On the Impact of Scheduler Settings on the Performance of Multi-Threaded SIP Servers

Ramesh Krishnamurthy[†], George N. Rouskas[†‡]
[†]North Carolina State University, Raleigh, NC 27695-8206 USA [‡]King Abdulaziz University, Jeddah, Saudi Arabia

Abstract—**Multi-threading is a widely used program execution model, where each thread executes independently while sharing some of the process resources. Multi-threaded processes are used for a range of network application servers including web servers, mail servers and SIP proxy servers (SPS) for Voice over IP (VoIP). The process scheduler is a core part of any Operating System and the policy it uses may have a significant impact on the various applications executing on the system. In this work, we investigate the impact of the Linux scheduler on the performance of OpenSIPS, an open source SIP proxy server. The version of Linux used in our study uses a scheduling policy known as "Completely Fair Scheduler" (CFS), and the Linux kernel provides several parameters that may be used to tune the CFS policy. We have collected a large set of experimental data, in a methodical fashion, to characterize the performance of SPS as a function of the number of server threads and the call arrival rate under (1) default CFS setting and (2) with CFS parameters tuned for improved performance. By fine tuning the scheduler parameters, SPS performance is improved in all scenarios, in some cases significantly. To the best of our knowledge, this is the first study that takes into account the scheduler parameters in improving the performance of the SPS. Our results indicate that network operators may increase server capacity without additional capital expenditures, by applying insightful configuration changes to scheduler policy.**

I. Introduction

Multi-threading is a widely used program execution model, where each thread executes independently of others. These threads share some of the process resources, including process state and memory space. Multi-threading can increase performance in terms of responsiveness by concurrent execution of these threads. The *process scheduler* is a core part of the operating system, and determines which process can execute on the system and the duration it runs. The scheduling policy needs to account for several objectives, including fairness, throughput and response time, which often may be contradictory. In this paper, we investigate the impact of the Linux Completely Fair Scheduler (CFS) on the performance of OpenSIPS, as a function of the number of threads and the call arrival rate.

In earlier work [1], we used waiting time as the metric for evaluating the performance of the SPS in a single-server, single-core system. We observed that the waiting time increases several orders of magnitude from a few microseconds to several milliseconds, as a function of the call arrival rate. Despite this relative increase in waiting time, the absolute values are sufficiently low that may not be noticeable to VoIP users. To provide a more representative performance measure of SPS for service providers, in this work we focus on SIP control packet drop rate as the key performance metric. Our contributions include: (1) collection of a large set of experimental data, in a methodical fashion, to characterize the performance of SPS as a function of the number of server threads and increasing call arrival rates; (2) characterization of the impact of the scheduler on the performance of a multi-threaded SPS, in terms of SIP control packet drop rate and waiting time; and (3) identification of the key scheduler parameters of CFS scheduler and concrete guidelines on tuning these parameters to achieve significant performance improvement.

There have been several studies in the literature on the impact of multi-threading on the performance of servers. The ones most relevant to our current research include [2]–[4]. In [2], the authors develop techniques to determine optimal allocation of threads for a specific Quality of Service (QoS) objective, and use realistic workloads on a typical web server to show the efficacy of the new methodology. In [3], the complexity of the optimal configuration of a multi-threaded web server for different workloads was explored. In [4], a simulation study was carried out to investigate the impact of multi-threading on cache performance. Several studies investigate the impact of the process scheduler in various contexts. An approach to improve the interactivity of user tasks in an Android smartphone environment by passing information about the user task from the Android framework layer to the underlying Linux CFS is implemented in [5]. In [6], the Linux scheduler was analyzed under the presence of network I/O and a certain parameter was tuned to mitigate starvation experienced by some processes. In [7], the authors designed a feedback method between the user process and the Linux scheduler so as to lower the global power budget. Our work differs from these studies in that: (1) we focus on the performance of SPS in terms of SIP control packet drop rate; (2) we modify the kernel and SPS code to obtain accurate packet measurements; and (3) we study the impact of the scheduler and tune its parameters so as to improve the SPS performance. There have also been several attempts in the literature to characterize the performance of the SPS, including [8]–[10]. These studies make specific assumptions about the service time and its distribution, whereas we have conducted extensive experiments to get an accurate measure of service time, waiting time and packet drop rate.

