Windows Server AppFabric Cache: A detailed performance & scalability datasheet

A Technical White Paper by Grid Dynamics

January, 2011

Grid Dynamics

Max Martynov, Roman Milyuchikhin, Sergey Torgashov

39300 Civic Center Drive, Suite 145

Fremont, CA 94538

Tel: +1 510-574-0872

Fax: +1 636-773-4528

http://www.griddynamics.com/

Table of Contents

Introduction

Windows Server AppFabric Cache provides a highly scalable in-memory distributed cache for application data. By using AppFabric, you can significantly improve application performance by avoiding repetitive calls to the data source. AppFabric Cache enables your application to scale with increasing demand by using a cache cluster that automatically handles the complexities of load balancing. Scalability can be achieved by adding more computers on demand. High availability allows AppFabric Cache to store two copies of the data across different nodes in the cluster. Depending on requirements, AppFabric Cache can be collocated with a distributed application or it can be set up to run on dedicated machines to avoid resource contention.

Previously, Grid Dynamics analyzed the performance of a Microsoft project code-named "Velocity" CTP3. Velocity is now called AppFabric Cache and the on-premise version released as part of the Windows Server AppFabric platform. The version of AppFabric Cache evaluated in this white paper, is the on-premise release from June 2010.

Caching is also available in the cloud, as part of Windows Azure AppFabric platform. Please see this link for more details.

The goal of this whitepaper is to provide performance and scalability data that can be used in capacity planning. We collected this data by varying usage patterns, workloads, and configuration of the cache.

We recommend going over this whitepaper during preparation of the architecture and capacity planning in the following manner:

1. Read the main part to gain an understanding of the performance patterns in Windows Server AppFabric Cache
2. Through tests, understand how the application is going to use Windows Server AppFabric Cache and how the cache is going to be configured
3. Estimate performance by applying the performance numbers from Appendix A, the considerations from the "Benchmarks" segment and the advice from the "Conclusion and recommendations" section, to the current architecture
4. If the projected performance is not satisfactory, return to step 2.

The information included in this whitepaper covers some common cases and will help attain a suitable capacity planning. However, you must validate the numbers based on performance tests using your hardware with the appropriate configurations and workloads. To support this, the test harness used for this benchmarking is available for download at this location. The test harness is configurable and it may help to reproduce performance tests described in the whitepaper, on the custom hardware, or serve as a base for more specific custom tests.

Test methodology

Like most systems, there are many parameters and test matrix combinations that can affect performance. This means that the function of dozens of arguments, where each argument varies within very broad boundaries, should be built. The number of configurations where latency, throughput, and other KPIs need to be measured can easily grow to millions and billions of points. Obviously, this is not feasible.

It is feasible, however, to find the most important configuration points to understand performance from these points. To accomplish this, a number of major factors that influence cache and client performance were found and their impact on performance was tested and understood.

The goal of the tests was to comprehend the performance of the cache itself, so almost all tests were done on the Windows Server AppFabric Cache cluster directly, without intermediate layers. There are two scenarios where intermediate layers were introduced. The first was aimed to investigate performance of Windows Server AppFabric Cache as an ASP.NET session state provider, which is a very common customer scenario with an out of the box session state provider. The second included a WCF proxy that simulated an application layer between the ultimate data consumer and the cache.

Test configuration

Typically, in each test, only one or two parameters were varied, in order to focus on their performance impact. The entire set of parameters includes:

1. **Load pattern**: cache usage pattern, i.e. percentage of 'Get', 'Put', 'BulkGet', 'GetAndLock', 'PutAndUnlock' operations
2. **Cached data size**: amount of data stored in cache during the test
3. **Cluster size**: number of servers in the cache cluster
4. **Object size**: size of objects after serialization that are stored in the cache
5. **Type complexity**: different .NET object types such as byte, string[], etc that are stored in the cache
6. **Security**: security settings of the cache cluster

To prevent unstructured and poorly understandable results, all tests were divided in a number of test cases where the parameters were varied based on known customer implementations:

Test case	Goal	Variable parameters
Cached data size	Vary the amount of data stored in the cache cluster and measure the performance across cache operations	Cached data size
Scalability	Measure the scalability aspects of Windows Server AppFabric Cache by varying the number of servers	Load pattern, cluster size
Object size	Vary size of objects that are read and written in the cache and measure performance	Object size, cache cluster size
Type complexity	Vary complexity of .NET type of objects stored in cache and measure performance	Object type, cache cluster size

Read performance	Measure the difference in performance between different "Get" methods in different situations	Get operation ("Get" with cache hit, "Get" with cache miss, "BulkGet"), cache cluster size
Locking performance	Measure performance between optimistic and pessimistic locking	Cache cluster size, locking algorithm (pessimistic vs optimistic)
Security	Vary the security settings and measure cache performance	Cache security type, cluster size, object size
ASP.NET session state performance	Measure performance of Windows Server AppFabric Cache ASP.NET session state provider	Object size
WCF proxy performance	Measure performance of Windows Server AppFabric Cache when the cache client is running as a separate tier as a WCF service	Cache cluster size

Measurement and result interpretation

During each test, a number of key metrics were gathered from cache and load generation servers using performance counters:

- **Throughput**: total number of requests per second
- **Latency**: request execution time in milliseconds
- **Windows Server AppFabric Cache performance counters**: cache hit count, cache miss count, read requests per second, write requests per second, etc.
- **.NET performance counters**: GC memory per generation, time spent in GC, etc.
- **Windows core performance counters**: CPU utilization, RAM utilization, network utilization, disk performance, etc.

Furthermore, throughput is an aggregated number of operations per second across all cache servers that all load agents and the entire system under test can perform. Latency is always an average number of milliseconds that a request can take. CPU, network and other metrics related to a server are always averaged: on a cache cluster CPU and network are averaged across the cache cluster; on load agents, CPU and network are averaged across the load generation servers.

Each test was implemented to start generating a small number of requests and then to gradually increase the load. Exact initial users number, step users number, max users number, step duration and other Visual Studio Load Test parameters varied from test to test to ensure efficient generation of load.

Gradual generation of load means that latency, in the beginning of each test, is low and it grows slowly until the cluster is saturated, at which point, it starts to quickly increase. Throughput, on the other side, is growing fast until the cluster is saturated. After the cluster is fully loaded, throughput can grow a little more, but latency will increase dramatically and become very high as well. Since different applications have different requirements for throughput and latency, this whitepaper shows the different benchmark mode combinations. The figure below shows different points at which performance data will be collected. The dependency represented on a graph below is measured for direct cache access on a 12 node cluster, with 16KB byte array objects, 90% reads/10% writes, default security with SecurityMode = Transport, and ProtectionLevel = EncryptAndSign.

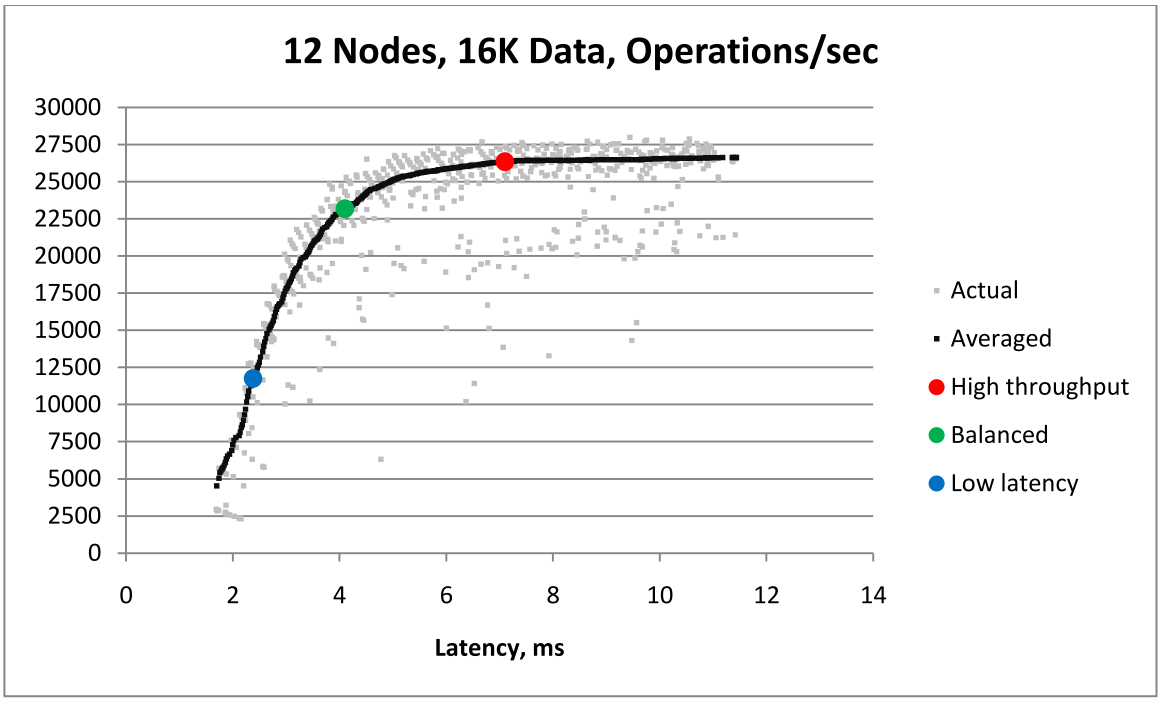

Figure 1: Dependency of throughput from latency for direct cache access (12 nodes cluster, 16KB byte array objects, 90% reads & 10% writes, default security (ON))

In the figure above, the following data collection points are shown:

- "**High throughput**" is measured when the cluster is fully loaded and when a further increase of throughput can be achieved only in exchange of dramatic growth of latency. In the figure above, "high throughput" is a red point with latency = 7.1 and throughput = 26,349.
- "**Balanced**" is measured when the cluster just starts being saturated and when further increase of throughput can be done in exchange of comparable growth of latency. In the figure above, "balanced" is a green point with latency = 4.1 and throughput = 23,187.
- "**Low latency**" is measured when the load is very low. In the figure above, "low latency" is a blue point with latency = 2.38 and throughput = 11,751.

We have applied this logic to classify the data collected in each test case. Depending upon your SLA requirements, you should use the appropriate data for your capacity planning. However, do consider that the “balance” point will give the best results in terms of performance versus resource consumption.

Almost all the graphics in the sections below use only "high throughput" and "balanced" points. "Low latency" points are included in the detailed performance numbers tables in Appendix A. Please, note that the logic described above, can have slightly different latency points for tests performed under similar conditions. This difference is almost negligible for "high throughput" points, but is noticeable for "low latency" points. This happens because in the case of "low latency" points, small changes can result in a significant difference of throughput.

Testing environment

The testing environment consists of a Windows Server AppFabric Cache cluster, a number of load generation agents and a number of support servers that are required by both cache cluster and load agents. All servers are in the same network with a single switch.

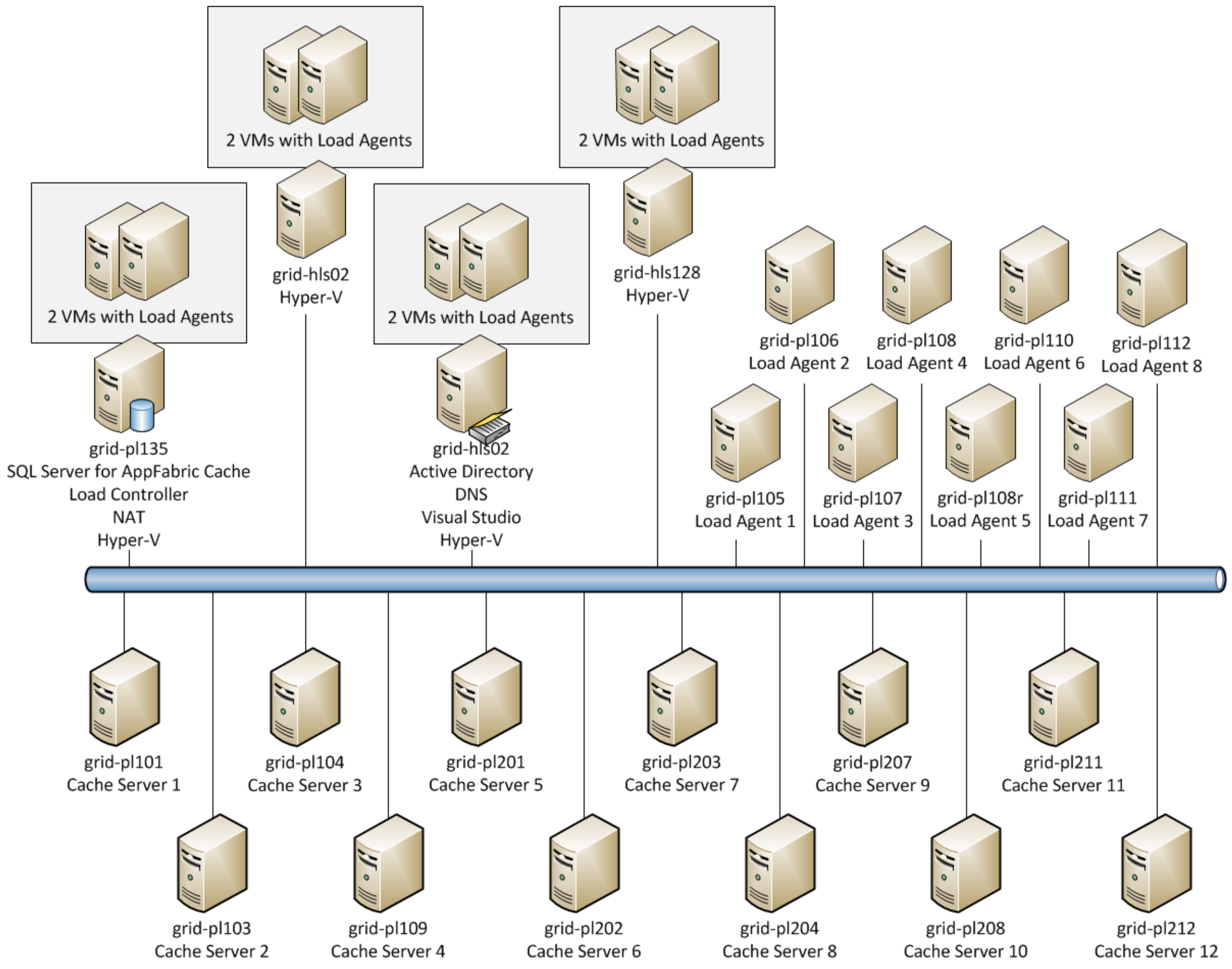

Figure 2: Testing environment hardware configuration

Detailed hardware specification of cache cluster and load generation agent servers is given below.

Windows Server AppFabric Cache cluster hardware configuration

The cache cluster consists of 12 physical servers:

# of servers	Platform	CPU	RAM	Disk	Network
4	DL380 G4	2 x Dual 3.0 GHz 64-bit Intel Xeon processors with 2MB L2 Cache, 800MHz front side bus and 64-bit extension	12 GB DDR2 ECC	3 x 146GB 10K RPM SCSI	1x1Gb NIC
8	Dell PowerEdge 2850	2 x Dual 3.0 GHz 64-bit Intel Xeon processors with E7520 chipset 2MB L2 Cache 800MHz front side bus	12 GB DDR2 ECC	3 x 72GB 10K RPM SCSI	1x1Gb NIC

The first group included 4 servers: grid-pl101, grid-pl103, grid-pl104, and grid-pl109. The second group consisted of 8 servers: grid-pl201, grid-pl202, grid-pl203, grid-pl204, grid-pl207, grid-pl208, grid-pl211, and grid-pl212. Although servers in the two groups were not completely identical, they had the same CPU, RAM, disk and network. During the tests, there was no evidence that servers in one group performed differently from those in another.

Windows Server AppFabric Cache cluster uses SQL Server installed on grid-pl135 to store configuration and manage cluster membership. All tests were performed with 'default' cache and 'default' region unless it is explicitly stated otherwise.

Load generation environment hardware configuration

The load generation environment consists of 12 physical servers:

# of servers	Platform	CPU	RAM	Disk	Network
2	HP ProLiant DL365 G5p	2 x 2.6 GHz Quad core AMD Opteron T Processor Model 2376 (2.3GHz, 75W ACP)	24 GB	(2) Hot-Swappable 2.5" SAS	2x1Gb NIC
8	DL380 G4	2 x Dual 3.0 GHz 64-bit Intel Xeon processors with 2MB Level 2 cache, 800 MHz frontside bus and 64-bit extension	12 GB DDR2 ECC	3 x 146GB 10K RPM SCSI	1x1Gb NIC
2	DL385 G5	2 x Quad 3.00 GHz 64-bit Intel Xeon Processor X5365, 6MB Level 2 cache, 1333 MHz FSB and 64-bit extension	32 GB	(4) Hot-Swappable 2.5" 146 GB SAS	2x1Gb NIC

The first group included 2 servers: grid-hls02 and grid-hls03. The second group consisted of 8 servers: grid-pl105, grid-pl106, grid-pl107, grid-pl108, grid-pl108r, grid-pl110, grid-pl111, and grid-pl112. The first group included 2 servers: grid-pl128 and grid-pl135.

Most powerful servers with 8 cores are used for specific purposes:

- grid-pls135 contains Load Controller and SQL Server for Windows Server AppFabric Cache
- grid-hls02 contains DNS and Active Directory for Windows Server AppFabric Cache

These specific roles require little use of CPU and RAM, so they do not influence Windows Server AppFabric Cache performance and do not interfere with Load Agents installed on VMs that use most of the CPU.

All servers are connected to the same Dell 5448 network switch that is used to connect cache servers.

Benchmark Test Results

Cache data size

The goal is to vary the amount of stored data in the cache cluster and measure the performance across cache operations. The data was analyzed and a summary of the impact is provided.

Set up

Tests in this category were performed with the following fixed parameters:

Parameter name	Value
Cluster size	12 nodes
Object size	16 KB
Object type	Byte array
Security	Default (SecurityMode=Transport, ProtectionLevel=EncryptAndSign)
Local cache	Not used

Configurations with the following variable parameters were tested:

Load pattern	Cached data size
90% reads & 10% writes	128MB, 3GB, 12GB, 24GB, 48GB
50% reads & 50% writes	128MB, 3GB, 12GB, 24GB, 48GB

Results

During the cache data size tests, latency did not depend on the cache data size and was approximately the following (in milliseconds):

Data collection point	90% reads & 10% writes, ms	50% reads & 50% writes, ms
High	6.7	7.2
Balanced	3.7	4
Low	2.1	2.6

The **throughput** varied within the following boundaries:

Data collection point	90% reads & 10% writes, ops/sec	50% reads & 50% writes, ops/sec
High	25,000	19,000-27,000
Balanced	20,000	14,000-25,000
Low	10,000	9,000-15,000

The tests showed that cache data size has no impact on cache operations throughput in the case of a high ratio of reads:

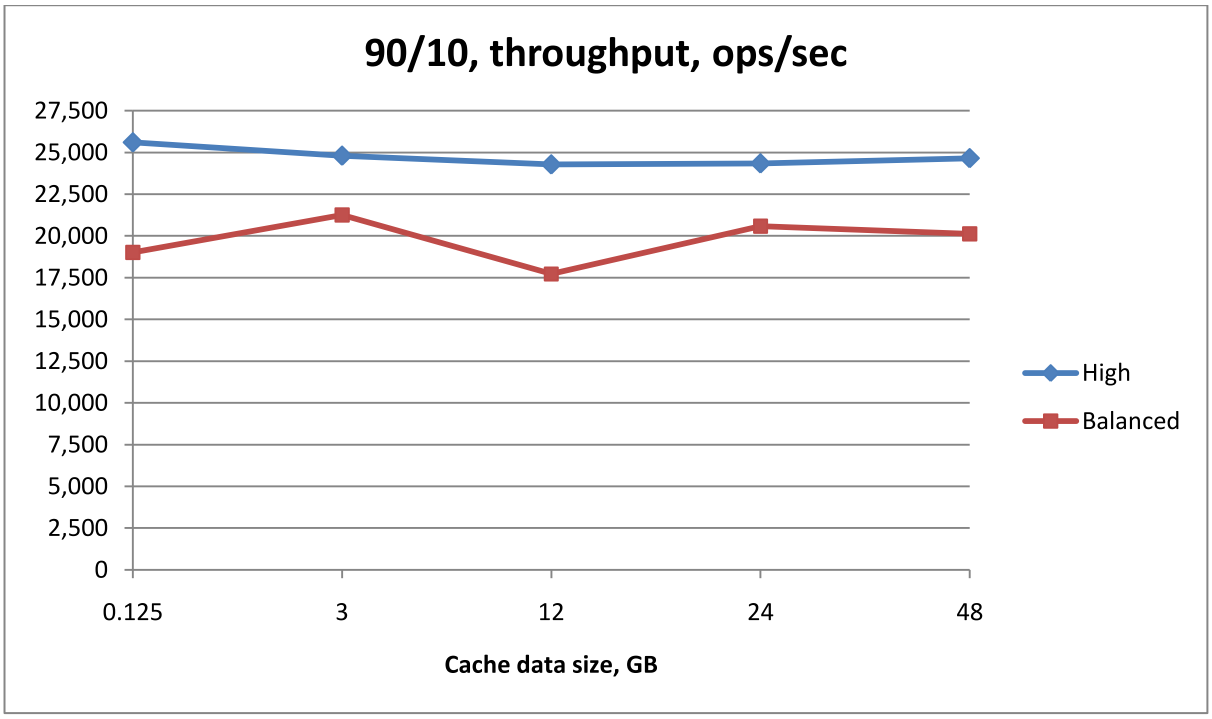

Figure 3: Dependency of throughput from cached data size for direct cache access (12 nodes cluster, 16KB byte array objects, 90% reads & 10% writes, default security)

