# Removing System Bottlenecks in Multi-threaded Applications

## Application Note

*September 2008*

Order Number: 320631-001US

## Contents

## Figures

## Tables

# Revision History

| Date | Revision | Description |
|---|---|---|
| September 2008 | 001 | Initial release |

## 1.0 Introduction

### 1.1 Purpose

This document describes the software optimization of a popular open-source embedded application. The goal was to document the efforts/procedure to multi-thread an existing single-threaded application.

### 1.2 References

The following list of documents constitutes the baseline and framework for this project:

1. Snort diagrams for developers http://afrodita.unicauca.edu.co/~cbedon/snort/snortdevdiagrams.pdf
2. Supra-linear Packet Processing Performance with Intel® Multi-core Processors - White Paper doc # 31156601

### 1.3 Acronyms

Table 1. Acronyms

| Acronym | Definition |
|---|---|
| SMP | Symmetric Multi-Processing |
| OS | Operating System |

### 1.4 Definitions

Table 2. Definitions

| Term | Definition |
|---|---|
| Flow-pinning | Packet segregation method in which all packets associated with same TCP flow are processed by same CPU core |
| Dual-core / dual-processor | A processor combining two independent cores into a single package |
| mpstat | A Linux-based tool that reports processor-related statistics |

## 2.0 Description of the project

### 2.1 Background and purpose

SNORT* is an open source network intrusion prevention and detection system utilizing a rule-driven language, which combines the benefits of signature, protocol and anomaly based inspection methods. With millions of downloads to date, SNORT* is the most widely deployed intrusion detection and prevention technology worldwide and has become the de facto standard for the industry.

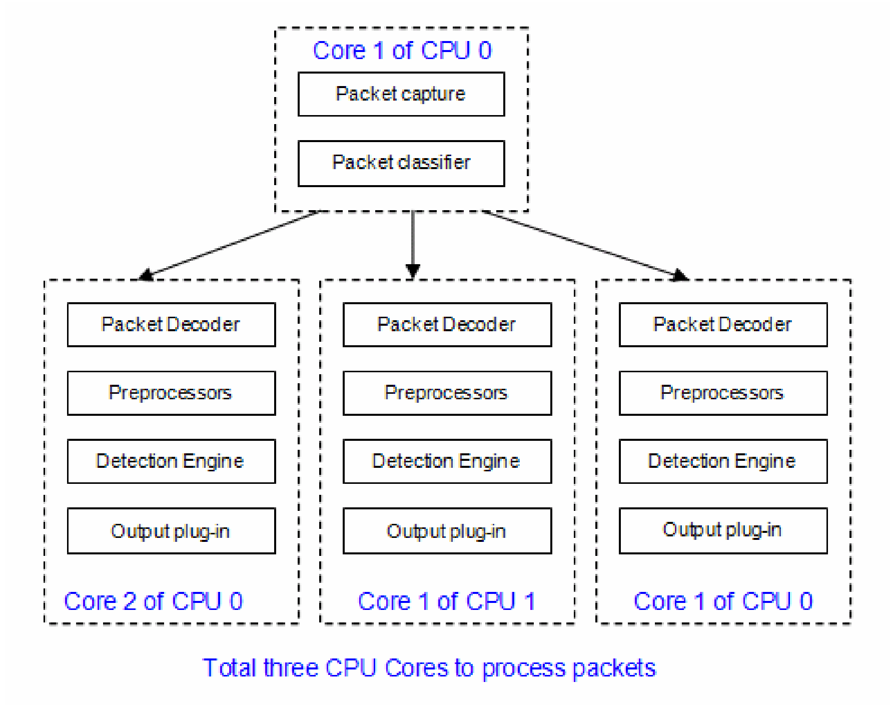
Multi-core has become a clear trend in the computing area to the extent that multi-core systems are available at an ever-expanding range of markets. While there are many ways to take advantage of the computing power of multi-core systems (ex. virtualization, SMP OS, etc.), multi-threading individual applications may be the best way to realize large performance gains on such platforms.

This document demonstrates one method to rewrite a serial piece of software - SNORT v2.8.0.1 - to take advantage of a multi-core system. Procedures for benchmarking, application analysis, and code modification that can be applied to many other multi-core development scenarios are also included.

## 2.2 Snort-2.2.0 optimization

In 2005 Intel conducted a study of SNORT version 2.2.0; the effort had a similar goal: to multi-thread a serial application for optimal performance on a multi-core platform. Several iterations of analysis and code rewrite converged on a solution featuring a two-stage pipelining and flow-pinning software architecture. Figure 1 shows this solution on an Intel dual-processor, dual-core platform.

