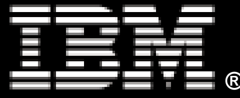


SOA Performance on Linux: Services and Reuse Entry Point

Tuning WebSphere Application Server for Web Services

Steve Dobbelstein
Khoa Huynh
Vivek Kashyap
Mark Peloquin

IBM Systems & Technology Group
June 2008

Contents

Abstract **3**
1. Introduction **4**
2. Performance Evaluation Methodology **4**
3. Systems Configurations **6**
3.1. Hardware 6
3.2. Software 7
4. Performance Results **8**
4.1. Results on the x86-based Web Server 8
4.2. Results on the POWER Web Server 19
5. Conclusions **28**
References **29**

Abstract

This paper is the first in a series of white papers on Service Oriented Architecture (SOA) performance in the Linux® environment. IBM has defined five SOA Foundation Entry Points to help a business get started with SOA. This paper focuses on the first entry point, which is the enablement of Web services and reuse; subsequent papers will evaluate the remaining entry points. More specifically, in this paper, we examine the impact of several parameters and features in WebSphere® Application Server, such as administrative security, heap size, garbage collection policy, thread pools, persistent HTTP connections, and performance monitors, as well as in Linux, such as the large page support, on the performance of Web services and how these parameters can be tuned for better performance. We considered two systems with different architectures to host the Web services: an IBM® System x™ 3850 M2 built on the eX4 chipset, which is the latest generation of IBM X-Architecture®, and an IBM Power™ 570 built on the POWER6™ processor technology. The paper shows the cumulative effect of tuning these parameters on improving the Web service request rates.

1. Introduction

IBM has defined five Service Oriented Architecture (SOA) Foundation Entry Points to help businesses get started with SOA in their enterprise environment. These five entry points are *People*, *Process*, *Information*, *Connectivity*, and *Reuse* [1]. This paper—the first in a series of white papers on SOA performance in the Linux environment—focuses on the *Reuse* entry point, which is the enablement of Web services and how they can be reused throughout the SOA implementation. Several articles on how to implement Web services for good performance are available. This paper looks at the performance issues and possible tunings that can be done in WebSphere and Linux that will help existing Web services run better in the Linux SOA environment.

2. Performance Evaluation Methodology

In this paper, we evaluate the performance of Web services implemented on the IBM WebSphere Application Server (WAS) v6.1. WAS is the IBM implementation of the Java® 2 Enterprise Edition (J2EE) platform, which conforms to V1.4 of the J2EE specifications [2]. WAS is the foundation for WebSphere Process Server (WPS), which is an SCA-compliant runtime element that provides a fully converged, standards-based process engine [3]. In our setup, we installed WPS v6.1.

We used a benchmark that models the Web services provided for a typical automobile insurance company [4]. This benchmark specifies a macro workload whose driver can generate an end-to-end workload similar to that of an actual production system in an SOA environment. It makes extensive use of IBM SOA platform products in the following areas:

- Enablement of Web services – using IBM WebSphere Application Server (WAS)
- Business process choreography – using integration and choreography features of IBM WebSphere Process Server (WPS)
- Integration of Web services – using IBM WebSphere Enterprise Service Bus (WESB) or DataPower appliances

Each of the areas mentioned above can be included or excluded from performance evaluations. In this paper, we only consider the first area: the enablement of Web services using WAS, which maps to the SOA *Reuse* Entry Point defined by IBM.

In our benchmark, the Web services are implemented as part of a *ClaimServices* application. These Web services represent the typical services that are involved in the processing of an automobile insurance claim, such as creating a claim, updating a claim, approving or denying a claim, checking insurance coverage, generating a list of approved repair shops, selecting a repair shop, and informing the customer. Some business logic is embedded in the implementation of these services. However, the presence of business logic might hinder us in evaluating the performance of the underlying middleware layers supporting Web services as well as investigating potential problems that might occur. As a result, we decided to keep the business logic in the Web services to a minimum; it only performs minimal calculations and returns responses. Although minimal, the amount of logic implemented in each Web service does vary depending on its type. For example:

- The Web service for creating a claim request (*create_request*) generates a unique Claim ID, puts the claim into the "accepted" state, formulates a fake payload verification response, and returns
- The Web service for getting a list of repair shops (*list_handlers*) builds a response message containing repair shop IDs in a given range, and returns
- The Web service for selecting a repair shop (*select_handler*) formulates a synthetic payload verification message and returns

For more details on the implementation of each Web service, see *SOA entry point: service creation and reuse* [4].

The benchmark's workload driver generates Web service requests to the benchmark's *ClaimServices* application running on WAS using the Service Oriented Access Protocol (SOAP) implemented on top of the HTTP transport protocol [9], as shown in *Figure 2.1*. The workload driver uses up to 50 concurrent

threads to generate as many service requests as the *ClaimServices* application can handle. We considered both x86 and IBM Power Architecture® platforms for the server hosting the Web services (i.e., running WPS v6.1 and the *ClaimServices* application). The workload driver runs on a separate x86 server (an IBM System x3650).

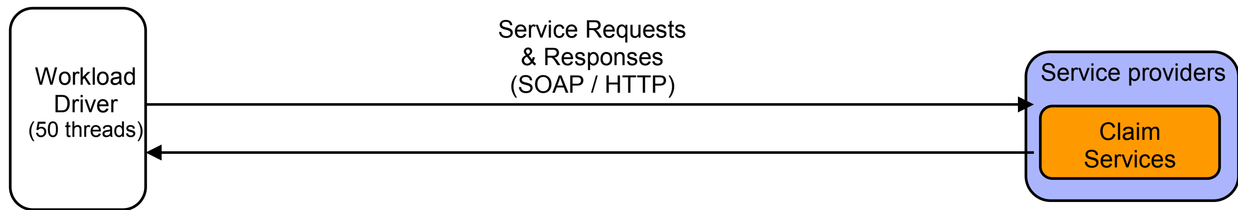


Figure 2.1 – Web Services

In our tests, we always started a warm-up run prior to actual data collection to ensure optimal and consistent results. Warm-up runs were especially needed because, by default, the IBM Java Virtual Machine (JVM) in WAS uses a higher optimization level for compiles, resulting in faster runtime performance, but at the expense of slower server startups.

We investigated the performance impact of many configuration parameters and settings for both WAS and the Linux operating system. However, in this paper, we identify only those parameters and tunings that result in a *discernable* performance improvement. *Section 4* presents these parameters and their *cumulative* performance impact, building up to the most optimal configuration that we believe is possible based on our testing results toward the end of *Section 4*. There were quite a few parameters that we thought would give us good performance benefits, but if those benefits were not reflected in our tests in any discernable way, they were excluded from our paper.

3. Systems Configurations

3.1. Hardware

As mentioned previously, we considered both x86 and IBM Power Architecture platforms for hosting Web services (i.e., hosting WPS v6.1 and the *ClaimServices* application).

3.1.1. x86 Architecture-Based Web Server

For the x86 platform, we used the IBM System x3850 M2 (Table 3.1), which implements the IBM eX4 chipset [7, 8].

Web Server	IBM System x3850 M2
CPU	4 x 64-bit Quad-Core Intel® Xeon® Processor X7350 (2.93 GHz)
Memory	64 GB (667 MHz DDR2)
Network	Integrated Dual-Port Gigabit Ethernet w/ TCP-IP off-load engine

Table 3.1 – x3850 M2 Configuration

The workload driver for the x86 test scenarios was an IBM System x3650 (Table 3.2) [8].
.

Workload Driver	IBM System x3650
CPU	2 x 64-bit Quad-Core Intel Xeon X5365 (3.0 GHz)
Memory	24 GB (667 MHz DDR2)
Network	Integrated Dual-port Gigabit Ethernet

Table 3.2 – x3650 Configuration

3.1.2. IBM Power Architecture-Based Web Server

For the Power platform, we used an IBM Power 570 (Table 3.3) [5] with POWER6 processors [6].

Web Server	IBM Power 570
CPU	2 x 64-bit Dual-Core IBM® POWER6® (4.7 GHz), 4 MB L2 cache per core, 32 MB L3 cache shared per two cores
Memory	32 GB (667 MHz DDR2)
Network	Dual-Port Gigabit Ethernet
Internal Storage	1 x SAS controller with 2 x 300 GB, 15K rpm SAS drives
Threading	Simultaneous Multi-Threading (SMT)™ Technology

Table 3.3 – Power 570 Configuration

The workload driver for the IBM Power test scenarios was an IBM System x3650 (Table 3.4).

Workload Driver	IBM System x3650
CPU	2 x 64-bit Quad-Core Intel® Xeon® X5460 (3.16 GHz)
Memory	24 GB (667 MHz DDR2)
Network	Integrated Dual-port Gigabit Ethernet

Table 3.4 – x3650 Configuration

All Web servers and workload drivers were connected to a Cisco Systems® Catalyst® 3750 Series Gigabit Switch (Model WS-C3750G-24TS-S).

3.2. Software

The Linux operating system on the x86 Web server (IBM System x3850 M2) was Novell SUSE Linux Enterprise Server (SLES) 10 Service Pack (SP) 1 for AMD64 & EM64T (x86_64).

The Linux operating system on the Power Web server (IBM Power 570) was Novell SUSE Linux Enterprise Server (SLES) 10 Service Pack (SP) 1 for PPC (ppc64).

Both Web servers ran WebSphere Process Server (WPS) v6.1.0 with the following fixes:

6.1.0.0-WS-WAS-IFPK61306, 6.1.0.0-WS-WBI-IFJR27892, 6.1.0.0-WS-WBI-IFJR27979, 6.1.0.0-WS-WBI-IFJR27983, 6.1.0.0-WS-WBI-IFJR27984, 6.1.0.0-WS-WBI-IFJR27985, 6.1.0.0-WS-WBI-IFJR27986, 6.1.0.0-WS-WBI-IFJR27987, 6.1.0.0-WS-WBI-IFJR27993, 6.1.0.0-WS-WBI-IFJR27994, 6.1.0.0-WS-WBI-IFJR28005, 6.1.0.0-WS-WBI-IFJR28008, 6.1.0.0-WS-WBI-IFJR28018, 6.1.0.0-WS-WBI-IFJR28019, 6.1.0.0-WS-WBI-IFJR28020, 6.1.0.0-WS-WBI-IFJR28034, 6.1.0.0-WS-WBI-IFJR28039, 6.1.0.0-WS-WBI-IFJR28042, 6.1.0.0-WS-WBI-IFJR28044, 6.1.0.0-WS-WBI-IFJR28045, 6.1.0.0-WS-WBI-IFJR28047, 6.1.0.0-WS-WBI-IFJR28048, 6.1.0.0-WS-WBI-IFJR28055, 6.1.0.0-WS-WBI-IFJR28056, 6.1.0.0-WS-WPS-IFJR27977, 6.1.0.0-WS-WPS-IFJR27981, 6.1.0.0-WS-WPS-IFJR27992, 6.1.0.0-WS-WPS-IFJR28023, and 6.1.0.0-WS-WPS-IFJR28041.

The operating system on both workload driver systems was Novell SUSE Linux Enterprise Server (SLES) 10 Service Pack (SP) 2 for AMD64 & EM64T (x86_64).

4. Performance Results

First we examine the performance data on the x86-based Web server.

4.1. Results on the x86-based Web Server

4.1.1. "Out of the Box" Configuration

The first step is to measure the performance using the default WebSphere Application Server settings. Figure 4.1.1 shows the relative throughput for each of the Web service scenarios on the x3850 M2. Relative throughput for a Web service scenario is calculated as the average request rate (number of requests per second) for that scenario divided by the request rate for the *create_request_bean* scenario in the "Out of the Box" configuration.

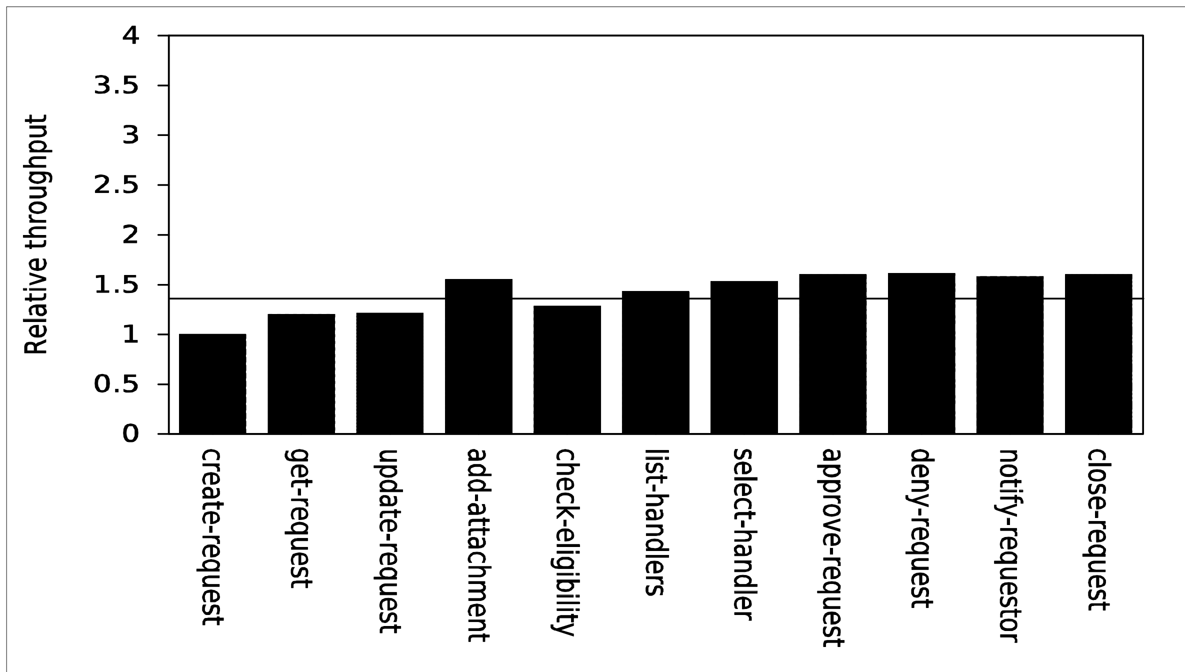


Figure 4.1.1 – Relative throughput for Web services on the x3850 M2 in "out of the box" configuration

The geometric mean of the relative throughput scores achieved on the x3850 M2 is 1.37.