Following the introduction, we describe the experimental testbed in Section II, and discuss the measurement methodology in Section III. In Section IV, we identify and tune the CFS scheduler parameters, and present experimental performance data. We conclude the paper in Section V.

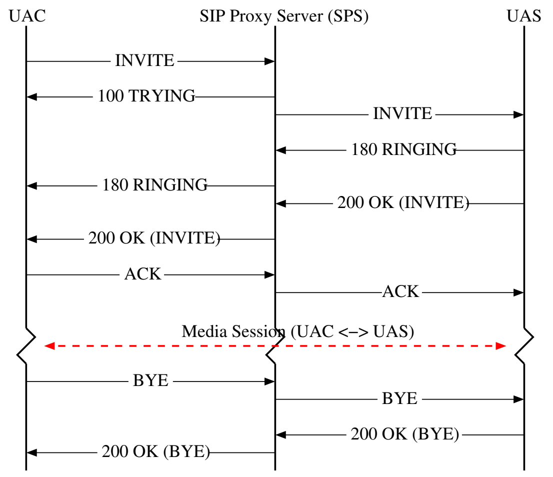

Fig. 1. SIP message exchange for call setup and teardown

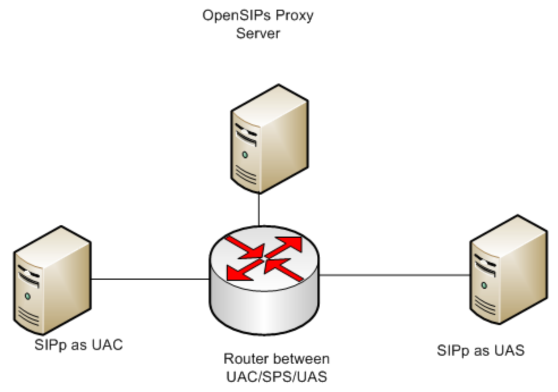

Fig. 2. Testbed for performance measurements of OpenSIPS SPS

II. Testbed and Experimental Setup

SIP is an application layer signaling protocol that can establish, modify, and terminate multimedia sessions such as Internet telephony calls [11], [12]. The SPS is a key component of the SIP infrastructure, and handles all SIP messages generated by the User-Agent Client (UAC) and User-Agent Server (UAS) during setup, modification or termination of each media session. In case of overload, the SPS may become a performance bottleneck that limits the ability of users to establish SIP sessions. Consequently, we only focus on the SPS in this work.

Figure 1 shows the exchange of SIP messages between the UAC and UAS through an SPS, for both the call setup and teardown operations. This is the message flow that we use in our experimental data collection and in measuring the SPS performance. Figure 2 shows the network testbed we used to generate SIP calls and collect measurement data so as to characterize the performance of the SPS. The hardware setup consists of:

- **OpenSIPS SPS.** OpenSIPS [13] is an open source implementation of a SIP proxy server. We installed the OpenSIPS SPS on a quad-core Intel® Xeon™ E5540 @2.53GHz processor with 8192 KB cache running the Debian 6.0.2 Linux distribution (2.6.32 Linux operating system). All our experiments were conducted after disabling two cores of available four cores and assigning the SPS process to one active core and `syslogd` to the other active core; this setup emulates a single processor environment for SPS process threads. We configured UDP as the transport layer protocol, and we modified the default configuration file (i.e., `opensips.cfg`) to set the number of children to the required number m of threads; in our experiments, we let $m = 2, 4, 6, 8, 16$.
- **SIPp UAC and UAS.** SIPp is a free open source test tool and traffic generator for the SIP protocol. For UAC and UAS we used two separate machines with quad Intel® Xeon® CPU E5540 @2.53GHz processors, running Redhat Linux 4.1.2-44 (Linux kernel version 2.6.18) OS.

