

Middleware Architecture for Hardware-Efficient Scaling

# Higher Hardware Performance via High Concurrency

Transactional Cascade® Technology Paper

Ivan Klianov, Managing Director & CTO  
Transactum Pty Limited



Published in June 2005 | Updated in March 2011

Copyright © 2005-2011 Transactum  
All rights reserved

## Abstract

Response time and throughput are the performance criteria of a computer system. Scaling the software operations with no hardware additions might, but not straightforwardly will result in higher performance as scaling normally leads to formation of performance bottlenecks. Efficient scaling requires fitting the specific way of expansion of software operations within the specifics of execution environment in a way delaying the formation of bottlenecks. The operating system and the virtual machine cannot obtain the necessary design and runtime information to perform this task efficiently. Apparently, only the middleware knows the details of its own scaling techniques and how to optimize at design and runtime the use of CPU caches, how to prevent the OS from being overloaded with unnecessary context switching, and how to influence the context switching order in a way enhancing the application performance. This mission is virtually impossible if the middleware runs in a virtual execution environment; otherwise, it is realistic and achievable.

## Table of Contents

|   |    |
|---|----|
| <b>Abstract</b> .....   | 2  |
| <b>Table of Contents</b> .....                                  | 2  |
| <b>Executive Summary</b> .....                                  | 3  |
| <b>1. Performance of Application Server</b> .....               | 4  |
| <b>2. Scaling Contentions and Performance Bottlenecks</b> ..... | 5  |
| <b>3. Techniques for Scaling Efficiency</b> .....               | 6  |
| <b>4. Improving the Performance of Application Server</b> ..... | 8  |
| <b>5. Performance with Transactional Cascade®</b> .....         | 9  |
| <b>6. Conclusion</b> .....                                      | 12 |

## Executive Summary

Response time and throughput are the performance criteria of a computer system. Both are correlated in a way that scaling the throughput increases the response time. Under a constant speed of all elements of a computer system, scaling the throughput adds more parallel elements and creates more work for each sequential element. The higher workload of sequential elements increases the response time. Any performance improvement is measured either as throughput maximization coupled with acceptable response time, or as response time minimization under constant throughput.

Scaling the software operations of a computer system with no hardware additions increases the performance of hardware executing the scaled software. Scaling a complex process requires increasing the volume of its running instances. The higher number of concurrent process instances on architecture with blocking IO (or with non-blocking IO and CPU-intensive processing) requires higher volume of threads. Running high volume of threads increases the contention for shared resources, which results into excessive context switching.

Excessive context switching might deteriorate the overall performance. A context switching causes processor L1 and L2 caches to be flushed and refilled and some memory lost. Moreover, the coherency of cache has a performance cost. A processor with invalidated cache line will have a cache miss and a delay the next time the line is addressed. Apparently, scaling requires fitting the specific way of expansion of software operations within the limits of execution environment, and particularly within the specifics of multiprocessor architecture.

The operating system and the virtual machine cannot obtain the necessary design and runtime information to perform this task efficiently. Only the middleware knows the details of its own scaling techniques and how to optimize at design and runtime the use of CPU caches, how to prevent the operating system from being overloaded with unnecessary context switching, and how to influence the context switching order in a way enhancing the application performance. This mission is virtually impossible if the middleware runs in a virtual execution environment. Otherwise, it is realistic and achievable.

A scaling architecture that interacts directly with the operating system enjoys few major advantages. It could activate the required number of process instances without overloading the operating system with context switching. At the same time, with cache-conscious programming techniques it could minimize cache misses and with locks-conscious threads/CPU scheduling it could minimize the contention for serialized resources and the number of context switches. It could, as well, influence the context switching order and, thereby, positively influence the response time.

This paper presents the hardware-efficiency aspects of scaling architecture of Transactional Cascade® middleware technology. This architecture combines the simplicity of blocking IO with the higher reuse of business logic objects that is typical for non-blocking IO. It controls the threads/CPU scheduling and thereby reduces the serialization-caused threads blocking. By controlling the size of processor queues with ready for scheduling threads it shortens the duration of nested transactions. These enable execution of high-volume concurrent application instances per unit of hardware and super linear scaling of performance.

