Master Thesis
Software Engineering
Thesis no: MSE-2009-35
September 2009

A Simple Throttling Concept for Multithreaded Application Servers

Fredrik Stridh

School of Computing
Blekinge Institute of Technology
Box 520
SE – 372 25 Ronneby
Sweden

This thesis is submitted to the School of Computing at Blekinge Institute of Technology in partial fulfillment of the requirements for the degree of Master of Science in Software Engineering. The thesis is equivalent to 20 weeks of full time studies.

Contact Information:
Author(s):
Fredrik Stridh
E-mail: whzfred@gmail.com

University advisor:
Dr. Daniel Häggander

School of Computing
Blekinge Institute of Technology
Box 520
SE – 372 25 Ronneby
Sweden

Internet : www.bth.se/com
Phone : +46 457 38 50 00
Fax : + 46 457 271 25

ABSTRACT

Multithreading is today a very common technology to achieve concurrency within software. Today there exists three commonly used threading strategies for multithreaded application servers. These are thread per client, thread per request and thread pool. Earlier studies has shown that the choice of threading strategy is not that important. Our measurements show that the choice of threading architecture becomes more important when the application comes under high load. We will in this study present a throttling concept which can give thread per client almost as good qualities as the thread pool strategy when it comes to performance. No architecture change is required. This concept has been evaluated on three types of hardware, ranging from 1 to 64 CPUs, using 6 alternatives loads and both in C and Java. We have also identified that there is a high correlation between average response times and the length of the run time queue. This can be used to construct a self tuning throttling algorithm that makes the introduction of the throttle concept even simpler, since it does require any configuring.

Keywords: throttling, multithreaded, server

CONTENTS

1 INTRODUCTION....3

2 APPLICATION SERVERS....4

3 THREADING STRATEGIES....5

3.1THREAD PER CLIENT....5
3.2THREAD PER REQUEST....5
3.3THREAD POOL....5

4 RESEARCH METHOD....6

4.1RESEARCH QUESTIONS....6
4.2SIMULATOR....6
4.2.1CLIENT....6
4.2.2SERVER....7
4.2.2.1IMPLEMENTATION OF THREADING STRATEGIES....8
4.2.2.1.1THREAD PER CLIENT....8
4.2.2.1.2THREAD PER REQUEST....9
4.2.2.1.3THREAD POOL....9
4.2.2.1.4THREAD PER CLIENT WITH REQUEST PROCESS THROTTLE....10
4.2.3BOTTLENECKS....11
4.2.3.1LOCK....12
4.2.3.2YIELD....12
4.2.3.3OPEN AND CLOSE FILE....13
4.2.3.4INTER SOCKET CONNECTION....13
4.2.3.5MEMORY ACCESS....14
4.3EXPERIMENT SETUP....15

5 RESULT....16

5.1PERFORMANCE EVALUATION....16
5.1.1C version running on Solaris 10 with 64 CPUs....16
5.1.2Java version running on Solaris 10 with 64 CPUs....19
5.1.3C version running on Solaris 10 with 8 CPUs....22
5.1.4Java version running on Solaris 10 with 8 CPUs....25
5.1.5C version running on Linux with 1 CPU....28
5.2CORRELATION....31

6 DISCUSSION....32

7 CONCLUSION....33

1 INTRODUCTION

Multithreading [1] is today a very common technology to achieve concurrency within software. In Java [6] it is even a part of the language. Multithreading software make it possible to utilize some types of parallel hardware, such as Symmetric Multiprocessors [11] and Multi-Core CPUs. It is also well suited for multithreaded hardware.

Three commonly used threading strategies for multithreaded application servers (see chapter 2) are thread per client, thread per request and thread pool (see chapter 3). Earlier studies have shown that the choice of threading strategies has limited effect when it comes to application performance [5]. Thread per client has slightly higher performance than thread pool and thread per request.

Our measurements show that the choice of threading architecture becomes more important when the application comes under high load. The thread pool gives a significant higher performance, particular when it comes to response times.

On the other hand, the thread pool strategy has a more complex architecture comparison to thread per client and thread per request strategies, which are based on a much more simple principle and does not require a request queue. Yet another disadvantage with the thread pool threading strategy is the client isolation level, clients are sharing the same execution resources. Thus, it is not unusual that the first choice of threading architecture is the thread per client or thread per request. A change of threading strategy from one of those into a thread pool can be both complex and time consuming architecture change.

We will in this study present a throttling concept which can give almost as good qualities as the thread pool strategy when it comes to performance. It can very simple be applied on a thread per client or thread per request based architecture, i.e. no architecture change is required. The concept has been evaluated on three types of hardware, ranging from 1 to 64 CPUs. The measurements have been made by using 6 alternative workloads. Measurements have been done in C [7] as well as in Java.

The results shows that by applying a simple concept can make a thread per client almost as effective as a thread pool when it comes to performance during high load. Finally, we have also show the possibility to improve the concept with a self tuning algorithm, which will make the concept introduction even simpler and in the same time remove client isolation issue. The self tuning algorithm can also be applied to the thread pool.

The rest of the report is structured in the following way. Chapter 2 gives an introduction to applications servers. Chapter 3 defines the three threading strategies, thread per client, thread per request, and thread pool, respectively. The method is, together with the throttling concept and simulator construction, described in chapter 4. Chapter 5 presents our results. A wider discussion on our concept can be found in chapter 6, while chapter 7 concludes the report.

2 APPLICATION SERVERS

Application servers could be defined as a software framework that allows you to run for example scripts or programs that support other applications. Web servers such as apache tomcat or telecom servers are good example of application servers. Figure 1 shows a typical workflow for an application server. In this example we use a web server.

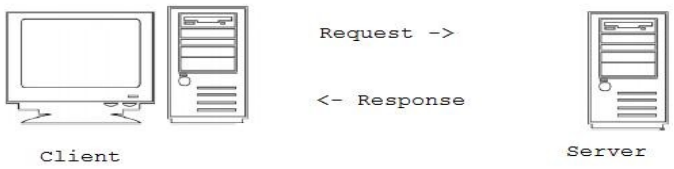

Figure 1.

A request is sent from a client to the application server (web server). This could for example be a request for a web page. The request is handled by the application server and a response is sent back to the client. In this example the response consists of a web page. The time elapsing from the point where the client sends the request and until the response is received is called response time.

Application servers must be able to handle multiple clients connected, therefore they use a architecture consisting of threads. When multiple clients are connected to the server performance is affected. The response time for a request starts to increase (see chapter 5) when multiple clients are connected and sending requests. It is important that the response times for application servers are as low as possible, since the client could be an end user or an application waiting for a response back, before it can continue to work.

Performance of applications servers can depend on many factors such as network speed, architecture and hardware [5]. In this thesis we will concentrate on threading architectures (see chapter 3) for application servers, so called threading strategies.

3 THREADING STRATEGIES

This chapter gives an introduction to the three commonly used threading strategies. The threading strategies described in this thesis are all well known and are evaluated in the paper "Performance Comparison of Middleware Threading Strategies" [5]. All these threading strategies operate at application level.

Today there also exists threading strategies in some operating systems that can be used in applications, but are controlled by the operating system. For example in Mac OS X we have the Grand Central Dispatch GCD) [14] that is controlled by the operating system. GCD uses a implementation based on the thread pool [15] that allows work to be distributed across available CPU cores.

3.1 THREAD PER CLIENT

In the thread per client threading strategy each client connected the server gets its own receiving thread. When a request is sent from the client to the server this thread is responsible for receiving the request. Also the same thread processes the request and then sends back a response to the client. After an answer is sent back to the client this thread is ready to receive a new request. This is the simplest form of threading architecture.

3.2 THREAD PER REQUEST

In the thread per request threading strategy each client connected the server gets its own receiving thread. When a request is sent from the client to the server this thread is responsible for receiving the request. After a request has been received a dispatcher thread is created and the receiving thread is ready to receive a new request. The dispatcher thread then processes the request and also sends back a response to the client. After the dispatcher thread has finished its work it is destroyed. With multiple clients connected to the server the number of threads can grow very fast [5]. Also overhead is added from destroying and creating dispatcher threads [5].

3.3 THREAD POOL

In the thread pool strategy there exist a number of dispatcher threads in a pool. When a client connects to the server it gets its own receiving thread. This thread is responsible for receiving the request. After a request has been received it is put in a FIFO queue and the receiving thread is ready to receive a new request. When a dispatcher thread becomes available the thread pool assigns it to process the first request in the FIFO queue. The dispatcher thread also sends back a response to the client after the request has been processed and then the dispatcher thread is returned to the thread pool. The thread pool strategy uses the consumer-producer pattern [2][3]. The thread pool addresses the problem with uncontrolled growth of dispatcher threads found in thread per request. A disadvantage with thread pool is the client isolation issue, because clients are sharing the same execution resource. This meaning that if one of the dispatcher threads from the thread pool crashes while processing a request, then this dispatcher thread is not returned to the pool and this thread cannot continue to process additional requests.

4 RESEARCH METHOD

In this chapter we present our research questions. To be able to answer our research questions a simulator had to be constructed. This simulator is also described in detail in this chapter and then we continue with a description of our experiment setup.

4.1 RESEARCH QUESTIONS

The main research questions are:

RQ1: What is the performance of thread per client, thread per request and thread pool using different hardware and bottlenecks under high load?

RQ2: Is it possible to introduce a throttle concept that can give the thread per client the same qualities as thread pool when it comes to performance?

RQ3: Is it possible to use the run time queue length to control the throttle?

