64-Bit Performance Benchmark: .NET 2.0 and IBM WebSphere 6.0

CachePerf: Examining the Impact of 64-bit Extended Memory Addressability on .NET and WebSphere Middle Tier Web Applications

November 2005

The information contained in this document represents the current view of Microsoft Corporation on the issues discussed as of the date of publication. Because Microsoft must respond to changing market conditions, it should not be interpreted to be a commitment on the part of Microsoft, and Microsoft cannot guarantee the accuracy of any information presented after the date of publication.

This White Paper is for informational purposes only. MICROSOFT MAKES NO WARRANTIES, EXPRESS OR IMPLIED, AS TO THE INFORMATION IN THIS DOCUMENT.

Complying with all applicable copyright laws is the responsibility of the user. Without limiting the rights under copyright, no part of this document may be reproduced, stored in or introduced into a retrieval system, or transmitted in any form or by any means (electronic, mechanical, photocopying, recording, or otherwise), or for any purpose, without the express written permission of Microsoft Corporation.

Microsoft may have patents, patent applications, trademarks, copyrights, or other intellectual property rights covering subject matter in this document. Except as expressly provided in any written license agreement from Microsoft, the furnishing of this document does not give you any license to these patents, trademarks, copyrights, or other intellectual property.

© 2005 Microsoft Corporation. All rights reserved.

Microsoft, the .NET logo, Visual Studio, Win32, Windows, and Windows Server 2003 are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries.

The names of actual companies and products mentioned herein may be the trademarks of their respective owners.

Microsoft Corporation • One Microsoft Way • Redmond, WA 98052-6399 • USA

Introduction 2
Full Disclosure Notice 2
The Limits of 32-bit Memory Addressability 2
32-bit Platforms 3
Per-Process Memory Limitations on 32-bit Platforms 3
64-Bit Platforms 4
CachePerf: The Benchmark Application Scenario 4
The Database Server Hardware 5
The Application Server Hardware 5
The Application Server Operating Systems Tested 5
The Application Server Software Tested 6
The Benchmark Database 6
The Benchmark Application 7
A Note on Object Caching vs. Servlet Caching In WebSphere 6 9
The Java Code 9
The .NET Code 12
Configuring the Amount of RAM available for Java and .NET Cache Entries 14
Configuring the IBM WebSphere Heap Size 15
Configuring the .NET Cache Size 15
Summary of CLR and JVM Memory Settings 16
IBM WebSphere 6.0.2.3 16
Microsoft .NET 2.0 18
The Test Methodology 18
The IBM WebSphere Cache Offload-to-Disk Feature 19
The Results in Detail 20
Comparisons 20
Breakdown by Platform 22
Tabular Results 25
Discussion of Results 25
Need to Cache More Data? 27
IBM WebSphere 6 Offload-To-Disk 27
Benchmark Results 29
32-bit .NET (no offload-to-disk) 29
32-bit IBM WebSphere 6 with DiskOffLoad Disabled 29
32-bit IBM WebSphere 6 with DiskOffLoad Enabled 30
Tabular Results 31
Discussion 31
Conclusion 32
Appendix 1: The Benchmark Source Code 33
The Java Code 33
The .NET Code 43
Appendix 3: The Tuning Parameters for the Benchmark 50
WebSphere 6 Tuning 50
.NET 2.0 Tuning 51
Mercury Settings 51

Introduction

With the arrival of 64-bit computing for mainstream developers, many developers and IT professionals are asking an important question: in what scenarios will 64-bit computing make a performance difference for line-of-business applications on the middle tier, and how big a difference can it make? This paper examines one core scenario where 64-bit platforms can have a tremendous impact on the performance of a wide variety of middle-tier applications. This scenario is the use of the extended memory addressability of 64-bit platforms vs. their 32-bit counterparts to dramatically expand the benefits of object caching on the middle tier. We analyzed this scenario using a Web-based benchmark application we created called CachePerf. CachePerf measures the performance of an application server as it receives incoming Web requests and services these using an object-level, server-based cache.

In addition, the CachePerf results presented in this paper directly compares the performance of .NET 2.0 and IBM WebSphere 6.0.2.3 on both 32-bit and 64-bit platforms. Specifically, benchmark results are presented for 32-bit and 64-bit .NET 2.0 on Windows Server 2003 and 32-bit and 64-bit IBM WebSphere Application Server 6.0 on both Windows Server 2003 Enterprise Edition and RedHat Linux Enterprise Server 4. All tests were conducted on AMD-Opteron hardware.

Full Disclosure Notice

The complete source code, all test scripts and all testing methodology for this benchmark are available online. Any reader may download and view the actual code for all implementations tested, and may further perform the benchmark for themselves to verify the results. The benchmark kit can be downloaded from http://msdn.microsoft.com/vstudio/java/compare/.

The Limits of 32-bit Memory Addressability

Before delving into the benchmark application scenario, it's useful to first discuss the technical difference between 32-bit and 64-bit platforms when it comes to addressing memory. While 64-bit platforms such as Windows Server 2003 can have a potential performance impact in other areas such as math-intensive applications with heavy use of floating point operations, by far the biggest benefit of the 64-bit platform is its ability to break through the limited memory addressability of 32-bit platforms, and allow applications to utilize much more memory. This can be important for memory-intensive desktop platforms, but can be even more important for the performance of backend servers that potentially serve thousands of concurrent users and manipulate large amounts of data and/or potentially cacheable files on disk. This includes database applications (such as Microsoft SQL Server), mail servers such as Microsoft Exchange, terminal-server based Windows applications, and with .NET 2.0 64-bit, middle tier .NET applications that can cache or otherwise need to manipulate large amounts of data in-memory on the middle tier. These benefits also extend to 64-bit Java applications that manipulate large amounts of potentially cacheable data on the middle tier.

32-bit Platforms

32-bit computing platforms such as Intel x86-based computers treat physical memory addresses as 32-bit integers. Hence, a byte-addressable 32-bit computer can address 2^{32} = 4,294,967,296 bytes of memory, or 4 gigabytes of RAM directly. Of course it's possible to put more than 4 gigabytes of RAM into an x86 computer and have it recognized by the operating system, since an operating system can virtualize memory addresses and do some fancy tricks to map memory addresses beyond the 4 gigabytes threshold back into 32-bit physical addresses. For example, while Windows Server 2003 Standard Edition can only see 4 gigabytes of RAM , Windows Server 2003 Enterprise Edition, is able to recognize up to 32 gigabytes of RAM, and Windows Server 2003 Data Center Edition can recognize up to 64 gigabytes of physical RAM . But on 32-bit platforms, that memory is made available to individual applications in much smaller parcels.

Per-Process Memory Limitations on 32-bit Platforms

The problem is, even with a 32-bit OS such as Windows Server 2003 Enterprise Edition running on a computer with 32 gigabytes of RAM, there is still a hard limit on what an individual process or application can use. On Windows, this limit is roughly 2 gigabytes of RAM per running process, no matter how much physical RAM is installed.[1] Hence, on a 32-bit operating system, any single .NET CLR process can only take advantage of 2 gigabytes of RAM inclusive of the CLR itself, the application logic, and application data. Similarly, 32-bit Java JVMs are limited by this same barrier---a 32-bit Java JVM can only address roughly 2 gigabytes of heap space, so the heap size for Java-based application servers such as IBM WebSphere 6.0 is limited to no more than ~2 gigabytes on all 32-bit platforms it runs on (including Windows and Linux).

For many middle-tier applications, 2 gigabytes is enough memory. However, applications that are designed to optimize performance for very high throughput under highly concurrent load – thousands of users and thousands of transactions per second – often use an aggressive caching strategy. The application retains the results of expensive computations, and returns those results to subsequent requestors. In this kind of design, the 2 gigabyte memory limit can be reached quickly. If you want to cache more than 2 gigabytes worth of data in cacheable record sets or other types of objects in the 32-bit .NET CLR or in a 32-bit Java JVM in-memory, then you are simply out of luck.

In fact, given the memory footprint of the core WebSphere Java process, on 32-bit Java platforms typically far less than 2 gigabytes of memory is available for its cache than the full 2 gigabytes the process can address. While the core .NET CLR memory footprint is smaller than WebSphere 6.0, part of the 2 gigabytes is also not available for the

---

[1] Some 32-bit applications, such as SQL Server 2000 and 2005 can use some or all of this 32-bit extended memory since they are specifically programmed to do so via a feature called Address Windows Extensions (AWE). So, for example, while a typical application on 32-bit Windows can only use 2GB of RAM (no matter how much is installed on the machine), SQL Server 32-bit Data Center Edition can actually use 64GB of RAM for its data cache. However, neither the .NET Common Language Runtime (CLR) or 32-bit Java JVMs can address more than roughly 2GB RAM on 32-bit platforms.

application itself on the .NET platform (typically 200-300 megabytes but this depends on the size of the application). So while developers have learned to successfully employ the advanced caching technologies available in both .NET and IBM WebSphere to increase application performance (for example, caching user-generated record sets so they do not have to be re-generated by the database on every user request), it's apparent that such applications can easily bump into the 2 gigabytes per-process limit on either platform. This in turn, can greatly limit the extent to which a middle-tier cache can be utilized on 32-bit platforms, creating a barrier to achieving higher levels of performance and overall transaction throughput.

64-Bit Platforms

A 64-bit computer such as an AMD Opteron-based server or an Intel EM64T-based server can theoretically address 2^{64} bytes (16 exabytes) of RAM. As of today, this means the memory addressability is essentially unlimited on 64-bit hardware. Current implementations of Windows Server 2003 64-bit now support up to 1 ***terabyte*** of physical memory (Standard Edition supports 32 gigabytes) installed on a 64-bit computer. Of course there are very few mainstream computers today where you would need 1 terabyte of RAM, considering this is 1,000 gigabytes of RAM. With a total virtual address space of 16 terabytes on Windows 64-bit platforms, an individual process on Windows Server 2003 (if it needs it) can now use up to 1 Terabyte of RAM directly, or ***500 times more RAM than available to an individual process on 32-bit Windows.***

In order to exploit the additional available RAM, the full software stack must also be 64-bit capable. This means for .NET, the.NET CLR must be 64-bit aware, and for Java, the JVM must be 64-bit aware. With .NET 2.0, Microsoft delivers 64-bit support with .NET for the first time. With a 64-bit OS and a 64-bit runtime, .NET 2.0-based applications can now utilize 500 times more memory for data such as server-based caches. Similarly, 64-bit JVMs such as the IBM JVM that ships with 64-bit IBM WebSphere 6.0 can also address extended memory-both on the Windows and Linux 64-bit platforms.

CachePerf: The Benchmark Application Scenario

In order to demonstrate and explore the potential impact a 64-bit hardware/software platform can have on application performance on the middle tier, we constructed a simple benchmark scenario. The scenario is an application that manipulates large objects that are stored in the database via a Web-based application. For this scenario, we stored a series of images, jpeg files each 500 kilobytes in size, as blobs in an Oracle 10G database. We then constructed a simple middle-tier application to retrieve these images and display them on a browser Web page. The application design includes a cache layer: when the application retrieves a row of data from the database (including the blob), the application stores that row as an object in an in-memory cache. The .NET version of the application was written in C# using ASP.NET and ADO.NET for the .NET platform. The Java version of CachePerf was written using JSPs, servlets and JDBC for the WebSphere 6.0 platform. The applications are 100% functionally equivalent, and they are coded in a similar fashion using best practices for each platform. It is important to note that the benchmark scenario measures the relative performance of each platform tested in terms of how efficient each platform is at:

1. Being able to cache large amount of data (which highlights the advantage of 64-bit platforms in terms of extended memory addressing).

2. Being able to receive process and deliver web-based requests for this data back to the Web user.

Hence, it does not isolate the performance of the cache mechanisms alone; rather it tests the full application platform from client to server for a simple scenario involving a server-based object cache. For .NET, the caching mechanism is the built-in .NET Cache API; for WebSphere it is the built-in Dynamic Cache Service. While each mechanism supports both object-level and page-level (output) caching, this benchmark tests object-level caching within the CLR and JVM application server processes to test a scenario where developers want to be able to retrieve and manipulate cached objects within their application logic.

We benchmarked all applications as follows on the exact same AMD Opteron-based hardware:

The Database Server Hardware

AMD 2-Processor Dual Core Opteron (4 Processors total) 2.4 GHz server (supplied by HardDrives Northwest)
16GB RAM
Gigabit networking
2 StoreCase Technology Fast RAID Arrays/SCSI controllers (14 drives total configured in RAID 0+1 configuration) for data files + a second identical storage array with separate controller for transaction logging.

The Application Server Hardware

HP DL585 4 x 1.8 GHz AMD Opteron Processor server
16GB RAM
Gigabit networking
HP Fast RAID array storage

The Application Server Operating Systems Tested

The application server was configured for separate tests with:

1. Windows Server 2003 Enterprise Edition 32-bit (SP1)
2. Windows Server 2003 Enterprise Edition 64-bit (SP1)
3. RedHat Linux Enterprise Server Release 4 64-bit[2]

[2] Note that for RedHat Linux servers, there is one distribution (32-64) for Opteron platforms that supports both 32-bit and 64-bit applications. Hence, to measure the performance of the 32-bit/Websphere version of the CachePerf application running on Linux, we used the 32-bit install of WebSphere 6.0.2.1 (including the 32-bit IBM HTTP Server, based on Apache HTTP Server 2.x) running on this distribution. For Windows, since ASP.NET relies on and is integrated with IIS 6.0, to test a full 32-bit stack we ran 32-bit Windows Server 2003 for x86 platforms for all 32-bit Windows tests on the HP AMD Opteron-based application server. While it is likewise possible to run 32-bit apps including .NET 2.0 and WebSphere on 64-bit Windows Server 2003 because of the integrated IIS

Given the 2 gigabyte per-process memory limitations of 32-bit Java heap sizes and 32-bit .NET CLR processes, the memory configuration of the machine dictates that even though the 32-bit operating system can recognize all the available 16 gigabytes of RAM in the machine, the 32-bit implementations of WebSphere 6 and .NET 2.0 can only use up to 2 gigabytes each. The benchmark is meant to demonstrate why 64-bit operating systems and hardware platforms matter: they remove the barrier for both individual Java and .NET processes to take advantage of the full memory in the machine.

The Application Server Software Tested

- We used the released-to-market version of .NET 2.0 for all .NET tests, available for free download from http://msdn.microsoft.com/netframework/.

- We used IBM WebSphere 6.0.2.3 Network Deployment Edition with the IBM HTTP Server (Apache) for all WebSphere tests.

