application, Errors Total
 - Indicates functional problems (should be 0)
- .NET CLR Exceptions\# Exceptions thrown / sec
 - Exceptions shouldn't be in normal execution (caveat: Response.Redirect)
 - Investigate deeper if count > than 5% of RPS
- ASP.NET App/Worker Process Restarts (IIS5 only)
 - Indicates serious functional problems (should be 0)

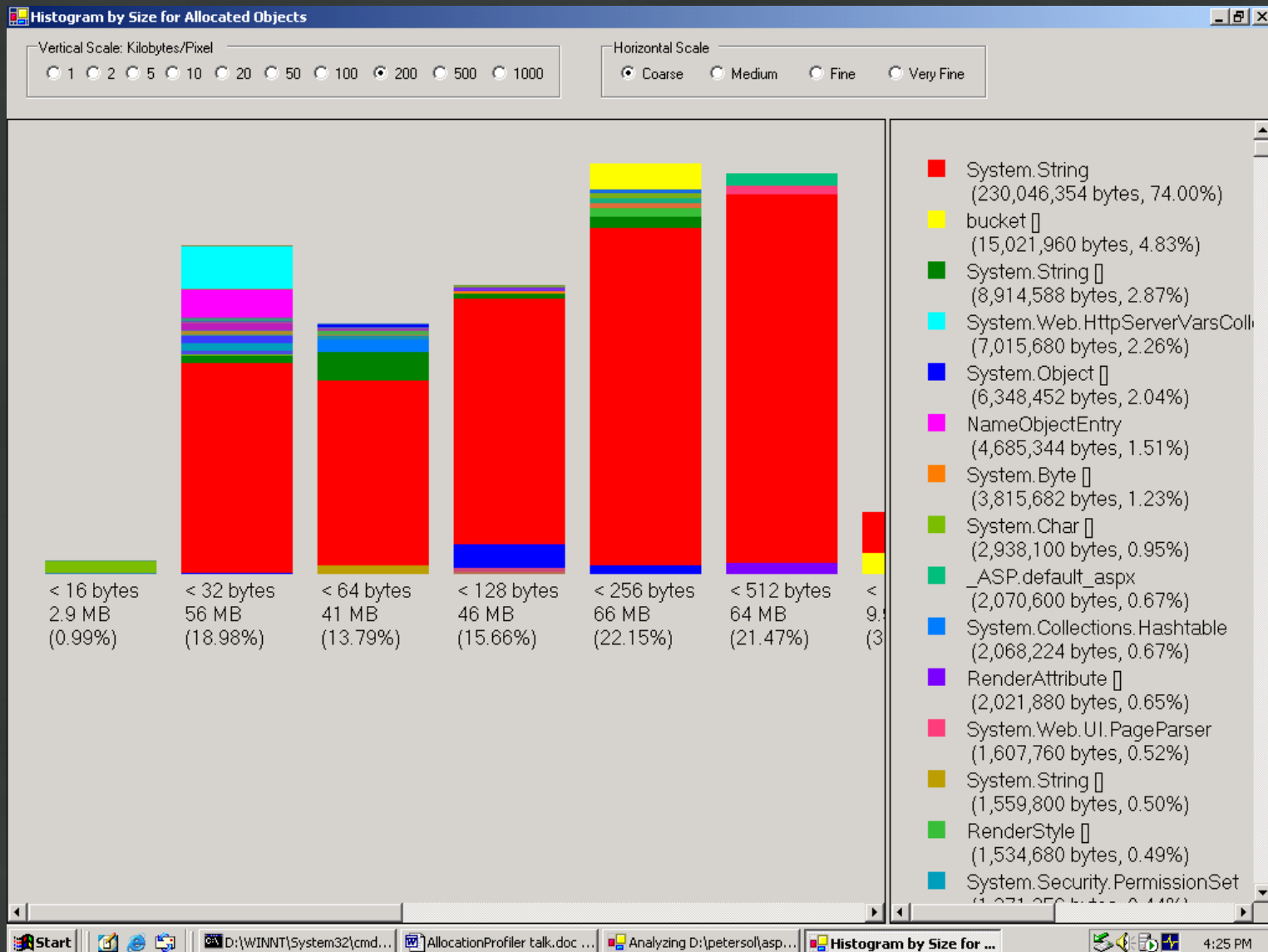
Common ASP.NET Memory Problems

- **Excessive Memory Usage Caused by:**
 - Creating excessive # or large sized cache entries
 - Excessive data in session state or too many sessions
 - Uploading or downloading large files
- **Excessive Memory Allocation Patterns Caused by:**
 - Per-request memory allocation rate
 - This is 99% of the causes in ASP.NET scenarios
 - Some Common Per-Request Examples:
 - Excessive or improper use of strings on a per-request basis
 - Large DataSets or XML documents on per-request basis
 - Returning too much data from a database
 - Too much page ViewState
 - Too many composition-based controls on a page