The results shown in the figure above are expected, because read operations are very lightweight and they do not depend on RAM utilization. For a high ratio of updates, however, cache data size impacts cache operations performance:

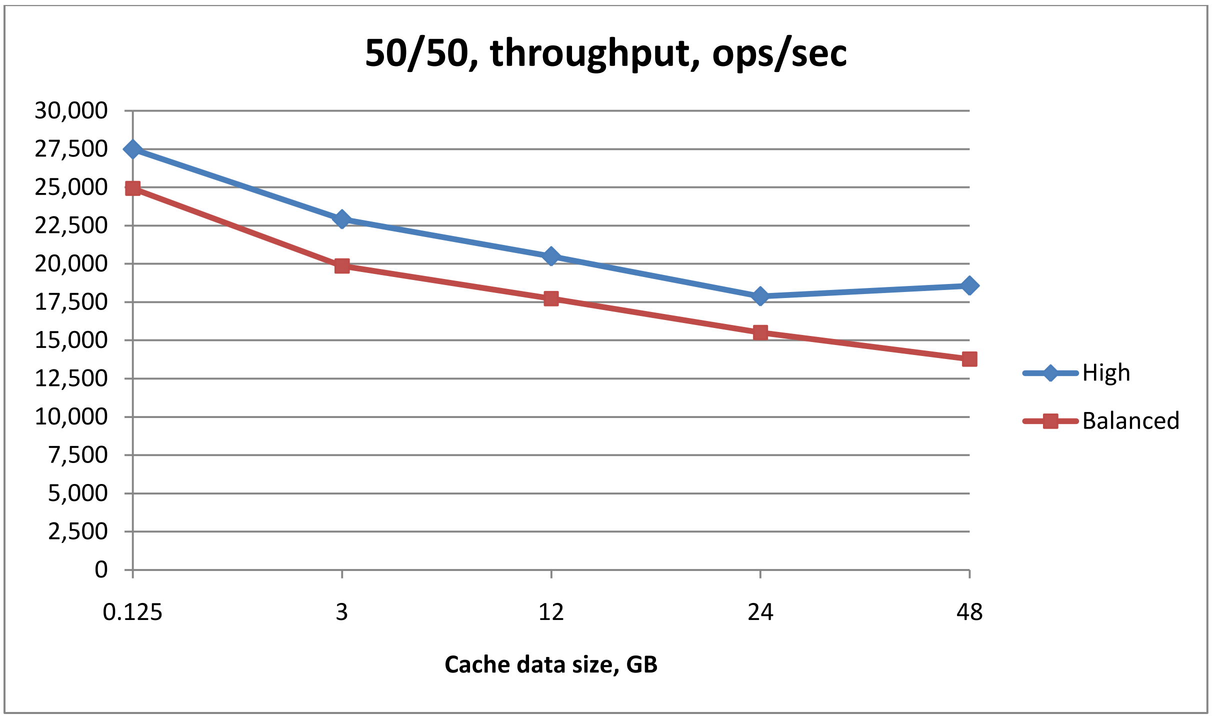

Figure 4: Dependency of throughput from cached data size for direct cache access (12 nodes cluster, 16KB byte array objects, 50% reads & 50% writes, default security)

The difference in performance between 128MB and 48GB cache data size is about 30% for both high and balanced throughput. This happens because of many factors. One of the factors is more pressure on GC when the size of the managed heap is large. For example, according to “% Time in GC” performance counter, for 128MB cache percentage of time spent in GC is almost zero, while for 48GB cache, it is around 5%.

In the subsequent tests, medium cache data sizes are used to maintain 1-2GB of data per cache cluster server.

Scalability

The goal of the tests in this category is to measure the scalability aspects of Windows Server AppFabric Cache, by varying the number of servers. To accomplish this goal, cache performance was tested on different cache cluster sizes.

Set up

Tests in this category were performed with the following fixed parameters:

Parameter name	Value
Cached data size	12GB for 2-node cluster, 24GB for other cluster sizes
Object size	16 KB
Object type	Byte array
Security	Default (SecurityMode=Transport, ProtectionLevel=EncryptAndSign)
Local cache	Not used

Configurations with the following variable parameters were tested:

Load pattern	Cluster size
90% reads & 10% writes	2, 3, 6, 9, 12 nodes
50% reads & 50% writes	2, 3, 6, 9, 12 nodes

Results

As with the cache data size tests, the scalability tests did not show any dependency of latency on the cache cluster size. Latency was approximately the following (in milliseconds):

Point	90% reads / 10% writes	50% reads / 50% writes
High	7.5	9
Balanced	4.3	4.3
Low	2.3	2.4

Scalability tests showed that Windows Server AppFabric Cache cluster can scale well from 2 to 12 nodes:

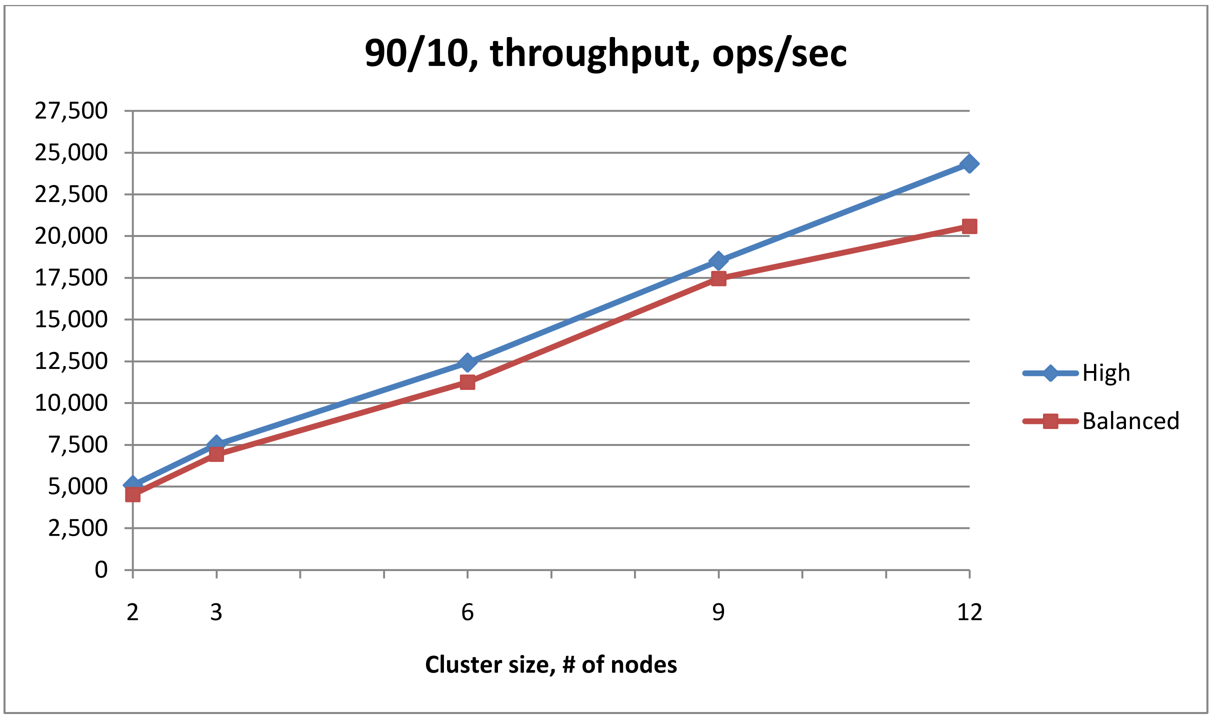

Figure 5: Dependency of throughput from cluster size for direct cache access (16KB byte array objects, 90% reads & 10% writes, default security)

As can be seen from the figure above, performance is almost linear for both high and balanced data collection points.

The previous figure showed results for 90% reads & 10% writes scenario. For 50% reads & 50% writes, Windows Server AppFabric Cache also scales well, as per the following graph:

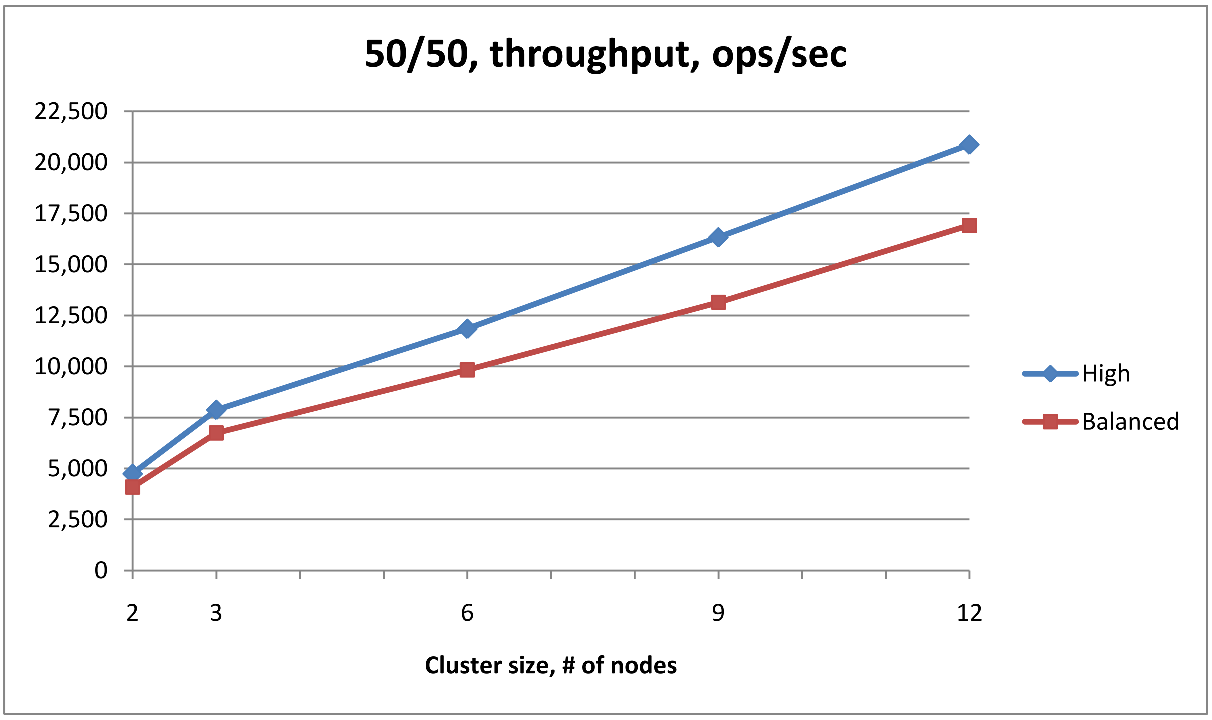

Figure 6: Dependency of throughput from cluster size for direct cache access (16KB byte array objects, 50% reads & 50% writes, default security)

In all the tests above, the bottleneck is in the CPU, not in the network:

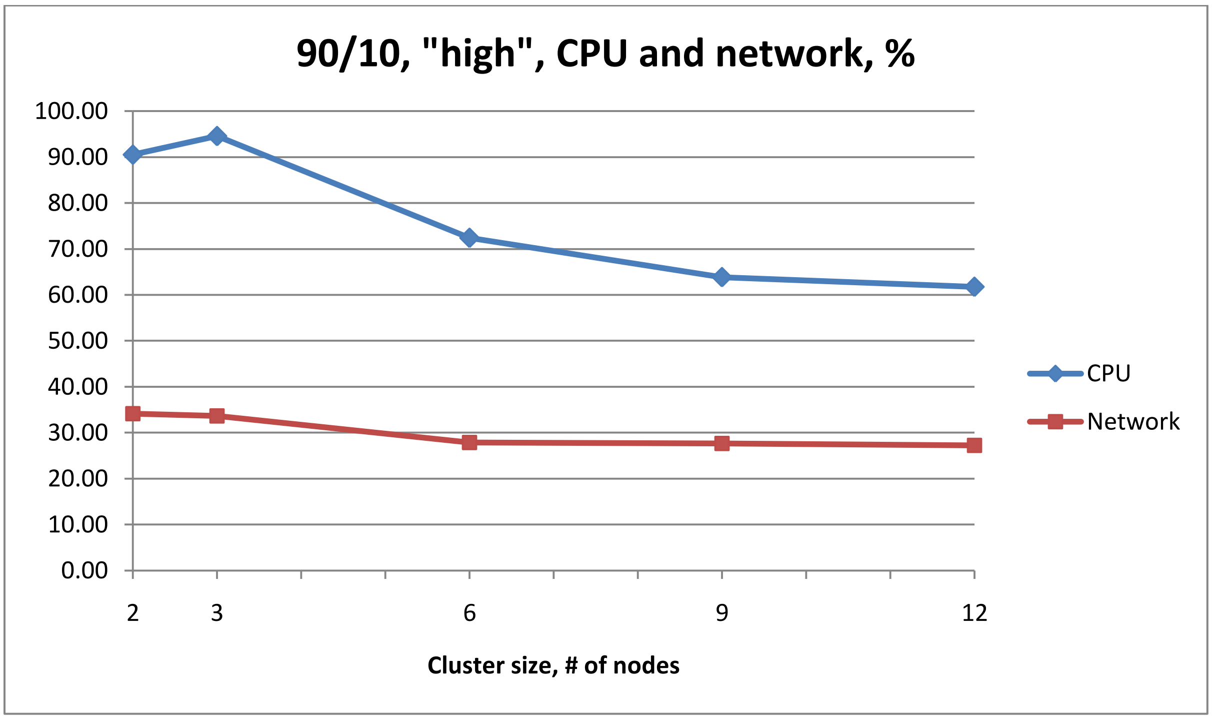

Figure 7: Dependency of CPU and network usage on cache servers from cluster size for direct cache access (16KB byte array objects, 90% reads & 10% writes, default security)

CPU utilization drops after 3 nodes and it can serve as an explanation for why scalability is not completely linear. The reason is related to the fact that the CPU usage displayed in the previous figure is average CPU usage across the cluster.

During the entire test, each Windows Server AppFabric Cache node received equal numbers of read and write requests and the data is distributed equally. However, if at any point of time there are more requests to a single node that may cause uneven load on larger clusters.

Object size

The goal of tests in this category is to measure performance while varying the size of objects that are read and written in the cache.

Set up

Tests in this category were performed with the following fixed parameters:

Parameter name	Value
Cached data size	• 2GB for 512B objects • 6GB for 2KB objects • 12GB for 16KB, 128KB and 1MB objects • 24GB for 4MB objects
Object type	Byte array
Security	Default (SecurityMode=Transport, ProtectionLevel=EncryptAndSign)
Local cache	Not used

Configurations with the following variable parameters were tested:

Cluster size	Load pattern	Object size
3 nodes	90% reads / 10% updates	512B, 2KB, 16KB, 128KB, 1MB, 4MB
12 nodes	90% reads / 10% updates	512B, 2KB, 16KB, 128KB, 1MB, 4MB
12 nodes	50% reads / 50% updates	512B, 2KB, 16KB, 128KB, 1MB, 4MB

Results

As expected, regardless of the cluster size and load pattern, larger objects yield lower throughput:

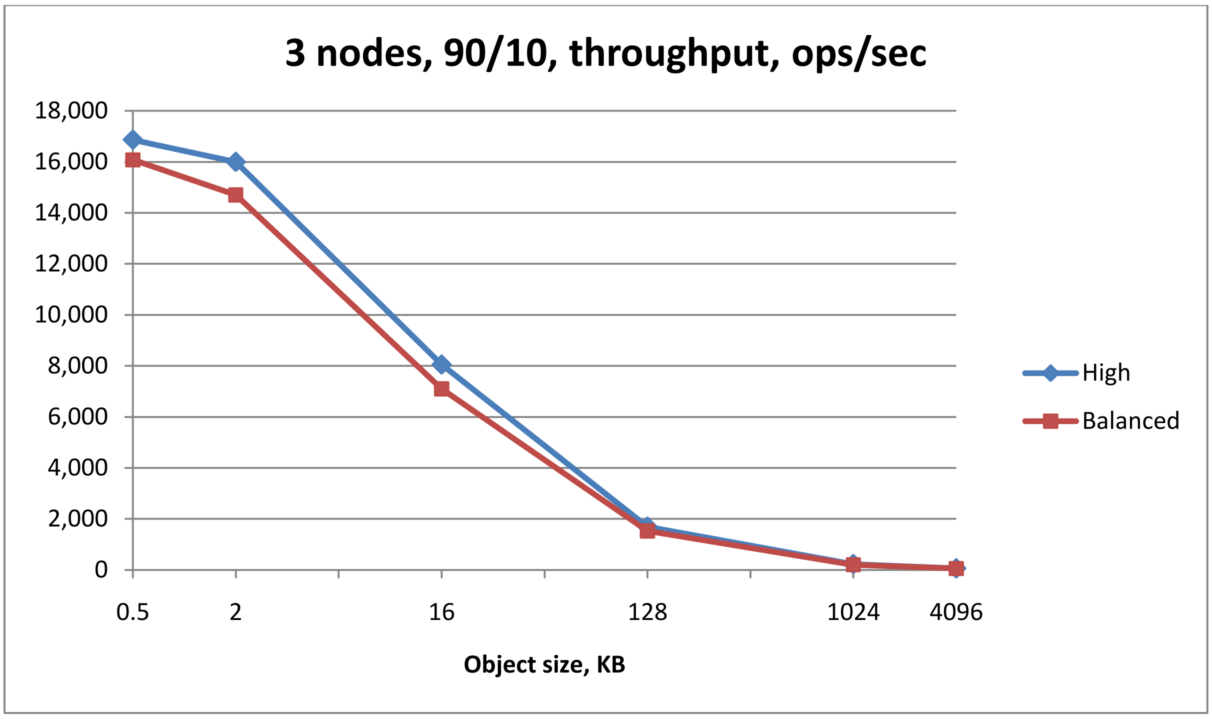

Figure 8: Dependency of throughput from object size for direct cache access (3 node cluster, 90% reads & 10% writes, byte array objects, default security)

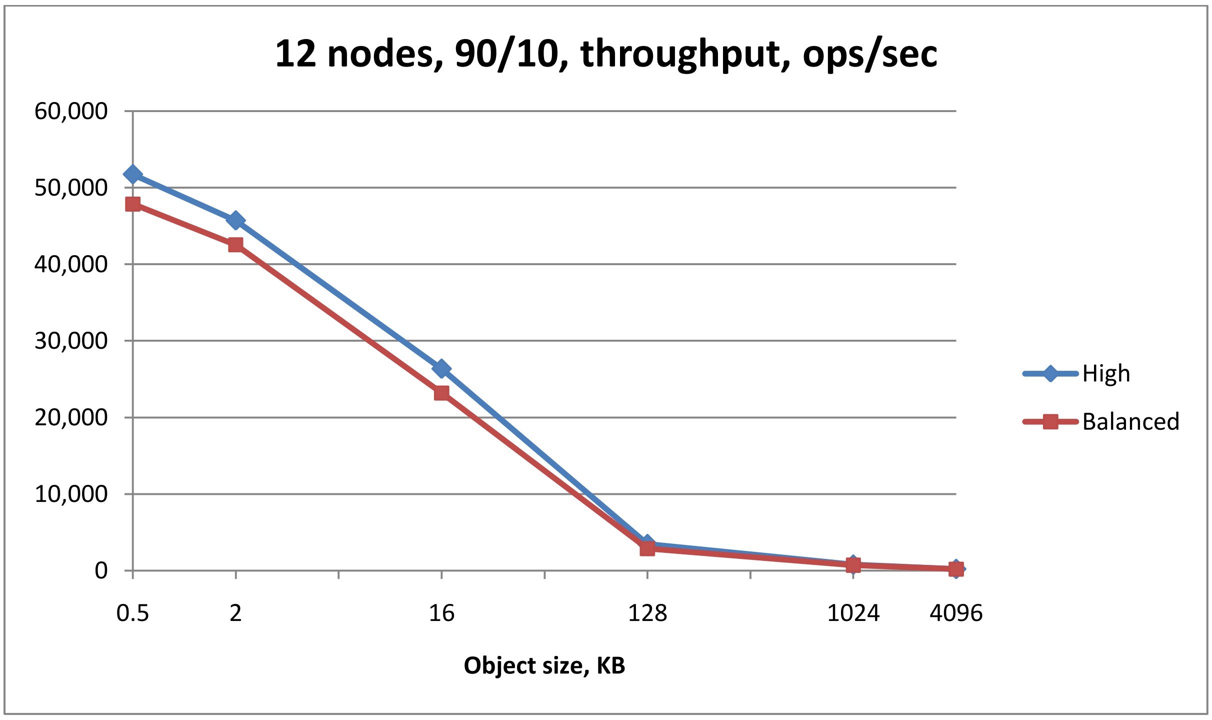

Figure 9: Dependency of throughput from object size for direct cache access (12 node cluster, 90% reads & 10% writes, byte array objects, default security)

