Volume 07 Issue 01 Published, February 19, 2003 ISSN 1535-766X

Intel® Technology Journal

Managed Runtime Technologies

# Developing and Optimizing Web Applications on the ASP.NET Platform

A compiled version of all papers from this issue of the Intel Technology Journal can be found at:
http://developer.intel.com/technology/itj/index.htm

# Developing and Optimizing Web Applications on the ASP.NET Platform

George Vorobiov, Software and Solutions Group, Intel Corporation
Carl Dichter, Software and Solutions Group, Intel Corporation
John Benninghoff, Software and Solutions Group, Intel Corporation
Charlie Hewett, Software and Solutions Group, Intel Corporation

Index words: ASP.NET, CLR, performance, tuning

## ABSTRACT

This paper discusses best practices in developing and tuning the performance on Intel architecture of Web applications based on the ASP.NET* platform. We provide an overview of the ASP.NET platform and discuss a number of optimizations that can be applied to this class of applications based on our findings from developing and characterizing an e-commerce workload. We also want to disseminate our knowledge to the industry on optimal software configuration and the use of the rich feature set provided by ASP.NET and Common Language Runtime (CLR).

Our major emphasis has been to characterize and improve the performance of these applications. To do this effectively the design of the application and configuration of the infrastructure to host this software application are discussed. The description of the application shows the components participating in request processing and response generation. We also present the analysis of performance problems and tradeoffs facing ASP.NET developers. Finally, we discuss the evolution of the workload to include different distributed computing scenarios using emerging technologies such as Web Services and .NET Remoting*.

* Other brands and names are the property of their respective owners.

## INTRODUCTION

ASP.NET* is a new Microsoft Web development platform, which is built on top of the Common Language Runtime (CLR) platform, and it inherits many familiar features of the ASP platform. Despite the similarities in the naming and APIs exposed, these two implementations do not have much in common in terms of their underlying technology and performance characteristics. While the ASP platform used an interpreted scripting language and was limited in the way it can interact with other components of the operating system (OS), the new ASP.NET platform enables the use of a variety of different programming languages, all of which are compiled into Intermediate Language (IL) and all of which can take advantage of all the features provided by CLR and Windows .NET.

Some of the benefits of the new programming model are given below.

- There is a true separation of the presentation logic code and HTML scripting. Visual Studio .NET* facilitates this process by automatically creating a code-behind class in a language of the developer's choice that can process events and dynamically modify the page presentation, based on the current application state.
- Session state management now supports both efficient in-process session state handling and a more scalable Microsoft SQL server-based solution that allows sharing of the session state by a large number of servers in a server farm scale-out scenario.

* Other brands and names are the property of their respective owners.

- There is support for modular page design, using extensible server-side controls, that enables component-based programming models to easily share functionality between multiple ASP.NET pages.
- New and improved ADO .NET* data access APIs provide both fast forward-only data retrieval methods and a more advanced DataSet approach for creating an offline view of the data that can be accessed and modified independently and synchronized with the database when needed.
- There are more opportunities for efficient caching using both object cache and output caching features.

CLR is a new managed runtime platform by Microsoft that is designed to increase programmer productivity by providing a rich set of features such as automatic garbage collection, built in support for multiple remote procedure call (RPC) mechanisms, full compliance with multiple XML standards, and a state-of-the-art object-oriented programming framework. It also provides easier deployment and administration support with code verification and type safety checking, global assembly cache (GAC) for shared libraries, and code versioning support to eliminate the infamous "DLL Hell"[1] problem.

ASP.NET applications also take advantage of Microsoft's new Internet Information Server (IIS) processing model that was introduced in version 6. IIS streamlines the Web request execution by using a new 2-tier kernel mode listener/worker process model instead of the former 3-tier model.

To characterize and optimize the new programming platform, we developed an e-commerce workload modelled after a small Web-based bookstore. This workload is designed to exercise most of the major features of ASP.NET and CLR. In this paper we discuss the findings from our work on optimizing the performance of this application using the standard Intel methodology described in the optimization section.

We hope that the information in this paper will be useful to developers who want to understand system and software design issues, and their respective impact on performance. It should also be of interest to those with a current implementation in ASP or who want to weigh the cost and benefits of moving forward with the latest software technology from Microsoft.

[1] DLL Hell is often used to refer to the problem with deploying different versions of the same library that leads to incorrect execution of the programs that depend on it.

# WORKLOAD DESCRIPTION

In order to successfully apply optimization techniques to a given application it is necessary to understand the design and performance characteristics of the system. This section concentrates on describing the architecture and configuration of the system as well as the hardware profile.

