SOA Performance on Linux: Connectivity Entry Point

Tuning WebSphere and Linux for Web Mediation Services

Khoa Huynh
Vivek Kashyap
Mark Peloquin

IBM Systems & Technology Group
August 2008

Contents

Abstract

This paper is the second 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 second entry point – the Connectivity Entry Point. More specifically, this paper examines the transformation and routing service mediations, which can be performed on service requests and responses between clients and Web servers, and how we can tune WebSphere® and Linux to achieve better performance for these service mediations. We will take a close look at the impact of several parameters and features in WebSphere®, such as Java heap size, garbage collection policy, thread pools, persistent HTTP connections, and performance monitors, as well as in Linux, such as the huge page support, on the performance of service mediation applications. We consider two servers with different architectures to host the Web services and the WebSphere Enterprise Service Bus (WESB): an IBM System x3850 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 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 second in a series of white papers on SOA performance in the Linux environment—focuses on the *Connectivity* entry point, which encompasses the Enterprise Service Bus (ESB). In particular, this paper takes a close look at the performance issues of transformation and routing service mediations on the ESB and how we can tweak WebSphere and Linux to optimize the performance of Web service mediations in the Linux SOA environment.

2. Performance Evaluation Methodology

In this paper, we evaluate the performance of various transformation and routing mediations and Web service bindings implemented on IBM WebSphere Application Server (WAS) and WebSphere Enterprise Service Bus (WESB) 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 and WESB are 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 third area: the integration of Web services using WESB and WAS, which maps to the SOA *Connectivity* Entry Point defined by IBM. The use of DataPower appliances is not considered in this paper.

In our benchmark, the Web services are implemented as part of a *ClaimServices* application. These Web services represent 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.

For service mediations, we consider both routing and transformation mediations. Transformation mediations can be performed on service requests, and in some cases, responses, using Extensible Stylesheet Language Transformations (XSLT). XSLT is an XML-based language used for the transformation of XML documents into other XML or human-readable documents. There are different levels of complexity for these transformations, and we will consider the following:

- *XSLT Value Transformation,* which transforms the value of a single element in the service request message using XSLT.
- *XSLT Namespace Transformation,* which transforms service requests and responses from one schema to another using XSLT. In this case, the schemas are largely the same but the name of an element differs and the two schemas have different namespaces.

- *XSLT Schema Transformation,* which transforms service request and response messages from one schema to another using XSLT. In this case, the schemas are completely different, but they contain similar data which is mapped from one to the other. In addition to the transformation, a value from the service request is transferred to the response message by storing it in a context header.
- *Composite Module* mediation consists of three mediation primitives wired together inside a single, composite mediation module. The three mediation primitives are authorization (a routing mediation which checks a password field in the body of the service request), route on body, and transform. The composite module approach is good for performance because it saves the overhead of inter-module calls, but at the expense of the ability to individually administer the pieces of the overall mediation.

Routing mediations route requests to different Web services based on content. In particular, we will examine *route-on-body* mediation which routes each request to the appropriate Web service based on the content of a field in the body of that request.

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 is a stand-alone multi-threaded HTTP client, which uses up to 50 threads with 20 maximum transactions to generate SOAP service requests to the *ClaimServices* application. In other words, our Web service mediation tests have the following characteristics:

- Stand-alone, multi-threaded HTTP client to produce SOAP service requests
- Synchronous SOA (XML) / HTTP request / response invocation
- XSLT transformation and route-on-body service mediations
- *ClaimServices* application hosts Web services

We will consider both x86 and IBM Power Architecture® platforms for the server hosting the Web services and the Enterprise Service Bus (i.e., running WPS v6.1 and the *ClaimServices* application). The workload driver runs on a separate x86 server (an IBM System x3650).

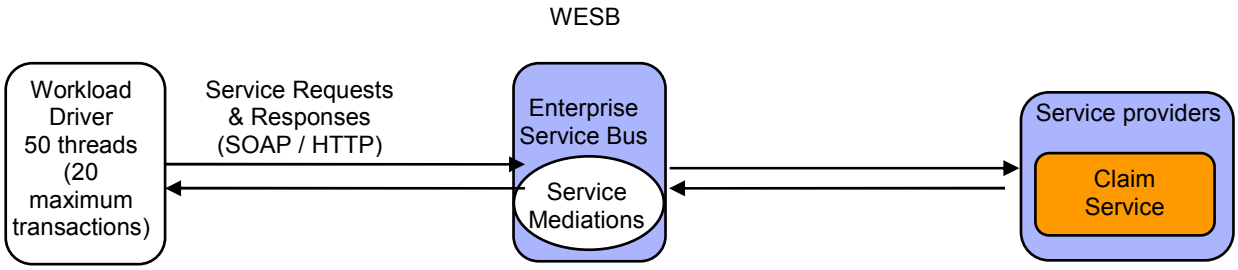

Figure 2.1 – Service Mediations

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.

In our study, we investigated the performance impact of many configuration parameters and settings for both WebSphere and the Linux operating system. However, in this paper, we will identify only those parameters and recommendations 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 would be excluded from this paper.

3. Systems Configurations

3.1. Hardware

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

3.1.1. x86 Architecture-Based WAS / WESB 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 – IBM System x3850 M2 Configuration

3.1.2. IBM Power Architecture-Based WAS / WESB Server

For the Power platform, we used an IBM Power 570 (Table 3.2) [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 3.2 – IBM Power 570 Configuration

3.1.3. Workload Driver

The workload driver was an IBM System x3650 (Table 3.3).

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 3.3 – IBM System x3650 Configuration

All servers 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 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 IBM Power 570 was Novell SUSE Linux Enterprise Server (SLES) 10 Service Pack (SP) 1 for PPC (ppc64).

Both 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 the workload driver system (IBM System x3650) was Novell SUSE Linux Enterprise Server (SLES) 10 Service Pack (SP) 2 for AMD64 & EM64T (x86_64).

4. Performance Results

First we look at the performance of the Web service mediations on the IBM System x3850 M2 server.

4.1. Results on the IBM System x3850 M2 server

Our testing indicated that WebSphere Java Virtual Machine (JVM) heap optimization, Linux huge page support, and tuning for maximum concurrency appear to yield noticeable performance improvement for service mediation scenarios. Let us first look at the service mediation performance with the payload size set to 3 KB for both service request and response messages.

4.1.1. Service Mediation Performance with 3-KB Payloads

Figure 4.1.1 shows the performance of WESB transformation and routing mediation scenarios and the performance gains achieved by WebSphere JVM heap tuning, Linux huge page support, and concurrency tuning with payload size set to 3 KB.

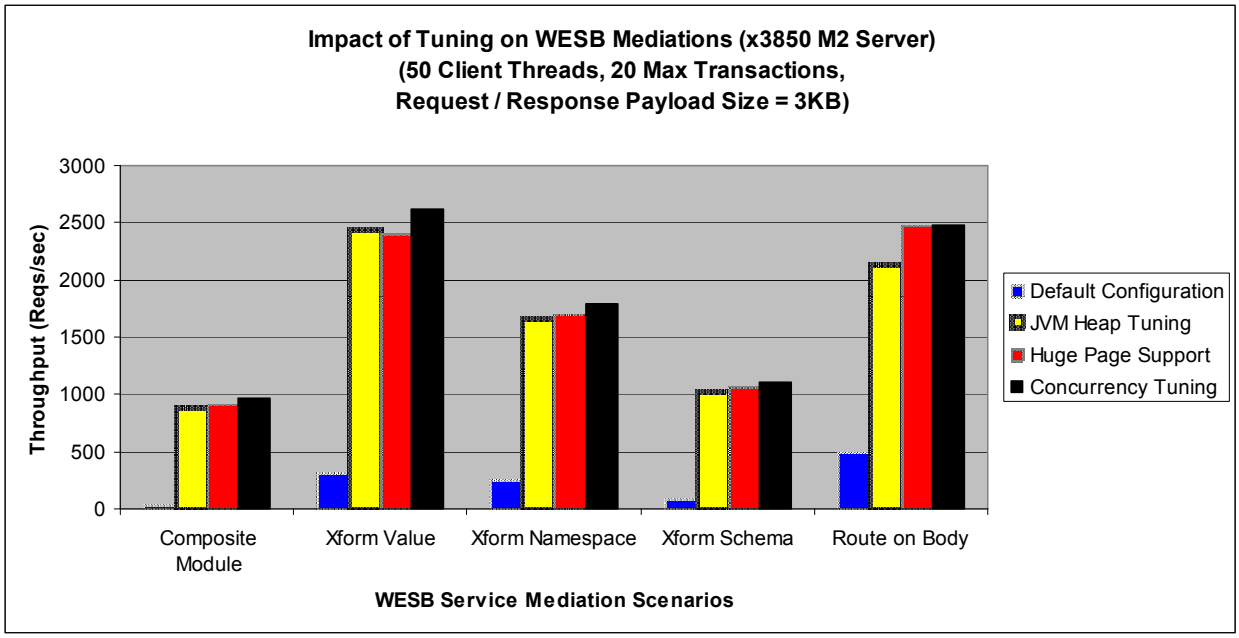

Figure 4.1.1 – Impact of JVM heap tuning, Linux huge page support, and concurrency tuning

Let us first look at the WebSphere JVM heap optimizations.

4.1.1.1. WebSphere 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 (GC) policy.

The size of the WebSphere JVM heap is probably the most important factor for the JVM 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.

In addition to the JVM heap size, the garbage collection (GC) policy can also affect performance. The WebSphere JVM supports four different garbage collection policies, as shown in *Table 4.1.1.1.1*. The default 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 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 being collected. Consequently, the **`optthruput`** policy results in longer garbage collection pause times, but in many cases, it yields the best overall throughput. In contrast, the **`optavgpause`** policy trades high throughput for shorter garbage collection pauses by performing some of the garbage collection concurrently. This is good for GUI applications where user interactive performance is critical. For transaction-based applications involving many short-lived objects, the **`gencon`** (Generational Concurrent) garbage collection policy should be better. Under the **`gencon`** policy, the JVM 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 (called a *nursery*). The **`subpool`** garbage collection policy is not available on IBM System x servers, so it is not considered in this section.