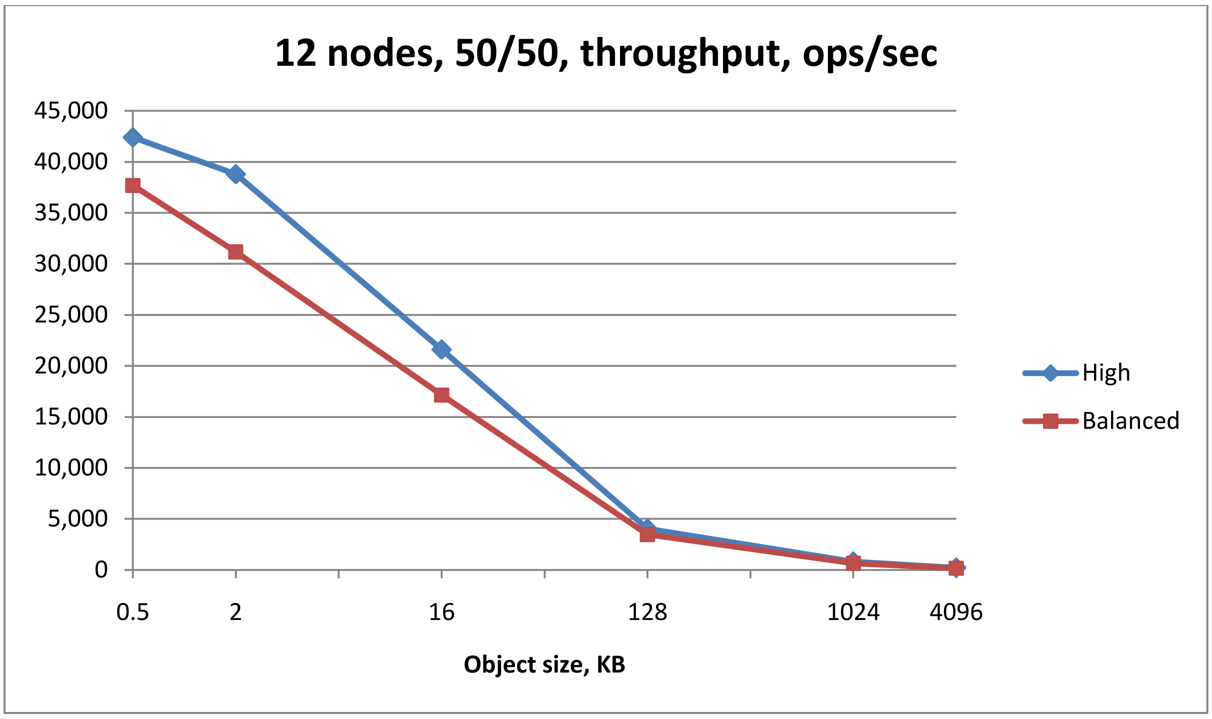

Figure 10: Dependency of throughput from object size for direct cache access (12 node cluster, 50% reads & 50% writes, byte array objects, default security)

Comparing to throughput of small objects, throughput for large objects is too low to grasp on usual graphics with linear scale. To be able to see performance numbers for all object sizes, it is helpful to look at the graphics above using the logarithmic scale on Y axis. Logarithmic scale on Y axis makes the graphic more informative, because X axis is effectively logarithmic as well:

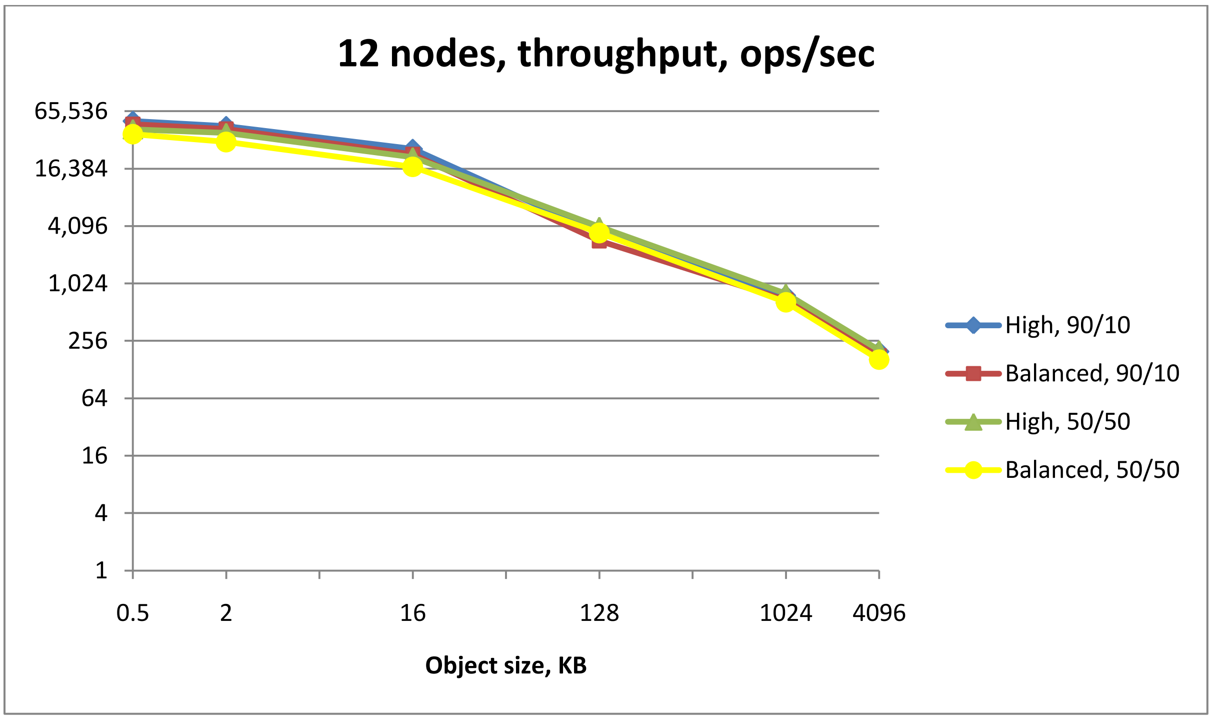

Figure 11: Dependency of throughput from object size for direct cache access (12 node cluster, byte array objects, default security)

Latency also varies greatly from a millisecond for 512B objects to almost a second for 4MB objects:

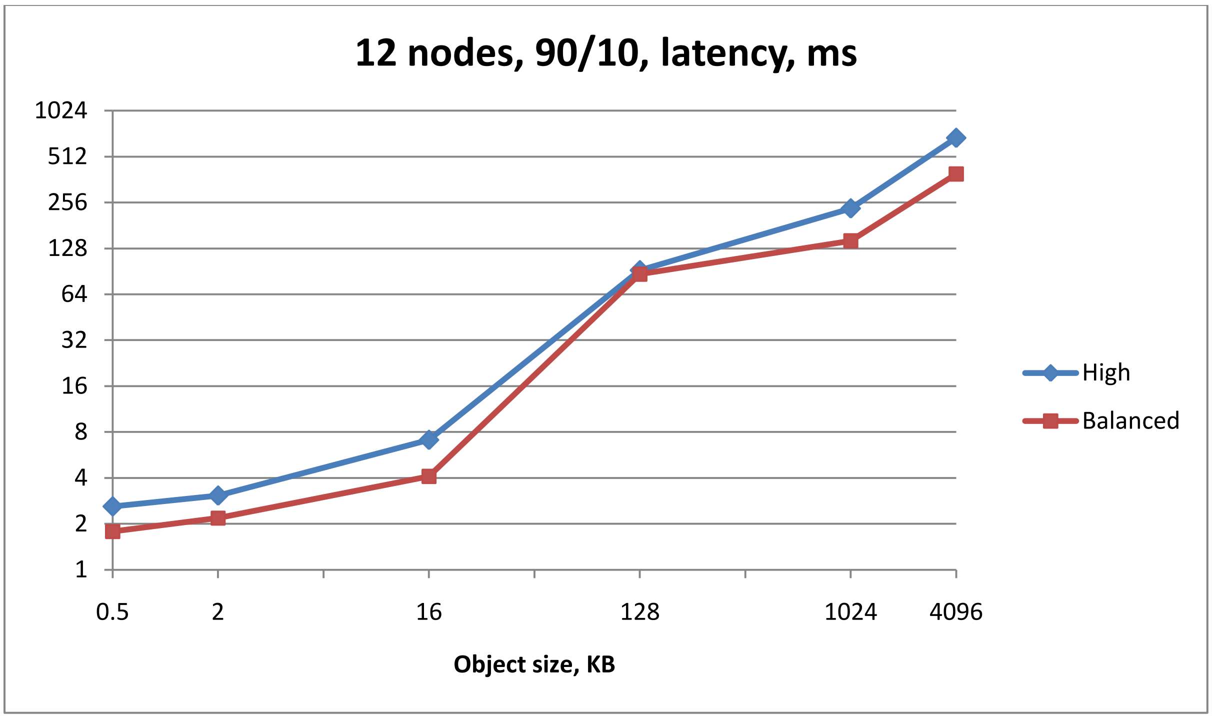

Figure 12: Dependency of latency from object size for direct cache access (12 node cluster, 90% reads & 10% writes, byte array objects, default security)

As can be seen from the graphics, the difference in throughput between 512B and 2KB objects is not very significant taking into account that the difference between object sizes is 4x. Throughput starts decreasing almost in the same pace as object sizes are increasing only after 16KB. To better understand this behavior, it will be useful to take a look at how CPU and network utilization changes depending on the object size:

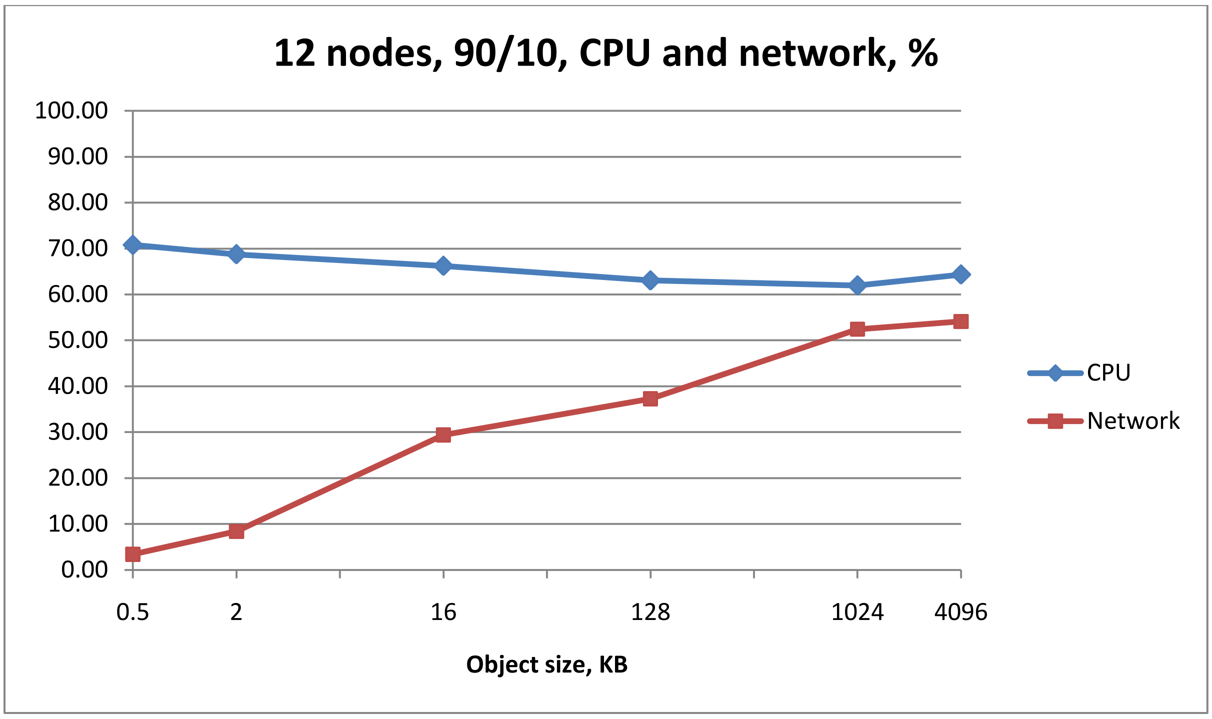

Figure 13: Dependency of CPU and network usage on cache servers from object size for direct cache access (12 node cluster, 90% reads & 10% writes, byte array objects, default security)

The same dependency is valid for the 3 node cluster:

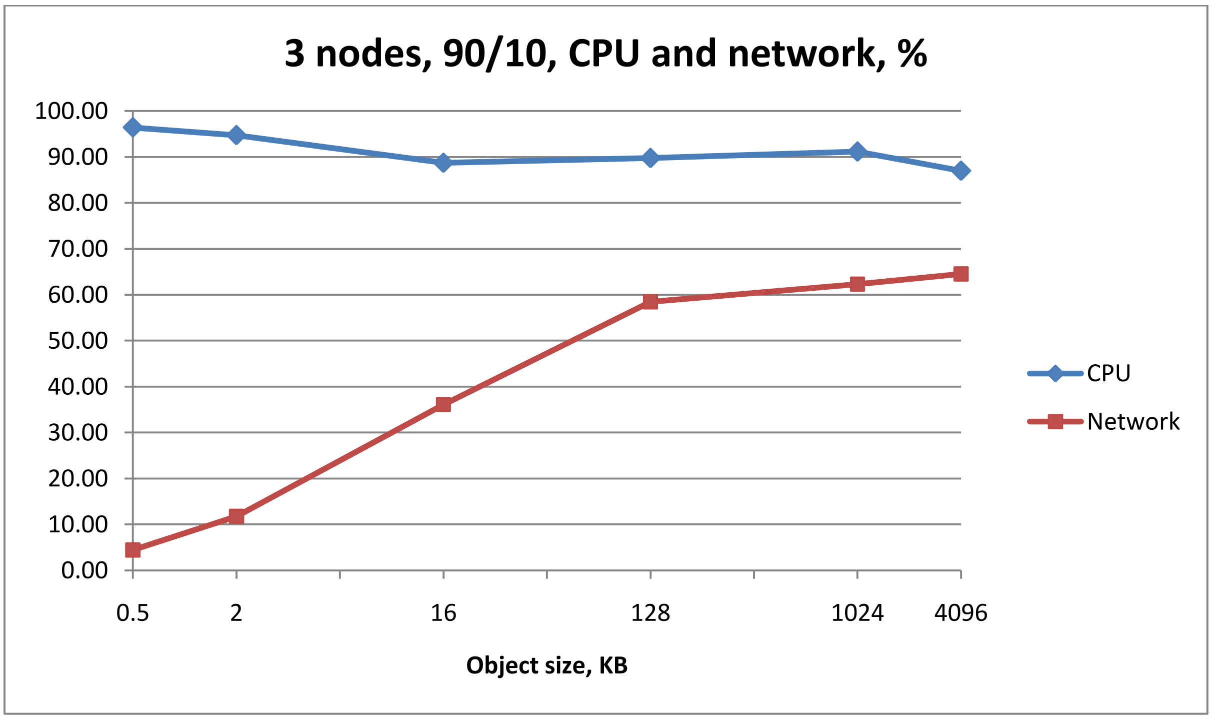

Figure 14: Dependency of CPU and network usage on cache servers from object size for direct cache access (3 node cluster, 90% reads & 10% writes, byte array objects, default security)

This is what happens on the cluster during the tests with different object sizes:

- When the object size is 512B, the most time consuming part is the cache servers' CPU. Most of the CPU time is spent by the cache servers and on receiving and sending small objects over the network. Since the default security setting of EncryptAndSign is used, encryption, decryption, and signing of data sent over the network also take CPU time. However, only a small percentage of the time is used for this, because the amount of data sent is negligible. The network is underutilized.
- When the object size changes from 512B to 2KB, the most time consuming part remains the cache CPU. The network is still underutilized. Since the overhead of exchanging very small objects with the cache over the network is high, there is only a small difference in sending a 512B object and sending a 2KB object. Hence, the throughput remains approximately the same.
- When the object size changes from 2KB to 16KB, the argument from the previous point still applies. The network is used more, but the main bottleneck is still in the CPU and it is still related to operations with cache. That is why the throughput decreases only twice while the object size increases eight times.
- Starting from 128KB objects, network utilization becomes high. This is most clearly seen on the graphic for the 3 node cache. Since the total throughput is relatively small and the objects are large, now the CPU is used mostly to encrypt and sign data sent over the network
- When the object size is 1MB or 4MB, the network is loaded significantly. CPU is used only for data encryption, signing and working with the network. At this stage, the cache cluster is working only on sending and receiving data with the fixed speed in terms of bytes per second. Larger objects decrease throughput in terms of objects per second proportionally to the increase of object size. This explains why throughput for 1MB objects is eight times less than throughput for 128KB objects, and why throughput for 4MB objects is four times less than throughput for 1MB objects.

Type complexity

The goal of the tests in this category is to measure performance while varying the complexity of .NET type of objects stored in cache.

Set up

Tests in this category were performed with the following fixed parameters:

Parameter name	Value
Cached data size	2GB for 512B objects, 6GB for 2KB objects, 12GB in other cases
Object type	Complex type called “First” with the definition listed below
Security	Default (SecurityMode=Transport, ProtectionLevel=EncryptAndSign)
Local cache	Not used

The definition of the complex type is the following:

```
 [Serializable]
private sealed class First
{
    private byte[] byteArray;
    private Second inner;
    ....
}

[Serializable]
private sealed class Second
{
    private int[] intArray;
    private Third inner;
    ....
}

[Serializable]
private sealed class Third
{
    private string[] stringArray;
    ....
}
```

The “First” class used in the tests in this category contains two nested classes. After instantiation, each object of the “First” class ultimately contains: one object of the “Second” class; one object of the “Third” class; one byte array; one array of integers; and one array of strings. Lengths of arrays are chosen appropriately to yield a pre-configured amount of serialized data. This amount of serialized data is configured by the “Object size” parameter.

The table below lists the most interesting configurations of the variable parameters. Tests were performed only in these configurations:

Cluster size	Load pattern	Object size (post-serialization)
2 nodes	90% reads / 10% updates	512B, 2KB, 16KB, 128KB
3 nodes	90% reads / 10% updates	512B, 2KB, 16KB, 128KB, 1MB
3 nodes	50% reads / 50% updates	512B, 2KB, 16KB, 128KB
4 nodes	90% reads / 10% updates	512B, 2KB, 16KB, 128KB
6, 12 nodes	90% reads / 10% updates	16KB

Results

The interesting fact about performance of complex types is that usage of complex types does not influence cache performance at all. This can be seen from the figures below, where cache operations with complex types perform essentially the same as cache operations with simple types:

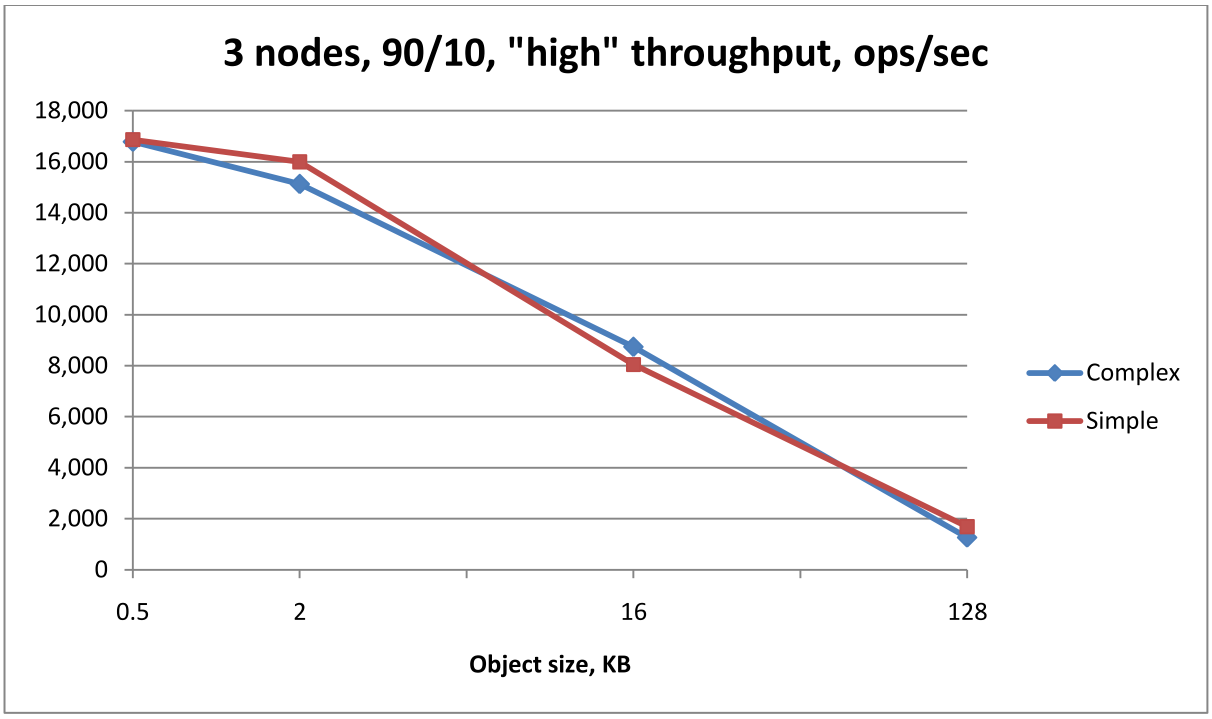

Figure 15: Dependency of throughput from object size for simple and complex types for direct cache access (3 node cluster, 90% reads & 10% writes, default security)

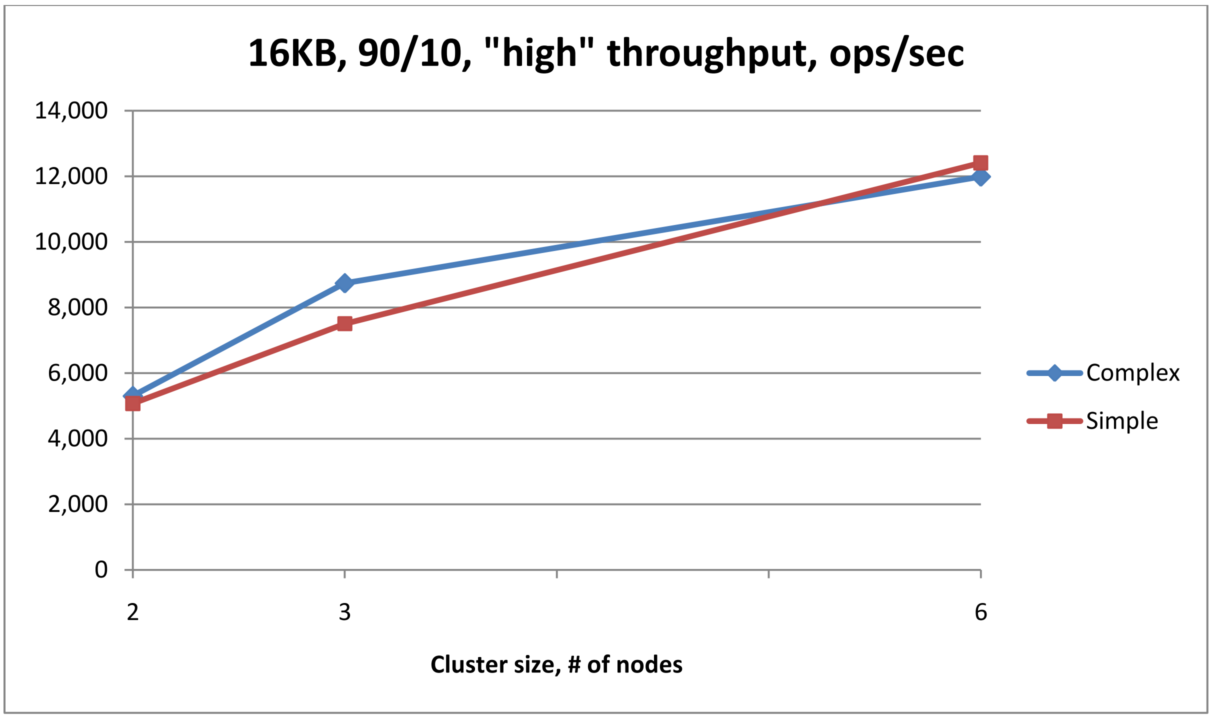

Figure 16: Dependency of throughput from cluster size for simple and complex types for direct cache access (90% reads & 10% writes, 16KB objects, default security)

Throughput is minimally affected, because Windows Server AppFabric Cache servers work only with serialized data. Objects are never deserialized on the server side. So, for the cache servers, type complexity does not make any difference as soon as the size of serialized data is the same.