4.2 SIMULATOR

To be able answer our research questions a simulator had to be constructed. The simulator consists of a server and a client part. We have developed both a C and a Java version of the simulator. Both versions of the simulator use TCP/IP connection for sending data between the server and client. The C versions uses Pthreads for threading and the Java version uses standard Java threads supported by the JVM version 5. Pthreads can be explained as a set of functions used for POSIX threads.

4.2.1 CLIENT

The client's main function is to send XML requests over a TCP connection to server for processing. The following important parameters are passed when the simulator is started:

- -CL, number of client threads to be started. Each thread represents a client.
- -LOOP, number of times each request shall be processed by the server.
- -RPS, number of requests to send per second (throughput).
- -RQ, total number of request to be sent for all client threads

Before the client starts sending requests it waits for all client threads to get a connection to the server. After all threads have received a connection, then requests are starting to be sent based on the throughput setting (RPS). Each client thread blocks after a request has been sent and waits for a response before sending addition request. There is also little difference between thread per client and thread per request, since the sequential behavior described above only allows the server to process one concurrent request per client.

To control the throughput in the client, time stamps are used. They are used to tell the client when a new request shall be sent. The time stamps are generated using the formula [13] in figure 2 based on the rps setting. It is also possible to load and save a time stamp file so that the same settings can be used later.

$$F(t) = 1 - e^{-\lambda t}$$
$$t = \ln(x) / \lambda$$
$$x = \text{random}\ [0..1]$$

Figure 2.

The number of events generated per second in average is lambda and t equals the random time between two events.

The following is an example of a XML request used in our simulator and it consist of a loop parameter and random generated strings.

```
<request>
        <loop>1000</loop>
        <strings>
                <string>arefmdfdt</ string>
                < string >btrtgfgfdf</string>
                < string >rqlgfrtrtrd</string >
                .
                .
                .
                <string>sdslgfrdfd</string>
        </strings>
</request>
```

The random strings are generated in forehand by the client and are the same for all requests. It is possible to set the number of strings. The strings are generated and then saved to a file. Also it is possible to load this file, so that the same strings can be used in future simulations.

4.2.2 SERVER

The Server accepts an incoming XML request from the client over a TCP connection. When starting the server it is possible to configure which threading mode it shall use.
All threading modes except our proposed throttle concept is implemented based on descriptions from the paper “Comparison of Middleware Threading Strategies” (5).
Each request sent to the server goes through six main parts of the simulator server. These parts are: listen for client connection, receive request, parse XML request, process request, generate XML response and send response. All these parts are the same for all threading strategies. Later in the implementation description we will refer to these functions.

- Listen For Client Connection: A TCP socket is setup and listens for connections from clients. When a connection has been received it is passed to the receiver thread.

- Receive Request: A request is received over a TCP connection from the client. After the request has been received the data is passed to the parse XML Request function.

- Parse XML Request: The XML request is parsed. Strings and the loop parameter are extracted.

- Process Request: All strings extracted from the XML request are sorted and stored into an array. This is then repeated equal to the value of the loop parameter.

```
for(i = 0; i < loop; i++)
{
        sort_strings();
}
```

- Generate XML Response: A XML response is generated containing the strings in sorted order.

- Send Response: The XML response is sent back to the client over the TCP connection.

4.2.2.1 IMPLEMENTATION OF THREADING STRATEGIES

Here we present an overview of the implementation of the threading strategies described in the threading strategies chapter. In this chapter we also describe the implementation of our throttle concept that we briefly described in the introduction. Also when constructing our simulator we tried to not use too many language specific functions, since we both developed a version in C and java. The threading strategies are presented using pseudo code. We use some terms that we would like to explain more in detail to get a better understanding of how the threading strategies where implemented.

Mutex: A mutex is a synchronization method used for thread synchronization. A call to *mutex_acquire()* makes the calling thread owner of the mutex. Then no other threads are allowed to pass before a call to *mutex_release()* has been done by the thread owning the mutex. Threads waiting to acquire the lock are put in a suspended state by the thread scheduler. A mutex is often used to protect a shared resource that is not allowed to be access by more than one thread at a time.

FIFO Queue: A queue is a data structure that is used to store data in an ordered order. In this case the order is first in, first out.

4.2.2.1.1 THREAD PER CLIENT

The thread per client strategy uses a listening thread for listening for incoming connections from a client. When a connection has been received a new receiver thread is created and the listening threads returns to listen for new connections.

Listening Thread:

```
repeat
        listen_for_client_connection();
        create_receiver_thread(connection);
forever
```

Receiver Thread:

```
do
```

receive_request();
parse_XML_request();
process_request();
generate_XML_response();
send_response();
while(connection_is_active)

4.2.2.1.2 THREAD PER REQUEST

The thread per request strategy uses a listening thread for listening for incoming connections from a client. When a connection has been received a new receiver thread is created and the listening threads returns to listen for new connections. The receiver thread creates a new dispatcher thread for each incoming request. The dispatcher thread is then responsible for processing the request.

Listening Thread:

repeat
listen_for_client_connection();
create_receiver_thread(connection);
forever

Receiver Thread:

do
receive_request();
parse_XML_request();
create_dispatcher_thread(request);
while(connection_is_active)

Dispatcher Thread:

process_request();
generate_XML_response();
send_response();

4.2.2.1.3 THREAD POOL

The thread pool strategy uses a listening thread for listening for incoming connections from a client. When a connection has been received a new receiver thread is created and the listening thread returns to listen for new connections. The receiver thread en-queues the request in a FIFO request queue. The thread pool assigns a dispatcher thread to de-queue the request and process it. After the request has been processed the dispatcher thread is returned to the thread pool. When the server is started a fix number of dispatcher threads are created.

Listening Thread:

repeat
listen_for_client_connection();
create_dispatcher_thread(connection);

forever

Receiver Thread:

```
do
        receive_request();
        parse_XML_request();
        create_dispatcher_thread(request);
while(connection_is_active)
```

Dispatcher Thread:

```
process_request();
generate_XML_response();
send_response();
```

4.2.2.1.4 THREAD PER CLIENT WITH REQUEST PROCESS THROTTLE

Since our throttling concept itself is not a complete threading strategy, we decided to combine it with the thread per client strategy in this implementation. The reason for choosing the thread per client strategy is that this strategy is the most simple. When implementing our throttling concept we decided to give it the name, *request process throttle*.

A listening thread is used for listening for incoming connections from a client. When a connection has been received a new receiver thread is created and the listening thread returns to listen for new connections. When you start the server a parameter is passed to set the concurrency limit (how many request that are allowed to be processed at the same time). When the concurrency limit is reached, threads are put in wait state and en-queued on a FIFO queue. After a request has been processed the FIFO queue is checked if it contains any threads, if so the first thread in the FIFO queue is de-queued and is allowed to process the request. By using the request process throttle it is possible to control the number of request being processed at the same time. The algorithm also uses a variable to store the number of threads currently being processed.

Listening Thread:

```
repeat
        listen_for_client_connection();
        create_dispatcher_thread(connection);
forever
```

Receiver Thread:

```
do
        receive_request();
        parse_XML_request();

        if(limit is reached)
        {
                queue_thread();
                thread_wait();
        }
```

```
                process_request();
                generate_XML_response();
                send_response();

                if(threads on queue)
                {
                                deqeueue_thread();
                                thread_signal();
                }
while(connection_is_active)
```

In the simulator the request process throttle is implemented using two major function. These two functions are an entry and exit point for the throttle. In a real application it is possible to put all code for the throttle in an external library and just use these function calls in your application. The entry point is called before the code for processing the request and the exit point is called after the processing is done. This makes it easy to include the throttle in any server application without any architecture change. The request process throttle comes with another benefit also. At any time it is possible to call the exit function and then this part of the application does not have limited concurrency. For example this can be called if you want to access the network. When you are done accessing the network you can call the entry function again for the throttle to limit the concurrency.

```
entry_throttle();
                process_request();
exit_throttle();


process_request()
{
                .....
                exit_throttle();
                                access_network();
                entry_throttle();
                .....
                // here we are done processing the request
}
```

Implementing this behavior for the thread pool would require a lot of complex work, since you need to have multiple queues and also a change in the architecture is required.

4.2.3 BOTTLENECKS

In the real world there does not exists perfect parallel applications. There are always bottlenecks in applications. Therefore we have introduced the concept of bottlenecks in our server. In the simulator you can pass a parameter to tell how many times each bottleneck shall be called per request. Bottlenecks are called inside the main process request loop.

The bottlenecks are presented using pseudo code. To be able to understand the pseudo code better we would like to introduce the concept of semaphores.

Semaphore: A semaphore is also a synchronization method used for thread synchronization. In our server we have used counting semaphores that has a number of permits. The number

of permits is set when the semaphore is initialized. Each call to *semaphore_acquire()* gives a permit to the calling thread. When the permits reach zero the calling thread are put in a suspended state by the thread scheduler until new permits becomes available. A call to s*empahore_release()* releases a permit and returns it back to the semaphore. A semaphore is often used to restrict the number of threads accessing a protected resource.

4.2.3.1 LOCK

The purpose of this bottleneck is to introduce locking behavior in the simulator. In our simulator we have used a simple mutex lock to implement this bottleneck. In real applications a mutex could be used to give exclusive access to a resource. We believe that this is something that is done frequently in server applications. Inside the lock we have placed code to update a variable, so that the compiler does not through away the lock when optimizing the code. The lock function is then called inside a semaphore so that we can experiment with the effect of many or a few threads trying to gain the lock.