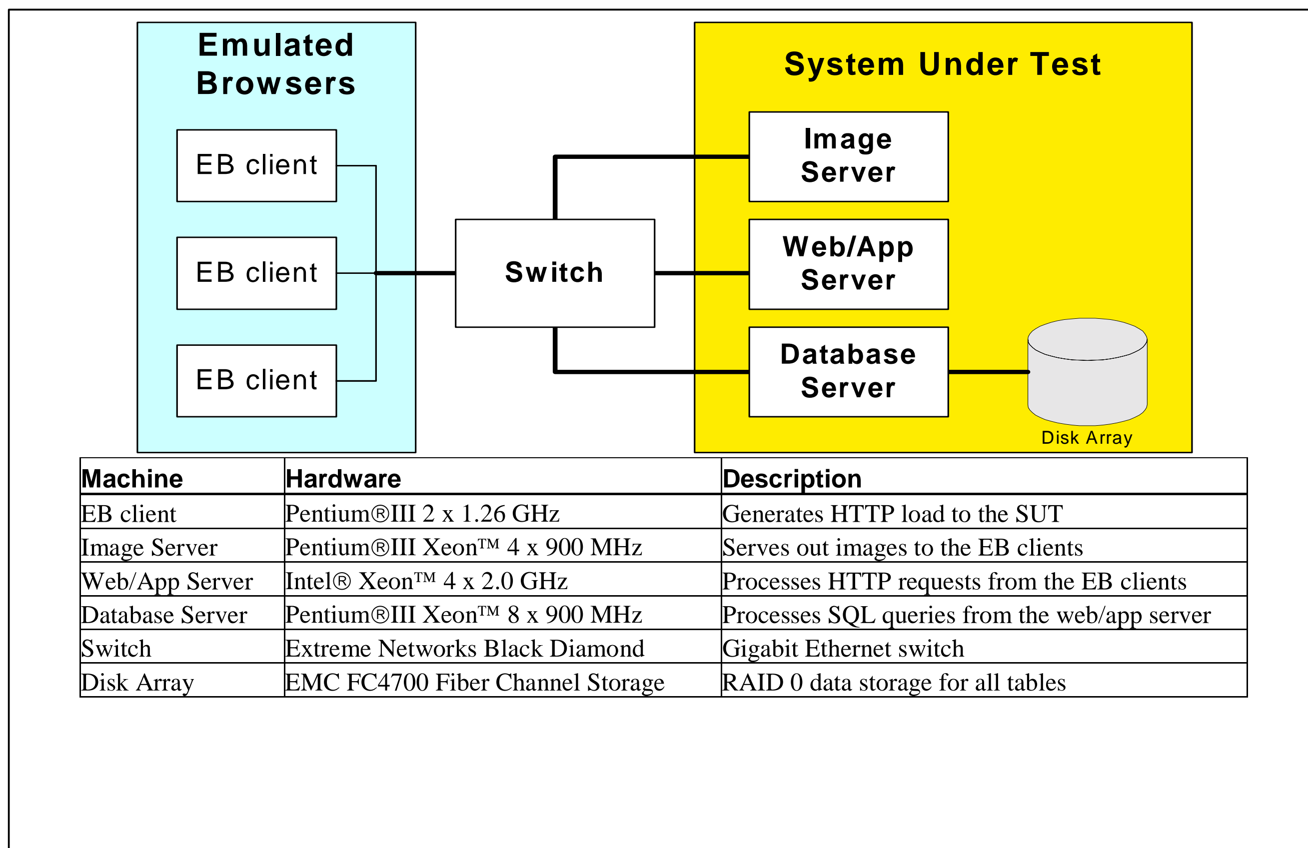
## Hardware Description

Our e-commerce workload can be functionally divided into two distinct parts: emulated browsers, or EBs, and the system under test, or SUT. Every EB emulates a number of distinct users, each generating their own unique HTTP traffic to the SUT. The SUT is the collection of servers that makes up the e-commerce solution that accepts these requests. Figure 1 depicts the hardware layout.

Major components of the workload required to provide the full implementation of the application and assist in load generation include the following:

- Web/Application Server. This machine is running IIS with the actual ASP.NET application containing presentation, business logic, and data access components.
- Database Server. This contains an application database and a set of stored procedures to provide optimized data access support.
- Image Server. This Web server, running IIS, handles all the HTTP image requests from the clients.
- Emulated Browsers. These run our custom load generator tool to provide sufficient load on the system for workload characterization and performance problem identification purposes.

A typical Web request from an EB client is passed through the switch and accepted by the Web/app server, which parses the request and generates queries to the Database (DB) server if necessary. After receiving the response from the DB server, the Web/app server generates its response to the EB client. The EB client will then parse the response page and retrieve any images from the image server. Based on the response from the Web/app server and a few other conditions, the EB will randomly generate its next Web request, and the pattern repeats itself until the end of a run.




| Machine | Hardware | Description |
|---|---|---|
| EB client | Pentium®III 2 x 1.26 GHz | Generates HTTP load to the SUT |
| Image Server | Pentium®III Xeon™ 4 x 900 MHz | Serves out images to the EB clients |
| Web/App Server | Intel® Xeon™ 4 x 2.0 GHz | Processes HTTP requests from the EB clients |
| Database Server | Pentium®III Xeon™ 8 x 900 MHz | Processes SQL queries from the web/app server |
| Switch | Extreme Networks Black Diamond | Gigabit Ethernet switch |
| Disk Array | EMC FC4700 Fiber Channel Storage | RAID 0 data storage for all tables |

**Figure 1: Hardware deployment diagram and descriptions for the e-commerce test suite**

## System Software Description

The workload runs as a Web application in the instance of the CLR platform (version 4322) hosted by the IIS worker process (W3WP.EXE). All machines in the SUT are running on Windows* Enterprise Server* version 3663. As seen in Figure 2 below, incoming request processing starts at the kernel mode listener (HTTP.SYS) and is then passed to the IIS worker process running our application. There can be more than one worker process if multiple Web applications are running concurrently. CLR is loaded inside the worker process, using the ISAPI extension mechanism common to all Web application types under the IIS.

The ASP.NET application is controlled by the web.config configuration file located in the application directory. This file specifies multiple aspects of the application runtime behavior, such as the type of session state handling to use (in-process is the default, but an SQL Server instance or a custom implementation can also be specified), or the custom error processing page, which provides a unified way of handling uncaught exceptions by displaying a more user-friendly message instead of a standard stack trace. You can also specify a different authentication mode, such as the use of the Windows authentication mechanism or the Microsoft Passport* service.

Since our workload is a managed application running on top of CLR, it is also controlled by the CLR machine.config file located in the config subdirectory of the framework installation. This XML file contains multiple settings for ASP.NET, such as the number of the worker threads available to process application requests and the size of the ASP.NET request queue. Optimal values for these settings are discussed later.

IIS settings are common to all types of Web applications and can be applied using the Web Administration Service, as shown in Figure 2. You can configure multiple application pools and assign different applications to a given pool. Application pool settings provide a set of configurable parameters to improve ASP.NET application robustness and performance.

* Other brands and names are the property of their respective owners.

