“High Performance Websites: ADO versus MSXML”

Timothy M. Chester, PhD, MCSD
Senior Systems Analyst & Project Manager
Computing & Information Services, Texas A&M University

Summary: This article is about comparing the ASP/ADO and XML/XSL programming models. The emphasis is not on technique (although sample code is provided) but on performance. This article asks the question, “MSXML is really cool, but how does it perform when compared to the ASP/ADO model I am already familiar with?” Like most web related issues, both methods have tradeoffs. I’ll build two versions of a simple website, one using ASP and ADO to generate the user interface (UI), the other using MSXML. Then I will conduct benchmarks and compare the performance of both models. In some scenarios, ASP/ADO was found to perform better than MSXML. However, in other situations MSXML provided a ten-fold increase in performance.

Tools Required:
Microsoft Visual Studio 6.0, Service Pack 4 (VB, Vbscript)
Microsoft Data Access Components (MDAC) 2.5, Service Pack 1
Microsoft XML Parser (MSXML) 3.0
Microsoft Web Application Stress Tool

Further Reading:
“25+ ASP Tips to Improve Performance and Style”
MSDN Online, April 2000
“Architectural Decisions for Dynamic Web Applications”
MSDN Online, November 2000
“Performance Testing with the Web Application Stress Tool”
MSDN Online, January 22, 2001
“Web Developer: MSXML 3.0 Gains Power”
VBPJ, October, 2000
“Beyond ASP: XML and XSL-based Solutions Simplify your Data Presentation Layer”
MSDN Magazine, November 2000
“Inside MSXML Performance
MSDN Online, February 21, 2000
“Inside MSXML3 Performance”
MSDN Online, March 20, 2000

Other Resources:
Microsoft Developer Network
http://msdn.microsoft.com
http://msdn.microsoft.com/xml/
http://homer.rte.microsoft.com
http://www.microsoft.com/data

Source Code and this white paper
http://enroll-project.tamu.edu/

Skill Level: Intermediate

"High Performance Websites: ADO versus MSXML"

Timothy M. Chester, PhD., MCSD

The Internet has evolved from simple static websites to web-based computing systems that support thousands of users. This evolutionary process experienced tremendous growth with the introduction of Microsoft's Active Server Pages (ASP), an easy-to-use and robust scripting platform. ASP makes it easy to produce dynamic, data driven webpages.

The next big step came with ActiveX Data Objects (ADO) and the Component Object Model (COM). These tools allow developers to access multiple datasources easily and efficiently and best of all, in a way that is easy to maintain. Together, ASP and ADO provide the basic infrastructure for separating data from business and presentation logic, following the now infamous "n-Tier architecture".

With the introduction of XML and XSL, websites are now taking another gigantic leap forward. In this article, I will compare the latest evolutionary leaps with an eye toward website performance by building two versions of a simple website - one using ASP and ADO to generate the user interface (UI) and the other using Microsoft's MSXML parser to transform XML/XSL documents. I will then conduct benchmarks and compare the throughput (transactions per second) of both models. Like most web related issues, both methods have tradeoffs. In some scenarios ASP/ADO performs better than MSXML. In other situations, however, MSXML provides an incredible performance advantage.

n-Tier Architecture: A Quick Review

Client applications typically manage data. To do that effectively, business rules are established that govern the ways information is created, managed, and accessed. The presentation layer allows access to data in conjunction with these rules. When program code is grouped into these three distinct building blocks (data, rules, presentation), it said to follow a logical "n-Tier architecture." There is a simple reason for doing this: it allows the reuse of code objects throughout the software application, thereby reducing the amount of code being written.

One common mistake that is made results from confusing a logical n-Tier architecture with a physical n-Tier architecture. Physical n-Tier implementations usually involve web-based software where most code runs on the server, and little (if any) runs on the client. It is important to understand that one can write a Visual Basic .exe application that separates these layers (thereby taking advantage of code reuse) even if the software runs entirely on the client. However, there is a downside: the software is more difficult to maintain. When changes need to be made, they must be distributed across all clients. Web-based computer systems follow a physical n-Tier implementation. When changes are necessary, they occur on the server as opposed to the client, thus, maintenance costs are reduced.