The function *bottleneck_lock()* is not done very loop, instead this is controlled by predefining how many times this function should be call per request when starting the application

```
process_request()
{
        for(i = 0; i < loop; i++)
        {
                sort_strings();
                bottleneck_lock();
        }
}


bottleneck_lock()
{
        semaphore_acquire ();
                lock();
        semaphore_release();
}


lock()
{
        mutex_acquire();
                update_variable();
        mutex_release();
}
```

4.2.3.2 YIELD

The purpose of the Yield bottleneck is to simulate the effect of a context switch [10]. A context switch is the processing of storing and restoring the state of a thread. In this case a call to *sched_yield()* forces the current running thread to immediately suspend running allowing waiting threads to be able to run. The function *bottleneck_yield()* is not done very

loop, instead this is controlled by predefining when starting the application how many times this function should be call per request.

```
process_request()
{
        for(i = 0; i < loop; i++)
        {
                sort_strings();
                bottleneck_yield();
        }
}

bottleneck_yield()
{
        sched_yield();
}
```

4.2.3.3 OPEN AND CLOSE FILE

The purpose of the open and close file bottleneck is to simulate a system call. In a real application there can exists many system calls and we want to see if this affects performance. A system call is a done to request a service from the operating system, in this case open and close file. The open and close functions are called inside a semaphore so that we can control the number of threads accessing the open and close file bottleneck. The function *bottleneck_open_close_file()* is not called every loop, instead this is controlled by predefining when starting the simulator how many times this function should be call per request.

```
process_request()
{
        for(i = 0; i < loop; i++)
        {
                sort_strings();
                bottleneck_open_close_file();
        }
}


bottleneck_open_close_file()
{
        semaphore_acquire ();
                open(filename);
                close(filename);
        semaphore_release();
}
```

4.2.3.4 INTER SOCKET CONNECTION

The purpose of the inter socket connection bottleneck is to simulate a database query. This is something that is done often in real applications. This is simulated using a additional running

instance of the simulator server. A request is sent away by the *sendRequest()* function for additional strings to sort. A response is sent back containing additional strings and these are added to the strings that are going to be sorted by the *sort_strings()* function. Connection to the server is established when the simulator is started and the connections are then sorted in a connection pool. A call to the *get_available_network_connection()* function returns a free connection. Also this bottlenecks needs to be protected by a semaphore, since we do not want a thread accessing the connection pool when there is no free connection available.

```
process_request()
{
        for(i = 0; i < loop; i++)
        {
                bottleneck_network();
                sort_strings();
        }
}

bottleneck_network()
{
        semaphore_acquire();
                get_available_network_connection();
                send_request();
                get_response();
        semaphore_release();
}
```

4.2.3.5 MEMORY ACCESS

The purpose of the memory access bottleneck is to see if there is any effect on the performance results when it comes to cache effects. This is done by calling the *memory_access()* function every loop from inside the *process_request()* function. A parameter is also passed when starting the simulator to set the size of the memory to access. When calling the *memory_access()* function a structure containing random generated numbers are passed to the function. This structure is then later access by the function. In the Java implementation a object is passed to the function, instead of a structure, since a structure does not exists in the Java programming language.

```
process_request()
{
        for(i = 0; i < loop; i++)
        {
                memory_access ();
                sort_strings();
        }
}

memory_access()
{
        access_memory();
}
```

4.3 EXPERIMENT SETUP

All measurements have been done using our own simulator described in section 3.2. Both the client and the server are on the same machine, because we do not want the network to be a limiting factor in our experiment. The simulator is overloaded with more requests than it can handle. This is done by setting a high throughput of requests.

Using our throttle concept combined with thread per client we have compared the average response time for a request with concurrency limit and average run time queue length. The average run queue length is sampled using vmstat. This has been done for all 5 bottlenecks described earlier in this chapter and also using a combination of all the bottlenecks together. Then we have done a comparison between request process throttle and thread pool. We have also done a comparison between thread per client, thread per request, thread pool and our throttle concept combined with thread per client (request process throttle). Finally we have varied the number of clients and done a comparision between thread per client, pool and request process throttle.

All measurements have then been done on a Sun T5220 using 64 CPUs (8 CPUs x 8 cores) running Solaris 10, Sun T5220 using 8 CPUs (1 CPU x 8 cores) running Solaris 10 and Pentium 4 using 1 CPU system running Linux(Slackware 12). Also all measurements have been repeated both for Java and C version of the simulator. For the 1 CPU system is was not possible to overload the Java version of the simulator, so therefore there are no Java measurements available for this system.

The number of maximum clients differs between systems and we have used the maximum number available to run on the different systems in our experiment. Also we had to vary the number of request on the different systems, since it would take to long time on the systems with a low amount of processing units.

Also the average response time for one request varies on the different systems and we tried to keep the response time for one request between 250 – 1500 milliseconds. This was done by setting number of clients to one and calculates the average response time for 10 requests. The average response time can then be controlled by altering the loop parameter in the simulator.

For the open/close file and inter socket connection bottleneck we choose to do this in semaphore with the value of four. The reason for choosing this value is that we do not want the number of file descriptors to increase when many threads tries to get a file descriptor. Also in a real application there are usually limited resources available.

The yield and lock bottleneck was done inside a semaphore with the value of 1000, since we wanted many threads at the same time accessing these bottlenecks.

5 RESULT

The result is divided into two parts. First we present the data from our experiment described in chapter 4. Then we continue with correlation between run time queue and average response times.

In this chapter our throttling concept combined with thread per client will be given the name: request process throttle (RPT). Response times in the charts are average response time for a request in milliseconds. The total run time are also measured in milliseconds and run time queue is the average length.

5.1 PERFORMANCE EVALUATION

The data from our experiment in chapter 4 is presented here.

5.1.1 C version running on Solaris 10 with 64 CPUs

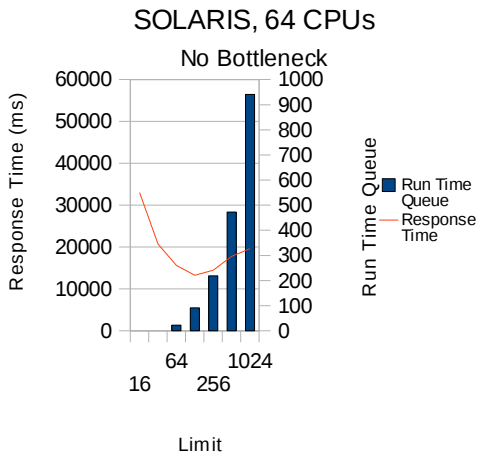

Diagram 1.

Simulator Settings
Clients: 1000
Requests: 2000
Loop: 40000
Comment: Request process throttle, concurrency limit is varied.

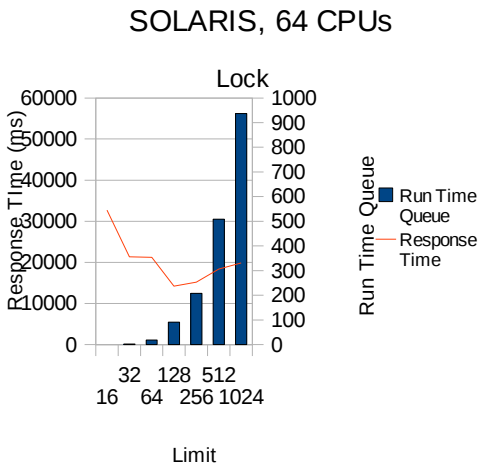

Diagram 2.

Simulator Settings
Clients: 1000
Requests: 2000
Loop: 40000
Comment: Request process throttle, concurrency limit is varied.

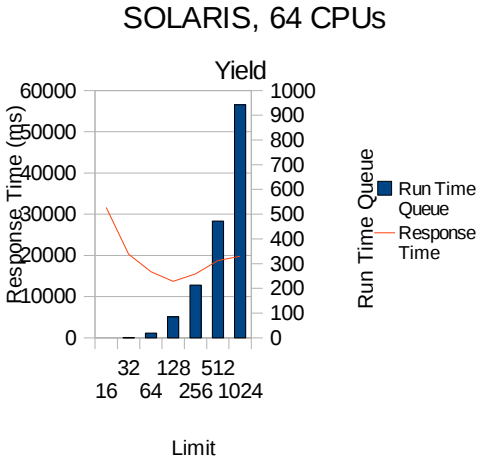

Diagram 3.

Simulator Settings
Clients: 1000
Requests: 2000
Loop: 40000
Comment: Request process throttle, concurrency limit is varied.

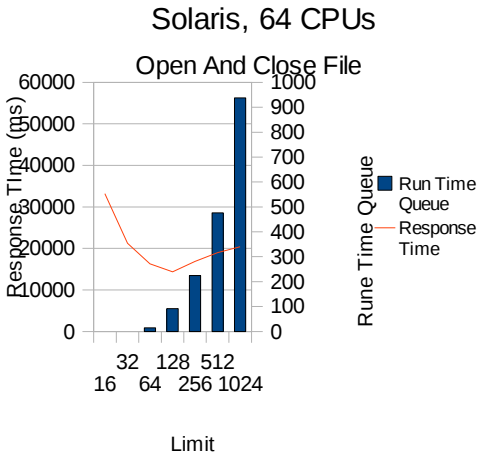

Diagram 4.

Simulator Settings
Clients: 1000
Requests: 2000
Loop: 40000
Comment: Request process throttle, concurrency limit is varied.

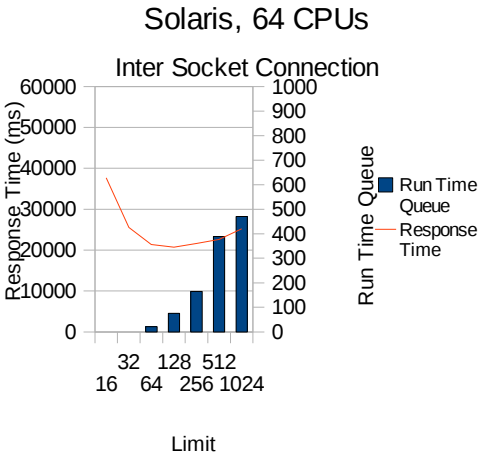

Diagram 5.