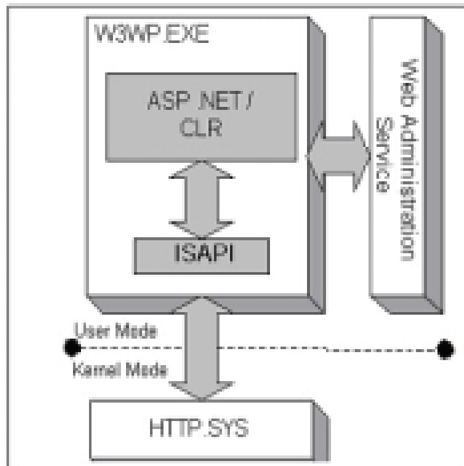



**Figure 2: System software configuration**

## Application Description

Our workload is designed to provide a representative yet simple Web application, exercising most of the major features of ASP.NET and CLR, such as garbage collection (GC), ADO .NET data-access framework, server-side controls, and XML support. It is modeled after a small bookstore Website that provides catalog browsing, shopping-cart maintenance, ordering, and support for administrative functions. The workload logical architecture provides true separation of Web application layers by defining clear interfaces between these layers and providing implementations adhering to the defined interfaces. These logical application layers are the Presentation, Business Logic, and Data Access layer, respectively.

The design of the workload is based on the patterns and practices for developing distributed applications for .NET [5]. The main reasons to use such a multi-tiered logical structure were to make sure that the workload is representative of this class of applications and to facilitate future design changes to utilize distributed Web Services and Remoting components.

The Presentation layer of the workload consists of a set of ASP.NET Web pages (ASPXs) and custom server-side controls (ASCXs) with their code-behind classes. The code-behind classes dynamically modify the resulting document by requesting up-to-date information from the Business Logic layer and manipulating the properties of the controls on the page.

The Business Logic layer contains a set of stateless services, implementing the business interfaces that provide current application data, based on a set of business rules. This layer uses the Data Access layer to retrieve and update the data, based on parameters received from the Presentation layer.

The Data Access layer handles all data retrieval and update requirements defined by the application. It uses the ADO .NET framework to access an SQL Server database, using the .NET SQL Server driver. This layer can use either dynamically built SQL commands or a set of stored procedures residing in the database.

To better understand the system behavior, we present the sequence diagram of request processing execution in ASP.NET as shown in Figure 3. Inside the CLR, the request is handled by an instance of the HttpApplication object representing a single pipeline of execution. It maps the incoming request URL to a specific ASPX page, creates an instance of the code-behind class for this page, and executes any event handlers defined for the page or for the user-defined server-side controls defined on the page.

In most cases the application logic is executed by the Load event script. It starts by validating the input parameters and initializing data structures. Then the script requests an instance of the business logic class, specific to the request type. The application is structured so that an instance of a different stateless service class can be loaded at startup, depending on whether remote method invocation via Remoting or Web Services is used, or on whether the service runs locally. We discuss the latter in this paper, but part of the future direction of our workload is to run business and data access logic on a separate application server.

Once the request is passed to the service object, some additional business rules are applied to ensure the validity of the request, and an appropriate data access provider class is invoked. Our workload is configured to allow for different types of data access and retrieval, the two main alternatives being the use of SQL Server stored procedures and dynamically generated SQL statements. (Find out more about the comparison of these two alternatives in the analysis section.) The default for the workload is to use stored procedures for all data access. In this case, an object of the class implementing the stored procedure-based data access is instantiated dynamically at application startup.