- For the .NET tests, we used the .NET Data Provider for Oracle that Microsoft ships with the .NET Framework, used in conjunction with Oracle10G Client Release 1 (32-bit Windows) and the Developer Release of the Oracle10G Client for Windows x64 platforms (Oracle makes this available on their Web site—while they have a shipping version of the Oracle10G 64-bit Client for Intel Itanium, they were still in beta with their AMD Opteron/Intel EMT64 versions at the time this document was published).

- For the WebSphere tests, we used the Oracle thin JDBC driver (ojdbc14.jar) that ships with their 10G Release 1 Client software.

- Connection pooling was used across all tests (as is the default with WebSphere 6 and .NET).

- For complete tuning information for all platforms, please see the Appendix.

The Benchmark Database

The database used to store and retrieve the images consists of a single table, called "Image." The definition for the Oracle 10G table is detailed below.

Column Name	Data Type
IMAGEID	Number (Primary Key)
IMG_NAME	Varchar2 (50)
IMG_DATA	Blob

6.0 component (which is a 64-bit app on Windows Server 2003 for AMD Opteron platforms), to test a full 32-bit stack for .NET for a fair comparison to a full 32-bit WebSphere 6 stack on Windows Server 2003, all 32-bit tests for Windows Server 2003 were run on Windows Server 2003 Enterprise for x86 platforms, as opposed to running 32-bit versions of .NET and WebSphere on the 64-bit Windows Server 2003 OS. The goal was to always be running either a full 32-bit app server stack for .NET and WebSphere, or a full 64-bit stack for .NET and WebSphere.

IMG_CONTENTTYPE	Varchar2(50)
IMG_URL	Varchar2(50)

The database is loaded with 40,000 rows of data. Each image stored as a blob in the database is 500K for this benchmark.

The Benchmark Application

The benchmark application is a very simple browser-based application. It takes an input parameter from the query string, which is the primary key to look up an image from the database. The application then retrieves that record from the database and displays the image in an HTML page. For the purpose of the benchmark, the display of the image is accomplished by a separate JSP-Servlet (WebSphere) and ASP.NET (.NET) page so that the benchmark can be easily run with or without actually sending the image to the benchmark clients (this is determined based on another query string parameter from the clients). Once a record is retrieved from the database, its columns are mapped to a model class that is then cached in a middle tier cache so that subsequent requests do not need to access the database to work with the data. For our tests, the cache is set to expire every 8 hours so that during a benchmark run, once the middle tier cache is populated, it remains populated until the benchmark is complete. Throughput measurements for the benchmark (in transactions per second) are not taken until the cache is completely populated, or “warm”, for each specific test run.

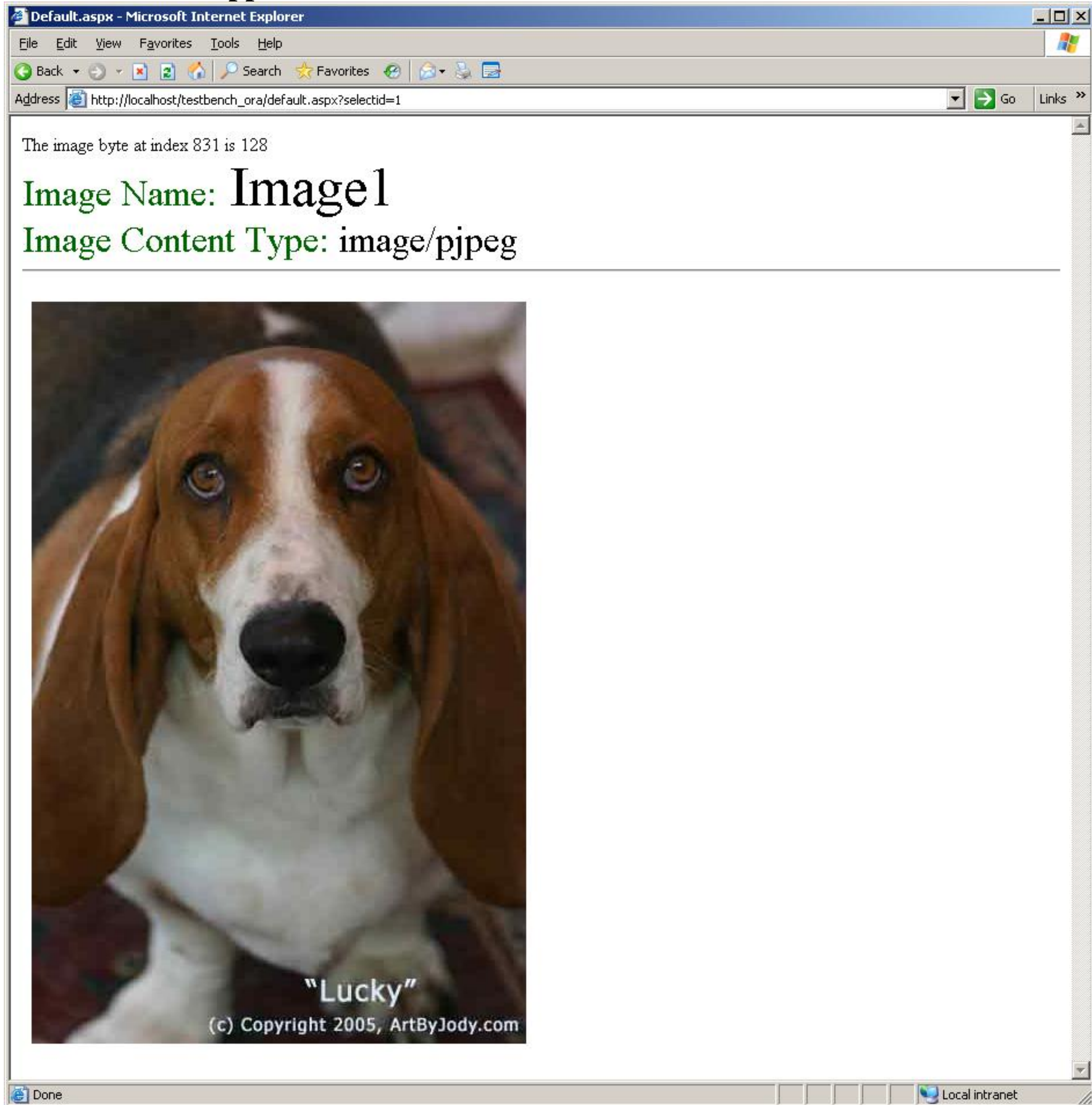
Thus it is important to note that CachePerf as tested here shows a “best-case” scenario: the difference between always being able to service items from the cache (64-bit) vs. memory-constrained scenarios on 32-bit platforms where the application server must continually make very expensive requests for large objects from the database. With downloadable code, however, customers can easily use the application as the basis for more complex scenarios involving smaller objects, mixes of different types of objects, and cache invalidations and/or timeouts during the benchmark runs.

As with any benchmark, these results are hence scenario-specific. Obviously real-world applications will have much more complex caching policies, a wide variety of transaction types, and different security and reliability requirements. Since the CachePerf benchmark tests only a single transaction type that interacts with the in memory cache mechanisms of .NET and WebSphere, the performance differentials between various benchmark runs are very dramatic. Customer applications will not see such dramatic differentials because they will have many other transaction types that will mandate that not all data can be cached in the application. However, the use of caching still offers real-world applications the potential for very dramatic performance gains. And the ability of 64-bit Java and .NET platforms to address extended memory can result in very dramatic performance boosts for mid-tier applications that can take advantage of extended memory beyond 2 gigabytes for their respective caches.

Thus, the results of this benchmark are both interesting and useful, because they measure and end-to-end Web-based scenario that heavily stresses the application servers’ Web stack as well as the backend JVM and CLR cache layers typically used in real-world

applications. While the performance of a real-world application will likely not match the results shown by this benchmark, we can draw general conclusions about the benefits of being able to cache more data in 64-bit environments, and about the relative performance of .NET 2.0 versus WebSphere 6 for middle-tier Web-based caching as required by large-scale applications with many concurrent users.

The CachePerf Application Run from a Browser



A Note on Object Caching vs. Servlet Caching In WebSphere 6

The Dynamic Cache Service in IBM WebSphere 6 offers several core methods to achieve object-level caching for the CachePerf benchmark. These include pure object caching (via their Distributed Map technology and a server-based cache instance); and servlet caching. We tested both, and found the results for this benchmark roughly equivalent using both mechanisms, although object-level caching proved to be more efficient in terms of memory usage for this benchmark. Each technique uses the same IBM Dynamic Cache Service as its underlying caching mechanism. Object caching, however, tends to offer the developer much finer grained control over the objects in the cache and how they can be used within with program logic. Hence, we ran the application using IBM WebSphere object caching-- however customers can easily modify the provided code to run the benchmark with servlet caching.

The Java Code

For a full listing of the Java source see the Appendix. Also, the Java code can be downloaded as an EAR file with full source from MSDN. The essential elements of the logic for the Java/JDBC version are as follows:

1. The browser makes a request to the MainServlet passing an image id on the query string.
2. If that object is not already cached (either via servlet caching or object caching) by the IBM Dynamic Cache Service, then the MainServlet calls into the data access class (DAL.java) to retrieve the item from the database using JDBC.
3. The columns from the database, including the image itself, are instantiated into a model class (item.java). The image binary data (a blob in Oracle) is stored in a bytearray[] that is a member of this class. The item class is then placed into the cache with an 8 hour expiration.
4. The servlet then uses test.jsp to format the results. The results include a display of the item name, its content type, and a randomized lookup into an individual byte of the image byte array. The test.jsp page is processed on every request (it is not cached, only the associated item model class is cached). This is by design, so that we demonstrate programmatic manipulation of cached data for both .NET and Java. In this case, that manipulation consists of simply looking up and displaying a single random byte of the image in the browser at the top of the Web page. So even if an image is already in the cache, the JSP page will still generate unique data for the user on every request via the random lookup into the cached image byte array. Note that using page-level caching, which are available to both .NET and IBM WebSphere, it would be possible to cache the entire output of the JSP (or ASPX) page, but in this case no objects would be available individually for further programmatic manipulation (such as the item class), and the JSP page would be processed one time only for each unique image id requested (at least until cache invalidation for the item requested). Hence it would not generate unique views into the cached data for each user request since even the random byte array lookup results would be cached. This would represent a very different

benchmark. Instead, for this benchmark we want to demonstrate the ability to cache and work with objects using Java and .NET, not just to return cached pages. This approach simulates a common business scenario such as the caching of a dataset as an object such that a user can still manipulate and work with the object even though the application does not have to re-request that data from the database (for example, re-sorting, filtering, etc. on previously cached data objects/record sets). This can be considered 'object-level- caching as opposed to page-level caching.

5. Additionally, if the browser does not send the query string parameter 'displayitem=false', then the browser will make a separate request to the ImageServlet, which actually displays the image in the browser. Note that for all benchmark runs including .NET and Java, the 'displayitem=false' is passed in such that we do not actually pass the image back to the benchmark client. Instead, just the backend caching and programmatic manipulation of the cached objects via the JSP page are processed, and simple HTML is passed back to the clients for validation. This is done so that we do not saturate and stress the network and/or client CPUs by transmitting all of that image data from server to clients which would need to process all of that image data. The goal of the benchmark is to isolate and measure the performance of mid-tier object caches and web-based delivery of cached content, not to measure delivery of static images from server to client.

The following model class is used to cache the record from the database via object caching:

Item.Java (model class to hold and cache a database record)

```
package ImageBench;

import java.io.Serializable;

/*
 * Created on Oct 1, 2005
 *
 * TODO To change the template for this generated file go to
 * Window - Preferences - Java - Code Style - Code Templates
 */

/**
 * @author Administrator
 *
 * TODO To change the template for this generated type comment go to
 * Window - Preferences - Java - Code Style - Code Templates
 */
public class Item implements Serializable{

        //
        // TODO: Add constructor logic here
        //

        // Internal member variables
        private int _imageId;
        private String _imageName;
        private byte[] _imageData;
        private String _imageContentType;
```

```
        /// <summary>
        /// Default constructor
        /// </summary>
        public Item()
        {
        }

        /// <summary>
        /// Constructor with specified initial values
        /// </summary>

        public Item(int imageID, String imageName, byte[] Image, String ImageContentType
)
        {
            this._imageId = imageID;
            this._imageName = imageName;
            this._imageContentType = ImageContentType;
            this._imageData = Image;


        }


        // Properties
        //image ID - Unique Identifier
        public int getImageID()
        {
             return _imageId;
        }
        //filename of image
        public String getImageName()
        {
            return _imageName;
        }
        //content mime type (used for generating the image)
        public String getImageContentType()
        {
             return _imageContentType;
        }
        public void setImageId(int imagecounterID){

            _imageId = imagecounterID;
            return;
        }

        public byte[] getImage()
        {
            return this._imageData;
        }

    }
```

The following is a code snippet showing how items are retrieved from and placed into the object cache in WebSphere:

```
        item = (Item) dm1.get(imageID);
        if (item==null)
        {
                int id = Integer.parseInt(imageID);
                DAL dal = new DAL();
                item = dal.getItemImageItem(id);
                dm1.put(imageID,item,1,28800,EntryInfo.NOT_SHARED,null);
        }
```

The .NET Code

For a full listing of the .NET/C# source see the Appendix. Also, the code can be downloaded from MSDN. The essential elements of the logic for the .NET/ASPX version follow that for the Java version:

1. The browser makes a request to the default.aspx web form, passing an image id on the query string.
2. If the item requested is not found in the .NET middle tier cache, then the default web form calls into the data access class (DAL.cs) to retrieve the item from the database using ADO.NET.
3. The columns from the database, including the image itself, are instantiated into a model class. The image binary data (a blob in Oracle) are stored in a bytearray[] that is a member of this class. The class is then cached for that item using the .NET Cache API which is the main caching mechanism for ASP.NET applications.
4. The default.aspx page then formats the results for display. The results include a display of the item name, its content type, and a random lookup into the image byte array, just as with the Java version. The default.aspx page is processed on every request, including the random lookup into one byte of the image byte array to demonstrate programmatic access of the cached data within managed code. So even if an image is already in the cache via the .NET cache API, the ASPX page will still generate unique data on every request via the random lookup into the cached image byte array—just like the JSP/Java equivalent.
5. Additionally, if the browser does not send the query string parameter 'displayitem=false', then the browser will make a separate request to displayimage.aspx, which actually displays the image in the browser. Note that for all benchmark runs including .NET and Java, the 'displayitem=false' is passed in such that we do not actually pass the image back to the benchmark client (and hence saturate our clients). Instead, just the backend caching and programmatic manipulation of the data is processed.