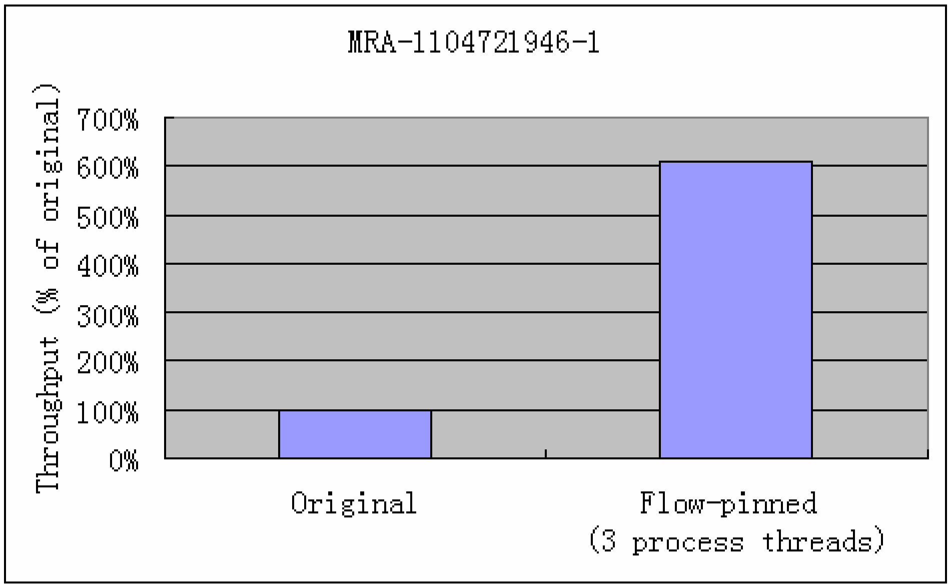
**Figure 1. Architecture of two-stage pipeline and flow-pinning optimization on Dual processor Dual core system**



Here, SNORT's packet classifier module runs on Core 1 of Processor 0 and directs network traffic to all the other cores based on a hash of fields in the packet header, specifically the IP source address and destination address fields, as well as the TCP source port and destination port fields. This hash algorithm ensures that packets from the same TCP flow are always assigned to the same core; this type of algorithm is known as flow-pinning.

Refer to Document [2] for the full details. The outcome of the first study was a performance boost of about six times the original serial version for test cases with large numbers of TCP connections (~25,000 in MRA-1104721946-1), as shown in Figure 2.

**Figure 2. Offline trace file test result**



Since that time, SNORT was updated rapidly by the open-source community and serial performance improved prominently. Benchmarking shows that the current "out of the box" release of SNORT version 2.8.0.1 is 10 times faster than the baseline version of Snort v2.2.0. In addition, this latest serial version of SNORT is 3 times faster than the parallel version created in Intel's first study (that is, Snort-2.2.0-parallel) in offline HTTP-32KB data trace file[1] test.

It is paramount to start with the most optimized serial version possible when approaching the task of multi-threading a given application. It is much easier to develop/debug a single-threaded application than a multi-threaded one.

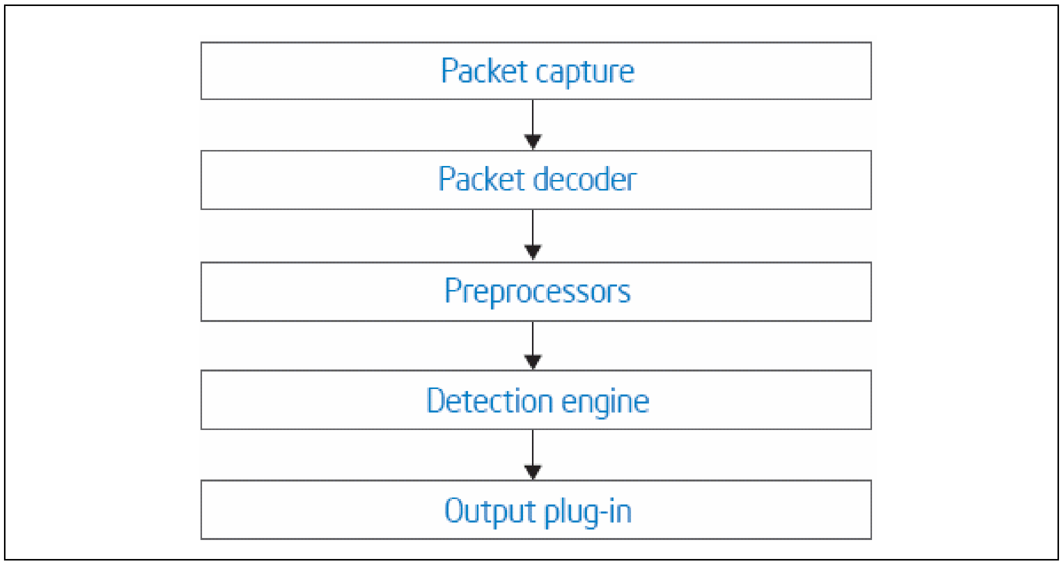
With this background in mind, this document uses Snort v2.8.0.1 as its baseline for analysis.

## 3.0 Snort v2.8.0.1 optimization

The overall SNORT software architecture has not changed much from Snort v2.2.0 to Snort v2.8.0.1. The dataflow can be subdivided into five functional processes, as shown in Figure 3.

