White Paper

Dual-Core Intel® Xeon® Processor LV 2.0 GHz

Communications and Networking Applications

Supra-linear Packet Processing Performance with Intel® Multi-core Processors

1 Executive Summary

Advances in semiconductor technology are making way for low-power multi-core processors. This transition enables embedded and communications applications to more broadly deploy systems with multiple execution cores due to the more favorable economic and thermal characteristics of this generation of processor technology.

This transition to multi-core processors promises more than an increase in the number of execution cores to increase computation capability. It offers additional flexibility for development and optimization of higher performance applications. In particular, multi-core architectures can significantly improve program flow so that cache memory associated with each individual execution core is used more effectively. With multiple caches available to software developers, it is possible to optimize data locality driving higher cache-hit rates and improved overall application performance.

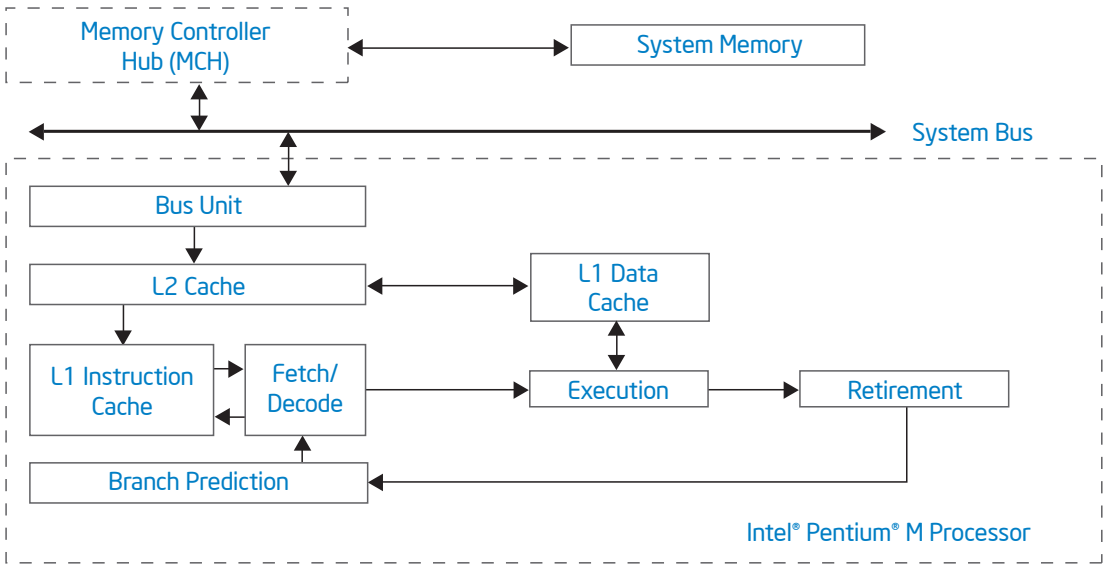
This paper describes the performance benefits that can be realized from the utilization of multiple processing cores for communications applications. The paper illustrates the programming concepts of pipelining and flow-pinning in a packet-processing application and presents performance results from the use of these techniques on a popular open source intrusion detection application. Using a real-world network traffic scenario with a high number of TCP connections, the results show supra-linear (more than 6.2X) performance improvements when IP traffic flows are distributed among four cores versus running the same application on a single core.

2 Performance Improvements Derived From Cache Memory and Locality of Reference

The performance of most CPUs is highly dependent upon the availability of data and instructions to the execution unit. To lower data latency, the execution unit is surrounded by small pieces of high speed static RAM (SRAM) known as cache memory. This approach minimizes the need for the CPU to fetch data from system memory, avoiding significant delays.

A simplified block diagram of the Intel® Pentium® M processor and its cache implementation is shown in Figure 1. The system bus interfaces to the Memory Controller Hub (MCH) which accesses system memory. Level 2 (L2) cache is connected to the bus unit and can store prefetched instructions and data. Level 1 (L1) caches feed the instruction decoder and supply data to the execution unit. The fetch/decode, execution and retirement units

Figure 1: The Intel® Pentium® M Processor Microarchitecture



execute instructions and comprise what is generally known as the execution pipeline. The branch prediction unit is an integral component that predicts which memory will be needed by the execution unit and instructs the pre-fetcher to load the caches accordingly.