While ASP and ADO make it very easy to isolate the data layer into a separate COM object, the separation between business and presentation logic is less intuitive. In fact, ASP does not provide an efficient way to force this separation. So most developers usually follow a similar process. A disconnected recordset is created using ADO and then an ASP page loops through this recordset to render some HTML (see code in Figure 1). This can be referred to as the "ASP/ADO model". While simple to create, such pages provide little or no separation between the business and presentation layers.

(Figure 1 Goes Here)

While the ASP/ADO model works, it is not easy to extend to different types of clients, browsers, and platforms (i.e., not extensible).

XML and XSL compensate for this shortcoming. MSXML is a tool for merging XML data with an XSL style sheet. With MSXML, true separation between data, business, and presentation logic is achievable (see code in Figure 2).

(Figure 2 Goes Here)

This programming model results in much cleaner code:

- Create two instances of the MSXML parser.
- Load one with XML data and the other with an XSL stylesheet.
- Transform the first object with the second.
- Write out the results using the response.write statement.

ASP continues to play a vital role under this new model. However, instead of being the primary method for generating UI, ASP becomes the glue that binds data and presentation in a reusable way (i.e. the business layer). This type of coding can be referred to as the "XML/XSL model". The chief advantage of this approach is the ability to easily generate a different user interface (UI) for different devices: web browsers, cell phones, handheld organizers, etc. When data needs to be presented differently, all that has to be produced is a new XSL style sheet. Figure 3 graphically illustrates the differences between the ASP/ADO and XML/XSL models described thus far.

(Figure 3 Goes Here)

Both the ASP/ADO and XML/XSL examples I've provided rely on a separate COM object (written in Visual Basic) to access data. The ASP/ADO data object returns data in a disconnected ADO recordset. The XML/XSL model uses the ADO Stream object to persist data in XML that is then returned as a string. Figures 4 and 5 show sample code using both approaches.

(Figure 4 Goes Here)

(Figure 5 Goes Here)

The example in Figure 4 creates an ADO Recordset object based on a SQL statement that is passed as an input parameter. Once the recordset object is opened it is disconnected from the database connection and then returned. The example in Figure 5 uses the ADO stream object to persist the recordset to XML format. Now, all that remains is to write the stream's contents to the return object. Both these COM functions can be called in your ASP code (See Figures 1 and 2).

There are other possible approaches (see the Further Readings section for more detail). While the ASP/ADO and XML/XSL models could be implemented using pure ASP, this solution will be more difficult to maintain, your options for code reuse will be limited, and the performance of your website reduced.

PERFORMANCE TESTING PRIMER

Performance testing is one of the most crucial aspects of deploying a web based application, but is often overlooked. Performance is typically measured in

throughput, the number of requests that can be processed in one second. There are a couple options for testing a website's performance. One option is to coordinate with hundreds of users who could browse the website at a designated point in time. An easier option is to use a performance-testing tool that simulates this load using a much smaller number of client machines. For these purposes, I am going to use the Microsoft's feature-rich (and free!) Web Application Stress (WAS) tool.

To use WAS, install it on several workstations, with one of these serving as your test controller. The controller's task is to coordinate a performance test with the other clients. After installing the software, create a script that walks through a website, just like an average user. When this script is completed, set the performance test parameters - the stress level, the length of the test, and the types of performance counters that should be tracked. When you begin the test, your WAS controller machine coordinates the test with the other clients. It will then collect the necessary performance data to prepare a detailed report (see "Further Readings" for more information).

Throughput is one of a handful of important indicators of a website's performance. A report generated by WAS will provide the total throughput and also the average Time Till Last Byte (TTLB) for each web page tested. This is the average time required to load a page. When designated, the report will also contain counters from the Windows Performance Monitor. The ASP performance object exposes several important counters.

- ASP requests per second measures throughput, but does not include static content such as images or static HTML pages. It will fluctuate depending on the complexity of the ASP code.

- ASP request wait time is the amount of time (in milliseconds) the most recent request waited in the processor queue prior to being processed.

- ASP request execution time is the number of milliseconds the request actually took to execute.

Together, these last two counters measure the time necessary took for a server to respond to an ASP page request. When subtracted from the TTLB, the remainder is a good estimate of the network time required for the response to travel from the server.