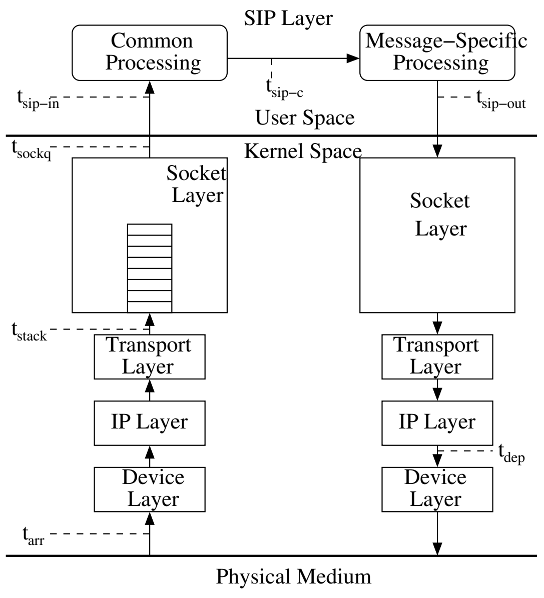

Fig. 3. Path of packets through the Linux network stack and SIP layer

III. Measurement Methodology and Experiments

Our goal is to get exact time measurements for each SIP packet in the SPS, from the arrival instant (i.e., the time it is received by the device driver) to the departure instant (i.e., the time it is transmitted by the device driver after it has undergone processing at the SIP layer). Figure 3 illustrates the packet receiving and sending operations within the Linux kernel network stack. We identify three main components that make up the total time a packet spends within the SPS:

1) K_{rcv}: This component represents the time spent within the kernel from the instant the packet is received at the kernel device layer until the instant it is handed off to the SIP layer. Note that this component consists of four distinct subcomponents: K_{stack}, the time it takes the packet to undergo processing at the device, network, and transport layers; K^{w}_{sockq}, the time the packet spends waiting at the socket queue before it can be delivered to the application (SIP) layer; K^{s}_{sockq}, the time it takes the kernel to process the packet once it is in

the queue, including the time to wake the receiving user level process to indicate the availability of data and handling the dequeue request from the user process; and K_{copy}, the time needed to copy the data from the kernel space to user space. Clearly, $K_{rcv} = K_{stack} + K^{w}_{sockq} + K^{s}_{sockq} + K_{copy}$.

2) T_{sip}: This component represents the time that the packet undergoes processing within the SIP layer. The SIP layer receives one packet at a time from the socket, processes it, and passes it to the kernel for forwarding before it receives the next packet from the socket. Therefore, T_{sip} reflects the service time of the packet within the SIP layer, and does not include any waiting time.

3) K_{snd}: This component represents the time it takes the packet to traverse the kernel on the sending side, until it is transmitted on the physical medium. The SIP layer passes one packet to the kernel at a time and then is blocked till the kernel has processed the packet and sent it to the device layer.

Figure 3 shows the instances at which we record the timestamps for the SIP packet at it moves through the SPS. Each recorded data is logged as the packet is processed. Post-experiment processing of logged data is performed to obtain the waiting and service times for the packet.

The main time components K_{rcv}, T_{sip} and K_{snd} are calculated using the recorded timestamps as shown in Figure 3. The kernel and SPS time components are obtained using the following calculations from the recorded timestamps:

$$\begin{aligned} K_{rcv} &= t_{sip-in} - t_{arr}, \\ K_{sockq} = K^{s}_{sockq} + K^{w}_{sockq} &= t_{sockq} - t_{stack}, \\ T_{sip} &= t_{sip-out} - t_{sip-in}, \\ K_{snd} &= t_{dep} - t_{sip-out}. \end{aligned} \tag{1}$$

In our experiments, we initiated calls between the UAC and UAS via the SPS. For each experiment, 200,000 calls were started. For each call, the messages exchanged between the UAC and UAS are as shown in Figure 1. For each message processed by the SPS, we measured the time components described above to determine the waiting and service times of the message through the SPS; we also kept a count of any messages dropped. To ensure that the performance of the SPS was not affected by any other process and did not take advantage of multiple cores, all SPS threads were configured to run on one core of the quad-core processor; `syslogd` was run on a different core, and the other two cores were disabled. The study of a multi-threaded SPS on multiple cores is the subject of our ongoing work.