The following model class is used to cache the record from the database via the .NET cache API:

TestImageItem.cs

```
using System;

namespace DAL
{

    /// <summary>
    /// Summary description for TestImageInfo
    /// </summary>
    [Serializable]
    public class TestImageInfo
    {
        //
        // TODO: Add constructor logic here
```

```
        //

        // Internal member variables
        private int _imageId;
        private string _imageName;
        private byte[] _imageData;
        private string _imageContentType;




        /// <summary>
        /// Default constructor
        /// </summary>
        public TestImageInfo()
        {
        }

        /// <summary>
        /// Constructor with specified initial values
        /// </summary>

        public TestImageInfo(int imageID, string imageName, byte[] Image, string
ImageContentType )
        {
            this._imageId = imageID;
            this._imageName = imageName;
            this._imageContentType = ImageContentType;
            this._imageData = Image;


        }


        // Properties
        //image ID - Unique Identifier
        public int imageID
        {
            get { return _imageId; }
        }
        //filename of image
        public string imageName
        {
            get { return _imageName; }
        }
        //content mime type (used for generating the image)
        public string imageContentType
        {
            get { return _imageContentType; }
        }
        public void setImageId(int imagecounterID){

            _imageId = imagecounterID;
            return;
        }

        public byte[] getImage()
        {
            return this._imageData;
        }
    }
}
```

You can see that the model class is essentially equivalent to the Java version of the class.

The following code is used in default.aspx to cache the item class using the .NET Cache API. Both .NET and WebSphere have mechanisms for setting cache priorities and

invalidating the cache based on dependencies, including time. As with WebSphere, items added to the cache are set to expire after 8 hours.

Code Snippet to Retrieve/Add Items to .NET Cache via the .NET Cache API

```
Object cacheddata = Cache.Get(selectid.ToString());
            if (cacheddata != null)
            {
                testimageinfo = (TestImageInfo)cacheddata;
            }
            else
            {
                DAL.DAL myDAL = new DAL.DAL();
                testimageinfo = myDAL.GetTestImage(selectid);
                Cache.Add(selectid.ToString(), testimageinfo, null, DateTime.MaxValue, 
new TimeSpan(8, 0, 0), CacheItemPriority.High, null);
            }
```

Configuring the Amount of RAM available for Java and .NET Cache Entries

The goal we had for CachePerf on the platforms tested was to configure as much memory as possible for object caching while still making most efficient use of overall system resources—avoiding operating system swapping and out-of-memory exceptions, for example. With .NET, the administrator configures the maximum amount of RAM available for the ASP.NET CLR runtime via the IIS metabase. If memory usage of an individual worker process exceeds this threshold, a new worker process is started automatically and the old one is killed off after all current requests have been processed. This helps prevent runaway applications consuming all available memory on the machine. It also ensures that the cache entries can never consume all available memory, such that .NET will not throw 'out of memory exceptions' when using the ASP.NET cache, and not crash. This means .NET applications can deliver high availability, even under extreme load since .NET is automatically managing and monitoring the memory usage.

IBM WebSphere, on the other hand, sets the maximum physical *number* of entries that can be inserted into the cache via the administrator console (no matter the size of the cached objects). Hence IBM WebSphere requires a bit of manual calculation or trial and error to determine the maximum entries that can be allowed given the memory available based on the Java heap size. If this setting is set too high, the application can easily exceed the total amount of memory configured in the Java heap, which will cause the Java "OutOfMemoryError" error, and actually crash the application server and cause a heap dump after a few minutes. Of course this has direct implications on application availability and reliability.

While WebSphere has sophisticated caching technology, the simplistic cache size tuning of the IBM WebSphere Dynamic Cache Service is not ideal. While it works for a benchmark where all cached objects are exactly the same size, it will not typically work for a real application scenario where the object cache is populated by a mix of objects of different sizes in unknown ratios, something that is typical with unknown and varying mixes of queries submitted by users of the application. In these cases, the WebSphere

administrator must work with probabilities and guesses. He or she must estimate the mix of various objects in the cache, and then compute the maximum number of cache entries based on the actual size of the objects they estimate will ever be requested to be added to the cache (a number that is not apparent to the administrator, and which can change from day to day based on differing usage patterns of the application). Then it would be prudent to allow for a margin of error and always set the setting much lower than estimated to avoid application server crashes due to many Java "OutOfMemoryError" errors. If the administrator guesses incorrectly on the object sizes or on the relative proportion of various objects in cache, then WebSphere Application Server can crash at runtime once the available heap space is exceeded.

We believe the .NET approach is much better—the administrator knows how much RAM is installed in the server, and can simply configure the cache size to never exceed this limit. If more RAM is added, then this can easily be changed in the .NET configuration file for that server. In this way .NET will never attempt to use more memory than possible for its cache.

Configuring the IBM WebSphere Heap Size

For IBM on the 32-bit Windows platform, the maximum heap size we could use was 1640 megabytes RAM (the theoretical limit is 2048 megabytes or ~2 gigabytes, however we found the default install of WebSphere will not start on 32-bit Windows with a heap size greater than 1640 megabytes, possibly due to the fact the JVM requests a contiguous block of memory for the heap, and Windows 32-bit doles out up 1,640 megabytes of contiguous memory per process). On Linux, we were able to configure the 32-bit WebSphere heap size to 2,300 megabytes of RAM, which represents more RAM available for cache entries on Linux vs. Windows for 32-bit WebSphere. For 64-bit WebSphere on Windows, we were able to run with a heap size of 14,500 megabytes without incurring OS swapping. For 64-bit WebSphere on Linux, we were able to also run with a heap size of 14,500 megabytes without incurring OS swapping (see the heap size setting discussion for a more detailed explanation). For WebSphere, we found the maximum number of entries allowed in the cache (within a margin of roughly 500 entries) via trial and error -- simply by running with an estimate of 500K per entry up to available heap size, then running the test with a max cache entry value until we found the point at which WebSphere would not crash/heap dump. For 32-bit Windows/WebSphere, this number was 2500 entries. For 32-bit WebSphere on Linux this was 3000 entries. For 64-bit WebSphere the numbers were 27,000 allowable cache entries on both Windows 64-bit and Linux 64-bit, respectively given our matching configured Java heap sizes on these platforms.

Configuring the .NET Cache Size

As discussed earlier, .NET does not work by configuring the number of allowable cache entries (as does WebSphere). Instead, it works by letting administrators specify the amount of memory to make available to the ASP.NET cache. This avoids the WebSphere issue of possible catastrophic crashes and heap dumps due to eating up all available memory with the cache. .NET will automatically age items from the cache based on their priority and an LRU mechanism should requested cache inserts be greater

than the available memory set aside for the cache (WebSphere uses an LRU algorithm once the number of requested cache inserts exceeds the maximum configured allowable cache entries). In addition, unlike IBM WebSphere, ASP.NET has an added reliability feature to automatically recycle the CLR ASP.NET worker process should memory exceed a set threshold. For this benchmark, we turned worker process recycle off, however, since we made sure to configure the cache size itself to an appropriate amount of memory for both 32-bit Windows and 64-bit Windows, allowing .NET to automatically keep the cache within these memory-usage limits. For 32-bit .NET, we set the cache size limit to 1,700 megabytes, and we set it to 14,500 megabytes for 64-bit .NET. We made sure to leave enough memory in 32-bit mode for the CLR runtime and application logic itself, and on 64-bit .NET we made sure to leave enough free memory to avoid OS swapping.

Summary of CLR and JVM Memory Settings

IBM WebSphere 6.0.2.3

Platform	**Min Heap Size**	**Max Heap Size**	**Max Cache Entries Setting Given ~500K per cached item**	**Rationale: Maximize the memory allotted to cache entries for each platform without negatively impacting performance or reliability[3]**
32-bit IBM WebSphere 6.0.2.3 on Windows Server 2003 32-bit	1640MB	1640MB	2500	Maximum setting for 32-bit WebSphere heap on Windows. This means the cache only has available 1640 MB minus the memory footprint of WebSphere + app logic. We found WebSphere will eventually throw out of memory errors and then crash/heap dump if max cache entries is > 2500 entries for the benchmark on this platform given this setting.
32-bit IBM WebSphere 6.0.2.3 on RedHat Linux Release 4 for AMD Opteron	2300MB	2300MB	3000	Maximum 2GB Java heap on 32-bit Linux platforms. This allows max cache entries for WebSphere to be set at 3000 while still running w/o out-of-memory errors/crashes.
64-bit IBM WebSphere	14500MB	14500MB	27,000	90% of available memory on the machine, leaving 10% free for OS to

[3] Customers caching large amounts of data will typically not be able to configure for this amount of heap space for caching on 32-bit platforms because their applications will likely have much larger footprints than the simple CachePerf application. Hence, 64-bit memory addressing for many customers may take on even more significance.

6.0.2.3 on Windows Server 2003 64-bit (x64)				avoid swapping memory. Max cache entries can go to 27,000 without causing out of memory exceptions and WebSphere crash/heap dump.
64-bit IBM WebSphere 6.0.2.3 on RedHat Linux Release 4 for AMD Opteron	14500MB	14500MB	27,000	90% of available memory on the machine, leaving 10% free for OS to avoid swapping memory. Max cache entries can go to 27,000 without causing out of memory exceptions and WebSphere crash/heap dump.

Microsoft .NET 2.0

Platform	Cache privateBytesLimit	Rationale
32-bit .NET 2.0 on Windows Server 2003 32-bit	1700MB	~85% of available 2GB memory for a 32-bit process set aside for cache. This leaves ~300MB free for the CLR itself while still operating within the 2GB per-process limit on 32-bit platforms.
64-bit .NET 2.0 on Windows Server 2003 64-bit (x64)	14500GB	~90% of available physical memory set aside for cache. This still leaves plenty of space to avoid OS swapping.

The Test Methodology

The tests were conducted with Mercury LoadRunner 7.51 with 20 physical agent computers to simulate load. The LoadRunner test scripts are published on MSDN with the source code for the benchmark suite. The scripts make continual requests to the application for a randomly selected image id. The random image id variable submitted by each client ranges from a minimum of 1, to a maximum stepped from 500 to 40,000 to collect a range of datapoints. The more possible query combinations from the clients, the more memory the application will want to use for its cache. Each step that increases the possible unique queries is conducted as a separate test to generate a separate data point.

This varies the maximum number of items that are added to the cache for a given run. Hence we can see how each 32-bit and 64-bit platform responded as the maximum number of user queries reached and then exceeded the available amount of memory for the middle tier cache. With each cached blob representing roughly 500K, the benchmark run for 10,000 unique possible queries submitted randomly on an ongoing basis from the driver software, for example, represents roughly 5000 megabytes of data in the cache once it's fully populated. In this way, customers can see the dramatic impact 64-bit memory addressing has on TPS rates for large middle tier cache/memory intensive scenarios. 32-bit JVMs and 32-bit .NET simply cannot cache this amount of data, and entries must be continually ejected from the cache and re-requested from the database when queried again. 64-bit .NET and Java, however, have no problem caching 10,000 unique query results representing 5 gigabytes of data on the middle tier.

Each benchmark pass was run with no client think time, and extensive tuning and testing was done to ensure IBM WebSphere achieved its maximum throughput, as detailed in the appendix. In all cases, server saturation was achieved for the benchmark runs where all requested items could be maintained in the cache. This was evidenced by >97% CPU server saturation in these scenarios (both 32 bit and 64 bit Windows and Linux). In the benchmark where all items could not be maintained in the cache by either .NET or WebSphere, the application servers spent most of their time in wait-states waiting for data to be retrieved from the database. In these cases no amount of additional user load could push the application servers beyond about 30% CPU utilization.

When operating at peak throughput (all items maintained in the cache with no database retrieval necessary), it took 40 client threads to fully saturate the application servers. The number of requesting client threads was not varied for different benchmark steps---we ran consistently with 40 client threads against WebSphere (enough to saturate and achieve peak throughput in non-memory constrained scenarios and accurately report peak throughput achieved); and with 40 client threads against .NET.

The IBM WebSphere Cache Offload-to-Disk Feature

The IBM WebSphere 6 application server has a built-in cache feature that is not available to core .NET, although it is available as a free downloadable extension to .NET on MSDN as an application block (with full source code). This feature, called "Dynamic Cache Disk Offload", enables the application server to offload entries in the memory-based cache to disk when the number of items requested to be added to the cache exceeds the set maximum cache entry setting. This extends the number of cacheable entries to limits determined only by the amount of disk space configured for the server. This feature can have a potentially positive or negative impact on performance in memory-constrained scenarios where all requested cached items cannot fit in-memory –which is especially true for 32-bit IBM WebSphere and 32-bit .NET given the 2 gigabyte process limit. We examined the impact of the IBM WebSphere 6 offload-to-disk feature in an additional benchmark scenario with the results shown later in this paper.

Specifically, we conducted a second benchmark pass to simulate an application that does other work in addition to just making requests for cacheable data, which accurately reflects the workload mix in a typical customer application. The results in our test show that in such a scenario, the amount of additional strain placed on the application server to access the disk cache may in many scenarios outweigh the potential benefits. The application server must work very hard once the disk-based portion of the cache starts to see heavier and heavier access rates (in terms of CPU usage due to heavy disk I/O, including expensive serialization/deserialization operations to deconstruct/reconstruct objects to/from disk). This, in turn, can actually decrease the overall transaction throughput of the application server much like heavy use of an OS swap file can negatively impact performance. So any offload-to-disk feature must be used with care. For further information and the results of the benchmark that illustrate this effect, refer to the section titled "IBM WebSphere 6 Offload-To-Disk" in this paper.