The benchmarks provided in this article are not official. They are designed to illustrate the relative performance of both coding models. The sample code illustrates the most common approach to implementing these technologies. There are other ways. The list of suggested readings cover more advanced coding and server tweaks that can be used to gain additional improvements.

THE SAMPLE WEBSITE

From the earlier discussion it should be obvious that the XML/XSL model provides increased flexibility. It has all the advantages of n-Tier architectures: reusability, maintainability, and extensibility. Now, for the really important questions, "How does it perform? Is it faster than the ASP/ADO model? If it doesn't perform quite as well, what are the tradeoffs?" To answer these questions I have built two simple websites: one uses the ASP/ADO model, the other uses the XML/XSL model. The website is a simple drill-down collection of pages that contain information on courses offered at Texas A&M University (see Figure 6).

(Figure 6 Goes Here)

The first page lists the semesters (Fall, Spring, Summer) for which information is available. A student can click a link and then see a list of departments offering courses for the selected semester. Then, a list of courses offered by a selected department is viewed. From there the student selects a course, and a list of available sections, instructors, times and locations is displayed. The sample code contains all of the ASP, XSL, and VB code. Although the production data resides in a Microsoft SQL Server 2000 server, I have included a Microsoft Access database that contains similar data.

I have installed the two websites on a Windows 2000 Advanced Server (SP1), a Compaq Proliant DL580 with 4 Xeon 800 processors and 4 GB Ram. Everything necessary for the websites, including Microsoft SQL Server 2000, runs directly on this server. The performances tests were conducted using four Gateway workstations running Windows 2000 Professional and the Microsoft Web Application Stress Tool. The WAS tool was used to generate 200 threads of load, which simulates about 2000 active users in this case.

The first results may be surprising. The ASP/ADO model performed better, recording a maximum throughput of 166.99 requests per second, including images. The ASP pages per second counter reported a throughput of 39.40. This compares to 139.84 requests per second and 33.39 ASP pages per second for the XML/XSL model. On average, the ADO/ASP pages required 6.3 seconds to load, compared to 8.1 seconds for the XML/XSL pages.

(Figure 7 Goes Here)

To account for this difference two things are apparent. First, it takes more processor time to transform XML/XSL as opposed to looping through an ADO recordset. Second, the XSL is loaded from disk for each request and this also reduces performance. The other performance statistics also bear this out. The XML/XSL approach consumed 98.61% of available processor time, compared to 90.01% for the ASP/ADO solution. Also, the wait and execution times were also longer for the XML/XSL solution, 4803.41 and 2156.95 versus 4155.48 and 1890.67 milliseconds. Figure 7 summarizes the results of this test.

The ASP/ADO model performs better than the XML/XSL model under this scenario. While the difference is slight, it is real. But this is only once piece of the puzzle. The code used in this test retrieves information from our database each time that a page is requested. If our business rules dictate that our data be fresh for each and every request, the code in Figure 1 and 2 is the best option. But sometimes data doesn't have to be current up to the last second. For example, class schedule course information doesn't change often. So, it is not necessary to retrieve it each and every time from the database. In this situation, MSXML can be used to cache the presentation of data in the ASP Application object. This is something the ASP/ADO model can not easily do, and it can provide a tremendous performance advantage.

CACHING PRESENTATION OF DATA

The great thing about XML, XSL, and HTML is that they are ordinary string objects. The ASP Application object was designed for storing this type of information, thus it can be used to cache XML, XSL, or the HTML presentation of data. This will only work when the data does not have to be current for each page

request. This will also depend on the business rules underlying the website. There is no technical reason why data cannot be cached for days, hours, or even seconds.

Using this method a database and processor intensive website can be transformed into a memory intensive website, provided that the web server has sufficient RAM when doing this. How much? Again, it depends on how much data needs to be cached. The entire sample website used here consists of approximately 15,000 courses and sections. When cached, this requires approximately 120 MB of RAM.

Figure 8 suggests one way an HTML presentation of data can be cached using the ASP Application object

(Figure 8 Goes Here)

In this code, two variables are stored in the application object. The rendered HTML is stored in one variable using the MSXML transform node method. Then, another application variable stores a timestamp that denotes when the HTML was cached. When requested, the ASP code determines whether the HTML exists in the cache and whether it is current. If the answer to either question is "no", the data is retrieved from the database server, transformed using MSXML, and cached in the application object. The current date and time is also cached. When this occurs the page executes more slowly simply since the cache is refreshed. However, subsequent requests will use the cached HTML, thereby executing much faster. In the example code, the HTML presentation is cached every 30 minutes. This could just as easily be 30 hours, 30 minutes, or 30 seconds.

