Parallel Packet Processing on Multi-core and Many-core Processors

Andy Harvath, Hiroaki Nishi
Graduate School of Science and Technology, Keio University, Japan
harvath@west.sd.keio.ac.jp, west@sd.keio.ac.jp

Abstract—The Service-oriented Router (SoR), a highly functional router based on a novel router architecture, enables unprecedented web services traditional routers were unable to provide. The SoR performs Deep Packet Inspection (DPI) to analyze Layer 7 information, which is becoming increasingly difficult due to the substantial increase in Internet traffic. Meanwhile, multi-core processors and general-purpose many-core processors are increasing in popularity. These highly programmable many-core processors are suited for parallel packet processing and propose a parallelization method for packet inspection. The method is applied to NEGI, a software SoR simulator. The parallelized software is then implemented on a multi-core Xeon CPU to test for effectiveness and scalability. While the results confirm its scalability, we find that the throughput of the output process must be improved for SoRs to benefit from the proposed method.

Keywords—Service-oriented router; multi-core; many-core; packet processing

I. INTRODUCTION

With the growing ubiquity of smartphones and other communication devices in the past decade, the Internet has become increasingly accessible. As a result, contents transferred over the Internet are expanding in both amount and diversity. For the Internet to provide high-quality services under these circumstances, it must not only transfer data from one end host to another at high speed, but also provide the user with valuable information efficiently.

Examples of an approach to improve delivery efficiency include Content-Centric Networking (CCN) [1]. CCN was introduced as an alternative networking paradigm based on named data rather than named hosts and is believed to enable efficient content distribution. The Service-oriented Router (SoR) [2] is a router architecture that takes a similar information-centric approach. The SoR is capable of observing traffic data stream, inspecting packet payloads, and storing data in databases. Content-based routing can be performed by an SoR router using the extracted data, sending information to where it is demanded. SoR also enables other unprecedented web services that traditional routers were unable to provide, such as a router-based Network Intrusion Detection System (NIDS).

One of the existing challenges to the SoR is to achieve wire-rate throughput. The rapidly growing Internet traffic demands high transfer speed for backbone routers, which means that SoRs must extract data from packets at an equally high processing speed.

Meanwhile, parallelization has become the most common approach for speedup, both at the hardware level and at the software level. We have entered the “era of higher processor parallelism” [3], in which superior microprocessor performance is gained from high parallelism and not from high clock speed. A majority of processing units that are currently in use implement multi-core architectures. In addition to these multi-core processors, many-core processors have been introduced for applications that make use of even higher parallelism. Many-core generally refers to computing devices that have exceptionally large numbers of processors on a single chip. Examples of many-core processors include Graphic Processing Units (GPU) and Intel’s Many Integrated Core (MIC) processors.

Given these backgrounds, parallelizing the SoR’s packet inspecting process is a natural approach to solving the throughput problem. Because a large number of streams flow through a router simultaneously, there should be sufficient concurrency to utilize the highly parallel hardware. Moreover, each TCP stream is entirely independent, which means that all can be processed concurrently without the need to lock resources.

Because one of the main ideas behind the SoR is to make it flexible and programmable, a hardware-based approach is not preferable. One of the most popular forms of many-core processors are GPUs, and a section of a software can be run on a GPU to increase its throughput. However, GPUs are optimized for floating-point Single Instruction Multiple Data (SIMD) instructions. The threads in a GPU are grouped into fixed sized batches called warps, and all threads in a warp has to execute the same instruction. This makes conditional branches in parallel codes very slow. While GPUs lack the the flexibility for complicated packet processing functions, the MIC architecture consists of 50 to 60 simple in-order x86 cores connected to a bidirectional ring bus and is known to be highly programmable. In addition, because MIC processors support the x86 instruction set, codes that are optimized for any x86 CPU will also run on and MIC chip. Although the theoretical speed for current implementations of MIC is known to be extremely difficult to achieve [4], we believe that the high parallelism and programmability of MICs are suited for parallel packet processing.

This study focuses on the parallelization of NEGI [5], a software implementation of SoR written in C/C++. We
propose a parallel method to reconstruct TCP streams, extract content information, and output results to a database from Internet packets. The proposed method was implemented on a 12-core, 48 thread Intel Xeon Processor to test scalability and prepare for its future implementation on Intel MIC processors.

The rest of this paper is structured as follows. In section II, several works related to parallel packet processing and many-core parallelization are introduced. Section III describes the NEGI application in detail. The parallelization method is proposed in Section IV, and its test results are presented in Section V. Finally, we conclude the paper in Section VI.

