System Requirements for Flow Processing

Raj Srinivasan
Director, Software Engineering
Bivio Networks, Inc.
12 Sept 2006

Abstract

With the ever growing array of threats to computer systems, security applications have evolved from simple firewalls to extremely complex entities with very sophisticated requirements. These requirements need to address functionality, performance, and scalability. Functionality spans firewalls, VPNs (IPSec and SSL), IDS/IPS, AV/AS, and server off-load functions, among other things. Performance includes both the ability to pass packets through the system at multi-gigabit rates, as well as being able to do deep packet inspection/processing at high speeds. Scalability must address both of these aspects of performance - i.e. packet throughput and processing power. It is also important to consider system management, maintainability, ease of use, and reliability. In this paper, we examine these aspects of the system, and propose an architecture that not only meets these requirements, but will also be flexible enough to support future application needs.

Introduction

In this paper, we first describe an architecture for processing flows. We then take a look at each aspect of the architecture, and qualitatively describe how it meets the requirements for processing flows in an optimal way.

Flow Processing Requirements

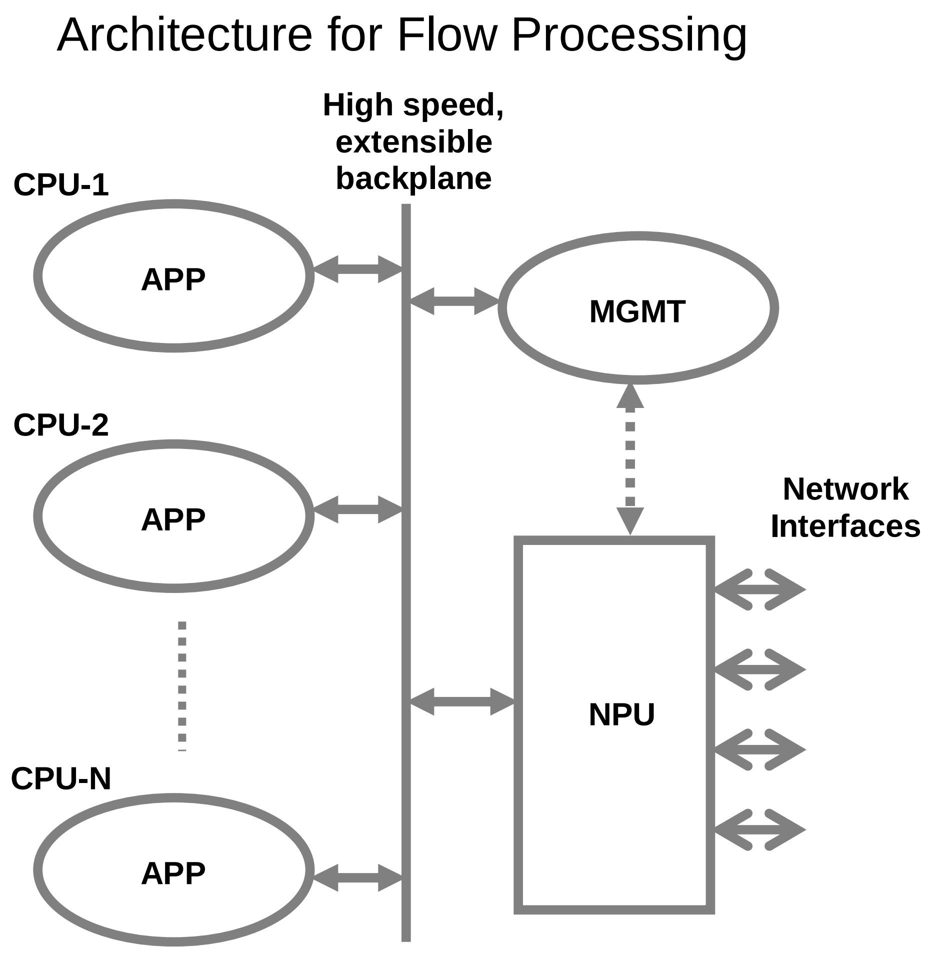
The following requirements are addressed in this document.