4.1.2. Without WAS Administrative Security

The default profile for WebSphere Application Server has administrative security enabled. Enforcing security involves additional overhead for authentication and access validation. If you do not need to enforce administrative security, it is better to create a Web server profile with administrative security disabled.

Figure 4.1.2 shows the performance improvement when administrative security is disabled.

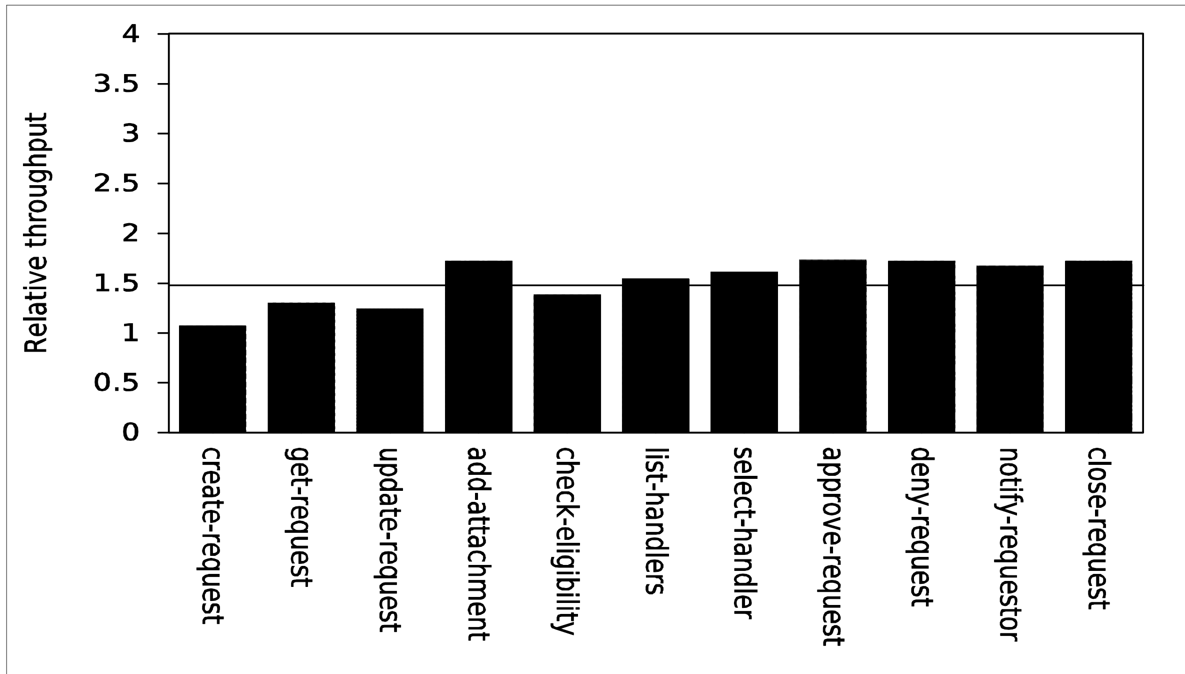


Figure 4.1.2 – Relative throughput for Web services on x3850 M2 without administrative security enabled

The geometric mean of the relative throughput scores is 1.49, an 8.63% improvement over the out-of-the-box configuration. As before, relative throughput for a Web service scenario is calculated as the average request rate (number of requests per second) for that scenario divided by the request rate for the *create_request_bean* scenario in the "Out of the Box" configuration.

4.1.3. WAS JVM Heap Tuning

In Java, the Java Virtual Machine (JVM) heap configuration has a significant impact on performance. There are several parameters available for tuning the JVM heap for better performance. Two basic JVM heap parameters are the size of the heap and the garbage collection policy. If the size of the heap is too small, it would cause garbage collection to happen more often. If the heap is too large, garbage collection would happen less often, but when it does, it would take longer to compact the larger heap. There are several tools available, such as the Tivoli® Performance Viewer (included with WebSphere), which can help analyze and monitor heap usage and garbage collection so that the heap can be tuned for a specific workload.

The default size of the JVM heap is typically too small. To change the heap size, we can use the WAS Administrative Console as follows:

- Go to Servers → Application Servers → *server name* → Server Infrastructure → Java and Process Management → Process Definition → Additional Properties → Java Virtual Machine
- Enter new sizes in the "Initial Heap Size "and "Maximum Heap Size" boxes (the size should be specified in MB)

In our tests, we found that increasing the heap size to 2048 MB gave a significant performance improvement. *Figure 4.1.3.1* shows the performance improvement gained from increasing the WAS JVM heap on the x3850 M2. The geometric mean of the relative throughput scores is 2.60, a 74.65% increase over the default heap size. (Relative throughput for a Web service scenario is calculated as the average request rate for that scenario divided by the request rate for the *create_request_bean* scenario in the "Out of the Box" configuration.)

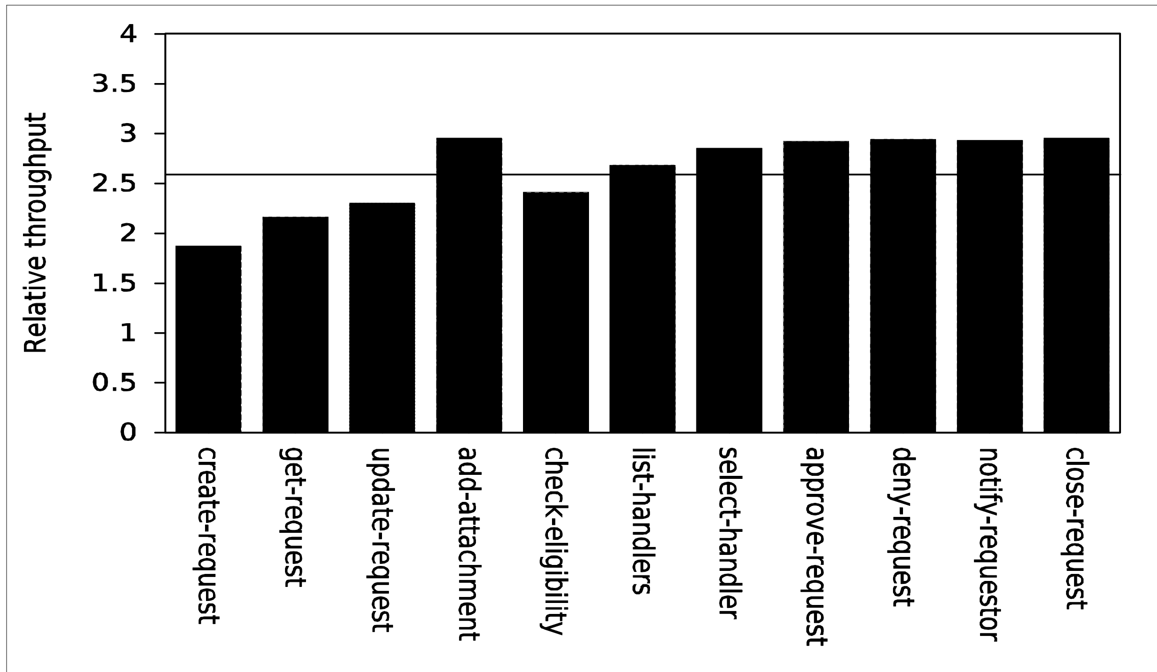


Figure 4.1.3.1– Relative throughput for Web services on the x3850 M2 with heap increased to 2 GB

In addition to the JVM heap size, the garbage collection policy can also affect performance. The default garbage collection policy is **`optthruput`**. The **`optthruput`** garbage collector scans all objects in the heap, marking any object that is in use, sweeps up the unused objects, reclaims their memory, and then compacts the remaining memory to reduce fragmentation. The entire process can take some time. All application threads are stopped while the garbage is collected. The **`optthruput`** policy provides high throughput, but with longer garbage collection pause times.

The WAS JVM also supports three other garbage collection policies. We considered the **`gencon`** (Generation Concurrent) policy which handles short-lived objects differently from objects that are long-lived. Under the **`gencon`** policy, the heap is split into new and old segments. Long-lived objects are promoted to the old segment while short-lived objects are garbage collected quickly in the new segment, which is also called the *nursery*.

We configured the **`gencon`** garbage collection policy and a 1536 MB nursery through the WAS Administration Console as follows:

- Go to Servers → Application Servers → *server name* → Server Infrastructure → Java and Process Management → Process Definition → Additional Properties → Java Virtual Machine
- Enter **`-Xgcpolicy:gencon -Xmn1536M`** into the “Generic JVM arguments” box

Figure 4.1.3.2 shows the performance improvement gained from switching to the **`gencon`** garbage collection policy. The geometric mean of the relative throughput scores is 2.77, which is a 6.72% improvement over the 2 GB heap configuration with the default garbage collection policy.

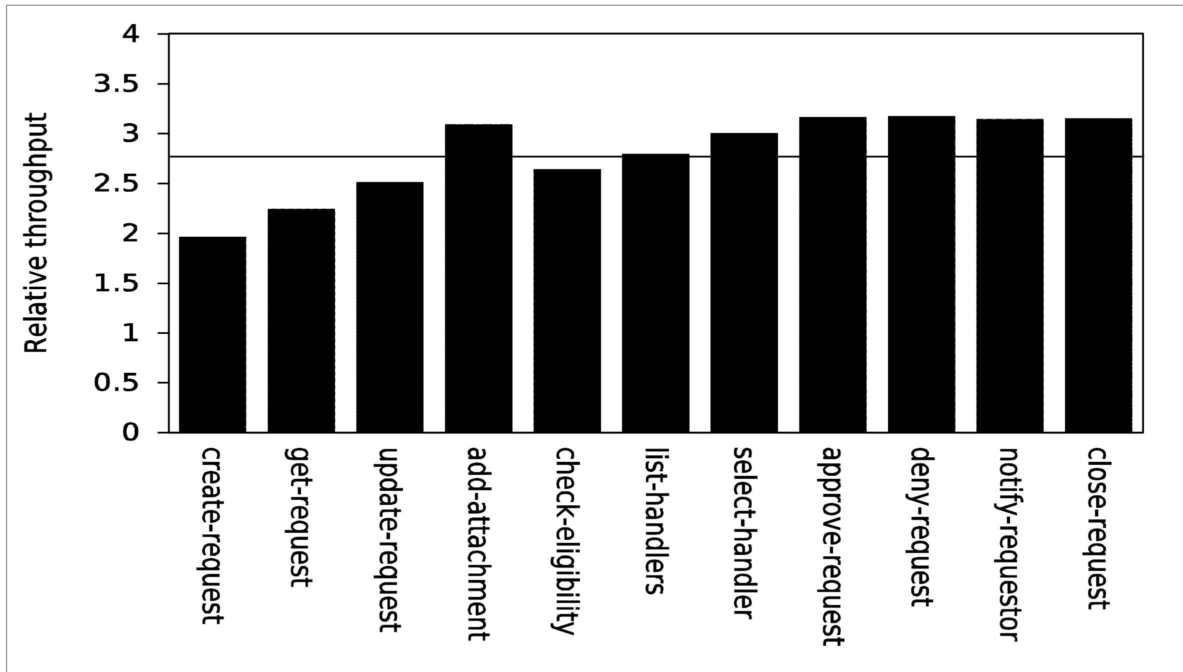


Figure 4.1.3.2 – Relative throughput for Web services on the x3850 M2 with **`gencon`** *GC policy*

4.1.4. Large Page Support in Linux

One of the factors to consider in system performance is memory access. Processors keep a cache of mappings from virtual addresses to physical addresses in a Translation Look-aside Buffer (TLB) so that they don't have to walk through the page tables for every virtual address used. The TLB holds a fixed number of entries. When the processor encounters a virtual address that is not in the TLB, called a TLB miss, it must walk the virtual address through the page tables to get its physical address. The new mapping is written to the TLB, overriding an existing entry.

Linux has support for large pages (sometimes also known as huge pages). On the x86 architecture, normal pages are 4 KB in size and large pages are 2 MB. The use of large pages can improve memory performance in two ways. First, since a single entry in the TLB maps to more memory (2 MB instead of 4 KB), fewer TLB entries are needed to map a larger area of memory. With fewer TLB entries, the occurrence of TLB misses is reduced. Second, the page table setup for mapping large pages uses one fewer table to determine the physical address, making virtual-to-physical-address mapping faster than for normal pages.