It can be noticed that the figures above show only test results for small cache cluster sizes and small object sizes. These points were chosen to ensure that the load cluster will not be a bottleneck, because type complexity affects the client performance dramatically. On load agents, CPU is used significantly for serialization and deserialization, especially for large objects:

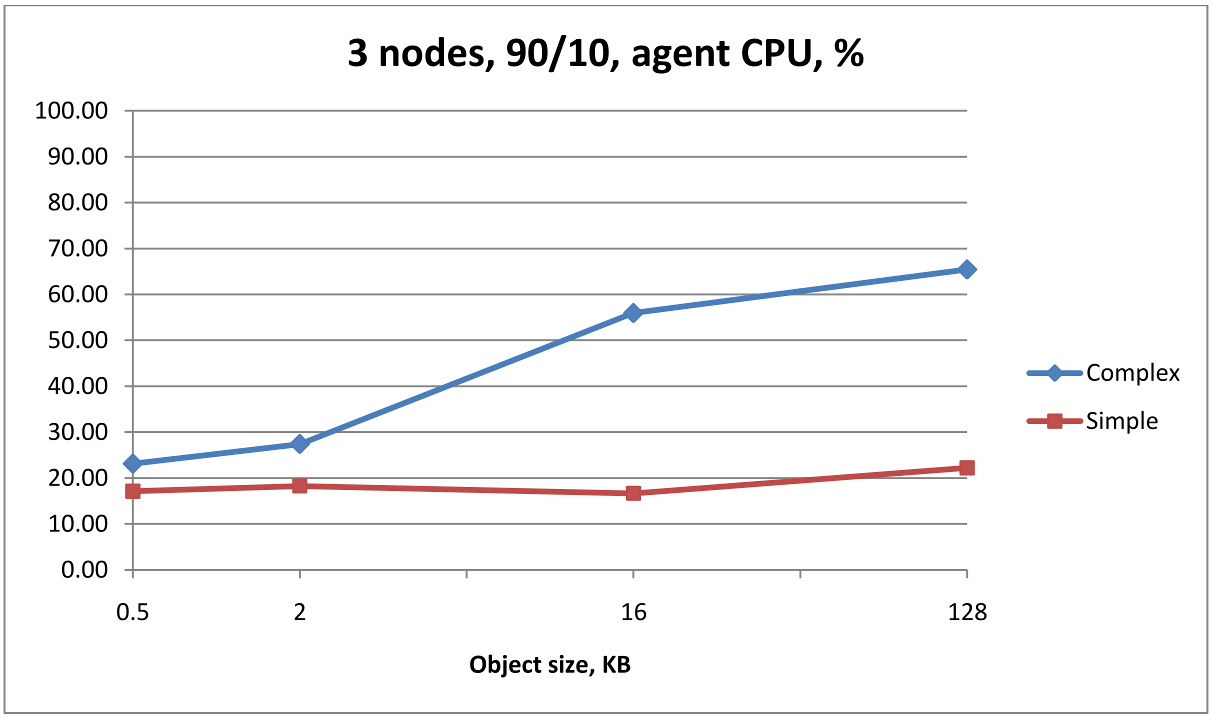

Figure 17: Dependency of CPU usage on load agents from object size for simple and complex types for direct cache access (3 nodes, 90% reads & 10% writes, default security)

On cache servers, however, the CPU for both simple and complex objects is the same. It is still close to 100%, because load generation agents are not exhausted and can still utilize the cache cluster in full.

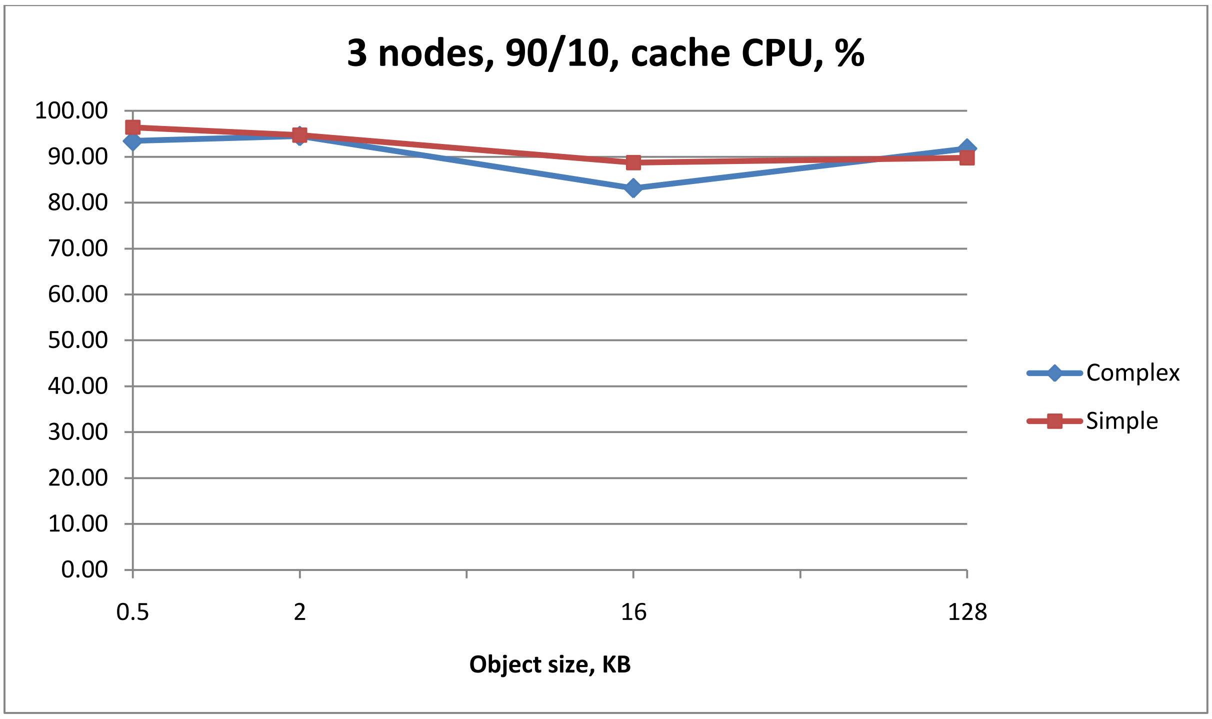

Figure 18: Dependency of CPU usage on cache servers from cluster size for simple and complex types for direct cache access (3 nodes, 90% reads & 10% writes, default security)

Higher CPU utilization on load generation agents means that for complex types, scalability of clients is more important than scalability of servers.

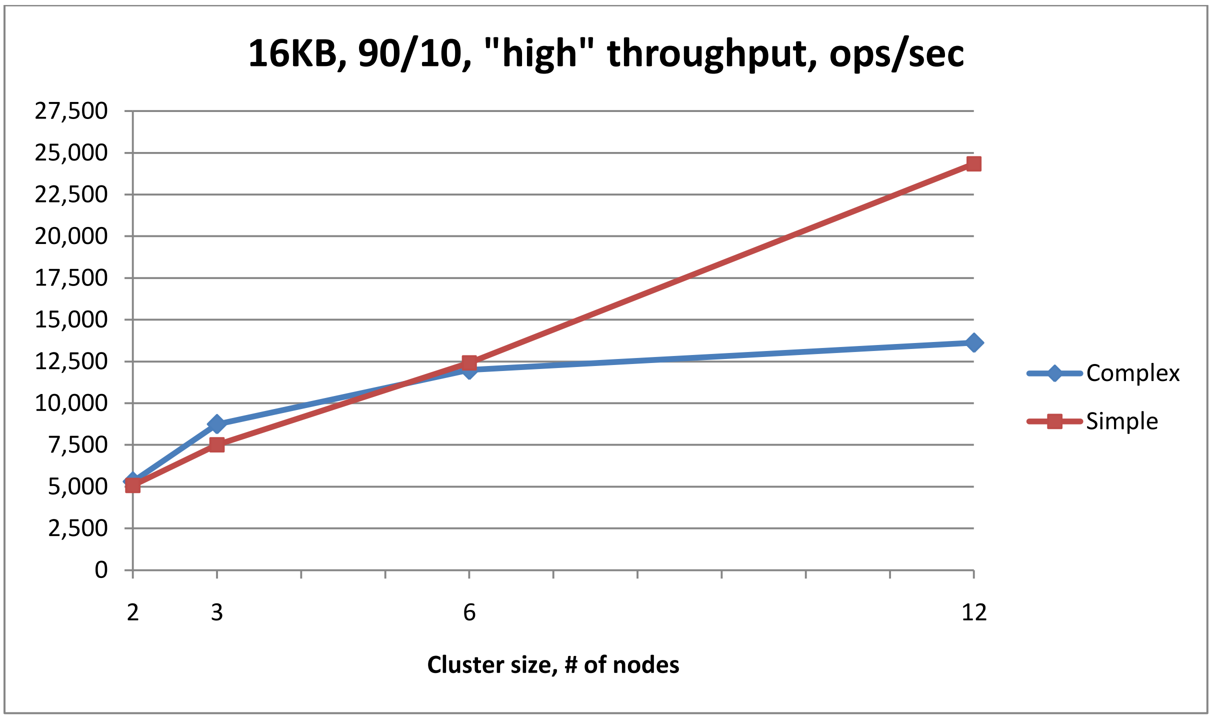

Figure 19: Dependency of throughput from cluster size for simple and complex types for direct cache access (90% reads & 10% writes, 16KB objects, default security)

Although the previous graph appears to suggest that throughput of complex types is lower for larger cache sizes, this is not the case. If there were more load agents, they would be able to fully utilize the cache cluster and throughput would be the same.

Depending on how complex the type is, more clients will be needed to load the same number of cache servers. The reason is that since serialization and de-serialization happen on the client side, additional client machine CPU is used for that. As has been shown in the previous figure, the load agents cluster that was able to easily load the cache cluster with simple types is not able to load it with complex types:

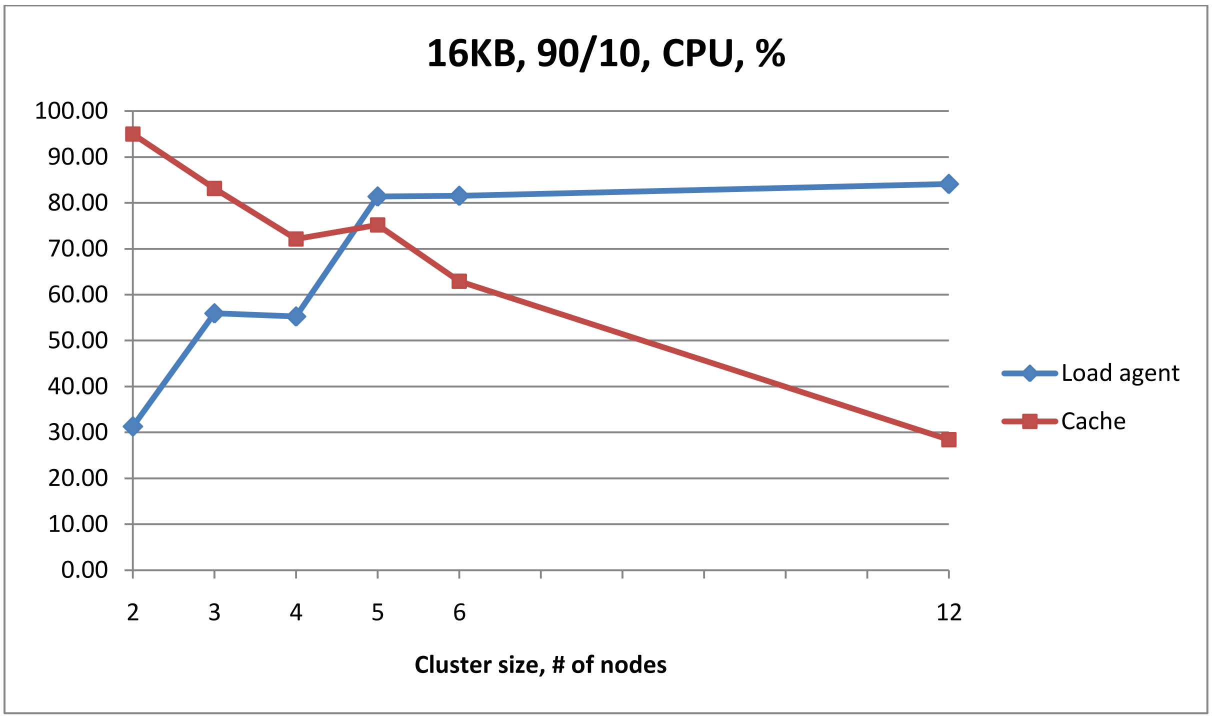

Figure 20: Dependency of CPU and network usage on load agents and cache servers from cluster size for direct cache access (90% reads & 10% writes, 16KB complex type objects, default security)

As can be seen from the figure above, a two or three times larger load generation cluster is required to load 12 cache servers. In some cases, custom serialization may help to improve client-side scalability and the overall performance when working with complex types.

Read performance

The goal of the tests in this category is to measure the difference in performance between "Get non-existing object", "Get existing object" and "BulkGet" cache operations in different situations. Get non-existing object means that the client invokes a "Get" operation with a key that does not exist in the cache, and no object is returned to the client. In other words, this is a *cache miss*. Get existing object means that a "Get" operation is invoked with a key that exists in the cache, and an object is returned to the client. In other words, this is a *cache hit*.

It should be taken into account that in real applications cache miss usually results in worse performance, because it is usually followed by a request to external data source. Since external data source access time is application specific, this section shows performance of pure cache miss.

Set up

Tests in this category were performed with the following fixed parameters:

Parameter name	Value
Cached data size	3GB for 3-node cluster, 12GB for 12-node cluster
Object type	Byte array
Security	Default (SecurityMode=Transport, ProtectionLevel=EncryptAndSign)
Local cache	Not used

Variable parameters were the following:

Load pattern	Cluster size	Object size (post-serialization)
Get existing (cache hit)	3, 12 nodes	512B, 16KB
Get non-existing (cache miss)	3, 12 nodes	512B, 16KB
Bulk get of 10 objects	3, 12 nodes	512B, 16KB
Bulk get of 100 objects	3, 12 nodes	512B, 16KB

Since a "BulkGet" operation works with an explicit region, a large number of regions were created in the cache and they were picked randomly for each bulk get operation. The main reason for this is that named regions exist on a single cache host and the performance numbers would be affected if all the operations were going to a single region on a single host. Default region, which was used in the previous tests, is an exception and it distributes data across the whole cache cluster. So, even data distribution and load balancing should be considered carefully when using named regions.

Results

As expected, “Get non-existing” is very cheap operation, because there was no need to return any data on the client. In real world, it is followed by request to an external data source and Put cache request, so performance is very application specific. Obviously, cache may handle more “Get non-existing” operations than “Get existing”, especially, in the case of relatively large objects.

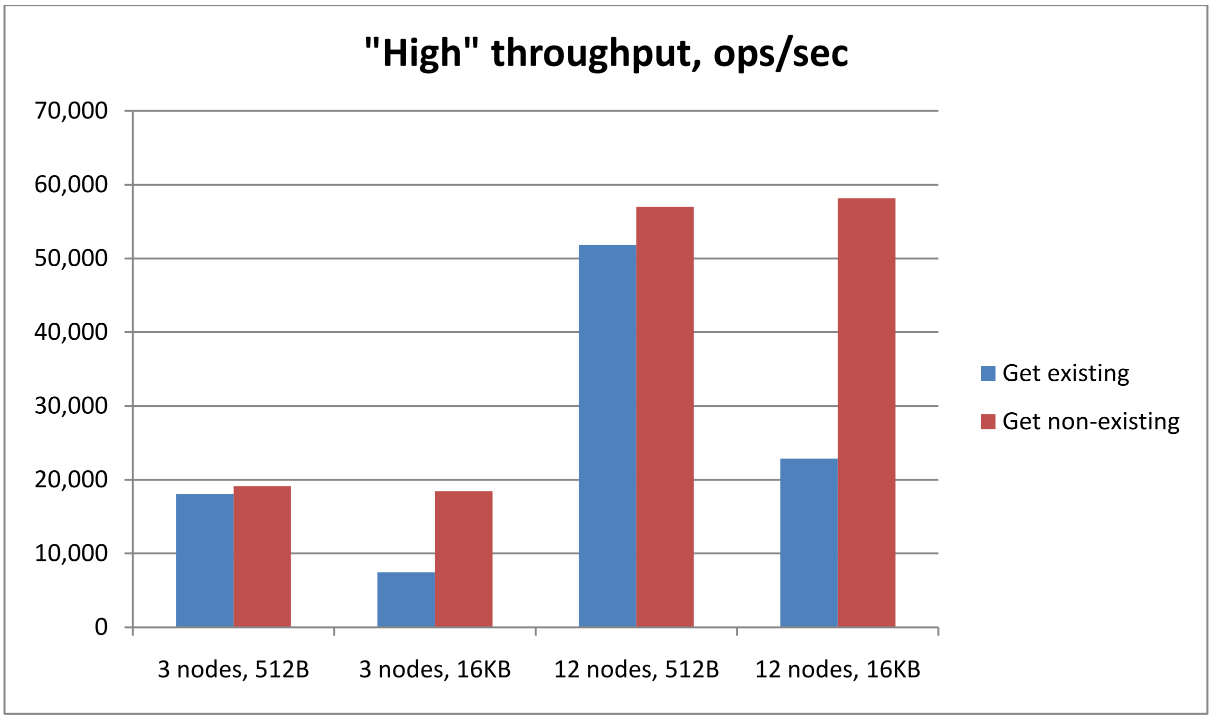

Figure 21: Dependency of throughput from cluster size, object size and different "Get" methods for direct cache access (byte array objects, default security)

In case of small object, the difference between "Get existing" and "Get non-existing" is not so significant, because even in case of cache misses, AppFabric Cache still needs to serve requests and some data still travels over the network.