1. Dumped the packets generated by Spirent Avalanche 2700C. HTTP 32KB data file and TCP connection speed 2,000 per second

Figure 3. Snort v2.8.0.1 Dataflow



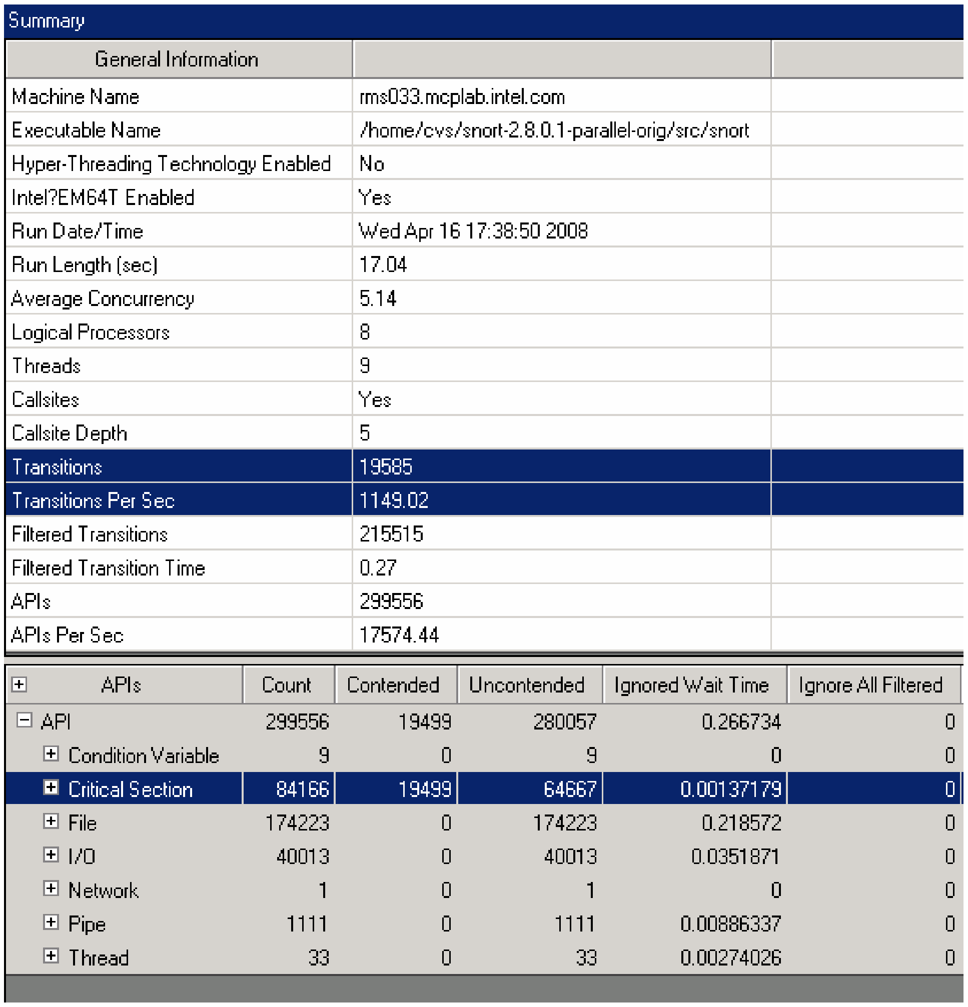
Given the structural similarity between SNORT versions 2.2.0 and 2.8.0.1, it is feasible to employ many of the same optimizations in this project (i.e. of Snort v2.8.0.1). Snort v2.8.0.1 was modified to implement a two-stage pipeline and flow-pinning software architecture.

However, performance did not improve; in fact, this new version was slower than original serial version. For example, in an offline HTTP 1KB data trace file test, the multi-threaded snort-2.8.0.1 was approximately 35% slower than the serial version (22.68 seconds versus 16.80 seconds).

## 3.1 Using the Intel® Thread Profiler

The team used the Intel® Thread Profiler plug-in for the VTune™ Performance Analyzer to determine where the biggest performance bottlenecks were. The tool indicated - via the Profiling summary window - that the multi-threaded version of SNORT spent a lot of time context swapping between threads, most likely due to high levels of contention to enter critical sections (that is, the threads are fighting over shared resources protected by synchronization locks). Figure 4 shows these results.