## 1. Performance of Application Server Hardware

The responsibility of application server for the overall system performance of Web applications with three-tier architecture (Figure 1) is naturally positioned to grow. Before the Internet-demanded level of scaling, frequently the entire business logic resided on the database server in form of stored procedures. This was the easiest way to deal with application consistency. Now, the business logic has to be taken out of the database server. On top of increasing complexity of business processes and transactions, the trend for application server is to take active role in atomic execution of business transactions and in consistency of applications.

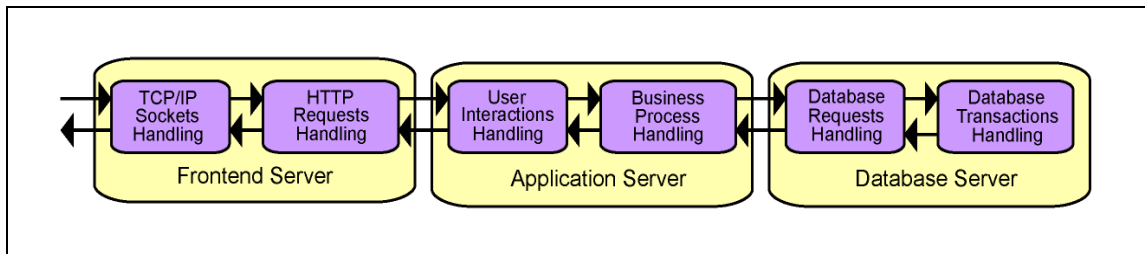


Figure 1 – Web Application with Three-Tier Architecture

Response time and throughput are the performance criteria of a computer system. Both are correlated in a way that scaling the throughput increases the response time. Under a constant speed of all elements of a computer system, scaling the throughput adds more parallel elements and creates more work for each sequential element. The higher workload of sequential elements increases the response time. Any performance improvement is measured either as throughput maximization coupled with acceptable response time, or as response time minimization under constant throughput.

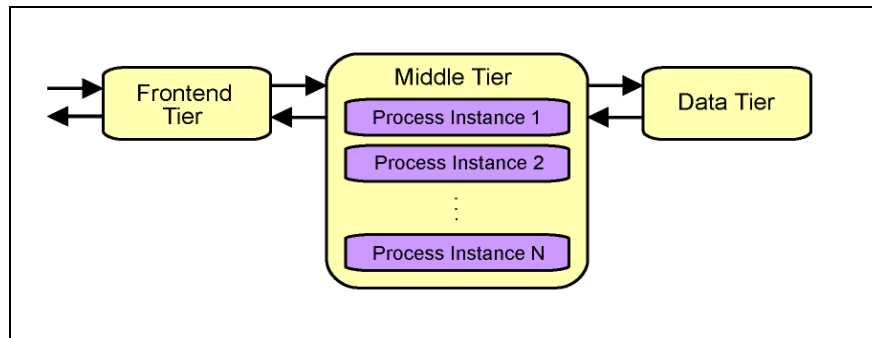


Figure 2 – Scaling a Business Process Application

One of the parallel elements of a Web application with three-tier architecture and a complex business process is an instance of its process. Scaling such application requires increasing the volume of its running process instances (Figure 2). Here, a process instance means one or multiple threads executing one or multiple business objects in the context of a particular user request. In practice, as a consequence of scaling, the overall performance might deteriorate

## 2. Scaling Contentions and Performance Bottlenecks

Application scaling is a source of variety of contentions. Some of them are caused by serialization of database transactions. The others are caused by the specific way of operation of multiprocessor systems.

Contention for database transaction locks is outside the scope of this paper. It does affect the application performance, and therefore, the performance of application server hardware. Executing multiple applications on the processor cores that run the middleware is the solution that will eventually necessitate creation of sufficient number of process instances to keep these cores performing useful work.

Application scaling is not linear. On symmetric multiprocessor systems and on chip multiprocessor systems, the major reasons for scaling non-linearity are:

- Memory contentions caused by the fact that all processors share the memory.
- Contention on resources shared between threads executed on different processors.
- Contention on threads scheduling
- Flushing and refilling of caches triggered by context switching.
- Memory lost with caches flushing and refilling.
- Increased cache misses because of higher threads scheduling rate.
- Cache-invalidation-caused reading from another cache to maintain coherency.