In the case of a "BulkGet", bulk operations performed much better in the case of small objects. The difference became even more significant for larger bulk operations. A Bulk get of 100 of 512B objects performed much better than Bulk get of 10 of 512B objects:

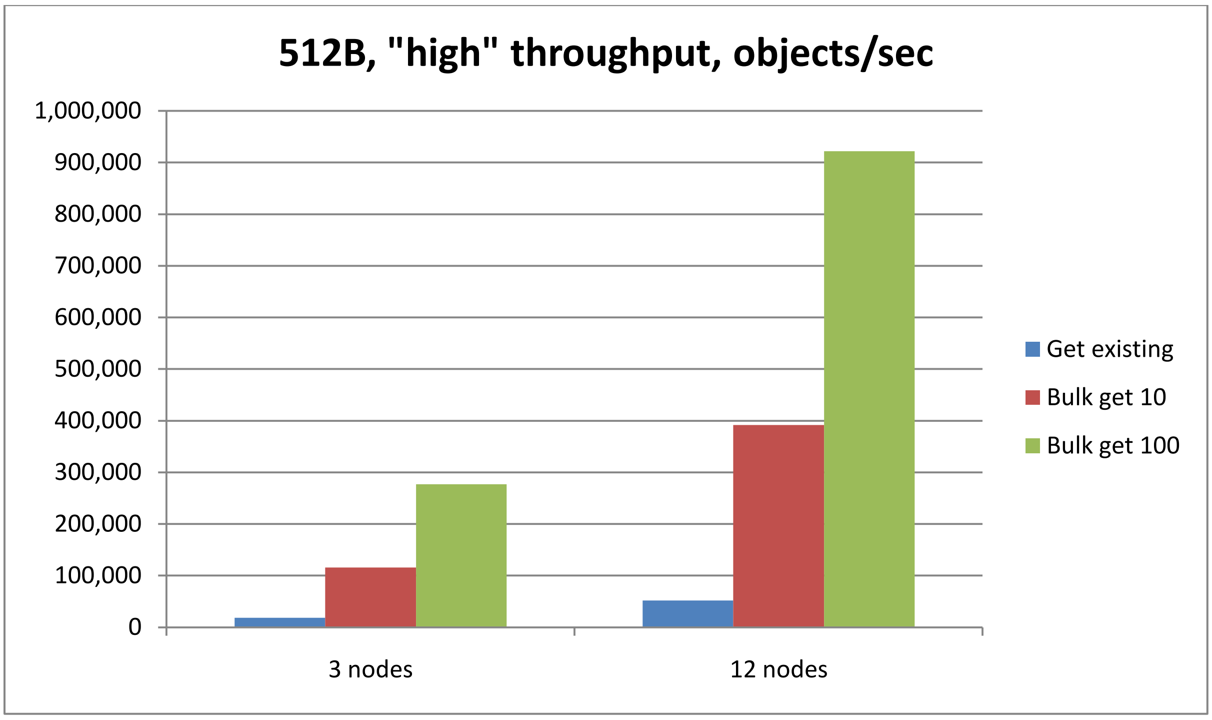

Figure 22: Dependency of throughput from cluster size and different "Get" methods for direct cache access (512B byte array objects, default security)

For larger objects, there is almost no difference between a Bulk get of 10 objects and a Bulk get of 100:

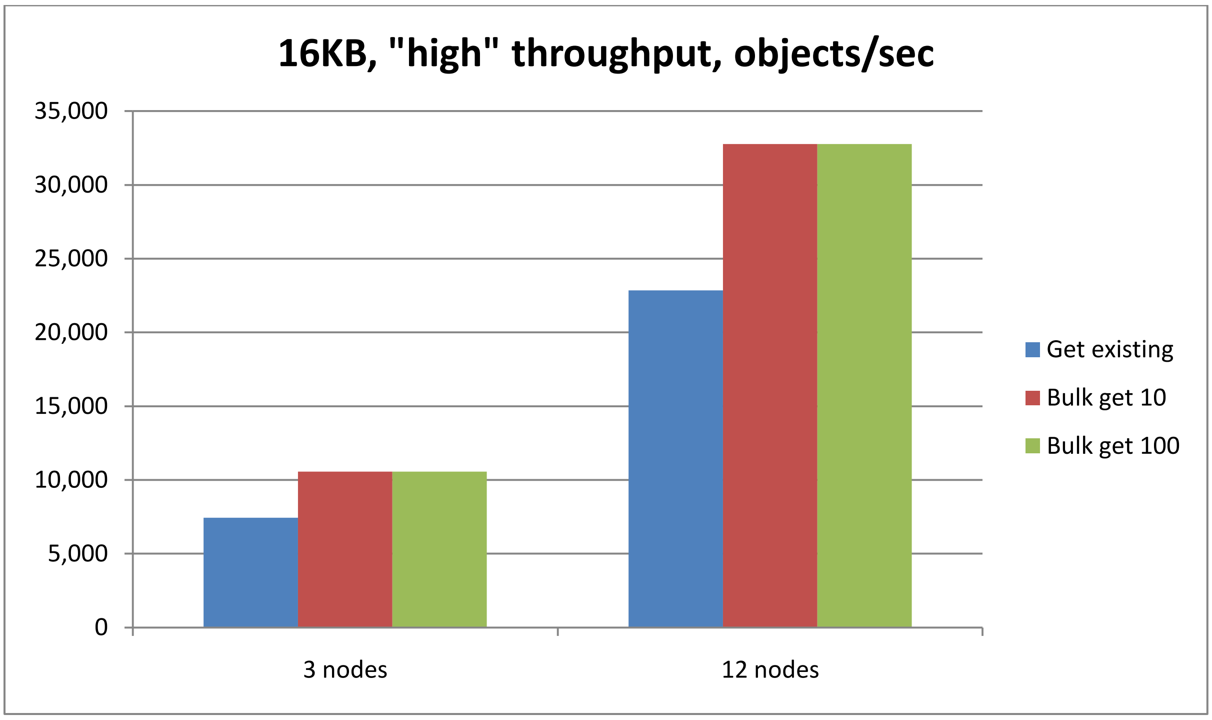

Figure 23: Dependency of throughput from cluster size and different "Get" methods for direct cache access (16KB byte array objects, default security)

Please note that the above figures show the number of objects read per second. In terms of operations per second, a BulkGet of 100 items will obviously give less throughput than a "Get" of a single element.

As was mentioned in the section related to testing different object sizes, invoking cache operations with small objects causes more network overhead. It is more efficient to get a 5KB data chunk consisting of ten 512B objects than retrieving ten 512B objects with 10 distinct calls. Reading a 50KB data chunk consisting of a hundred 512B objects shows even better performance since the network overhead is significantly reduced. Bulk operations also reduce CPU cycles on the cache servers since it needs to process one command and send back bulk results. Hence the throughput goes up

16KB objects, on the other hand, saturate the network much faster and have relatively less network overhead when compared to 512B objects. This explains why getting ten 16KB objects does not improve performance the same as in the case of 512B objects. Larger batches of a hundred 16KB objects does not improve performance at all, because most of the time is spent on serializing, encrypting and transmitting the objects over the network.

Locking performance

The goal of the tests in this category is to measure performance between optimistic and pessimistic locking operations from the Windows Server AppFabric Cache API. To accomplish this goal, tests with high and low concurrency were implemented. High concurrency means a high percentage of conflicting update operations when the same object is updated from different threads. Low concurrency means a low percentage of conflicting updates. Different cached data sizes were used to simulate concurrency.

Set up

Tests in this category were performed with the following fixed parameters:

Parameter name	Value
Cluster size	12 nodes
Object size	512B was chosen to improve latency and eliminate the impact of concurrency inside Windows Server AppFabric Cache
Object type	Byte array
Security	Default (SecurityMode=Transport, ProtectionLevel=EncryptAndSign)
Local cache	Not used

Variable parameters were the following:

Load pattern	Cached data size
Updates with optimistic locking	1K, 10K, 100K objects
Updates with pessimistic locking	1K, 10K, 100K objects

Optimistic update was implemented using the following pseudo-code:

```
try
{
    cache.Get(random_key, out version);
    sleep(50 + random(50));
    cache.Put(random_key, value, version);
}
catch (VersionMismatch)
{
    sleep(100);
    go to the beginning of try block;
}
```

Pessimistic update was implemented using the following pseudo-code:

```
try
{
    cache.GetAndLock(random_key, out lockHandle);
    sleep(50 + random(50));
    cache.PutAndUnlock(random_key, value, lockHandle);
}
catch (Locked)
{
    sleep(100);
    go to the beginning of try block;
}
```

For experimental purposes, the code above uses constant backoffs. It means that the wait time after a failed update is constant.

Results

In this section, an operation means a complete cycle of getting and putting an object in the cache and effectively consists of several Get and Put requests. A single operation may consist of more than just one pair of Get and Put requests if the Put request fails with conflict. In this case, Get&Put requests will be repeated until the object is finally updated in the cache.

The test uses random object access and, hence, conflict rate decreases when the number of objects in the cache increases:

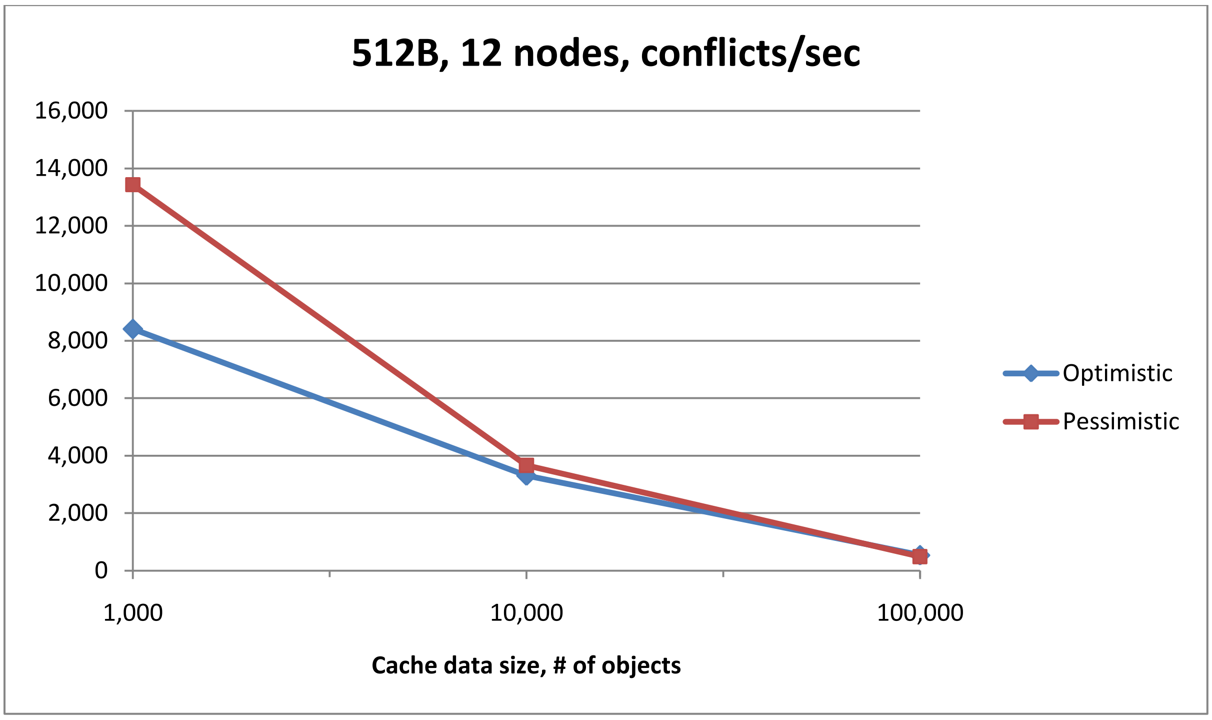

Figure 24: Dependency of conflict rate from cached data size for direct cache access (12 nodes, 512B byte array objects, default security)

Starting from 100,000 objects the conflict ratio is very low and does not influence cache performance. Both latency and throughput improve when the conflict ratio decreases:

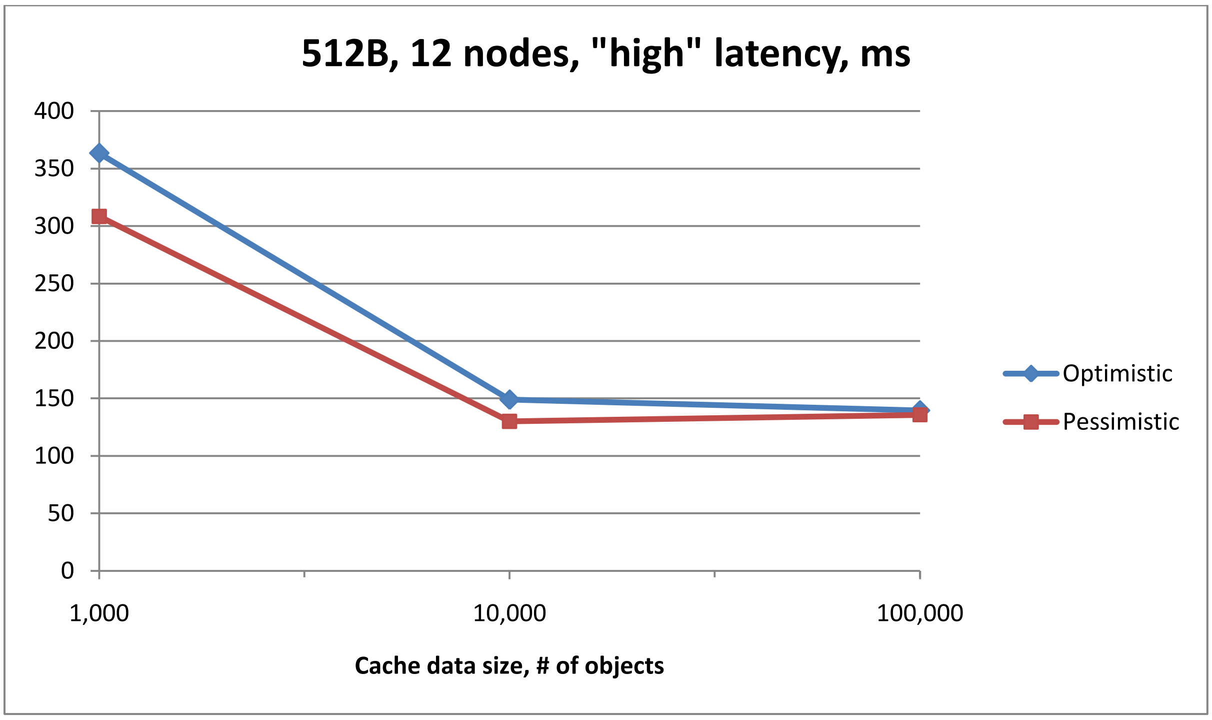

Figure 25: Dependency of latency from cached data size for direct cache access (12 nodes, 512B byte array objects, default security)

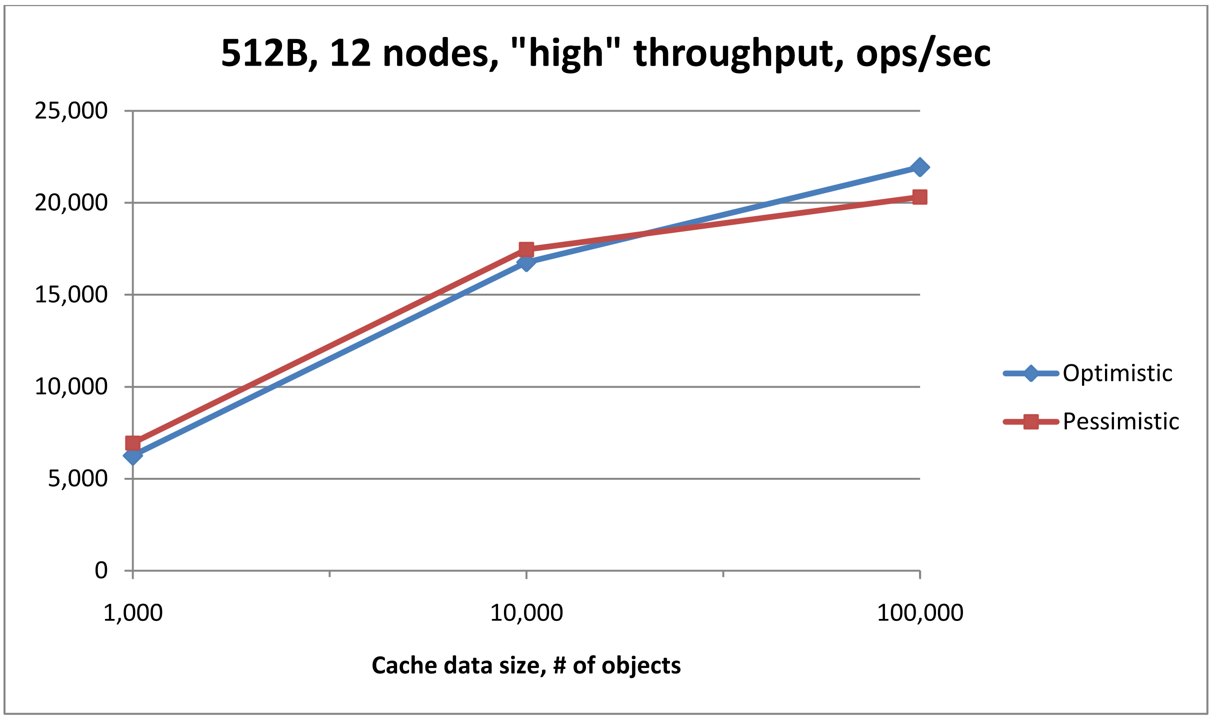

Figure 26: Dependency of throughput from cached data size for direct cache access (12 nodes, 512B byte array objects, default security)

Surprisingly, there is almost no difference between pessimistic and optimistic locking for both low and high conflict rate. This happens because GetAndLock and PutAndUnlock operations are just as fast as their non-locking analogs. Of course, this does not mean that pessimistic and optimistic locking will perform the same in all situations. This benchmarking uses a very simple workload and very fine-grained locks. In real-world applications, usual considerations regarding optimistic and pessimistic locking still apply.

It is important to notice that usage of constant backoffs on conflicts may cause the majority of conflicts to fall on one node. This situation is especially dangerous if there are thousands of clients trying to update the same object located on the some node. After every update cycle only one client will succeed and others will retry again until all clients succeed. If there are 1,000 clients trying to update a single object concurrently, about 500,000 requests will be sent to a single node to update an object.

If that node is not fully loaded then this situation is temporary; if the node is saturated and there are many new requests then such misbalance can last very long and even cause denial of service. Of course, average cluster performance will decrease dramatically. The following graphs show such a situation:

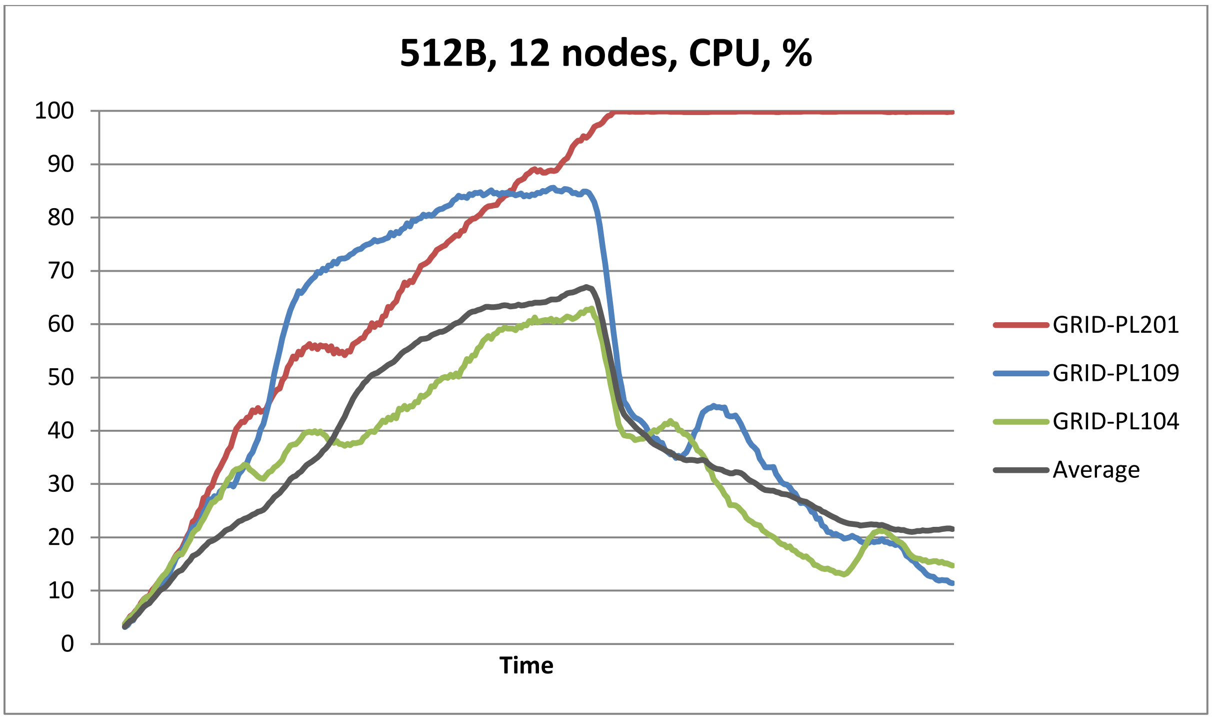

Figure 27: CPU usage on cache servers showing the situation when GRID-PL201 node is flooded with repeated concurrent update requests to the same object (12 nodes, 512B byte array objects, default security)

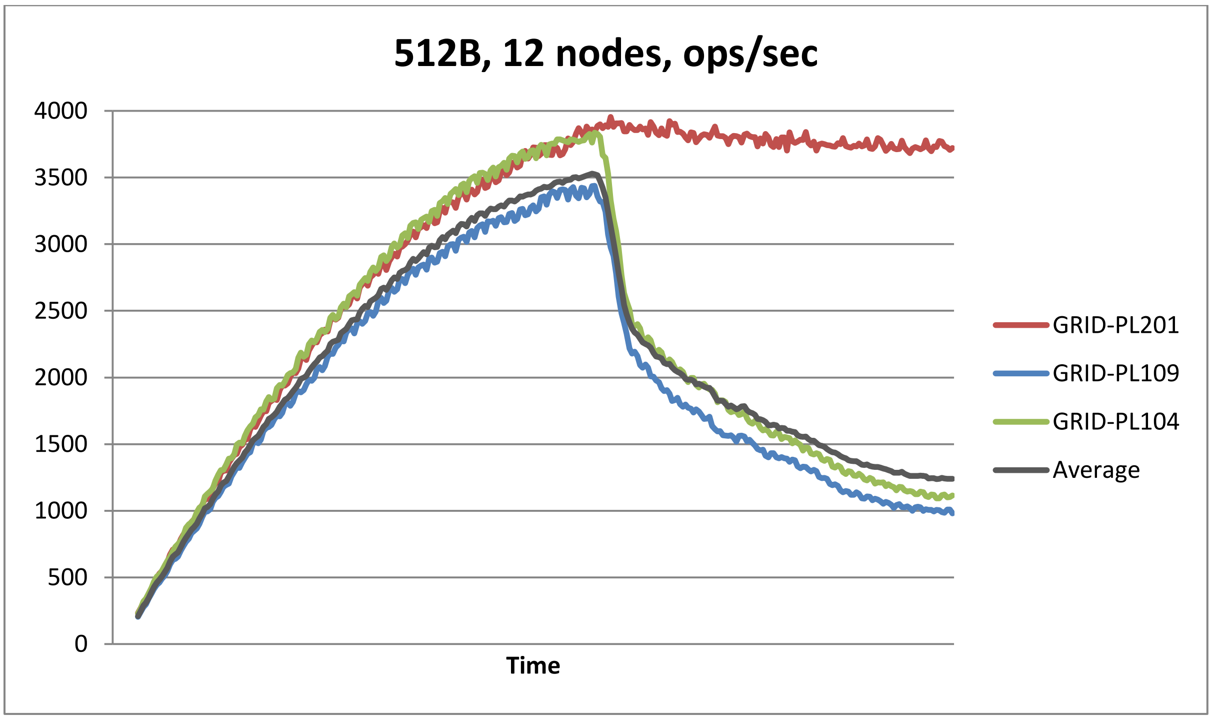

Figure 28: Throughput on cache servers showing the situation when GRID-PL201 node is flooded with repeated concurrent update requests to the same object (12 nodes, 512B byte array objects, default security)

This behavior can happen with both optimistic and pessimistic locks. It is recommended to use exponential or random backoff to avoid it. Information about exponential backoff can be found on

Wikipedia. In the exception handler in the pseudo-code for pessimistic and optimistic locking in the beginning of this chapter, implementing random backoff requires replacing the constant time with random time passed as an argument to the *sleep* function.