Each experiment is characterized by two parameters:

1) *Call arrival rate.* The call arrival rate range was from 200 cps to 4000 cps, where each call results in six arrivals at the SPS as shown in Figure 1. The maximum call rate was set to the point where the SPS starts getting overloaded and beyond which no meaningful result can be obtained. We modified the SIPp tool (see http://rouskas.csc.ncsu.edu/Projects/SPS/) to generate call arrivals using a specified time between inter-arrivals. We generated the times separately such that these inter-arrivals were exponentially distributed around the mean call rate that we desired. The modified SIPp tool was used at the UAC to initiate the calls.

2) *The number of server threads,* The experiments were conducted for 2, 4, 6, 8 and 16 server threads.

Upon completion of an experiment for a specific call rate and number of server threads, we process the logged data and obtain the sample mean values for K_{rcv}, K_{stack}, K_{sockq} T_{sip}, and K_{snd}; we obtain the mean value for each SIP message type as well as the over-all mean across all six message types for each of our experiments. We also estimate 95% confidence intervals around the overall mean. From these values, we define the overall service time as $E[x] = K_{stack} + K^{s}_{sockq} + K_{copy} + T_{sip} + K_{snd}$. In addition, for each experiment, the overall drop rate is measured by using the `netstat` command, to obtain the statistics for packets in the system.

IV. Impact of Process Scheduler on SPS Performance

The main purpose of the process scheduler is to provide fairness among different processes, maintain high-throughput for the system and achieve maximum utilization of the CPU. The Linux kernel uses a scheduling mechanism called completely fair scheduling (CFS) [14]. CFS is a variant of weighted fair queuing (WFQ), and its objective is to maintain fairness in providing processor time to tasks. CFS uses red-black trees to get the next process to run based on the concept of a "virtual run-time".

The main design principle of CFS is to model an ideal, precise multi-tasking CPU. Note that a CPU can run only a single task at a given time, while other tasks are waiting. To ensure fair access to the CPU across all tasks, CFS tracks a task's "fairness imbalance" via a per-task variable referred to as `wait_runtime`, that captures the task's waiting time. `wait_runtime` is the amount of time the task should be allowed to run on the CPU under completely fair and balanced scheduling. CFS tries to enforce fairness among all its runnable tasks by scheduling the task that has the maximum `wait_runtime` value and, thus, is most in need of CPU time.

CFS also encompasses the concept of `group scheduling`, introduced with the 2.6.24 kernel. Group scheduling allows the scheduler to provide fair access to CPU time across all tasks in the system, and enforces hierarchical fairness among tasks, when a task spawns multiple child tasks.

CFS uses nanosecond granularity to account for the process times. Linux provides several parameters for tuning the behavior of the CFS scheduler, including the following three that we considered in our study:

- `sched_latency_ns`: A period in which each task runs once.
- `sched_min_granularity_ns`: The minimum time after which a task becomes eligible to be preempted. The scheduler tries to maintain this equality:

$$\texttt{sched\_min\_granularity\_ns} = \frac{\texttt{sched\_latency\_ns}}{\texttt{nr\_tasks}}$$

where `nr_tasks` is the number of tasks in the queue. If the equality is not met, the CFS scheduler tries to increase the `sched_latency_ns` time to match the increased number of tasks in the queue.

- `sched_wakeup_granularity_ns`: The ability of the task being awaken to preempt the current task. A larger value for this parameter makes it difficult for other tasks to force preemption. This parameter is used to reduce overscheduling.

The above parameters may be used to tune the scheduler's behavior to "desktop" workloads (where the objective is low latency) or "server" workloads (where the goal is to achieve good batching of jobs). The scheduler defaults to a setting suitable for desktop workloads. The values of these parameters are a function of the number of CPUs in the system. For the two-core system used for our experiments, the default values are: `sched_latency_ns` = 10,000,000 (10 ms), `sched_min_granularity_n` = 2,000,000 (2 ms), and `sched_wakeup_granularity_ns` = 2,000,000 (2 ms).