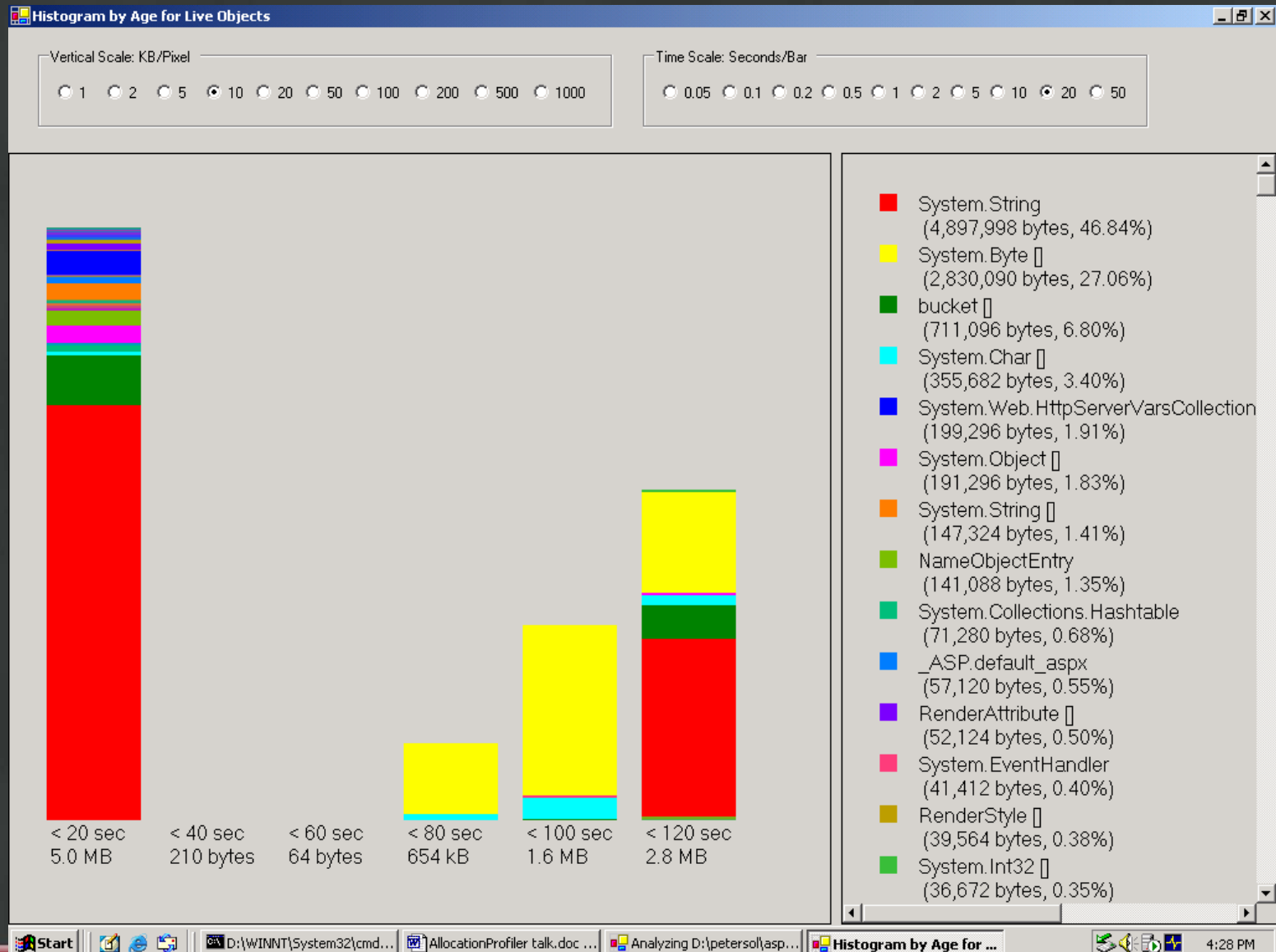
Measuring Memory Allocation

- **Memory allocation can be analyzed with a CLR profiler**
 - Run simulated usage load simultaneously against the server, then figure out what you are allocating
 - Note: server performance will be $\sim 1/10^{\text{th}}$ while under profiler
- **Free Microsoft Memory Profiler Available for Download:**
 - <http://msdn.microsoft.com/library/en-us/dndotnet/html/highperfmanagedapps.asp>
 - Built-in ASP.NET support
 - Full source-code also included

Allocated Size Breakdown



Object Lifetime



Web Performance Best Practice Recommendations

Some Code Best Practices

- **Write clean/organized code**
 - Don't 'hack' solutions (keep code simple)
 - Easier to optimize
 - Easier to maintain
- **Follow good design practices:**
 - Data Access
 - Server Controls
 - Output Caching

Data Recommendations

Connection Pooling

- **ADO.NET has built-in connection pooling**
 - Automatic caching/re-use of connections
 - No need to write any code for this to happen
- **Code Recommendation:**
 - “Open connections in your code late, and then close them early”
 - Don’t hold on to connections for long periods of time – do not try to build your own “smart” connection pool logic
 - Close the connection as soon as you are finished with it (this returns it to the pool)

Watch for Connection Leaks

- **Always explicitly close data connections**
 - Otherwise connection will remain open until the next Garbage Collection
 - Leaking connections slows perf dramatically
- **Specifically watch for leaks during stress:**
 - Monitor user connection count on database
 - Watch the .NET CLR Data Perf Counters
 - Look for steady state behavior (growing = bad)
- **Tip: Database server connections count should roughly correspond to ASP.NET Pipeline Instance Count**

Connection Pooling

- **Optimization Tip:**
 - Different connection strings can generate multiple different connection pools
 - Store single connection string in Web.Config
 - Using ConfigurationSettings.AppSettings to access it programmatically at runtime