How does this solution perform? "Truly astounding" is perhaps the only way to accurately describe the results. The web server was able to respond to 1398.84 requests per second, including images and static content. This included 332.04 ASP pages per second, roughly ten times the throughput of the ASP/ADO model. On average, the XML/XSL cached pages required less than a second to load (.548 seconds) compared to 6.3 seconds for the ASP/ADO pages. Figure 9 compares these results.

(Figure 9 Goes Here)

SUMMARY

Website performance is not a black and white subject, but is actually very, very, gray. One basic premise is often overlooked: the ways in which a website is coded has as much (or more) to do with performance than the power of the underlying web server. ADO and MSXML are tools that can be used to create high performance websites. MSXML provides increased flexibility to developers, but at a cost. When drawing data directly from a database, MSXML performs slower than ADO. However, MSXML provides an easy way to cache the presentation of data, thereby providing up to a ten fold increase in website performance. This is a viable solution for websites that need to support thousands of concurrent users.

ABOUT THE AUTHOR

Timothy M. Chester is a Senior Systems Analyst and Project Manager for Texas A&M University in College Station, Texas. He specializes in using XML to integrate Microsoft DNA solutions with mainframe systems and other non-traditional data sources. Currently, he is working to web-enable the course registration process for the 44,000 plus students at Texas A&M.

Figures & Code Snippets

Figure 1

```
<% Language=VBScript %>
<!-- ADO/ASP Model Example Code -->

<%
	'////Let's create some objects
	Dim objRecordset, objData
	'///create an adodb recordset object
	set objRecordset = server.createobject("adodb.recordset")
	'///create an instance of my custom data access object
	set objData = server.createobject("mydataobject.selectdata")

	'///go get some data
	strSQL = "Select Title, Year, Term from Courses"
	set objRecordset = objData.SelectSQL(strSQL)

	'///now render the top of my HTML table
%>
	<table>
		<tr>
			<td>Title</td>
			<td>Year</td>
			<td>Term</td>
		</tr>
<%
	'///now loop through my recordset to generate
	'///my table rows

	Do while objRecordset.eof <> True

%>
		<tr>
			<td><%=objRecordset("Title")%></td>
			<td><%=objRecordset("Year")%></td>
			<td><%=objRecordset("term")%></td>
		</tr>
<%
		objRecordset.MoveNext
	Loop

	'///now finish the bottom of my HTML table
%>
	</table>

<%
	'///now destroy my objects
	set objRecordset = Nothing
	set objData = Nothing

%>
```

Figure 2

```
<% Language=VBScript %>
<!-- XML/XSL Model Example Code -->
<%
      '////Let's create some objects
      Dim objData, objXML1, objXML2
       '///create an instance of my custom data access object
      set objData = server.createobject("mydataobject.selectdata")


       '///create two instances of the MSXML DomDocument Object
      set objXML1 = server.createobject("msxml.domdocument")
      set objXML2 = server.createobject("msxml.domdocument")

      '///run my parsers in blocking mode
      ObjXML1.async = FALSE
      ObjXML2.async = FALSE

      '///set msxml1 to my xml data returned from my data object
      '///go get some data, set parser1 to xml data
      strSQL = "Select Title, Year, Term from Courses"
      objXML1.LoadXML(objData.SelectSQL(strSQL))
      objXML2.Load(server.mappath("Courses.xsl"))

      '///now transform the xml and xsl and write it
      '///to the browser
      response.write(objXML1.TransformNode(objXML2))
%>
```