II. RELATED WORKS

Parallel packet processing has drawn a lot of interests recently. As more and more network processors implement multi-core architectures, the demand for parallel applications that exploit these architectures is growing. Vert Paxon et al. introduced an event-based framework for parallelizing Network Intrusion Prevention Systems (NIPS) on multi-core processors [6]. Yunchun Li et al. proposed a packet processing model and calculated the theoretical speedup of parallelization of DPI systems [7]. Most of the research in parallel packet processing target network processors, and there are no studies on the use of MIC processors for packet processing.

III. SoR SIMULATOR: NEGI

NEGI is a Layer 7 information extractor developed for simulating and evaluating SoR. NEGI is written in the C/C++ programming language, and uses the libpcap library to process packets from either a Linux Ethernet device or a pcap file. The current version of NEGI loads a user-defined filter and applies it to the incoming packets before saving the resulting data in a SQLite database file, along with basic information such as source/destination IP addresses, source/destination port numbers, and the protocol number. Below is a more detailed description of how NEGI operates.

Figure 1 shows the architecture of the NEGI application, which can be divided into several function blocks that interact with each other. These modules include:

- Packet capture engine
- TCP reconstructor
- Layer 7 decoder
- String matching engine
- Database insertion engine
- TCP timeout manager

A. Packet capture engine

As noted, NEGI makes use of the libpcap library to monitor a Linux Ethernet device. The packets are written in NEGI’s shared memory, and its pointers are added to a message queue. Since certain packets can be instantly discarded, the processing time for the packet capture engine is not uniform. The packet engine, therefore, works independently from the succeeding modules. The interface between the packet capture engine and the TCP reconstructor engine is provided by a message queue. By splitting the capture engine and the processing engines, NEGI conceals the unevenness in processing each packet.

B. TCP reconstructor

Figure 2 is a diagram that shows how the TCP reconstructor processes each packet using a context switch. A, B and C in the figure refers to different TCP packet streams. In addition, the units labeled A1, B1, etc. each represents a single packet. For example, B2 is a packet with sequence number 2 that belongs to TCP stream B. Finally, the packets stored in Stream Reconstruct Information are prefixed with asterisks to emphasize the fact that they are merely pointers that point to where the packets are stored.

In figure 2, packets A1, B1, A2, and C1 have already passed, and packet C2 is being processed. When a packet is sent to open a TCP connection, the TCP reconstructor acknowledges it, creates a new TCP stream information frame, and passes the information to the subsequent modules. After all the main modules process the packet, the processing states of each module is saved as Stream Reconstruct Information. This information is recalled when another packet from the same stream is loaded. In figure 2, C1 has created Relevant Information C and Intermediate State C, which are loaded by the TCP reconstructor to process C2. When the TCP reconstructor detects an end of a stream, the stream results are finalized and saved to the database. The stream information is then freed from memory.

Fig. 1. The NEGI architecture

Fig. 2. TCP reconstruction in NEGI
The TCP reconstructor assigns packets to different streams by analyzing the TCP header. Namely, the source and destination IP addresses, source and destination Port Numbers, and the Protocol numbers are used to distinguish TCP connections. Stream information is created when the TCP reconstructor detects a new SYN packet, and destroyed when a FIN or a RST packet is captured.

C. Layer 7 decoder

The Layer 7 decoder uses the stream context information to decode various application protocols to enable string matching and other processes. The targets include HTTP/1.1’s chunk encode and gzip encode. When the decoder reaches the end of a packet, intermediate states are saved in a format specific to the protocol type.

D. String matching engine

The string matching engine loads user-defined matching rules from a database and applies them to incoming packets. When matching strings or patterns are detected in the packet content, they are passed to the database insertion engine. When there is a matching pattern between multiple packets, the string matching engine will reach the end of a packet while it is matching a string. In such cases, the state is saved as stream context and recalled when a succeeding packet arrives. The version of NEGI used in this study does not support advanced regular expression matching.

E. Database insertion engine

If any defined strings or patterns are found in a TCP stream, the database insertion engine stores the following information in a database: an ID to identify the stream, timestamp of its arrival, destination IP address, source IP address, destination port number, header information, and an ID to label which rule was applied. If any strings were extracted, the database insertion engine also saves the strings to the database with corresponding TCP stream IDs. As previously noted, the version of NEGI that was used in the study uses a SQLite3 database, which comes in the form of a single file, to store the results.

