RC24501 (C0802-004) February 15, 2008
Computer Science

IBM Research Report

Evaluating and Optimizing the Scalability of Multi-core SIP Proxy Server

Jia Zou [1,2], Zhiyong Liang[1], Yiqi Dai[2]

[1]IBM Research Division
China Research Laboratory
Building 19, Zhouguancun Software Park
8 Dongbeiwang West Road, Haidian District
Beijing, 100094
P.R.China

[2]Department of Computer Science and Technology
Tsinghua University
P.R. China

Evaluating and Optimizing the Scalability of Multi-core SIP Proxy Server

Jia Zou[1,2], Zhiyong Liang[1], Yiqi Dai[2]
[1]IBM China Research Lab, [2]Dept. of Computer Science and Technology, Tsinghua Uinversity
zouj03@mails.tsinghua.edu.cn, liangzhy@cn.ibm.com, dyq@theory.cs.tsinghua.edu.cn

Abstract

The Session Initiation Protocol (SIP) is one popular signaling protocol used in many collaborative applications like VoIP, instant messaging and presence. In this paper, we evaluate one well-known SIP proxy server (i.e. OpenSER) on two multi-core platforms: SUN Niagara and Intel Clovertown, which are installed with Solaris OS and Linux OS respectively. Through the evaluation, we identify three factors that determine the performance scalability of OpenSER server. One is inside the OSes: overhead from the coarse-grained locks used in the UDP socket layer. Others are specific to the multi-process programming model: 1.overhead caused by passing socket descriptors among processes; 2. overhead brought by sharing transaction objects among processes. To remedy these problems, we propose several incremental optimizations, including out-of-box dispatcher, light-weight connection dispatcher and dataset partition, then achieve the significant improvements: for UDP and TCP transport, on SUN Niagara, speedup are improved from 1.5 to 5.8 and from 2.2 to 6.2, respectively; on Intel Clovertown, speedup are improved from 1.2 to 3.1 and from 2.6 to 4.8, respectively.

1. Introduction

Throughout the history of modern computing, application developers have been able to rely on new processor chips to deliver significant performance improvement. Unfortunately, physical constraints on power consumption and heat dissipation have made this "free lunch" over [5]. Therefore, in recent years, chip vendors have introduced multi-core architecture as the strategy for continuing to increase computing power, e.g. SUN Niagara [6], Intel Clovertown [7], AMD Barcelona [8], IBM Cell [9], and so on. But, the multi-core trend will not "automatically" benefit all applications. People need understand workload characteristics on multi-core systems, and carefully design and develop their applications to fully exploit multi-core computing power.

The Session Initiation Protocol (SIP) [1] is used to create, modify and terminate sessions between two or more communication parties. It is one critical signaling protocol in 3G IP Multimedia Subsystem (IMS) and Next Generation Network (NGN). It is also becoming popular in the Internet and enterprise network for various collaborative applications, such as VoIP, instant messaging and presence. To enable these applications, several key SIP severs are defined in RFC 3261 [1], including proxy, redirect and registrar server. Their performance is very crucial for SIP-based network infrastructure.

However, to our best knowledge, how SIP server performs on multi-core systems is not well studied yet. In this paper, we explore the main factors that determine the scalability of SIP proxy server on multi-core systems. Particularly, we focus our efforts on the factors from the OS kernel and programming model, which are expected to be generic for other network servers with similar design.

We set up two testing servers running well-tuned OpenSER SIP proxy server [10], which is popular and widely used in the SIP community. One testing server is installed with one SUN Niagara eight-core chip [6] and Solaris OS. The other is installed with two Intel Clovertown quad-core chips [7] and Linux OS.

Our experiments firstly identify two problems in the implementation of OS UDP socket layer, which degrade the scalability significantly for UDP transport: 1. In Solaris OS, for the *recvfrom()* system call, a coarse-grained lock is used to serialize the access to the socket structure; 2. In Linux OS, a similar coarse-grained lock is implemented in the *sendto()* system call. Accordingly, OpenSER server suffers from receiving/sending the messages through one single UDP port. To remedy this problem, we design an out-of-box dispatcher to allow the OpenSER server to receive/send the messages through multiple UDP ports.

Having eliminated the above obstacle, we identify other two problems in the traditional multi-process (MP) programming model: 1. overhead caused by passing the socket descriptors of TCP connections among multiple processes, especially when using one single process for dispatching the socket descriptors; 2. synchronization overhead caused by sharing transaction objects among multiple processes.

To remedy the first problem, we modify the OpenSER server to enable each worker process to cache the TCP connections in its own local memory, so that they need not query the dispatcher process for the socket descriptors of TCP connections each time, and then significantly reduce the workload of the dispatcher process.

To remedy the second problem, we split the shared transaction objects into a number of smaller groups, and then divide one single shared memory block into multiple smaller memory blocks. In this way, we can effectively reduce the lock granularity and contention overhead.