A. Baseline Server Mode

In our preliminary experiments, we determined that configuring the scheduler in "desktop" mode results in poor performance for the SPS. Therefore, following the recommendations in [15], we modified the values of the three scheduler parameters to move the default scheduler policy to "server" mode, as follows: `sched_latency_ns` = 1,000,000 (1 ms), `sched_min_granularity_n` = 100,000 (100 μ s), and `sched_wakeup_granularity_ns` = 25,000 (25 μ s). With these values, the scheduler allows the threads that are spawned as part of the server to be scheduled more often, improving the overall system performance. We will refer to this configuration as the *baseline* "server" mode; we note that these parameter values are recommended in [15] as a generic server configuration and do not take into account characteristics or workloads specific to SPS.

Consider an SPS process with multiple server threads. As the system load increases, context switching overhead and data cache pollution increases, hence performance suffers. Similarly, as the number of threads in the system increases, there is further process scheduling overhead, as well as pollution of the instruction cache, in addition to the data cache pollution. Therefore, it is clear that no set of fixed values for the scheduler parameters will work well across the range of system loads and number of threads under which a server is expected to operate. As we mentioned above, CFS attempts to adapt to an increase in the system load (i.e., an increase in the number of tasks in the queue) by increasing the value of parameter `sched_latency_ns`. By doing so, in effect, CFS relaxes the fairness policy so as to reduce the context switching overhead; as a result, the CPU is better utilized and system throughput increases.

B. Enhanced Server Mode

Based on the above observations, we have carried out a large number of experiments to measure the impact of the three scheduler parameters identified above on three performance metrics: (1) the service time, T_{sip} of a packet in the SIP layer, (2) the kernel time, K_{rcv} that includes the waiting time at the socket queue, and (3) the drop rate of SIP control packets, measured by the ratio of `RcvErrors` to `Total number of Messages` provided by the `netstat` command. Our finding are as follows.

1) *sched_latency_ns*: Setting this parameter to the fixed value 800,000, independent of the number of threads, achieved the best results. Recall that this parameter controls the latency of CPU bound tasks, and is dynamically adjusted by the scheduler in response to variations in the system load (in our case, packet arrival rate). Therefore, using a low value for this parameter provides the scheduler with significant flexibility in adjusting this value upwards to control the context switching overhead following an increase in system load.
2) *sched_min_granularity_ns*: This parameter controls the amount of time that tasks may run without preemption. Therefore, it is desirable to set it to a value that corresponds to the amount of time needed to complete a task (in this case, process a SIP packet), so as to minimize context switching overhead. To this end, we set this parameter to a quantity that roughly corresponds to the measured value of mean service time T_{sip} at the point where the system started experiencing overload. The specific values we used for this parameter were 100,000, 150,000, 200,000, 250,000, and 400,000, for 2, 4, 6, 8 and 16 server threads, respectively.
3) *sched_wakeup_granularity_ns*: This parameter controls the wake-up latency of a task, i.e., the amount of time it must lapse before it can preempt the current task. Since we set the amount of time that a task may run without preemption to a value that ensures that most tasks will complete before being preempted (see the discussion on the second parameter above), it follows that we should allow a new task to immediately preempt the current task. Indeed, setting the value of this parameter to zero achieved the best results across all thread configurations.

We will refer to the configuration of the CFS scheduler with these parameter values as the *enhanced* "server" mode.

C. Experimental Results

We now present the results of experiments we have conducted to compare the performance of the SPS under the baseline and enhanced "server" mode configuration for the scheduler. In the experiments, we vary both the number of SPS threads and the traffic load, expressed as number of calls per second (cps). Our goal is to identify the traffic load (in cps) at which the SIP control packet drop rate starts to exceed 1%, implying that end users may experience call establishment problems due to packet drops. Service providers may use this traffic load as a trigger to add more capacity to the system *before* significant losses of SIP control packets start to occur. Based on our survey of industry standards, the threshold for acceptable drop-rate in a VoIP environment is about 1% [16]. This value is considered in the industry as the threshold below which voice calls have quality comparable to toll quality. Note that this threshold is typically applied to voice packets to characterize the performance of the *data plane*. In this study, we use 1% as a reasonable value above which the drop rate will negatively impact the operation of the *control plane*; however, our methodology can be applied with any appropriate threshold value.