- Performance – while we don’t state a specific number, it is clear that today’s performance requirements are in the multi-gigabit range. Small and large packets must be processed at wire speeds, which could range from 1Gbps to 10Gbps or higher.
- Scalability – the system architecture must be scalable so that different performance requirements can be addressed with meaningful cost structures.
- Functionality – the architecture must be versatile, in that it should support multiple applications. This translates to the requirement to be able to process flows of different types. Flows may constitute single or multiple network associations (i.e. connections, for example) between cooperating applications distributed over a network.

- Manageability – the architecture should be easily manageable. Ease of configuration, management and maintenance are crucial to the adoption of this architecture.
- Application maintenance and portability – when applications are developed for this architecture, they should still maintain a general flavor that will support other architectures. The architecture must shield applications from getting locked into specific hardware or software features.

System Architecture

The following diagram shows our proposed system architecture.


Architecture for Flow Processing
High speed, extensible backplane
CPU-1
APP
MGMT
CPU-2
APP
Network Interfaces
NPU
CPU-N
APP

This architecture utilizes a combination of general purpose processors (CPUs) running the Linux operating system, and a new generation Network Processing Unit (NPU) for performing various flow operations. One processor (MGMT), also running Linux, is reserved for purely management functions, while other CPUs run applications. All of the

CPUs and the NPU reside on, and communicate over, an extensible, high speed backplane that operates at multiples of 10Gbps. The external network interfaces, consisting typically of 1Gbps or 10Gbps Ethernet, are connected to the NPU.

In addition, each application CPU has attached to it one or more co-processors over a very fast bus (PCI Express, for example). These co-processors typically perform such functions as cryptographic encryption and decryption, key generation, regular expression processing, etc.

The MGMT CPU will typically have its own private Ethernet port and console port for management purposes. It will also support storage, for example, in the form of an attached SCSI/RAID disk system.

The Networking Environment

The system supports one or more network interface cards, with each card supporting multiple ports (for example, eth0, eth1, and so on). All configurations are done on MGMT. Through a process of virtualization, these interfaces are replicated on every application CPU. The system software ensures that interface states are identical on all the processors. Thus, applications have direct access to the external interfaces, from every processor on the system. There is a special Ethernet interface in each CPU – bp0 – that enables applications to communicate over the backplane. The backplane essentially looks like a very high speed, private Ethernet network, not visible from the external network. Applications may communicate with each other over the backplane using standard socket based communication methods. Thus, every processor, including MGMT, has an identical networking environment.

The File System Environment

Application CPUs all boot using standard protocols from the management CPU, and mount the storage attached to MGMT. Thus all CPUs have identical file system environments.

NPU Functions

The NPU performs the following functions:

- Load-balance incoming traffic to applications based on multiple algorithms.
- Support bindings, set using special APIs, that enable special flow processing. For example:
    - Cut through flows.
    - Direct specific flows to applications requesting them.
- Perform QoS functions.
- Maintain, and report full flow statistics.

- Support application specific flow processing, either dynamically through APIs, or statically through application specific code, linked to NPU processing through architected hooks for intercepting packets between input and output.

NPU bindings are set by applications over the backplane using protocols designed for this purpose. These bindings must be such that the API calls used to set them must have very low latency.

Benefits of Proposed Architecture for Meeting Flow Requirements

We are now ready to consider the benefits of this architecture in fulfilling flow requirements. We start by with comparing the architecture with specialized hardware implementations.

Comparison with Specialized Hardware

One approach to flow processing is to use hardware developed specifically to process flows in certain ways. There is no doubt that specialized hardware will outperform more general systems for specific functions. However, such systems necessitate a number of compromises, among which are:

- Flexibility. Unlike general purpose hardware which is programmable, often ASICs are designed to be used in very limited ways, with limited programmability. These are okay for well established protocols which operate at low levels, but definitely not for generalized flows where applications are constantly evolving. It is not feasible to require a new spin of an ASIC in order to present new features to customers, for most application vendors.
- Scalability. Often, ASICs lack the general purpose hardware's ability to be clustered, or stacked. Thus scalability needs to be built with external "glue" hardware, often at a huge cost.
- Performance. This may seem like a contradiction, since ASICs are built with performance in mind (in addition to volume considerations for decreasing cost). However, we need to keep in mind that in order to make ASICs support multiple applications, some amount of generality needs to be embedded in it. This makes it more like a CPU, and often, it runs at a fairly low clock frequency. Thus, while a specific function (such as cutting through flows, or bulk encryption) may outstrip the capabilities of generalized hardware, combinations of functions will severely limit performance.
- Application portability. When portions (or all of) applications are changed to use specialized hardware, the application will lose its generality. It is often a huge effort to split an application so that certain portions of processing are delegated to special hardware. This makes it very hard to maintain the applications portability to other systems – for example, in a product range that meets different

performance levels – and locks in the application to the specific hardware components.