 - Watch the “.NET CLR Data” Perf Counters to keep track of the number of connection pools maintained by ADO.NET

DataReaders vs. DataSets

- **DataReader provides forward only data cursor over a query resultset**
 - Lightweight and fast – but connection stays in use until Reader closed or finished
- **DataSet provides disconnected data access collection for data manipulation**
 - Internally uses a DataReader to populate
- **Which is better?**
 - Depends on size of data returned, and confidence that devs will close DataReader

ADO.NET Optimizations

- Only return data you need from the database
 - Memory allocations increase the more you return
- `SqlCommand.ExecuteScalar` method
 - Tuned for scenarios where only a single value is returned for database
- `SqlCommand.ExecuteNonQuery`
 - Tuned for scenarios where resultsets are not returned (except as params)

Use Stored Procedures

- **Recommend SPROCs for data access**
 - Enable easier performance tuning by a DBA
 - Help eliminate database round trips
 - Avoid distributed tx costs by using DB transactions
- **Performance Tip:**
 - Avoid naming sprocs: “sp_”, “fn_”, “sys”
- **Interesting Tip:**
 - Can turn off dynamic SQL support via Enterprise manager to enforce SPROC usage

Watch the Database

- **Carefully monitor your DB during stress**
 - Track CPU performance of DB server
 - Track number of SQL user connections
- **Use SQL Profiler to analyze DB activity**
 - Track queries accessed and execution time
 - Carefully review indexes based on access
- **Recommendation:**
 - Ken England's: "Microsoft SQL Server 2000 Performance Optimization and Tuning Handbook" (Digital Press)

Server Control Performance Recommendations

Server Controls

- Provides a clean programming abstraction
 - Recommended way to build ASP.NET pages
 - Makes profiling your code a lot easier
- Controls do more work than old-style `<%= %>`
 - Should understand and optimize this
- Two areas to review for optimization:
 - ViewState
 - Number of controls generated (especially for lists)