By applying all the optimizations, we significantly improve the scalability of the OpenSER server: 1. on Sun Niagara from 1.5 to 5.8 for UDP transport, and from 2.2 to 6.2 for TCP transport; 2. on Intel Clovertown, from 1.2 to 3.1 for UDP transport, and from 2.6 to 4.8 for TCP transport. Although our work is based on SIP proxy server, the identified problems as well as our optimization techniques can also be applied for other network servers with similar design, especially those using UDP transport and MP-like programming model.

The remainder of paper is organized as follows. Section 2 introduces the background and motivation of our work. Section 3 describes the environment and methodology. Section 4, Section 5 and Section 6 presents the experimental results on performance evaluation and optimization. Section 7 discusses the related work. Section 8 concludes the whole paper and discusses the future work.

2. Background and Motivation

2.1 SIP Overview

SIP is an application-layer signaling protocol that can establish, modify, and terminate multimedia sessions, e.g. telephony call. SIP can also be used to invite participants to one existing session, e.g. multiparty conference.

Similar with HTTP and SMTP, SIP uses the text-based message format. SIP requests include INVITE (invite a peer to join a call/session), ACK (confirm receipt of message), BYE (terminate call/session), and etc. SIP responses are also similar with HTTP and SMTP responses, which are composed of a 3-digit number and an interpreting phrase, e.g. 100 Trying (provisional response indicating the effort to reach the target is occurring), 180 Ringing (provisional response indicating the phone is ringing), 200 OK (final response indicating the request is processed successfully).

A transaction is an important object defined in RFC 3261 [1]. A SIP transaction consists of one request and all the responses to this request, which can include zero or more provisional responses and one or more final responses.

In different application scenarios, various SIP servers are required, e.g. registrar server is used to provide the location service; proxy server is used to forward the messages between the end-points or proxies.

Figure 1 gives out a typical example of exchanging the SIP messages between two end-points in VoIP application.

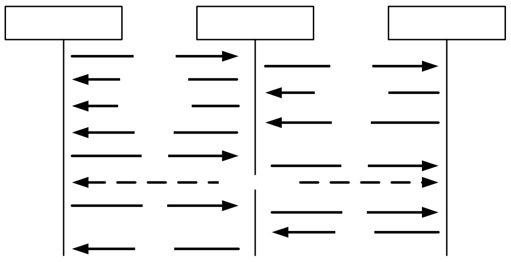


Figure 1: An example of exchanging SIP Messages

2.2 Programming Model for Network Server

In general, there are three basic programming models to develop a concurrent network server: multi-process (MP), multithreaded (MT), and event-driven state machine (EDSM) [13] [14] .

In the MP model, all the processing on one TCP connection or UDP message is handled by an individual process. In UNIX system, it's widely used in various network severs, such as Apache Web server [12] and OpenSER SIP server [10].

The popularity of MP programming model mainly lies in several advantages it can offer:

1. Robustness. Since each process has its own private address space, if one process crashes, other processes will not be affected.
2. Less synchronization overhead. Some standard library functions, such as *malloc()* and *free()*, use global locks to achieve thread safety. This will bring potential contention overhead to thread-based model [17]. However, MP model doesn't have such problem since global locks are not required for these library functions across multiple processes.
3. Easy implementation. In the MP model, programmer need less care about the synchronization than the MT model, and can utilize the abundant existing libraries, while EDSM architecture is monolithic and usually need to be implemented from the ground up [13] [19]

2.3 Potential Scalability Problems of a MP-based SIP Proxy Server on Multi-core Systems

We expect three obstacles for the performance scalability of a MP-based SIP proxy server with the increasing number of cores/hardware threads:

- Scalability of UDP protocol stack
- Overhead of passing socket descriptors among multiple processes;
- Overhead of sharing SIP transaction objects among multiple processes.

2.3.1. Scalability of UDP protocol stack. According to RFC 3261 [1], a SIP server is mandatory to support both UDP and TCP transport. Performance scalability of TCP protocol stack on multi-core systems was addressed in several previous work, e.g. [4] [15] . However as for UDP protocol stack, to our knowledge, there are not much research work to explore its scalability on multi-core systems yet. In this paper, we will examine the scalability of UDP protocol stack for supporting an upper-layer SIP proxy server.

2.3.2. Overhead of passing socket descriptors. In Unix environment, each process is allocated one open file table to store the indexes to file/socket objects which are stored in a kernel table shared by all processes. Passing a socket descriptor means the index to one socket object should be passed correctly from one process to another, so that multiple processes can share the same file/socket object.

Unix programming environment usually provides two approaches to pass socket descriptors: 1. use the *sendmsg()* and *recvmsg()* system calls to pass socket descriptors; 2. use STREAM pipe [14] [16] . Both the approaches may incur significant overhead. Especially when using one single dispatcher process to manage all the connections, this dispatcher may become the scalability bottleneck.