For the Intel Pentium M processor, the latency of system memory, L2 cache and L1 cache are on the order of 240, 14, and 3 internal clock cycles, respectively. In other words, L1 cache provides better than a 5 times improvement in data latency compared to system memory. Cache efficiency is the performance linchpin to most modern single and multi-processor systems.

A measure of cache efficiency (e.g. L2) is the cache-hit rate which indicates the percentage of time the execution unit is fed from the L2 cache. The cache-hit rate often correlates to the application program's locality of reference. Locality of reference is the degree to which a program's memory accesses are limited to a relatively small number of addresses. Conversely, a program that accesses a large amount of data from scattered addresses is less likely to use cache memory efficiently. For example, a packet-processing application that is performing TCP reassembly on a large number of TCP flows is likely to access a large amount of data over a large range of memory locations. This results in a lower locality of reference than an application that is performing TCP reassembly on a smaller number of TCP flows.

By distributing TCP reassembly among several execution cores, it is possible to improve the locality of reference in a packet processing application by "pinning" individual TCP flows to a single execution core. This approach can be generalized to apply to other types of network connections between devices such as IPSec flows or IP header compression contexts. Since there is typically little or no locality of reference between different traffic flows like these, reducing the number of different traffic flows processed by each core can improve cache efficiency. Each execution core services a subset of the flows which confines memory accesses to a smaller set of memory addresses and potentially increases cache efficiency.

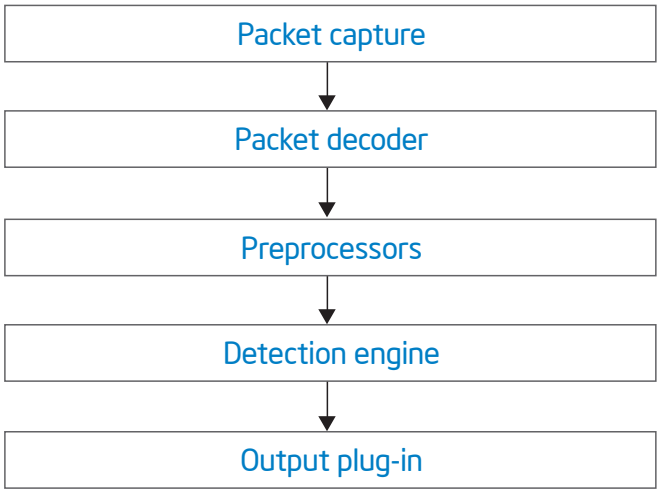
3 Intrusion Detection Application and Locality of Reference

A popular intrusion detection application, Snort*[1], was chosen to evaluate performance improvements from flow-pinning with multi-core processors. The Snort application was modified to run multiple threads and to pin flows to execution cores. The modified application was tested on three system configurations that varied the number of execution cores and the use of pipelining and flow-pinning.

3.1 Snort* DataFlow and Functional Decomposition

The Snort dataflow can be subdivided into five functional processes, shown in Figure 2. Incoming network traffic is processed using the PCAP (packet capture) library, an open source network traffic capture library used by a wide variety of BSD, Linux* and UNIX* applications. This step captures packets in their raw form. Next the packets are passed to the packet decoder which identifies the protocol and breaks down the packet according to its particular datalink specification. Next the preprocessors facilitate stage four, the detection engine, by reassembling IP fragments and TCP streams. The preprocessors also check the packets for suspicious activities that are outside the scope of the detection engine and potentially modify the packets so the detection engine can process the packets more effectively. The detection engine performs a series of tests to detect intrusions based on a set of user-defined intrusion signatures. Finally, reports of suspicious activities are generated by one or more output plug-ins.

Figure 2. Snort* Dataflow



3.2 Snort Functional Decomposition

The Snort dataflow shown in Figure 2 was adapted to run in three configurations and the resulting performance was measured. These configurations varied the number of execution cores and the use of functional pipelining and flow-pinning. Functional pipelining is a technique that sub-divides application software into multiple sequential stages and assigns these stages to dedicated execution units. The first execution unit runs the first application stage and then passes the intermediate results to a second execution unit that runs the next application stage, and so on. Pipelining can increase locality of reference since each execution unit runs a subset of the entire application, potentially increasing the cache hit rate associated with executing instructions.