.NET customers, if interested in such a feature, can download the Caching Application Block that is part of the Microsoft Patterns and Practices Site at:

- http://www.msdn.microsoft.com/practices/guidetype/AppBlocks/default.aspx?pull=/library/en-us/dnpag2/html/caching1.asp

Also, as suggested reading, please visit:

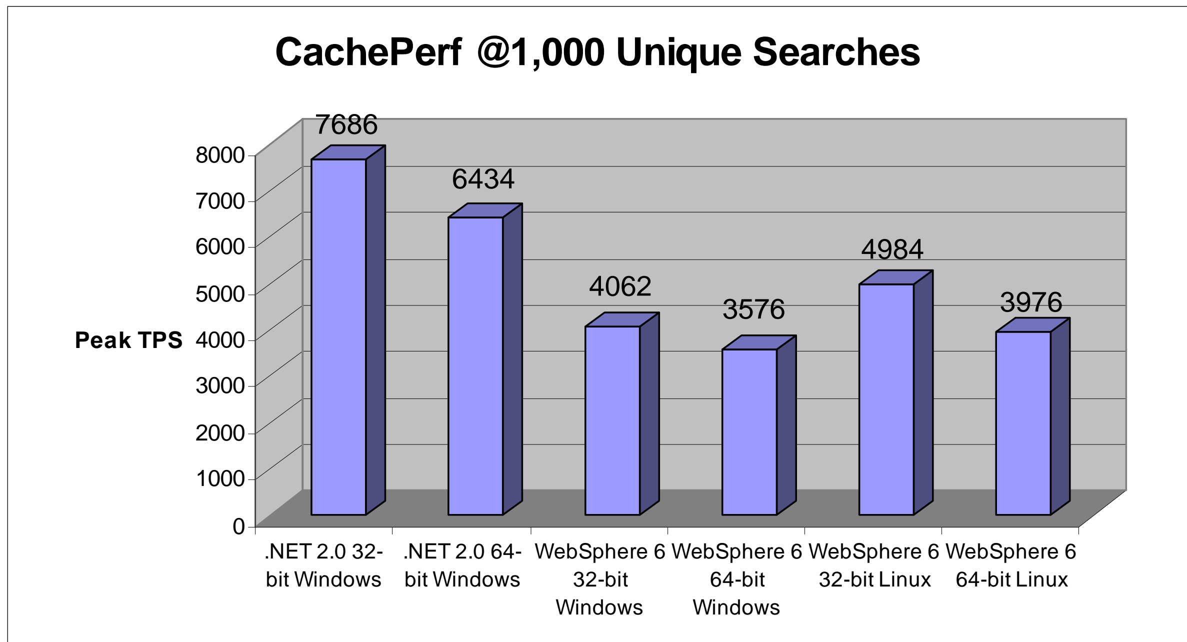
- http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnbda/html/CachingArch.asp

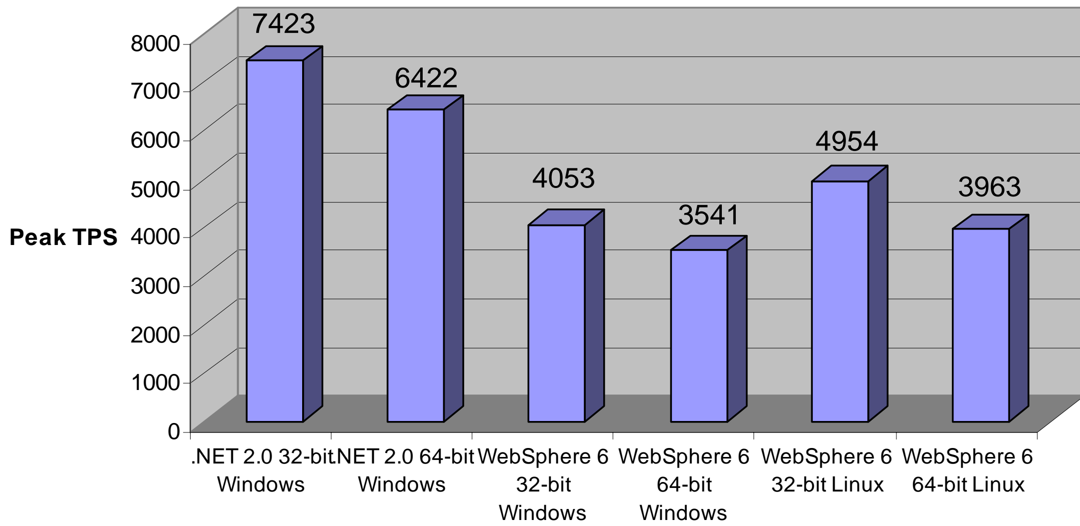
The Results in Detail

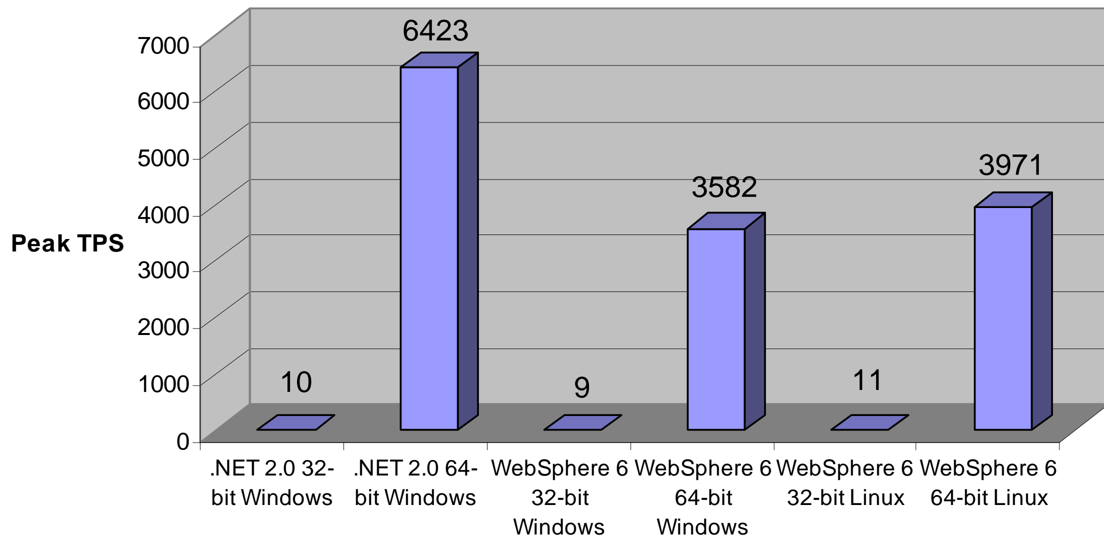
Comparisons

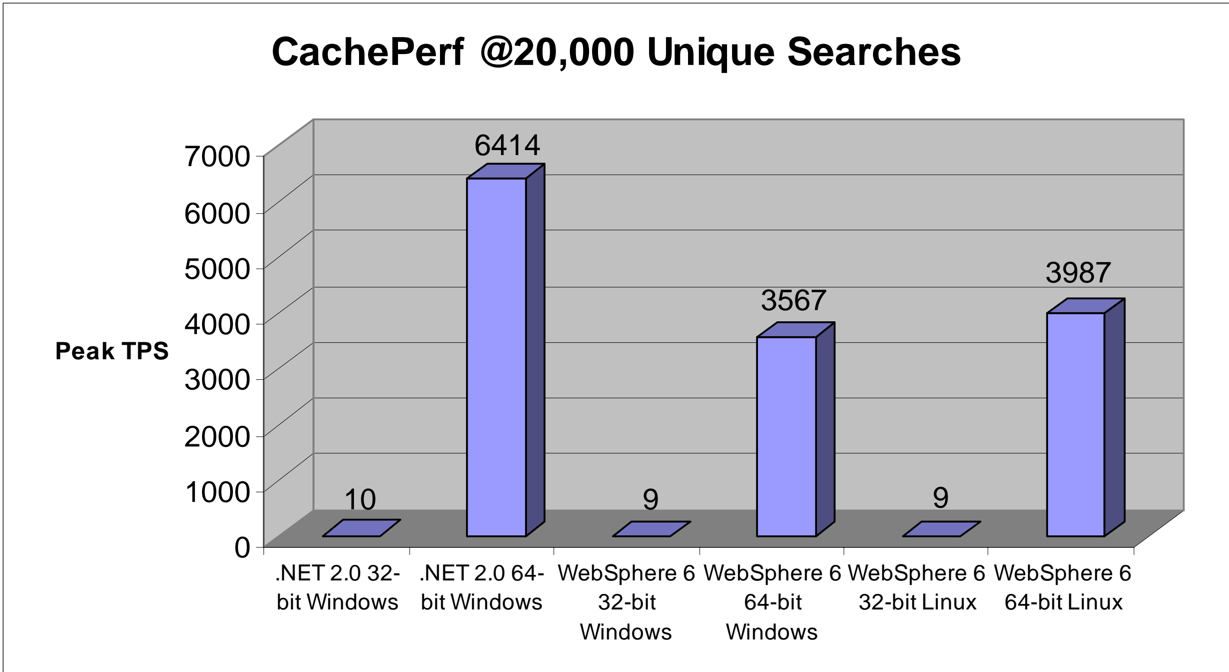
The following charts compare the peak TPS rates achieved by the middle tier application as tested at 1,000, 2000, 10,000 and 20,000 unique possible queries submitted by the clients on an ongoing basis.

They illustrate that 32-bit platforms when not memory constrained, can actually operate faster than 64-bit platforms for typical backend business application processing. However, they also illustrate how 32-bit platforms can quickly become memory-constrained, and how a 64-bit platform can ultimately lead to much higher performance once the memory requirements of the application can no longer be efficiently met within a 2 gigabyte process space.



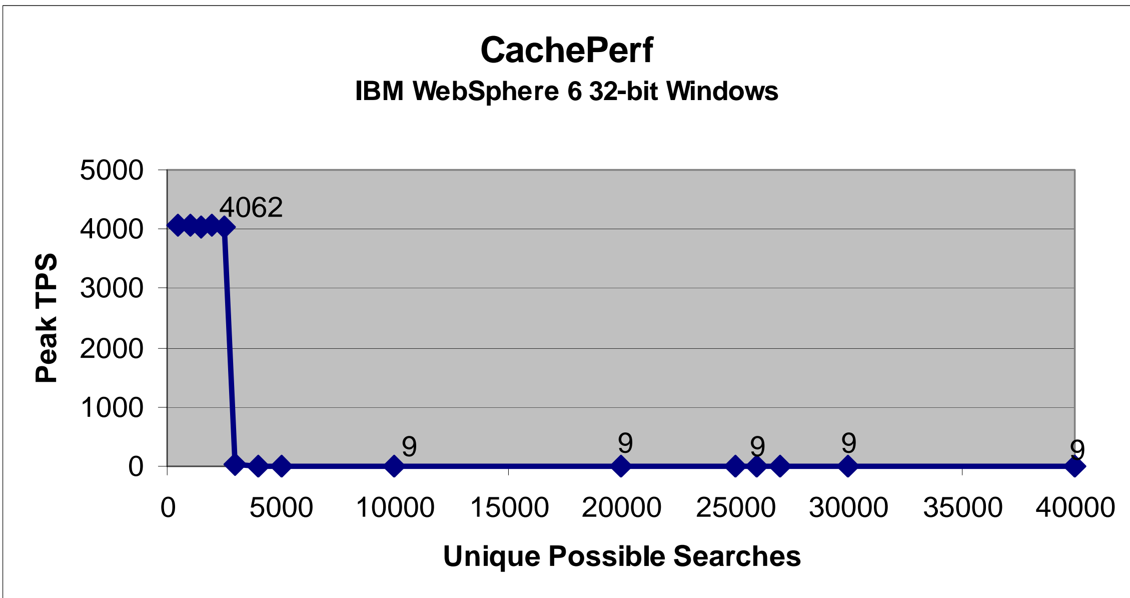

CachePerf @2,000 Unique Searches
Peak TPS
8000
7000
6000
5000
4000
3000
2000
1000
0
7423
6422
4053
3541
4954
3963
.NET 2.0 32-bit Windows
.NET 2.0 64-bit Windows
WebSphere 6 32-bit Windows
WebSphere 6 64-bit Windows
WebSphere 6 32-bit Linux
WebSphere 6 64-bit Linux

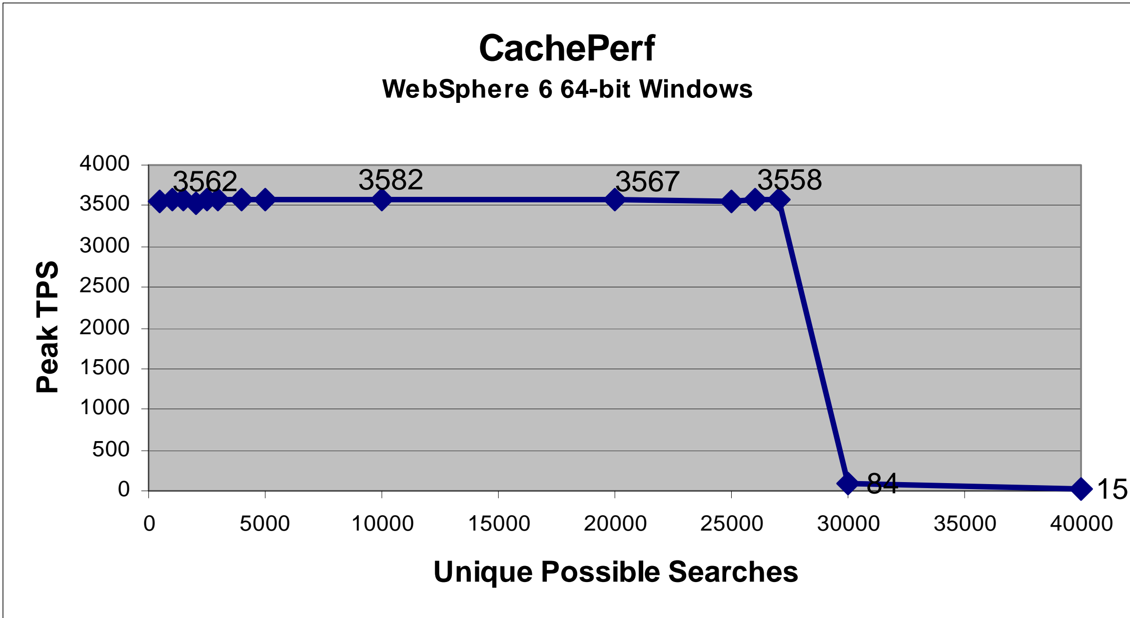

CachePerf @10,000 Unique Searches
Peak TPS
7000
6000
5000
4000
3000
2000
1000
0
10
6423
9
3582
11
3971
.NET 2.0 32-bit Windows
.NET 2.0 64-bit Windows
WebSphere 6 32-bit Windows
WebSphere 6 64-bit Windows
WebSphere 6 32-bit Linux
WebSphere 6 64-bit Linux