Simulator Settings
Clients: 1000
Requests: 2000
Loop: 40000
Comment: Request process throttle, concurrency limit is varied.

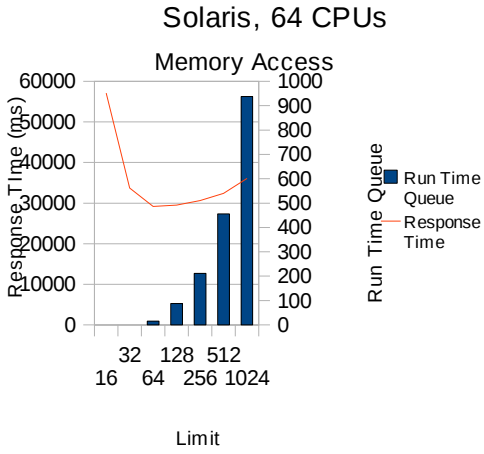

Diagram 6.

Simulator Settings
Clients: 1000
Requests: 2000
Loop: 40000
Comment: Request process throttle, concurrency limit is varied.

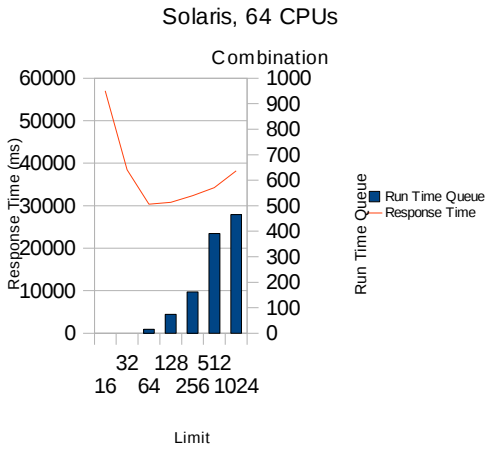

Diagram 7.

Simulator Settings
Clients: 1000
Requests: 2000
Loop: 40000
Comment: Request process throttle, concurrency limit is varied.

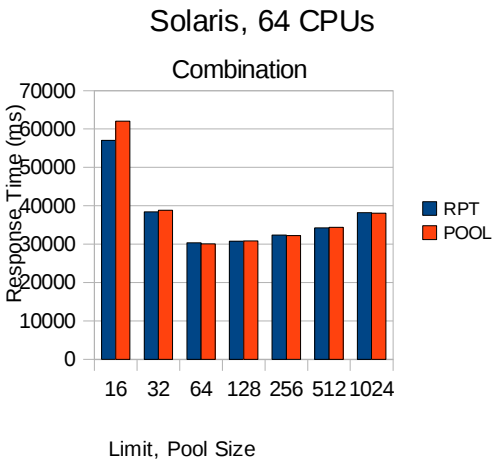

Diagram 8.

Simulator Settings
Clients: 1000
Requests: 2000
Loop: 40000
Comment: The thread pool has almost identical response times compared to the request process throttle. Concurrency limit and pool size of 64 gives best result.

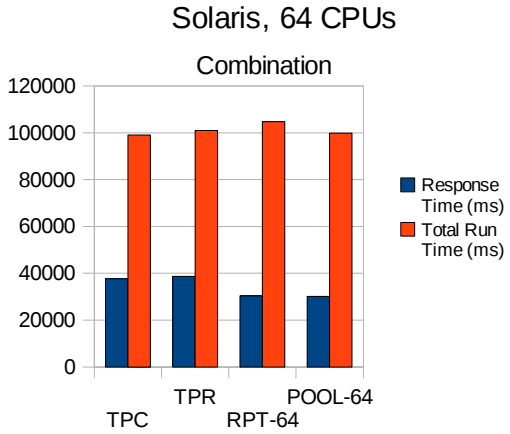

Diagram 9.

Simulator Settings
Clients: 1000
Requests: 2000
Loop: 40000
Comment: Request process throttle has been used with the limit of 64 and pool size is 64. We can see that there is little difference between request process throttle and thread pool.

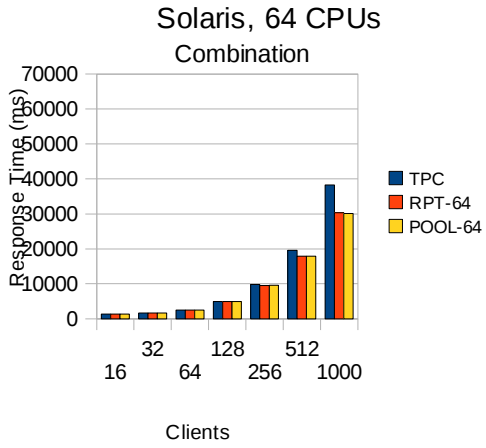

Diagram 10.

Simulator Settings
Clients: Varies
Requests: 2000
Loop: 40000
Comment: The number of clients have been varied between 16 and 1000.

5.1.2 Java version running on Solaris 10 with 64 CPUs

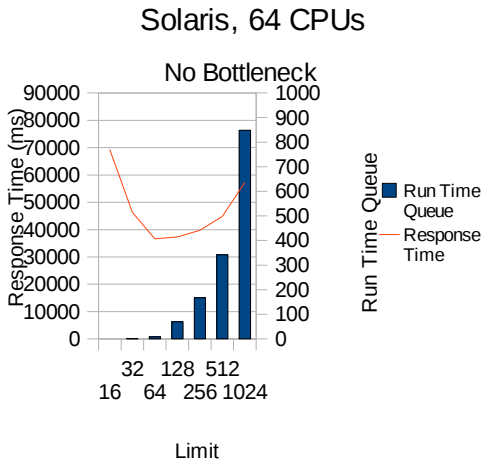

Diagram 11.

Simulator Settings
Clients: 1000
Requests: 2000
Loop: 40000
Comment: Request process throttle, concurrency limit is varied.

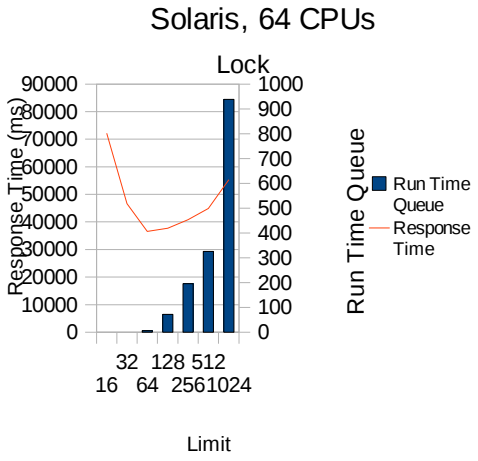

Diagram 12.

Simulator Settings
Clients: 1000
Requests: 2000
Loop: 40000
Comment: Request process throttle, concurrency limit is varied.

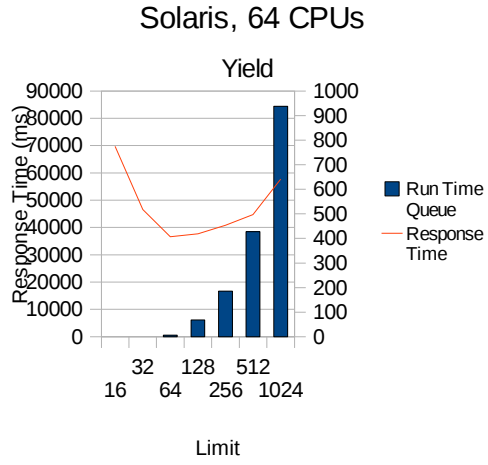

Diagram 13.

Simulator Settings
Clients: 1000
Requests: 2000
Loop: 40000
Comment: Request process throttle, concurrency limit is varied.

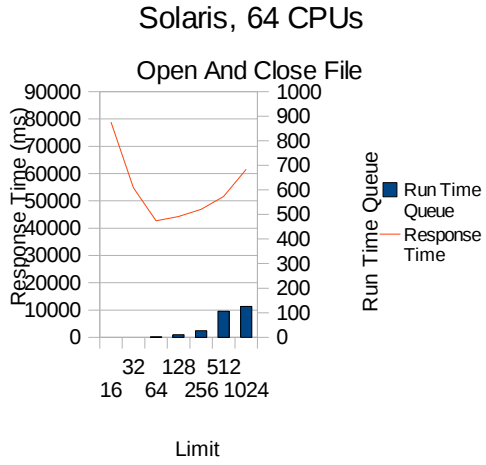

Diagram 14

Simulator Settings
Clients: 1000
Requests: 2000
Loop: 40000
Comment: Request process throttle, concurrency limit is varied.

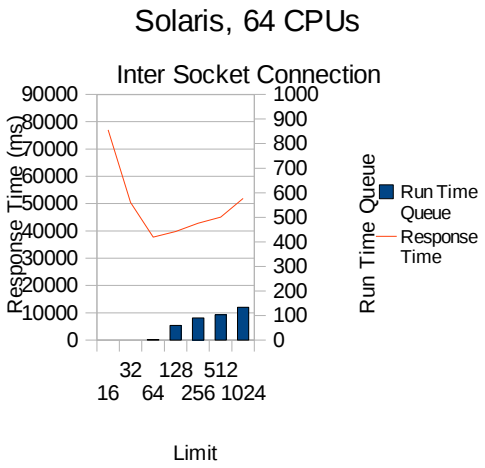

Diagram 15.

Simulator Settings
Clients: 1000
Requests: 2000
Loop: 40000
Comment: Request process throttle, concurrency limit is varied.