Security impact

The goal of the tests in this category is to measure Windows Server AppFabric Cache operations performance with the different security settings.

Set up

Tests in this category were performed with the following fixed parameters:

Parameter name	Value
Cached data size	3GB
Load pattern	90% read & 10% update
Object type	Byte array
Local cache	Not used

To better understand the impact of security and to minimize the number of tests, the following configurations were tested:

Cluster size	Object size	Security settings
3 nodes	16KB	None, Encrypt, EncryptAndSign
6 nodes	16KB	None, Encrypt, EncryptAndSign
6 nodes	128KB	None, Encrypt, EncryptAndSign
12 nodes	16KB	None, Encrypt, EncryptAndSign
12 nodes	128KB	None, Encrypt, EncryptAndSign

Results

By default, Transport security with EncryptAndSign is used. This puts additional pressure on the CPU. Unlike complex types and serialization, security impacts CPU on both cache servers and client. Tests showed that lesser security improves performance dramatically:

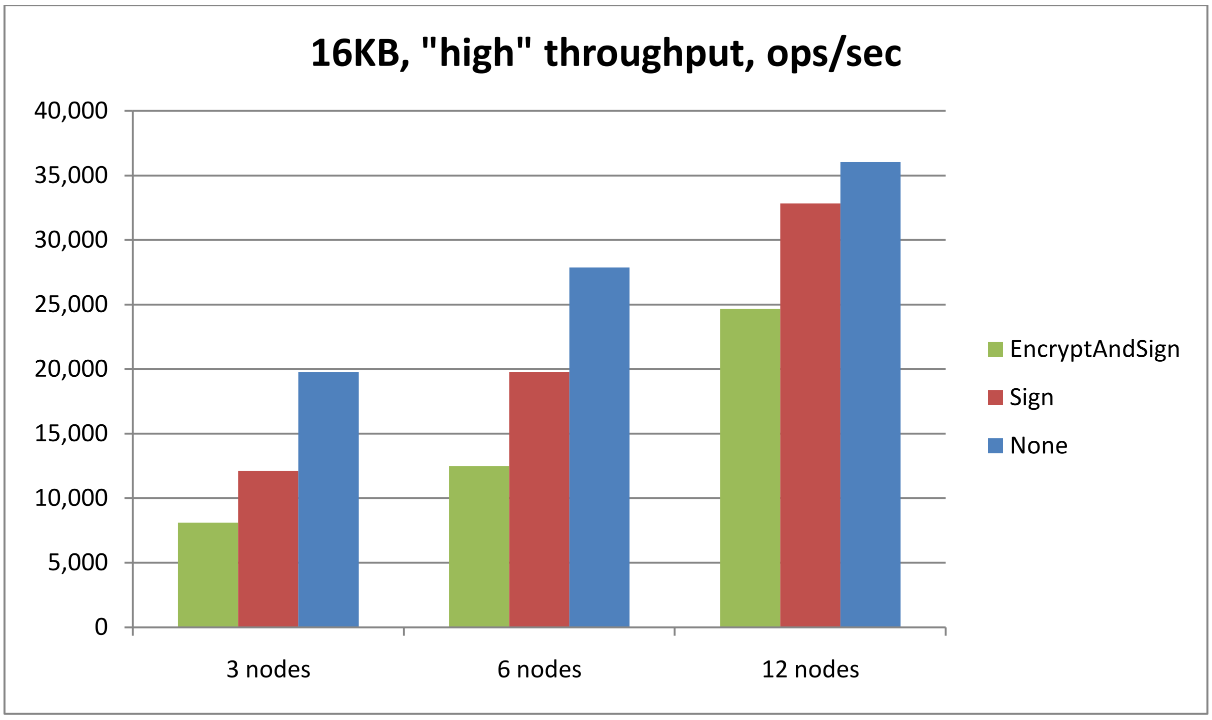

Figure 29: Dependency of throughput from security settings and cluster size for direct cache access (16KB byte array objects, 90% reads & 10% writes)

As can be seen from the figure above:

- Just using ‘Sign’ instead of default ‘EncryptAndSign’ gives about a 1.4x boost in operations per second
- Disabling security instead of default ‘EncryptAndSign’ gives about a 2.0x boost in operations per second

NOTE: All the throughput numbers in this WP for other sections were measured using the default security settings. Hence there could be a further boost in performance if the security settings are modified.

With security turned off, the CPU alone is not the main bottleneck. The CPU is still used heavily, but 1GB network quickly becomes saturated as well:

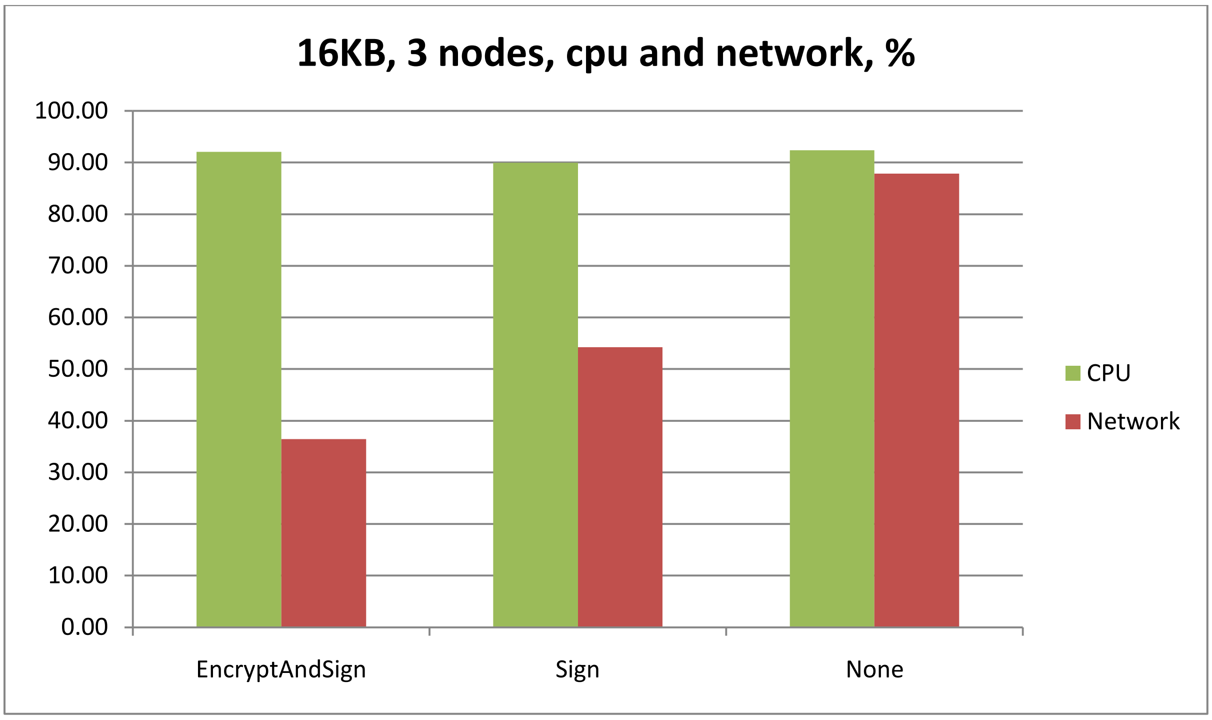

Figure 30: Dependency of CPU and network usage on cache servers from security settings for direct cache access (3 nodes, 16KB byte array objects, 90% reads & 10% writes)

With security turned off, the cache cluster is used most efficiently, because all server resources are used fully. It means that Windows Server AppFabric Cache provides the best possible performance in this configuration. Since the network is fully loaded, it will not be possible to get more throughput on the same hardware.

For larger cluster sizes or larger objects, other factors come into play and overall cluster resource usage is suboptimal. There can be several root causes for this behavior. Most probably, the performance will depend upon the workload. Refer to Appendix A for detailed performance numbers that were observed during this benchmarking.

ASP.NET session state performance

The goal of the tests in this category is to measure the performance of Windows Server AppFabric Cache as an ASP.NET session state provider. For this purpose, a simple ASP.NET website with shopping cart was implemented. The website was collocated with Windows Server AppFabric Cache and was installed on the same nodes. Hardware or software load balancing was not used in these tests. Instead, load balancing logic was implemented on the client side by sending requests to random web servers.

Tests were conducted for a buying scenario, which consisted of the following steps:

1. Go to the categories page, which displays two random categories.
2. Pick a category and view products for that category. The products page displays 5 random products.
3. Pick a product; visit the product buying page, where the purchased product is added to the session state.

4. After buying the product, visit the view cart page, where the session state is retrieved and products selected so far are displayed.

In total, session state is read once and written once during each scenario. However, each scenario consists of four requests, so each request on average contains a half operation with cache.

Set up

Tests in this category were performed with the following fixed parameters:

Parameter name	Value
Cached data size	Initially empty
Cluster size	12 nodes
Load pattern	Determined by test scenario above
Object type	Session state
Security	Default (SecurityMode=Transport, ProtectionLevel=EncryptAndSign)
Local cache	Not used

Variable parameters were the following:

Object size
2KB
16KB
128KB

Please, note that the object size above is not the size of the entire ASP.NET session state object. It is the size of the session state payload measured after serialization.

Results

In the case of Windows Server AppFabric Cache as an ASP.NET session state provider, larger objects still cause less throughput same as in the section related to testing performance of object sizes, but the difference is not too significant:

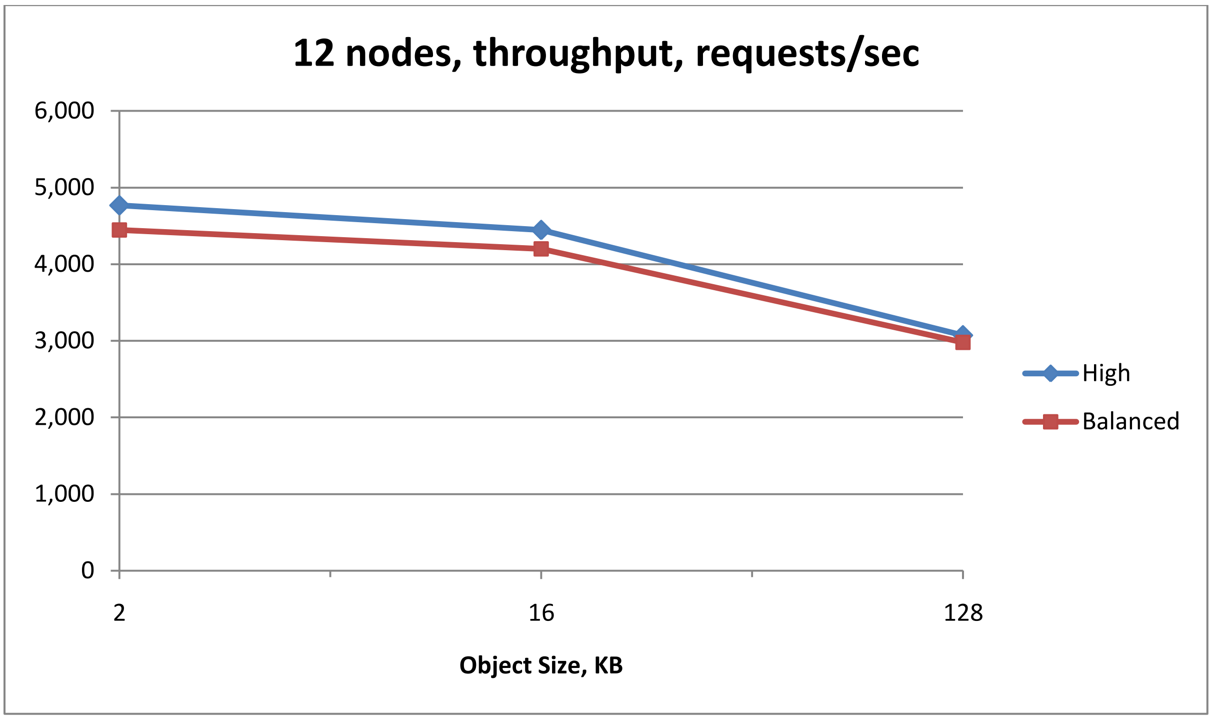

Figure 31: Dependency of throughput from object size for ASP.NET session state provider (12 nodes, default security)

The throughput numbers shown in the figure above are much less than the numbers in the case of direct access to the cache: 4,700 versus 39,000 for 2KB objects, 4,400 versus 21,000 for 16KB objects and 3,000 versus 4,000 for 128KB objects. The reason for this behavior is that in these tests there are many factors besides the cache that influence overall website performance. These factors include object serialization, ASP.NET performance, time to transfer pages to the client, etc. The fact that four requests to an ASP.NET website contain only two cache operations with non-empty objects should also be taken into account.

The bottleneck in the case of an ASP.NET web site is clearly in the CPU, because the ASP.NET application is collocated with Windows Server AppFabric Cache:

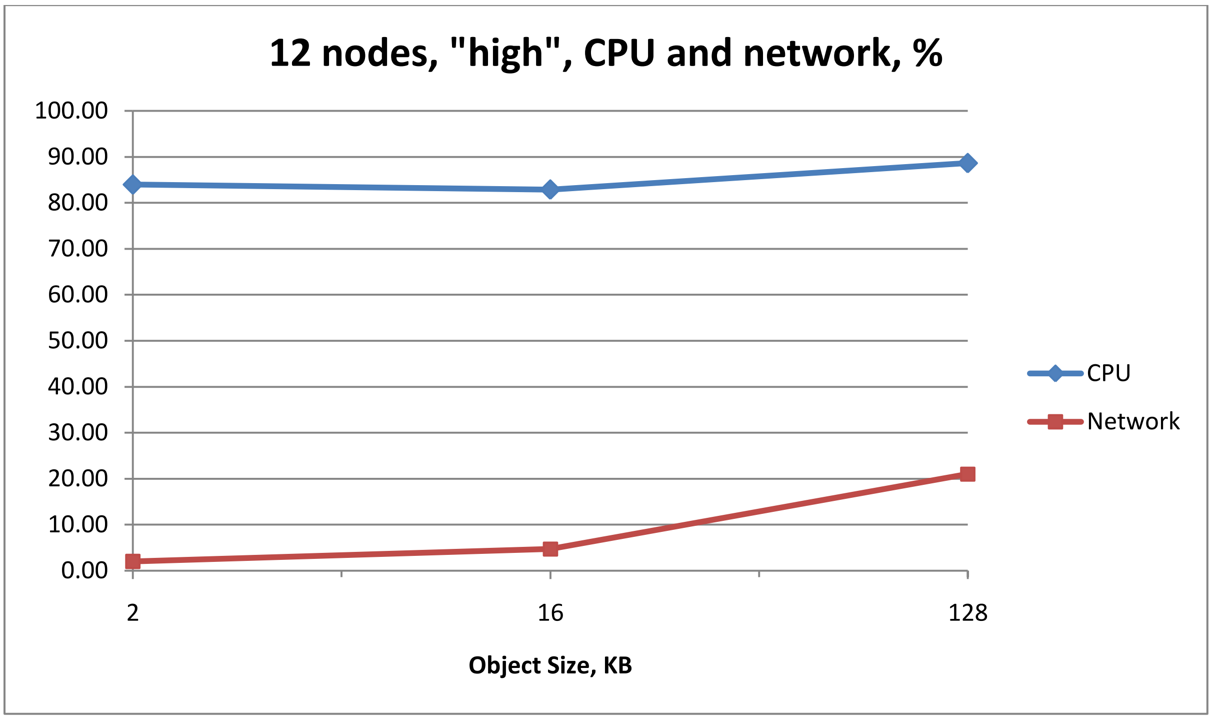

Figure 32: Dependency of CPU and network usage on web farm servers from object size for ASP.NET session state provider (12 nodes, default security)

It is also worth noting that the cache itself does not use a lot of CPU. Most of the time is spent in the ASP.NET host process:

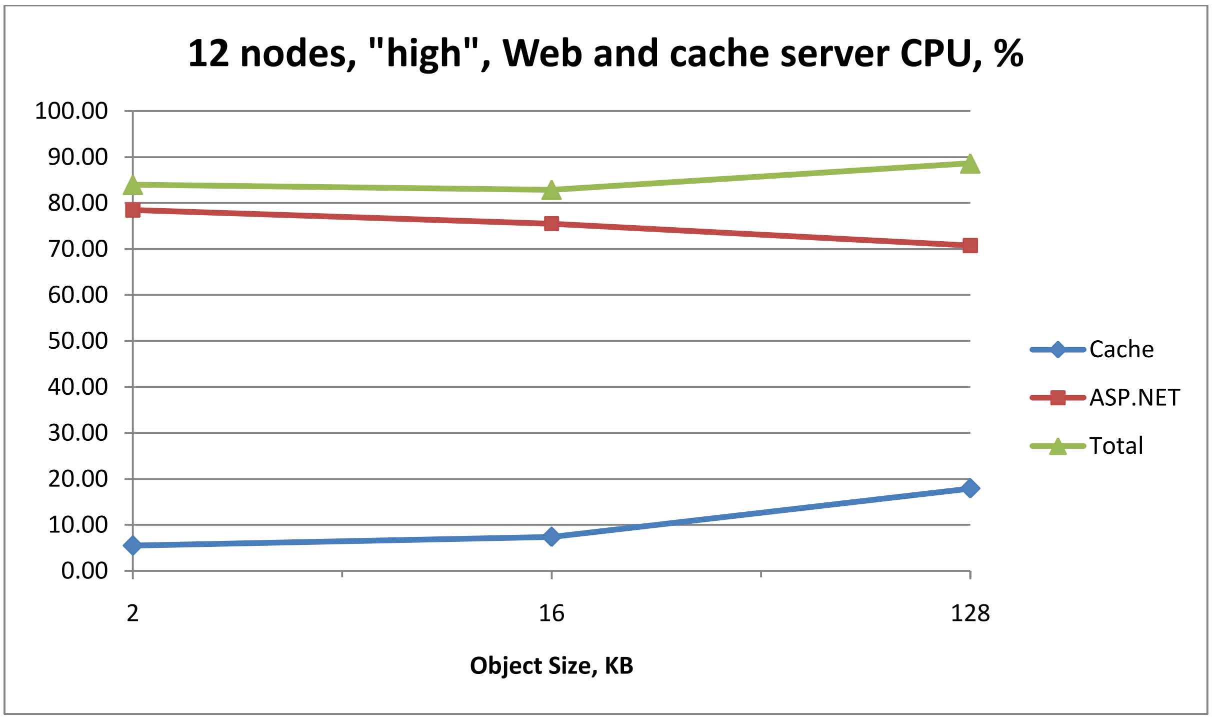

Figure 33: Dependency of CPU usage on web farm servers from object size for ASP.NET session state provider (12 nodes, default security)

WCF proxy performance

Most of the previous tests were performed against the Windows Server AppFabric Cache client API directly. However, in some cases, the cache client is wrapped and accessed via a proxy with additional application or domain logic. Oftentimes, performance of such applications is much different from the Windows Server AppFabric Cache cluster itself. The goal of the tests in this category is to show performance of a middle tier application with additional logic and compare it with performance of direct access to the cache.