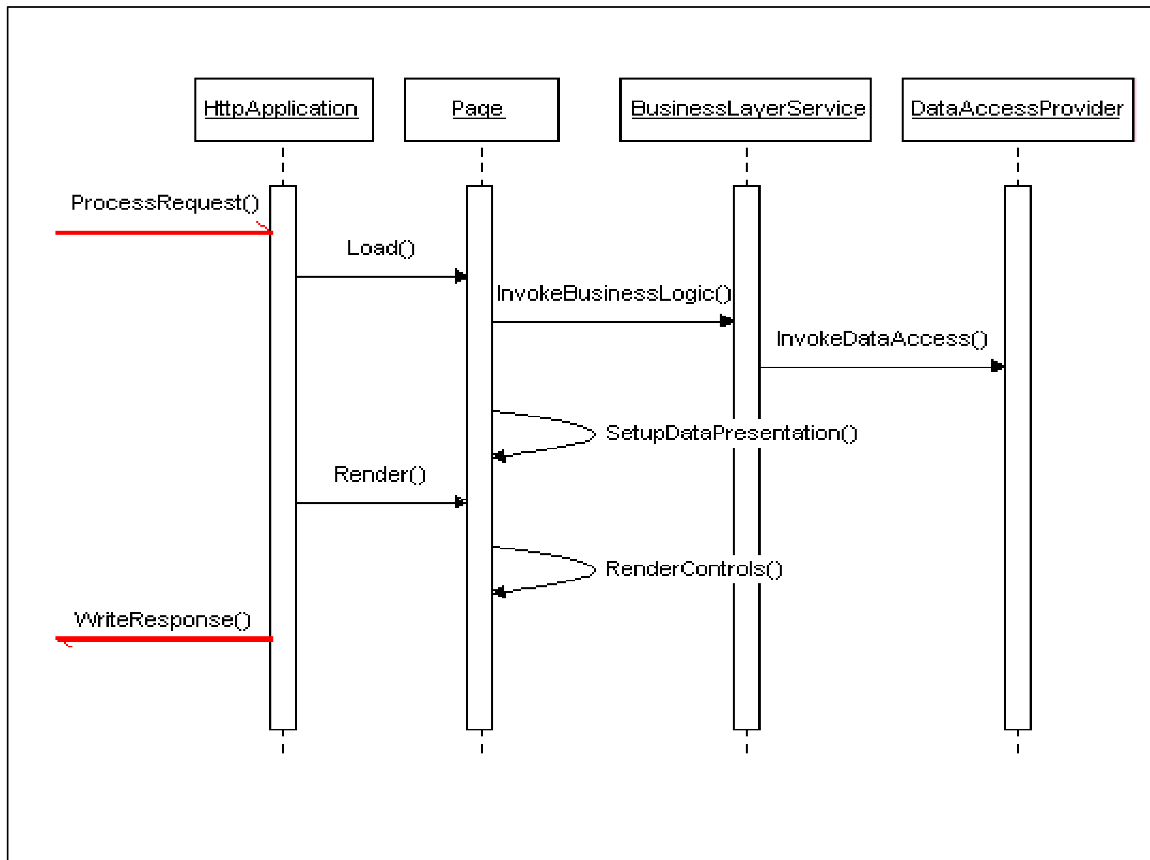



**Figure 3: Workload sequence diagram**

Once the stored procedure implementation is invoked, it constructs a new SQL command to call a stored procedure on the database server by using the .NET driver for the Microsoft SQL Server. The .NET driver has a built-in capability to provide connection pooling, so after the interaction with the database is completed, the current data connection utilized by the current request is returned to the pool of available connections. This helps eliminate the overhead of connection creation on every request, while freeing the developer from manually implementing connection pooling.

After the stored procedure results are received, the control passes back to the Load event script. The rest of the logic in it manipulates the controls on the page to show the results returned from the database. Once the script execution for the page is completed, the Load scripts for all custom controls on the page are invoked if necessary to execute control-level logic.

Finally, the engine proceeds with rendering the page, which involves recursively calling the *Render* method for every server-side control on the page and writing the result to the Response object output stream.

Note that if the output caching is defined for any of the controls on the page, or the page itself and the control or page output is currently in the ASP.NET output cache, then none of the event scripts will be executed for it, and the previously saved (cached) output from the page or control will be written to the output stream. Output caching is discussed in detail later in this paper.

## Performance Data

Figure 4 shows the throughput scaling for one, two, and four Intel® Xeon™ 2.0 GHz processor configurations on the application server with Hyper-Threading enabled for our workload. As you can see, it scales fairly well with a two-processor scaling of 1.87, and a four-processor scaling of 3.27. A number of system- and application-level tunings were applied to enable good CPU utilization, which is critical to achieve such scaling. These optimizations are discussed in the following sections.

---

Intel® Xeon™ is a trademark of Intel Corporation or its subsidiaries in the United States and other countries.

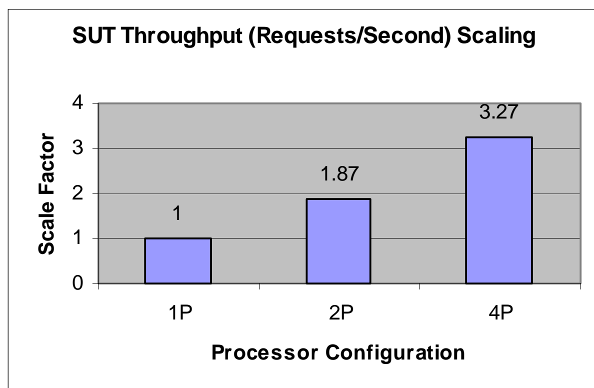



**Figure 4. Throughput scaling diagram**

# OPTIMIZATION

The goal of applying optimizations is typically to achieve maximum throughput and/or to lower response time. Optimizing a complex system consisting of a number of hardware and software components can be difficult, but by applying a systematic approach, optimizations can be discovered and applied effectively.

The optimization process begins by collecting data on the system and analyzing the data. A performance issue is identified, and alternatives are proposed and explored. Next, a solution is applied to the system, and data is collected again to analyze the performance difference. The change is then accepted or rejected and the performance analysis cycle begins again.

In addition to this iterative approach, a top-down methodology is used. There are three levels of optimization: system, application, and micro-architectural. The top-down methodology means that tuning begins with higher impact changes such as I/O and database configuration at the system level. Only after the bottlenecks have been removed from the higher level of optimization, can the next level of optimization take place. This means that system-level optimizations have to come before application-level optimizations, which have to come before micro-architecture-level optimizations.

Furthermore, after applying a change at a lower level, you must go back again to the top level and begin the optimization again. For instance, an application-level change could help performance, but also expose a system-level bottleneck that then needs to be alleviated. Generally, when optimizing a complex system, a majority of the time is often spent identifying system-level issues.

Each different class of tuning requires a different set of tools and performance tests. For instance, system-monitoring tools such as Microsoft's Perfmon* are excellent at finding system-level issues, but will not find micro-architectural-level issues. The Intel® VTune™ Performance Analyzer is an excellent performance analysis tool, providing Call Graph, Counter Monitor, and other valuable data. It is also flexible enough to be applied in system-, application-, or micro-architecture-level tuning, but it may not provide the necessary specific data. For instance, if you know you have a network issue, Microsoft's Netmon* tool can be used specifically for that purpose.

In tuning an application, you will need to apply a systematic, top-down approach, and use the correct tool to analyze the performance of your system. The following sections will explain system- and application-level optimization. Although it is possible to achieve performance benefits with micro-architectural tuning, they are typically smaller and not within the scope of this article.

A great deal of this paper is dedicated to caching. This is because it is the single biggest source code or configuration file optimization that you can perform in most Web applications.

## System-Level Optimizations

Our methodology is to look at the system-level issues first, making sure that the performance bottleneck is not in the physical capacity of I/O devices, disks, and network, etc., but in the application- and system-level code. Once the system-level optimizations are applied, we can proceed with our top-down methodology, analyzing performance and applying changes to the application, configuration, and tunable parameters.

To apply system-level tuning we utilized Perfmon*, Microsoft's performance-monitoring tool. When using Perfmon, a number of counters are selected to monitor such metrics as Processor, Network, and Disk utilization, as well as more application-specific counters such as "ASP.NET\Requests Queued" and ".NET CLR Memory\Number of Bytes in all Heaps." Perfmon, along with some knowledge about the physical limits of the components you are studying, can help diagnose many system-level issues.

---

* Other brands and names are the property of their respective owners.

Intel® VTune™ is a trademark of Intel Corporation or its subsidiaries in the United States and other countries.