Configurations:

1) All five Snort functional processes run on a single execution core, Figure 3.
2) All five Snort functional processes run on each of four execution cores in parallel, Figure 4. TCP packets are distributed to each core in a round-robin order. This configuration does not implement pipelining or flow-pinning.
3) The first Snort stage, Packet Capture, is pipelined and it runs on one execution core. The remaining four Snort functional processes all run on cores 2, 3, and 4 and TCP flows are assigned to these cores using flow-pinning, Figure 5.

Figure 3: Snort running on a single execution core

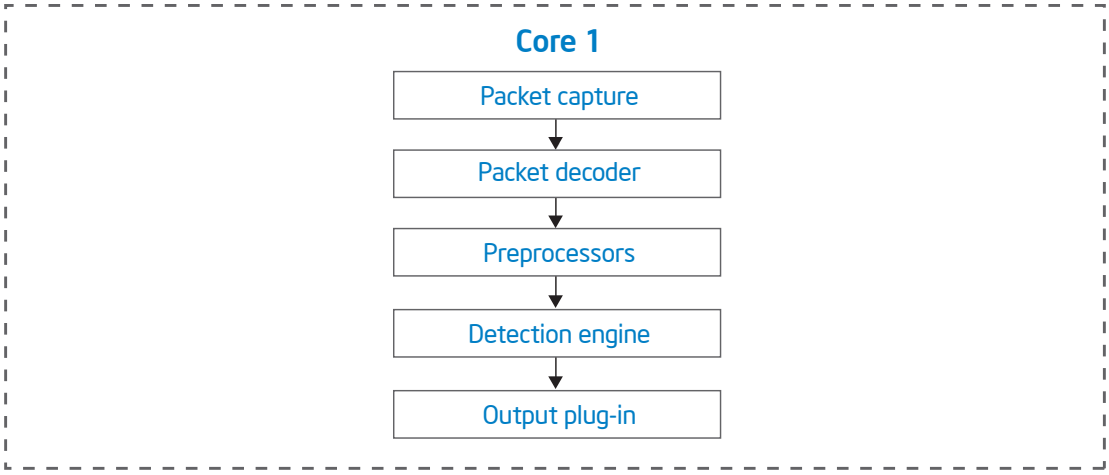


Figure 4: Snort running on four execution cores in parallel without pipelining or flow-pinning

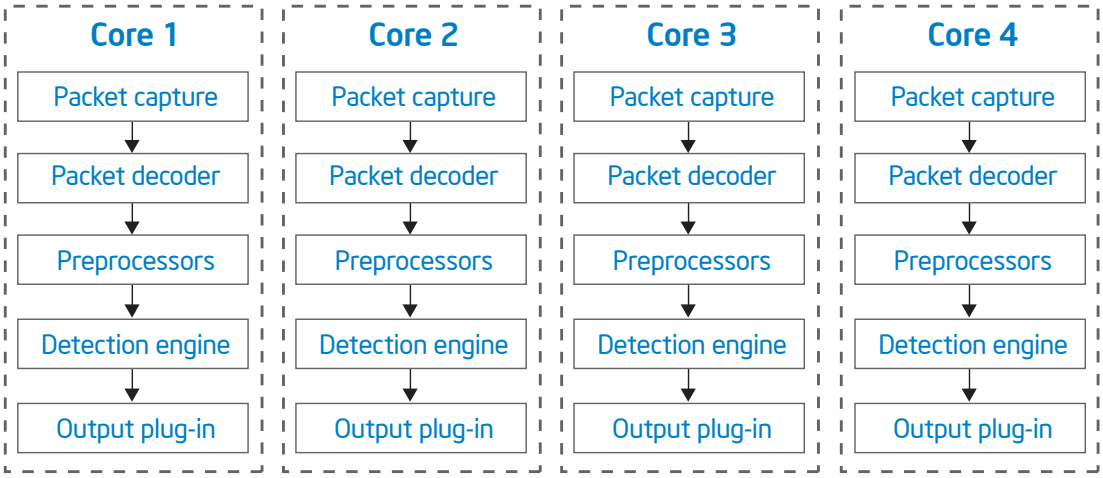
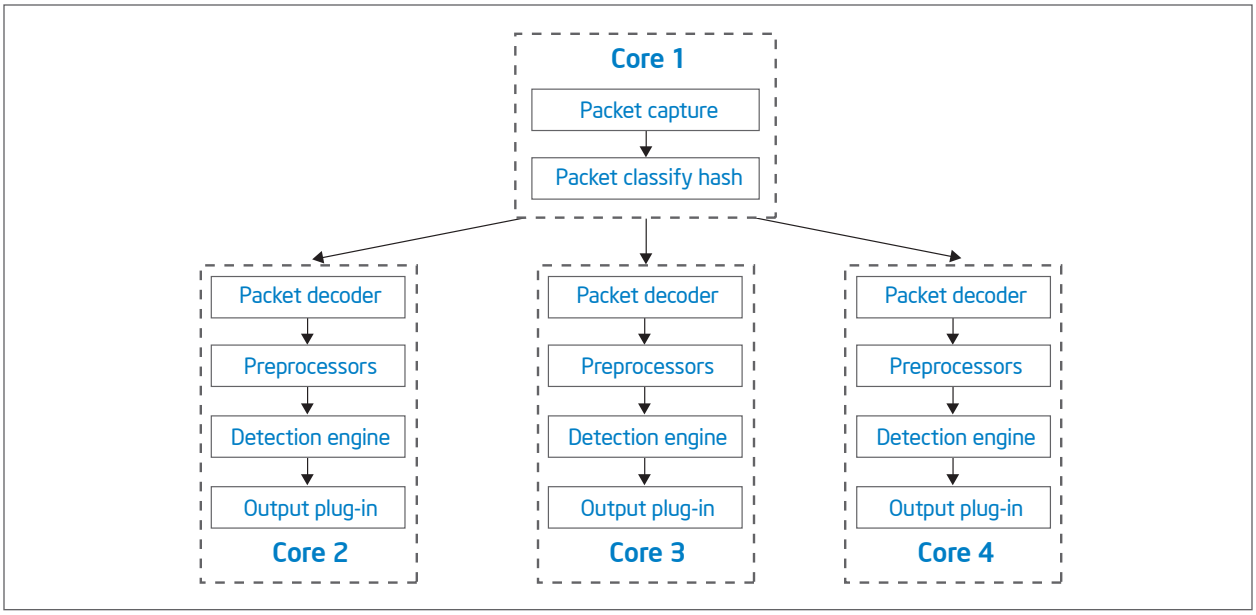