To accomplish the goal, a simple WCF application was implemented that provided access to the cache and contained additional logic of populating the cache from an external data source if the requested object is not yet in the cache. The WCF service has two methods:

1. The "Get" method accepts a key to read data from the cache. It first checks if there is an object with the provided key in the cache. If yes, the object is returned to the client right away. If not, it sleeps 50ms simulating reading the object from a database, puts it in the cache and returns it to the client.
2. The "Put" method accepts a key and an object. This method sleeps 50ms simulating writing the object to a database and puts the object to the cache.

The WCF service was deployed in the ASP.NET site installed on cache servers. All tests were performed using BasicHttpBinding.

Set up

Tests in this category were performed with the following fixed parameters:

Parameter name	Value
Cached data size	12GB
Load pattern	45% get existing, 45% get non-existing with cache population, 10% put
Object type	Complex type from the test case related to complex type impact
Object size	16KB
Security	Default (SecurityMode=Transport, ProtectionLevel=EncryptAndSign)
Local cache	Not used

Variable parameters were the following:

Cluster size
3 nodes
6 nodes
12 nodes

Results

Just as in the scalability tests, the WCF tests did not show any dependency of latency on the cache cluster size. Latency was approximately the following:

Point	Latency, ms
High	123
Balanced	70
Low	63

The WCF tests showed that Windows Server AppFabric Cache and WCF itself scale well from 3 to 12 nodes with a scalability index of about 90%:

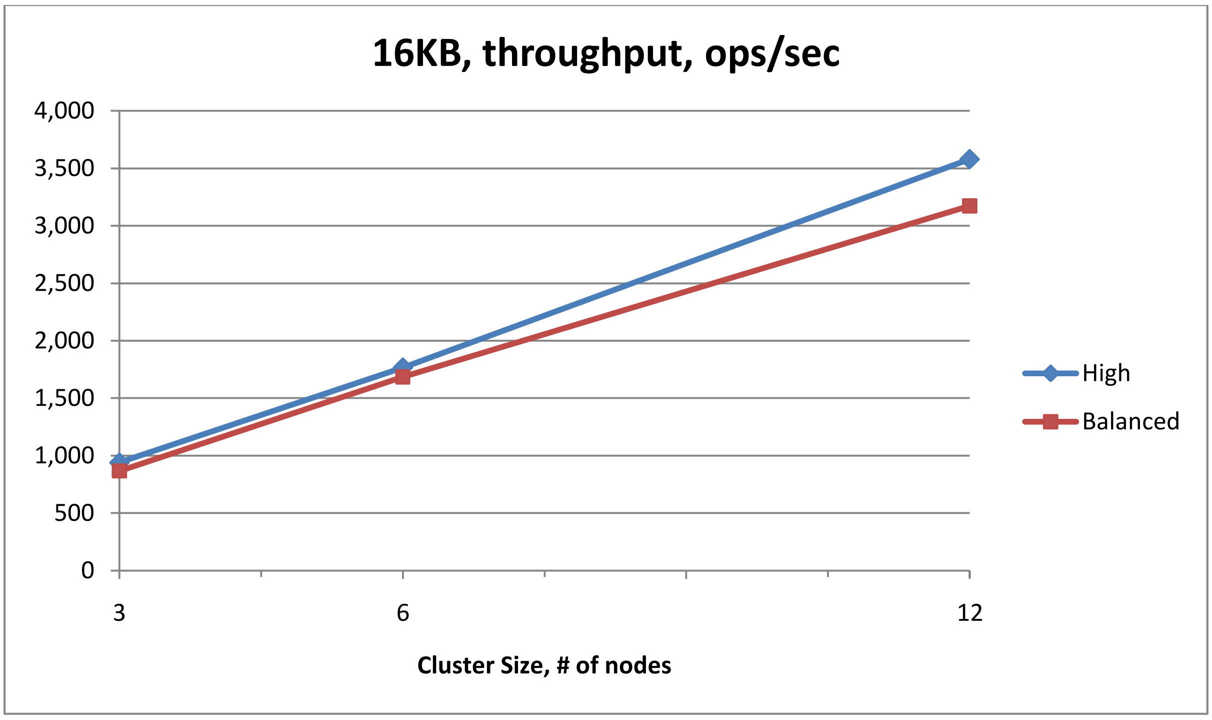

Figure 34: Dependency of throughput from cluster size for WCF proxy (16KB complex type objects, default security)

In all the tests, the bottleneck is in the CPU, not in the network:

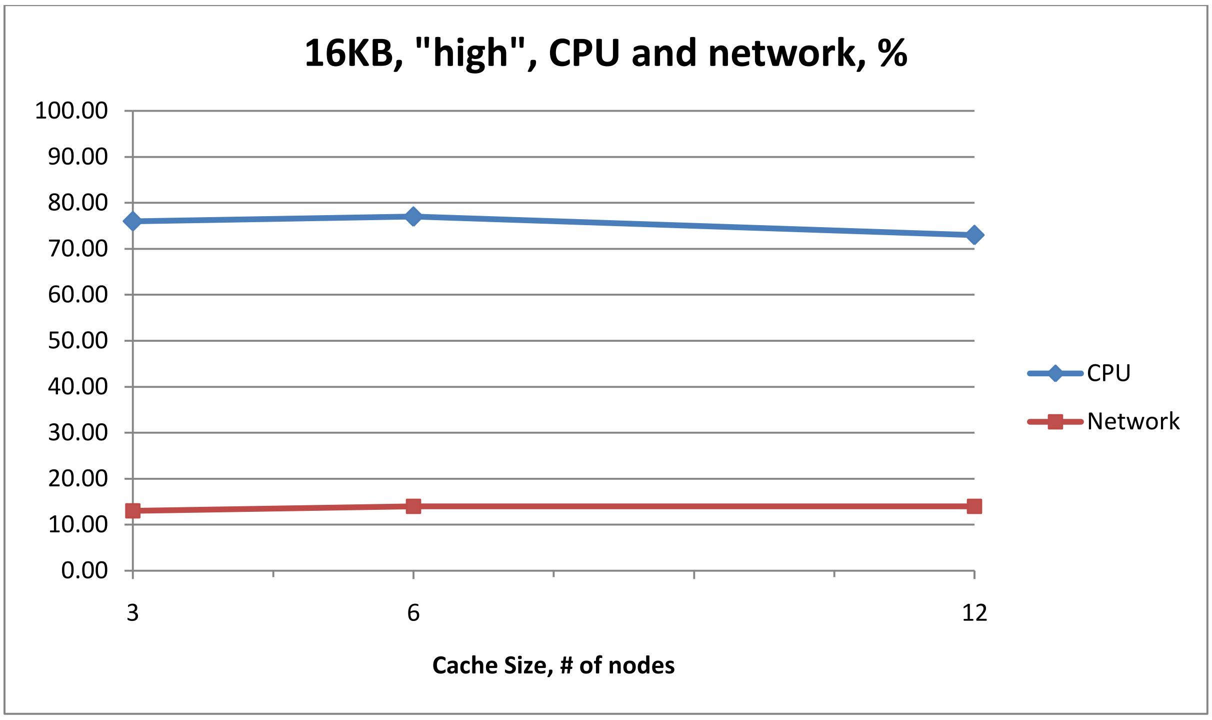

Figure 35: Dependency of CPU and network usage on WCF servers from cluster size for WCF proxy (16KB complex type objects, default security)

In the case of the WCF proxy tests, total throughput is much less than in the case of direct access to the cache. The reason is the same as in the ASP.NET session state provider tests: the CPU is mostly consumed by the WCF host process on object serialization and WCF internals. Windows Server AppFabric Cache use only about 15% of CPU.

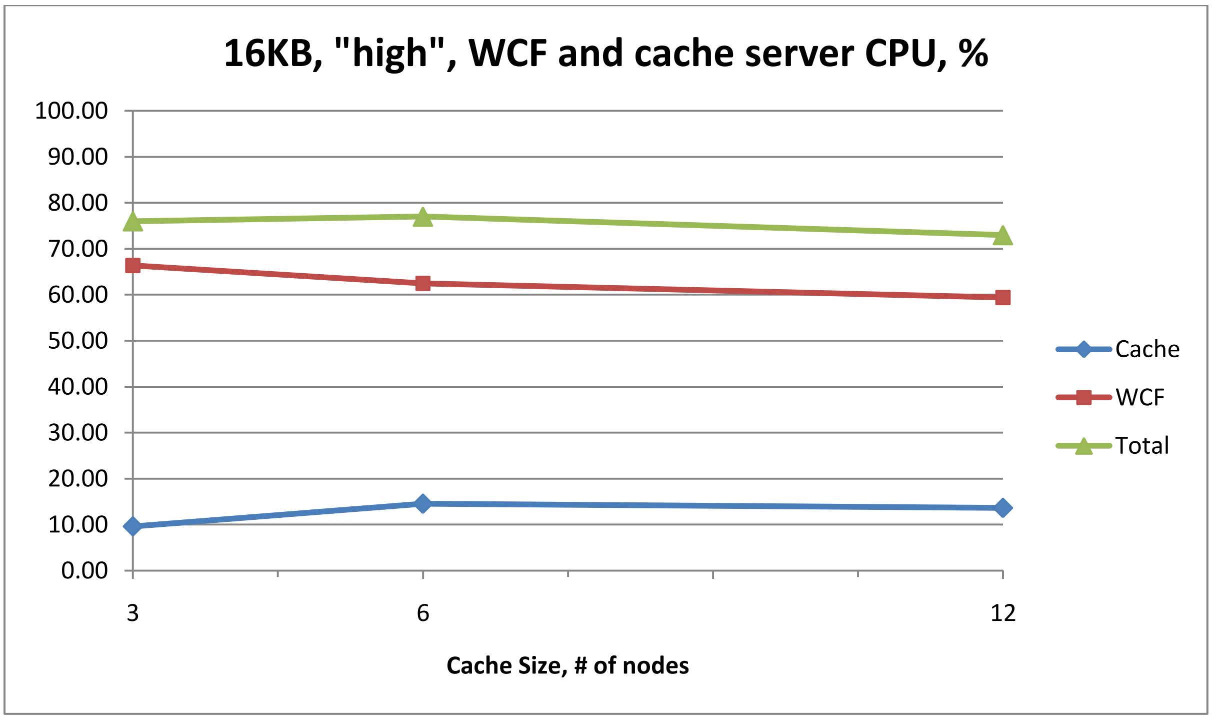

Figure 36: Dependency of CPU usage on WCF servers from cluster size for WCF proxy (16KB complex type objects, default security)

Conclusion and recommendations

During this benchmarking, Windows Server AppFabric Cache performance was studied in different configurations. Benchmarking results and analysis show how the performance of Windows Server AppFabric Cache depends on the environment and the usage patterns, and how it changes when certain configuration parameters change. The main observations and best practices that were found during this benchmarking are listed below:

- Cache size has an insignificant influence on performance when the number of read operations is much more than the number of write operations. When there are many writes, a larger cache size results in less throughput. However this can be scaled out by adding more servers
- The complexity of the types of objects stored in the cache impacts only the clients. More complex types means more time is spent in serialization and deserialization on the client side and this results in more clients needed to saturate the cache cluster. Consider using custom serialization to improve client side performance.
- Consider using larger objects and batching (BulkGet) to achieve better cache cluster resource utilization.
- Direct access to a cache is much more efficient than access via custom written proxies, especially when taking into account that custom proxies should also be scalable. To avoid overhead on application layer proxies to Windows Server AppFabric Cache cluster, avoid implementing custom proxies as a separate physical tier, especially if the application layer is very tiny. In case it can't be avoided, consider custom serialization or consider implementing these proxies as a WCF forwarding service. In most cases, however, direct access should be sufficient, because the Windows Server AppFabric Cache cluster is exposed as a WCF service itself.
- Pessimistic locking operations in Windows Server AppFabric Cache are fast and have virtually no overhead. They can be used whenever appropriate. Although AppFabric Cache pessimistic locking itself is fast, it is recommended to carefully analyze its performance for specific use case. Some applications may benefit from optimistic locking; some will work best with pessimistic locking.
- To achieve the best possible performance, consider setting cache cluster WCF security to 'None' (but only when appropriate) when accessing Windows Server AppFabric Cache cluster. Of course, this decision is dependent on a proper threat analysis and after consideration of the type of data stored in the cache.
- To avoid a bottleneck in the network, it is recommended to use a dedicated network between the application servers and the cache cluster servers. This recommendation should be especially helpful if security is turned off.

For the actual performance numbers for capacity planning see Appendix A.

Appendix A: Detailed performance numbers

Cache data size impact

Cache cluster size: 12 servers

Object size: 16KB

Object type: byte array

Security: default (EncryptAndSign)

Load pattern: 90% reads / 10% writes

Result type	Cache data size, GB	Latency, ms	Throughput, ops/sec	Cache CPU, %	Agent CPU, %	Network, %
High	0.125	5.66	25,612	64.96	63.35	28.56
High	3	6.02	24,807	64.50	61.77	27.70
High	12	7.2	24,281	61.81	60.57	27.13
High	24	7.66	24,341	64.59	61.23	27.22
High	48	5.58	24,657	58.62	60.67	27.53
Balanced	0.125	3.08	19,008	43.71	42.59	21.27
Balanced	3	3.72	21,257	51.63	50.16	23.81
Balanced	12	3.14	17,721	40.49	39.89	19.89
Balanced	24	3.84	20,583	51.15	48.43	23.04
Balanced	48	3.24	20,124	44.54	46.16	22.56
Low	0.125	2.02	7,010	13.20	12.47	7.87
Low	3	2.19	9,086	18.00	17.00	10.00
Low	12	2.23	7,789	15.00	17.00	9.00
Low	24	2.38	11,996	25.00	24.00	13.00
Low	48	2.34	11,786	24.00	23.38	13.18

Load pattern: 50% reads / 50% writes

Result type	Cache data size, GB	Latency, ms	Throughput, ops/sec	Cache CPU, %	Agent CPU, %	Network, %
High	0.125	6.48	27,488	70.53	68.61	30.60
High	3	7.3	22,910	63.22	55.90	25.61
High	12	6.34	20,489	55.08	49.15	22.91

High	24	7.86	17,875	51.75	43.38	20.01
High	48	7.9	18,573	53.06	44.55	20.82
Balanced	0.125	4.22	24,932	62.93	57.85	27.76
Balanced	3	4.22	19,868	52.88	44.06	22.19
Balanced	12	4.2	17,725	46.01	39.06	19.81
Balanced	24	4.1	15,513	42.34	33.31	17.37
Balanced	48	3.54	13,776	36.46	27.96	15.42
Low	0.125	2.78	15,674	36.88	31.94	17.51
Low	3	2.66	10,874	26.17	20.78	12.16
Low	12	2.5	10,748	25.00	22.00	12.00
Low	24	2.48	8,937	21.61	16.29	10.01
Low	48	2.51	9,318	23.00	17.00	11.00

Scalability

Cache data size: 12GB for 2 node cluster, 24GB otherwise
Object size: 16KB
Object type: byte array
Security: default (EncryptAndSign)

Load pattern: 90% reads / 10% writes

Result type	Cache cluster size, #	Latency, ms	Throughput, ops/sec	Cache CPU, %	Agent CPU, %	Network, %
High	2	7.96	5,072	93.25	10.02	34.14
High	3	7.54	7,506	90.03	21.88	33.66
High	6	5.28	12,414	71.20	27.02	27.84
High	9	7.36	18,518	67.71	44.65	27.67
High	12	7.66	24,341	64.59	61.23	27.22
Balanced	2	4.1	4,525	79.97	8.20	30.41
Balanced	3	4.32	6,925	84.00	13.00	31.00
Balanced	6	3.4	11,245	61.24	22.98	25.22
Balanced	9	4.48	17,456	61.75	40.31	26.09

Balanced	12	3.84	20,583	51.15	48.43	23.04
Low	2	2.4	3,617	60.00	6.00	24.00
Low	3	2.3	3,935	46.00	10.00	18.00
Low	6	2.43	7,968	39.00	16.86	19.83
Low	9	2.3	8,968	40.73	26.81	19.32
Low	12	2.38	11,996	25.00	24.00	13.00

Load pattern: 50% reads / 50% writes

Result type	Cache cluster size, #	Latency, ms	Throughput, ops/sec	Cache CPU, %	Agent CPU, %	Network, %
High	2	9.56	4,742	90.53	8.79	31.81
High	3	8.99	7,876	94.58	60.80	36.42
High	6	8.58	11,848	72.42	24.75	26.51
High	9	9.84	16,333	63.86	38.11	24.35
High	12	7.86	20,875	61.75	43.38	20.01
Balanced	2	4.01	4,094	79.00	7.71	27.00
Balanced	3	4.56	6,736	76.89	44.91	30.02
Balanced	6	3.64	9,833	58.51	18.35	22.01
Balanced	9	3.98	13,147	49.42	26.60	19.60
Balanced	12	4.1	16,913	42.34	33.31	17.37
Low	2	2.5	3,085	58.00	6.02	21.00
Low	3	2.3	4,211	53.00	0.05	19.00
Low	6	2.46	6,044	32.52	10.33	13.55
Low	9	2.31	7,745	25.00	14.00	12.00
Low	12	2.48	8,937	21.61	16.29	10.01

Object size impact

Cache data size: from 2Gb for 512B objects, 6GB for 2KB objects, 24GB for 4MB objects, 12GB for other objects
Object type: byte array
Security: default (EncryptAndSign)

Cache size: 3 nodes, load pattern: 90% reads / 10% writes

Result type	Object size, KB	Latency, ms	Throughput, ops/sec	Cache CPU, %	Agent CPU, %	Network, %
High	0.5	4.92	16,867	96.43	17.10	4.43
High	2	4.2	15,993	94.73	18.26	11.74
High	16	7.48	8,047	88.72	16.66	36.09
High	128	41.04	1,691	89.76	22.20	58.46
High	1024	219.88	226	91.16	20.24	62.30
High	4096	997.46	59	87.00	19.50	64.55
Balanced	0.5	2.52	16,079	91.75	15.30	4.20
Balanced	2	2.3	14,705	87.56	15.49	10.79
Balanced	16	3.42	7,098	72.66	13.50	31.87
Balanced	128	18.54	1,527	77.78	17.89	52.77
Balanced	1024	93.9	203	72.35	16.47	55.87
Balanced	4096	607.24	54	77.87	16.73	59.93
Low	0.5	1.26	11,080	56.42	8.53	2.85
Low	2	1.58	12,428	70.66	12.39	9.11
Low	16	2.5	5,040	50.00	9.00	23.00
Low	128	12.3	964	46.00	13.00	33.00
Low	1024	48.7	139	40.00	19.00	38.00
Low	4096	293.58	48	63.44	13.26	52.51

Cache size: 12 nodes, load pattern: 90% reads / 10% writes

Result type	Object size, KB	Latency, ms	Throughput, ops/sec	Cache CPU, %	Agent CPU, %	Network, %
High	0.5	2.6	51,747	70.78	57.71	3.38
High	2	3.06	45,714	68.75	58.85	8.38
High	16	7.1	26,349	66.22	65.65	29.39
High	128	91.98	3,448	63.04	66.25	37.30
High	1024	234.12	764	61.96	75.33	52.44
High	4096	679.4	197	64.34	68.76	54.11

Balanced	0.5	1.78	47,866	64.62	51.37	3.11
Balanced	2	2.18	42,544	62.97	53.13	7.80
Balanced	16	4.1	23,187	56.11	55.00	25.91
Balanced	128	86.88	2,858	34.84	46.40	30.92
Balanced	1024	143.52	688	51.66	67.87	47.16
Balanced	4096	394.14	178	53.65	61.12	48.77
Low	0.5	1.38	36,716	47.17	36.66	2.38
Low	2	1.58	28,586	38.93	32.51	5.24
Low	16	2.38	11,751	24.23	23.57	13.19
Low	128	13.5	1,772	14.00	23.00	15.00
Low	1024	85.45	448	28.00	40.00	31.00
Low	4096	302.42	157	42.00	52.00	43.00

Cache size: 12 nodes, load pattern: 50% reads / 50% writes

Result type	Object size, KB	Latency, ms	Throughput, ops/sec	Cache CPU, %	Agent CPU, %	Network, %
High	0.5	2.86	42,406	63.40	47.30	2.75
High	2	3.8	38,794	64.44	50.09	7.09
High	16	7.94	21,598	59.36	52.51	24.14
High	128	142.24	4,027	44.63	64.98	34.68
High	1024	355.24	803	70.83	74.81	54.98
High	4096	1020.66	205	76.22	68.46	55.98
Balanced	0.5	1.92	37,686	55.45	39.81	2.43
Balanced	2	2.14	31,170	49.97	37.16	5.70
Balanced	16	3.8	17,141	44.64	36.63	19.18
Balanced	128	85.86	3,466	38.62	54.98	29.97
Balanced	1024	164.66	651	54.00	58.00	45.00
Balanced	4096	573.72	163	60.00	51.00	45.00
Low	0.5	1.45	28,848	40.00	29.00	2.00