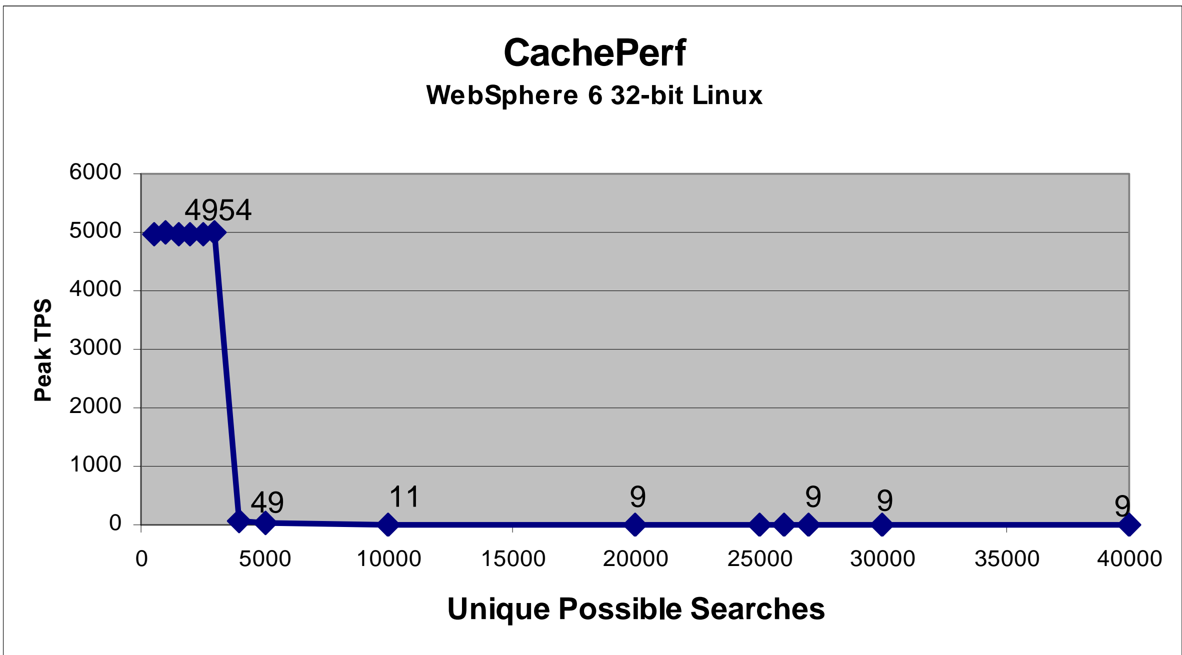


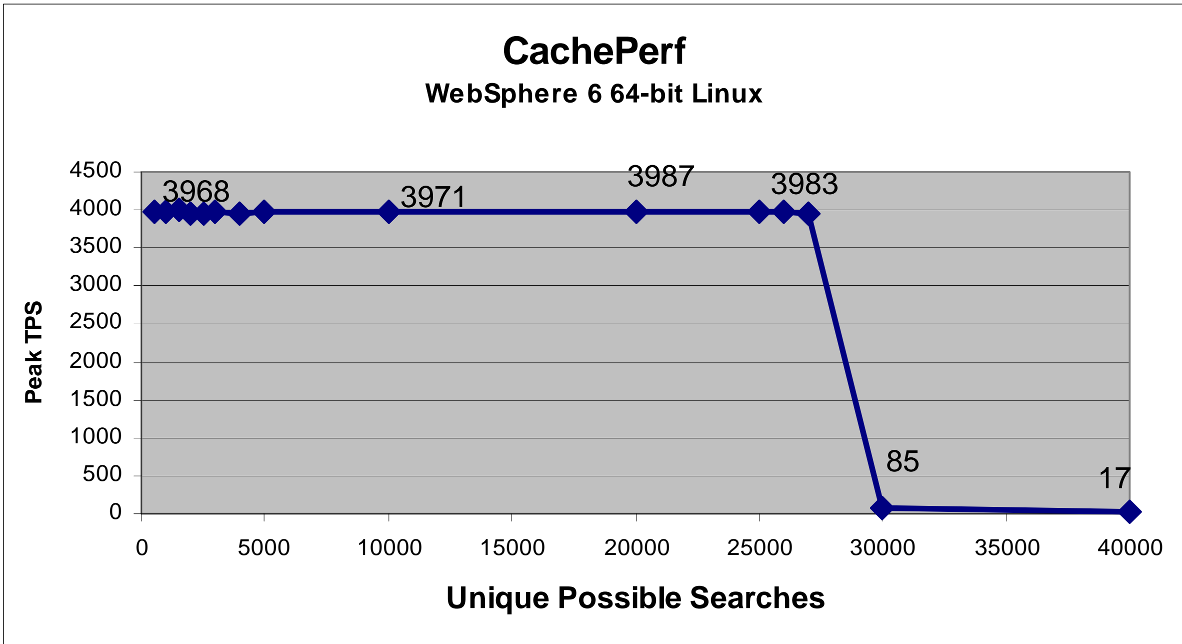
Breakdown by Platform

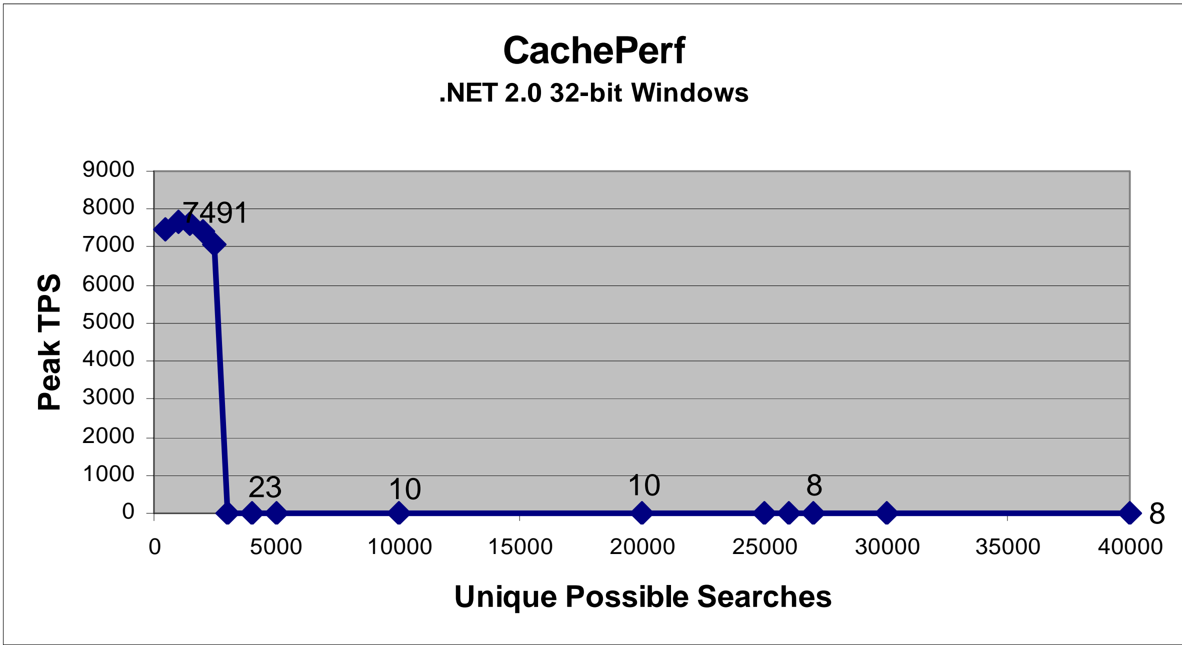
The following charts show each platform as the number of unique, continually submitted queries is increased for each separate benchmark run. All data points are taken after the cache is fully populated.

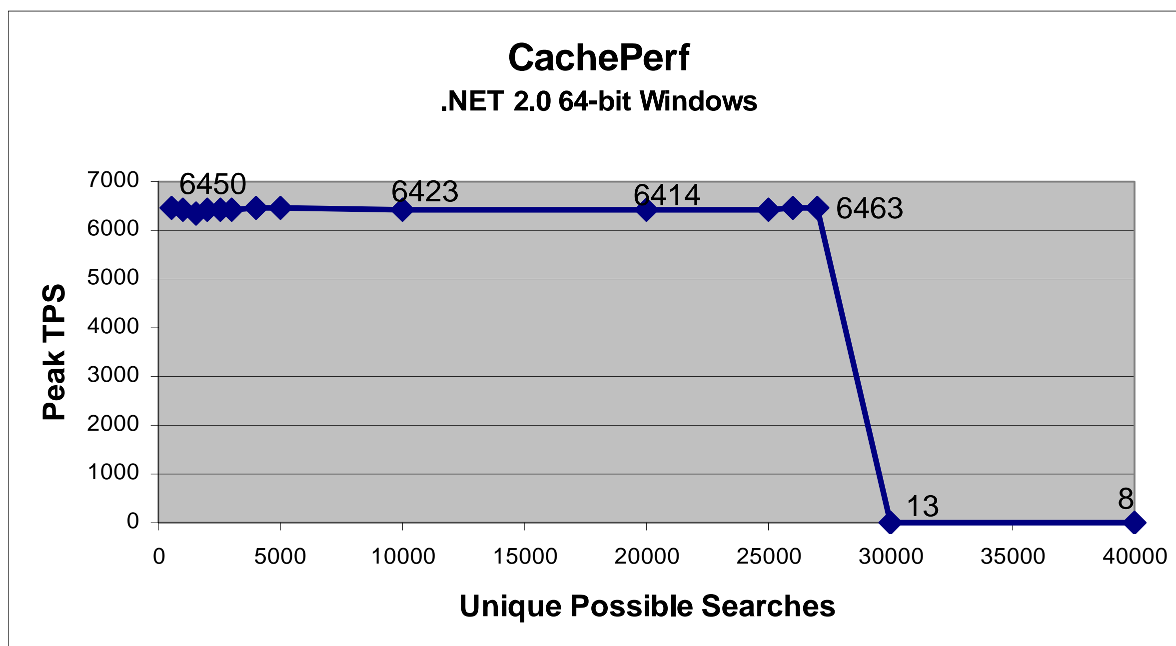



CachePerf
WebSphere 6 64-bit Windows
Peak TPS
4000
3500
3000
2500
2000
1500
1000
500
0
3562
3582
3567
3558
84
15
0
5000
10000
15000
20000
25000
30000
35000
40000
Unique Possible Searches


CachePerf
WebSphere 6 32-bit Linux
Peak TPS
6000
5000
4000
3000
2000
1000
0
4954
49
11
9
9
9
9
0
5000
10000
15000
20000
25000
30000
35000
40000
Unique Possible Searches


CachePerf
WebSphere 6 64-bit Linux
Peak TPS
4500
4000
3500
3000
2500
2000
1500
1000
500
0
3968
3971
3987
3983
85
17
0
5000
10000
15000
20000
25000
30000
35000
40000
Unique Possible Searches


CachePerf
.NET 2.0 32-bit Windows
Peak TPS
9000
8000
7000
6000
5000
4000
3000
2000
1000
0
7491
23
10
10
8
8
0
5000
10000
15000
20000
25000
30000
35000
40000
Unique Possible Searches



Tabular Results

Unique Submitted Searches	.NET 2.0 32-bit Windows	.NET 2.0 64-bit Windows	WebSphere 6 32-bit Windows	WebSphere 6 64-bit Windows	WebSphere 6 32-bit Linux	WebSphere 6 64-bit Linux
500	7491	6450	4053	3562	4967	3968
1000	7686	6434	4062	3576	4984	3976
1500	7594	6357	4044	3572	4982	4015
2000	7423	6422	4053	3541	4954	3963
2500	7082	6426	4045	3575	4962	3961
3000	23	6406	23	3568	4989	3989
4000	16	6478	14	3575	49	3964
5000	14	6452	12	3569	24	3980
10000	10	6423	9	3582	11	3971
20000	10	6414	9	3567	9	3987
25000	8	6432	9	3558	9	3983
26000	8	6451	9	3578	9	3968
27000	8	6463	9	3586	9	3954
30000	8	13	9	84	9	85
40000	8	8	9	15	9	17

Discussion of Results

- **32-bit WebSphere 6 on Windows.** For benchmark runs with only 500-2500 possible random searches, all items can be maintained in memory, so throughput is at its maximum, or roughly 4000 requests processed per second (TPS). But once the maximum number of user searches exceeds 2500, throughput rapidly

falls as the cache hit ratio quickly falls, given the cost of pulling 500K blob images from any database is quite expensive. Throughput falls to a minimum of around 9 TPS. If the blob items were much smaller than the 500K used in this benchmark, more entries could be maintained in the cache, and the peak TPS rate would be maintained for many more test runs beyond 2500 possible random queries. It's not the number of unique possible searches that matters, but rather the multiple of the number of unique searches times the size of each object being cached. It can also be expected with a smaller object size, the minimum TPS even when no items are cached (or the cache hit ratio is extremely small) would be much higher, because the database bottleneck would be greatly reduced. This would also be true for .NET 2.0, of course.

- **32-bit WebSphere 6 on Linux.** This data shows that with a slightly bigger maximum heap size possible, WebSphere 32-bit on Linux is able to maintain more items in the cache than when running on Windows 32-bit. Peak throughput is roughly 4900 TPS. Peak throughput is maintained up until the benchmark run with 3500 possible random queries (vs. 2500 for WebSphere 32-bit on Windows). Once the possible random queries exceeds this number, 32-bit WebSphere on Linux has no more available heap space to hold all the requested items. As with 32-bit WebSphere on Windows, throughout rapidly falls off at this point until it reaches a minimum of roughly 9 TPS as more and more queries must be continually re-submitted to the remote database.

- **32-bit .NET on Windows Server 2003 32-bit.** The .NET CLR can maintain all entries in the cache up to 2500 unique queries with our cachePrivateBytes setting at 1700MB. The peak throughput of .NET 2.0 32-bit is over 52% better than that of WebSphere 6 32-bit (Linux) at roughly 7600 TPS. This may have as much to do with more efficient Web processing in ASP.NET vs. IBM WebSphere as it has to do with any inherent differences in cache access performance itself on the two different application server platforms. Again, this test does not isolate just cache access speeds—it is an end-to-end scenario that includes server-side manipulation of the cached content and delivery of a subset of this content to Web-based clients.

- **64-bit WebSphere on Windows Server 2003 64-bit.** 64-bit WebSphere on Windows 64-bit was able to utilize dramatically more memory in the Java heap on our 16GB computer, leaving plenty of memory to cache up to 27,000 entries without any OS swapping or out of memory errors in the JVM. While it's peak throughput at 500-2500 possible random searches was lower than that of 32-bit WebSphere on 32-bit Windows, it was able to maintain a huge performance advantage over its 32-bit WebSphere counterparts beyond 2500 unique queries (~3500 TPS vs. 10 TPS) as the extended memory addressing of the 64-bit platform allowed all of these entries to be maintained in memory for subsequent user requests. This also, of course, has a huge benefit for the remote database server as well—its load is at zero in this scenario as long as all requested query

results can be maintained on the middle tier. This frees the database to more efficiently service other applications.

- **64-bit WebSphere on RedHat Linux 64-bit.** The results show the expected similar pattern as 64-bit WebSphere on Linux. The results also show slightly better peak performance for WebSphere on Linux than on Windows Server in this scenario. One possible reason might be the inherent differences between the architecture of the IBM HTTP Server (based on Apache) that ships with WebSphere for Linux vs. the architecture of the version that ships with WebSphere for Windows. This product is multi-threaded and capable of spawning multiple processes (Apache instances) on Linux. On Windows Server, IBM does not yet provide a version capable of spawning multiple processes (athough it is multi-threaded). This is true of both 32-bit and 64-bit implementations of the IBM HTTP Server.