Tables I and II present the measured values of T_{sip}, K_{rcv}, and SIP packet drop rate under the baseline and enahanced "server" mode, respectively. Each column in these tables presents the average of thirty experiments for the stated number of threads and load (cps). For each value of the number of threads, we present results for two load values: the load value at which the drop rate exceeds 1% for the first time, and the immediately lower value (in our experiments, we used an increment of 200 cps for load values). We make two observations from these tables. For the same load value, configuring the scheduler parameters to the enhanced "server" mode, always improves the SPS performance in terms of drop rate and waiting time (which is included in the kernel time, K_{rcv}). Furthermore, in some cases (i.e., for two and 16 threads), the load at which the drop rate crosses the 1% threshold we imposed is higher for the enhanced mode than the baseline mode.

To better illustrate the performance improvement under the enhanced mode, Table III compares the drop rate and K_{rcv} values for specific number of threads and load pairs; these pairs were selected as they correspond to the scenarios where the drop rate under the baseline mode exceeds 1%. As we can see, the performance improvement is between 26-40% for the drop rate and between 7-34% for K_{rcv}. In fact, the enhanced mode results in better performance in all the scenarios, and the degree of improvement increases with the number of threads. In other words, by adjusting the scheduler parameters to align with the packet processing time at various degrees of multi-threading, the enhanced "server" mode is capable of reducing the context switching overhead associated with larger thread numbers. We emphasize that *the benefits of the enhanced mode are available for free* (other than the one-time cost of carrying out the off-line experiments), in that they are achievable simply by setting the scheduler parameter to appropriate values and do not involve any resource tradeoffs. Finally, we note that our methodology for optimizing the scheduler mode, although carried out in the context of SPS, is independent of the application layer, and hence it may be applied to a spectrum of servers.

V. Concluding Remarks

We have investigated the impact of the Linux scheduler settings on the performance of single-core, multi-threaded SIP proxy servers, in terms of packet service time, waiting time, and drop rate. Based on the results of a large set of experiments across a wide range of values for the number of server threads and traffic load, we have developed a methodology to configure the scheduler parameters that results in significant gains in SPS performance compared to industry-recommended "server" mode operation. Our methodology is not limited to SPS and may be applied to any application-layer server. Importantly, the gains in performance are the result of simply setting the scheduler parameters to appropriate values, without the need for adding server capacity or other capital expenditures. In our future work, we plan to investigate the performance of multi-threaded SPS on multiple CPU cores.

References

[1] Ramesh Krishnamurthy and George N. Rouskas. Evaluation of sip proxy server performance: Packet-level measurements and queuing model. In *Proceedings of IEEE ICC*, pages 2330–2336, June 2013.

[2] H. Jamjoom, C.T. Chou, and K.G. Shin. Impact of concurrency gains on the analysis and control of multi-threaded internet services. In *Proceedings of IEEE INFOCOM*, pages 827–837, March 2004.

[3] V Beltran, J Torres, and E Ayguade. Understanding tuning complexity in multithreaded and hybrid web servers. In *Proceedings of IEEE Parallel and Distributed Processing*, pages 1–12, April. 2008.

[4] Ben Lee *et al.* Hantak Kwak. Effects of multithreading on cache performance. *IEEE Transactions on Computer*, 48(2):176–184, Feb. 1999.

[5] Jonghun Yoo Sungju Huh and SeongSoo Hong. Improving interactivity via vt-cfs and framework-assisted task characterization for linux/android smartphones. In *Proceedings of IEEE International Conference on Embedded and Real-Time Computing Systems*, 2012.

[6] S.Zeaddally K.Salah, A.Manea and Jose M.Alcaraz Calero. On liux starvation of cpu-bound processes in the presense of network i/o. *Computer and Electrical Engineering*, 30, 2011.

[7] Ajoy K. Datta and Rajesh Patel. Cpu scheduling for power/energy management on multicore processors using cache misses and context switch data. *IEEE Transactions on Parallel and Distributed Sys*, pages 1190–1199, May 2013.

[8] S.V. Subramanian and R. Dutta. Comparative study of M/M/1 and M/D/1 models of a SIP proxy server. In *Proceedings of the Australasian Telecommunications Networking and Application Conference (ATNAC)*, pages 397–402, December 2008.