2.3.3. Overhead of sharing SIP transaction objects. According to RFC 3261 [1], a SIP proxy server can be either stateless or stateful. In order to implement some advanced functionalities like call accounting, forwarding on busy, voice-mail and etc., stateful processing will be needed.

In stateful processing, a proxy server need create a new transaction object when receiving a new request and update transaction state when receiving a retransmitted request or any response.

A transaction object is the context to do stateful processing. This context will be kept across multiple messages, e.g. in Figure.1, the proxy server need keep one transaction object across seven messages from INVITE request to 200OK response. Accordingly, with the MP model to implement a SIP proxy server, transaction objects have to be shared among multiple processes. Then, synchronization overhead to access these shared objects may become the scalability obstacle.

2.4 Goal of this Work

The goal of this work can be summarized as following:

- Measure the scalability of a MP-based SIP proxy server on two different multi-core systems;
- Identify the main factors that determine the scalability of a SIP proxy server on the two multi-core systems;
- Propose the solutions for any scalability problem identified.

3. Environment and Methodology

3.1 Software

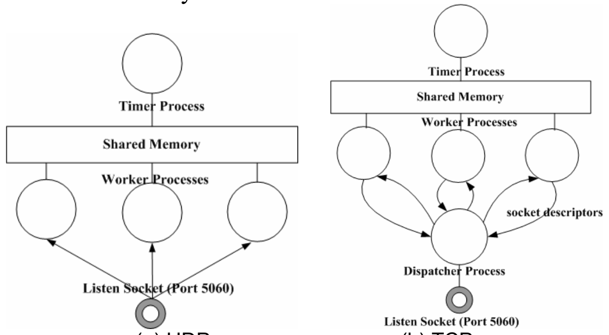
3.1.1. Software for SIP proxy server. In our experiments, we use the OpenSER server [10] as our testing server. OpenSER is an open source SIP proxy server, widely used in the SIP community, and well recognized with its high performance and reliability.

OpenSER uses two different MP-based programming models for UDP and TCP transport respectively. For UDP transport shown as Figure 2(a), multiple pre-forked worker processes will invoke blocking-mode *recvfrom()* to concurrently wait on the *well-known* port for incoming messages.

For TCP transport shown as Figure 2(b), a dispatcher process listens on the *well-known* port to accept each incoming connection, and pass its socket descriptor to one selected worker process using *sendmsg()* and *recvmsg()*. The selected worker process will watch and serve the messages over that connection until it is closed or timeout. Then the worker process will pass this dead connection to the dispatcher for cleaning up. In the meantime, if one worker process attempts to forward the message over an existing connection belonging to another process, it has to query the dispatcher for the connection socket descriptor.

All worker processes use shared memory as interprocess communication mechanism, using memory-mapped I/O (*mmap()*) [14] to map a file into shared buffer to share memory pool structures and transaction objects including transaction hash table and timer lists.

As shown in Figure 2, a single timer process will be forked to traverse timer lists to check expired timers and trigger corresponding callback functions. According to the expiring time duration of different timer events, some timer lists are traversed every ten milliseconds, and others are traversed every second.



(a) UDP (b) TCP

Figure 2: MP-based programming model in OpenSER

3.1.2. Software for Load Generators. In our experiments, we use multiple SIPp[11] instances to work as User Agent Client (UAC) and use one stateless OpenSER instance as User Agent server(UAS). SIPp is the popular software for SIP performance testing.

3.2. Hardware and OS Configurations

3.2.1. Hardware and OS for SIP proxy server. Since SIP proxy server may have different performance issues on different multi-core systems, we choose two typical multi-core systems as our test bed. One uses two Intel Clovertown quad-core chips [7] , and each core is with 2.2 GHz and exploits instruction-level parallelism techniques. The other uses one SUN Niagara eight-core chip [6] , which relies on explicitly thread-level parallelism. On Niagara, each core is with 1.2 GHz and has four hardware threads, and then it can support up to 32 different software threads or processes to be executing simultaneously on one single chip.

We install RedHat ES5.0 with a Linux 2.6.18 kernel on Intel Clovertown, and Solaris 10 on Sun Niagara.

3.2.2. Hardware and OS for load generators. We use four IBM blade servers to serve as UACs and another server to serve as UAS. All these machines are installed with Redhat ES 5.0.

3.3. Environment Setup and Measurement

We configure OpenSER as a stateful proxy server, and perform SIPstone Proxy 200 test [2] for both UDP and TCP transports. This test scenario is quite typical, and also used by other SIP performance evaluation work [3]. Message flow in this scenario is the same with the example shown in Figure 1 in Section 2.1.

In all the following experiments, the number of worker processes is set to be equal to the number of active hardware threads. For example, when enabling all hardware threads in 8 cores (4 hardware threads per core) on Niagara, we will start 32 worker processes.