For our workload, we found network bottlenecks in both our application and image servers. By looking at the Perfmon logs, we were able to ascertain that we were packet limited on the application server, due to the large number of requests entering from the emulated browsers (EBs), and that we were throughput limited on the image server, due to the large size of some of the images being accessed. In each case, the problem prevented us from saturating our application server processors. Installing Gigabit (1000Mbps) Ethernet cards in both the application and image servers alleviated the network bottlenecks.

Database optimizations were also necessary, including the addition of indices and tuning of specific database parameters. In the early stages of our workload development, we found that the disk I/O subsystem on the database server was overutilized. Moving the database and log files to physically separate disks helped, but did not completely alleviate the problem. In order to reduce latency and alleviate the disk bottleneck, it was finally necessary to move the database and log files each to their own RAID 0 disk array.

It is obvious that without applying system-level optimizations such as those described above, application-level tuning will not provide any benefit. For example, if the processor is busy only 10% of the time, making the code twice as efficient would only result in a modest performance improvement. However, if the processor is busy 100% of the time, code efficiency improvements result in significant performance improvement. The goal of system-level tuning is to ensure that the bottleneck lies in code that the developer controls, such that application-level optimizations can then be applied.

## Application-Level Optimizations

After system-level bottlenecks have been alleviated, developers can further increase performance through changes in the application, including caching, data access optimizations, and application tunable parameters. The following is a discussion of some caching and tuning options provided by ASP.NET and ADO.NET.

In our efforts to optimize the workload, we concentrated on achieving better use of the platform features and not improving the efficiency of our application code, since our application code only consumes less than 1% of the CPU, as measured by the Intel VTune Performance Analyzer. The main reason for this is the fact that the platform automates most of the common programming tasks that used to be handled by the application code, such as memory management or session state handling.

**Caching**

Popular Web sites receive millions of hits per day. Normally, that means many millions of disk accesses hit their databases, large amounts of processing are used to serve their application, and possibly other systems are used for Web services or other remote objects.

What would happen if every request for a Web page could be met by sending the image of a page already created? In this ideal situation, the Web server would only have to serve up existing pages from memory, along with statically included information such as pictures. This integrated cache is part of the power of Microsoft's .NET. It has the ability to cache entire Web pages, fragments of a Web page, or any object.

Overall, there are two types of caching: output caching and data caching. Output caching can be either caching of entire Web pages or fragments of Web pages. This caching in .NET is unlike caching of static Web pages because it works with dynamic content (pages built with ASP.NET). Each time a page (or a fragment of a page) is requested with the same dependencies as another recent request, the information is delivered from cache. (Dependencies and expiration periods are more fully described later.)

Data caching, which is also called Object Caching, is the ability to cache the output of any method. Typically it is used for methods with a high-performance cost, such as methods that access a database. Until .NET, many serious applications would build their own data cache, but few did it well.

If properly used, .NET's caching options can provide huge performance improvements. Proper use of the cache results in less object creations, less processing, and most important of all, lower dependency on slow system resources (such as disk I/O) or external facilities such as databases, remote objects, or Web services [1].

We now provide a detailed analysis of how different types of caching can be applied to a Web application to improve its performance.

***How to Use .NET's Object Cache***

Using the object cache is easy; using it properly is difficult.

We cover the basic syntax for using the object cache. More detailed information about how to use the cache can be obtained from the Microsoft Developer Network (MSDN) and *GotDotNet* (www.gotdotnet.com) Web sites.

The object cache can be used wherever a method returns an object. There are also several forms for inserting data into the cache, including versions that allow you to set

expiration times for the item. In our example, the cache has a simple key as the parameter to the *fetchMyData* method.

```
DataView Source;
// See if the object is in cache
Source = (DataView)Cache["MyDataKey"];
// If object isn't in cache, get
if(Source == null )
{
        Source = fetchMyData("MyDataKey");
        // Put data in cache
        Cache["MyDataKey"] = Source;
}
// use the data
MyDataGrid.DataSource = Source;
MyDataGrid.DataBind();
```

The cache key must contain all the dependencies that would affect the output of the method to be cached. The key is always derived from some or all of the parameters to the method being called; however, it may also use other variables that can affect the result. For example, fetchMyData might contain internal state information – such as my user information – such that it would return a different value for another user. In this case, it is prudent to include the user ID as part of the key to the cache.

In addition to simple variables, dependencies can also include files, directories, or the keys for other objects in the cache.

Incorrectly identifying the dependencies for caching can result in incorrect program output or low cache hit rates. Determining these dependencies requires a programmer's skill, i.e., don't expect to see tools to automate the process anytime soon.

***Proper Use of the Object Cache***

Using the cache properly to optimize performance is not a simple matter. There are two performance problems that are prevalent in many .NET applications:

- Under-caching. To under-cache is to fail to take advantage of caching where it would be a benefit.
- Over-caching: Using the cache where you shouldn't is known as over-caching.

Under-caching is caused by the fact that unlike a processor cache, .NET caching is not something that happens automatically. If you haven't implemented caching, you are missing out on performance.

The second problem, over-caching, is equally serious. On applications that abuse the object cache, the object cache itself becomes a major bottleneck that limits the performance of the application.

Part of the problem with over-caching is that putting something in cache might imply "kicking-out" something else that is needed. With .NET, a bigger part of the problem is that the cache object is a synchronized (thread safe) collection; therefore, it can become a major source of contention when multiple threads are trying to access it at the same time.

Proper use of the .NET cache includes putting all things in cache if they will be used again, and only if they will be used again.

***Finding Under-Caching and Over-Caching***

The best way to find these problems would be to have a tool do all the work; unfortunately, that is not possible today. These tools are probably coming in the near future, but they will require additional instrumentation in the .NET framework.

In the meantime, here are some strategies to consider and some tips on how each might be applied in various situations.

- Start from the ground up: implement caching only where appropriate.
- Observe what is removed from cache.
- Manually instrument every use of cache to determine hit/miss rates.

What follows is a description of each approach and an analysis of each from the perspective of accuracy, ease-of-use, and ease-of-implementation.