Any incidence of contention in fact increases the fraction of time when the system with parallel performance performs, at least partially, in sequential manner.

An increase of frequency of contention incidents or increase of contention extent in time might cause the following effects individually or in any combination:

- Accelerate the frequency or extent of another contention.
- Sharply deteriorate the overall system performance.
- Become a performance bottleneck.

For example, a contention for shared resource, say for a synchronization lock, blocks the concurrent execution of multiple threads. This triggers scheduling of another batch of multiple threads and associated switching of multiple contexts. A context switch causes processor's L1 and L2 caches to be flushed and refilled. With too many context switches, the lost memory will cause cache misses and a necessity to perform the slow readings from main memory. This will deteriorate the performance and might eventually totally reverse the benefits of scaling.

The above scenario will not necessarily trigger large reading from main memory. It frequently repeated, however, it will overload the operating system with excessive context switching and become a performance bottleneck. The formation of performance bottlenecks is inevitable with scaling. Some bottlenecks might get eliminated while others are part of the scaling nature and will always be presented. With efficiency of software, the formation of bottlenecks that cannot be completely eliminated could at least be delayed to a later scaling stage.

### 3. Techniques for Scaling Efficiency

During the development of Transactional Cascade® technology, we performed experiments with multiple versions of its scaling architecture. From the gained experience and achieved results, we recommend the following techniques for delaying the formation of bottlenecks and for better application server performance:

#### Technique #1

Structure the middleware threads into groups with common features, for example, usage of the same thread-function. Direct the OS to schedule a group of threads only on a particular processor, from those the process is allowed to run on.

##### Effect 1

Interrupting a thread and later scheduling it back to the same processor increase the probability that the cache of this processor might still contain cache lines belonging to this thread. If the group consists of threads with the same thread-function, the probability for both L1 Instructions cache and L1 Data cache still containing lines belonging to this thread keeps increasing. If only the threads with same thread-function are scheduled to this processor, the probability that the cache of this processor contains cache lines belonging to this thread is the highest.

Vice versa, interrupting a thread and scheduling it to the different processor will most probably result in multiple cache misses that will trigger retrieving of cache working set from the cache of other processor or, in the worst case scenario, from RAM.

##### Effect 2

Having all threads with the same thread-function being executed on the same processor, in some cases eliminates the need for resource synchronization and, in other case, reduces the probability or the extend of waiting for synchronization locks. Once the access to a resource cannot be requested simultaneously from more than one thread, there is no need for synchronization. Similarly, reducing the number of threads that might attempt to access a resource simultaneously will reduce the probability or the extent of eventual waiting for synchronization locks.

#### Technique #2

Minimizing the number of invalidated cache lines by concentrating together the frequently updated parts of memory in groups with the size of multiple cache lines.

##### Problem description

Processor cache line represents the state of a particular part of memory. Caches of multiple processors might contain copies of the same line. Whenever a processor has modified a word in its cache, it broadcasts a message. Other processors detecting this message invalidate that particular word on their copies of the line. Any processor with an invalidated cache line will have a cache miss when that line is addressed.

Effect

Concentrating together the frequently updated parts of memory reduces the number of invalidated cache lines and thus the number of cache misses and the following retrieving from the cache of other processor.

**Technique #3**

Never let the normal processing flow develop spikes in number of threads with state *ready* for scheduling. Implement a mechanism to serialize the flow of events that change thread's state from *waiting* an event to *ready* for scheduling.

Effect 1

This technique maintains the number of *ready* for scheduling threads in the processor queue as low as practically possible. Having a small number of threads with *ready* state guarantees that any thread with *waiting* state (because of execution of a blocking IO with the database server) will not have to wait unnecessarily long to be scheduled after the completion of its IO operation.

Effect 2

As a direct consequence of Effect 1, this technique is especially important for applications performing nested database transactions where the controlling transaction is initiated and completed from the application server. Its implementation enables database locks to be held for shorted time intervals. As a result of this, the average application response time decreases and application throughput increases.