GC Policy	JVM Command Line Option	Description
Optimize for throughput	*-Xgcpolicy:optthroughput* (default)	This is the default policy. It is typically used for applications where raw throughput is more important than short GC pauses. The application is paused every time while garbage is being collected.
Optimize for pause time	*-Xgcpolicy:optavgpause*	This policy trades high throughput for shorter GC pauses by performing some of the garbage collection concurrently. The application is paused for shorter periods.
Generational concurrent	*-Xgcpolicy:gencon*	This policy handles short-lived objects differently from long-lived objects. Applications that have many short-lived objects can see shorter pause times with this policy while still producing good throughput.
Subpooling	*-Xgcpolicy:subpool*	This policy uses an algorithm similar to the default policy, but employs an allocation strategy that is more suitable for multiprocessor machines. This policy is recommended for SMP machines with 16 or more processors. This policy is only available on IBM pSeries® and zSeries® platforms. Applications that need to scale on large machines can benefit from this policy.

Table 4.1.1.1.1 – Garbage collection policies supported in WebSphere 6.1.

To change the JVM heap size or set the garbage collection policy in WebSphere, we can use the WebSphere Application Server (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)
- Enter appropriate JVM command line option for the garbage collection policy (as shown in the middle column of *Table 4.1.1.1.1*) into the “Generic JVM arguments” box. For example, if we want to use the **`gencon`** garbage collection policy with a nursery size of 1536 MB, we would enter “**`-Xgcpolicy:gencon -Xmn1536M`**” into the “Generic JVM arguments” box.

Table 4.1.1.1.2 shows the impact of various JVM heap settings, including the garbage collection policy, on the performance of the Composite Module mediation scenario with the request and response payload

size set to 3 KB. (To obtain data on the overall GC overhead and total GC pause times, we enabled the verbose GC output in the WebSphere JVM.)

GC Policy	JVM Heap Size (MB)	Throughput (Reqs/sec)	Overall GC Overhead	Total GC Pause
Default (optthruput)	1024	592.34	21%	88 secs
	2048	740.42	12%	49 secs
	3072	799.84	11%	37 secs
	4096	804.79	9%	29 secs
Gencon (nursery = 75% heap)	2048	743.83	12%	49 secs
	3072	782.79	10%	37 secs
	4096	859.87	9%	31 secs
	5120	870.04	3%	10 secs
	6144	895.06	2%	9 secs
	6144 (no gc trace)	900.23	N/A	N/A
	8192	883.4	3%	8 secs
optavgpause	2048	612.04	4%	17 secs
	6144	640.79	4%	13 secs

Table 4.1.1.1.2 – Impact of JVM heap settings on Composite Module mediation with payload size = 3KB.

With the payload size for both requests and responses set to 3 KB and with the default JVM settings, we could only achieve an average Web service request rate of 38.54 requests per second in the Composite Module mediation scenario. To optimize the JVM heap, we first considered the default GC policy (**optthruput**). By setting both the initial JVM heap size and the maximum heap size to 1024 MB, we immediately got a significant performance boost to 592.34 requests per second – more than 15X improvement!

However, the overall garbage collection overhead was 21%, which is too high. For optimal JVM performance, it is recommended that the overall GC overhead should be less than 10%. Increasing the JVM heap size to 4096 MB reduced the overall GC overhead to a more acceptable 9% and resulted in another 35% performance boost in the average service request rate.

Since increasing the JVM heap from 3072 MB to 4096 MB resulted in a very small performance boost (0.5%), we believe that increasing the JVM heap size further would not do much good. Instead, we wanted to see if we could optimize the heap usage further by switching to another garbage collection policy. As the data in *Table 4.1.1.1.2* indicates, the Generational Concurrent (**gencon**) policy, with the nursery occupying 75% of the total heap, outperformed the default **optthruput** policy for heap sizes ranging from 2048 MB to 4096 MB. More specifically, with the heap size set to 4096 MB, the **gencon** GC policy resulted in 7% performance gain over the default policy. The **gencon** policy also outperformed the **optavgpause** policy; in fact, the **optavgpause** policy yielded even worse performance than the default **optthruput** policy. Therefore, it is safe to conclude from this data that the **gencon** policy is the most suitable garbage collection policy for Web service mediation scenarios.

Using the **gencon** GC policy, we increased the JVM heap size to 5120 MB and received a noticeable performance boost. Increasing the heap to 6144 MB also resulted in another sizable performance boost. However, increasing the JVM heap beyond 6144 MB did not yield any noticeable performance gain. As a result, we believe that the optimal JVM heap size is 6144 MB with the **gencon** garbage collection policy. In fact, as *Figure 4.1.1* shows, this optimal JVM heap configuration accounted for the largest performance improvement across all Web service mediation scenarios considered in our study.

We also found that enabling the verbose garbage collection output, which was needed for analyzing the garbage collection performance, did not appear to adversely impact the Web service rate. In fact, with the JVM heap size set to 6144 MB under the **gencon** garbage collection policy, the verbose GC output only resulted in approximately 0.5% performance overhead.

4.1.1.2. Huge Page Support in Linux

One of the factors to consider in server performance is memory access overhead. 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, i.e., a TLB miss, it must walk the virtual address through the page tables to get its physical address. The new mapping is then written to the TLB, overriding an existing entry.

Linux has support for *huge pages* (also known as *large pages* in many processor hardware manuals). In the x86 architecture, normal pages are 4 KB in size and huge pages are 2 MB. The use of huge 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 given area of memory. With fewer TLB entries, the occurrence of TLB misses is reduced. Second, the page table setup for mapping huge pages uses one fewer table to determine the physical address, making virtual-to-physical-address mapping faster than for normal pages.

Huge 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 huge pages for the WebSphere JVM heap. As we discussed in the preceding section, the most optimal JVM heap size for Web service mediation scenarios was 6 GB. To be on the safe side, we configured 8 GB worth of huge pages (8 GB / 2 MB per huge page = 4096 huge pages). We set the amount of shared memory to 10 GB so that there would be enough room for the 8 GB of huge pages in the shared memory pool (10 GB is 10737418240 bytes). We added the following lines to the **`/etc/sysctl.conf`** file:

```
vm.nr_hugepages = 4096
kernel.shmmax = 10737418240
kernel.shmall = 10737418240
```

The WebSphere 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.

The data in *Figure 4.1.1* shows that the huge page support did not result in any significant performance improvement for transformation mediation scenarios with 3-KB payload size, although it did yield a 16% performance gain for route-on-body service mediation. This is rather expected because the transformation operations on 3-KB messages are not expected to cross normal 4-KB page boundaries too frequently. As we will see later, with payload sizes larger than 3 KB, the huge page support did provide some noticeable performance advantage. Route-on-body service mediations involve fetching certain values in the message bodies, and then, based on those values, jumping to different web services in memory at comparatively higher service request rates (more than 2,000 requests per second). This would cause more cache and TLB misses than for other mediation types, so huge page support would help here – providing a 16% performance improvement for routing mediation scenarios.

4.1.1.3. Tuning for Maximum Concurrency

After optimizing the JVM heap memory usage, the next area for potential performance improvement is concurrency. There are several WebSphere Web Container settings that can be tweaked for maximum concurrency in our tests. If there are not enough HTTP connections available, incoming service requests 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, we found that setting the maximum number of HTTP persistent requests to **`unlimited`** gave us a noticeable performance gain.

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"

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.

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

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 server since the more CPUs we have, the more threads can be executed concurrently. The x3850 M2 server has 4 quad-core processors, so that's 16 processor cores that can work in parallel. To ensure that we have enough available threads in the thread pools, we increased 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).

To further optimize the performance of Web service mediation scenarios in our tests, we 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 WebSphere 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 4.1.1 shows the cumulative impact of increasing the maximum number of persistent HTTP connections, the maximum number of threads in the default and Web Container thread pools, as well as disabling all performance monitoring, tracing, and logging. These tweaks provided the most performance gain (9%) over what we were able to achieve with optimal JVM heap configuration and huge page support for the Transform Value mediation scenario. The Transform Value mediation scenario was the least compute-intensive of all mediation scenarios considered in our study, and therefore, it could sustain the highest service request rates. As a result, setting up for maximum concurrency resulted in a significant performance boost for this mediation scenario. In contrast, the other Transform mediation scenarios were much more compute-intensive, and therefore, could only handle relatively low service request rates, so tuning for maximum concurrency did not help much.

4.1.2. Impact of Large Payload Sizes

Let us now look at the performance of the Web service mediation scenarios with payload (message) sizes greater than 3 KB. *Figures 4.1.2.1* through *4.1.2.5* show the impact of the tuning items that we discussed in *Section 4.1.1* on these mediation scenarios with payload sizes greater than 3 KB.

As payload size increased, the Web service rates (throughput) dropped – as expected. The size of the drop in throughput depends on the amount of processing that is required to perform the mediation on the payload in the service requests and responses. For example, *Figure 4.1.2.1* shows that the Composite Module mediation scenario posted approximately the same throughput for both {Request = 3KB, Response = 10 KB} and {Request = 10 KB, Response = 3 KB} configurations because the same amount of mediation processing was done on both requests and responses. On the other hand, the fact that the Transform Value mediation scenario posted approximately the same throughput for both {Request = 10KB, Response = 3KB} and {Request = 10KB, Response = 10KB} configurations, as shown in *Figure 4.1.2.2*, indicates that the Transform Value mediation was only performed on requests (not responses) because the difference in the response payload size did not affect throughput. The relatively high throughput for the Transform Value and Route-On-Body mediation scenarios indicates that the amount of processing required to perform these mediations was less than the processing required by the other mediations.

It is important to note that the default WebSphere settings did not have enough capacity to even support payload sizes greater than 3 KB (all scenarios would fail after some time). Setting the initial and maximum JVM heap sizes to 1024 MB was the minimal step that would allow all mediation scenarios to complete successfully. Indeed, the JVM heap optimizations discussed in *Section 4.1.1* accounted for the largest performance improvement across all mediation scenarios – regardless of payload sizes.

For many mediation scenarios, such as Composite Module and Transform Schema, the Linux huge page support provided larger performance gains at payload sizes greater than 3 KB. This is expected because performing transformations on payloads larger than 3 KB would cross normal 4-KB page boundaries more frequently, so huge pages would help in these cases.