F. TCP timeout manager

When dealing with real internet traffic, it is not guaranteed that all TCP connections close normally. If for some reason the closing packets are not detected, the context information of the stream could be stored in memory as long as the NEGI process is up and running. To avoid memory leaks in this situation, the TCP timeout manager monitors each stream. The TCP manager has two main functions. First, if no packets are received from a stream for duration, the TCP manager deems it as closed and frees all relevant information. Another function for the TCP timeout manager is to destroy stream data depending on available memory. If a situation of memory overuse is detected, the TCP timeout manager deletes the least active streams to fit the memory requirements. The timeout duration and the maximum available memory are user-defined and written in a configuration file.

IV. PARALLELIZATION METHOD

To keep pace with the substantial increase in internet traffic, we proposed a method to parallelize the process of NEGI and increase its throughput. The parallel version of NEGI (pNEGI) is written in the C language and is designed to be run on an x86-based multicore system and ultimately on an MIC processor. pNEGI uses the POSIX threads library to create threads and utilize the multiple cores provided by the hardware.

A. Limitations

The software architecture of NEGI is as previously shown in Figure 1. Out of the modules illustrated in this diagram, the packet capture engine cannot be parallelized at the software level. This is due to the libpcap library’s single thread nature; the packets are inputted serially. Since NEGI operates on the SQLite3 library, the output is a single file, and hence, also serial. A parallel write to one file generally does not give permission because the file must be locked frequently. Therefore, NEGI can be assumed a single input, single output model.

On the other hand, multiple instances of other modules that operate in the main thread, namely the TCP reconstructor, the L7 decoder, and the string match engine can be created and run simultaneously. This allows multiple threads to handle different streams at the same time. However, it must be noted that packets have sequence numbers, and ones that belong to the same stream must be processed in order. In addition, for there to be no dependency between processing threads, the same stream has to be processed in the same thread.

B. Architecture

Figure 3 shows the architectural structure of pNEGI. Parallelizing a single-input, single-output software model like one of NEGI requires some kind of a fork-join structure. As shown in figure 3, pNEGI implements this by placing ring
buffer queues at the diverging and converging points. These queues not only allow for single-to-multiple and/or multiple-to-single message passing between threads, but also let the threads work asynchronously. This is especially important for this architecture, because neither the number of packets assigned to each core nor the processing time for each packet is known.

C. Behavior

When pNEGI is started, the main thread performs the initializing tasks, which include creating new processing threads and various module instances. The number of threads is user-defined; it will be read from the configuration file. The main thread then opens either an Ethernet device or a pcap file and begins reading.

When the main thread captures a packet, the header and content data is distributed to the processing threads. Specifically, each thread owns a packet queue, and the main thread pushes the packets to them. As previously mentioned, the main thread cannot simply distribute the data evenly in a round robin fashion. Packets belonging to a certain stream depend on each other and have to be processed in a single thread. Both to assign each stream to the same thread and to balance the load, pNEGI uses a hashing technique to determine the receiving thread.

Cyclic Redundancy Check (CRC) is a method to detect accidental changes to data. CRC is mainly used in networking to check for unintended bit errors, but can also be used as a hash function in a non-security context. pNEGI inputs a concatenation of source IP address, destination IP address, source port number, and destination port number to a CRC function that outputs an 8-bit hash value. This hash value is divided by the number of threads and the remainder is used to determine the thread to assign packet data. CRC was implemented because of its simplicity and efficiency, and the output was set to 8bits (0-255) to fit the maximum number of threads on an MIC processor, which is currently 244.

Each processing thread receives a signal when a new packet is added to its queue. The TCP reconstructor responds to the signal by dequeuing the packet and processing it. The packet data is then transferred to the L7 decoder and the String match engine in the same manner as the single-thread NEGI. When a packet is done processing, its intermediate status is saved to the thread’s unique stream information pool. Each processing thread has its own memory area to store information for stream reconstruction for increased independency among threads. For the same reason, each processing thread also has its own TCP timeout manager that monitors each stream.

If a processing thread has valuable data or information that has to be stored in the database, it generates a SQL command string to insert the data. The SQL string is enqueued to the SQL queue and eventually passed to the database insertion engine, which executes the given command to the SQLite3 database. The database insertion engine itself runs in an independent thread, which we call the SQL thread, to prevent multiple threads trying to open the database file, and to take away computation from the processing threads.