Large pages are configured in Linux by adding the following lines to the **`/etc/sysctl.conf`** file:

```
vm.nr_hugepages = <number of pages>
kernel.shmmax = <number of bytes>
kernel.shmall = <number of bytes>
```

In our study, we wanted to configure enough large pages for the WAS JVM heap. The JVM heap was 2 GB. To be on the safe side, we configured 3 GB worth of large pages (3 GB / 2 MB per large page = 1536 large pages). We set the amount of shared memory to 4 GB so that there would be enough room for the 3 GB of large pages in the shared memory pool (4 GB is 4294967296 bytes). We added the following lines to the **`/etc/sysctl.conf`** file:

```
vm.nr_hugepages = 1536
kernel.shmmax = 4294967296
kernel.shmall = 4294967296
```

The WAS JVM was then configured to use large pages by adding the parameter **`-Xlp`** in the same "Generic JVM arguments" box where the garbage collector parameters were set.

Figure 4.1.4.1 shows the performance of the Web services with the Java heap using large pages.

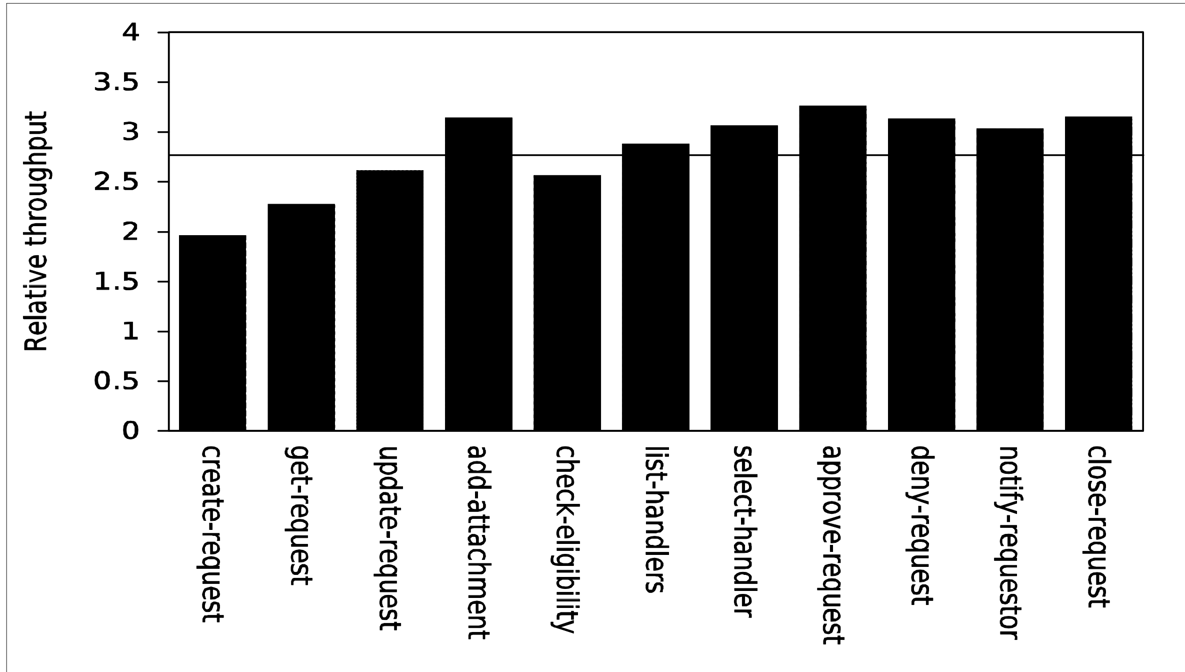


Figure 4.1.4.1 – Relative throughput for Web services on the x3850 M2 with added large-page support

The geometric mean of the relative throughput scores is 2.78, which is only a 0.36% improvement over the previous configuration. (Relative throughput for a Web service scenario is calculated as the average request rate for that scenario divided by the request rate for the *create_request_bean* scenario in the "Out of the Box" configuration.)

However, we had anticipated a larger improvement. At first look, it appears that large pages don't help in a Web service workload. Realizing that, at this point, we had already applied some heap tunings (increased to 2 GB, used **`gencon`** garbage collector), we hypothesized that any improvement to be gained from large pages was probably lost in the gains obtained from the previous tunings. We took a couple steps back and added only large page support to the default heap configuration. The results are shown in *Figure 4.1.4.2*.

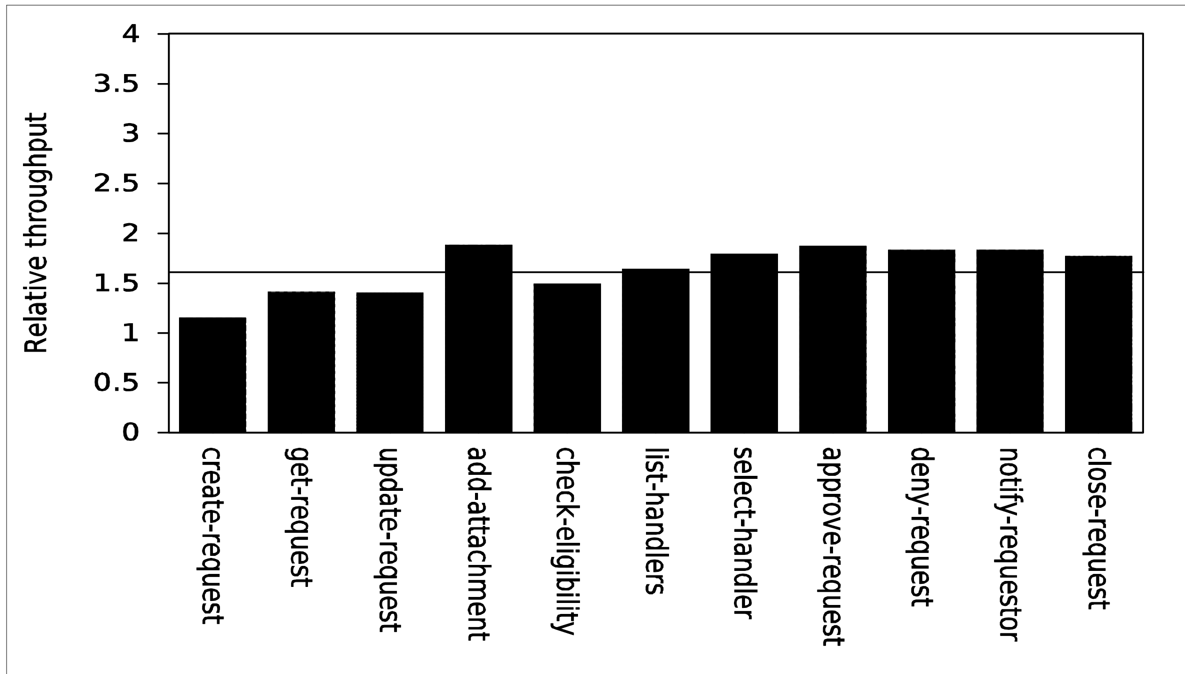


Figure 4.1.4.2 – Relative throughput for Web services on the x3850 M2 using only large pages

The geometric mean of the scores is 1.61, an 8.1% increase over the score of 1.49 for the default heap configuration results shown in Figure 4.1.2. Large pages do make a difference in performance.

We saw previously that increasing the heap size significantly improved the performance of the Web services, demonstrating that a 1 GB heap is too small for this workload. Perhaps using large pages on a bigger heap might give us further gains in performance. Figure 4.1.4.3 shows the performance results using large pages on a 2 GB heap.

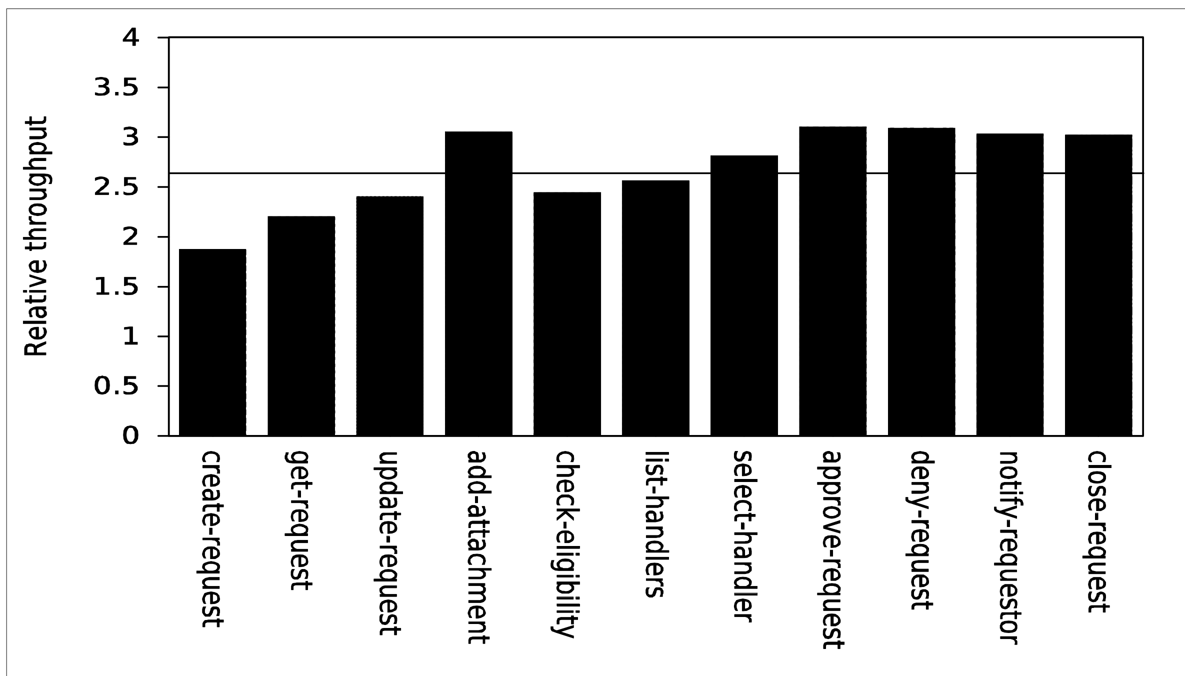


Figure 4.1.4.3 – Relative throughput for Web services on the x3850 M2 using 2 GB heap and large pages

The geometric mean of the scores is 2.64, which is only a 1.78% improvement over the score of 2.60 for only configuring a 2 GB heap shown in Figure 4.1.3.1. For this particular workload, it appears that large pages only help when the Java heap is constrained to a size that cannot accommodate all objects. If the heap can be made large enough, then large pages would not add much of a performance benefit.

4.1.5. WAS Thread Pool Settings

Thread pools are another WebSphere resource that can affect performance. For our Web service scenarios, the two thread pools that come into play are the default thread pool and the Web container thread pool.

We changed the number of threads in a pool through the WAS Administration Console as follows:

- Go to Servers → Application Servers → *server name* → Additional Properties → Thread Pools
- Click on the thread pool you want to change and enter new values in the "Minimum Size" and "Maximum Size" boxes

The default maximum number of threads for the default thread pool is 20. The default maximum number of Web container threads is 50. We increased the maximum number of default threads to 200 and the maximum number of container threads to 100. The results are shown in *Figure 4.1.5*.

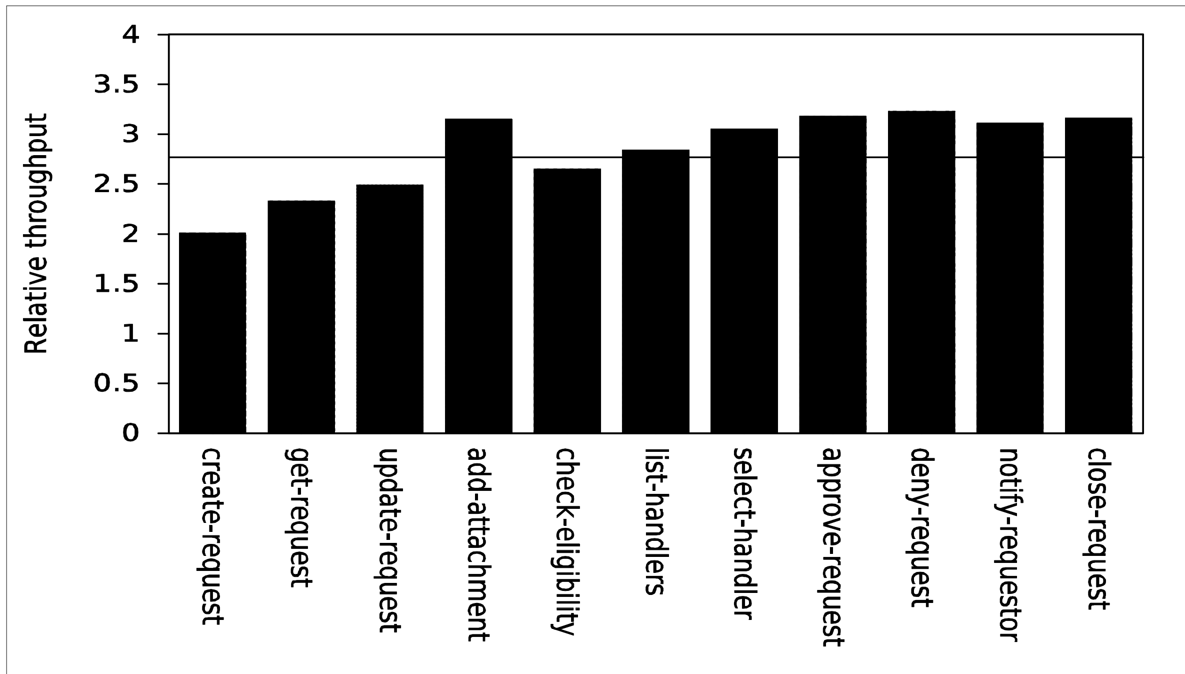


Figure 4.1.5 – Relative throughput for Web services on the x3850 M2 with increased thread pools

The geometric mean of the relative throughput scores (relative throughput for a Web service scenario is calculated as the average request rate for that scenario divided by the request rate for the *create_request_bean* scenario in the "Out of the Box" configuration) is 2.77, which is a negligible decrease (-0.15%) from the score of 2.78 for the JVM heap tunings in Figure 5.1.4.1. Therefore, thread pools were not a significant factor in the performance of this workload on the x86 platform. However, we will see later that the thread pool settings do affect performance on the Power platform.