**Figure 4. Snort v2.8.0.1-Parallel-orig TProfile summary**



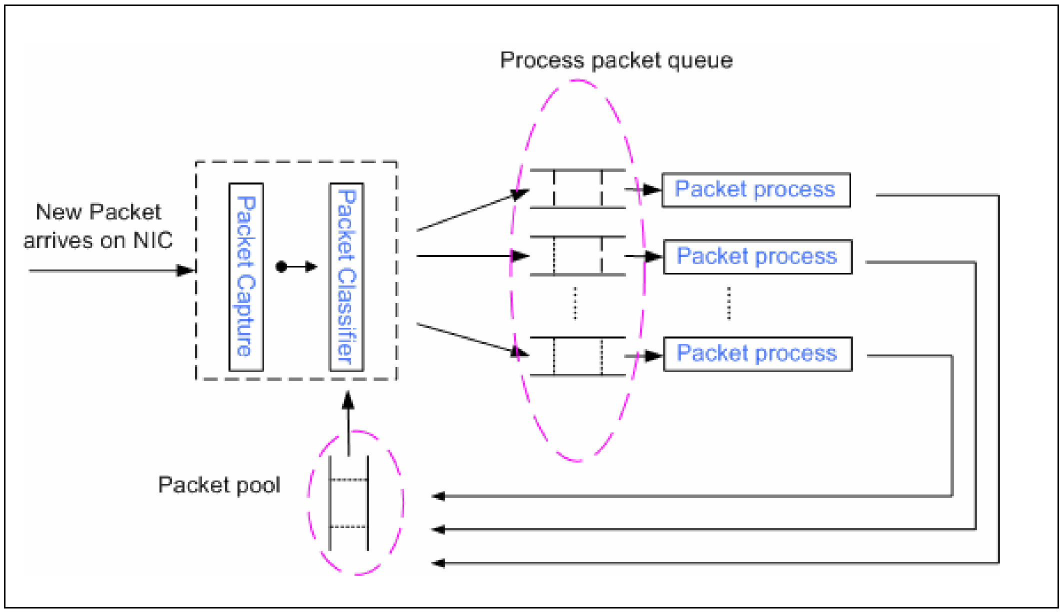
Summary

| General Information | | |
|---|---|---|
| Machine Name | rms033.mcplab.intel.com | |
| Executable Name | /home/cvs/snort-2.8.0.1-parallel-orig/src/snort | |
| Hyper-Threading Technology Enabled | No | |
| Intel?EM64T Enabled | Yes | |
| Run Date/Time | Wed Apr 16 17:38:50 2008 | |
| Run Length (sec) | 17.04 | |
| Average Concurrency | 5.14 | |
| Logical Processors | 8 | |
| Threads | 9 | |
| Callsites | Yes | |
| Callsite Depth | 5 | |
| Transitions | 19585 | |
| Transitions Per Sec | 1149.02 | |
| Filtered Transitions | 215515 | |
| Filtered Transition Time | 0.27 | |
| APIs | 299556 | |
| APIs Per Sec | 17574.44 | |

| APIs | Count | Contended | Uncontended | Ignored Wait Time | Ignore All Filtered |
|---|---|---|---|---|---|
| API | 299556 | 19499 | 280057 | 0.266734 | 0 |
| Condition Variable | 9 | 0 | 9 | 0 | 0 |
| Critical Section | 84166 | 19499 | 64667 | 0.00137179 | 0 |
| File | 174223 | 0 | 174223 | 0.218572 | 0 |
| I/O | 40013 | 0 | 40013 | 0.0351871 | 0 |
| Network | 1 | 0 | 1 | 0 | 0 |
| Pipe | 1111 | 0 | 1111 | 0.00886337 | 0 |
| Thread | 33 | 0 | 33 | 0.00274026 | 0 |

Overall, out of a total of 84,166 synchronization lock attempts, 19,499 were contended, which means that approximately 1 out of 4 times a thread ready to work (i.e. enter a given critical section) was not able to do so because another thread was already there and had locked the resource. Overall, this application performed context switches 1149.02 times per second, mostly because of this 23% lock contention rate, and this became one of the greatest factors that led to poor performance.

Figure 5. Snort v2.8.0.1-Parallel -orig Architecture



Armed with this data from the Intel® Thread Profiler, the team analyzed the code itself and found that almost every thread in the system faced a synchronization lock in order to write itself into the packet pool, as shown in Figure 5. Given that this is a common operation, improving this area of code became the number one area for improvement.

## 4.0 Making improvements

Software “locking” mechanisms (ex. critical sections, mutex, etc.) are commonly used to regulate access to data structures shared between threads. However, contention for locked resources usually becomes one of the major bottlenecks to multi-threaded application performance.

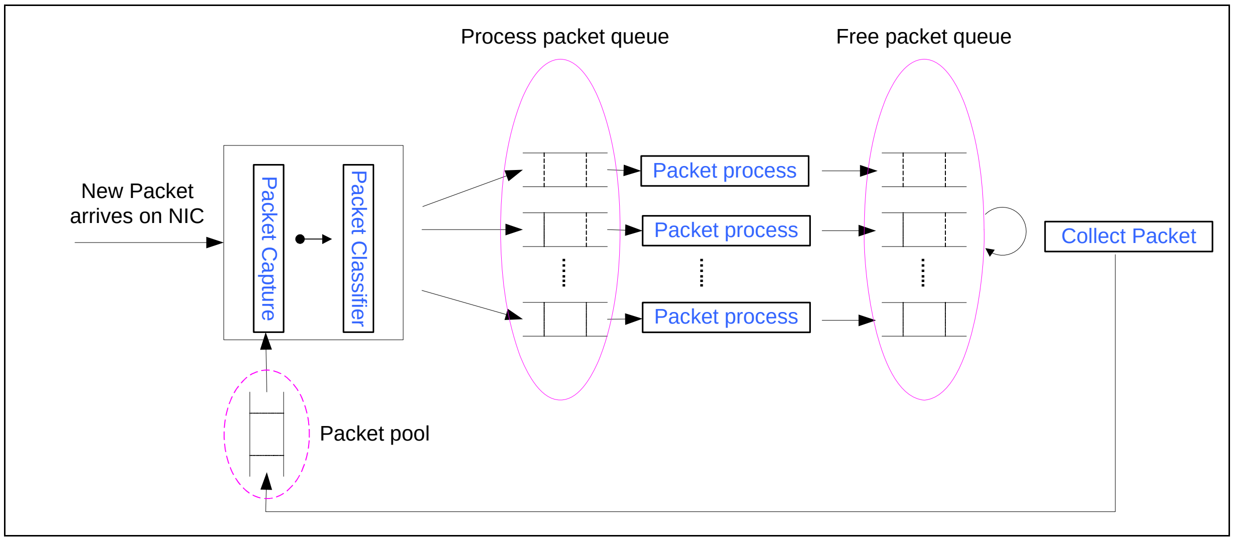
In fact, the multi-threaded version of SNORT employs locks to protect read-modify-write operations on the application's Packet Processing queues. Re-architecting the application to remove these locks is a priority for achieving higher packet throughput.