Concurrency tuning, which includes setting the maximum persistent HTTP connections to **`unlimited,`** increasing the sizes of the default and Web Container thread pools, as well as disabling all performance monitoring, tracing, and logging, provided larger benefits at higher throughput. In fact, based on the data in *Figures 4.1.2.1* through *4.1.2.5*, it appears that concurrency tuning only provided noticeable performance gains at service request rates higher than 600 requests per second.

Figure 4.1.2.6 shows the optimized performance for transformation mediation scenarios at different payload sizes. As the payload sizes increased, the Web service request rates dropped due to larger computational requirements for performing transformation and routing mediations on larger payloads. In fact, as we increased the payload size to 100 KB for both requests and responses, the 16 processor cores on the x3850 M2 server started to become the limiting factor: the CPU utilization increased past 90% – with the exception of the Transform Value mediation scenario (shown in *Figure 4.1.2.7)*. The Transform Value scenario required the least amount of computational power as it was the simplest among the mediation scenarios considered in our study, and therefore, it could sustain relatively high service request rates. In this scenario, with 100-KB payload in both requests and responses, the amount of network traffic and latency – not the CPUs – quickly became the limiting factor: the network traffic between the client and the server was measured approaching 500 Mbps, limiting the server CPU utilization to less than 80%.

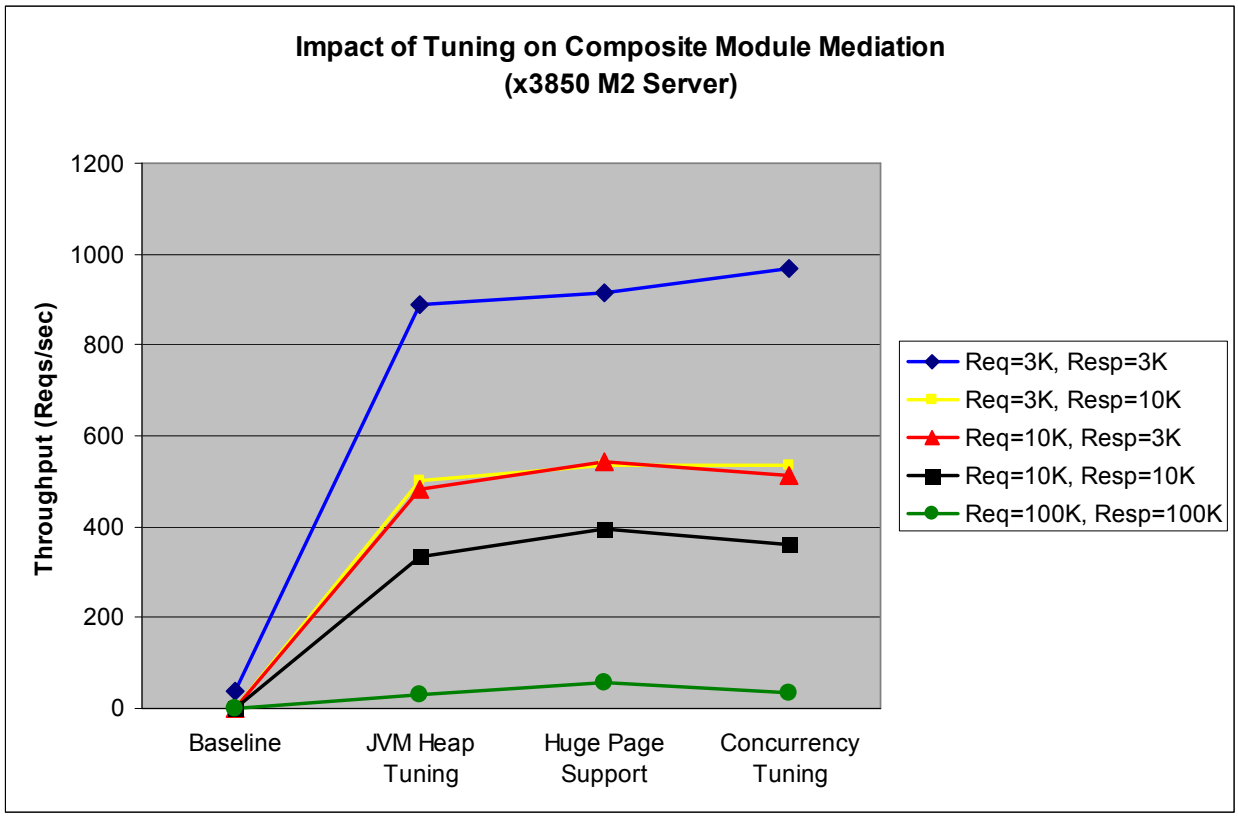

Figure 4.1.2.1 – Impact of tuning on Composite Module mediation scenario with different workload sizes

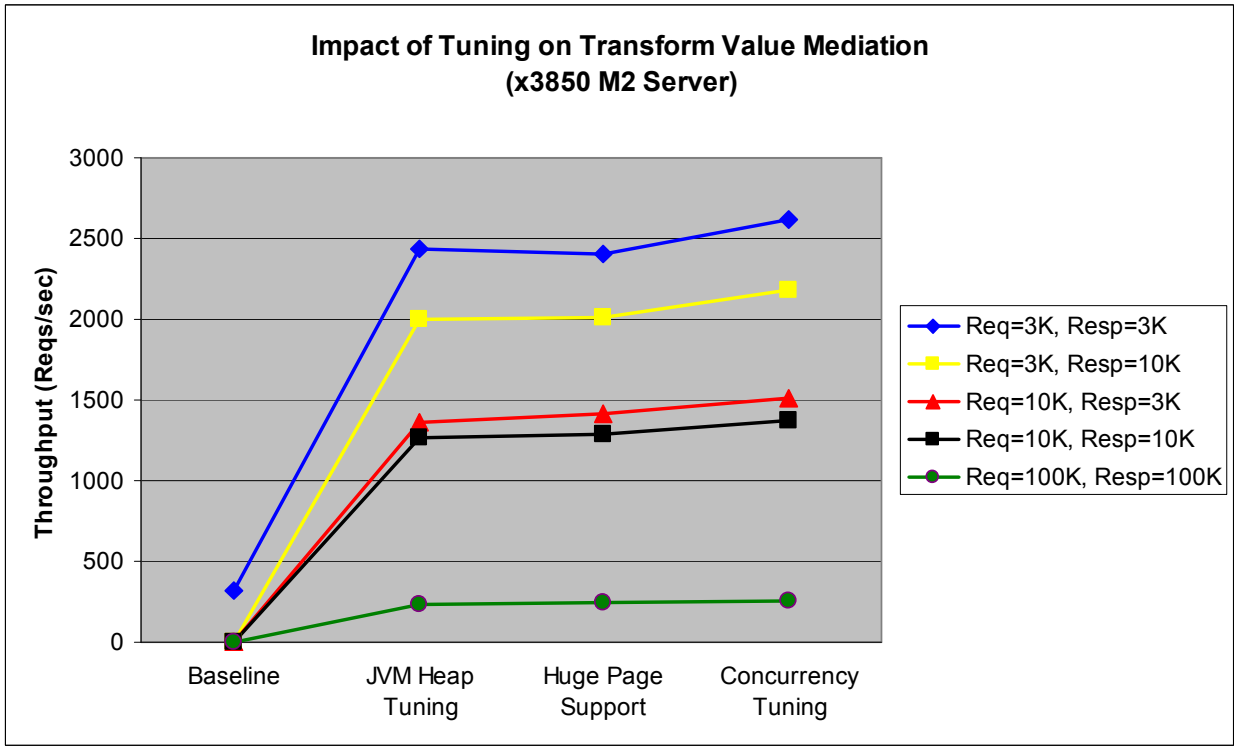

Figure 4.1.2.2 – Impact of tuning on Transform Value Mediation with different workload sizes

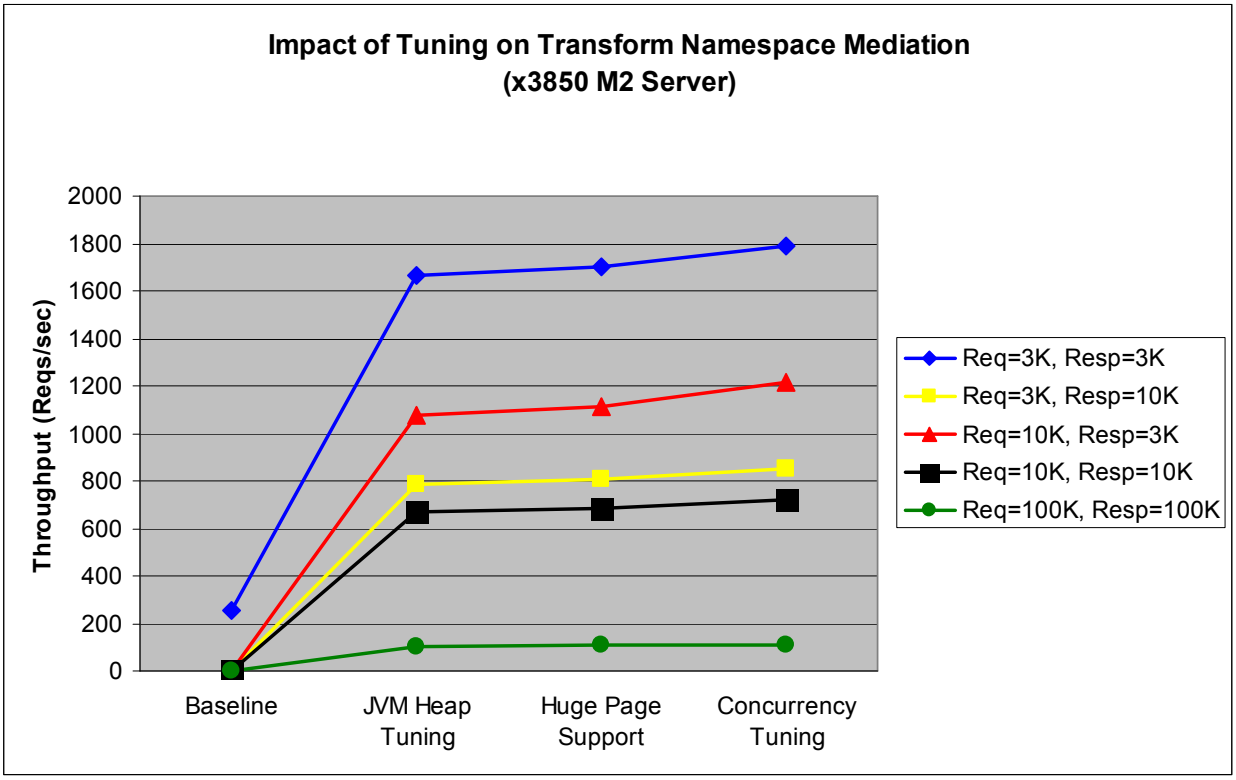

Figure 4.1.2.3 – Impact of tuning on Transform Namespace Mediation with different workload sizes