**Key Points**

Scaling the number of process instances executed on a multiprocessor application server might deteriorate application performance. It starts and accelerates:

- The contention for system and application resources.
- The frequency of threads scheduling and context switching.
- Memory losses from CPU cache flushing and refilling, and cache misses.
- The frequency and extent of operations related to cache coherency.
- The delay in scheduling of threads after completion of blocking IO.

Formation of these scaling-caused sources of scaling non-linearity could be mitigated with implementation of the proposed techniques.

## 4. Improving the Performance of Application Server

Efficient scaling requires fitting the specific way of expansion of software operations with the limits of execution environment, and particularly with the specifics of multiprocessor architecture. The operating system and the virtual machine cannot obtain the necessary design and runtime information to perform this task efficiently. Only the middleware knows the details of scaling techniques it implements and the necessary runtime performance-related information. Thus, it could further perform runtime tuning of these techniques for better fitting with the multiprocessor architecture's cache operations and threading.

The eventual use of a virtual machine imposes a level of indirection in middleware interactions with the hardware and the operating system. Thus a middleware, which runs in a virtual execution engine or in environment that implements hardware or operating system virtualization, cannot fully implement the proposed techniques for delaying the formation of bottlenecks and better application server performance.

A scaling architecture that interacts directly with the operating system has major advantages. Some of them are:

### **Minimized contention for serialized resources**

By taking the responsibility from the OS for threads/CPU scheduling, the middleware can implement fine-grained techniques aimed at minimization of contention for application resources with serialized access. Directing the OS to schedule a particular group of threads only on a particular processor, from those the process is allowed to run on, frequently results in complete elimination of contention.

### **Minimized preempting of threads**

Minimized contention for shared resources, or complete elimination, in effect minimizes the necessity of operating system to preempt a running thread before its time-slice expires or before it initiates a blocking IO call. Minimized preempting of threads minimizes the number of performed context switches per unit of time.

### **Minimized cache misses**

Minimized context switches minimize the frequency and extent of cache misses that result from the context-switch-caused flush and refill of processor caches. The minimized cache misses minimize the deterioration of performance that results from the necessity to perform the slow readings from main memory.

### **Minimized delays in rescheduling of threads after completion of blocking IO**

By ensuring the size of processor queue of threads ready for scheduling is small, the middleware guarantees that any thread with a just-completed blocking IO will not have to wait unnecessarily long to be scheduled again. This shortens the application response time and increases its throughput. Moreover, with nested transaction the holding time of database locks gets shorter. This additionally accelerates the already gained performance improvement.

### Higher volume of process instances

Thanks to the all just-mentioned advantages, an instance of a middleware could create and execute high volume of process instances on a multiprocessor application server with no deterioration of performance. In fact, the only significant performance limit is established by the process nature, which defines the conceptual ability to parallelize the execution of its transactions.

#### Key Points

- The OS and the VM cannot optimize the performance of a middleware.
- Only the middleware could tune itself at runtime for efficient scaling.
- The runtime self-tuning is virtually impossible in a virtual execution environment.
- Otherwise, the runtime self-tuning is realistic and achievable.

## 5. Performance with Transactional Cascade®

Transactional Cascade® middleware technology is embedded in Transactum Engine (Figure 3). An instance of Transactum Engine runs on 2 processor cores and executes up to 1,000 concurrent process instances of up to 8 applications. The Engine dynamically scales up and down the volume of process instances of each application, according to the fluctuations of its workload in time.

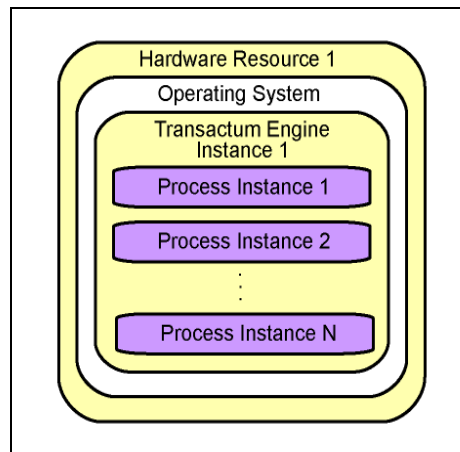


Figure 3 – Process Instances Scaling with Transactional Cascade®

Figure 4 presents the implementation of process instances scaling with Transactional Cascade® technology. During the initialization of a process, a dedicated pool of threads is created per process step. Each thread instantiates a component object that contains the business functionality of a relevant process step. A thread is normally in a *waiting* state until it receives an assignment to execute the process step and after the assigned work is completed.