4.1.6. WAS HTTP Connection Settings

Another factor that can affect the performance of WebSphere is the number of concurrent HTTP connections available. If a sufficient number of concurrent HTTP connections are not available, incoming

clients will not be able to connect until a connection is freed. If the server's CPUs are not fully utilized, and there is memory to spare, the number of HTTP connections can be increased to improve the server's performance.

To change the number of HTTP connections available for a given port through the WAS Administration Console:

- Go to Servers → Application Servers → *server name* → Communications, and click on the Ports link
- Find the port number in the table and click on "View associated transports" for that port
- Click on the transport chain that is listed. Click on "HTTP inbound channel (HTTP_n)," where "n" denotes channels 1 through 4
- Either click on "Maximum persistent requests per connection" and enter a number in the "Specify maximum number of persistent requests" box, or click on "Unlimited persistent requests per connection"

The default number of HTTP connections for each port is 100. Since our Web server has more than enough CPU power and memory, we set the number of HTTP connections to `unlimited`. The results are shown in *Figure 4.1.6*.

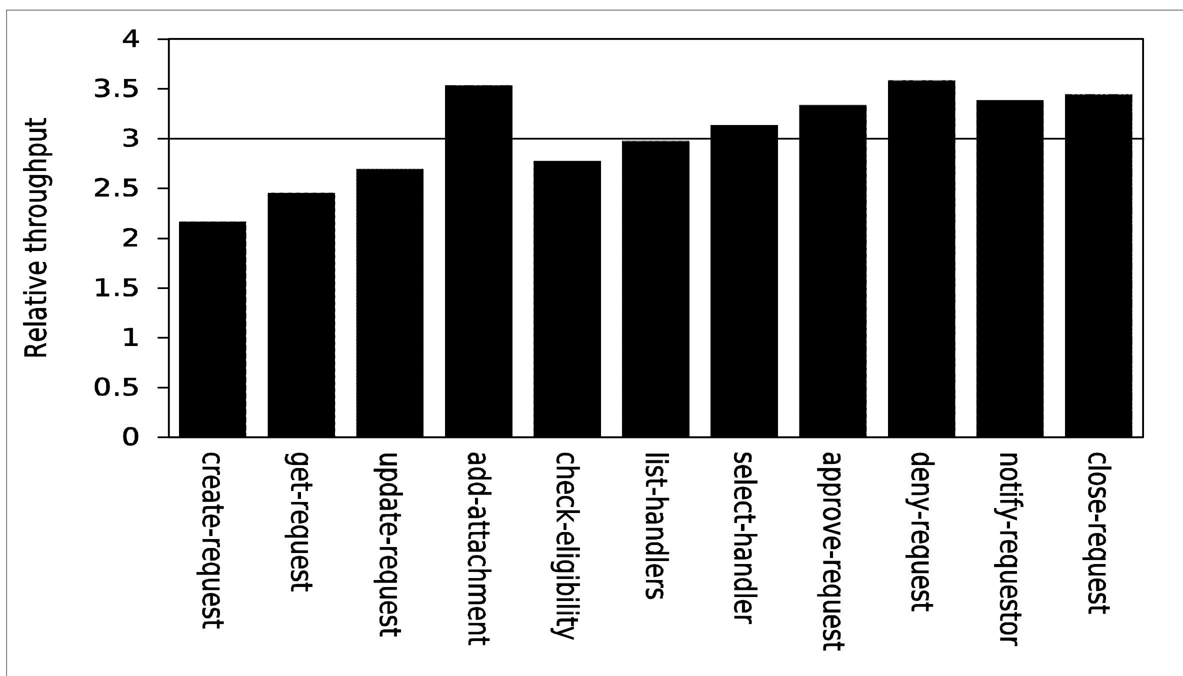


Figure 4.1.6 – Relative throughput for Web services on the x3850 M2 with unlimited HTTP connections

The geometric mean of the relative throughput scores (relative throughput for a Web service scenario is calculated as the average request rate for that scenario divided by the request rate for the *create_request_bean* scenario in the "Out of the Box" configuration) is 3.01, an 8.34% increase over the last configuration. The previous configuration was being limited by the number of HTTP connections. Removing that limit improved the performance of the benchmark.

4.1.7. Tracing and Logging

WebSphere has facilities for tracing and logging that are useful for debugging problems, but at a cost to performance. These facilities are not needed in a production environment.

To disable tracing, on the WAS Administrative Console:

- Go to Troubleshooting → Logs and Trace → *server name* → Change Log Detail Levels

- Set both the Configuration and Runtime to **`*=all=disabled`**

To change the PMI level:

- Go to Monitoring and Tuning → Performance Monitoring Infrastructure (PMI) → *server name*
- Uncheck the “Enable Performance Monitoring Infrastructure (PMI)” box and, in the “Currently Monitored Statistics Set” box, select “None”

Figure 4.1.7 shows the throughput results with the logging and tracing turned off.

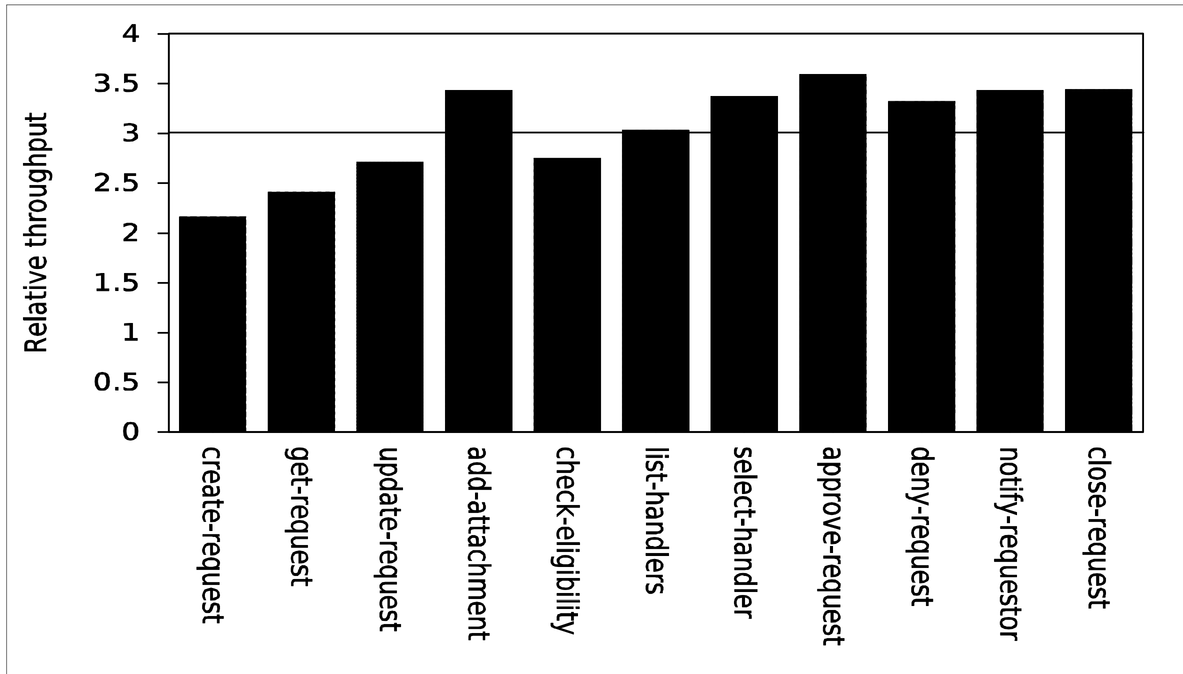


Figure 4.1.7– Relative throughput for Web services on the x3850 M2 with logging and tracing turned off

The geometric mean of the relative throughput scores (relative throughput for a Web service scenario is calculated as the average request rate for that scenario divided by the request rate for the *create_request_bean* scenario in the “Out of the Box” configuration) is 3.02, a negligible 0.37% increase over the previous configuration. Disabling logging and tracing didn't add any performance benefit to the previous configuration. At this point, we had already done a lot of tuning. Any benefit from disabling logging and tracing may have been lost in the optimizations made to that point. It is possible that with other configurations, disabling logging and tracing may help.

4.1.8. Summary

Figure 4.1.8.1 shows a summary of the various geometric mean scores for the configurations presented. The biggest gain was from increasing the heap size. Disabling security, using the **`gencon`** garbage collection policy, and using unlimited HTTP connections each had a notable impact on performance. Large pages, increasing the thread pools, and disabling logging and tracing did not have noticeable impacts on performance. Overall, the cumulative performance gain for all the tunings was 120% over the “out of the box” configuration.

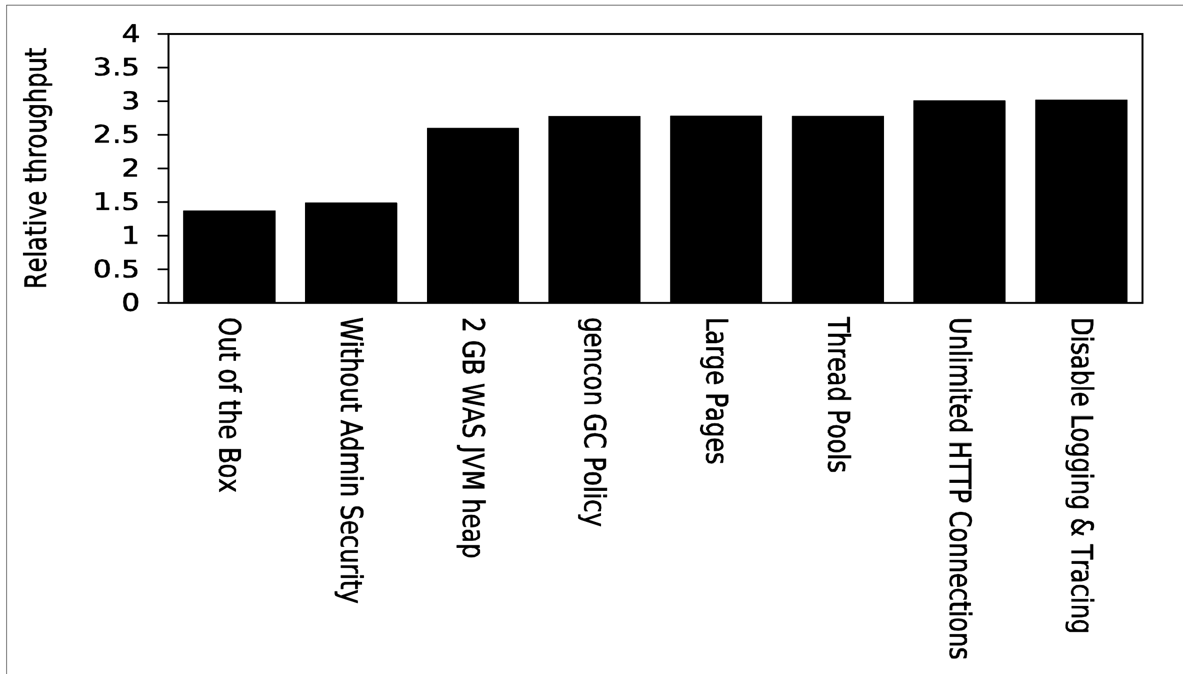


Figure 4.1.8.1 – Summary of throughput for each configuration

Finally, we need to determine if there is any performance bottleneck in the most optimal configuration in our test setup. Figure 4.1.8.2 shows the average CPU utilization on the Web services host. Although it varied a little during different Web service invocations, the CPU utilization was never higher than 60%, and, in fact, it remained at around 30% to 40% most of the time. As Figure 4.1.8.3 shows, the average CPU utilization on the workload driver system was higher, going over 90% occasionally, and averaging at around 80% most of the time.