Figure 5: Snort running on a four execution cores with pipelining and flow-pinning



Flow-pinning is orchestrated by the packet classify hash running on core 1 which directs network traffic to cores 2–4. The hash algorithm assigns packets to cores 2–4 to increase the locality of reference and to better utilize cache space.

The hash function in this flow-pinning example classifies packets using the IP source address and destination address fields, as well as the TCP source port and destination port fields. This hash function ensures that packets from the same TCP flow will always be assigned to the same core.

Each of the three configurations was tested with two sets of packet traces. One packet trace was obtained from a database server within Intel and had a relatively low number of concurrent TCP connections (about 175). The other trace was obtained from the National Laboratory for Applied Network Research (NLANR) on a link that services a wide-area network and had a relatively large number of TCP connections (about 25,000, typical of a small-to-medium enterprise LAN).

The test system combined the Dual-Core Intel® Xeon® processor LV2.0 GHz in a dual processor configuration with the Intel® E7520 Memory Controller Hub and 3GB of DDRII memory. The Intel® VTune™ Performance Analyzer was used to measure L2 cache hit rates. The Intel VTune Performance Analyzer enables runtime sampling with minimal disruption to program execution.

3.3 Test Results

Testing indicated that average L2 cache hit rates and associated throughput vary significantly with the number of connections serviced. For the trace with a small number of connections, performance was similar for single core and multi-core configurations. In addition, implementation of pipelining and flow-pinning techniques yielded only minimal incremental throughput (See Table 1).

Table 1

	175 TCP connections	
	Throughput (Mb/s)	**L2 Cache hit rate**
Single execution core without pipelining or flow-pinning (see Figure 3)	566	99%
Four execution cores without pipelining and without flow-pinning (see Figure 4)	543	86%
Four execution cores with pipelining and flow-pinning (see Figure 5)	576	94%

This may be explained by the ability of the single core system to maintain a high L2 cache hit rate of 99% while servicing the low number of TCP connections.

Results were significantly different for the trace with a larger number of TCP connections. In this scenario, the single execution core system L2 cache hit rate decreased to 41% (from 99%) and throughput decreased dramatically. (See Table 2).