***Start From the Ground Up***

In this approach, start without any caching, and then use strategies to implement caching only where it is strongly indicated.

This approach is applicable if you have not implemented any caching or when you have caching that may be improperly implemented. It is easier to find under-caching than it is to find over-caching.

The implementation effort is very high with this approach, especially for programs that already have caching. First, you must remove (comment out) all caching, then you must use a methodology to decide where to cache, and finally you must re-implement caching where appropriate.

One way to find out where to cache is to start with the objects that have the highest creation rates, implement caching there, and observe the hit rate.

When applying this approach, keep the following in mind:

- String and other objects will show up as high usage in many places. You must determine whether the same data are ever likely to be returned, or time will be wasted implementing caching where the hit rate will be low.
- Incorrectly implementing caching can result in incorrect program output, or low cache hit rates. You must correctly identify dependencies to implement proper caching; unfortunately, determining dependencies is not a trivial task. Dependencies may be more than just parameters to the method; they may include files or time information.

Ideally, you should only cache where it will yield the most benefits. In reality, your results will depend a lot on the skill and effort of the implementer. Intel® Solution Services has helped many customers determine appropriate caching schemes for their applications, often improving application performance in the process.

***Observe What is Removed From Cache***

If the code already makes extensive use of caching, use this technique to determine if you are caching when you shouldn't or have implemented caching incorrectly.

For this technique, you can use the *CacheItemRemovedCallBack* to tell when an item is removed and to track the type of objects being evicted. The most evicted objects might represent objects that were placed into the cache but never used because unused items in cache will get evicted due to the Least Recently Used algorithm the cache employs.

There are three problems with this technique. One is the complexity of implementation. You'll have to implement cache (with the *CacheItemRemovedCallback*) in all the places where it is likely to be helpful. You will also need to implement the actual method that will be called by the callback mechanism. You will then have to figure out where in the program these callbacks are happening. Unless you use a different callback, you will only have the name and value of the key to guide you.

Another problem is there is no way of knowing how long these items were used in cache before they were evicted. They might have been worth caching. They might have been used for a while then evicted when they expired or a dependency changed.

® Intel is a registered trademark of Intel Corporation or its subsidiaries in the United States and other countries.

***Manually Instrument Use of Cache***

In this approach, you modify the application's source code to instrument every use of cache to determine hit/miss rates, hit/miss counts, and the ratio for that use. For example:

```
DataView Source;
Source = (DataView)
Cache["MyDataKey"]; // Cache lookup.
if(Source != null )
   // Add instrumentation
   // to count cache hits
            XXXXXXXXX();
else {
            // Add instrumentation to
   // count cache misses
            YYYYYYYY();
 // Get the data
            Source = fetchMyData("MyDataKey");
// Put the data in cache
            Cache["MyDataKey"] = Source;
}
// use the data
```

This is very accurate: it simulates processor cache counters to show where cache is effective and where it is not. Unfortunately, extensive code changes are required to implement this instrumentation, and they can be more complicated in certain code situations.

First, caching has to be implemented on every likely candidate. Implementing caching is difficult because although it is only a couple lines of code, you must correctly identify dependencies.

Second, since C# does not provide macros like *__FILE__* and *__LINE__* or the ability to build macros which would combine these built-in counters with instrumentation, implementing the cache manually is labor intensive.

***Using ASP.NET Output Caching***

Output caching represents another way of reusing the previously generated data instead of performing the processing-intensive operation again.

Output caching is used when the whole page, or part of it, can remain static for some period of time. Obviously, the biggest benefits can be reaped if the whole page is stored in cache for future use, but this may not always be the case. In this case, the page can be restructured to make the cacheable part of it a separate user-defined server-side control. Then caching can be applied to the output from this control. The easiest way to enable output caching for both user control and the whole page is to put the OutputCache directive at the top of the ASPX page containing its definition:

```
<%@ OutputCache Duration="#ofseconds"
     VaryByParam="parametername" %>
```

This approach is the best for the pages and controls that can always be cached for a specified period of time. You can also use the VayByParam attribute to specify caching by parameter, which caches the output for every unique value of the input parameter. If the decision of whether to cache the output depends on more complex criteria, the framework provides a programmatic API to dynamically enable the output caching:

```
Response.Cache.SetCacheability(HttpCacheability.Public);
Response.Cache.SetExpires(DateTime.Now.AddSeconds(30));
```

Based on the output from our workload, we experienced a 23% performance degradation in the case where output caching is disabled. Based on our experience, significant performance gains can be attained if you examine your application to see which output pages do not need to always show the current snapshot of your data, so that output caching can be applied.

Better yet, the new IIS 6 architecture allows for an output of dynamic ASP.NET pages to go into kernel mode cache. This increases the benefit from caching, since the processing of requests does not need to transition from kernel to user mode, a transition that involves a context switch and is, therefore, much more expensive.

Use of output caching can also help in scale-out scenarios, where all the cache requests can be serviced by a separate system running the Microsoft ISA Server*.

***How to Choose Which Type of Caching to Use***

When choosing a type of caching you need to examine your application and determine whether the data you can cache are used by a single page or by a number of pages. If only one page (or control) is affected, it is likely that output caching will work better. The reason for this is that in the case of output caching, you eliminate both data retrieval and presentation logic, so the amount of work to satisfy the request is minimized. In the case of object caching, only the data retrieval part is eliminated, but the application still needs to execute the programming logic that produces the resulting HTML document that is based on the data retrieved. One potential problem with the output caching API, as compared to object caching, is that it currently lacks the ability to dynamically invalidate cached pages that are based on criteria other than expiration time.

The object cache approach works better when multiple pages are using the data object cached, or the source of the data is a file on the application server.

**Session State Usage Tuning**

Session state is a service provided to your Web application by default; the assumption is that all pages in your application require session state data to process incoming requests. If any of your pages do not rely on session state, you can get a limited performance benefit by disabling it for a given page.