### 4.1 Removing Lock Contention

To eliminate the excessive contention rates around packet queue access in the packet receive/processing architecture, all queues were rewritten to be Lock-Free. A “Free Packet pool” data structure was created for each Packet process thread; and one independent thread, “Collect Packet”, was created to be the sole software entity to read from the “Free Packet pool” queues (round-robin), and write this data back into the “Packet pool” queue. While the algorithm requires slightly more data storage to implement this intermediate step, it allows most of the threads to work independently, which is a key characteristic of a good multi-threaded algorithm.

Figure 6 demonstrates the new Snort v2.8.0.1 parallel architecture.

**Figure 6. Snort v2.8.0.1 Parallel architecture (Lock free)**



A general rule of thumb in multi-threaded software is to share as little data as possible between threads because the synchronization overhead can really limit performance.

## 4.2 Removing Excess Memory Copies

In the process of inspecting the Snort v2.8.0.1 multi-threaded version of the source code (and performing the queue rewrites described in the previous section), the team found a duplicated memory copy for each packet going from the Packet Capture module to the Packet Classifier module.

When a new packet arrives on the Network Interface Card, the Packet Capture module copies it to a pre-allocated memory block (i.e. one set aside during module initialization). In the serial version of SNORT, a pointer to this memory block is passed from the Packet Capture module to the Packet Processing module, so that there is only one overall memory copy. On the other hand, the multi-threaded version of SNORT passes a pointer to the memory block to Packet Classifier module (i.e. a new module not in the serial version), which, in turn, copies the memory block itself in order to perform a node read from the Packet pool data structure.

This data copy is simply done in an effort to keep the modules as separate as possible, per the vision of the original SNORT architecture. So the Packet pool data structure and code that needs to read/write it was moved from the Packet Classifier module to the Packet Capture module, as implied by Figure 6.

In this updated implementation of the Packet Capture module, packet memory blocks are not allocated during the initialization phase. Instead, the Packet Capture module takes a memory block from the Packet pool only as each new packet arrives on the NIC.

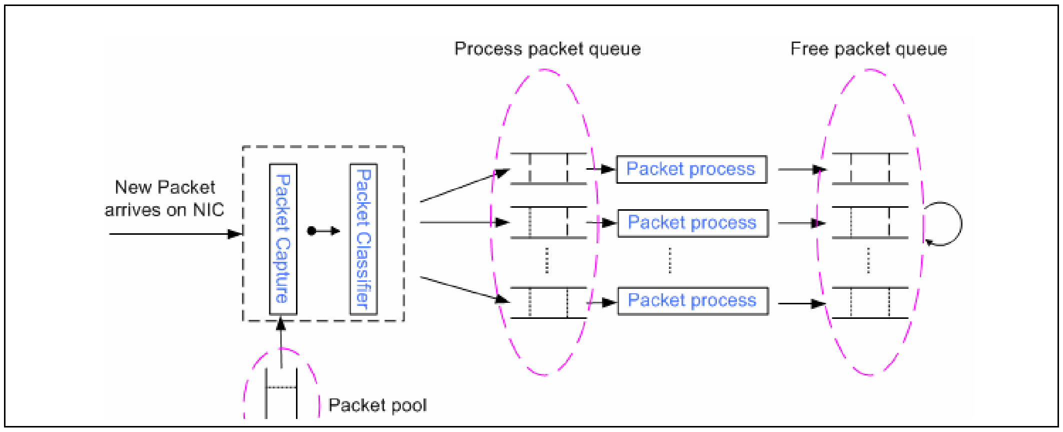
After packets are copied from the NIC to memory, the Packet Capture module passes a pointer to the Packet Classifier module, which, in turn, performs a hash on the packet header. As mentioned before, this hash is used to distribute packets to multiple Packet Processing modules running on different Cores. Here, again, the Packet Classifier passes just the packet pointer to the Packet Processing module in order to reduce the number of memory copies per packet.

## 5.0 Benchmark Test and Results

After making the changes described in the sections above, the team retested the multi-threaded version of SNORT v2.8.0.1; the system configuration and results follow.

Note: The full benchmark procedure can be found in Appendix A, "Performing Benchmarks".