A similar performance decrement was observed when four execution cores ran the application in parallel without pipelining or flow-pinning. However, when pipelining and flow-pinning techniques were applied using four execution cores, and the resulting throughput was more than 6.2 times as great as the single core system. This configuration also had a substantially greater L2 cache hit rate than the single execution core system (70% versus 41%). (see Table 3).

Table 2

	175 TCP connections		25,000 TCP connections	
	Throughput (Mb/s)	**L2 Cache hit rate**	**Throughput (Mb/s)**	**L2 Cache hit rate**
Single execution core without pipelining or flow-pinning (see Figure 3)	566	99%	30	41%

Table 3

	25,000 TCP connections	
	Throughput (Mb/s)	**L2 Cache hit rate**
Single execution core without pipelining or flow-pinning (see Figure 3)	30	41%
Four execution cores without pipelining and without flow-pinning (see Figure 4)	29	42%
Four execution cores with pipelining and flow-pinning (see Figure 5)	188	70%

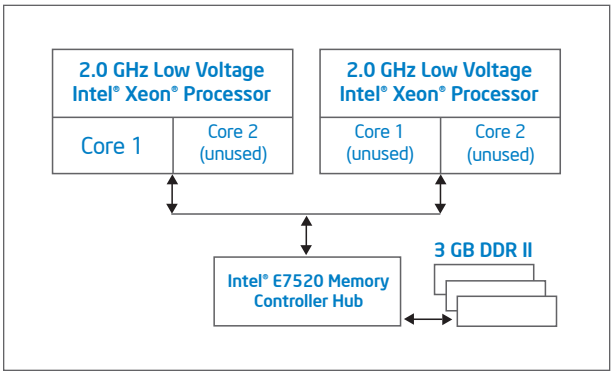
4 Conclusion

Results indicate that Intel multi-core processors can significantly increase performance for packet processing applications such as intrusion detection. Running the application in parallel across four execution cores utilizing pipelining and flow-pinning programming techniques generated more than six times the throughput of a single core configuration. In a test environment approximating "real-world" network traffic (25,000 TCP connections), this hardware/software combination increased L2 cache-hit rate significantly. Software tuning to increase L2 cache hit rate beyond the 70% recorded here could potentially yield even greater performance improvement.

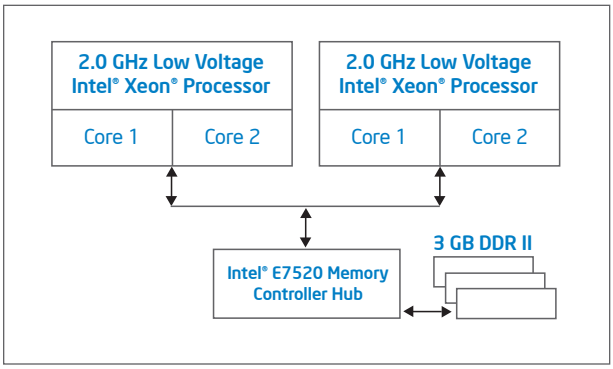
Since the Snort application is a rather typical example of packet processing, it is likely that the supra-linear performance gains from adding execution cores can be generalized to other applications with a high degree of packet processing requirements.

5 Appendix

System Configuration #1: Single execution core (see Figure 3)

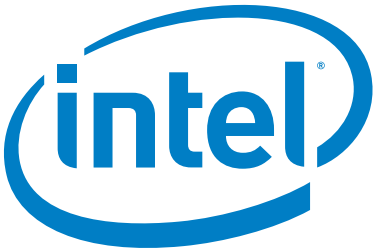


System Configuration #2 and #3: Four execution cores (see Figure 4 and Figure 5)



Platform Configuration:

1. 2 x Dual-Core Intel® Xeon® Processor LV 2.0GHz, Intel® E7520 MCH, DDR2 - 400 MHz 3072M, 8 DIMMS, each with 512MB memory, 667 MHz Front-Side Bus Speed, Redhat* Linux* EL4.0 Update 1 (smp kernel 2.6.9-11), Intel® VTune Performance Analyzer 7.2 (Remote Data Collector), BIOS: ALGHB025



www.intel.com

[1] Snort is an open source system utilizing a rule-driven language that combines the benefits of signature, protocol and anomaly based inspection methods. Please see www.snort.org for further details on this security application.

0206/KSC/QUA/PDF ♻ Please Recycle 311566-001US