Figure 3

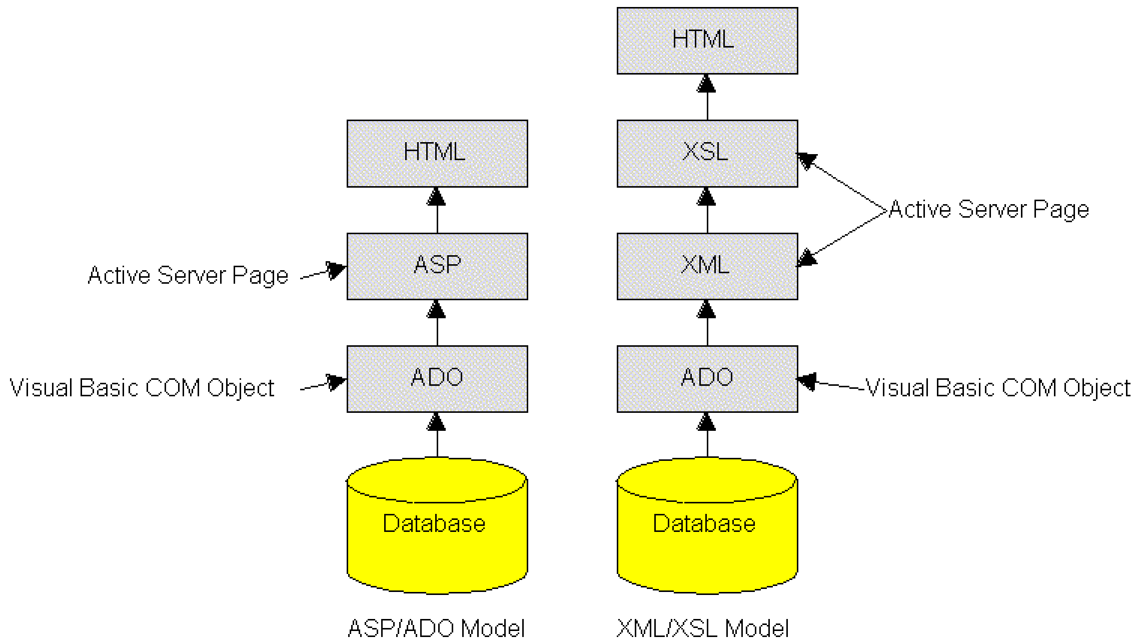


Figure 4

```
Public Function SelectSQLStatement(byval strSQLStatement _
      as String) As adodb.Recordset

      '////public method that executes a sql statement
      '///passed as a parameter and returns the results
      '///as an ado recordset object

      '///create my objects
      dim objConnection as adodb.Connection
      dim objRecordset as adodb.recordset

      '///initialize my objects
      Set objConnection = New adodb.Connection
      Set objRecordset = New adodb.Recordset

      '///use client side cursor
      objConnection.CursorLocation = adUseClient

      '///open connection on connection string
      objConnection.Open "Provider=Microsoft.Jet.OLEDB.4.0;" _
            & "Data Source=" & App.Path & "\courses.mdb"

      '////open my recordset on my sql statement
      objRecordset.Open strSqlStatement, objConnection, _
            adOpenForwardOnly, adLockReadOnly, adCmdText

       '///disconnect my recordset from my connection
      set objRecordset.ActiveConnection = Nothing

      '///set recordset to my return object
      Set SelectSQLStatement = objRecordset

      '///close my database connection
      objConnection.Close

      '///destroy my objects
      Set objConnection = Nothing

End Function
```

Figure 5

```
Public Function SelectSQLStatement(byval strSQLStatement _
      as String) As Variant

      '////public method that executes a sql statement
      '///passed as a parameter and returns the results
      '///as an string

      '///create my objects
      dim objConnection as adodb.Connection
      dim objRecordset as adodb.recordset
      dim objStream as adodb.stream

      '///initialize my objects
      Set objConnection = New adodb.Connection
      Set objRecordset = New adodb.Recordset
      Set objStream = new adodb.stream

      '///use client side cursor
      objConnection.CursorLocation = adUseClient

      '///open connection on connection string
      objConnection.Open "Provider=Microsoft.Jet.OLEDB.4.0;" _
            & "Data Source=" & App.Path & "\courses.mdb"

      '////open my recordset on my sql statement
      objRecordset.Open strSqlStatement, objConnection, _
            adOpenForwardOnly, adLockReadOnly, adCmdText

      '//persist recordset to xml in stream object
      objRecordset.Save objStream, adPersistXML

      '///start stream at first position
      objStream.Position = 0

      '///return xsl as function output
      SelectSQLStatement = objStream.ReadText
      '///set recordset to my return object

      '///close my database objects
      objStream.Close
      objRecordset.Close
      objConnection.Close

      '///destroy my objects
      Set objConnection = Nothing
      Set objRecordset = Nothing
      Set objStream = Nothing

End Function
```