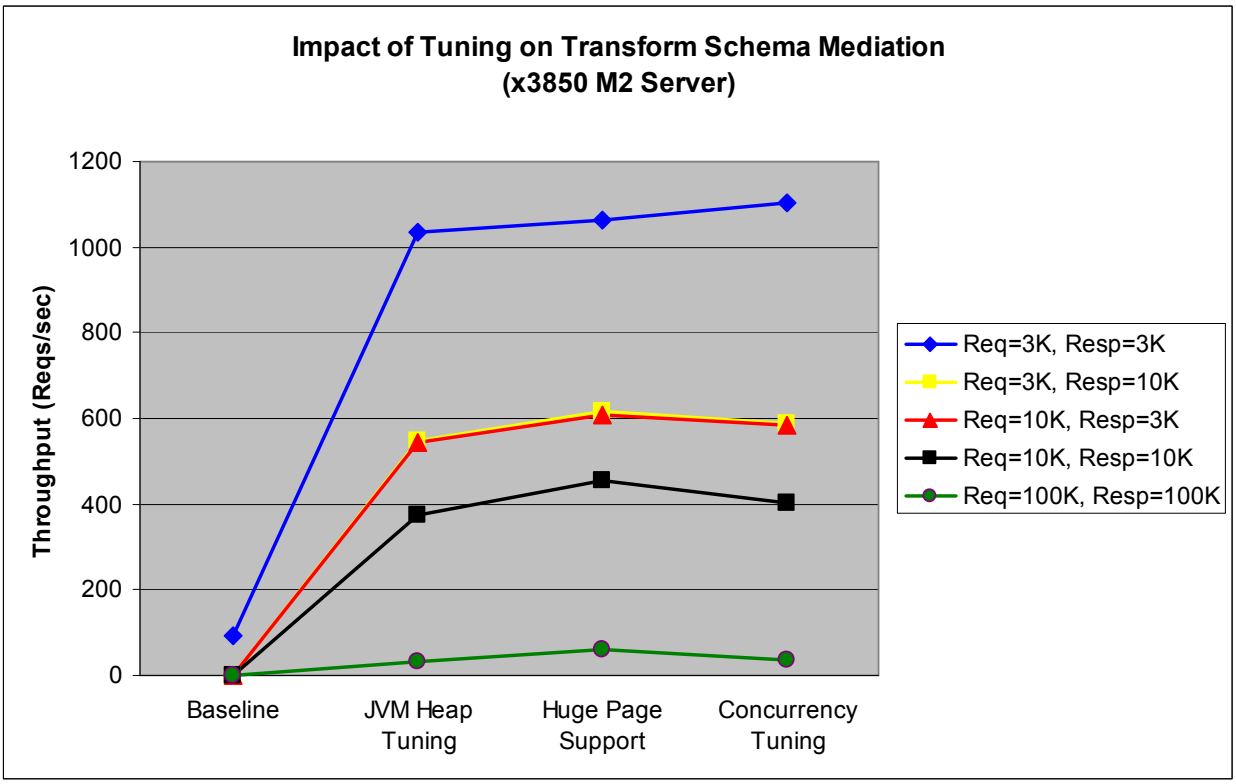

Figure 4.1.2.4 – Impact of tuning on Transform Schema Mediation with different workload sizes

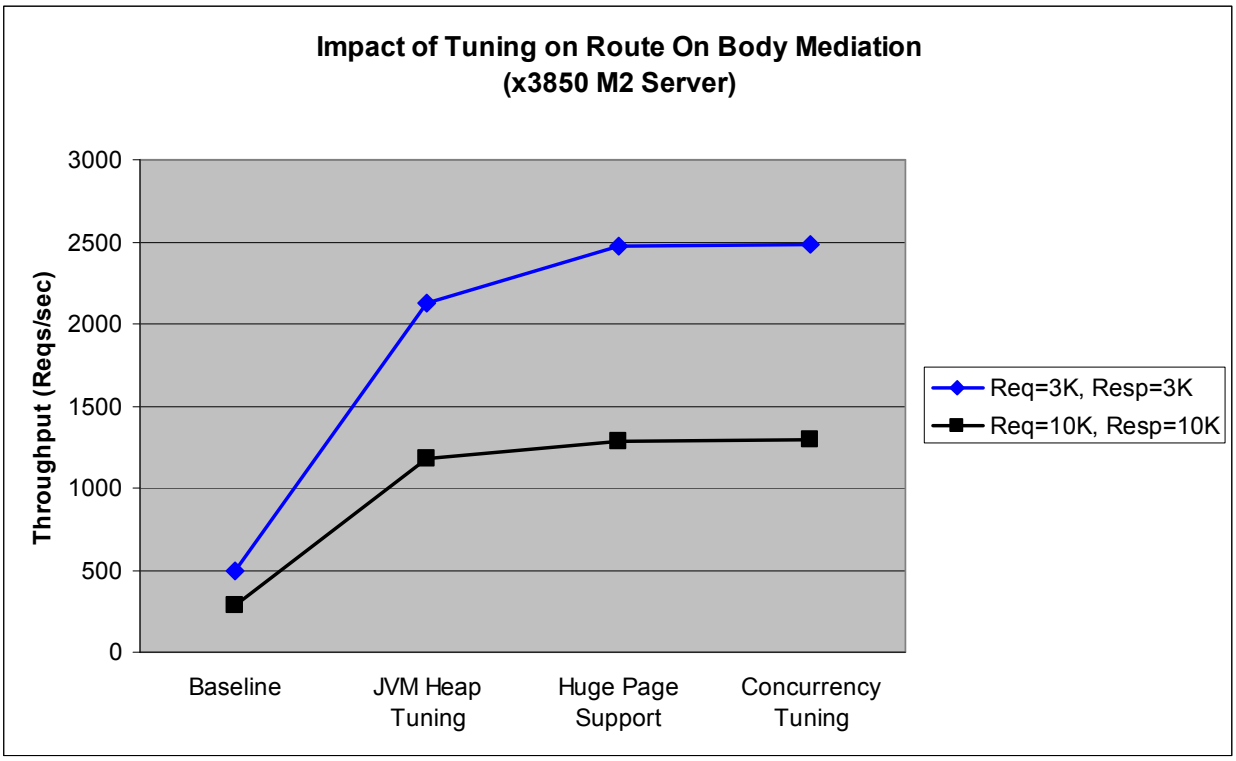

Figure 4.1.2.5 – Impact of tuning on Route On Body Mediation with different workload sizes

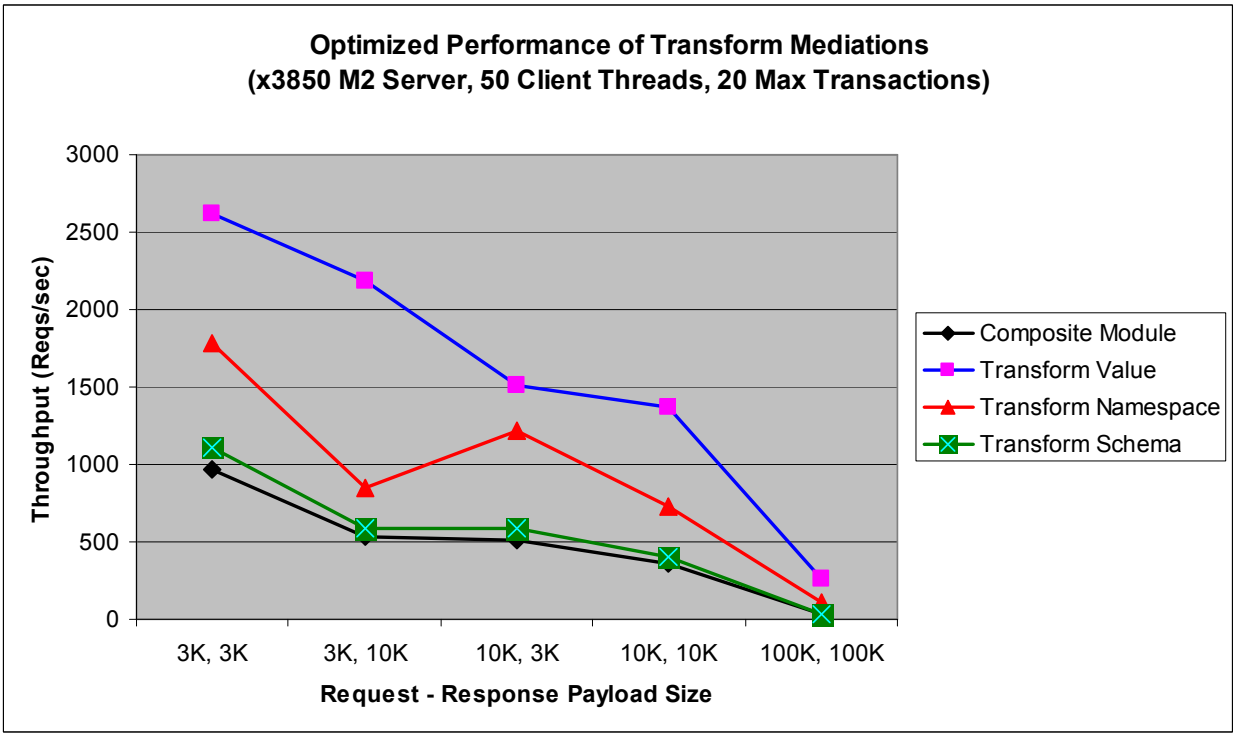

Figure 4.1.2.6 – Optimized performance of transformation mediations across different payload sizes