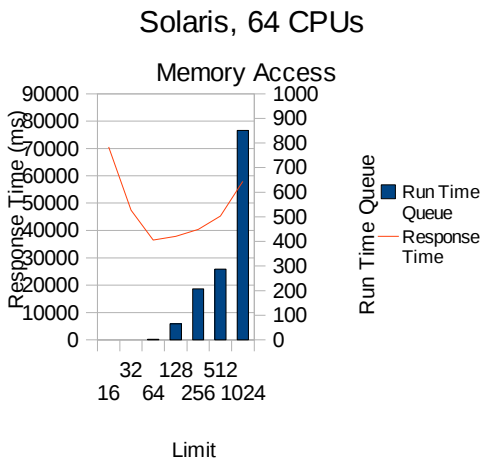

Diagram 16.

Simulator Settings
Clients: 1000
Requests: 2000
Loop: 40000
Comment: Request process throttle, concurrency limit is varied.

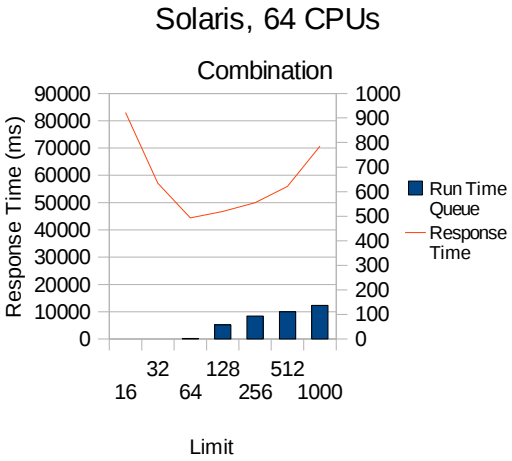

Diagram 17.

Simulator Settings
Clients: 1000
Requests: 2000
Loop: 40000
Comment: Request process throttle, concurrency limit is varied.

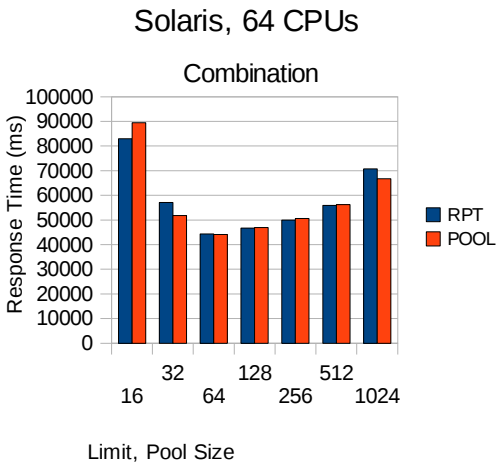

Diagram 18.

Simulator Settings
Clients: 1000
Requests: 2000
Loop: 40000
Comment: The thread pool has almost identical response times compared to the request process throttle. Concurrency limit and pool size of 64 gives best result.

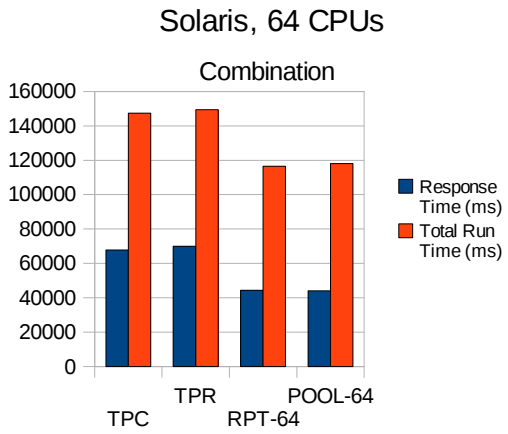

Diagram 19.

Simulator Settings
Clients: 1000
Requests: 2000
Loop: 40000
Comment: Request process throttle has been used with the limit of 64 and pool size is 64. We can see that there is little difference between request process throttle and thread pool.

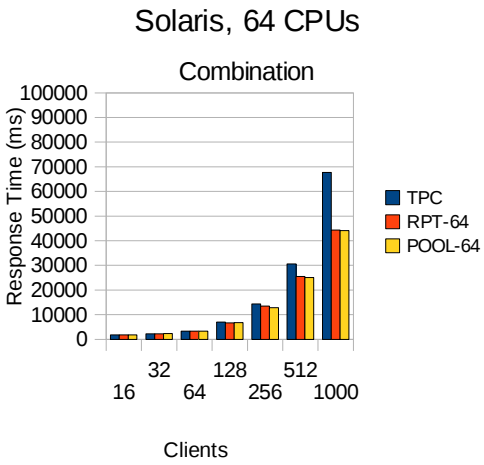

Diagram 20.

Simulator Settings
Clients: Varies
Requests: 2000
Loop: 40000
Comment: The number of clients have been varied between 16 – 1000.

5.1.3 C version running on Solaris 10 with 8 CPUs

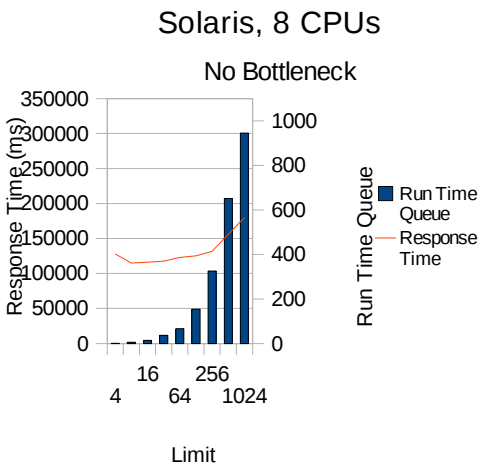

Diagram 21.

Simulator Settings
Clients: 1000
Requests: 1000
Loop: 40000
Comment: Request process throttle, concurrency limit is varied.

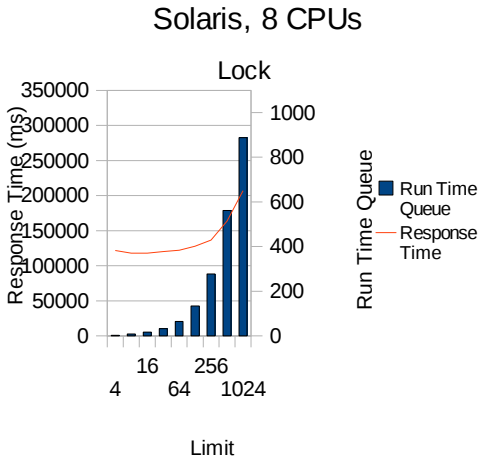

Diagram 22.

Simulator Settings
Clients: 1000
Requests: 1000
Loop: 40000
Comment: Request process throttle, concurrency limit is varied.

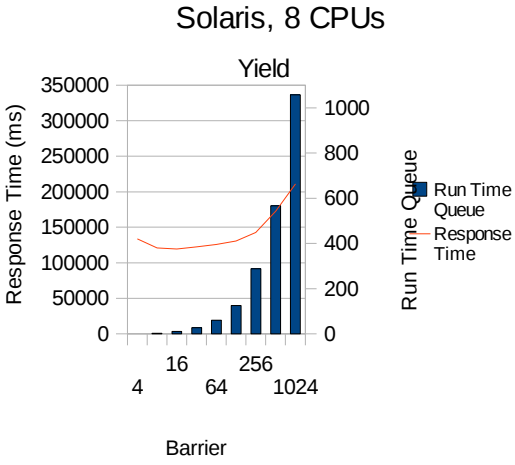

Diagram 23.

Simulator Settings
Clients: 1000
Requests: 1000
Loop: 40000
Comment: Request process throttle, concurrency limit is varied.

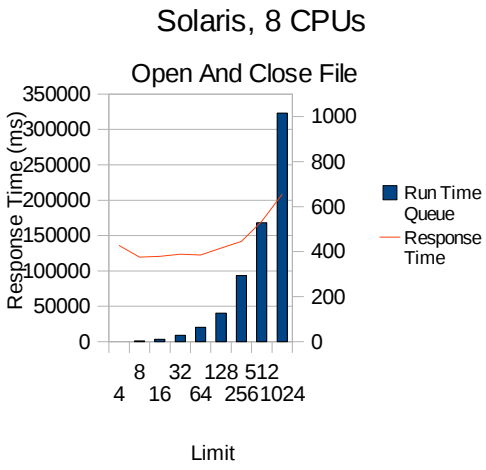

Diagram 24.

Simulator Settings
Clients: 1000
Requests: 1000
Loop: 40000
Comment: Request process throttle, concurrency limit is varied.

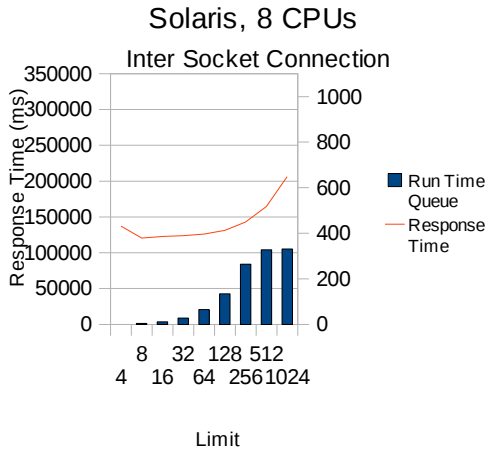

Diagram 25.

Simulator Settings
Clients: 1000
Requests: 1000
Loop: 40000
Comment: Request process throttle, concurrency limit is varied.