We use the throughput as the performance metric. In our paper, if not specified, the throughput is defined to be the highest call rate (calls per second) with failure rate (completed calls per second / total calls per second) less than 0.01%, which is consistent with previous work [2] [3] .

To obtain OS statistics and functional profiling results, we use oprofile and mpstat on Intel Clovertown, and lockstat, prstat and dtrace on Sun Niagara.

4. Overview of Scalability Results

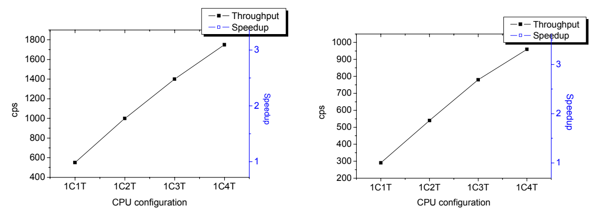
4.1. Single-core Baseline

Figure 3 presents the single-core results on Niagara system. We can see that the scalability of throughput with the increasing hardware threads is pretty good for both TCP and UDP transport, which indicates explicit thread-level parallelism on Niagara chip can significantly benefit SIP proxy workload. Figure 4 shows the single core results on Clovertown system.

From Figure 3 and Figure 4, one observation is that for both systems, OpenSER sever with UDP transport has better single-core throughput than TCP transport, i.e. 1.8x on Niagara and 2.1x on Clovertown. According to the profiling results, the main factor is that OpenSER server uses a dispatcher process for TCP transport, which brings additional socket descriptor passing overhead as well as context switch overhead.

4.2 Scalability Results

Figure 5 and Figure 6 shows the overall scalability results, which is very poor for both systems (where ideal speedup is 8). We will identify the main factors that cause such poor scalability and our corresponding optimizations step by step in later sections.



(a) UDP (b) TCP

Figure 3: Single-core results on Niagara system (xCyT denotes that x cores and y hardware threads per core are activated)

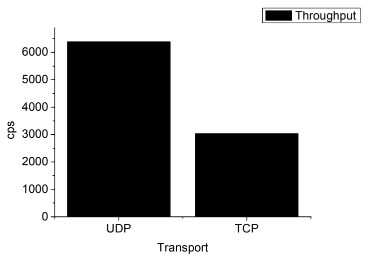
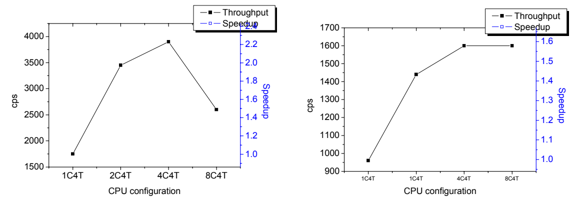
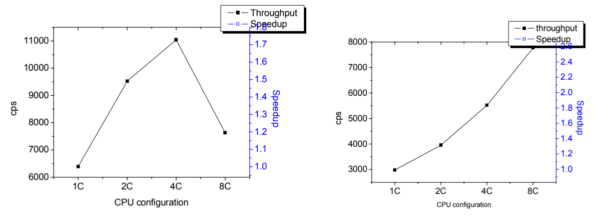


Figure 4: Single core results on Clovertown system



(a) UDP (b) TCP

Figure 5: Scalability results on Niagara system



(a) UDP (b) TCP

Figure 6: Scalability results on Clovertown system (*x*C denotes that *x* cores are activated)

5. Scalability Issues in UDP protocol Stack

5.1. Issues in Solaris UDP Protocol Stack

5.1.1. Problem identification. The overall scalability results of UDP transport on Niagara are shown in Figure 5(a). The poor scalability is mainly caused by a coarse-grained lock used for synchronizing accesses to UDP socket structure in the *recvfrom()* system call.

In Solaris, the system call *recvfrom()* corresponds with the kernel function *sotpi_recvmsg()*. Figure.7 shows the code path for *sotpi_recvmsg()*

kstrgetmsg() copies received packets from kernel socket buffer to user buffer. The conditional variable *SO_READ* is the mutex used to serialize the copy operations on one socket. Thus, the operation *mutex_vector_enter(so_lock)* which tries to acquire the spin lock *so_lock* will be frequently called when a number of processes invoke *recvfrom()* on the same socket. Besides, in the function *so_unlock_read()*, all processes that sleep for the mutex will be awaken, thus cause the “thundering herd” problem [18],where many processes will be waken up and spin together to contend for the *so_lock*, and accordingly significant CPU cycles will be wasted.

```
sotpi_recvmsg(){
    …

so_lock_read_intr();
    kstrgetmsg();
    so_unlock_read();
  …
 }
```

```
so_unlock_read(){
   SO_READ=0;
   cv_broadcast()
 }
```

```
so_lock_read_intr(){
   while (SO_READ==1)
   {
      cv_wait_sig();
      cv_block();
      swtch();
      mutex_vector_enter(so_lock)
   }
   SO_READ=1;
 }
```

Figure 7: Code path for Solaris *sotpi_recvmsg()*

According to our profiling results, in 8C4T case, the overhead of *recvfrom()*, *mutex_vector_enter()*, *sotpi_recvmsg()*, and *so_lock_read_intr()* is significant. The total overhead caused by the *recvfrom*() system call is larger than 50% CPU time. In result, as illustrated in Figure 8, for 8C4T, most of CPU time is spent in the kernel.