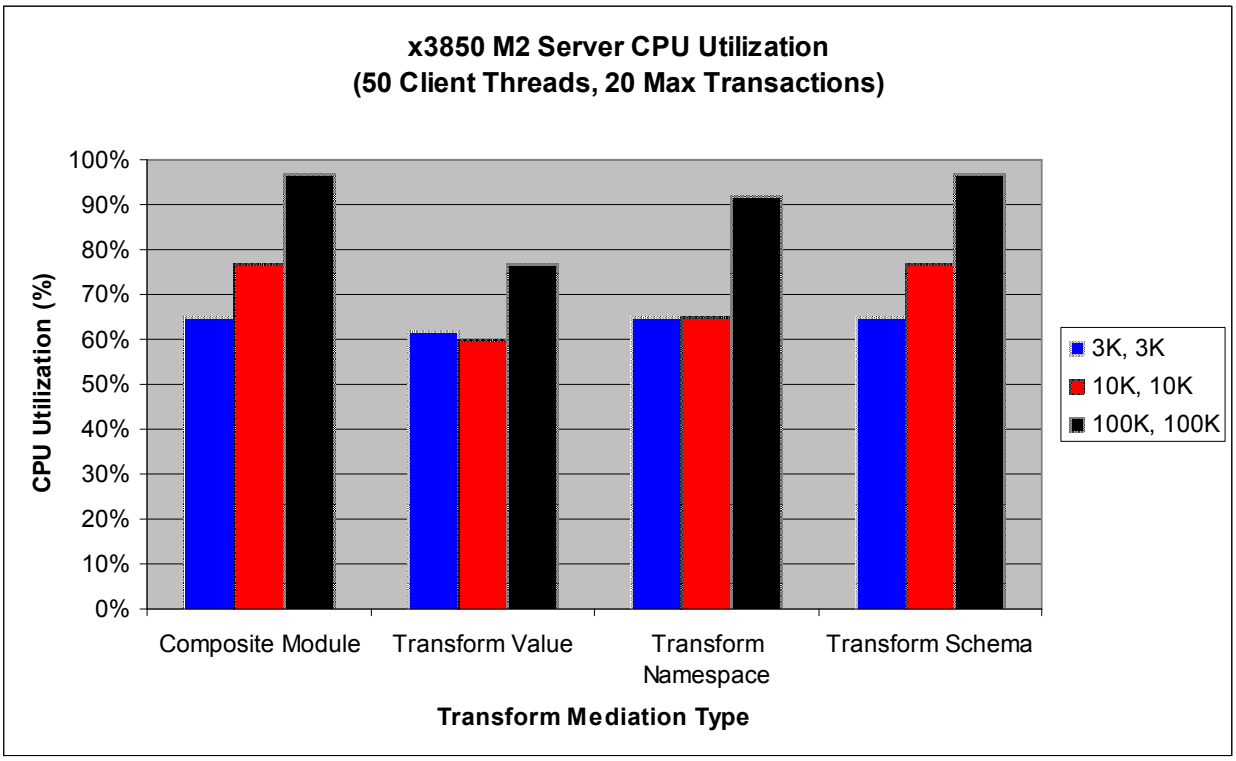

Figure 4.1.2.6 – Server CPU utilization for transformation mediations at different payload sizes

4.1.3. Summary for Results on the IBM System x3850 M2 Server

On the IBM System x3850 M2 server, the default WebSphere and Linux settings could only support 3-KB payload size for service requests and responses in the Web service mediation scenarios considered in our study. Setting the WebSphere JVM heap size to 6144 MB and using the Generational Concurrent (`gencon`) garbage collection policy resulted in more than 23X performance improvement. The Linux huge page support provided another 16% performance gain for routing mediations. For transformation mediations, however, the huge page support did not yield any noticeable performance gain with payload size of 3 KB or less. At larger payload sizes, the huge page support did yield a small, but noticeable, performance improvement for relatively complex mediation scenarios, such as Composite Module and Transform Schema, where mediation operations on large payloads often cross the normal 4-KB page boundaries. The concurrency tuning only provided noticeable performance gains at service request rates greater than 600 requests per second.

4.2. Results on the IBM Power 570 Server

As with the x3850 M2 server, we found that WebSphere JVM heap optimization, Linux huge page support, and tuning for maximum concurrency yielded noticeable performance gains. Let us first look at the service mediation performance with the request and response payload size set to 3 KB.

4.2.1. Service Mediation Performance with 3-KB Payloads

Figure 4.2.1 shows the performance gain achieved by each of the tuning items for various Web service mediation scenarios with payload size of 3 KB.

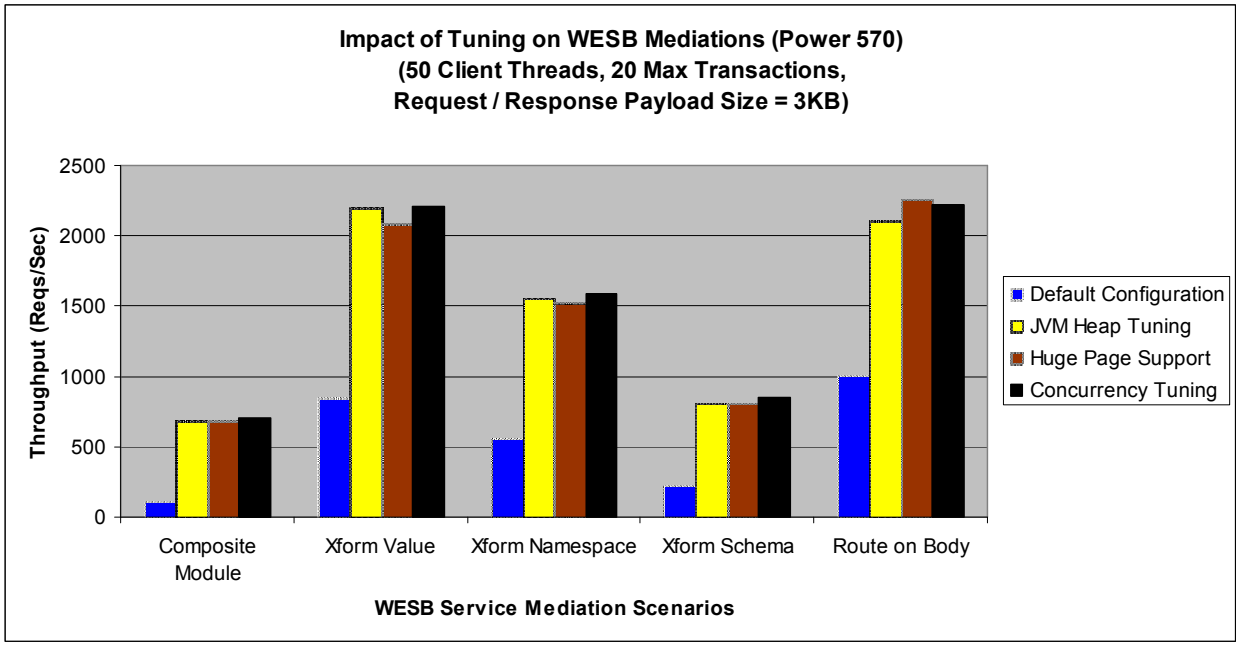

Figure 4.2.1 – Optimized performance of transform mediations across different payload sizes

Let us first look at the WebSphere JVM heap optimizations.

4.2.1.1. WebSphere JVM Heap Tuning

As we found in the case with the x3850 M2 server, the JVM heap configuration had a significant impact on the performance of Web service mediation scenarios on the Power 570 server. We followed the same tuning approach as discussed in *Section 4.1.1.1*. The data for the Power 570 server is shown in *Table 4.2.1.1*.

GC Policy	JVM Heap Size (MB)	Throughput (Reqs/sec)	Overall GC Overhead	Total GC Pause
Default (optthruput)	2048	664.26	5%	11 secs
	3072	702.57	5%	8 secs
	4096	688.94	4%	6 secs
	6144	693.48	4%	6 secs
Gencon (nursery = 75% heap)	3072	712.96	2%	4 secs
	4096	711.51	2%	3 secs
optavgpause	3072	670.41	2%	4 secs
subpool	3072	698.88	5%	8 secs

Table 4.2.1.1 – Impact of tuning on Composite Module mediation scenario with payload size = 3 KB

As discussed in *Section 4.1.1.1*, in order to change the WebSphere JVM heap size or set the garbage collection policy, we can use the WebSphere Application Server (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)
- Enter appropriate JVM command line option for the garbage collection policy (as shown in the middle column of Table 4.1.1.1) into the "Generic JVM arguments" box. For example, if we want to use the **`gencon`** garbage collection policy with a nursery size of 1536 MB, we would enter "**`-Xgcpolicy:gencon -Xmn1536M`**" into the "Generic JVM arguments" box.

With payload size for both requests and responses set to 3 KB and with the default WebSphere JVM settings, we could only achieve an average Web service request rate of 111.03 requests per second in the Composite Module mediation scenario. To optimize the JVM heap, we first considered the default GC policy (**optthruput**). By setting both the initial JVM heap size and the maximum heap size to 2048 MB, we immediately got a significant performance boost to 664.26 requests per second – almost 6X improvement! The overall garbage collection overhead was 5%, which is quite acceptable.

Increasing the JVM heap size to 3072 MB yielded another 6% performance improvement. However, increasing the JVM heap size to 4096 MB and beyond did not seem to make any difference. As a result, we decided to keep the heap size at 3072 MB, but tried to see if we could optimize the heap usage better by switching to another garbage collection policy.

The WebSphere JVM supports four different garbage collection policies, as shown in *Table 4.1.1.1.1*. Please refer to *Section 4.1.1.1* for a more detailed description of these garbage collection policies. With the JVM heap size kept at 3072 MB, the Generational Concurrent (**gencon**) garbage collection policy delivered the best performance for the Composite Module mediation scenario – as shown in *Table 4.2.1.1*. The **optavgpause** policy delivered the worst performance. The **subpool** policy, which is available only on IBM System p and z servers, yielded approximately the same performance as the default policy (**optthruput**), but not as good as the **gencon** policy.