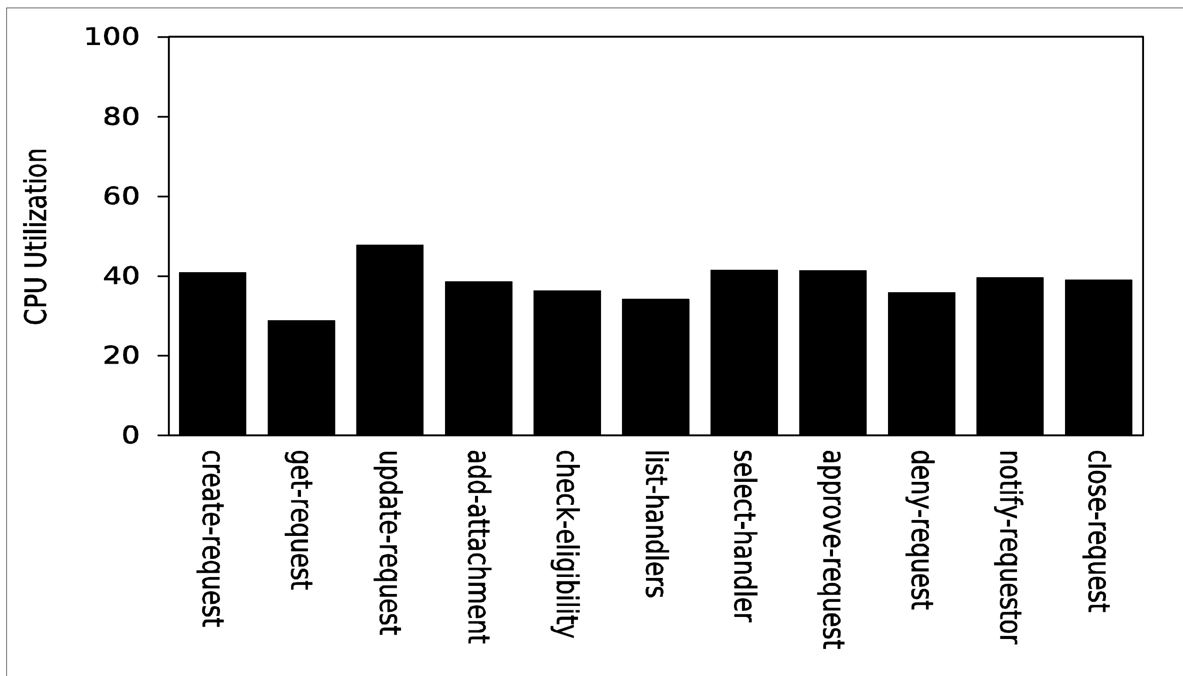


Figure 4.1.8.2 – Average CPU utilization on the x3850 M2

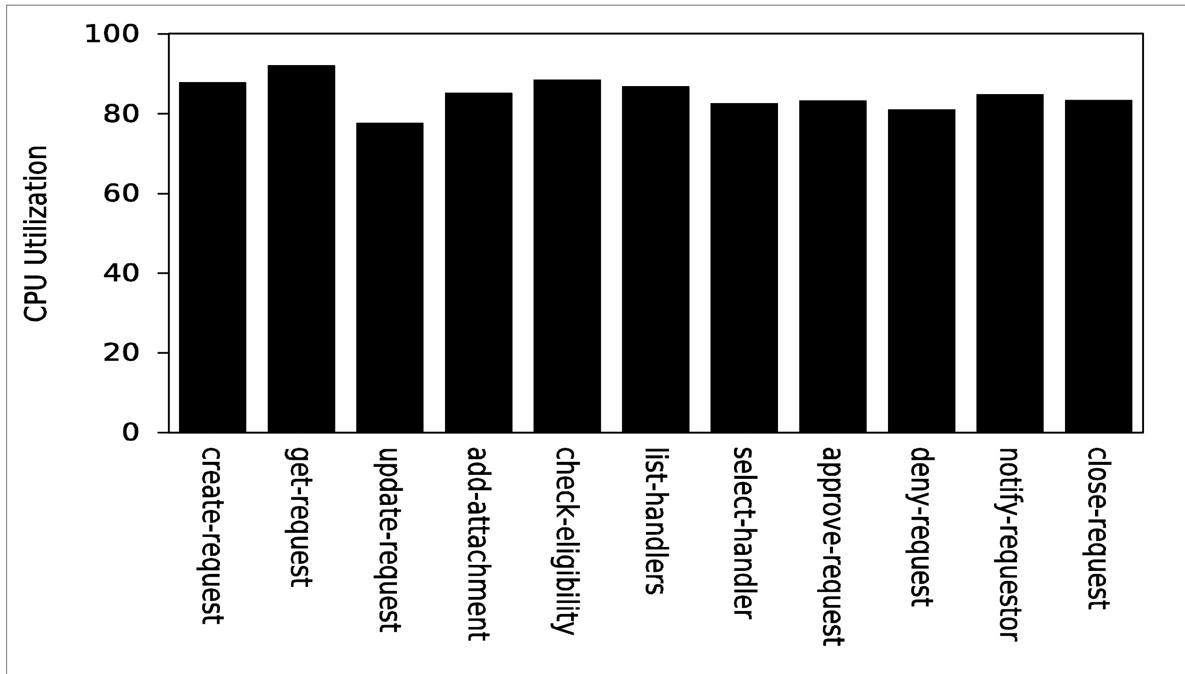


Figure 4.1.8.3 – Average CPU utilization on the workload driver system

Figure 4.1.8.4 shows the network throughput (in Mbytes/sec) in the most optimal configuration. The network throughput always stayed below 50 Mbytes/sec—well within the available bandwidth of the private Gigabit Ethernet network in our lab.

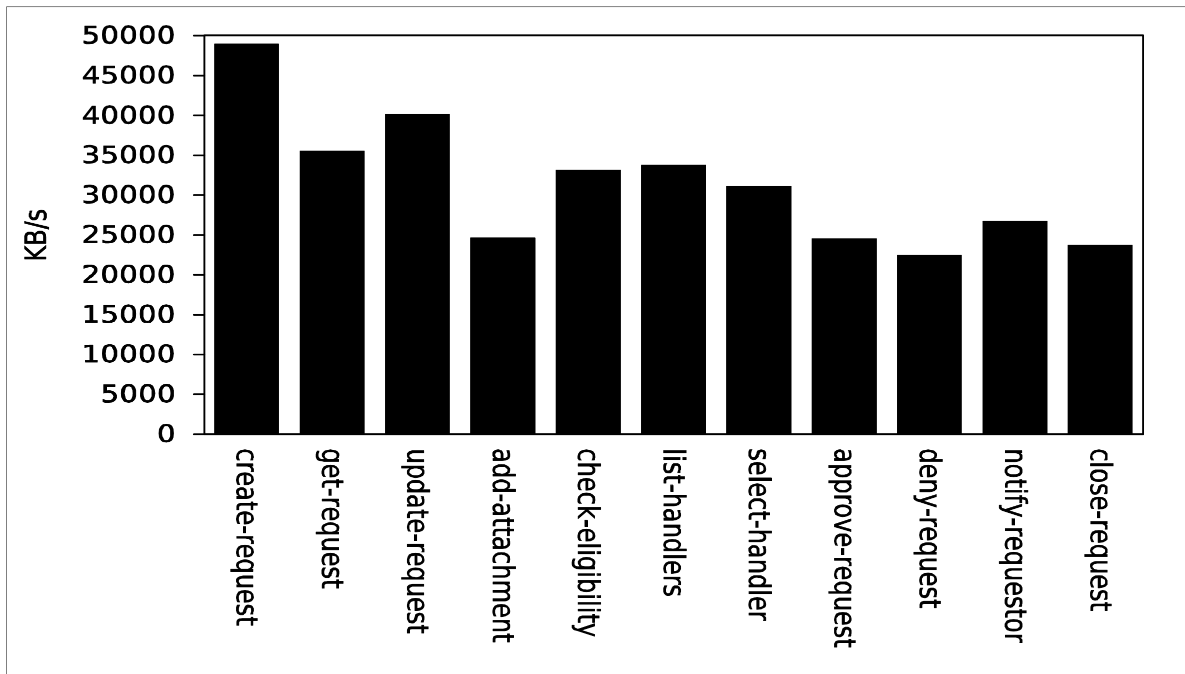


Figure 4.1.8.4 – Average network throughput

4.2. Results on the POWER Web Server

Next we examine the results obtained on the POWER6 Web Server (IBM Power 570).

4.2.1. "Out of the Box" Configuration

Figure 4.2.1 shows the relative throughput for each of the Web Service scenarios on the Power 570 server in the "out of the box" configuration—with all default settings for WebSphere Application Server (WAS) and Linux. Relative throughput for a Web service scenario is calculated as the average request rate (number of requests per second) for that scenario divided by the request rate for the *create_request_bean* scenario in the "Out of the Box" configuration.

Note that the default profile for WAS has administrative security enabled. Enforcing security involves additional overhead for authentication and access validation.

The geometric mean of the relative throughput scores on the Power 570 server is 1.55.

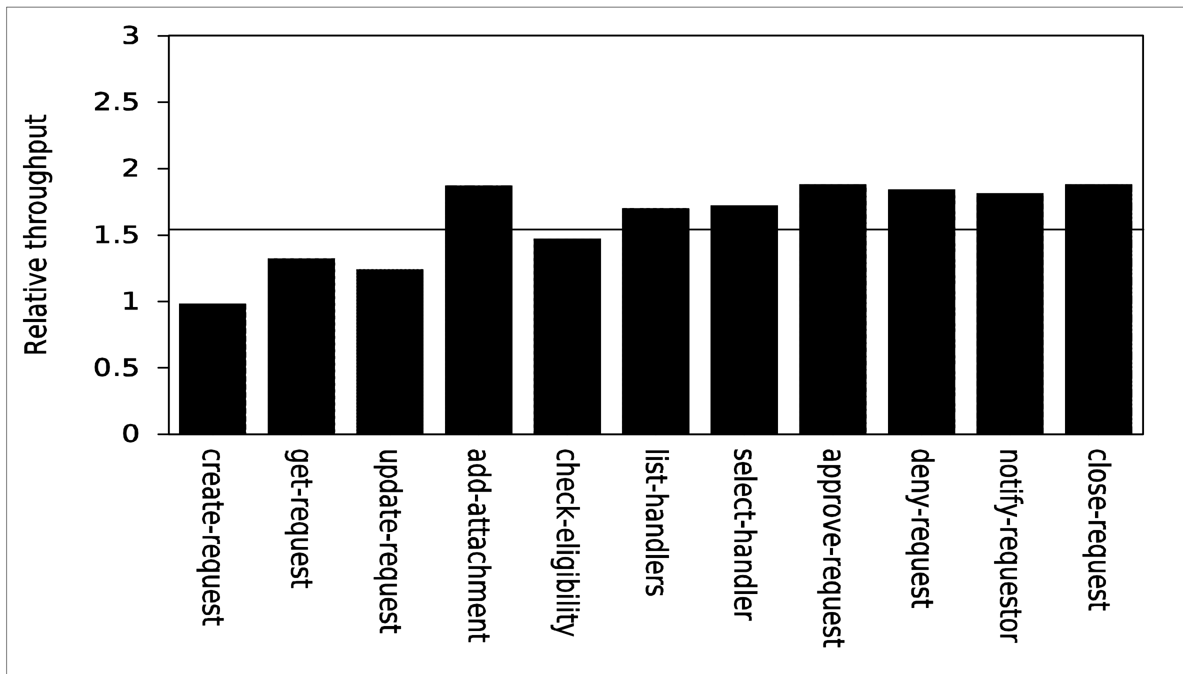


Figure 4.2.1 – Relative throughput for Web services on the Power 570 in "out of the box" configuration

4.2.2. Without WAS Administrative Security

As mentioned in the previous section, the default profile for WebSphere Application Server has administrative security enabled. Enforcing security involves additional overhead for authentication and access validation. If there is no requirement to enforce administrative security, it is better to create a WAS profile with administrative security disabled. Figure 4.2.2 shows the relative throughput for each of the Web service scenarios on the Power 570 server without administrative security enabled. Relative throughput for a Web service scenario is calculated as the average request rate (number of requests per second) for that scenario divided by the request rate for the *create_request_bean* scenario in the "Out of the Box" configuration.

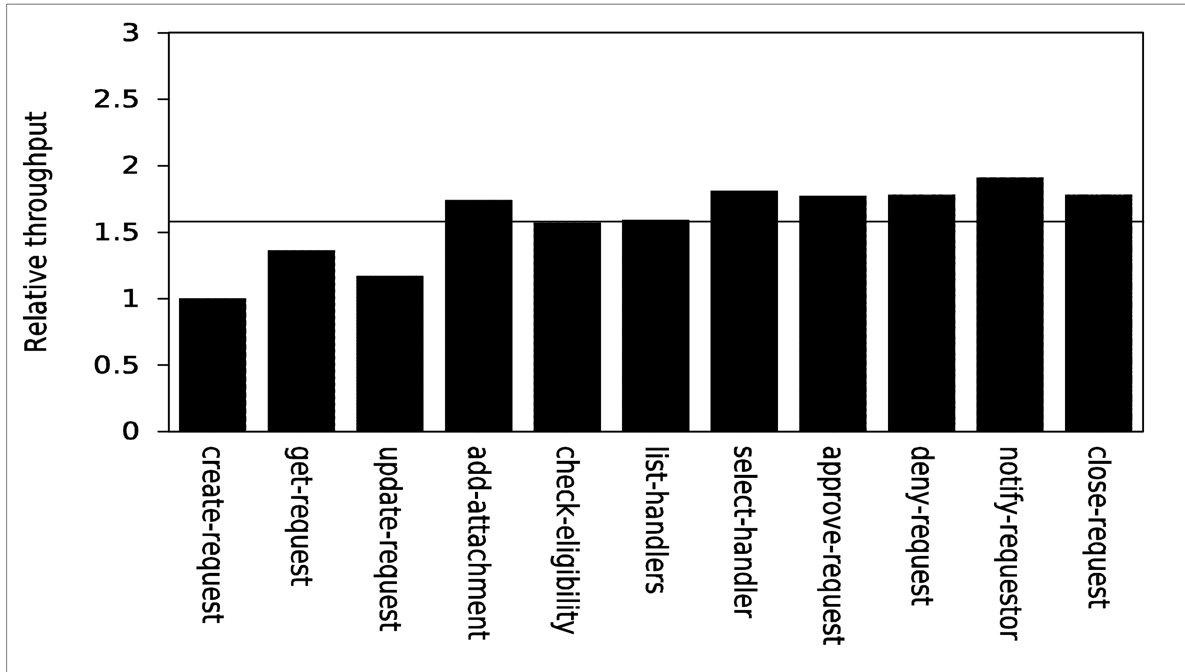


Figure 4.2.2 – Relative throughput for Web services on the Power 570 without administrative security

The geometric mean of the relative throughput scores on the Power 570 without administrative security is 1.59, a 2.6% performance improvement over the "out of the box" configuration.