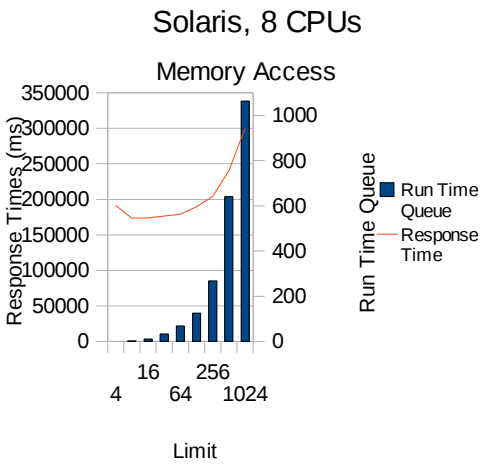

Diagram 26.

Simulator Settings
Clients: 1000
Requests: 1000
Loop: 40000
Comment: Request process throttle, concurrency limit is varied.

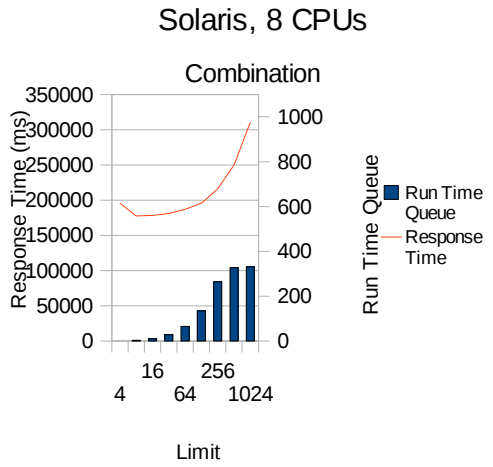

Diagram 27.

Simulator Settings
Clients: 1000
Requests: 1000
Loop: 40000
Comment: Request process throttle, concurrency limit is varied.

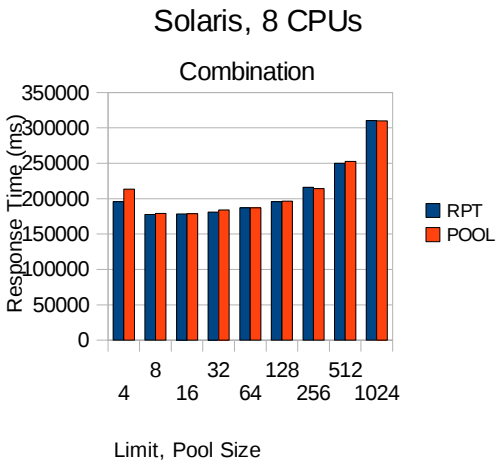

Diagram 28.

Simulator Settings
Clients: 1000
Requests: 1000
Loop: 40000
Comment: The thread pool has almost identical response times compared to the request process throttle. Concurrency limit of 8 and pool size of 16 gives best result.

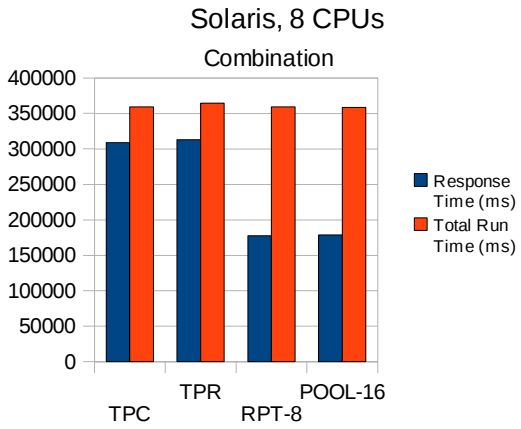

Diagram 29.

Simulator Settings
Clients: 1000
Requests: 1000
Loop: 40000
Comment: Request process throttle has been used with the limit of 8 and pool size is 16. We can see that there is little difference between request process throttle and thread pool.

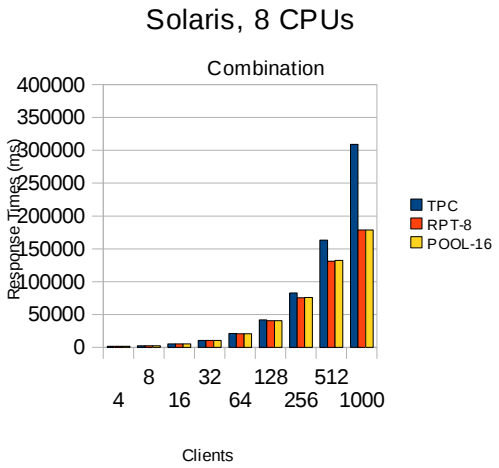

Diagram 30.

Simulator Settings
Clients: varies
Requests: 1000
Loop: 40000
Comment: The number of client varies between 4 – 1000.

5.1.4 Java version running on Solaris 10 with 8 CPUs

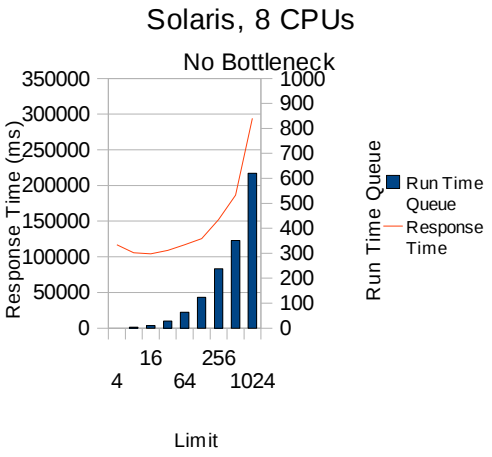

Diagram 31.

Simulator Settings
Clients: 1000
Requests: 1000
Loop: 20000
Comment: Request process throttle, concurrency limit is varied.

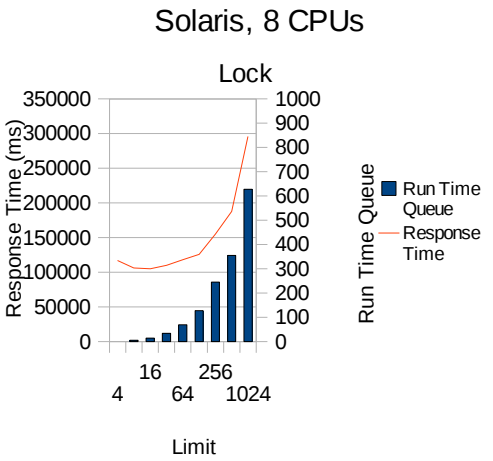

Diagram 32.

Simulator Settings
Clients: 1000
Requests: 1000
Loop: 20000
Comment: Request process throttle, concurrency limit is varied.

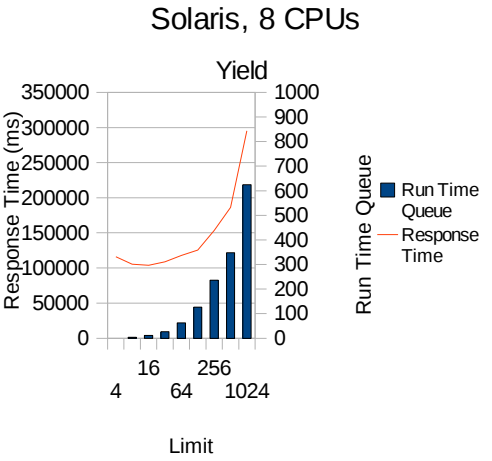

Diagram 33.

Simulator Settings
Clients: 1000
Requests: 1000
Loop: 20000
Comment: Request process throttle, concurrency limit is varied.

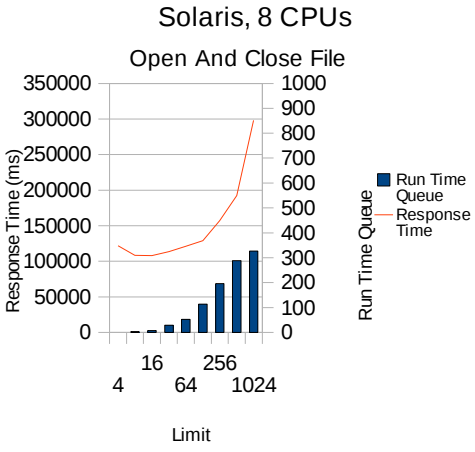

Diagram 34.

Simulator Settings
Clients: 1000
Requests: 1000
Loop: 20000
Comment: Request process throttle, concurrency limit is varied.

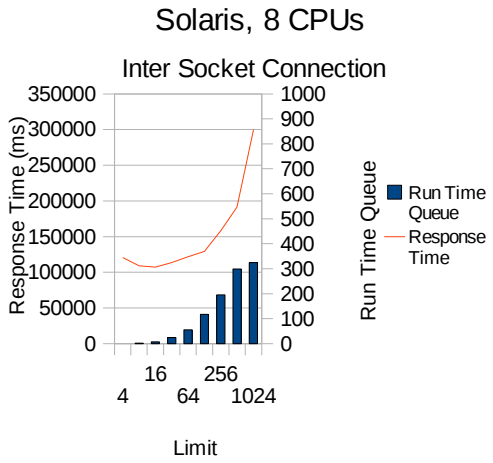

Diagram 35.

Simulator Settings
Clients: 1000
Requests: 1000
Loop: 20000
Comment: Request process throttle, concurrency limit is varied.

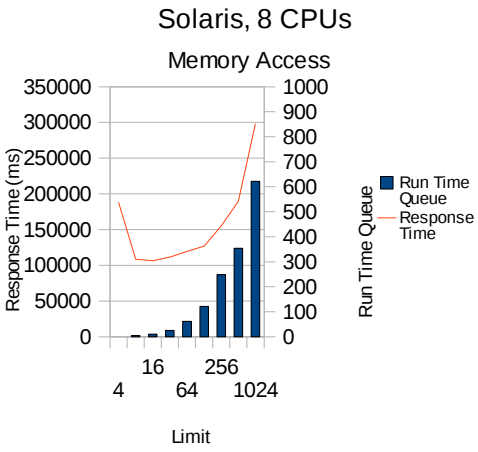

Diagram 36.