Based on the data in *Table 4.2.1.1*., we can conclude that the **gencon** garbage collection policy, together with the JVM heap size of 3072 MB, represent the most optimal JVM heap configuration for the Composite Module mediation scenario (with payload size = 3KB for both requests and responses). In fact, as *Figure 4.2.1* shows, this optimal WebSphere JVM heap configuration accounted for the largest performance gain across all Web service mediation scenarios considered in our study.

4.2.1.2. Huge Page Support in Linux

Since the WebSphere JVM heap size was set to 3072 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, which cannot be demand-paged from a disk file or swapped out to the swap partition.

To allow the WebSphere 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 4-GB pool of huge pages by adding the following lines to the **/etc/sysctl.conf** file:

```
#Number of huge pages (256 x 16 MB = 4 GB)
vm.nr_hugepages = 256
#Size of shared memory is set to 6 GB (6442450944 bytes)
kernel.shmmax = 6442450944
kernel.shmall = 6442450944
```

(Note that we set the amount of shared memory to 6 GB so that there would be enough room for 4-GB pool of huge pages in the shared memory pool.)

The WebSphere 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.

The data in *Figure 4.2.1* shows that the huge page support did not result in any significant performance improvement for the transformation mediation scenarios with 3-KB payload size, although it did yield a noticeable (8%) performance gain for the routing mediation scenario. This is rather expected as transformation operations on 3-KB payloads are not expected to cross normal 4-KB page boundaries too frequently. On the other hand, route-on-body service mediations involve fetching certain values in the message bodies, and then, based on those values, jumping to different web services in memory at

comparatively higher service request rates (more than 2,000 requests per second). This would cause more cache and TLB misses than for other mediation types, so huge page support would help here – providing an 8% performance improvement for routing mediation scenarios.

4.2.1.3. Tuning for Maximum Concurrency

As in the case with the x3850 M2 server, after optimizing the WebSphere JVM heap usage, the next area for potential performance improvement is concurrency. As discussed in *Section 4.1.1.3*, we focused on the following:

- Setting the maximum number of persistent HTTP connections to **`unlimited`**.
- Increasing the maximum number of threads in the default and Web Container thread pools. The Power 570 server has 2 dual-core POWER6 processors with SMT (Simultaneous Multi-Threading) enabled, so that's 8 logical processors working in parallel. To ensure that we had enough available threads in the thread pools, we increased 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).
- Disabling the WebSphere Performance Monitoring Infrastructure (PMI).

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"

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

Disabling the WebSphere 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 4.2.1 shows the cumulative impact of increasing the maximum number of persistent HTTP connections, the maximum number of threads in the default and Web Container thread pools, as well as disabling PMI. These tweaks provided the most performance gain for the Transform Value mediation scenario – 6% gain over what we were able to achieve with optimal JVM heap configuration and huge page support. The Transform Value mediation scenario was the least compute-intensive of all scenarios considered in our study, and therefore, could sustain relatively high service request rates, so setting up for maximum concurrency resulted in a sizable performance boost. In contrast, the other transformation mediation scenarios were much more compute-intensive and had relatively low service request rates, so tuning for maximum concurrency did not help much.

4.2.2. Impact of Large Payload Sizes

Let us now look at the performance of the Web service mediation scenarios running on the Power 570 server with payload (message) sizes greater than 3 KB. *Figures 4.2.2.1* through *4.2.2.5* show the impact of the tuning items that we discussed in *Section 4.2.1* on the Web service mediation scenarios with payload sizes greater than 3 KB. It is important to note that the default WebSphere settings did not even

have enough capacity to support payload sizes greater than 3 KB (all scenarios would fail after some time).

As payload size increased, the Web service rates (throughput) dropped – as expected. The size of the drop in throughput depends on the amount of processing that is required to perform the mediation on the payload in the service requests and responses. For example, *Figure 4.2.2.1* shows that the Composite Module mediation scenario posted approximately the same throughput for both {Request = 3KB, Response = 10 KB} and {Request = 10 KB, Response = 3 KB} configurations because the same amount of mediation processing was done on both requests and responses. On the other hand, the fact that the Transform Value mediation scenario posted approximately the same throughput for both {Request = 10KB, Response = 3KB} and {Request = 10KB, Response = 10KB} configurations, as shown in *Figure 4.2.2.2*, indicates that the Transform Value mediation was only performed on requests (not responses) because the difference in response payload size did not affect throughput. The relatively high throughput for the Transform Value and Route-On-Body mediation scenarios indicates that the amount of processing required to perform these mediations was less than the processing required by the other mediations.

As in the case with the x3850 M2 server, the JVM heap optimizations discussed in *Section 4.2.1* accounted for the largest performance gain across all mediation scenarios – regardless of payload sizes – on the Power 570 server. The huge page support does not appear to provide any significant performance benefit on the Power 570 server – except for the route-on-body mediation scenario. Tuning for maximum concurrency, which includes setting the maximum persistent HTTP connections to **`unlimited`**, increasing the sizes of the default and Web Container thread pools, as well as disabling PMI, provided an average of 6% performance improvement on top of what we were able to achieve with optimal JVM heap configuration and huge page support. However, as shown in *Figure 4.2.2.7*, the CPU utilization on the Power 570 server was approaching 99% as we increased the payload size to 100 KB, so the server CPUs became the limiting factor, preventing us from obtaining higher throughput.

Figure 4.2.2.6 shows the optimized performance for transformation mediation scenarios at different payload sizes. As the payload sizes increased, the Web service request rates dropped due to larger computational requirements for performing transformation and routing mediations on larger payloads.

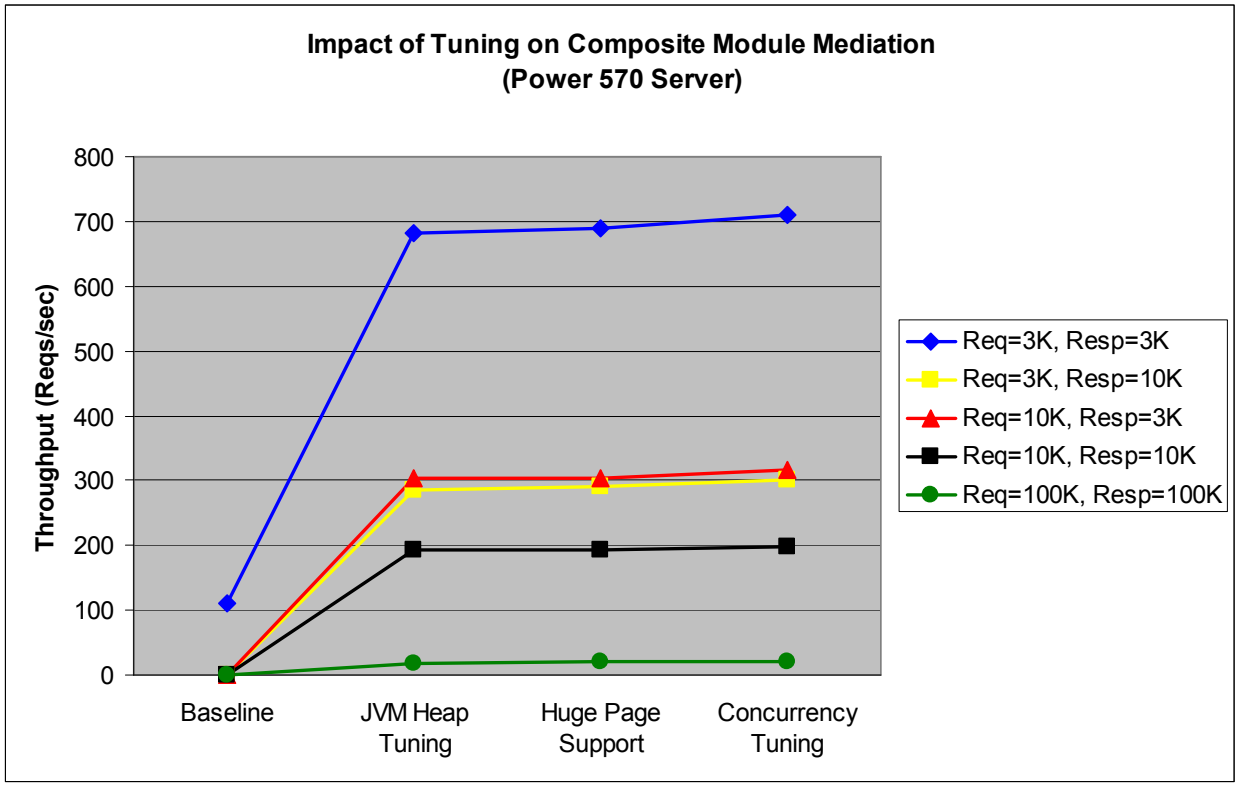

Figure 4.2.2.1 – Impact of tuning on Composite Module mediation at different payload sizes