4.2.3. WAS JVM Heap Tuning

In the world of Java, the JVM heap configuration has a significant impact on performance. There are several parameters available for tuning the JVM heap for better performance. Two basic JVM heap parameters are the size of the heap and the garbage collection policy.

The size of the WAS JVM heap is probably the most important factor for its performance. A too-small heap causes garbage collection to happen more frequently. A too-large heap size causes garbage collection to happen less frequently, or to take longer to compact the large heap. There are several tools available, such as the Tivoli® Performance Viewer (included with WebSphere), that can help analyze and monitor the heap usage and garbage collection so that the heap can be specifically tuned for a particular workload.

The default size of the JVM heap is typically too small. In our tests, by looking at the number of memory pages allocated to the heap that were actually in use while the Web service scenarios were being run, we found that the heap usage was between 1536 MB and 2048 MB. As a result, we set the JVM heap size for WAS at 2048 MB. This larger heap gave us a significant performance improvement over the default heap size. Since the Power 570 server has 32 GB of physical memory, which is more than adequate for our tests, we decided to set the minimum and maximum heap size to the same value (2048 MB) to avoid the overhead of frequent heap size changes and ensure optimal performance.

In addition to the JVM heap size, the garbage collection policy can also affect performance. The WAS JVM supports four different garbage collection policies. The default garbage collection policy is **`optthruput`**. During a garbage collection cycle under the **`optthruput`** policy, all application threads are stopped for mark, sweep, and compaction if needed. The garbage collector scans all the objects in the heap, marking any object that is in use, sweeps up the unused objects, reclaims their memory, and then compacts the remaining memory to reduce fragmentation. The entire process can take some time. All application threads are paused while the garbage is collected. Consequently, the **`optthruput`** policy results in longer garbage collection pause times, but more often than not, it yields the best overall

throughput. However, we decided to use the **gencon** (Generation Concurrent) garbage collection policy, which handles short-lived objects differently from long-lived objects, because our testing on the x86 Web server showed that the **gencon** policy was more suitable for Web service scenarios (see Section 4.1.3). Under the **gencon** policy, the heap is split into new and old segments. Long-lived objects are promoted to the old space while short-lived objects are garbage collected quickly in the new space (called a *nursery*).

We also set the size of the *nursery* to 1536 MB (75% of the total heap size) by going through the WAS Administration Console:

- Go to Servers → Application Servers → *server name* → Server Infrastructure → Java and Process Management → Process Definition → Additional Properties → Java Virtual Machine
- Enter **-Xgcpolicy:gencon -Xmn1536M** in the “Generic JVM arguments” box. Note that the JVM heap’s maximum and minimum sizes can also be set on this page; in our test, both the minimum and maximum heap sizes were set to 2048 MB

Figure 4.2.3 shows the relative throughput for each of the Web service scenarios on the Power 570 server with the WAS JVM heap size set to 2048 MB under **gencon** garbage collection policy with 1536 MB nursery. Relative throughput for a Web service scenario is calculated as the average request rate (number of requests per second) for that scenario divided by the request rate for the *create_request_bean* scenario in the “Out of the Box” configuration.

The geometric mean of the relative throughput scores on the Power 570 is 2.09 —an improvement of more than 35% over the “out of the box” configuration.

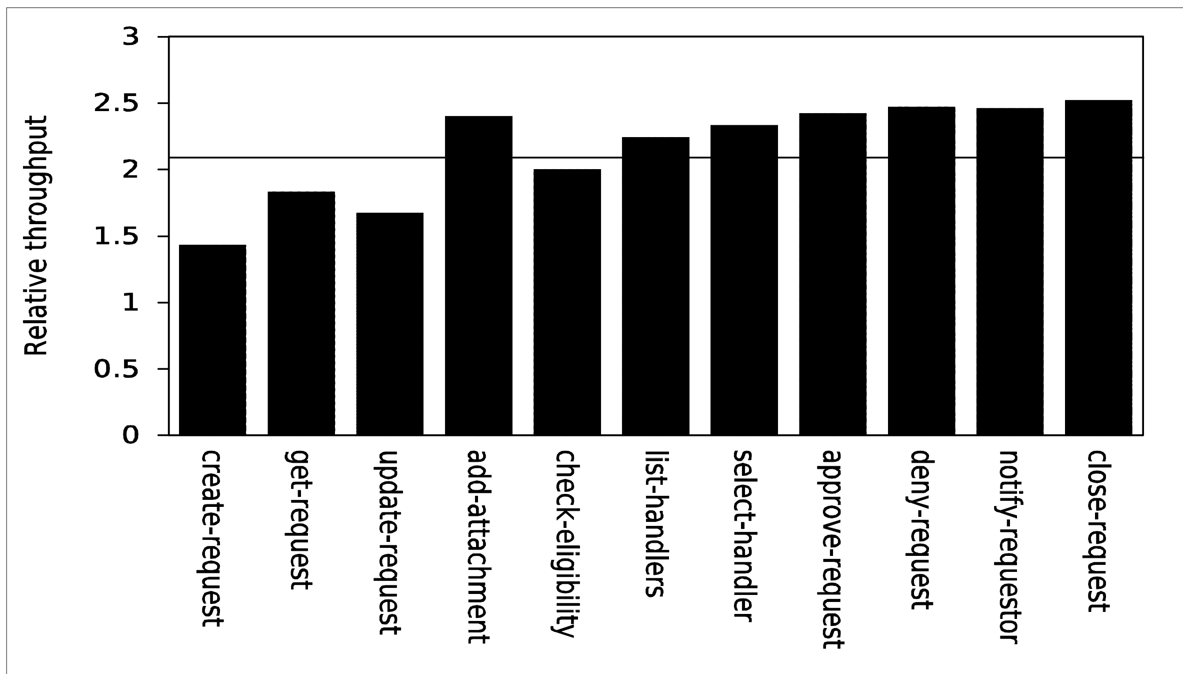


Figure 4.2.3 – Relative throughput for Web services on the Power 570 with WAS JVM heap tunings

4.2.4. Huge Page Support in Linux

Since the WAS JVM heap size was set to 2048 MB, which is much larger than the default size, the CPU overhead of managing and keeping track of memory in this large heap can be reduced by exploiting the Huge (Large) Page support provided by the Power architecture and supported by the Linux kernel. The Power processors support 16 MB huge pages.

To allow WAS JVM heap to use huge pages, we must first enable Linux to create and maintain a pool of huge pages. In our tests, we created a pool of 3 GB of huge pages (a total of 192 of 16 MB huge pages) by adding the following lines to the **/etc/sysctl.conf** file:

```
#Number of huge pages (192 x 16 MB = 3 GB)
vm.nr_hugepages = 192
#Size of shared memory is set to 4 GB (4294967296 bytes)
kernel.shmmax = 4294967296
kernel.shmall = 4294967296
```

(Note that we set the amount of shared memory to 4 GB so that there would be enough room for the 3 GB of huge pages in the shared memory pool.)

The WAS JVM was then configured to use huge pages by adding the parameter **-Xlp** in the same "Generic JVM arguments" box where the garbage collector parameters were set.

Figure 4.2.4 shows the relative throughput for each of the Web service scenarios on the Power 570 server with the WAS JVM heap tunings and huge page support. Relative throughput for a Web service scenario is calculated as the average request rate (number of requests per second) for that scenario divided by the request rate for the *create_request_bean* scenario in the "Out of the Box" configuration.

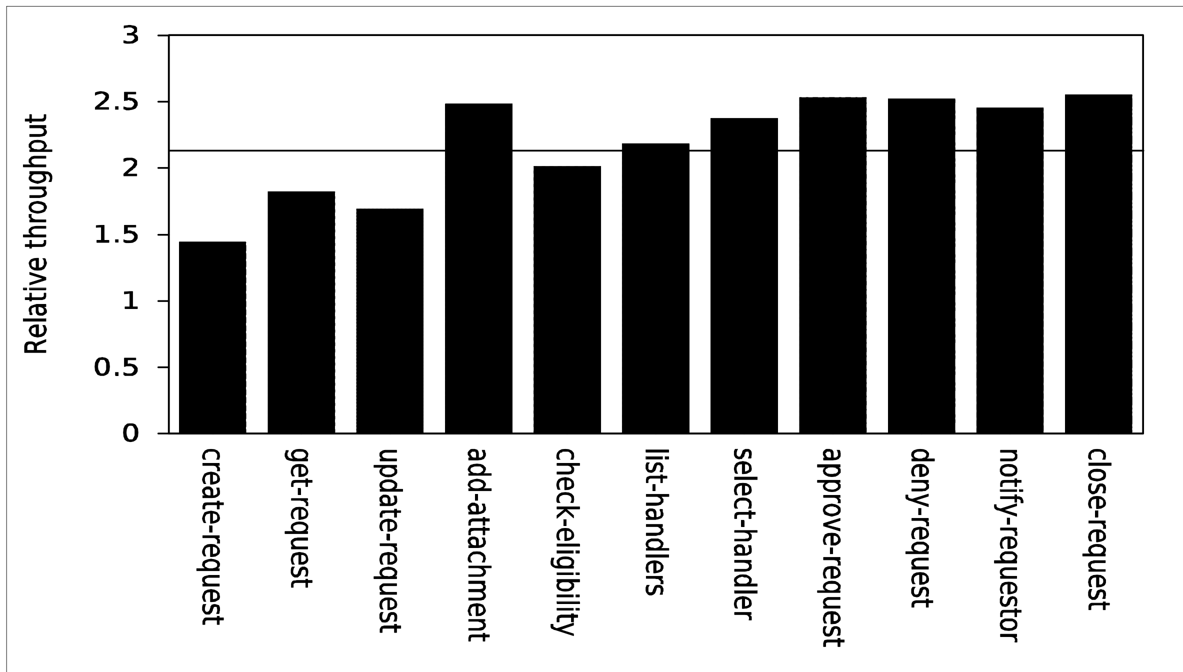


Figure 4.2.4 – Relative throughput for Web services on the Power 570 with huge page support

The geometric mean of the relative throughput scores on the Power 570 is 2.14, which is another 2.1% improvement over what we achieved with the WAS JVM heap tunings. Together with the WAS JVM heap tunings, the huge page support yields a total of 38% performance improvement over the "out of the box" configuration.

4.2.5. WAS HTTP Connection Settings

There are several WAS Web Container settings that can be tweaked for maximum concurrency in our tests. If there are not enough HTTP connections available, incoming clients will not be able to connect until a connection is freed. If the server's CPUs are not fully utilized, there is no memory constraint, and

there is available network bandwidth, the number of persistent HTTP connections for each port can be increased from the default value of 100 to improve the server's performance.

In our tests, setting the maximum number of HTTP persistent requests to **`unlimited`** gave us a noticeable performance improvement over what we had with WAS JVM tunings and huge page support. The results are shown in Figure 4.2.5.

To change the number of HTTP connections available for a given port, we used the WAS Administration Console as follows:

- Go to Servers → Application Servers → *server name* → Communications and click on the Ports link
- Find the port number in the table and click on "View associated transports" for that port
- Click on the transport chain that is listed
- Click on "HTTP inbound channel (HTTP_n)," where "n" denotes channels 1 to 4
- Either click on "Maximum persistent requests per connection" and enter a number in the "Specify maximum number of persistent requests" box, or click on "Unlimited persistent requests per connection"

As shown in Figure 4.2.5, the geometric mean of the relative throughput scores on the Power 570 is 2.27, which is another 6.1% improvement over what we achieved with the WAS JVM heap tunings and huge page support. (Relative throughput for a Web service scenario is calculated as the average request rate for that scenario divided by the request rate for the *create_request_bean* scenario in the "Out of the Box" configuration.) Together with the WAS JVM heap tunings and huge page support, the Web container tunings yield a total of 46% performance improvement over the "out of the box" configuration.