- **64-bit .NET 2.0 on Windows Server 2003 64-bit**. 64-bit .NET 2.0 was likewise able to utilize virtually all of the memory available on the machine for its cache, and hence maintained a dramatic performance advantage over 32-bit .NET 2.0 from 3000 possible random searches to 27,000 possible random queries. However, the tests at lower numbers of unique searches (where 32-bit .NET had enough memory to maintain all items in its cache) show that if the extended memory is not needed by this application, as with WebSphere, a performance penalty can be incurred when running on 64-bit platforms. Like 32-bit .NET 2.0, 64-bit .NET 2.0 maintained a much higher peak throughput than IBM WebSphere/Windows and IBM WebSphere/Linux.

Need to Cache More Data?

One very important point is that on 32-bit platforms, it simply is not possible to cache more data in a single CLR or Java process beyond about 2 gigabytes (actually less, leaving room for the application server/CLR logic itself) . However, with 64-bit platforms, while our machine was configured with 16 gigabytes of RAM, we could have easily simply added more RAM to the machine to increase its caching capabilities for the middle tier CachePerf application and thus maintain peak performance for even more possible unique queries and their associated larger cache requirements. This is a very big advantage the 64-bit platform offers vs. 32-bit platforms for both .NET and Java-based applications—the memory ceiling is, practically speaking, removed.

IBM WebSphere 6 Offload-To-Disk

As discussed, for the core benchmark runs we ran with IBM WebSpere 6 offload-to-disk feature disabled for the object cache, and we also did not use the Microsoft .NET Caching Application Block available on MSDN to test disk offload with .NET. While this feature can be useful in some memory constrained scenarios—essentially using disk as an extension of memory much like an OS swap file, it could also easily have a negative impact on performance for other applications. For example, in a typical

application there are many different transactions, pages, servlets and modules that do different types of work. Some of this may involve cache access, but for many interactions work is being done that does not utilize the cache at all. We wanted to investigate how the offload-to-disk feature of IBM WebSphere 6 might impact overall application performance in such a scenario. To do so, we extended the main CachePerf benchmark with a second servlet that does other work besides cache access (in this case it creates new model objects and displays dynamically generated data from them on a JSP page). We then ran a concurrent mix of benchmark agents with 60% of the agents accessing this new page ("other work") that involves no cache interactions, and 40% of the agents accessing the core MainServlet that accesses the object cache. We also implemented this same workload extension to the .NET CachePerf application. Keep in mind that the .NET application did not employ disk offload. With this modification in the benchmark workload, we then conducted three throughput tests and measured the overall TPS rates for the application (total tps, other work tps, and cache retrieval tps).

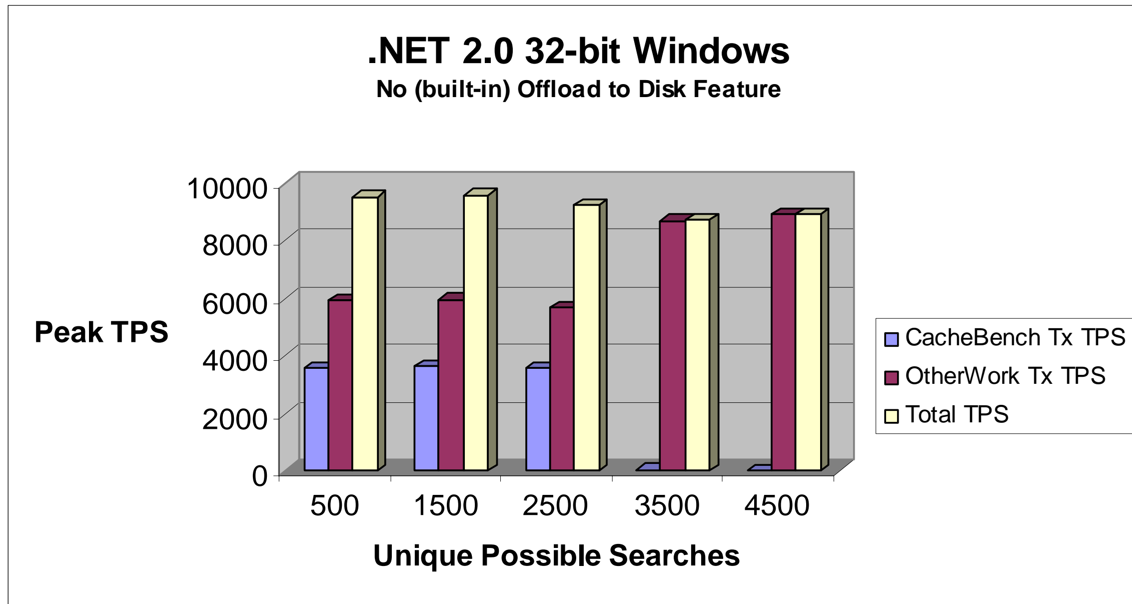
1. IBM WebSphere 6 32-bit with DiskOffload disabled, run from 1000 to 5000 possible unique searches in steps of 1000. We found it necessary in this case, to avoid out of memory errors, to cap cache entries at 2,000 (vs. 2,500 for the previous benchmark results) to provide more heap space for the other work going on (which involves the creation of many new object instances, although they are short-lived).

2. .NET 2.0 32-bit run (no disk offload) from 1000 to 5000 possible unique searches in steps of 1000.

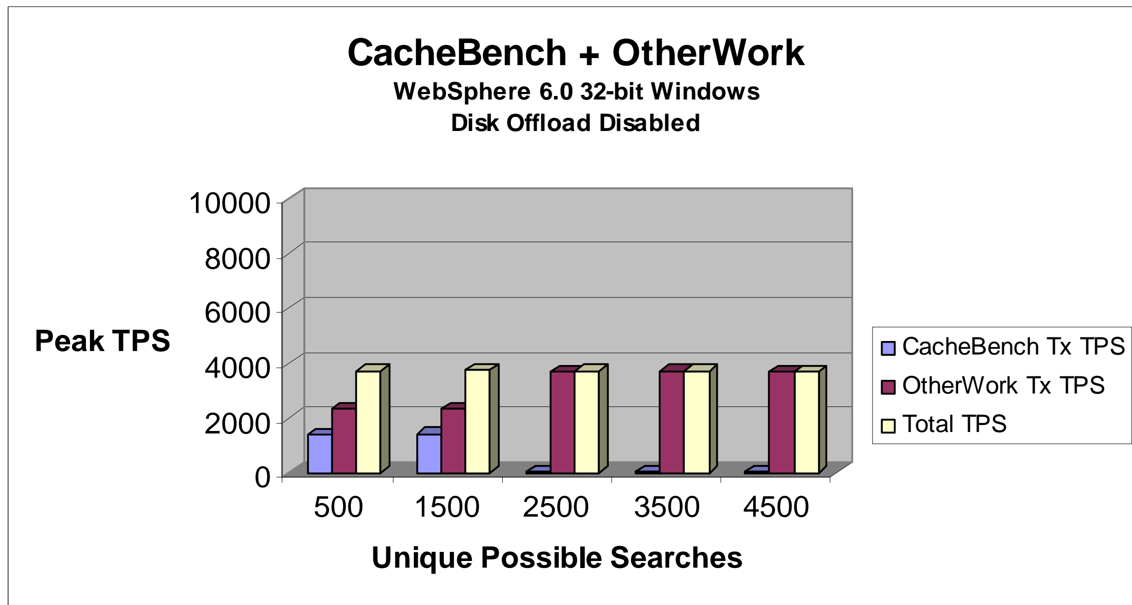
3. IBM WebSphere 6 32-bit with DiskOffload **enabled**, run from 1000 to 5000 possible unique searches in steps of 1000.

Benchmark Results

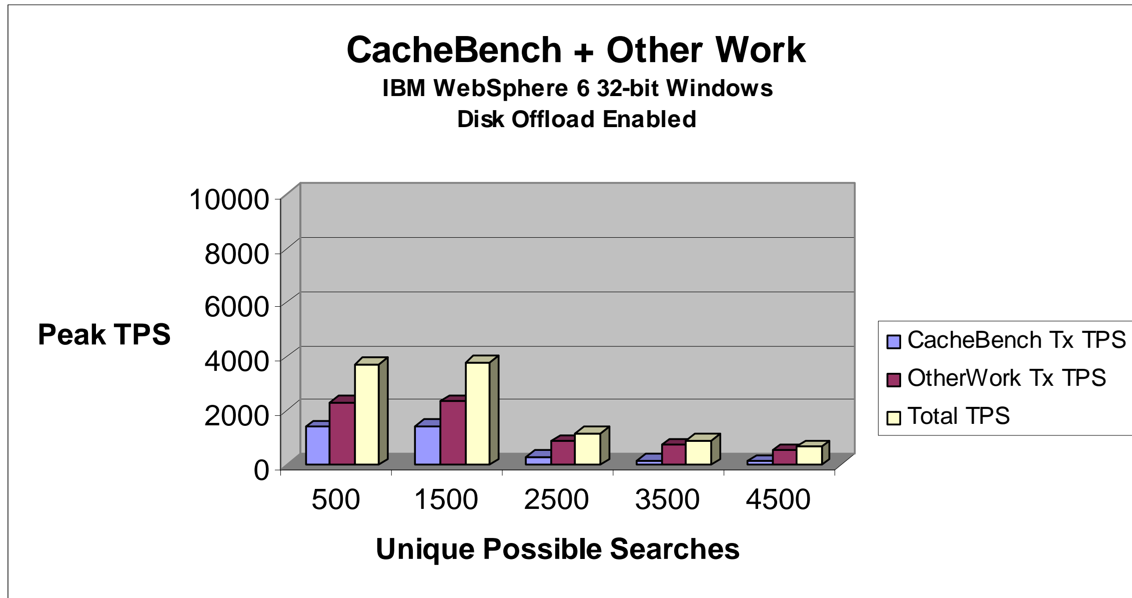
32-bit .NET (no offload-to-disk)



32-bit IBM WebSphere 6 with DiskOffLoad Disabled



32-bit IBM WebSphere 6 with DiskOffLoad Enabled



Tabular Results

.NET 2.0 32-bit Windows Server 2003 (no disk-based cache offload)

	CacheBench Tx TPS	OtherWork Tx TPS	Total TPS
500	3578	5923	9501
1500	3631	5959	9590
2500	3560	5661	9221
3500	44	8676	8720
4500	12	8906	8918

IBM WebSphere 6 32-bit Windows Server 2003 Disk Offload Disabled

	CacheBench Tx TPS	OtherWork Tx TPS	Total TPS
500	1376	2328	3704
1500	1423	2330	3753
2500	38	3690	3728
3500	18	3705	3723
4500	12	3674	3686

IBM WebSphere 6 32-bit Windows Server 2003 Disk Offload Enabled

	CacheBench Tx TPS	OtherWork Tx TPS	Total TPS
500	1407	2305	3712
1500	1435	2356	3791
2500	317	880	1197
3500	184	734	918
4500	127	557	684

Discussion

The offload-to-disk feature of IBM WebSphere 6 (and the .NET Cache Application Block available on MSDN for .NET) can be a valuable feature in some memory-constrained scenarios. However, as this data shows, the feature should be used with care because of the extra strain based on intense file I/O and object serialization/deserialization CPU costs that it can put on the application server, which could end up hurting, not helping, overall application server performance. In this case, as soon as the disk-off-load file begins to be used heavily (at 2500 users), so much of the CPU resources of the WebSphere application server are going towards accessing the offloaded cache on disk that the "other work" transaction throughput falls off dramatically. Essentially, the application gains a little bit of throughput for the cache access transaction by using disk-offload, but by sacrificing most of the throughput of the

other work going on in the application. When stressing the disk offload feature in WebSphere, overall peak throughput falls from over 3600 tps to about 600 tps.

The benefits of using a disk-offload feature really depends on the transaction mix and importance given to various transactions by the developer, which obviously will be application-specific. The feature may still be a positive in many scenarios since it can reduce the strain put on the database---freeing the database up to service other applications or transaction types. According to the test results here, however, a better option than using a disk-backed extended cache is to use 64-bit memory addressing, now available to both .NET and Java applications. This has the potential to eliminate the need to ever offload in-memory caches to disk in the first place, and thus avoid performance anomalies associated with resource contention in a mixed transaction workload.

Conclusion

The results show that for line-of-business applications that can take advantage of smart middle tier caching techniques, the potential performance benefits of 64-bit platforms can be substantial. The ability to directly utilize more memory on 64-bit platforms gives customers valuable new options to employ object-level caching more extensively when possible. The results also demonstrate, however, that middle tier applications that do not require extended memory addressing may actually perform slower on 64-bit platforms than 32-bit platforms.

Different types of applications can take advantage of caching to varying degrees and in many different ways. In the end, determining the right type of cache policies based on individual application requirements and potential cache benefits is one of the most important architectural decisions that can be made for an application that requires high levels of performance. Both IBM WebSphere and Microsoft .NET offer flexible middle-tier caching options, with features such as cache dependencies, cache invalidation callbacks and notifications. While this paper does not attempt to address the determination of optimal cache policies and feature tradeoffs, a variety of resources are available on MSDN on this topic.