Low	2	1.63	24,594	37.00	27.00	4.00
Low	16	2.33	8,489	19.00	15.00	9.00
Low	128	15.83	1,639	16.00	18.00	14.00
Low	1024	99.64	380	28.86	31.15	26.06
Low	4096	408.1	111	40.38	31.97	30.49

Type complexity impact

Cache data size: from 2Gb for 512B objects, 6GB for 2KB objects, 12GB for other objects
Object type: complex custom class (see "Benchmarks/Type complexity impact/Set up" section)
Security: default (EncryptAndSign)

Cache size: 2 nodes, load pattern: 90% reads / 10% writes

Result type	Object size, KB	Latency, ms	Throughput, ops/sec	Cache CPU, %	Agent CPU, %	Network, %
High	0.5	6.54	12,120	97.69	16.91	6.38
High	2	7.62	10,465	97.06	19.44	11.55
High	16	11.14	5,219	93.82	27.54	35.10
High	128	83.52	1,265	91.82	65.41	65.53
Balanced	0.5	3.68	11,678	94.59	15.29	6.14
Balanced	2	3.04	9,841	91.00	16.00	11.00
Balanced	16	6.98	4,890	87.57	24.09	32.89
Balanced	128	35.18	1,011	67.26	39.39	52.40
Low	0.5	1.26	7,392	58.00	11.77	5.42
Low	2	1.62	6,117	54.00	13.99	9.92
Low	16	5.48	4,369	75.76	20.78	29.42
Low	128	26.72	692	41.00	26.00	36.00

Cache size: 3 nodes, load pattern: 90% reads / 10% writes

Result type	Object size, KB	Latency, ms	Throughput, ops/sec	Cache CPU, %	Agent CPU, %	Network, %
High	0.5	4.1	16,785	93.44	23.11	5.89
High	2	4.48	15,127	94.53	27.38	11.13

High	16	8.12	8,741	83.16	55.92	39.17
High	128	83.52	1,265	91.82	65.41	65.53
High	1024	491.26	162	46.77	72.16	44.40
Balanced	0.5	2.22	15,613	85.91	19.80	5.46
Balanced	2	2.78	14,279	88.60	24.14	10.49
Balanced	16	6.6	8,409	77.51	50.90	37.71
Balanced	128	35.18	1,011	67.26	39.39	52.40
Balanced	1024	276.4	144	39.01	52.46	39.68
Low	0.5	2.22	15,613	85.91	19.80	5.46
Low	2	1.72	14,403	78.00	18.74	5.00
Low	16	4.78	6,225	54.26	32.08	27.88
Low	128	26.72	692	41.00	26.00	36.00
Low	1024	211.81	119	28.00	37.00	33.00

Cache size: 3 nodes, load pattern: 50% reads / 50% writes

Result type	**Object size, KB**	**Latency, ms**	**Throughput, ops/sec**	**Cache CPU, %**	**Agent CPU, %**	**Network, %**
High	0.5	3.94	14,792	89.83	19.91	5.16
High	2	4.78	13,495	91.57	23.08	9.89
High	16	11.44	7,995	92.21	50.35	35.84
High	128	76.62	1,275	57.75	68.41	44.09
High	1024	468.6	175	58.52	67.95	47.80
Balanced	0.5	2.42	13,978	84.19	17.38	4.86
Balanced	2	2.76	12,392	84.57	19.56	9.09
Balanced	16	7.36	7,487	84.22	42.93	33.55
Balanced	128	51.62	1,220	54.33	59.50	42.15
Balanced	1024	274.46	155	47.84	51.83	42.40
Low	0.5	1.58	10,551	61.21	11.74	3.64
Low	2	2.16	10,927	73.14	16.46	8.01

Low	16	5.04	5,844	59.26	29.30	26.20
Low	128	36.73	1,048	44.00	44.00	36.00
Low	1024	201.8	127	34.00	34.00	35.00

Cache size: 4 nodes, load pattern: 90% reads / 10% writes

Result type	Object size, KB	Latency, ms	Throughput, ops/sec	Cache CPU, %	Agent CPU, %	Network, %
High	0.5	3.68	19,343	32.52	29.94	1.70
High	2	4.86	18,011	80.97	35.92	9.94
High	16	16.86	10,012	87.97	74.76	33.63
High	128	73.6	1,235	35.67	66.82	32.03
Balanced	0.5	2.2	17,309	25.62	23.96	1.51
Balanced	2	2.98	16,639	75.13	30.36	9.17
Balanced	16	8.4	8,999	76.12	61.47	30.23
Balanced	128	54.96	1,204	34.47	59.98	31.21
Low	0.5	1.49	12,021	18.00	17.00	0.92
Low	2	1.78	11,383	48.00	17.00	4.63
Low	16	5.01	5,975	43.00	31.00	20.00
Low	128	42.5	1,103	31.71	51.20	28.61

Cache size: 16KB objects, load pattern: 90% reads / 10% writes

Result type	Cluster size, #	Latency, ms	Throughput, ops/sec	Cache CPU, %	Agent CPU, %	Network, %
High	2	14.1	5,302	94.99	31.28	35.67
High	3	8.12	8,741	83.16	55.92	39.17
High	4	8.84	8,554	72.14	55.24	28.75
High	5	12.22	12,150	75.19	81.38	32.73
High	6	11.26	11,992	62.92	81.54	26.87
High	12	9.18	13,617	28.41	84.11	15.24
Balanced	2	7.84	4,986	89.48	27.29	33.54

Balanced	3	6.6	8,409	77.51	50.90	37.71
Balanced	4	6.36	7,974	63.36	46.60	26.80
Balanced	5	6.36	9,682	54.61	62.33	26.03
Balanced	6	5.58	8,557	39.24	50.71	19.18
Balanced	12	5	8,733	16.58	51.54	9.79
Low	2	6.16	4,503	78.56	23.35	30.28
Low	3	4.78	6,225	54.26	32.08	27.88
Low	4	4.4	5,095	35.21	24.61	17.14
Low	5	3.4	3,379	15.95	13.98	9.18
Low	6	4.2	5,824	24.89	27.93	13.06
Low	12	3.44	3,291	5.39	13.85	3.70

Read performance

Cache cluster size: 3GB for 3 nodes, 12GB for 12 nodes
Object type: byte array
Security: default (EncryptAndSign)

Get existing (cache hit)

Result type	Cluster size, #	Object size, KB	Latency, ms	Throughput, ops/sec	Cache CPU, %	Agent CPU, %	Network, %
High	3	0.5	4.18	18,063	92.10	18.61	4.74
High	3	16	4.7	7,439	72.80	14.48	33.38
High	12	0.5	2.28	51,809	68.05	59.69	3.38
High	12	16	5.34	22,848	53.68	56.58	25.51
Balanced	3	0.5	2.78	17,603	89.73	17.20	4.60
Balanced	3	16	4.7	7,439	72.80	14.48	33.38
Balanced	12	0.5	1.58	46,157	59.27	50.70	3.00
Balanced	12	16	4.04	22,153	50.52	53.29	24.75
Low	3	0.5	0.86	10,959	51.15	10.30	2.78
Low	3	16	4.7	7,439	72.80	14.48	33.38
Low	12	0.5	0.98	6,377	4.90	4.36	0.41

Low	12	16	3.18	19,623	42.65	45.45	21.97

Get non-existing (cache miss)

Result type	Cluster size, #	Object size, KB	Latency, ms	Throughput, ops/sec	Cache CPU, %	Agent CPU, %	Network, %
High	3	0.5	3.94	19,131	92.48	18.89	3.63
High	3	16	5.04	18,422	22.65	19.70	0.88
High	12	0.5	2.22	56,970	70.98	63.76	2.69
High	12	16	2.24	58,164	71.37	65.99	2.74
Balanced	3	0.5	3.1	18,893	91.21	18.02	3.58
Balanced	3	16	2.34	17,799	21.33	16.64	0.84
Balanced	12	0.5	1.32	41,500	48.72	42.29	1.94
Balanced	12	16	1.58	51,971	64.02	57.65	2.44
Low	3	0.5	2.4	18,306	88.40	16.93	3.46
Low	3	16	0.88	14,872	13.06	9.79	0.56
Low	12	0.5	0.82	6,676	4.14	3.22	0.30
Low	12	16	1.02	17,242	17.41	15.01	0.80

Bulk get 10

Result type	Cluster size, #	Object size, KB	Latency, ms	Throughput, ops/sec	Cache CPU, %	Agent CPU, %	Network, %
High	3	0.5	5.96	11,598	94.97	18.91	11.79
High	3	16	89.58	1,114	95.57	17.34	48.91
High	12	0.5	5.44	39,189	77.47	75.62	9.96
High	12	16	43.04	3,449	60.55	65.19	37.71
Balanced	3	0.5	3.02	9,702	75.50	15.30	9.84
Balanced	3	16	50.68	1,076	90.54	16.07	47.25
Balanced	12	0.5	2.96	32,607	61.24	62.41	8.28
Balanced	12	16	29.06	3,061	49.67	52.50	33.41

Low	3	0.5	1.38	6,769	47.12	10.09	6.82
Low	3	16	41.92	1,040	86.09	15.22	45.63
Low	12	0.5	1.92	15,630	24.30	24.74	3.97
Low	12	16	19.22	2,066	28.98	31.45	22.60

Bulk get 100

Result type	Cluster size, #	Object size, KB	Latency, ms	Throughput, ops/sec	Cache CPU, %	Agent CPU, %	Network, %
High	3	0.5	25.8	2,767	78.13	19.48	23.73
High	3	16	649.02	106	84.54	17.88	48.52
High	12	0.5	13.88	9,218	59.05	74.52	19.74
High	12	16	734.62	328	56.15	68.51	37.53
Balanced	3	0.5	14.9	2,641	73.41	17.69	22.67
Balanced	3	16	506.4	104	82.95	17.23	48.00
Balanced	12	0.5	11.16	8,839	54.75	69.43	18.93
Balanced	12	16	304.8	243	36.99	43.99	27.80
Low	3	0.5	12.78	2,463	68.92	16.86	21.09
Low	3	16	336.84	98	74.17	15.29	45.04
Low	12	0.5	8.52	7,551	43.32	55.25	16.17
Low	12	16	255.76	213	30.97	37.26	24.40

Locking performance

Cache cluster size: 12 nodes
Object size: 512B
Object type: byte array
Load pattern: 100% update with locks
Security: default (EncryptAndSign)

Result type	Cache size, # of objects	Lock type	Latency, ms	Throughput, ops/sec	Conflict rate, conflicts/sec	Cache CPU, %	Agent CPU, %	Network, %
High	1K	Optimistic	363	6259	8409	41.88	18.33	1.95

High	10K	Optimistic	149	16775	3310	63.32	43.74	2.63
High	100K	Optimistic	140	21936	540	69.59	59.07	2.94
High	1K	Pessimistic	308	6936	13434	52.35	15.24	1.62
High	10K	Pessimistic	130	17459	3663	68.17	39.67	2.52
High	100K	Pessimistic	136	20323	487	72.03	52.33	2.74
Balanced	1K	Optimistic	168	3434	1556	16.40	1.43	0.68
Balanced	10K	Optimistic	116	13664	1897	46.90	19.02	2.04
Balanced	100K	Optimistic	108	20190	388	61.04	40.87	2.69
Balanced	1K	Pessimistic	148	4160	2497	17.08	1.52	0.70
Balanced	10K	Pessimistic	110	15447	2477	52.69	24.08	2.20
Balanced	100K	Pessimistic	110	19555	387	68.02	41.89	2.64
Low	1K	Optimistic	119	1761	391	6.95	0.42	0.31
Low	10K	Optimistic	94	6301	410	17.34	2.81	0.88
Low	100K	Optimistic	103	19260	343	57.21	34.84	2.56
Low	1K	Pessimistic	106	1971	508	7.03	0.41	0.32
Low	10K	Pessimistic	88	5451	274	16.36	1.46	0.76
Low	100K	Pessimistic	97	17579	286	60.84	29.12	2.37

Security impact

Cache cluster size: 3GB
Object type: byte array
Load pattern: 90% reads / 10% writes

16KB objects

Result type	**Cluster size, #**	**Security**	**Latency, ms**	**Throughput, ops/sec**	**Cache CPU, %**	**Agent CPU, %**	**Network, %**
High	3	None	4.84	19,758	92.36	23.17	87.89
High	3	Sign	7.2	12,112	89.98	20.56	54.22
High	3	EncryptAndSign	11.08	8,107	92.07	17.04	36.43
High	6	None	3.74	27,869	68.36	34.75	61.71
High	6	Sign	4.56	19,784	73.34	35.51	44.21
High	6	EncryptAndSign	6.14	12,492	70.86	26.76	28.02

High	12	None	3.3	36,049	36.90	53.13	39.87
High	12	Sign	5.42	32,842	57.62	64.04	36.50
High	12	EncryptAndSign	9.2	24,672	64.81	61.88	27.58
Balanced	3	None	2.12	16,271	71.59	17.52	72.54
Balanced	3	Sign	4.4	11,661	86.39	18.92	52.12
Balanced	3	EncryptAndSign	6.54	7,781	88.72	15.54	34.97
Balanced	6	None	2.88	27,297	66.66	33.03	60.40
Balanced	6	Sign	3.84	19,554	71.87	34.24	43.70
Balanced	6	EncryptAndSign	5.08	12,345	69.23	25.92	27.69
Balanced	12	None	1.46	20,001	16.88	21.64	22.14
Balanced	12	Sign	3.14	25,680	42.82	47.67	28.53
Balanced	12	EncryptAndSign	4.88	22,611	57.62	53.54	25.26
Low	3	None	1.12	8,889	34.72	8.10	39.89
Low	3	Sign	3.2	10,574	78.68	16.51	47.26
Low	3	EncryptAndSign	4.04	6,816	77.33	12.82	30.69
Low	6	None	2.12	23,798	55.55	27.05	52.69
Low	6	Sign	2.94	18,294	65.39	30.64	40.85
Low	6	EncryptAndSign	3.96	11,335	61.33	22.91	25.49
Low	12	None	1.02	14,935	10.43	12.44	16.56
Low	12	Sign	1.4	9,553	12.56	11.68	11.09
Low	12	EncryptAndSign	2.9	15,298	34.83	32.60	17.18

128KB objects

Result type	Cluster size, #	Security	Latency, ms	Throughput, ops/sec	Cache CPU, %	Agent CPU, %	Network, %
High	6	None	117.06	2,407	13.31	24.59	42.02
High	6	Sign	99.02	2,636	25.78	32.63	45.91
High	6	EncryptAndSign	109.2	2,596	58.26	41.12	45.22
High	12	None	156.84	3,642	7.97	40.10	31.67
High	12	Sign	135.38	4,245	18.31	53.37	36.83
High	12	EncryptAndSign	170.52	3,378	29.63	60.77	29.31

Balanced	6	None	97.32	2,167	12.20	21.37	37.83
Balanced	6	Sign	91.64	2,496	24.42	30.65	43.52
Balanced	6	EncryptAndSign	61.22	1,934	37.16	28.85	33.65
Balanced	12	None	92.86	3,096	8.10	32.15	26.94
Balanced	12	Sign	70.78	2,809	11.93	34.14	24.50
Balanced	12	EncryptAndSign	78.66	2,590	21.13	42.09	22.63
Low	6	None	97.32	2,167	12.20	21.37	37.83
Low	6	Sign	91.64	2,496	24.42	30.65	43.52
Low	6	EncryptAndSign	61.22	1,934	37.16	28.85	33.65
Low	12	None	90.52	2,983	7.91	30.71	25.97
Low	12	Sign	70.78	2,809	11.93	34.14	24.50
Low	12	EncryptAndSign	78.66	2,590	21.13	42.09	22.63

ASP.NET session state performance

Cache cluster size: 12 nodes
Cache data size: initially empty
Load pattern: determined by test scenario
Object type: session state
Security: default (EncryptAndSign)

Result type	Object size, KB	Latency, ms	Throughput, ops/sec	Cache + ASP.NET CPU, %	Agent CPU, %	Network, %
High	2	350.27	4768	84	23	2
High	16	396.82	4446	83	23	5
High	128	1164.6	3070	89	15	21
Balanced	2	75.64	4448	78	20	1
Balanced	16	81.47	4199	78	20	4
Balanced	128	434.44	2979	86	13	20
Low	2	46.32	4243	73	19	1
Low	16	48.96	4084	75	19	4

Low	128	214.64	2909	83	13	20

WCF proxy performance

Cache data size: 12GB
Object size: 16KB
Load pattern: 45% get existing, 45% get non-existing with cache population, 10% put
Object type: complex
Security: default (EncryptAndSign)

Result type	**Cluster size, #**	**Latency, ms**	**Throughput, ops/sec**	**Cache + WCF CPU, %**	**Agent CPU, %**	**Network, %**
High	3	129.77	940	76	5	13
High	6	130.89	1767	77	11	14
High	12	110.15	3580	73	55	14
Balanced	3	66.92	867	71	4	12
Balanced	6	73.04	1686	70	9	13
Balanced	12	75.05	3174	62	43	13
Low	3	61.2	828	67	4	12
Low	6	63.1	1605	65	9	12
Low	12	64.8	3122	60	41	13

Appendix B: Reproducing performance tests

Configuring environment

The test harness requires the following to be configured before use:

1. Windows Server AppFabric Cache cluster, refer to the Installing and Configuring Windows Server AppFabric guide.
2. Visual Studio 2008 load generation cluster, refer to the Setting Up the Controller and Agent Functionality guide.
3. Internet Information Services with deployed WCF and ASP.NET applications (required only for running WCF or ASP.NET tests)

Before running tests you should specify Test Load Controller name and counter mapping:

1. Open Sources.VS9\Remote.testrunconfig and set controller name in Remote tab.
2. Open Sources.VS9\CacheTests\Load.loadtest, navigate to right-click on Run Settings and select Manage Counter Sets. Remove all computers, add your AppFabric servers and select Application group for each server.

The test harness uses custom performance counters that must be installed on all load agents before running tests. The easiest way to do this is to run the Project \ Sources.VS9 \ CacheTests \ Setup.loadtest test. After that the Test Load Controller service must be restarted.

Running tests

Start-LoadTests.ps1

Tests are run using a PowerShell script located in the Project\Scripts\Start-LoadTests.ps1. It accepts the following parameters:

Name	Type	Description
OperationMixes	string[]	An array of operation mixes to run.
CacheName	string	Cache name to use for tests.
CacheHostCounts	int[]	An array of host counts.
DataFactory	string	Data factory type to use. Test harness contains 3 factories: ByteArray, ComplexType and ModerateType.
DataSizes	int[]	An array of data sizes to use, in bytes. For complex and moderate types this parameter specifies the size of internal arrays.
DataCounts	int[]	An array of cache sizes to use, in objects.
RegionCount	Int	Region count to create.
RegionDataCount	Int	Number of objects in each region.
Prefix	string	Prefix that will be added to test names.

The script runs tests for all possible combinations of multi-value parameters. All parameters can be set in global context. In this case they can be omitted when invoking script.

Example:

```
$global:CacheName="test"
$global:DataFactory="ByteArray"
$global:Prefix="CacheSize"
$global:DataSizes=16384
$global:OperationMixes="90GetExisting10Update","50GetExisting50Update"
$global:CacheHostCounts=12
Start-LoadTests.ps1 -DataCounts 8192,16384,32768
```

OperationMixes.xml

The Project\Sources\CacheTests\OperationMixes.xml file defines operation mixes that can be specified in script parameters. Each mix defines a set of operations and their relative frequencies.

Example:

```
<?xml version="1.0" encoding="utf-8" ?>
<mixes>
  <mix name="90GetExisting10Update">
    <operation name="GetExisting" weight="90"/>
    <operation name="Update" weight="10"/>
  </mix>
</mixes>
```

The following operations are supported:

Name	**Description**
GetExisting	Invokes "Get" operation using existing key (cache hit).
GetNonexisting	Invokes "Get" operation using non-existing key (cache miss).
GetExistingFromRegion	Invokes "Get" operation using existing region and key.
GetNonexistingFromRegion	Invokes "Get" operation using non-existing region and key.
BulkGetExisting	Invokes "BulkGet" operation. Bulk size is configured in the test's app.config.
Update	Invokes "Put" operation.
OptimisticUpdate, PessimisticUpdate	Updates cached object using optimistic or pessimistic locking.
WcfGetExisting, WcfGetNonexisting, WcfUpdate	Invoke corresponding operations using WCF service.

About Grid Dynamics

Grid Dynamics is the global leader in scaling mission-critical systems. We help customers architect, design, and deliver business systems that handle peak loads, scale on demand, and always stay up. Using the latest advances in grid and cloud computing, our customers turn monolithic applications into scalable services and static, underutilized server clusters into virtualized compute clouds. The results: better performance, higher availability, faster time-to-market, and lower operational costs.

Grid Dynamics' .NET Scalability Practice offers solutions that enable customers to develop and scale applications with the Windows Server AppFabric Cache platform, Windows HPC Server, Windows Azure, and other Microsoft .NET technologies.

Grid Dynamics Consulting Services, Inc.
39300 Civic Center Drive, Suite 145
Fremont, CA 94538
Tel: +1 510-574-0872
Fax: +1 636-773-4528
http://www.griddynamics.com/