**Figure 7. Snort v2.8.0.1-Parallel Architecture**



### 5.1 System Configuration

- CPU: Quad-Core Intel® Xeon® Processor X5355 (8M Cache, 2.66 GHz, 1333 MHz FSB)
- Chipset: Intel® 5000P Chipset with 6311ESB I/O Controller Hub
- MEM: 4GB FBDIMM DDR2 667MHz
- NIC: Built-in Dual Port GbE 631xESB/632xESB
- OS: RedHat AS4 Update 4 x86-64bit with Linux kernel 2.6.23
- PF-Ring: Trunk version 3427
- SNORT* Version: 2.8.0.1
- SNORT* Rules: snortrules-snapshot-2.8.tar.gz
- SNORT* Configuration: Default configure file.
- Packet Generator: Sprient Avalanche 2700C and 2700B
- Network Switch: Cisco* 3550-12T Gigabit switch

### 5.2 Results

The team performed two tests:

1. Packet throughput for an offline packet trace file
2. Packet throughput for a "live" packet trace

In the first "offline" packet trace file test, packets are read from trace file stored on the system's hard drive. The second test is more "true-to-life", where the packets are received off a network via the system's NIC.

Figure 8 shows that in the offline packet trace file test, the new multi-threaded version of SNORT* gained a 26% performance boost when using HTTP 1KB data packet trace file over the original serial version, and a 10% performance boost when using HTTP 32KB data packet trace file.

**Figure 8. Offline test result**

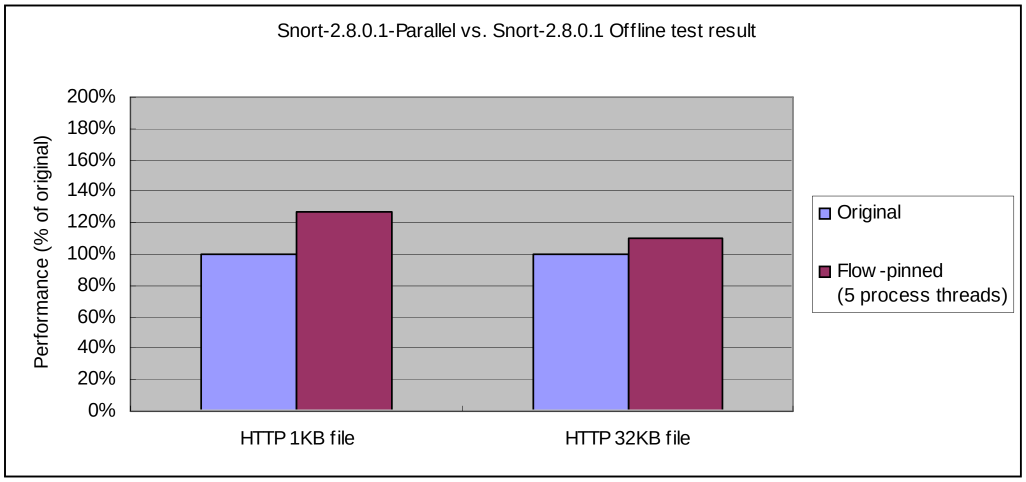
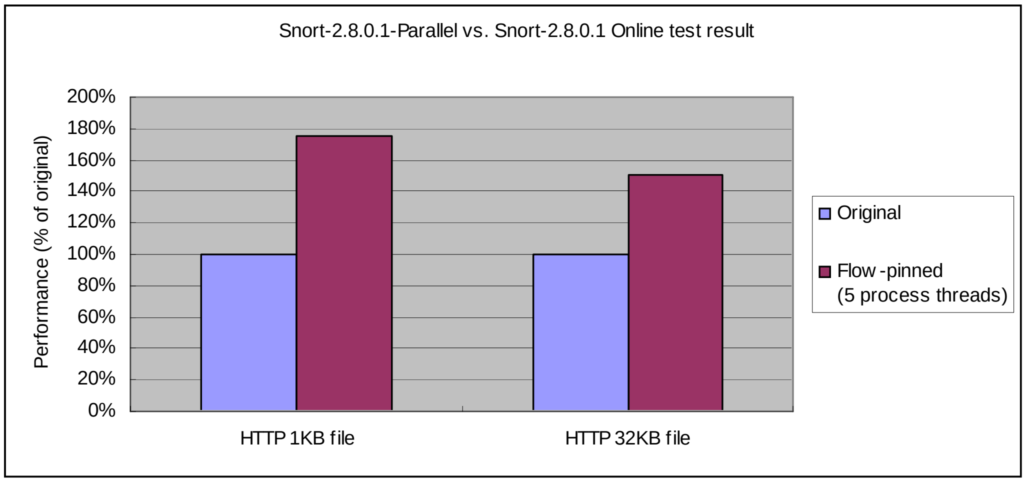


Figure 9 shows that in the “online” test, the multi-threaded version performance is 75% better than the serial version when using HTTP 1KB data file, and 50% better when using HTTP 32KB data file.

**Figure 9. Online test result**



Overall, the new multi-threaded code based on the SNORT v2.8.0.1 baseline can process between 10%-75% more packets per second, depending on the application workload.