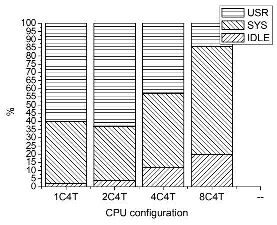


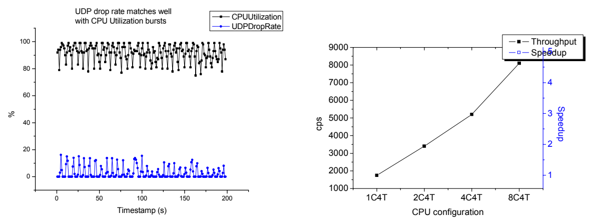
Figure 8: CPU utilization at the throughput points in figure 5(a)

5.1.2. Optimization. To solve the above problem, we configure the OpenSER sever to listen on multiple ports and design an out-of-box dispatcher to dispatch incoming packets evenly to those ports. With this optimization, the CPU time spent in the kernel space at the throughput point of 8C4T case has been significantly reduced from 66% to less than 30%. The total overhead caused by the *recvfrom()* system call has also been reduced from >50% CPU time to less than 10%.

However, we observe the jitters in both CPU utilization and packet drop ratio of UDP socket layer, as shown in Figure 9 (a). In the OpenSER server, the timer process periodically scans the timer lists and in the meantime, the worker processes also need access the timer lists for inserting or modifying the timers. Therefore, the worker processes will be stalled by the timer process when the timer lists are locked and scanned for a long time. So we infer that the jitters are mainly caused by such contention.

The jitters will bring the significant call failures. If these failures are counted to measure the throughput, we will see that the throughput isn't improved much although CPU utilization is reduced significantly. But, if we do not count in the failures, the speedup on Niagara is increased from 1.5 to 4.6, as shown in Figure 9(b).

For the jitter problem, we will deal with it in Section 6.2.



(a) Jitters at 8100cps, 8C4T (b) Scalability

Figure 9: Results for using out-of-box dispatcher on Niagara for UDP transport

5.2. Issues in Linux UDP Protocol Stack

5.2.1. Problem identification. Linux has a similar problem in the *sendto()* system call, where one coarse-grained lock is used. It causes the poor scalability for UDP transport on Clovertown system, shown as Figure 6(a).

In Figure 10, we can see that in the code path of the function *udp_sendmsg()*, which is the corresponding kernel function of *sendto()*, *lock_sock()* is invoked to synchronize the copy operation from kernel buffer to the user buffer. This coarse-grained lock will bring significant contention overhead.

```
udp_recvmsg()
{
     …
     lock_sock();
     ip_append_data();
     release_sock();
    …
 }
```

```
Ip_append_data(){
     …
   ip_generic_getfrag();
     …
}
```

```
ip_generic_getfrag(){
   …
  //copy data from kernel to user buffer
memcpy_fromiovecend()
   …
}
```

Figure 10: Code path for Linux *udp_sendmsg()*

5.2.2. Optimization. We solve the above problem in *sendto()* by modifying OpenSER codes to allow sending the packets over multiple sockets. The speedup is improved from 1.2 to 3.1, as shown in figure 11.

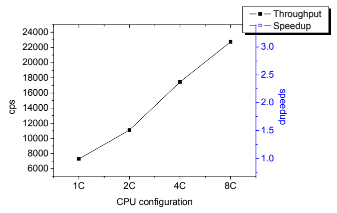


Figure 11: Scalability results for using multi-send-sock on Clovertown for UDP transport

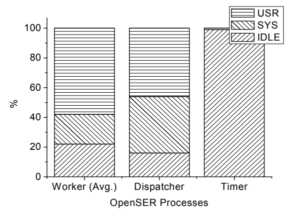
6. Performance Issues in Programming Model

After remedying the problems in UDP protocol stack, we further identify two performance issues in the MP programming model used by the OpenSER server: 1) Overhead of passing connection socket descriptors among multiple processes; 2) Synchronization overhead caused by sharing transaction objects among multiple processes.

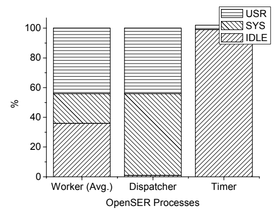
6.1. Overhead for Passing Socket Descriptors

6.1.1. Problem identification. As shown in Figure 6(b), on Niagara system, when the number of TCP worker processes is larger than 16, increasing the number of worker processes will not bring any further performance gain. That's mainly because worker processes query the dispatcher for passing the socket descriptors of TCP connections. This will be a heavy workload for the dispatcher.