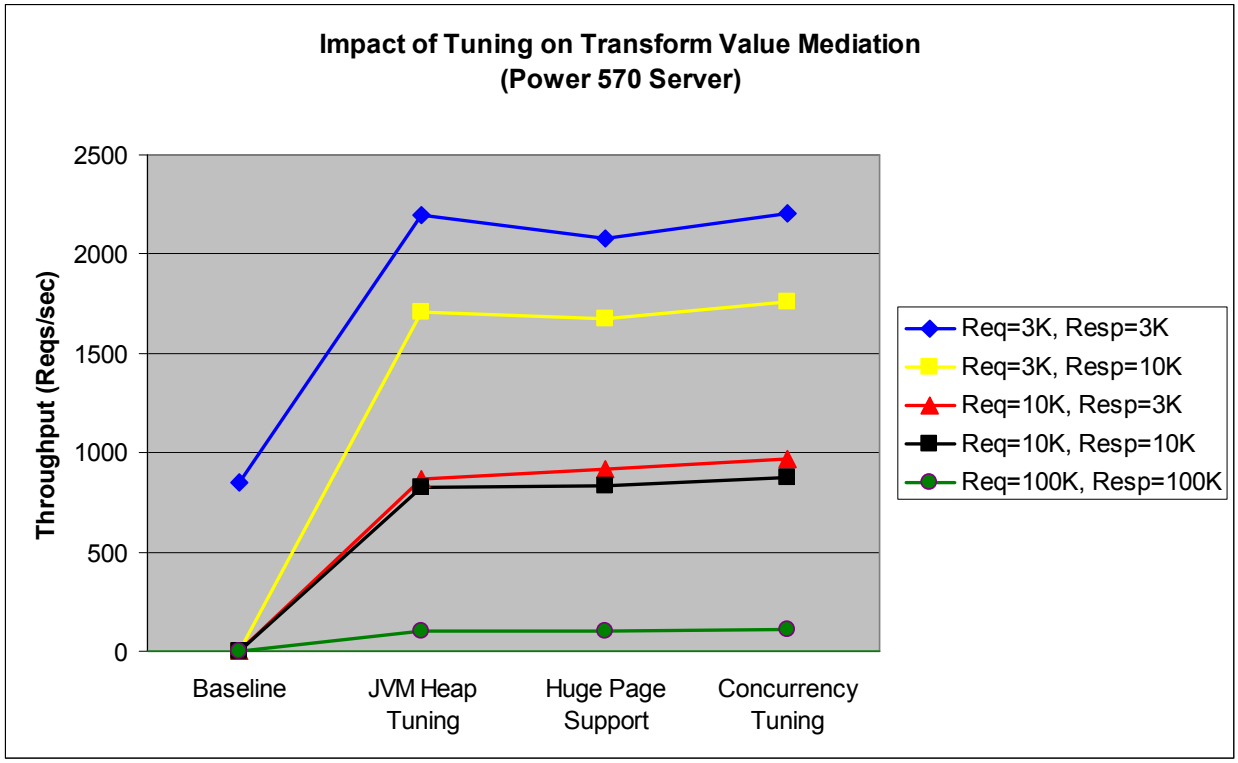

Figure 4.2.2.2 – Impact of tuning on Transform Value mediation at different payload sizes

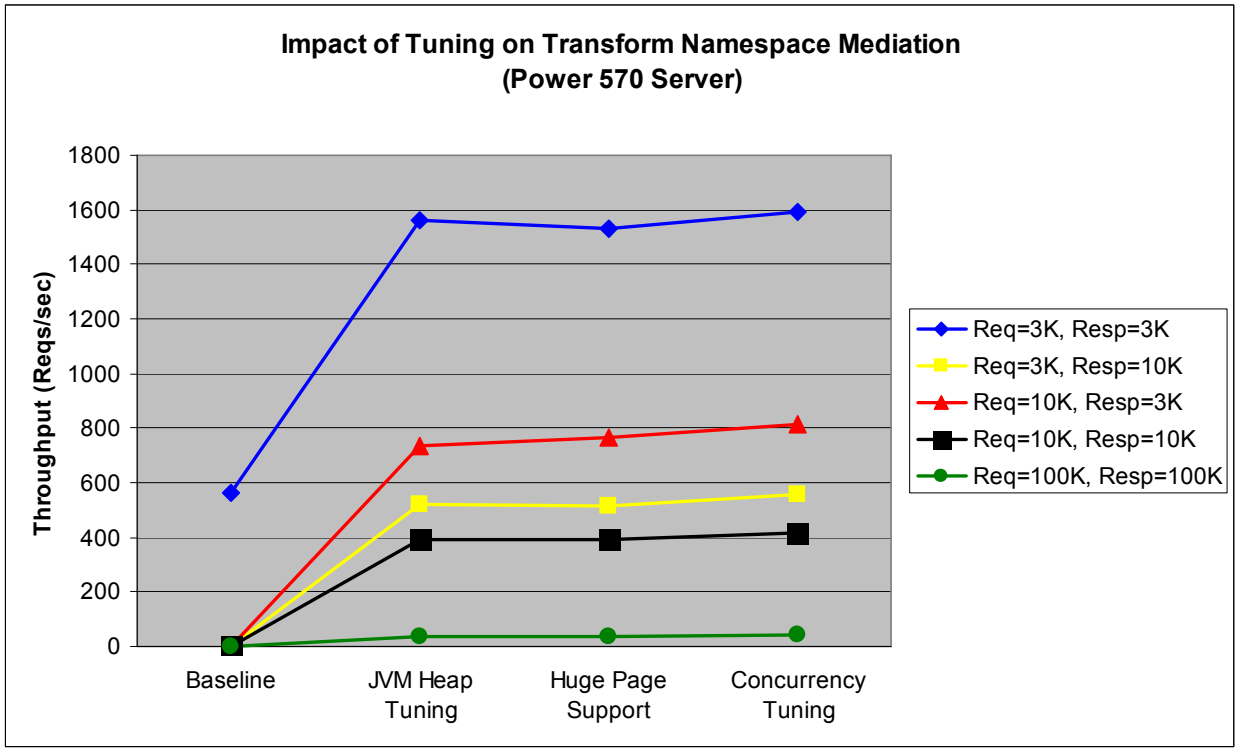

Figure 4.2.2.3 – Impact of tuning on Transform Namespace mediation at different payload sizes

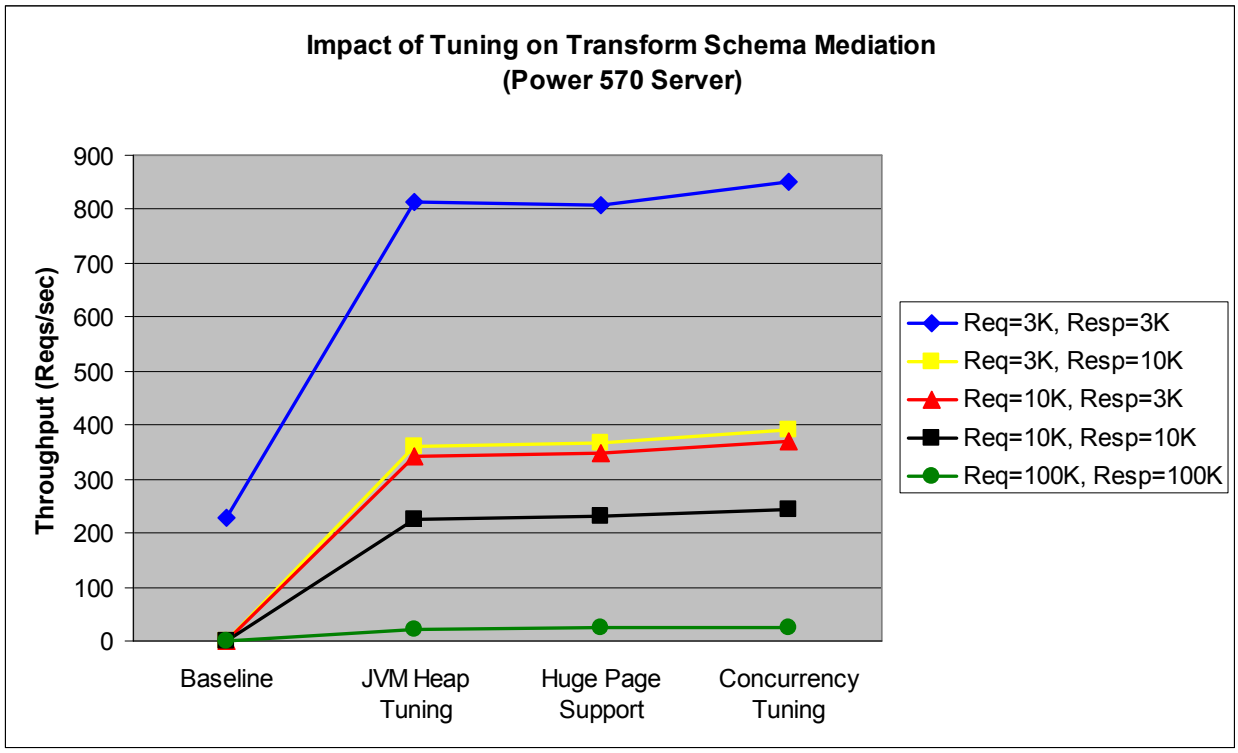

Figure 4.2.2.4 – Impact of tuning on Transform Schema mediation at different payload sizes

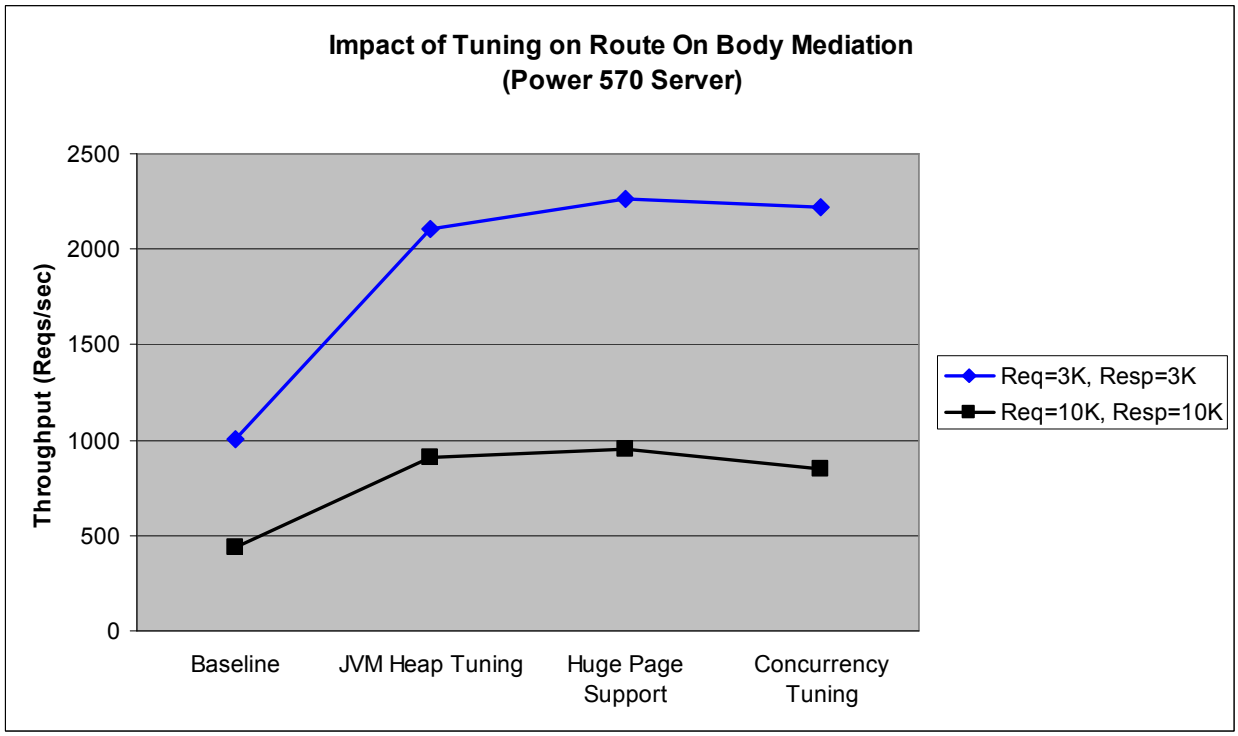

Figure 4.2.2.5 – Impact of tuning on Route-On-Body mediation at different payload sizes