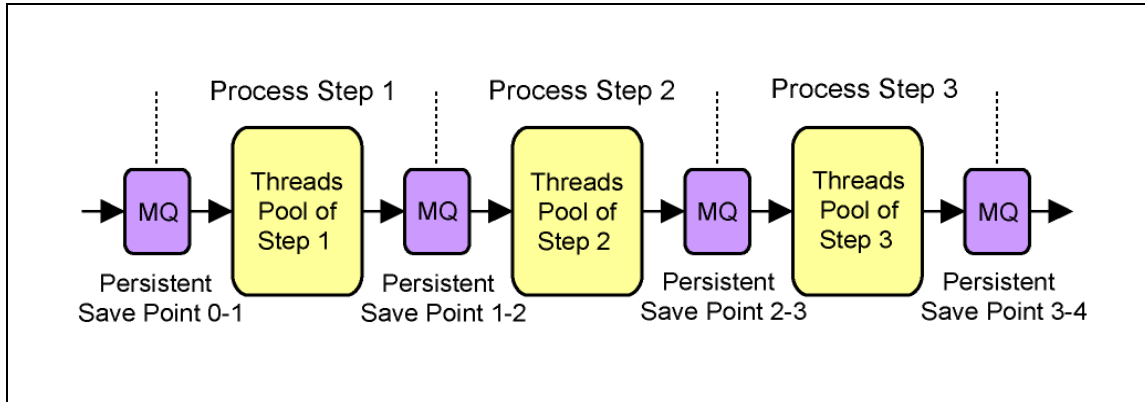


Figure 4 – Implementation of Process Instances Scaling with Transactional Cascade®

A business process step starts and ends with a Persistent Save Point implemented with a transactional Message Queue. A Persistent Save Point separates every two adjacent process steps. Physical arrival of a message in a Persistent Save Point' message queue triggers the following sequence of actions:

1. A thread from the pool is assigned to deal with this message.
2. It reads the content of the message.
3. It passes the content to the business object.
4. The business object performs its activities
5. It composes a message for the next process step.
6. On exit, it returns the new message.
7. The thread physically fetches the old message from its input MQ.
8. The thread places the new message in its output MQ.

Business objects of applications executed on Transactum Engine perform blocking database IO calls. Compared to applications implemented on middleware products with blocking IO calls, the applications implemented with Transactional Cascade® reuse the instances of business object at a significantly higher rate – similarly to architectures with non-blocking IO calls.

Mainstream middleware products with blocking IO create a worker thread per user connection. Thus, if a process executes 10 steps with a business object per step, serving 1,000 concurrent users requires creation of 1,000 worker threads with a total of 10,000 object instances. In contrast, the presented on Figure 4 implementation of scaling requires creation of 1,000 worker threads with a total of 1,000 object instances – 10 times less object instances.

The presented in Chapter 3 techniques for scaling efficiency enable Transactum Engine to execute high-volume concurrent application instances and to perform super linear scaling of application throughput. The increase of the throughput is usually paid with an increased response time. The case with Transactum Engine is exactly the opposite. Where scaling shows super-linearly increased throughput, it is paired with reduced response time. The reduced response time is in fact the primary cause of the super linear scale of throughput.