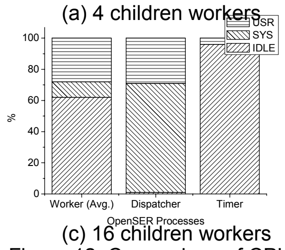
Figure 12 shows the breakdown of CPU time for Niagara system. It can be seen that when starting more than 8 worker processes on Niagara system, TCP dispatcher process will get overloaded ahead of other processes. In the meantime, the profiling results show that the functions relevant with passing socket descriptors: *send_fd()*, *receive_fd()*, *sendmsg()*, and *recvmsg()*, account for more than half of dispatcher's processing time. It shows that the overhead of passing socket descriptors brings the dispatcher process into a performance bottleneck.



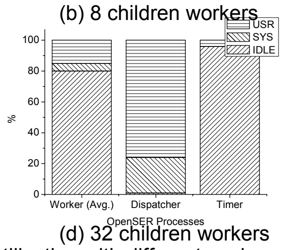
(a) 4 children workers



(b) 8 children workers



(c) 16 children workers



(d) 32 children workers

Figure 12: Comparison of CPU utilization with different workers

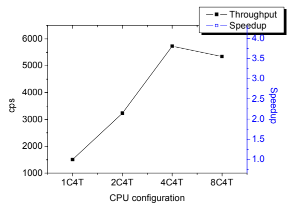
The profiling results on Clovertown system also show that although at the throughput point, the CPU utilization is less than 60%. The functions relevant with passing socket descriptors also account for significant overhead (more than 35% for 8C case).

6.1.2. Optimization and Results. To avoid the bottleneck of the central dispatcher, we propose the "lightweight connection dispatcher" optimization to offload the workload from the dispatcher to the worker processes, thus eliminating the central bottleneck. More specifically, each worker process will cache the connection socket descriptor in its local memory. Each time a worker process wants to send out a message over one connection, it firstly looks for the connection in its local memory, if not found, it will try to establish a new connection and cache this connection rather than query the dispatcher. Each worker process is in charge of cleaning all the connections cached in local memory rather than passing dead connections to the dispatcher for cleanup. By such way, we can not only reduce the overhead of passing socket descriptors but also reduce the context switch overhead, because the dispatcher process will be invoked less frequently.

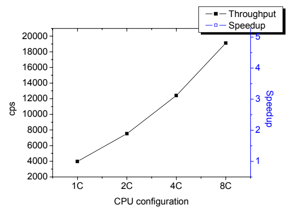
We implement the lightweight dispatcher optimization upon OpenSER server.

On Niagara system, the speedup is improved to 3.5. We can see that when worker processes are less than 16, scalability is close to linear (speedup from 1C4T to 4C4T is 3.8). However, when the number of worker processes increase further, the performance degrades. We will discuss the reason for this in Section 6.2.

The lightweight dispatcher optimization also can bring benefits to the Clovertown system. The speedup of the OpenSER server on Clovertown system has been improved by 85%, from 2.6 to 4.8, as shown in figure 13(b).



(a) Niagara



(b) Clovertown

Figure 13: Results for OpenSER with a lightweight dispatcher for TCP transport

6.2. Synchronization Overhead for Sharing Transaction Objects

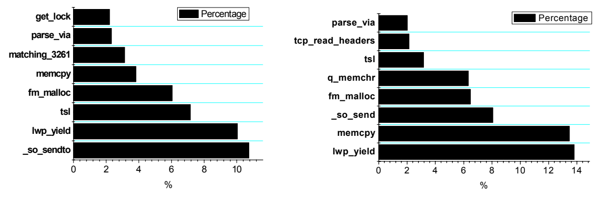
6.2.1. Problem identification. According to OpenSER's programming model, transaction objects are shared among the multiple worker processes, including three important data structures are shared: memory pool, timer list and transaction table.

OpenSER server implements its own spin lock utility in the user space. The function *tsl()* will spin to acquire a lock, and if the lock is not obtained after spinning for a number of times (default is 1024), *lwp_yield()* will be invoked to put the corresponding worker process to the end of the OS scheduling queue and schedule another process to run.

On Niagara system, for both UDP and TCP transports, after applying all the optimizations in the prior sections, when worker processes are more than 16, synchronization overhead on transaction objects becomes significant and dominant.

Shown as in Figure 14, we can see that the functions *tsl()* and *lwp_yield()* accounts for significant CPU time. Other profiling results show that the major function contributing to *lwp_yield()* is *set1_timer()*, which need to acquire the lock before scanning the timer lists for the expired timers. This will cause the jitters described in Section 5.1.2. Next two largest synchronization overheads are due to sharing memory pool and transaction hash table.