Simulator Settings
Clients: 1000
Requests: 1000
Loop: 20000
Comment: Request process throttle, concurrency limit is varied.

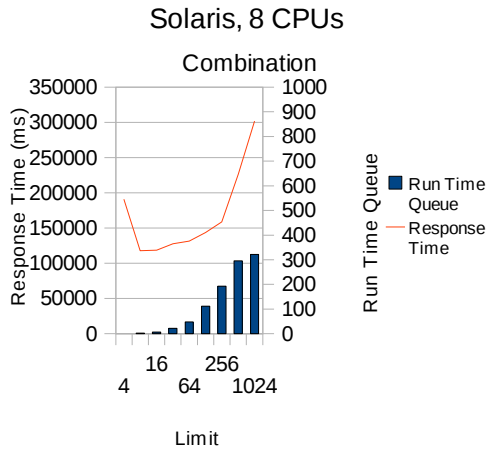

Diagram 37.

Simulator Settings
Clients: 1000
Requests: 1000
Loop: 20000
Comment: Request process throttle, concurrency limit is varied.

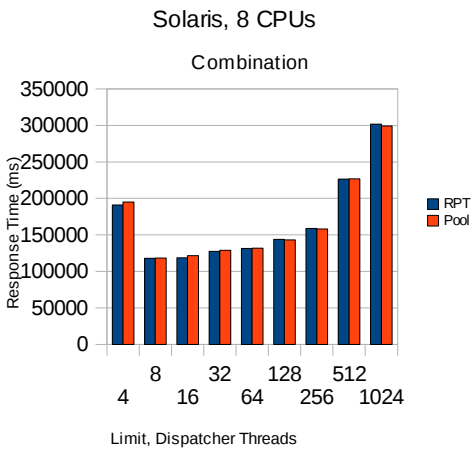

Diagram 38.

Simulator Settings
Clients: 1000
Requests: 1000
Loop: 20000
Comment: The thread pool has almost identical response times compared to the request process throttle. Concurrency limit and pool size of 8 gives best result.

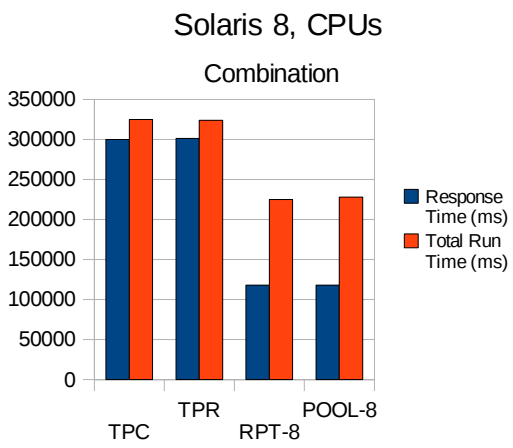

Diagram 39.

Simulator Settings
Clients: 1000
Requests: 1000
Loop: 20000
Comment: Request process throttle has been used with the limit of 8 and pool size is 8. We can see that there is little difference between request process throttle and thread pool.

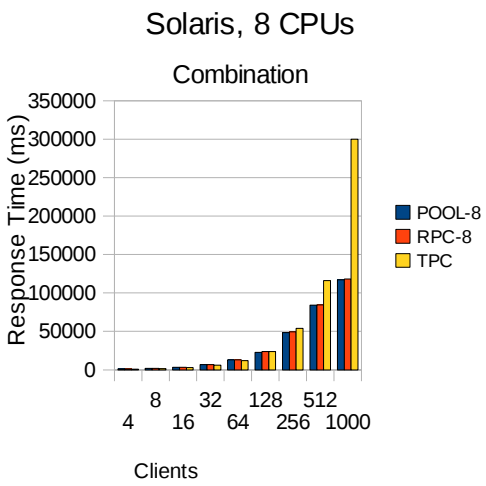

Diagram 40.

Simulator Settings
Clients: Varies
Requests: 1000
Loop: 20000
Comment: The number of clients varies between 4 – 1000. We can see that thread per client has an extreme value at 1000 connected connected clients.

5.1.5 C version running on Linux with 1 CPU

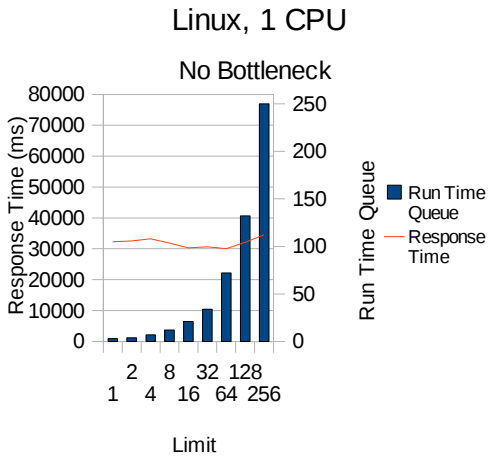

Diagram 41.

Simulator Settings
Clients: 250
Requests: 500
Loop: 40000
Comment: Request process throttle, concurrency limit is varied.

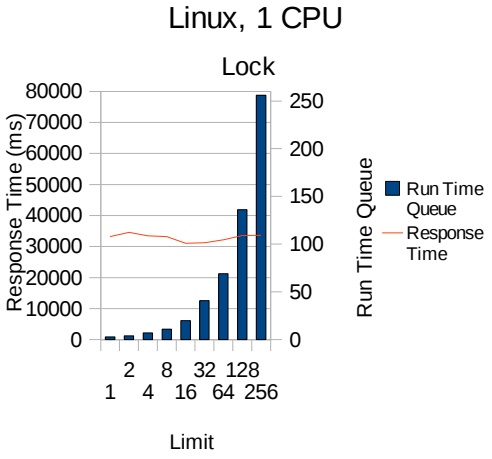

Diagram 42.

Simulator Settings
Clients: 250
Requests: 500
Loop: 40000
Comment: Request process throttle, concurrency limit is varied.

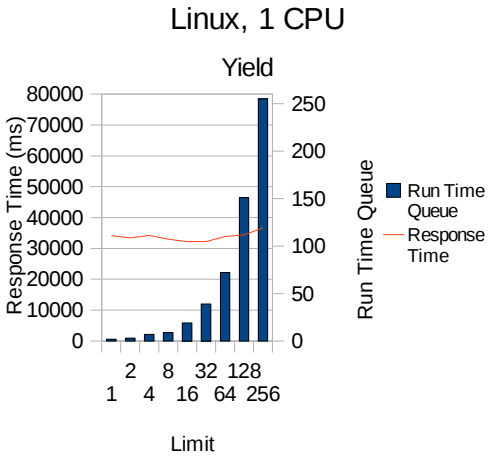

Diagram 43.

Simulator Settings
Clients: 250
Requests: 500
Loop: 40000
Comment: Request process throttle, concurrency limit is varied.

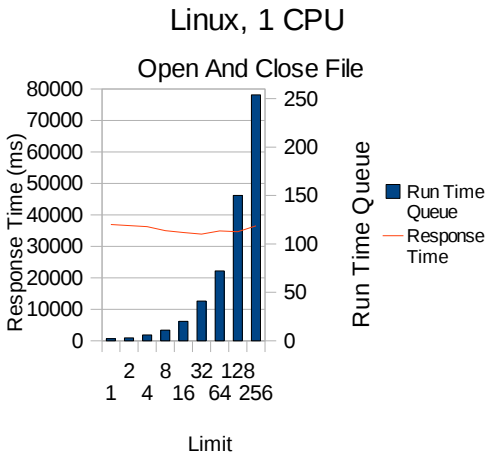

Diagram 44.

Simulator Settings
Clients: 250
Requests: 500
Loop: 40000
Comment: Request process throttle, concurrency limit is varied.

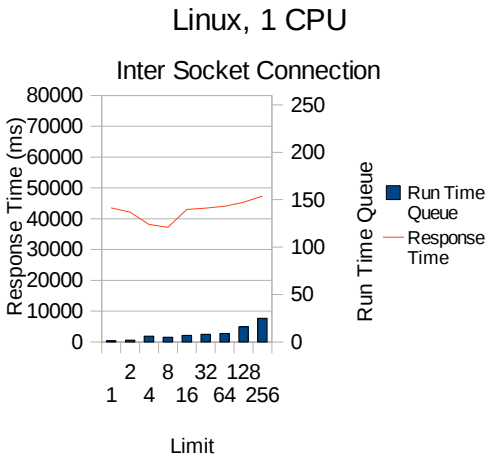

Diagram 45.

Simulator Settings
Clients: 250
Requests: 500
Loop: 40000
Comment: Request process throttle, concurrency limit is varied.

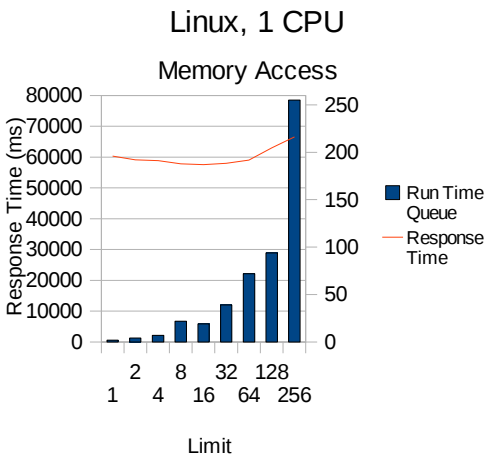

Diagram 46.

Simulator Settings
Clients: 250
Requests: 500
Loop: 40000
Comment: Request process throttle, concurrency limit is varied.