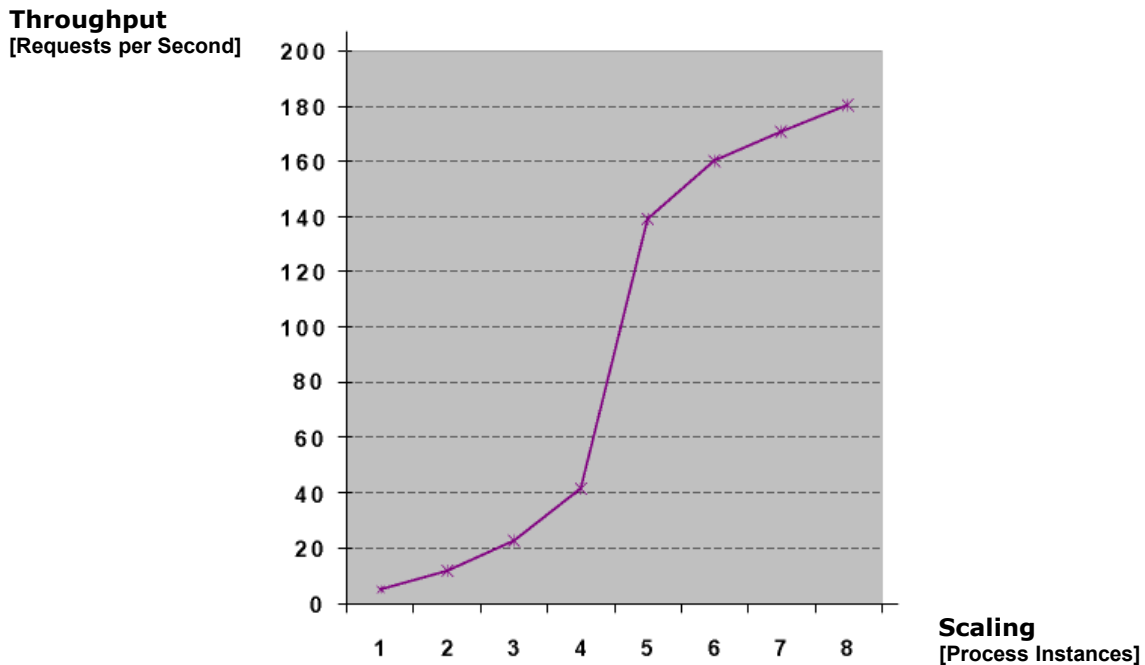


Figure 5 – Scaling an Application Performing 5 Serialized Transactions with Transactum Engine on 2 CPU Cores

Figure 5 presents the scaling diagram of an application performing 5 serialized database transactions. The experiments were performed on commodity hardware with a CPU with 4 CMP cores. To eliminate the effect of networking, both the Transactum Engine and the database server were on the same computer, each running on 2 CPU cores.

Scaling curves normally start with a linear part where increasing the throughput does not deteriorate the response time until a formation of major bottleneck starts. With this application, the formation of a major bottleneck starts after scaling the number of process instances beyond 5, because of the contention for database serialization locks.

With a linear scaling, increasing the number of process instances from 1 to 5 increases the throughput 5 times. On the diagram of Figure 5, scaling the number of process instances from 1 to 5 scales the throughput 28 times and decreases the application response time 5.6 times. With the execution 6<sup>th</sup> concurrent process instance, the super-linear performance of Transactum Engine starts fading, offset by the contention for database serialization locks.

### Key Point

Transactional Cascade® technology improves the performance of hardware, measured with:

- Increased overall application throughput; and
- Shortened average application response time.

## 4. Conclusion

This paper briefly presents the sources of contention leading to performance bottlenecks when the application performance is scaled on an application server with multiprocessor hardware. It proposes some proven techniques for enhancing the hardware performance with more efficient scaling of software operations. These techniques were formulated as a result of the experiments Transactum Pty Limited performed with multiple versions of the scaling architecture of its Transactional Cascade® technology.

Further, the paper describes how exactly the proposed techniques improve the overall application server performance and thereby the performance of its hardware. It illustrates the efficiency of presented concepts with a result achieved with the scaling architecture that implements them. The presented example shows the scaling curve of an application from a category known with its notoriously difficult scaling – the applications performing multiple serialized database transactions.

The paper does not touch the subject of contention for database transaction serialization locks and the mitigation of their effect on application performance. It is discussed in our next technology paper [Interactive Responsiveness and Concurrent Workflow](#).

### Value Proposition

Transactional Cascade® can make the scaling of any new category Web applications:

- More hardware-efficient;
- More energy-efficient; and
- Less impacting the environment.

[Give a feedback or ask questions](#)

TRANSACTIONUM PTY LTD

Tel: + 61 2 9567 1150

Fax: + 61 2 9567 1160

[WWW.TRANSACTIONUM.COM](http://WWW.TRANSACTIONUM.COM)

PO Box 324, Brighton-Le-Sands NSW 2216, Australia