To disable session state for a page, set the EnableSessionState attribute in the @Page directive to false as follows:

```
<%@ Page EnableSessionState="false" %>
```

**Server-Side Control Guidelines**

The general advice is not to use any server-side controls unless you have specific reasons to do so. Some of the reasons to use server-side controls over plain HTML might be to programmatically modify/access control properties, or to implement the same functionality on multiple pages, or use fragment caching on them.

If you don't have these reasons to use server-side controls, use plain HTML tags (located under the HTML tab in the WebForm Designer Toolbox) as a better performing alternative, since HTML tags do not require additional objects to be instantiated and manipulated during page processing on the Web server.

## Software Configuration Issues

There are a number of ASP.NET settings that reside in the machine.config file provided with the .NET framework that can be tweaked to improve system performance.

The following is an example of some of the important components of the machine.config file: httpRuntime, sessionState, and processModel.

```
<httpRuntime
        executionTimeout=""
        minFreeThreads=""
        minLocalRequestFreeThreads=""
        appRequestQueueLimit= ""
    />

<sessionState
        stateNetworkTimeout=""
        timeout=""/ >

<processModel
        requestLimit=""
        requestQueueLimit=""
```

* Other brands and names are the property of their respective owners.

```
    memoryLimit=""
    maxWorkerThreads=""
    maxIoThreads=""
/>
```

The two parameters that will likely need to be tuned on an ASP.NET server are the request queue limit and the maximum number of worker threads.

**Request Queue Limit**

When system resources such as the CPU, or available worker threads, become saturated, ASP.NET will direct the request to the ASP.NET request queue. The requests in the queue are then serviced in the order in which they arrived (FIFO) as resources become available. The request queue is very effective in handling heavy or non-steady-state loads, but a limit must be placed on this queue, because throughput and response times will degrade as the queue grows larger.

The request queue limit can be found at \System\Web\httpRuntime\@appRequestQueueLimit in the machine.config file. This parameter is the maximum number of requests that will be queued before 503 "Server too busy" HTML errors are returned. For debugging, performance tuning, or any situation where error pages should not be returned, assigning a high value to the appRequestQueueLimit will prevent the HTTP 503 errors. In our configuration, we are testing performance and cannot tolerate errors being returned, so we set the queue size to be large enough such that requests will never be rejected. However, in a production environment, where the server can become overloaded with requests, a balance must be made between performance and not rejecting requests.

**Maximum Thread Count**

Many Web pages rely on an external resource such as a database or Web service, and waiting on these resources will often halt the processing of that request. If more CPU bandwidth is available, it is advantageous to process another request while the other is halted. Thus it is important to have the maximum thread count set high enough such that the CPU is saturated fully. However, having an excessively high thread count can cause too many requests to be processed concurrently, which can degrade performance.

The maximum worker thread limit can be found at \System\Web\processModel\@maxWorkerThreads in the machine.config file. This parameter is the maximum number of worker threads that will process requests. It is beneficial to run with as few threads as possible. However, without sufficient worker threads, the CPU will not be fully utilized, so the optimum value must be experimentally determined.

Unfortunately, ASP.NET does not provide a performance counter to measure the number of worker threads in use. The only way to estimate this number is to use the "ASP.NET Applications\Pipeline instance count" counter. Pipeline in this context is an instance of your HttpApplication-derived class named Global, which resides in the Global.asax.cs file in your Web application directory. This class defines the logic of HTTP request handling inside ASP.NET, common to all types of request processing, whether HTML (ASPX pages) or XML based (Web Services). At any given moment, a pipeline instance can process only one request, so this counter indicates how many requests can be processed concurrently, which has some correlation to the number of worker threads.

One example of potentially needing a higher thread count is when the ASP.NET server has long-standing requests out to a database (or other external resource). In the early stages of our workload development, the database implementation was slow and inefficient and response times were poor. Because of this, more threads were needed on the ASP.NET server so that it could continue processing requests while waiting for responses from the database.

However, in our current, more optimized workload, after varying max threads with 1, 5, 10 and 15, the results revealed that 10 threads (per processor) provided the maximum throughput for this application. This setting was found to be quite different if the SQL server did not use stored procedures. It is therefore likely that you will need to run a series of experiments to determine the optimal configuration. This procedure may need to be repeated if the design of your application changes.

## ADO .NET Related Optimizations

ADO .NET provides a number of alternatives for accessing and modifying the data residing in the SQL Server database. The DataSet API provides a unified way to manipulate the data in the offline mode and synchronize it with its source. The DataSet API supports different types of data sources: an SQL Server database, an XML file, or some custom data provider for a proprietary format.

A different set of APIs supports fast data retrieval of ResultSet objects and execution of either dynamically generated SQL stat elements or stored procedures. The main difference between the last two methods is the fact that the SQL Server needs to parse and recompile the SQL statement submitted in the case of dynamic SQL statements, whereas stored procedure execution uses a precompiled version of the SQL statement, so these two steps are no longer required. We have also discovered that the previous assumption about the high overhead of

executing the stored procedure compared to a dynamic SQL statement is not valid in the case of the Microsoft SQL Server. This is because in the case of dynamic SQL statements, the .NET client implicitly invokes a stored procedure called sp_executesql.

To show the benefits of the stored procedure approach over dynamic SQL generation, we ran a series of tests on the current workload implementation. In the run where we replaced the stored procedure implementation with the dynamically generated SQL alternative, the throughput of the system as a whole dropped by over 18%. Moreover, in order to compensate for the worse response times of the database (DB) server, we needed to increase the number of ASP.NET worker threads to 20 from the original setting of 10.

# WORKLOAD EVOLUTION

The current version of the workload is geared towards improving the performance of ASP.NET* Web applications by exercising most of the common features of the new engine. However, the current computing trends point toward more scalable scenarios, where the Web server integrates the data provided by the remote application services. The .NET platform provides a number of choices for implementing distributed computing.