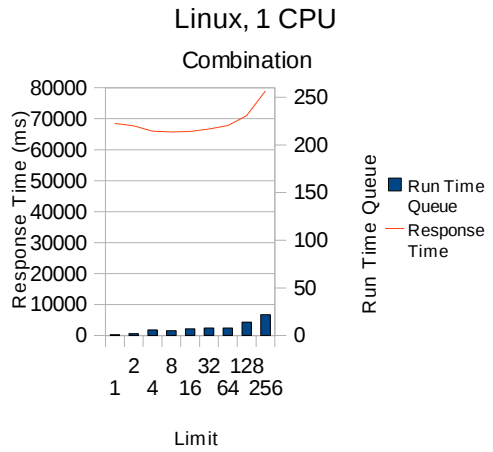

Diagram 47.

Simulator Settings
Clients: 250
Requests: 500
Loop: 40000
Comment: Request process throttle,

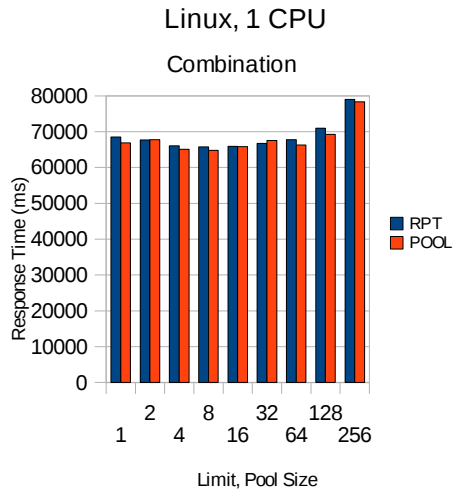

Diagram 48.

Simulator Settings
Clients: 250
Requests: 500
Loop: 40000
Comment: The thread pool has almost identical response times compared to the request process throttle. Concurrency limit and pool size of 8 gives best result.

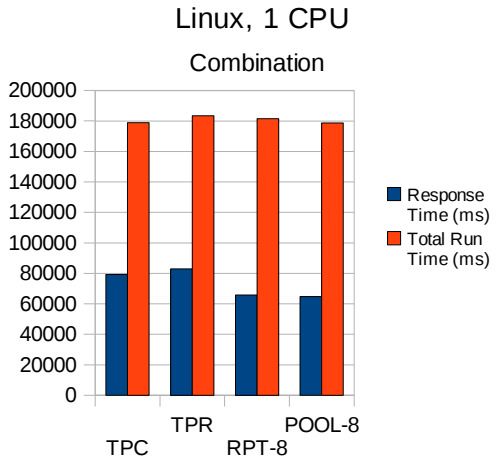

Diagram 49.

Simulator Settings
Clients: 250
Requests: 500
Loop: 40000
Comment: Request process throttle has been used with the limit of 8 and pool size is 8. We can see that there is little difference between request process throttle and thread pool.

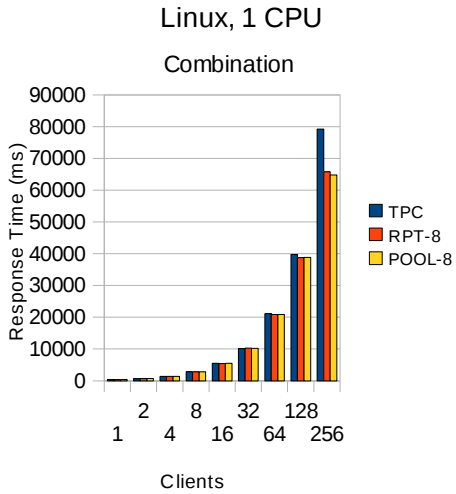

Diagram 50.

Simulator Settings
Clients: Varies
Requests: 500
Loop: 40000
Comment: The number of client varies between 1 -256.

5.2 CORRELATION

Correlation between run time queue length and average response time has been calculated for the combination load using Pearson [8] correlation. When calculating the correlation not all values where used, because from the measurements we can see that performance get worst when setting concurrency limit under the number of execution units available. We started using the values from concurrency limit equal to the number of executing units. A value of 1 meaning that there is a perfect correlation and 0 meaning no correlation.

Language / System	**Correlation**
C, 64 CPUs	0.95
Java, 64 CPUs	0.77
C, 8 CPUs	0.88
Java, 8 CPUs	0.93
C, 1 CPU	0.84

Table 1.

From the result in table 1 we can see that for all measurements there is a high correlation between run time queue and average response time.

6 DISCUSSION

As we expected before doing our simulations the performance for the request process throttle is best when setting the concurrency limit 1-2 times the number of execution units. This is also true for thread pool when setting the size of the pool. Our measurements that we have done also confirms this for the 64 CPUs and 8 CPUs system. This effect has been seen before [9] in telecom applications where they recommended setting number of threads to two times the number of processors used. We believe that the performance effect we are seeing on response times is that it is better to process a small amount of requests in parallel then process a huge amount. Let us give an simple example of this. In our example we use one CPU. If 10 requests that each require 1 second of the CPU and each request is processed before the next one is allowed to be processed, then we have an average response time of 5.5 seconds for a request. If the 10 requests are sliced up and they each require 0.5 + 0.5 seconds of the CPU, then the average response times are 10.5 seconds for a request. Also performance is affected by overhead added from thread contention.

For the 1 CPU system we can see that performance is best when setting the concurrency limit or the pool size to 8. Also this system is not a multi-core CPU and differs much in hardware architecture.

As mentioned before it was not possible to overload the Java version of the simulator on the 1 CPU system. The reason for this could be that the system is not powerful enough. But one important finding here is that when moving a server application from a single CPU system to a multi CPU system, then performance characteristics could be changed dramatically.

When comparing the result from different bottlenecks with each other used in this experiment, we cannot see any huge effect as we excepted when limiting the concurrency. Since this experiment was done in a controlled environment this could differ if we would redo this experiment using a real application server. In a real application there could exists more complex locking and resource behavior then there exists in our simulator.

Comparing the Java and C version measurements with each other we can see that for Java we can lower both response times and total run time. We must also keep in mind that another difference between the Java version and C is that Java uses a virtual machine [12] for executing byte code and C does not.

As it is possible to see in our measurements there is a connection between the length of the run time queue and average response times. We have also showed this by calculating the correlation. From this we draw the conclusion that it is possible to construct a self tuning request process throttle. The idea is to vary the concurrency limit for the throttle during run time. As the concurrency limit varies, the length of the run time queue changes. This can be seen in our measurements. The length of the run time queue can then be controlled by the throttle and should be tried to be kept at an optimal length. With optimal run time queue we mean that the length should not be more than 1-2 times the number of execution units for 64 CPUs and 8 CPUs system. For the 1 CPU system this shall be 8. The self tuning throttle comes with the benefit of eliminating the problem with client isolation. For example if one thread crashes, then this thread does not show up in the run time queue. Then the throttle will allow additional requests to be allowed to run and therefore the throttle concept is self healing. If the same problem would show up on the thread pool, then the dispatcher thread is lost and cannot be returned to the pool. Also the introduction of the throttle concept in a new server does only require minimal configuring, since the throttle itself determines the concurrency limit dynamically during run time. Also it is possible to apply the self tuning throttle on the thread pool and thread per request architecture.

7 CONCLUSION

Our results show that the choice of threading architecture is important when the server is under high load. The thread pool gives a significant higher performance, particular when it comes to response times. Also we could not identify any bottleneck that has high impact on server performance.

By applying a simple throttlling concept it is possible to make a thread per client architecture almost as effective as a thread pool when it comes to performance during high load. Performance is best for the throttle when setting the concurrency limit 1-2 times the number of execution units for 64 CPUs and 8 CPUs system. For the 1 CPU system performance is best when setting the concurrency limit to 8.

We have also identified that there is a high correlation between average response times and the length of the run time queue. This can be used to construct a self tuning throttling algorithm that makes the introduction of the throttle concept even simpler, since it does require any configuring.

REFERENCES

1. Multithreading in the SolarisTM Operating Environment, A Technical White Paper, Sun Microsystems, 2002.

2. Evaluating and Optimizing Thread Pool Strategies for Real-Time CORBA. Douglas C. Schmidt, Irfan Pyarali, Marina Spivak, Ron Cytron, 2001.

3. A Predictive Performance Model to Evaluate the Contention Cost in Application Servers, Shiping Chen, Ian Gorton, 2002.

4. Creating interface-controllers using model driven architecture, Carl Björk ,Per Salomonsson, 2004.

5. Performance Comparison of Middleware Threading Strategies, M. Harkema, B.M.M. Gijsen, R.D. van der Mei, L.J.M. Nieuwenhuis, 2004.

6. Java, http://java.sun.com [accessed 2009-09-01]

7. C, http://en.wikipedia.org/wiki/C_(programming_language) [accessed 2009-09-01]

8. Pearsons correlation, http://en.wikipedia.org/wiki/Pearson_product_moment_correlation_coefficient [accessed 2009-09-01]

9. Resource Allocation guidelines – Configure a large telecommunication application, Daniel Eriksson, 2001.

10. Context switch, http://en.wikipedia.org/wiki/Context_switch [accessed 2009-09-01]

11. Symmetric multiprocessing, http://en.wikipedia.org/wiki/Symmetric_multiprocessing [accessed 2009-09-01]

12. Virtual Machine, http://en.wikipedia.org/wiki/Virtual_machine [accessed 2009-09-01]

13. Memory Allocation Prevented Server Application to be Parallelized for Better Database Utilization, Daniel Häggander, Lars Lundberg, 1999.

14. Grand Central Dispatch, http://www.apple.com/macosx/technology/ [accessed2009-11-29]

15. Grand Central Dispatch, http://en.wikipedia.org/wiki/Grand_Central_Dispatch [accessed 2009-11-29]