Figure 6

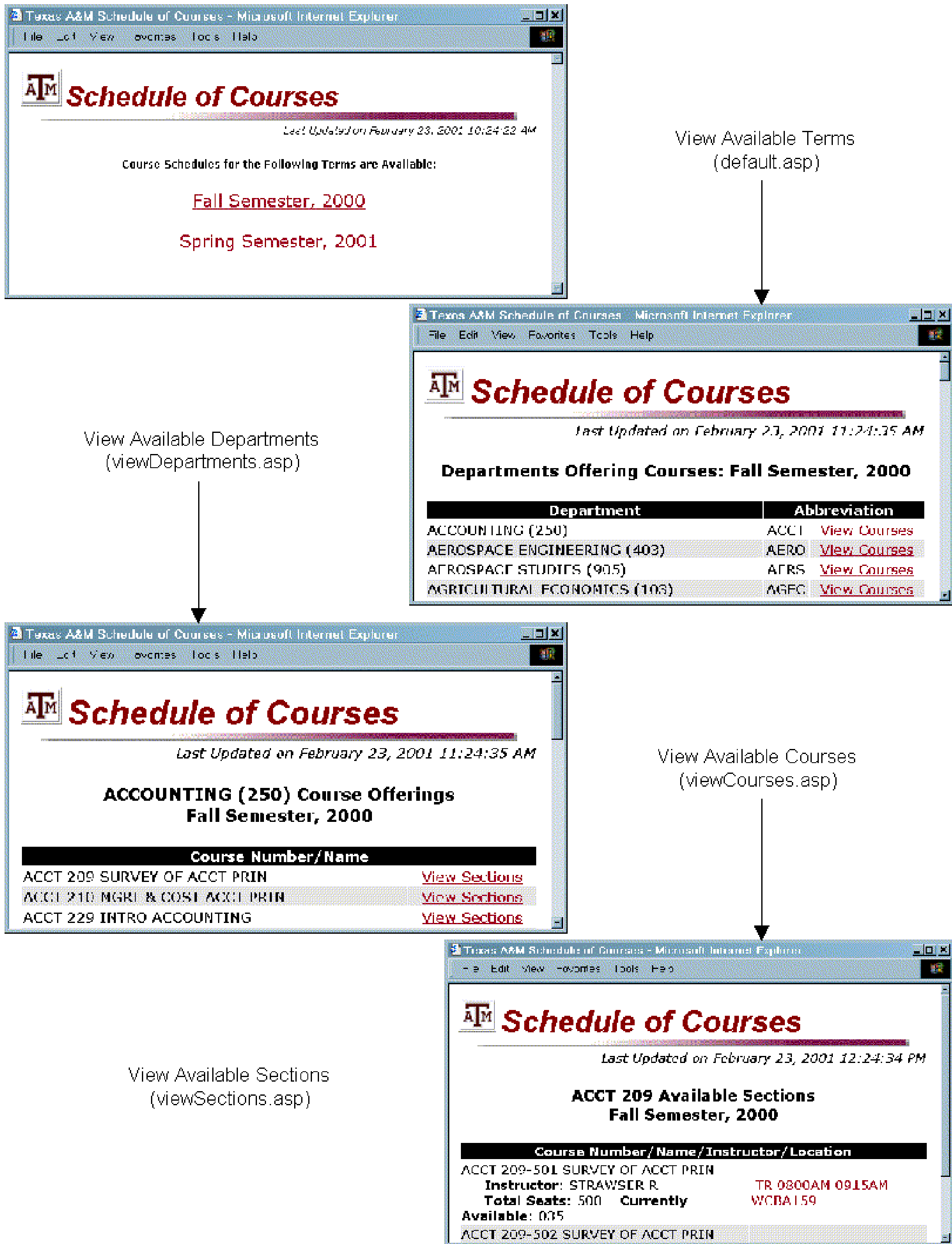
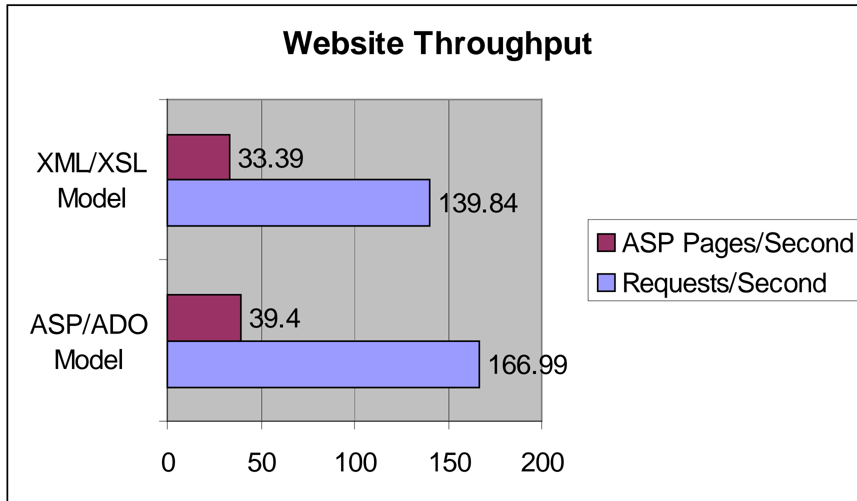