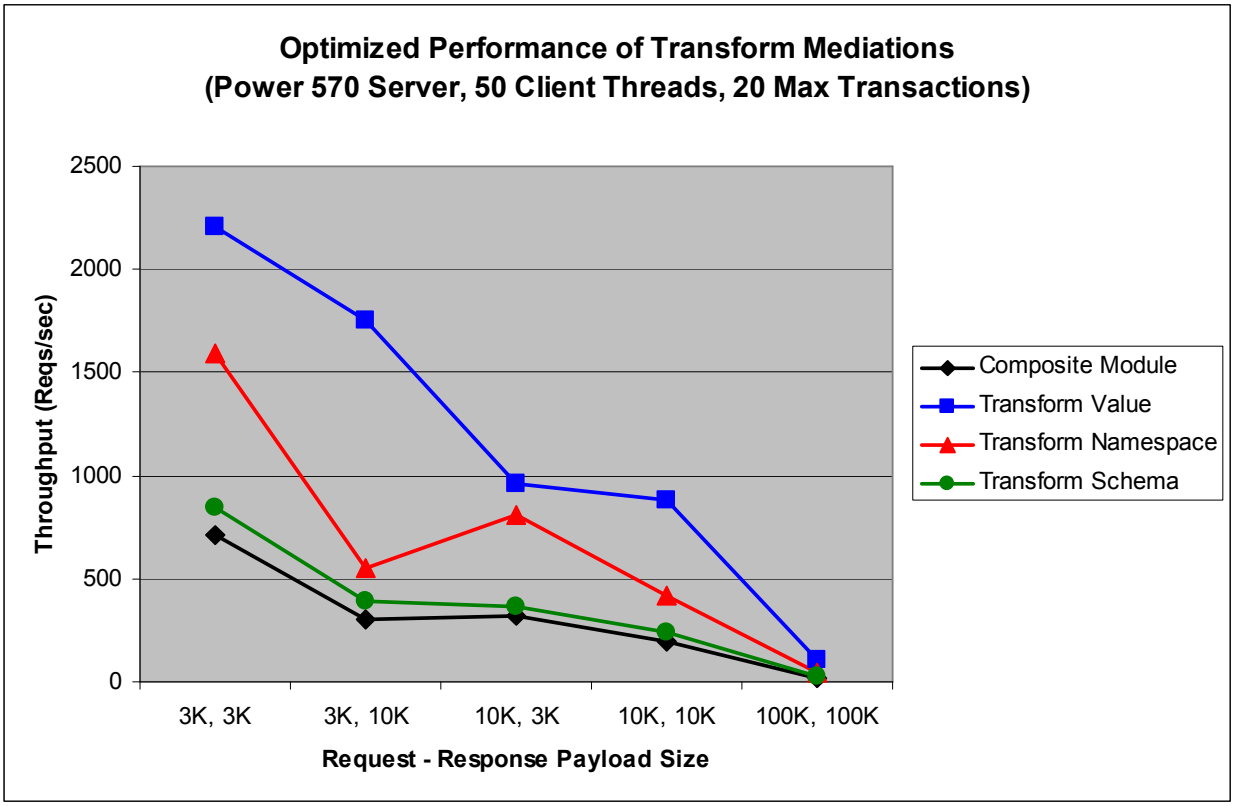

Figure 4.2.2.6 – Optimized performance for transformation mediation scenarios across payload size

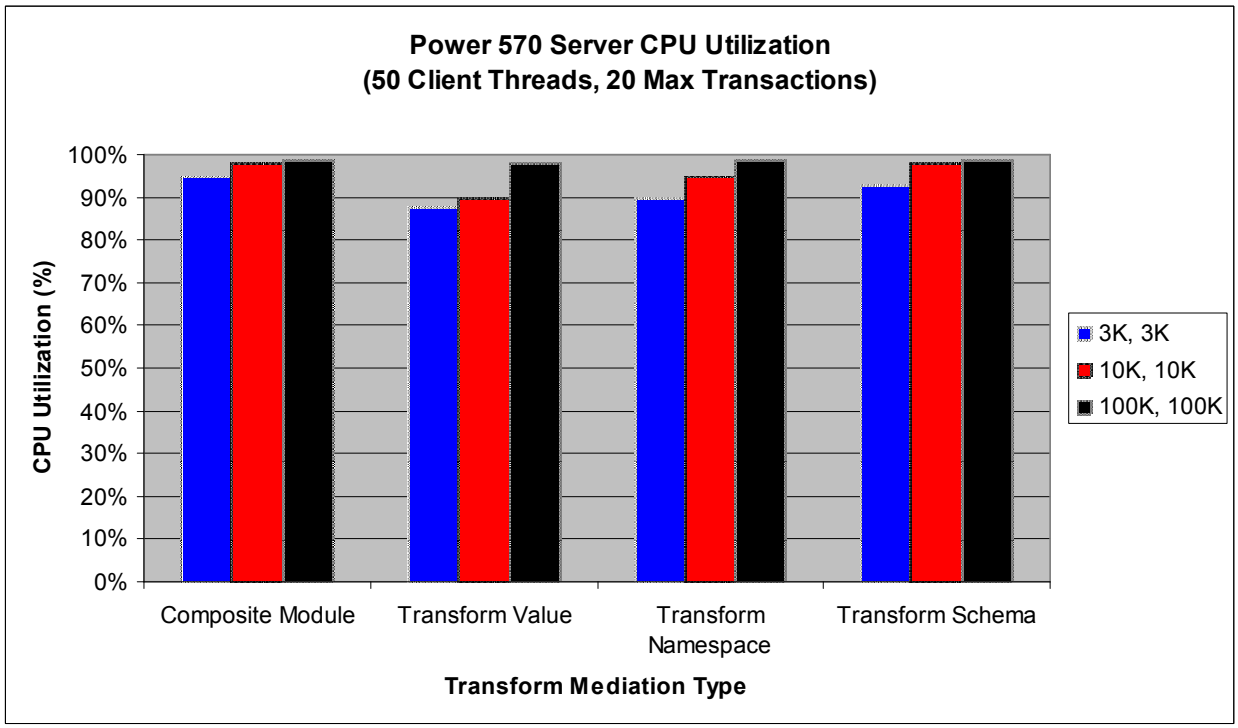

Figure 4.2.2.7 – Server CPU utilization for transformation mediations at different payload sizes

4.2.3. Summary of Results on the IBM Power 570 Server

On the Power 570 server, the default WebSphere and Linux settings could only support 3-KB payloads in the service requests and responses for the Web service mediation scenarios considered in our study. Setting the WebSphere JVM heap size to 3072 MB and using the Generational Concurrent (**`gencon`**) garbage collection policy resulted in more than 6X performance improvement. The Linux huge page support provided another 8% performance boost for routing mediations, and the concurrency tuning yielded an average of 6% performance improvement for all mediation scenarios across different payload sizes. However, as we increased the payload size to 100 KB, the CPU utilization on the Power 570 server was approaching 99%, so the server CPUs became the limiting factor, preventing us from obtaining higher throughput.

5. Conclusions

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

- For transformation and routing service mediations, the default WebSphere and Linux settings could only support payload size of 3 KB or less for service requests and responses.

- Optimizing the WebSphere JVM heap configuration had the greatest performance impact – up to 23X improvement on the x3850 M2 server. It is therefore very important to have the JVM heap sized appropriately for the workload. Generational Concurrent (**`gencon`**) garbage collection policy appears to be the most suitable for Web service mediations.

- For routing mediations, the huge page support in Linux provided 16% and 8% performance gains on the x3850 M2 and Power 570 servers, respectively. For transformation mediations, however, the huge page support did not yield any noticeable performance improvement with payload size of 3 KB or less. At larger payload sizes, there was a small, but noticeable, performance improvement for relatively complex mediation scenarios, such as Composite Module and Transform Schema, where mediation operations on large payloads often cross the normal 4-KB page boundaries.

- On the x3850 M2 server, tuning for maximum concurrency only delivered noticeable performance gain when the service request rate was relatively high (more than 600 requests per second). On the Power 570 server, only the Transform Value mediation scenario could sustain a high enough service request rate for concurrency tuning to yield a small (6%) performance improvement; the remaining mediation scenarios did not see any performance gain. (Concurrency tuning includes setting the maximum number of persistent HTTP connections to **`unlimited`**, increasing the sizes of the default and Web Container thread pools, as well as disabling all performance monitoring, tracing, and logging.)

- As the payload size in the service request and response messages increased, the Web service rates (throughput) dropped – as expected. The size of the drop in throughput depends on the amount of processing that is required to perform the mediation on the message payloads. On the x3850 M2 server, as we increased the payload size to 100 KB for both request and response messages, the server CPU utilization increased past 90%, where the server CPUs started to become a performance bottleneck, for all WESB service mediation scenarios – with the exception of the Transform Value scenario. In the Transform Value scenario, the network traffic and latency became the limiting factor before the server CPUs because of the high rate of service requests and responses, each with a large 100-KB payload. However, on the Power 570 server, the server CPU utilization was always very high, and as the payload size increased to 100 KB, the server CPU became the absolute bottleneck, preventing us from getting higher throughput.

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

Produced in the USA
06-08

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.

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.