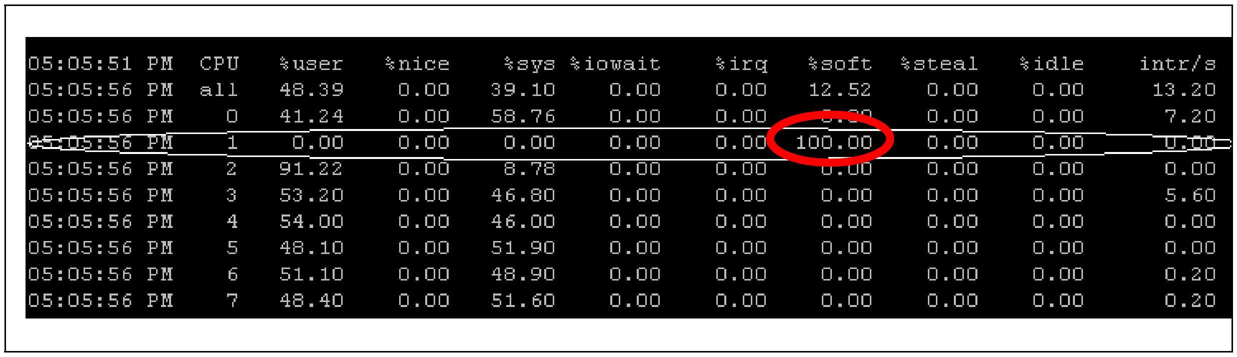
## 5.3 New System Bottlenecks

While a 75% performance improvement is great, given that there are 8 Cores in the test system, one might ask, “Why not an 800% improvement?”

After the last set of benchmarks, the team went back to reanalyze system behavior to see if they had, indeed, removed some of the bottlenecks related to thread synchronization and data copies in memory as discussed in the previous section. They did. However, there was a new bottleneck in the physical receipt of data into the system.

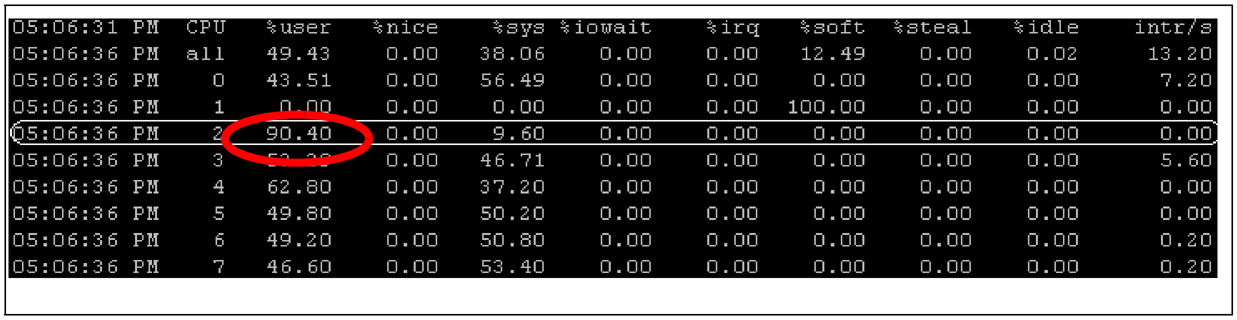
The mpstat output in Figure 10, recorded during the online tests, shows that the packet capture library (i.e. Pcap or PR_RING), was a new bottleneck within the Packet Capture module. Here, the CPU is fully subscribed by servicing software IRQs, as indicated by the mpstat data in the red circles in Figure 10 and Figure 11.

**Figure 10. Packet Capture module Kernel part and NIC driver CPU Utilization**



```
05:05:51 PM  CPU   %user   %nice    %sys %iowait    %irq   %soft  %steal   %idle    intr/s
05:05:56 PM  all   48.39    0.00   39.10    0.00    0.00   12.52    0.00    0.00     13.20
05:05:56 PM    0   41.24    0.00   58.76    0.00    0.00    0.00    0.00    0.00      7.20
05:05:56 PM    1    0.00    0.00    0.00    0.00    0.00  100.00    0.00    0.00      0.00
05:05:56 PM    2   91.22    0.00    8.78    0.00    0.00    0.00    0.00    0.00      0.00
05:05:56 PM    3   53.20    0.00   46.80    0.00    0.00    0.00    0.00    0.00      5.60
05:05:56 PM    4   54.00    0.00   46.00    0.00    0.00    0.00    0.00    0.00      0.00
05:05:56 PM    5   48.10    0.00   51.90    0.00    0.00    0.00    0.00    0.00      0.00
05:05:56 PM    6   51.10    0.00   48.90    0.00    0.00    0.00    0.00    0.00      0.20
05:05:56 PM    7   48.40    0.00   51.60    0.00    0.00    0.00    0.00    0.00      0.20
```