Appendix 1: The Benchmark Source Code

The Java Code

Test.jsp

(Web page called to format display from MainServlet)

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">

<HTML>
<HEAD>
<%@ page language="java" session="false" contentType="text/html; charset=ISO-8859-1"
       pageEncoding="ISO-8859-1" import="java.util.Random"%>
<META http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<META name="GENERATOR" content="IBM Software Development Platform">
<META http-equiv="Content-Style-Type" content="text/css">

<TITLE>Test.jsp</TITLE>
</HEAD>
<BODY>

<%
ImageBench.Item item = (ImageBench.Item) request.getAttribute("Item");
String ImageID = Integer.toString(item.getImageID());
String ImageName = item.getImageName();
String ImageContentType = item.getImageContentType();
String displayImage = (String) request.getAttribute("DisplayImage");
Random generator = new Random();
int index =  generator.nextInt(1000);
%>
The image byte at index <%=index%> is <%=item.getImage()[index]%><br>
<span style="font-size: 24pt"><span style="color: #006600">Image Name: <%=ImageName
%><br />
Image Content Type:  <%=ImageContentType %><br>

</span></span>
<hr />
<br />
<% if (displayImage==null) {

%>
<img src='/ImageBench/servlet/ImageServlet?action=getimage&imageID=<%=ImageID %>'>
<%} %>
</BODY>
</HTML>
```

MainServlet.Java

(the main servlet that requests a record from the database by calling the DAL.Java function getItemImageBytes)

```
import javax.servlet.ServletConfig;
import javax.servlet.ServletContext;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.naming.InitialContext;
import com.ibm.websphere.cache.*;

import ImageBench.Item;


public class MainServlet extends HttpServlet   {
```

```
        /* (non-Java-doc)
         * @see javax.servlet.http.HttpServlet#HttpServlet()
         */

        DistributedObjectCache dm1 = null;


        /**
         * Process incoming HTTP GET requests
         *
         * @param request Object that encapsulates the request to the servlet
         * @param response Object that encapsulates the response from the servlet
         */
        public void doGet(javax.servlet.http.HttpServletRequest request,
                        javax.servlet.http.HttpServletResponse response)
                        throws ServletException {
                performTask(request, response);
        }

        /**
         * Process incoming HTTP POST requests
         *
         * @param request Object that encapsulates the request to the servlet
         * @param response Object that encapsulates the response from the servlet
         */
        public void doPost(javax.servlet.http.HttpServletRequest request,
                        javax.servlet.http.HttpServletResponse response)
                        throws ServletException {
                performTask(request, response);
        }


        /* (non-Java-doc)
         * @see javax.servlet.Servlet#init(ServletConfig arg0)
         */
        public void init(ServletConfig config) throws ServletException {
                super.init(config);
                InitialContext ic = Util.getInitialContext();
                try{

                 dm1 = (DistributedObjectCache) ic.lookup("services/cache/instance_one");

                }
                catch (Exception e)
                { e.printStackTrace();
                  throw new ServletException(e.getMessage());
                }


        }
        public void performTask(HttpServletRequest req, HttpServletResponse resp)
        throws ServletException {
                Item item = null;
                try{
                String imageID = (String) req.getParameter("imageID");
                String displayImage = (String) req.getParameter("displayImage");
                req.setAttribute("DisplayImage", displayImage);
                switch (Util.USE_CACHE)
                {
                        case 1:{
                                item = (Item) dm1.get(imageID);
                                if (item==null)
                                {
                                int id = Integer.parseInt(imageID);
                                DAL dal = new DAL();
                                item = dal.getItemImageItem(id);
                               dm1.put(imageID,item,1,Util.TIMEOUT,
                               EntryInfo.NOT_SHARED,null);
                                }
                                break;
                                }
```

```
                        case 0: {
                                int id = Integer.parseInt(imageID);
                                DAL dal = new DAL();
                                item = dal.getItemImageItem(id);
                                break;
                                }
                }
                req.setAttribute("Item", item);
                requestDispatch(getServletConfig().getServletContext(), req, resp,
                        "Test.jsp");

                }
                catch (Exception e)
                {
                        e.printStackTrace();
                        throw new ServletException(e);
                }
                }
        /**
        * Request dispatch
        */
        private void requestDispatch(ServletContext ctx, HttpServletRequest req,
HttpServletResponse resp, String page) throws Exception
        {
         resp.setContentType("text/html");
         ctx.getRequestDispatcher(page).forward(req, resp);
        }

        }
```

DAL.Java

(the database access class that uses JDBC to retrieve a record from the database)

```
/*
 * Created on Sep 27, 2005
 *
 * TODO To change the template for this generated file go to
 * Window - Preferences - Java - Code Style - Code Templates
 */

/**
 * @author
 *
 * TODO To change the template for this generated type comment go to
 * Window - Preferences - Java - Code Style - Code Templates
 */

import java.sql.*;


import ImageBench.Item;
public class DAL {


         private javax.sql.DataSource ds = (javax.sql.DataSource) Util
                .getDataSource(Util.DS_NAME);

        public Item getItemImageItem(int id) throws Exception
        {
                {
                        Connection conn = null;
                        ResultSet rs = null;
                        try {


                                conn = ds.getConnection();
```

```
                              PreparedStatement selectByID =
conn.prepareStatement("Select img_name,img_data,img_contenttype from Image where imageid
= ?");
                              selectByID.setInt(1, id);
                              rs = selectByID.executeQuery();
                              rs.next();
                              Blob blob = rs.getBlob(2);
                              int len = (int) blob.length();
                              byte[] image = blob.getBytes(1, len);
                              Item item = new Item(id, rs.getString(1), image,
rs.getString(3));
                              rs.close();
                              selectByID.close();
                              conn.close();


                              return item;
                      } catch (Exception es) {
                              rs.close();
                              conn.close();
                              es.printStackTrace();
                              throw new Exception( es);
                      }
              }


       }
}
```

Util.Java

```
/*
 * Created on Oct 1, 2005
 *
 * TODO To change the template for this generated file go to
 * Window - Preferences - Java - Code Style - Code Templates
 */

/**
 * @author
 *
 * TODO To change the template for this generated type comment go to
 * Window - Preferences - Java - Code Style - Code Templates
 */


import java.util.Hashtable;
import javax.naming.InitialContext;
import javax.naming.NamingException;

public class Util {

       private static InitialContext initCtx = null;
    private static Hashtable DsTable = new Hashtable();
       public static final String DS_NAME = "jdbc/ds1";

        static public javax.sql.DataSource getDataSource(String jndiName) {
       javax.sql.DataSource ds = (javax.sql.DataSource) DsTable.get(jndiName);
        if (ds == null) {
            try {
                InitialContext ic = getInitialContext();
                if (ic != null) {
                    ds = (javax.sql.DataSource) ic.lookup(jndiName);
                    if (ds != null) {
                        DsTable.put(jndiName, ds);
                    }
                }
            }
            // Naming Exception will cause a null return.
```

```
            catch (NamingException e) {
                System.out.println("Util.getDataSource(): Exception: " + e);
            }
        }
        return ds;
    }

    /**
     * Return the cached Initial Context.
     *
     * @return InitialContext, or null if a naming exception.
     */
    static public InitialContext getInitialContext() {
        try {
            // Get InitialContext if it has not been gotten yet.
            if (initCtx == null) {
                // properties are in the system properties
                initCtx = new InitialContext();
            }
        }
        // Naming Exception will cause a null return.
        catch (NamingException e) {}
        return initCtx;
    }

}
```

ImageServlet.Java (not called in benchmark, only for browser display)

```
//
//"This sample program is provided AS IS and may be used, executed, copied and modified
without royalty payment by customer (a) for its own
//instruction and study, (b) in order to develop applications designed to run with an IBM
WebSphere product, either for customer's own internal use
//or for redistribution by customer, as part of such an application, in customer's own
products. "
//
//Product 5630-A36,  (C) COPYRIGHT International Business Machines Corp., 2001,2002
//All Rights Reserved * Licensed Materials - Property of IBM
//


import java.io.*;

import javax.naming.InitialContext;
import javax.servlet.*;
import javax.servlet.http.*;

import com.ibm.websphere.cache.DistributedObjectCache;
import com.ibm.websphere.cache.EntryInfo;

import ImageBench.Item;


/**
* Servlet to handle image actions.
*/
public class ImageServlet extends HttpServlet
{

       DistributedObjectCache dm1 = null;
 /**
  * Servlet initialization.
  */
 public void init(ServletConfig config) throws ServletException
 {
    super.init(config);
    InitialContext ic = Util.getInitialContext();
       try{
```

```
         dm1 = (DistributedObjectCache) ic.lookup("services/cache/instance_one");

        }
        catch (Exception e)
        { e.printStackTrace();
          throw new ServletException(e.getMessage());
        }



 }

 /**
  * Process incoming HTTP GET requests
  *
  * @param request Object that encapsulates the request to the servlet
  * @param response Object that encapsulates the response from the servlet
  */
 public void doGet(javax.servlet.http.HttpServletRequest request,
javax.servlet.http.HttpServletResponse response)
         throws ServletException, IOException
 {
    performTask(request,response);
 }

 /**
  * Process incoming HTTP POST requests
  *
  * @param request Object that encapsulates the request to the servlet
  * @param response Object that encapsulates the response from the servlet
  */
 public void doPost(javax.servlet.http.HttpServletRequest request,
javax.servlet.http.HttpServletResponse response)
         throws ServletException, IOException
 {
    performTask(request,response);
 }

/**
  * Main service method for ImageServlet
  *
  * @param request Object that encapsulates the request to the servlet
  * @param response Object that encapsulates the response from the servlet
  */
 private void performTask(HttpServletRequest req, HttpServletResponse resp)
         throws ServletException, IOException
 {
    String action = null;
    action = req.getParameter("action");
    String imageID = (String) req.getParameter("imageID");
    Item item= null;
    if (action.equals("getimage"))
    {
        try{
        switch (Util.USE_CACHE)
                {
                        case 1:{

                                item = (Item) dm1.get(imageID);
                                if (item==null)
                                {
                                        int id = Integer.parseInt(imageID);
                                        DAL dal = new DAL();
                                        item = dal.getItemImageItem(id);
                                        dm1.put(imageID,item,1,Util.TIMEOUT,
                                        EntryInfo.NOT_SHARED,null);
                                }
                                break;
                                }
                        case 0: {
                                int id = Integer.parseInt(imageID);
```

```
                                DAL dal = new DAL();
                                item = dal.getItemImageItem(id);
                                break;
                                }
                }
                byte[] buf = item.getImage();
                 if (buf != null)
                 {
                   resp.setContentType("image/jpeg");
                   resp.getOutputStream().write(buf);
                 }

     }
        catch (Exception e) {
        throw new ServletException(e);
        }
     }
 }

 /**
  * send redirect
  */
 private void sendRedirect(HttpServletResponse resp, String page)
         throws ServletException, IOException
 {
    resp.sendRedirect(resp.encodeRedirectURL(page));
 }

 /**
  * Request dispatch.
  */
 private void requestDispatch(
         ServletContext ctx,
         HttpServletRequest req,
         HttpServletResponse resp,
         String page)
         throws ServletException, IOException {
         ctx.getRequestDispatcher(page).include(req, resp);
 }
}
```

Item.Java (model class to hold and cache a database record)

```
package ImageBench;

import java.io.Serializable;

/*
 * Created on Oct 1, 2005
 *
 * TODO To change the template for this generated file go to
 * Window - Preferences - Java - Code Style - Code Templates
 */

/**
 * @author Administrator
 *
 * TODO To change the template for this generated type comment go to
 * Window - Preferences - Java - Code Style - Code Templates
 */
public class Item implements Serializable{

        //
        // TODO: Add constructor logic here
        //

        // Internal member variables
        private int _imageId;
        private String _imageName;
        private byte[] _imageData;
```

```
        private String _imageContentType;




        /// <summary>
        /// Default constructor
        /// </summary>
        public Item()
        {
        }

        /// <summary>
        /// Constructor with specified initial values
        /// </summary>

        public Item(int imageID, String imageName, byte[] Image, String ImageContentType
)
        {
            this._imageId = imageID;
            this._imageName = imageName;
            this._imageContentType = ImageContentType;
            this._imageData = Image;


        }


        // Properties
        //image ID - Unique Identifier
        public int getImageID()
        {
             return _imageId;
        }
        //filename of image
        public String getImageName()
        {
            return _imageName;
        }
        //content mime type (used for generating the image)
        public String getImageContentType()
        {
             return _imageContentType;
        }
        public void setImageId(int imagecounterID){

            _imageId = imagecounterID;
            return;
        }

        public byte[] getImage()
        {
            return this._imageData;
        }

   }
```

OtherWorkServlet.java

```
import java.math.BigDecimal;
import javax.servlet.ServletConfig;
import javax.servlet.ServletContext;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.util.*;
```

```
import ImageBench.Item;

public class OtherWorkServlet extends HttpServlet   {
	/* (non-Java-doc)
	 * @see javax.servlet.http.HttpServlet#HttpServlet()
	 */
	public OtherWorkServlet() {
		super();
	}

	/**
	 * Process incoming HTTP GET requests
	 *
	 * @param request Object that encapsulates the request to the servlet
	 * @param response Object that encapsulates the response from the servlet
	 */
	public void doGet(javax.servlet.http.HttpServletRequest request,
			javax.servlet.http.HttpServletResponse response)
			throws ServletException {
		performTask(request, response);
	}

	/**
	 * Process incoming HTTP POST requests
	 *
	 * @param request Object that encapsulates the request to the servlet
	 * @param response Object that encapsulates the response from the servlet
	 */
	public void doPost(javax.servlet.http.HttpServletRequest request,
			javax.servlet.http.HttpServletResponse response)
			throws ServletException {
		performTask(request, response);
	}


	/* (non-Java-doc)
	 * @see javax.servlet.Servlet#init(ServletConfig arg0)
	 */
	public void init(ServletConfig config) throws ServletException {
		super.init(config);
	}
	public void performTask(HttpServletRequest req, HttpServletResponse resp)
	throws ServletException {
		try
		{
		Random generator = new Random();
		int index =  generator.nextInt(1000);
		String imagename = "Image" + index;
		String imagecontenttype = "p/jpeg";
		Item item = new Item(index, imagename, null, imagecontenttype);
		req.setAttribute("Item", item);
	 requestDispatch(getServletConfig().getServletContext(), req, resp,
				"Test2.jsp");
		}
		catch (Exception e)
		{
			e.printStackTrace();
			throw new ServletException(e.getMessage());
		}



	}
	/**
	* Request dispatch
	*/
	private void requestDispatch(ServletContext ctx, HttpServletRequest req,
HttpServletResponse resp, String page) throws Exception
	{
	 resp.setContentType("text/html");
	 ctx.getRequestDispatcher(page).forward(req, resp);
```

```
        }

        }
```

Test2.jsp

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">

<HTML>
<HEAD>
<%@ page language="java" session="false" contentType="text/html; charset=ISO-8859-1"
       pageEncoding="ISO-8859-1"%>
<META http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<META name="GENERATOR" content="IBM Software Development Platform">
<META http-equiv="Content-Style-Type" content="text/css">
<LINK href="theme/Master.css" rel="stylesheet"
       type="text/css">
<TITLE>Test2.jsp</TITLE>
</HEAD>
<BODY>

<%
ImageBench.Item item = (ImageBench.Item) request.getAttribute("Item");
String ImageID = Integer.toString(item.getImageID());
String ImageName = item.getImageName();
String ImageContentType = item.getImageContentType();
%>
<span style="font-size: 24pt"><span style="color: #006600">Image ID: <%=ImageID
%><br/> Image Name: <%=ImageName %><br />
Image Content Type:  <%=ImageContentType %><br>

</span></span>

<br />
</BODY>
</HTML>
```