## Distributed Computing Scenarios

As mentioned earlier, our workload has a built-in flexibility that allows it to dynamically load different implementations of stateless business service instances. While the current version of the workload is configured to create instances locally to study the behavior of a single Web server, we currently have two more versions of the business service classes: one implemented as a Web Service and another one implemented as a Remoting server application. In the case of the latter, only remote object proxies are instantiated locally, not the actual implementations itself. The only thing that is required to enable this change is to change the application configuration file to specify a different class to be loaded. This process is totally transparent to the presentation layer: no change to the source code is required.

### Web Services

Web Services are the new emerging standard in the distributed computing arena. Their most appealing feature is that they utilize existing standards, such as HTTP and XML, which are the most widely deployed and used. The downside of this protocol is the relatively high overhead that is associated with it: it is based on XML and rides on top of HTTP. However, Web Services are clearly the best choice for heterogeneous computing environments where ease of integration and support are the key factors.

.NET provides the highest level of support for the Web Services standard, so the integration of components running on remote machines is almost transparent.

### .NET Remoting API

Remoting API is an alternative to Web Services, which provides an extensible framework that allows you to easily configure communication channels by using different protocols and encoding standards. This set of protocols allows for binary encoding and use of the TCP protocol instead of HTTP.

Our preliminary results indicate that this option provides significant performance benefits over Web Services, and that it is also better integrated with the .NET framework. The downside, however, is that it can only be used when both client and server applications are CLR-based, which means that it is not the right solution if cross-platform compatibility is required.

# CONCLUSION

ASP.NET* is a great new development platform for Web applications. It provides a rich feature set and automates a number of common tasks. However, because it hides the complexity of handling Web requests from the developer, it is easy to incorrectly estimate the performance impact of different implementation alternatives and system-level configuration options.

The object and output cache are powerful capabilities provided by .NET and using them properly is a key to high performance and scalable applications. As the object cache capability is fairly new, tools are not yet available to automate the process, but a skilled programmer can use these techniques and get great results. Output caching brings substantial performance benefits while requiring minimal coding efforts.

By applying systematic methodology and appropriate tools it is possible to identify and alleviate the performance bottlenecks in the system. Also, it is extremely important to tune the application software stack to achieve optimal performance. This is a complicated task that requires some experimentation in

* Other brands and names are the property of their respective owners.

* Other brands and names are the property of their respective owners.

order to find optimal values for your specific application. When the proper tuning has been applied, ASP.NET applications can scale well on Intel-based servers.

Web Services and .NET Remoting technologies enable the next generation of Web applications that consist of a number of distributed services, seamlessly integrated, using these protocols. .NET Remoting offers higher performance than Web Services, but it is not platform independent; it requires that both the client and server applications are CLR-based.

Ongoing research occurs at Intel to ensure that the current and emerging technologies such as ASP.NET, Web Services, and .NET Remoting perform well on Intel Architecture. We hope that this article has provided valuable information to assist in developing and optimizing ASP.NET applications on Intel-based servers.

# ACKNOWLEDGMENTS

Sam Warner provided a lot of advice and hands-on experience with tuning the HTTP and Microsoft's Internet Information Server (IIS) layers, and he also helped with identifying performance issues.

Paul Delvecchio was a great source of information on hardware setup and configuration as well as system-level tuning.

# REFERENCES

[1] F. Yeon, "ASP.NET Performance Tips and Best Practices," http://gotdotnet.com/team/asp/ASP.NET%20Performance%20Tips%20and%20Tricks.aspx, October 22, 2001.

[2] J. Richter, "Applied Microsoft .NET Framework Programming," Microsoft Press, January 2002, ISBN: 0735614229.

[3] Microsoft ASP.NET, http://asp.net/

[4] Microsoft Patterns and Practices, http://msdn.microsoft.com/practices/

[5] "Application Architecture for .NET: Designing Applications and Services," http://msdn.microsoft.com/library/?url=/library/en-us/dnbda/html/distapp.asp, December 2002.

[6] "Developing High-Performance ASP.NET Applications," http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpcondevelopinghigh-performanceaspnetapplications.asp, December 2002.

# AUTHORS' BIOGRAPHIES

**George Vorobiov** is a senior software engineer in the Software and Solutions Group in Bellevue, Washington. He is working on performance optimization of managed runtime technologies in the Web server application space. George holds an M.S. degree in Computer Systems from Kursk State Technical University, Russia; His e-mail is George.Voroibov@intel.com

**Carl Dichter** has been a systems and software engineer for over twenty years and has been at Intel since 1995. He has developed methodologies for optimizing server applications, especially managed code (C# and other .NET languages, as well as Java) and currently works with the processor architects to make sure our processors run today's applications best. Carl has filed 11 patents and written over 60 articles and one book (on software engineering and related subjects). His e-mail is cdichter@yahoo.com

**John Benninghoff** is a lead software engineer working on performance analysis of enterprise workloads on the .NET CLR Framework and ASP.NET. He has been at Intel for three years and has worked in the software industry for 20 years. Prior to Intel he worked at a major network software company doing performance analysis on Web and LDAP servers for their Internet portal, serving millions of registered users. Prior to that he worked for a major computer system vendor on graphics and network software. His e-mail is john.benninghoff@intel.com

**Charlie Hewett** is a software engineer in the Software and Solutions Group in Bellevue, Washington. He works on performance analysis and optimization of managed runtime technologies such as Microsoft's Common Language Runtime (CLR) and .NET Framework for Intel Architecture. Charlie holds a B.S. degree in Electrical Engineering from the University of Washington. His e-mail is charlie.j.hewett@intel.com

Copyright © Intel Corporation 2003. This publication was downloaded from http://developer.intel.com/.

Legal notices at http://www.intel.com/sites/corporate/tradmarx.htm.

For further information visit:
developer.intel.com/technology/itj/index.htm

Copyright © 2002, Intel Corporation. All rights reserved.
Intel is a trademark or registered trademark of Intel Corporation or its subsidiaries in the United States and other countries.
For a complete listing of trademark information vistit: www.intel.com/sites/corporate/tradmarx.htm