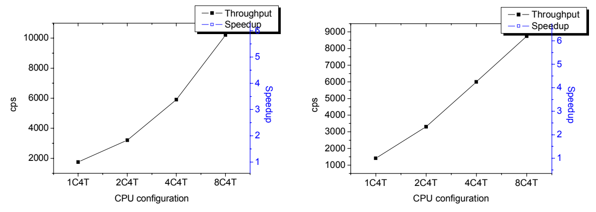
However, on Clovertown system, those functions only account for less than 1% of total overhead.



(a) UDP (b) TCP
Figure 14: Function profiling at the throughput point for 8C4T case on Niagara.

6.2.2. Optimization and Results. To solve the above problem, we propose the "dataset partition" optimization. Specifically, we use multiple shared memory blocks instead of one shared memory block, and each shared memory block contains its own timer lists, memory pool and transaction hash table. Then, for each incoming message, we use the Call-ID and CSeq values obtained from the corresponding fields of SIP messages to generate memory block ID, and all the information on the message will be stored in the memory block specified by the ID. We modify all the functions that need to access the shared data structure. For example, for the function *set1_timer()*, we convert it into scanning multiple smaller timer lists belonging to different memory blocks in a loop fashion.

We implement the "dataset partition" scheme upon the OpenSER server, using 32 shared memory blocks with the size of 256MB. Scalability results are shown in Figure 16, which show that such optimization brings significant benefits to Niagara system. On Niagara system, the speedup for UDP transport has been improved to 5.8, increased by 286%, if compared with the original case; the speedup for TCP transport has been improved to 6.2, increased by 182%. In addition, the jitters in CPU utilization and packet drop ratio described in Section 5.1.2 are not observed any more.



(a) UDP (b) TCP
Figure 15: Results for OpenSER with dataset partition on Niagara System

But the dataset partition optimization can not bring obvious performance gain to Clovertown system. That's mainly because much less worker processes are run on Clovertown system (i.e. 8 workers on Colvertown for 8C case, but 32 workers on Niagara for 8C4T case). In this situation, synchronization overhead to access transaction objects is not significant to the performance.

6.3. Discussion

Particularly, for Clovertown system, we also find some additional factors that will affect the performance:

1. By applying all the optimization, the scalability on Clovertown system is worse than Niagara system (i.e. for UDP transport, Niagara's speedup is 5.8, but Clovertown's is 3.1) This is mainly caused by IRQ affinity policy, which will direct all incoming network interrupts to one CPU for receive processing, thus this core will be one scalability bottleneck.

Although, we try to reduce such bottleneck effects by setting CPU affinity for the OpenSER processes to allow one core dedicated for packet receiving, this can not fully solve the problem. However, at the high load, this problem still becomes the scalability bottleneck. One possible solution to solve such problem is to use NIC designed with Receive-Side Scaling (RSS) [4] technology, which can bind each core with interrupts processing of a group of network flows. With RSS technology, packet receiving can happen in multiple CPUs so that above bottleneck can be avoided.

2. On Clovertown system, for a given core number, different core configuration can result in different performance. For example, the throughput for the configuration with two cores on the same chip is significantly higher than with two cores on different chips (e.g. for UDP transport, the throughput results are 9250 and 7250 respectively). We believe that is mainly due to the cache coherency overhead and cache locality. In our experiments, in order to obtain the best performance, we choose the configurations which enable physical cores as close as possible, e.g., for the case of two cores, we activate two cores sharing the same L2 cache on one chip; for the case of four cores, we activate four cores on one chip.

In addition, for Clovertown system and Niagara system, we have no intention to use the scalability results to suggest any one of the two systems is better than the other. That's mainly because they are too heterogeneous in many aspects of hardware design and OS, e.g. CPU frequency, cache size, OS network stack implementation, and etc., which will make such comparison elusive, complex and difficult to validate.

7. Related Work

As SIP has been adopted by more and more applications, there is a good deal of research work focusing on the performance issues of SIP server. Nahum et al [3] evaluate the performance of several common SIP scenarios like registrar, proxy, and proxy with authentication on the Intel single-core system. They also use OpenSER server as the testing server. Their work focus on how to design the comprehensive benchmark for various SIP severs. As to SIP optimization, Zou et al [20] design a scheme to offload SIP message parsing from the SIP server to a hardware accelerator. Janak [21] proposes the optimization for SIP server implementation, including lazy parsing, counted string, and memory pool. Compared with their work, our work is focus on identifying and solving the scalability issues of a SIP proxy server on multi-core systems.

As to the work around application performance on multi-core platforms, Veal et al[4] evaluate the scalability of Apache web server on one Intel multi-core system, examine a series of expected scalability obstacles and propose the solutions for the problems identified. Petrin et al [22] evaluate the performance of the Sweep3D software on Cell system, and identify a series of unexpected problems. Different from their work, we focus on the scalability of SIP protocol and our work helps to understand the scalability of SIP protocol on multi-core systems.