V. EXPERIMENTAL RESULTS

Both the NEGI and its parallelized version, pNEGI, were implemented on a 12-core Intel Xeon CPU to evaluate the proposed method. A pcap file was inputted to each software and the processing times were measured to calculate throughput. pNEGI was also tested for different numbers of threads.

A. Experimental environment

All experiments were performed on an Intel Xeon E5-2697 CPU. Table 1 lists the basic specifications for the processor. As shown in table 1, the Xeon processor used in the study has 12 cores with 2 implemented hardware threads on each core. With Intel’s Hyper-Threading Technology (HTT) [9], these cores can execute up to 4 threads concurrently with comparatively lower performance.

A 1.5MB pcap file was used as a controlled input. This file was dumped from our laboratory’s gateway server in a five-minute interval, and included totals of 1,391,020 packets and 47,971 TCP streams.

B. Results: Database insertion engine enabled

The results are shown in Figure 5 and Table 2. As it can be seen from both figures, pNEGI recorded a throughput approximately 3Mbps higher than that of NEGI. However, it can easily be deduced that the speedup is not a result of the parallelization of packet processes. pNEGI exhibited the best performance with only 1 processing thread; distributing packets among multiple processing threads did not increase the performance, but rather reduced it.

| TABLE 1. INTEL XEON E5-2697 PROCESSOR SPECIFICATIONS [8] |
| --- | --- | --- |
| # of Cores | 12 |
| # of Threads | 24(48) |
| Instruction Set | 64bit |
| Processor Base Frequency | 2.7GHz |
| Max Memory Size | 768GB |
| Max Memory Bandwidth | 59.7GB/s |

![Fig. 4. Speedup from extraction of insertion process](image-url)
It is easy to conclude that the 3Mbps speedup was due to the implementation of the SQL thread. Extracting the insertion process from the packet inspection process allows for a faster processing of packets, as shown in figure 4. The decrease in throughput at larger numbers of threads can be attributed to the resource costs of thread creation, and possibly to the more frequent locking and unlocking of the semaphore of the SQL queue to serialize its accesses. Finally, it can be presumed from the flat graph that the database insertion process was a bottleneck; parallelizing the packet inspection process had nearly no effect to the overall throughput because the database insertion engine could not process the SQL messages as quickly as the processing threads processed the packets.

C. Results: Database insertion engine disabled

Another experiment was performed to test the hypothesis that the output was a bottleneck. In this experiment, the database insertion engine received the SQL command strings, but did not execute the commands.

An overall increase in throughput can be observed from the results in figure 6 and table 3. As opposed to the previous results, increase in the number of threads generally resulted in high throughput. pNEGI scaled in what can be seen as a linear rate up to about 20 processing threads. This is because the Xeon processor has 24 hardware threads. For a bigger number of threads, the processor made use of Intel’s HTT technology to execute them using limited resource. Finally, at about 46 processing threads (48 total running threads), the processor had simply reached the maximum number of threads that it could run, and the performance plummeted. The results clearly supported the hypothesis that database insertion was NEGI’s bottleneck.

VI. Conclusion

In this study, we have proposed a parallelization model for SoR’s packet inspection process and applied it to the software SoR simulator, NEGI. The parallelized software was implemented on an Intel Xeon CPU, where its scalability was confirmed. We also found that for the SoR to benefit from this method, a critical bottleneck has to be removed: it will have to speed up the database insertion process if any data has to be saved. Possible methods of this include the parallelization of the database insertion process, the use of on-memory databases, the use of no DBMSs at all, or a combination of these methods.

The results suggest that pNEGI will also scale on an MIC processor. The fact that parallelization was effective for processing a dump file from a low-scale private network indicates that it will be equally, if not more effective in real-world situations, where greater concurrency is expected. The application will definitely not exhibit the same rate of increase in performance because although MIC processors have a much larger number of cores, each core is substantially slower than today’s CPU cores, especially for integer operations. Despite these limitations, we believe that the MIC architecture is suited for parallel packet processing.

Our ultimate goal is to implement pNEGI on an MIC processor and evaluate the effect of highly parallel hardware on parallel packet processing. The results of this study are essential because it ensures good scalability on the Xeon CPU and hence suggests scalability on MIC processors as well.
VII. Acknowledgement

This work was partially supported by the funds of SECOM Science and Technology Foundation, and by MEXT/JSPS KAKENHI Grant (B) Number 24360230 and 25280033.

VIII. References