**Figure 11. Packet Capture module userspace part and Packet Classifier CPU Utilization**



```
05:06:31 PM  CPU   %user   %nice    %sys %iowait    %irq   %soft  %steal   %idle    intr/s
05:06:36 PM  all   49.43    0.00   38.06    0.00    0.00   12.49    0.00    0.02     13.20
05:06:36 PM    0   43.51    0.00   56.49    0.00    0.00    0.00    0.00    0.00      7.20
05:06:36 PM    1    0.00    0.00    0.00    0.00    0.00  100.00    0.00    0.00      0.00
05:06:36 PM    2   90.40    0.00    9.60    0.00    0.00    0.00    0.00    0.00      0.00
05:06:36 PM    3   53.29    0.00   46.71    0.00    0.00    0.00    0.00    0.00      5.60
05:06:36 PM    4   62.80    0.00   37.20    0.00    0.00    0.00    0.00    0.00      0.00
05:06:36 PM    5   49.80    0.00   50.20    0.00    0.00    0.00    0.00    0.00      0.00
05:06:36 PM    6   49.20    0.00   50.80    0.00    0.00    0.00    0.00    0.00      0.20
05:06:36 PM    7   46.60    0.00   53.40    0.00    0.00    0.00    0.00    0.00      0.20
```

# 6.0 Conclusion

The two-stage packet pipeline and flow-pinning software architectures pioneered by the first SNORT* multi-threading effort (i.e. for v2.2.0, as documented in [1]) was applicable for multi-threading the latest version of SNORT (i.e. v2.8.0.1). However, it was not a “drop-in” set of changes; the team needed to modify the way data was shared between threads in order to gain real performance improvement.

Once system and application analysis and programming work was complete, the multi-threaded version of Snort v2.8.0.1 yielded between 10% and 75% performance improvement, depending on the test load and rule set, and was only really limited by the CPU being able to bring in data fast enough off the network.

## 7.0 Next Steps

During the final benchmarking, mpstat indicated that the current system bottleneck is in the Packet Capture module. Specifically, the CPU is spending almost 100% of its time responding to packet receive interrupts.

Intel® I/O Acceleration Technology (Intel® I/OAT) is a large set of hardware accelerations in Intel chipsets and network devices that aim to reduce CPU overhead for packet reception. Future versions of SNORT, multi-threaded or not, should take advantage of them.

For example,

- The Intel® Ethernet Controllers 82575 and 82598 contain multiple receive and transmit queues and L2 sorting logic that can be used to off load some or all of the Packet Classifier functionality.
- The Intel® 5100 chipset contains an Enhanced DMA engine that can be used to improve packet copy performance.

# Appendix A Performing Benchmarks

This Appendix contains instructions for performing the benchmarks described in Section 5.0.

## A.1 Offline packet trace file test

1. Compile and Install
   Enter SNORT* directory and compile the code
   Single-threaded (original) version:
   ```
   ./configure
   make
   ```
   Multi-threaded version:
   ```
   ./configure --enable-pipeline_mthread --enable-cpu_affinity --with-num-cpus=8 --with-num-process-threads=5
   make
   ```
2. Run SNORT* with an offline packet trace
   a. Enter SNORT* directory and start the executable with the default configuration file
      ```
      src/snort -c etc/snort.conf -l . -K none -k none -r ../http-1KB-35k.pcap
      ```
      Note on SNORT* options:

| | |
|---|---|
| — -c etc/snort.conf | Use the default configure file etc/snort.conf |
| — -l . | Use current directory as the log directory |
| — -K none | Disable logging |
| — -k none | Disable packet checksum |
| — -r | ./http-1KB-35k.pcap - Read and Process packet trace file http-1KB-35k.pcap |

3. Repeat with another offline packet trace
   ```
   src/snort -c etc/snort.conf -l . -K none -k none -r ../http-32KB-2k-30s.pcap
   ```
4. Record results
   Error! Not a valid link.
   Packet trace files info:
   http-1KB-35k.pcap:
   - — Total pkt: 11039801
   - — Total tcp sessions: 1004103
   - — TCP connection speed: 35,000/sec
   - — File size: 831MB

   http-32K-2k-30s.pcap:
   - — Total pkt: 4902174
   - — Total TCP sessions:88995
   - — TCP connection speed: 2000/sec
   - — File size: 359MB

## A.2 Online test

1. Compile and Install: (same as Offline test case)
2. Adjust TCP connection speed on a live network connection to make sure packet drop rate less than 0.05% (example below uses eth1).
3. Run SNORT* on the live network connection
   a. Enter SNORT* directory and start the executable with the default configuration file

      ```
      src/snort -c etc/snort.conf -l . -K none -N -i eth1 -n 5000000
      ```

      Note on SNORT* options:
   - -c etc/snort.conf -- Use the default configure file etc/snort.conf
   - -l . -- Use current directory as the log directory
   - -K none-- Disable logging
   - -i eth1 -- Listen on eth1
   - -n 5,000,000-- Exit after receiving 5,000,000 packets

Result:

**Table 3. Running SNORT* on the live network connection**

| File size | Snort v2.8.0.1 standard version (TCP Connections Requests per second) | Snort v2.8.0.1 parallel version (TCP Connections Requests per second) | Performance boost via optimization |
|---|---|---|---|
| HTTP 64KB data file | 1000 | 1500 | 50% |
| HTTP 32KB data file | 2000 | 3000 | 50% |
| HTTP 1KB data file | 20,000 | 35,000 | 75% |

§ §