ViewState Management

- **ASP.NET controls can maintain state across round trips**
 - State stored within “viewstate” hidden field
- **Some downsides:**
 - Increases network payload (both on render and postback)
 - Performance overhead to serialize values to/from viewstate
 - Additional Per-Request Memory Allocation
- **Viewstate Flexibility:**
 - Can disable viewstate entirely for a page
 - Can disable viewstate usage on a per control basis
 - Can use `<%@ Page Trace="true" %>` to track usage size
- **Recommendations:**
 - Always disable at page if you are not doing postback on a page
 - Disable on a control if you are always re-generating it on postback

Viewstate Discussion

- When does it make sense to use Viewstate versus recreating them on each request?
 - Depends on your scenario...
- Bandwidth pipe to browser a key consideration
 - Viewstate needs to be small (<1k) for 56k dialup users
 - Viewstate size less important on fast pipe Intranets
- Recommendation:
 - Always be aware and track viewstate sizes, you can then make decisions based on scenario needs
 - Important: You do not need to use viewstate in order to leverage ASP.NET controls

View State Management Tip

- If you want to be more explicit about usage of viewstate, you can configure ASP.NET to turn it off by default

- Machine.config:

```
<configuration>  
  <system.web>  
    <pages enableViewState="false"/>  
  </system.web>  
</configuration>
```

- Pages that need viewstate will then need to manually set it in page directive:

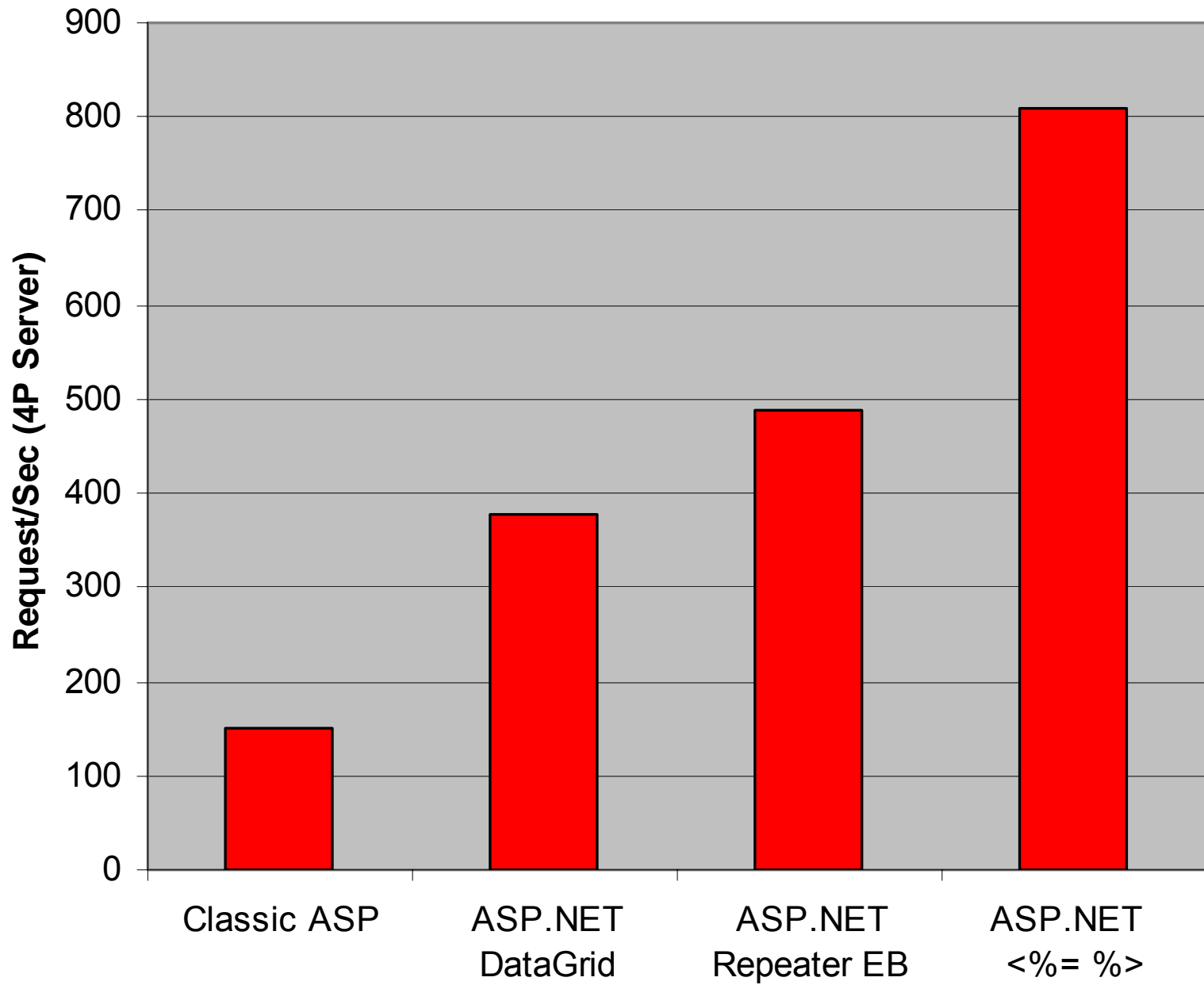
- `<%@ Page EnableViewState="true" %>`

Number of Controls Generated

- There is a fixed cost to each server control used on a page
 - Cost usually negligible on per control basis
 - But memory allocations can still add up...
- Composite controls can sometimes mask the number of controls used though
 - The aggregate cost can sometimes add up
- Eg: `<asp:datagrid>` w/ 10 rows + 7 cols
 - Uses `<asp:table>` internally with 10 table row controls, each with 7 column controls
 - 70 controls now used to display the data
 - 20 more controls if “edit” button displayed

Data Listing Rendering Test

- **Scenario:**
 - 50 Row Query from SQL Northwinds Database
 - Formatted into an html table
- **Five rendering techniques measured:**
 - Classic ASP <% %> w/ ADO
 - ASP.NET <% %> w/ ADO .NET
 - ASP.NET Repeater w/ ADO .NET
 - ASP.NET DataGrid w/ ADO .NET



List Rendering Optimizations

- In majority of scenarios the standard ASP.NET list/repeating controls provide more than enough performance
- To optimize performance, you can also create your own custom server controls
 - Can be as fast as inline `<%= %>` within page
 - Best of both worlds: tweaked performance for your scenario with clean server control programming model

Caching Performance Best Practices

Design For Caching

- Leverage the built-in ASP.NET caching features
 - Output Caching
 - Partial Page Caching
 - Cache API
- Recommendation:
 - Specifically design pages around these features
 - can lead to massive perf wins

Kernel Caching in IIS6

- **ASP.NET leverages kernel cache on IIS6**
 - As fast as static html in these cases
- **Requirements for kernel cache promotion:**
 - HTTP GET Request (no posted pages)
 - No VaryByParam
 - No VaryByHeader
 - No security restrictions to page
- **Note: ASP.NET Request/Cache counters will not update when kernel hit occurs**
 - Monitor “Web Service Cache” counter group

Misc Perf Config Gotchas

- **Make sure debug is turned off**
 - Web.config : `<compilation debug="false"/>`
 - VS.NET leaves in debug by default
- **Make sure tracing is turned off**
 - Web.config and page directive
- **IIS 6.0 Process Model Restarts**
 - Default restart is every 29 hours – turn off
- **IIS 6.0 Maximum Used Memory**
 - Set to 60% of physical memory (keep < 800mb)
 - Important to set if output cache used aggressively

Summary

- Building high performance web apps isn't hard if you design for perf up front
 - Measure and iterate along the way
- Always right clean, maintainable code
 - “Clever” hacks don't make you “smart”
- Follow the recommendations in the slides
 - You'll have clean, fast ASP.NET solutions

Email

rhoward@telligent.com

Resources

<http://www.telligent.com/>

<http://www.asp.net/whidbey>

<http://www.asp.net/forums>

<http://www.aspadvice.com/whidbey>

Blog, etc.