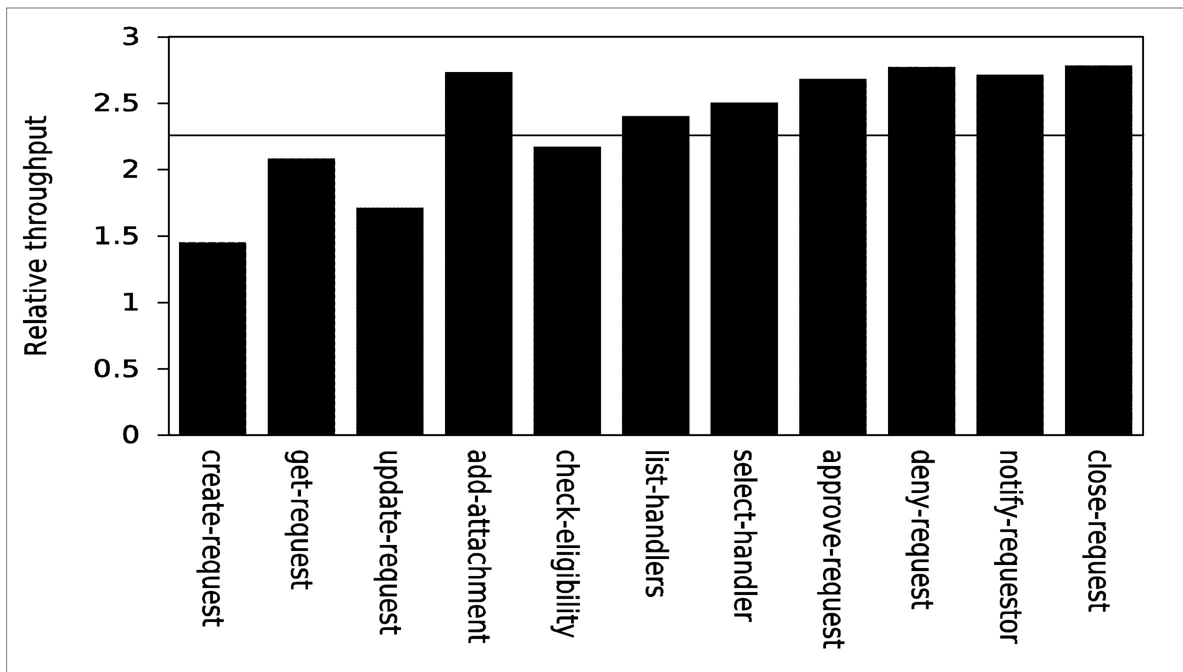


Figure 4.2.5 – Relative throughput for Web services on the Power 570 with Web container tunings

4.2.6. WAS Thread Pool Settings

There is another area that we can tune for maximum concurrency: the number of threads available to service requests from the clients. For example, threads in the Web Container thread pool are used for handling incoming HTTP and Web service requests. These thread pools are shared by all applications deployed on the server, so in many cases, these pools need to be larger than their default sizes. We

experimented with larger maximum numbers of threads in the default thread pool as well as the Web container thread pool. These numbers partly depend on the number of CPUs on the Web server since the more CPUs we have, the more threads can be executed concurrently. In our test setup, the Power 570 server had two dual-core POWER6 processors, and if we take Simultaneous Multi-Threading (SMT) into account, we would actually have 8 “logical” processors that can be seen by Linux. Based on recommendations from the WebSphere Performance Team, with 8 “logical” processors, we set the maximum number of threads in the default thread pool to 200 (default value is 20) and in the Web container thread pool to 100 (default value is 50). However, we found that, in our tests, just increasing the thread pool maximum values alone would actually degrade the performance since the JVM heap would become too small to accommodate the higher number of threads that can be executed concurrently. To get any performance benefit from the larger thread pools, we would need a larger JVM heap. Our test results confirmed this: larger thread pools together with a larger 3 GB WAS JVM heap yielded a small, but noticeable, performance improvement over the configuration described in the preceding section.

Figure 4.2.6 shows the performance of the Web services (in terms of relative throughput, which is calculated as the average request rate divided by the request rate for the *create_request_bean* scenario in the ”out of the box” configuration) using WAS JVM heap tunings (including larger 3 GB size), huge page support, Web container tunings, and larger thread pools.

Changing the number of threads in a thread pool can be done through the WAS Administration Console as follows:

- Go to Servers → Application Servers → *server name* → Additional Properties → Thread Pools
- Click on the thread pool you want to change and enter new values in the “Maximum Size” boxes

The geometric mean of the relative throughput scores on the Power 570 is 2.30, which is another 1.8% improvement over what we achieved previously. All of our tunings so far have yielded a total of 49% performance improvement over the “out of the box” configuration.

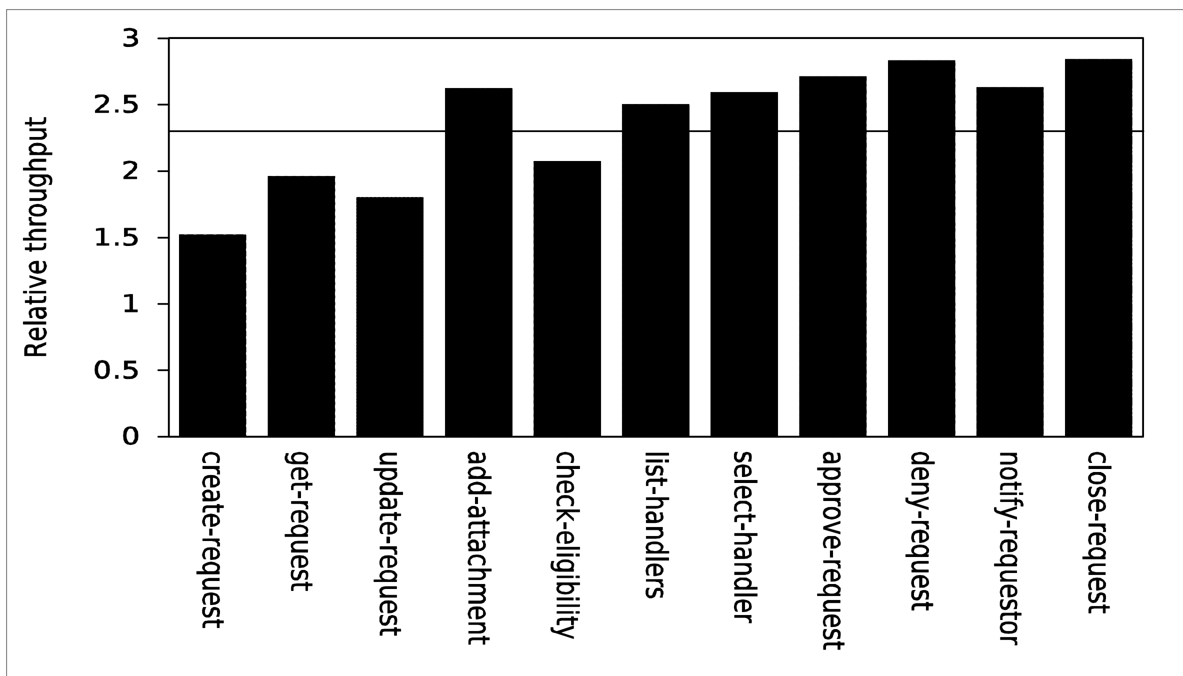


Figure 4.2.6 – Relative throughput for Web services on the Power 570 with WAS JVM thread pool tunings

4.2.7. Disabling Performance Monitoring Infrastructure (PMI)

To further optimize the performance of Web services in our tests, we next decided to turn off all performance monitoring, tracing, and logging. These are often necessary when setting up a server or when debugging problems or issues, but they do introduce some performance overhead. As a result, it is recommended that tracing and monitoring be used judiciously, and whenever possible, turned off entirely to ensure optimal performance.

Disabling the WAS Performance Monitoring Infrastructure (PMI) can be done through the WAS Administration Console as follows:

- Go to Monitoring and Tuning → Performance Monitoring Infrastructure (PMI) → *server name*
- Uncheck the “Enable Performance Monitoring Infrastructure (PMI)” box, and in the “Currently Monitored Statistic Set” box, select ”None”

Figure 4.2.7 shows the performance of Web services when we disabled all performance monitoring, tracing, and logging (on top of the previous tunings that we have done).

The geometric mean of the relative throughput scores on the Power 570 is 2.32, which is another 0.8% improvement over what we achieved previously. (Relative throughput for a Web service scenario is calculated as the average request rate for that scenario divided by the request rate for the *create_request_bean* scenario in the “Out of the Box” configuration.) All of our tunings so far have yielded a total of 50% performance improvement over the “out of the box” configuration.

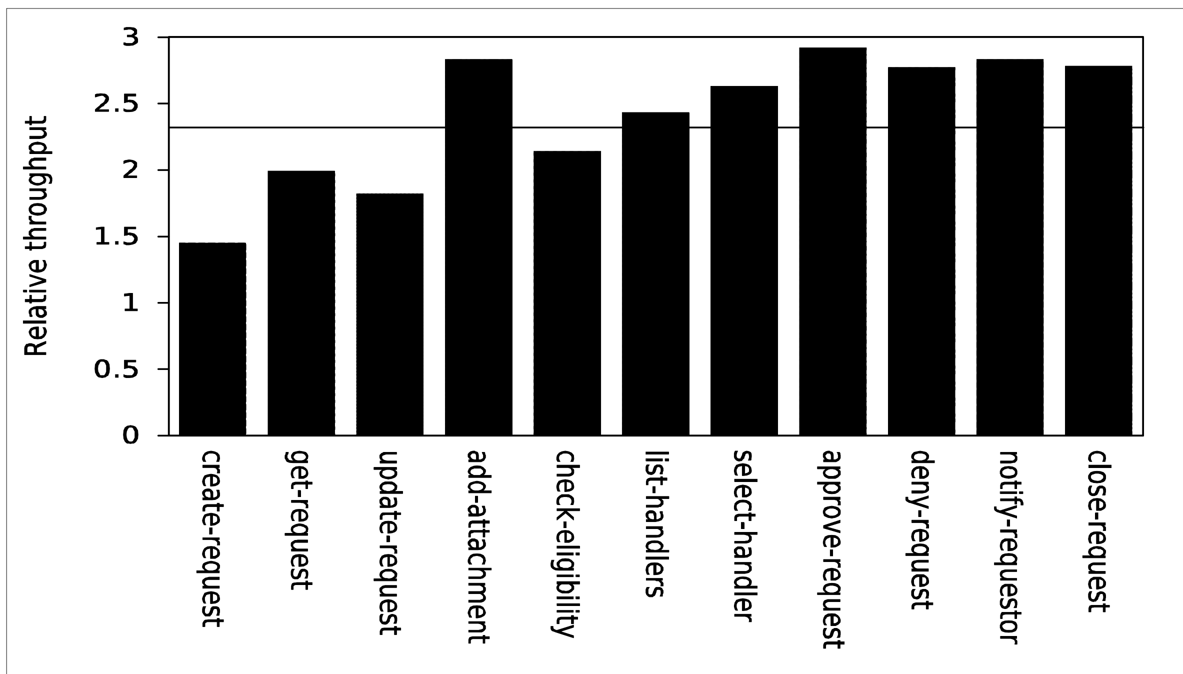


Figure 4.2.7 – Relative throughput for Web services on the Power 570 with WAS PMI disabled

4.2.8. Summary

Figure 4.2.8.1 shows the geometric mean scores for the tuning items that we identified in the preceding sections. It is clear that tuning the WAS JVM heap (larger 2 GB JVM heap, **`gencon`** garbage collection policy with a nursery size of 1.5 GB) gave us the largest performance gain, followed by setting the maximum number of persistent HTTP connections for Web container ports to **`unlimited.`** Other tuning

items yielded discernable, but not significant, performance benefits. Overall, the cumulative performance gain for all the tunings was 50% over the “out of the box” configuration.

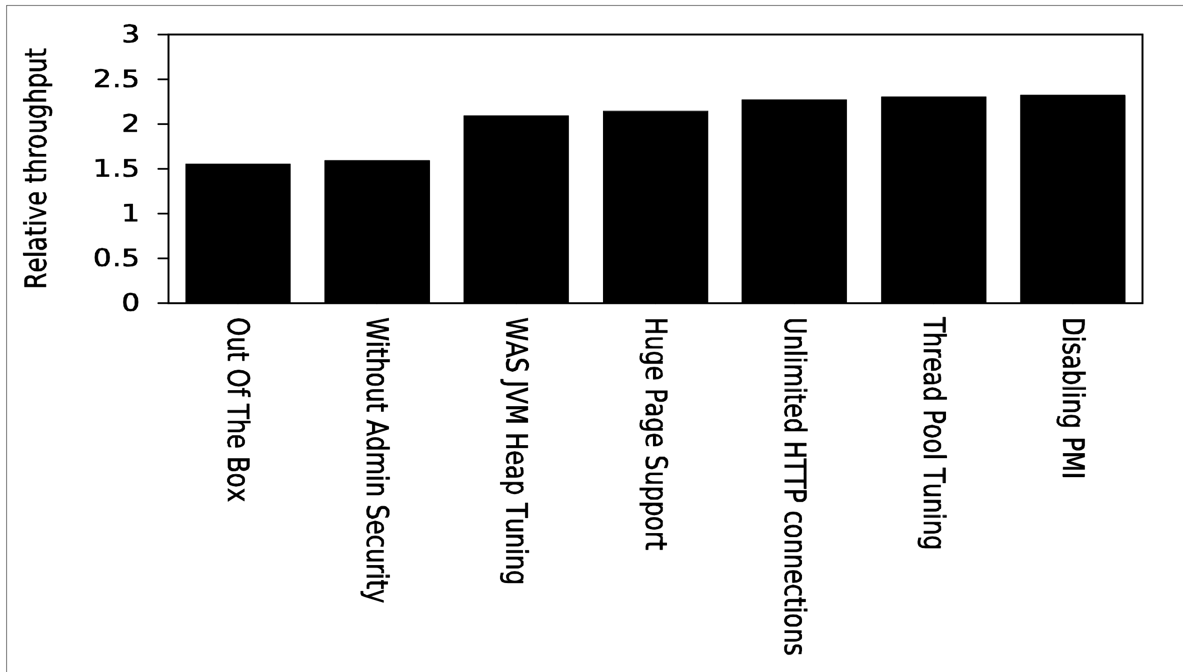


Figure 4.2.8.1 – Geometric mean of relative throughput for Web Services for tuning items

Finally, we determined whether there was any performance bottleneck in the most optimal configuration in our test setup. Figure 4.2.8.2 shows the average CPU utilization on the Web server. Although it varied a little during different Web service invocations, the CPU utilization was never higher than 80%, and, in fact, it remained at around 60% most of the time. As Figure 4.2.10 shows, the average CPU utilization on the workload driver system was higher, averaging around 80% most of the time.