8. Conclusions and Future Work

In this paper, we evaluate a well-known OpenSER server on two different but typical multi-core systems: Intel Clovertown and Sun Niagara. In result, we identify three scalability issues on UDP protocol stack and MP programming model. We also propose and implement a series of incremental optimization techniques and achieve the significant performance improvements, as shown in Table 1. It is important to emphasize that the problems we've identified in this paper will also have similar effects on other network servers with similar design on the multi-core systems. Therefore, our optimizations are also helpful for those servers.

Table 1: Summary of speedup results

	original	out-of-box dispatcher	TCP lightweight dispatcher	dataset partition[1]
Niagara-UDP	1.5	4.6	-	5.8
Niagara-TCP	2.2	-	3.5	6.2
Clovertown-UDP	1.2	3.1	-	-
Clovertown-TCP	2.6	-	4.8	-

From this work, we also learn the fact that programming with the threads and locks on multi-core systems is time-consuming and error-prone. One potential solution is to design a new SIP programming framework which will divide messages into logically independent groups according to the session-ID. The framework guarantees that all messages belonging to the same session-ID will be dispatched to the same process for stateful processing. By this way, we can eliminate inter-process communication because the messages belonging to different call sessions need not share the states and timers. Accordingly, programmers don't need the efforts to synchronize those processes any more.

9. References

[1] J.Rosenberg, H.Schulzrinne and etc. "SIP: session initiation protocol", RFC3261, June, 2002

[2] H. Schulzrinne, S. Narayanan, J. Lennox, and M. Doyle, "SIPstone: benchmarking SIP server performance," http://www.sipstone.com, 2007

[3] E. M. Nahum, J. Tracey, and C. P. Wright, "Evaluating SIP server performance", ACM SIGMETRICS Performance Evaluation Review, Volume 35, issue 1. Pages 349-350, Jun 2007.

[4] B. Veal, A.Foong, "Performance scalability of a multi-core web server", Proc. of 3rd ACM/IEEE Symposium on Architecture for networking and communication systems (ACNS 2007), Pages 57-66, Dec 2007.

[5] H. Sutter, "The free lunch is over: a fundamental turn toward concurrency in software", Dr. Dobb's Journal, Volume 30, issue 3. Mar, 2005.

[6] P. Kongetira, K.Aingaran, and K. Olukotun, "Niagara: a 32-way multithreaded Sparc processor", IEEE Micro, Volume 25, issue 2. Pages 21–29, Mar/Apr 2006.

[7] "Multi-core from Intel – products and platforms", http://www.intel.com/multi-core/products.htm, 2006.

[8] AMD Opteron, http://multicore.amd.com/us-en/quadcore/.

[9] IBM Cell, http://www.research.ibm.com/cell/.

[10] OpenSER, http://www.openser.org

[11] SIPp, http://sipp.sourceforge.net/ 2006

[12] Apache Software Foundation, http://www.apache.org.

[13] D. E. Comer, D. L. Stevens, "Internetworking with TCP/IP, Vol. III: client-server programming and applications (fourth edition)", Pearson Education, Inc., 2002.

[14] W. Richard Stevens, "UNIX network programming (second edition)", Prentice Hall, 1998.

[15] S. Tripathi. "FireEngine—a new networking architecture for the Solaris operating system", White paper, Sun Microsystems, Nov. 2004.

[16] W. Richard Stevens,"Advanced programming in the UNIX environment", Addison-Wesley, Inc, 1992.

[17] C. Lever, D. Boreham, "Malloc() performance in a multithreaded Linux environment", Proceedings of the USENIX Annual Technical Conference (USENIX 2000), Pages 56-66, June 2000.

[18] "Thundering herd problem", http://en.wikipedia.org/wiki/Thundering_herd_problem

[19] "State threads for Internet applications", http://state-threads.sourceforge.net/docs/st.html.

[20] J. Zou, W. Xue, Z. Liang, and etc., "SIP parsing offload: design and performance evaluation", Proceedings of IEEE Global Telecommunications Conference (Globecom 2007), Page 2774-2779, Nov 2007.

[21] J. Janak. "SIP proxy server effectiveness," Master thesis, Department of Computer Science, Czech Technical University, Prague, Czech, May, 2003

[22] F. Petrini, G. Fossum, and etc., "Multicore surprises: lessons learned from optimizing Sweep3D on the Cell broadband engine", Proceedings of IEEE Parallel and Distributed Processing Symosium (IPDPS 2007). Page 1-10, Mar 2007.

[1] For the column "dataset partition", we use "out-of box dispatcher + dataset partition" optimization for UDP transport, and "lightweight TCP dispatcher + dataset partition" optimization for TCP transport.