[9] S.V. Subramanian and R. Dutta. Measurements and analysis of M/M/1 and M/M/c queueing models of the SIP proxy server. In *Proceedings of the 18th International Conference on Computer Communications and Networks (ICCCN 2009)*, pages 1–7, August 2009.

[10] S.V. Subramanian and R. Dutta. Performance and scalability of M/M/c based queueing model of the SIP proxy server - a practical approach. In *Proceedings of the Australasian Telecommunications Networking and Application Conference (ATNAC)*, pages 1–6, December 2009.

[11] J. Rosenberg *et al.* SIP: Session Initiation Protocol. RFC 3261, June 2002.

[12] A. Johnston. *SIP: Understanding Session Initiation Protocol*. Artech House Publishers, 2nd Edition, 2004.

[13] OpenSIPS: Open source implementation of a SIP server. http://www.opensips.org/.

[14] Linux CFS Proces Scheduler. https://www.kernel.org/doc/Documentation/scheduler/sched-design-CFS.txt%.

[15] CFS Tuning by IBM. http://tinyurl.com/CFSTuning.

[16] Packet Drop Rate. http://www.telecompute.com/voip.asp/.

TABLE I
MEASURED SPS PERFORMANCE, BASELINE SERVER MODE

Model Parameters	Number of Server Threads									
	2 Threads		**4 Threads**		**6 Threads**		**8 Threads**		**16 Threads**	
	2200cps	2400cps	2000cps	2200cps	1800cps	2000cps	1600cps	1800cps	1200cps	1400cps
Arrival rate (packets/sec)	13200	14400	12000	13200	10800	12000	9600	10800	7200	8400
T_{sip} (μs)	76.9	79.58	130.81	146.25	167.97	203.9	213.47	279.36	303.00	429.83
K_{rcv} (μs)	1309.16	1752.26	1473.32	2184.27	1042.66	1981.73	893.26	1974.84	809.76	1394.47
RCV Errors	3926	7993	4103	8061	3163	7830	2166	8100	2613	5114
Total Messages	633487	598877	494568	513137	441154	490867	417454	447876	354911	372133
Drop rate	0.0062	**0.0133**	0.0083	**0.0157**	0.0072	**0.0159**	0.0052	**0.018**	0.0074	**0.0137**

TABLE II
MEASURED SPS PERFORMANCE, ENHANCED SERVER MODE

Model Parameters	Number of Server Threads									
	2 Threads		**4 Threads**		**6 Threads**		**8 Threads**		**16 Threads**	
	2400cps	2600cps	2000cps	2200cps	1800cps	2000cps	1600cps	1800cps	1400cps	1600cps
Arrival rate (packets/sec)	14400	15600	12000	13200	10800	12000	9600	10800	8400	9600
T_{sip} (μs)	77.21	85.11	129.28	138.76	178.11	186.85	189.37	242.29	392.25	540.13
K_{rcv} (μs)	1624.2	2322.44	1363.77	1805.35	1116.93	1418.35	594.44	1321.15	918.58	1864.63
RCV Errors	6283	10908	4559	6028	3312	5103	1698	4996	2987	8470
Total Messages	636057	642238	504670	525251	445583	461109	433703	440549	367064	363775
Drop rate	0.0098	**0.0169**	0.0090	**0.0115**	0.0074	**0.0110**	0.0039	**0.0113**	0.0081	**0.0233**

TABLE III
DROP RATE AND KERNEL TIME (K_{rcv}) COMPARISON

Server Threads and CPS	Drop Rate comparison			Kernel time, K_{rcv}, comparison (μs)		
	Baseline Mode	Enhanced Mode	$\%Lower$	Baseline Mode	Enhanced Mode	$\%Lower$
2 Threads, 2400cps	0.0133	0.0098	26.31%	1752.26	1624.2	7.3%
4 Threads, 2200cps	0.0157	0.0115	26.4 %	2184.27	1805.35	17.3 %
6 Threads, 2000cps	0.0159	0.0110	30.8%	1981.73	1418.35	28.4%
8 Threads, 1800cps	0.018	0.0113	37.22%	1974.84	1321.15	33.08%
16 Threads, 1400cps	0.0137	0.0081	40.87 %	1394.47	918.58	34.1%