<http://weblogs.asp.net/rhoward>

<http://www.rob-howard.net>

Extra Slides

Understanding Memory Usage

- Memory usage and allocation critical to building high-performance web applications
- CLR has Generational Sweep & Mark Garbage Collector
 - Heap managed via multiple *generations*
 - Generations are *collected* when room is needed for allocation
 - The cost of a collection increases as the generation goes higher
- CLR Generations:
 - Gen 0: New Objects
 - Gen 1: Slightly Long-Lived Objects
 - Gen 2: Long-Lived Objects
- CLR uses a “Server GC” algorithm with ASP.NET
 - Freezes all worker threads when doing a GC
 - Uses high-priority threads to complete GC as fast as possible
 - Optimized for high-server throughput

Optimizing GC Memory Allocation

- **Optimal GC Memory Allocation Pattern:**
 - Some objects allocated in beginning and then survive for the lifetime of the application
 - All other object allocations are short-lived
 - Long-lived objects contain few or no references to short-lived objects
- **Unfriendly GC Memory Allocation Pattern:**
 - Many objects survive long enough to make it into Gen2, but are then disposed
 - Many objects survive long enough to make it into Gen2, and then create and contain references to objects in younger generations during a GC
 - Many short-lived large objects (>80k) are created and then quickly disposed

Measuring Allocation in PerfMon

- .NET CLR % Time in GC
 - Greater than 15% = excessive time spent with memory management
 - Typical cause in ASP.NET: too much memory allocation per request
- .NET CLR Counters/Gen 0 Collections
 - Represents collections of all new objects
- .NET CLR Counters/Gen 1 Collections
 - Should be ~10% of Gen 0 Collections
- .NET CLR Counters/Gen 2 Collections
 - Should be ~10% of Gen 1 Collections

Measuring Memory Usage

- Process(aspnet_wp)\Private Bytes
 - Current size of committed memory owned by worker process
 - Max should be 60% of physical RAM or 800Mb in size (1.8 for /3Gb)
 - Memory leaks identified by prolonged increase in private bytes
- Process (aspnet_wp)\Virtual Bytes
 - Current size of virtual address space of worker process
 - Should be 600Mb less than size of virtual address space (1.4 or 2.4 Gb)
 - Performance degrades as this increases and fragmentation occurs
- Process (aspnet_wp)\# Bytes in all Heaps
 - # of Bytes committed by all managed objects in worker process
 - Value of this will always be less than Process\Private Bytes
 - (Private Bytes – # Bytes in All Heaps) = Unmanaged memory committed
 - First step towards understanding if excessive memory usage is a result of managed code (VB.NET/C#) or unmanaged code (COM/C++ DLLs)

ASP.NET Caching

Output Caching

High Level

Low Level

Partial Page

Web Service

Data Caching

Cache API

Output Caching

- **Caches the static result of an ASP.NET page**
 - Declarative `<%@ OutputCache %>` directive
 - Optional Output Cache APIs can also be called
- **Caching Options:**
 - Duration
 - Time item exists in the cache
 - VaryByParam
 - Varies cache entries by Get/Post params
 - Name param, separate by semi-colons, supports *
 - Location
 - Can save cache on the server, or push down to proxies
 - VaryByHeader
 - Varies cache entries by Http header
 - VaryByCustom
 - Override method within Global.asax to custom vary by whatever you want (you control the cache key)

Cache Performance Counters

- **ASP.NET Application, Output Cache Entries**
 - Total # of items stored in output cache
- **ASP.NET Application, Output Cache Hits**
 - Total # of items served from output cache
- **ASP.NET Application, Output Cache Misses**
 - Total # of cacheable items not served from cache
- **ASP.NET Application, Output Cache Turnover Rate**
 - # of additions and removals to output cache/sec
 - High turnover rate indicates that items are being quickly added and removed – which is expensive memory usage

Partial Page Caching

- Partial Page Caching allows caching page regions using user controls (.ascx)
 - User controls have `<%@ OutputCache %>` directive
- **Additional Features**
 - “VaryByControl” – Varies cached items by controls
 - “VaryByCustom” – Allows user cache key method
 - “Shared” – V1.1 attribute to share output cross pages
- **Recommendations:**
 - Look to use this feature very aggressively
 - If you don't think you can use it, look again because you haven't thought about it hard enough