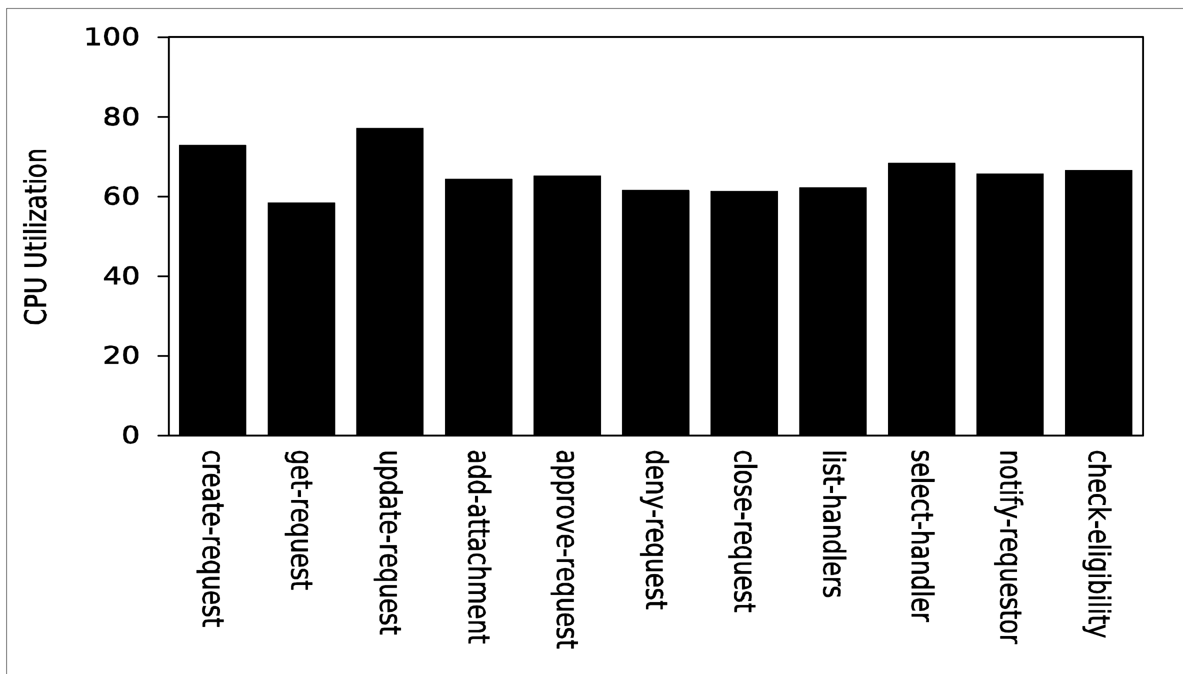


Figure 4.2.8.2 – Average CPU utilization on the Web Service Host

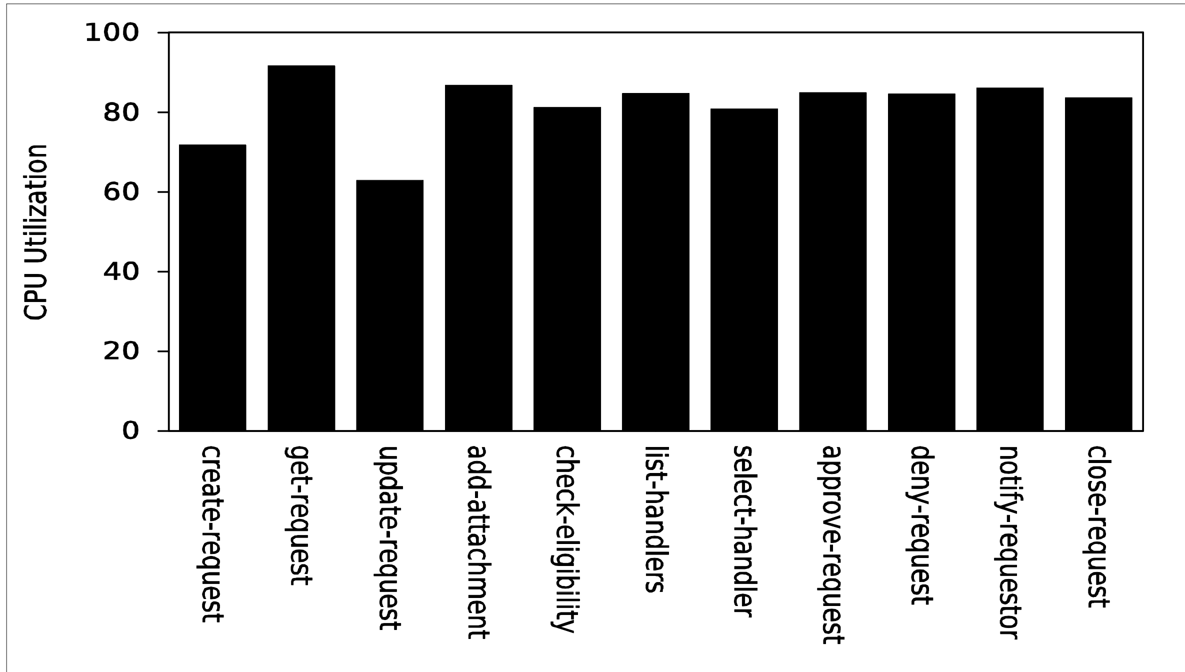


Figure 4.2.8.3 – Average CPU utilization on the Workload Driver system.

Figure 4.2.8.4 shows the network throughput (in Mbytes/sec) in the most optimal configuration. The network throughput always remained below 50 Mbytes/sec—well within the available bandwidth of the private Gigabit Ethernet network in our lab. As for the physical memory usage on the Power 570 Web server, the entire workload used only 6 GB of the 32 GB of memory available on the Power 570. The disk I/O traffic was fairly negligible—less than 100 Kbytes/sec on the Power 570 Web server.

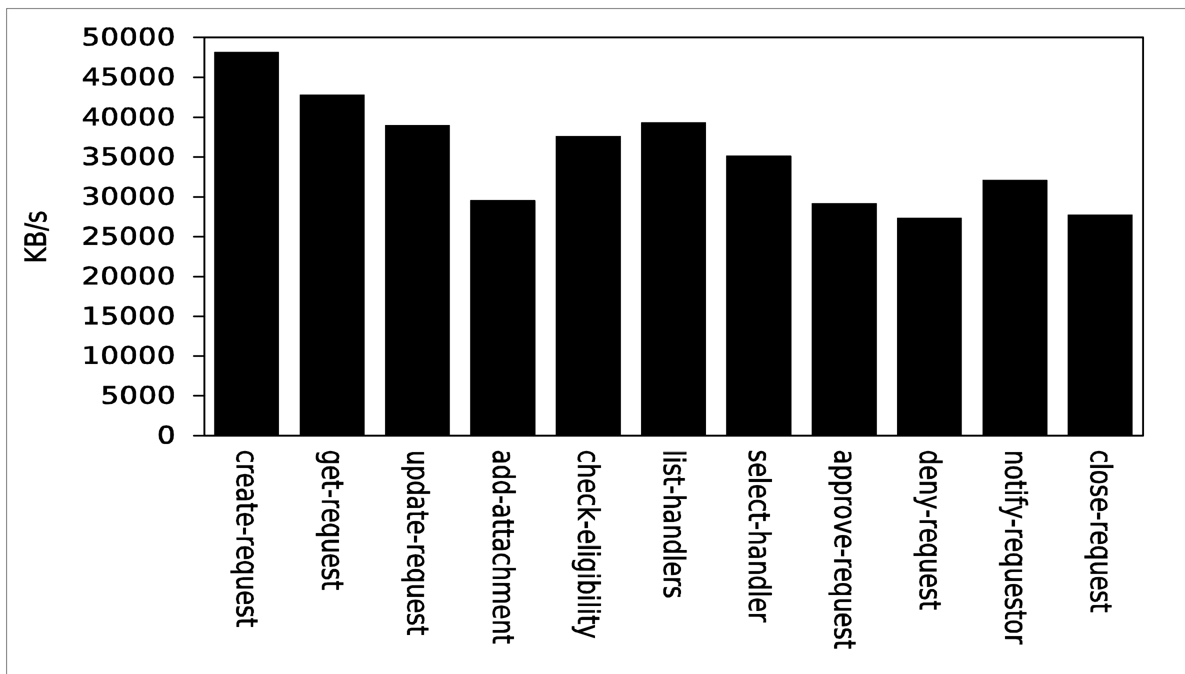


Figure 4.1.8.4 – Average network throughput

5. Conclusions

Based on the results of our testing with Web service scenarios, we can state the following conclusions:

- Of the various tuning parameters examined in this study, the size of the WAS JVM heap had the greatest performance impact on both the x86 and Power Web servers. It is very important to have the heap sized appropriately for the workload. It also helps to use a garbage collection policy that is appropriate for the workload.

- The benefit of huge (large) page support in Linux was mixed in our testing. On the Power Web server, we found that allocating huge 16 MB pages for the WAS JVM heap improved performance. However, on the x86 Web server with much smaller huge page size (2 MB), we found that the use of huge pages for the heap was only beneficial if the heap was constrained; if an appropriately sized heap was configured, huge pages would be of little benefit.

- Configuring adequate persistent HTTP connections yielded a modest improvement in performance on both platforms.

- Disabling WAS logging and tracing had a small benefit after the other tunings were in place.

- In our testing, increasing the size of the WAS thread pools did not appear to yield much benefit on the x86 Web server. On the Power Web server, increasing the size of the WAS thread pools helped performance only after configuring a larger JVM heap to accommodate more concurrent threads. This demonstrates that it is important to consider all parameter tunings as a whole, since their effects can interact with one another.

References

[1] Services Oriented Architecture (SOA) Entry Points,
http://www-
306.ibm.com/software/solutions/soa/entrypoints/index.html?S_TACT=107AG01W&S_CMP=campaign

[2] WebSphere Application Server,
http://www-306.ibm.com/software/webservers/appserv/was/

[3] WebSphere Process Server: IBM's New Foundation for SOA,
http://www.ibm.com/developerworks/websphere/library/techarticles/0509_kulhanek/0509_kulhanek.html

[4] SOA entry point: service creation and reuse,
http://www-
.ibm.com/software/solutions/soa/entrypoints/reuse.html?S_TACT=107AG01W&S_CMP=campaign

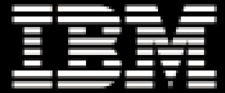
[5] IBM Power 570,
http://www-03.ibm.com/systems/power/hardware/570/

[6] IBM POWER6 Microarchitecture, IBM Journal of Research and Development,
http://www.research.ibm.com/journal/rd/516/le.html

[7] IBM System x3850 M2,
http://www-07.ibm.com/systems/includes/content/x/pdf/XSD03019USEN.pdf

[8] IBM System x3850 M2 Enterprise Server's X4 Technology,
http://www-03.ibm.com/systems/x/hardware/enterprise/x3850m2/x4/info.html

[9] The Basic Web Services Stack: IT Web Services: A Roadmap for the Enterprise, by Alex Nghiem,
Prentice Hall (October 8, 2002)



© IBM Corporation 2008

IBM Systems and Technology Group
3039 Cornwallis Road
Research Triangle Park, NC 27709

Produced in the USA
06-08
All rights reserved

Warranty Information: For a copy of applicable product warranties, write to: Warranty Information, P.O. Box 12195, RTP, NC 27709, Attn: Dept. JDJA/B203. IBM makes no representation or warranty regarding third-party products or services including those designated as ServerProven or ClusterProven.

IBM, the IBM logo, eServer, xSeries, X-Architecture, System x, System p, Power, POWER6, IBM Redbooks and BladeCenter are trademarks of the International Business Machines Corporation in the United States and/or other countries. For a complete list of IBM Trademarks, see www.ibm.com/legal/copytrade.shtml.
Intel, Xeon and Hyper-Threading Technology are trademarks or registered trademarks of Intel Corporation.
Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.
Linux is a registered trademark of Linux Torvalds in the United States, other countries, or both.
Microsoft and Windows are registered trademarks of Microsoft Corporation.

All other products may be trademarks or registered trademarks of their respective companies.

Information about non-IBM products is obtained from the manufacturers of those products or their published announcements. IBM has not tested those products and cannot confirm the performance, compatibility, or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

Performance is based on measurements using industry standard or IBM benchmarks in a controlled environment. The actual throughput that any user will experience will vary depending upon considerations such as the amount of multiprogramming in the user's job stream, the I/O configuration, the storage configuration, and the workload processed. Therefore, no assurance can be given that an individual user will achieve performance levels equivalent to those stated here.

IBM reserves the right to change specifications or other product information without notice. References in this publication to IBM products or services do not imply that IBM intends to make them available in all countries in which IBM operates. IBM PROVIDES THIS PUBLICATION “AS IS” WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. Some jurisdictions do not allow disclaimer of express or implied warranties in certain transactions; therefore, this statement may not apply to you.