Figure 7



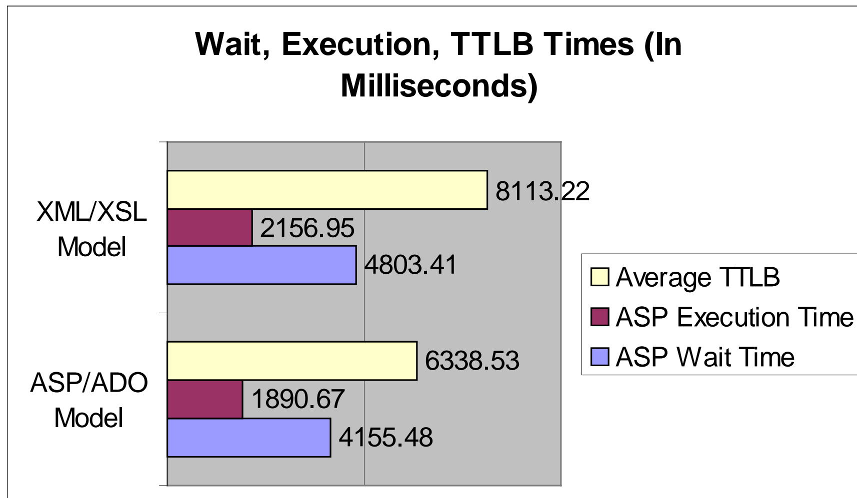


Figure 8

```
<%

'///check to see if I have presentation in my cache
'///and how old it is.  if it doesn't exist, or the data is
'///older than 30 minutes I refresh the application object

If isempty(application("AvailableTerms")) or _
(datediff("n", application("AvailableTermsTimeStamp"), now()) > 30) _
Then

      '///my cache is old or doesn't exist
      '///therefore I just load it

      '////create two msxml parsers as before
      '///create one data object
      set objData = server.createobject("xmlCourses.Select")
      set objXML1 = server.CreateObject("MSXML2.DOMDOCUMENT")
      set objXML2 = server.CreateObject("MSXML2.DOMDOCUMENT")

      '///run my parsers in blocking mode
      ObjXML1.async = FALSE
      ObjXML2.async = FALSE

      '///load xml data from my data object
      objxml1.loadXML(objdata.SelectTermsAvailable())
      '////load xsl from disk
      objxml2.load(server.MapPath("terms.xsl"))

      '//now refresh my application object
      '//and set a timestamp variable so I know how
      '//old my cache is
      Application.Lock
      Application("AvailableTerms") = objxml1.transformNode(objxml2)
      Application("AvailableTermsTimeStamp") = now()
      Application.UnLock


      '///now destroy my objects
      set objXML1 = Nothing
      set objXML2 = Nothing
      set objData = Nothing

End If

      '///now write out my results from the cache
      '///if the cache existed and wasn't old this
      '//would be the only thing to occur on this page
      Response.Write(application("availableterms"))

%>
```

Figure 9