The .NET Code

Default.aspx

```
<%@ Page Language="C#" contentType="text/html; charset=ISO-8859-1" responseEncoding="ISO-
8859-1" AutoEventWireup="true" EnableSessionState="false" EnableEventValidation="false"
CodeFile="Default.aspx.cs" Inherits="_Default" %>
<html>
<head runat="server">
<META http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<META name="GENERATOR" content="MS Development Platform">
<META http-equiv="Content-Style-Type" content="text/css">
<LINK href="theme/Master.css" rel="stylesheet" type="text/css">
<title>Default.aspx</title>
</head>
<body>
    <%Random myRandom = new Random(DateTime.Now.Millisecond);
     int index = (int) (myRandom.NextDouble()*1000 + 1);
       %>
    The image byte at index <%=index%> is <%=testimageinfo.getImage()[index]%><br>
    <span style="font-size: 24pt"><span style="color: #006600">Image Name:</span> </span>
     <asp:Label ID="Name" runat="server" Font-Size="XX-Large"></asp:Label><br />
    <span style="font-size: 24pt"><span style="color: #006600">Image Content Type:</span>
        <asp:Label ID="Type" runat="server"></asp:Label><br />
    </span>
    <hr />
    <br />
     
    <%if (Request["displayImage"] != "false") {%>
    <asp:Image ID="DisplayImage" runat="server" />
    <%}%>
</body>
</html>
```

Default.aspx.cs (code behind)

```
using System;
using System.Data;
using System.Configuration;
using System.Collections;
using System.Web;
using System.Web.Security;
using System.Web.UI;
using System.Web.UI.WebControls;
using System.Web.UI.WebControls.WebParts;
using System.Web.UI.HtmlControls;
using System.Web.Caching;
using DALLayer;
using Utility;


public partial class _Default : System.Web.UI.Page
{
    public TestImageInfo testimageinfo =null;

    protected void Page_Load(object sender, EventArgs e)
    {
        int selectid;
        if (Request["selectid"] != null)
                selectid = Convert.ToInt32(Request["selectid"]);
            else
            selectid = 1;
        try {
        switch(Util.CACHE){
            case 1: {
                        Object cacheddata = Cache.Get(selectid.ToString());
                        if (cacheddata != null)
                        {
                            testimageinfo = (TestImageInfo)cacheddata;
```

```
                        }
                        else
                        {
                            DAL myDAL = new DAL();
                               testimageinfo = myDAL.GetTestImage(selectid);
                               Cache.Insert(selectid.ToString(), testimageinfo, null,
DateTime.Now.AddSeconds(Util.TIMEOUT), System.Web.Caching.Cache.NoSlidingExpiration ,
CacheItemPriority.High, null);
                        }
                            break;
            }
             case 0: {
                            DAL myDAL = new DAL();
                                   testimageinfo = myDAL.GetTestImage(selectid);
                             break;
                     }
        }
        }
               catch (Exception ex)
               {
                      throw new ApplicationException("Error: " + ex.Message);
               }
        Name.Text = testimageinfo.imageName;
        Type.Text = testimageinfo.imageContentType;
        DisplayImage.ImageUrl="displayimage.aspx?selectid=" + selectid;


    }
}
```

DAL.cs

```
using System;
using System.Data;
using System.Data.OracleClient;
using System.Configuration;
using System.Collections;
using System.IO;
using Utility;


namespace DALLayer
{
    /// <summary>
    /// The DAL.cs class contains all the ADO.NET logic to retrieve data from the Image
table.
    /// </summary>
    ///
    public class DAL
    {
        //This code initializes the Connection String and Parameters Used
        public DAL()
        { }

        private static string SQL_SELECTRECORD = "Select
img_name,img_data,img_contenttype from Image where imageid = :imageID";
        private static string PARM_IMAGEID = ":imageID";


        //This function retrieves one row of data from the Image table, and populates a
new model class (TestImageInfo) with this data
        public TestImageInfo GetTestImage(int selectID)
        {
            OracleParameter parm = new OracleParameter(PARM_IMAGEID, OracleType.Int32,
10);
            parm.Value = selectID;
            TestImageInfo testimageinfo = null;
```

```
            using (OracleDataReader rs = ExecuteReader(Util.CONN_STRING_WEB,
CommandType.Text, SQL_SELECTRECORD, parm))
            {
                rs.Read();
                testimageinfo = new TestImageInfo(selectID, rs.GetString(0),
(byte[])rs["img_data"], rs.GetString(2));
            }
            return testimageinfo;

        }

        /// <summary>
               ///  Helper function to execute a generic select query that will return a
result set
               /// </summary>
               /// <param name="connString">Connection string</param>
               //// <param name="commandType">the CommandType (stored procedure, text,
etc.)</param>
               /// <param name="commandText">the stored procedure name or PL/SQL
command</param>
               /// <param name="commandParameters">an array of OracleParamters used to
execute the command</param>
               /// <returns></returns>
               public static OracleDataReader ExecuteReader(string connString,
CommandType cmdType, string cmdText, params OracleParameter[] cmdParms) {

                       //Create the command and connection
                       OracleCommand cmd = new OracleCommand();
                       OracleConnection conn = new OracleConnection(connString);


                       try {
                               //Prepare the command to execute
                               PrepareCommand(cmd, conn, null, cmdType, cmdText,
cmdParms);

                               //Execute the query, stating that the connection should
close when the resulting datareader has been read
                               OracleDataReader rdr =
cmd.ExecuteReader(CommandBehavior.CloseConnection);

                               cmd.Parameters.Clear();
                               return rdr;

                       }catch (Exception e) {

                               //If an error occurs close the connection as the reader
will not be used and we expect it to close the connection
                               conn.Close();
                               conn.Dispose();
                               throw e;
                       }
               }

               /// <summary>
               /// Helper function to prepare a generic command for execution by the
database
               /// </summary>
               /// <param name="cmd">Existing command object</param>
               /// <param name="conn">Database connection object</param>
               /// <param name="trans">Optional transaction object</param>
               /// <param name="cmdType">Command type, e.g. stored procedure</param>
               /// <param name="cmdText">Command test</param>
               /// <param name="cmdParms">Parameters for the command</param>
               private static void PrepareCommand(OracleCommand cmd, OracleConnection
conn, OracleTransaction trans, CommandType cmdType, string cmdText, OracleParameter[]
cmdParms) {

                       //Open the connection if required
                       if (conn.State != ConnectionState.Open)
                               conn.Open();
```

```
                        //Set up the command
                        cmd.Connection = conn;
                        cmd.CommandText = cmdText;
                        cmd.CommandType = cmdType;

                        //Bind it to the transaction if it exists
                        if (trans != null)
                                cmd.Transaction = trans;

                        // Bind the parameters passed in
                        if (cmdParms != null) {
                                foreach (OracleParameter parm in cmdParms)
                                        cmd.Parameters.Add(parm);

                        }

                }

    }

}
```

TestImageInfo.cs (model class to hold and cache a database record)

```
using System;
using System.Data;
using System.Configuration;
using System.IO;

namespace DAL
{

    /// <summary>
    /// Summary description for TestImageInfo
    /// </summary>
    [Serializable]
    public class TestImageInfo
    {
        //
        // TODO: Add constructor logic here
        //

        // Internal member variables
        private int _imageId;
        private string _imageName;
        private byte[] _imageData;
        private string _imageContentType;




        /// <summary>
        /// Default constructor
        /// </summary>
        public TestImageInfo()
        {
        }

        /// <summary>
        /// Constructor with specified initial values
        /// </summary>

        public TestImageInfo(int imageID, string imageName, byte[] Image, string
ImageContentType )
        {
            this._imageId = imageID;
            this._imageName = imageName;
            this._imageContentType = ImageContentType;
```

```
            this._imageData = Image;


        }


        // Properties
        //image ID - Unique Identifier
        public int imageID
        {
            get { return _imageId; }
        }
        //filename of image
        public string imageName
        {
            get { return _imageName; }
        }
        //content mime type (used for generating the image)
        public string imageContentType
        {
            get { return _imageContentType; }
        }
        public void setImageId(int imagecounterID){

            _imageId = imagecounterID;
            return;
        }

        public byte[] getImage()
        {
            return this._imageData;
        }

    }



}
```

DisplayImage.aspx (not called in benchmark, only for browser display)

```
<%@ Page Language="C#" %>
<%@ Import Namespace="System.Data.SqlClient" %>
<%@ Import Namespace="DALLayer" %>
<%@ Import Namespace="Utility" %>
<%
   // Clear out the existing HTTP header information
   Response.Expires = 0;
   Response.Buffer = true;
   Response.Clear();

   int selectid = Convert.ToInt32(Request["selectid"]);
   TestImageInfo testimageinfo = null;

   try
   {
       switch (Util.CACHE)
       {
           case 1:
               {
                   Object cacheddata = Cache.Get(selectid.ToString());
                   if (cacheddata != null)
                   {
                       testimageinfo = (TestImageInfo)cacheddata;
                   }
                   else
                   {
                       DAL myDAL = new DAL();
                       testimageinfo = myDAL.GetTestImage(selectid);
```

```
                        Cache.Insert(selectid.ToString(), testimageinfo, null,
DateTime.Now.AddSeconds(Util.TIMEOUT), System.Web.Caching.Cache.NoSlidingExpiration,
CacheItemPriority.High, null);
                    }
                    break;
                }
            case 0:
                {
                    DAL myDAL = new DAL();
                    testimageinfo = myDAL.GetTestImage(selectid);
                    break;
                }
        }
   }
   catch (Exception ex)
   {
       throw new ApplicationException("Error: " + ex.Message);
   }

   // Change the HTTP header to reflect that an image is being passed.

   Response.ContentType = testimageinfo.imageContentType;
   Response.BinaryWrite(testimageinfo.getImage());
   Response.End();

%>
```

```
Util.cs
using System;
using System.Collections.Generic;
using System.Text;
using System.Configuration;

namespace Utility
{
    public class Util
    {
        public static readonly int CACHE =
Convert.ToInt32(ConfigurationManager.AppSettings["Cache"]);
        public static readonly int TIMEOUT =
Convert.ToInt32(ConfigurationManager.AppSettings["CacheTimeout"]);
        public static string CONN_STRING_WEB = ConfigurationManager.AppSettings["DSN"];
    }
}
```

Test2.aspx (Other Work Page)

```
<%@ Page Language="C#" contentType="text/html; charset=ISO-8859-1" responseEncoding="ISO-
8859-1" AutoEventWireup="true" EnableSessionState="false" EnableEventValidation="false"
CodeFile="Test2.aspx.cs" Inherits="Test2" %>
<html>
<head id="Head1" runat="server">
<META http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<META name="GENERATOR" content="MS Development Platform">
<META http-equiv="Content-Style-Type" content="text/css">
<LINK href="theme/Master.css" rel="stylesheet" type="text/css">
<title>Test2.aspx</title>
</head>
<body>
    <span style="font-size: 24pt"><span style="color: #006600">Image ID:</span> </span>
     <asp:Label ID="ID" runat="server" Font-Size="XX-Large"></asp:Label><br />
    <span style="font-size: 24pt"><span style="color: #006600">Image Name:</span> </span>
     <asp:Label ID="Name" runat="server" Font-Size="XX-Large"></asp:Label><br />
    <span style="font-size: 24pt"><span style="color: #006600">Image Content Type:</span>
        <asp:Label ID="Type" runat="server"></asp:Label><br />
    </span>
    <hr />
    <br />
```

```
     

</body>
</html>
```

Test2.aspx.cs

```
using System;
using System.Data;
using System.Configuration;
using System.Collections;
using System.Web;
using System.Web.Security;
using System.Web.UI;
using System.Web.UI.WebControls;
using System.Web.UI.WebControls.WebParts;
using System.Web.UI.HtmlControls;
using DAL;

public partial class Test2 : System.Web.UI.Page
{
    protected void Page_Load(object sender, EventArgs e)
    {
        try{
        Random myRandom = new Random(DateTime.Now.Millisecond);
        int index = (int)(myRandom.NextDouble() * 1000 + 1);
        String imageName = "Image" + index;
        String imageContentType = "p/jpeg";
        int imageid = index;
        TestImageInfo testimageinfo = new TestImageInfo(index, imageName, null,
imageContentType);
        ID.Text = index.ToString();
        Name.Text = testimageinfo.imageName;
        Type.Text = testimageinfo.imageContentType;
        }
         catch (Exception e){
         throw e;
        }
    }
}
```

Appendix 3: The Tuning Parameters for the Benchmark

WebSphere 6 Tuning

WAS 6.0.2.3

```
Web Session State (cookies)Off
Access Log Turned Off
Performance Monitor Infrastructure Turned Off
Diagnostic Trace Turned Off
System Out Off
HTTP Channel maximum persistent requests = -1
HTTP Channel readTimeout = 6000
HTTP Channel writeTimeout = 6000
HTTP Channel persistentTimeout = 3000
Minimum Web Container threads = 75
Maximum Web Container threads = 75
Minimum ORB threads = 30
Maximum ORB threads = 30
Minimum Default threads = 20
Maximum Default threads = 20
Heap Sizes: see earlier discussion in the paper
```

IBM HTTP Server

```
IBM HTTP Server Windows:
        Access Log Off
        Max KeepAlive Requests 3000
        2048 Max threads
        2048 Threads/child
IBM HTTP Server Linux:
        Access Log Off
        Max KeepAlive Requests 3000
        ThreadLimit 50
        ServerLimit 10
        StartServers 10
        MaxClients 500
        MinSpareThreads 100
        MaxSpareThreads 100
        Threads/Child 50
        MaxRequests/Child 0
```

Linux Tuning

```
net.ipv4.tcp_max_syn_backlog=1024
kernel.msgmni=1024
kernel.sem=1000 32000 32 512
fs.file-max=65535
kernel.shmmax =4294967295
net.core.netdev_max_backlog = 20000
net.core.somaxconn = 20000
net.ipv4.tcp_fin_timeout = 30
net.ipv4.tcp_syn_retries = 20
net.ipv4.tcp_synack_retries = 20
net.ipv4.tcp_sack = 0
net.ipv4.tcp_timestamps = 0
```

```
net.ipv4.conf.all.arp_ignore = 3
net.ipv4.conf.all.arp_announce = 2
```

Open File Handle limit (soft) increased to 20000

Windows Server 2003 Tuning

Added registry parameter settings for tcpip service (CurrentControlSet/Services/tcpip/parameters):

```
        TcpTimedWaitDelay set to decimal 30
        MaxUserPort set to decimal 32768
```

.NET 2.0 Tuning

.NET Worker Process

Rapid Fail Protection off
Pinging off
Recycle Worker Process off
<cache privateBytesLimit> (in Web.config): see earlier discussion in the paper

ASP.NET

Session state off
Authentication set to “None” to match anonymous access of IBM WebSphere

IIS 6.0 Virtual Directory

Authentication Anonymous
Access Logging Off

Mercury Settings

Agents were set to:

1. Simulate a new user on each request (each requests opens, then closes its own connection to the app server, although http keepalives are enabled in all cases).

2. Download only HTML resources returned by the application server, and not graphics files or any other non-HTML resources.