Thus, where all of the above requirements are important, and it is of utmost importance to be able to change quickly to conform to new conditions, a more generalized hardware would be the most appropriate way to go. Note that what we say about ASICs applies to FPGAs also.

Performance

Until recently, simple architectures sufficed for a couple of reasons: firstly, performance requirements have not been very high for flow processing applications, and CPU speeds have been keeping up with Moore’s Law. Both of these considerations have undergone major changes recently.

Flow processing speeds are now required to be in the 10Gbps range by service providers. Even though CPU speeds are very high – Intel processors run close to 4GHz – flow throughput is still being limited by factors other than CPU speed. One of the limiting factors is memory access. Even with the fastest memory (DDR2, 600 MHz, 64-bit access), practical considerations limit access to well under 10Gbps. This bandwidth needs to be utilized for processing (running code and accessing local variables), as well as for transferring data in and out of memory. Thus, however fast the CPU may be, memory bandwidth considerations will limit the amount of data that can be processed to a few hundred megabytes of small packets to a few (usually 2) gigabits of large packets. This is true even for SMP clusters, since such a cluster will operate on a single memory (even if distributed among multiple banks using dual controllers).

Moreover, if there is a need to integrate multiple applications – as in the case of UTMs (Unified Threat Management systems) – in order to meet complex flow processing requirements, the amount of processing that needs to be done for each flow increases drastically. Since Moore’s law seems to have hit a limit (witness Intel’s emphasis on dual cores now) the only way to scale is to use more processors in a loosely coupled scheme. Each processor may be a multi-core unit in the above architecture, so one gets the best of both worlds.

Scalability

The most common design for a flow processing consists of a CPU subsystem (an Intel dual-core, typically) attached over a fast bus such as PCI Express to an NPU that incorporates a crypto engine, and a regular expression processing engine. Aside from the single system performance considerations we have noted above, one huge drawback of such system is their scalability. The missing piece is really the extensible backplane incorporated in the architecture we have presented. This backplane allows additional CPUs, NPUs, and their associated resources, to be added to the system.

One of the tremendous advantages of the architecture we have presented is that scaling is almost linear when more CPUs are added. This depends on the NPU to perform a clean partitioning of flows, so that the need to synchronize with other CPUs is minimized. We will discuss this some more later.

Also, having specialized engines for cryptographic functions and regular expression processing be attached to each CPU via a fast bus such as PCI Express allows scaling of applications that use these functions. If these resources were combined with the NPU – as do many architectures – two problems present themselves. One problem is that communication between the CPUs and the NPU could become considerable, thus affecting scaling. Secondly, since a given NPU has fixed co-processing capability, this resource cannot be easily scaled.

A good side-effect of our architecture is the built-in redundancy. With multiple CPUs available, if one CPU fails, it can be backed up using multiple schemes. Typically, the NPU will detect this condition, and load-share the flows targeted at the failed CPU among the active CPUs.

Functionality

By keeping the primary functionality in the application CPUs, and using a generally accepted operating system such as Linux, one maximizes the ease with which new functionality can be introduced. Indeed, there is a plethora of software available in the realm of open and free source for Linux, and it will be fairly easy (usually requiring a compilation only) to port this software to our architecture. Indeed, we have done precisely this in our implementation. Moreover, there are architected means to employ multiple applications in the same CPU, as well as divide CPUs into groups where each group processes flows requiring a specific application.

We are not saying that it is impossible to provide new functionality in other architectures. Only that it will be a non-trivial port, often requiring NPU specialists.

Manageability

This is very often an overlooked function, which requires considerable resources from the system. In most systems, management has to coexist with applications in the same CPU. If the system is busy, response time will be very poor when an administrator tries to perform configuration or maintenance. This is not acceptable when there are situations which require immediate action (such as a breach of security).

In our architecture, this problem is solved by dedicating a CPU (MGMT) which is solely used for management purposes. The CLI and GUI run on this CPU. It has its own, dedicated, management Ethernet port over which the system is always accessible and response is guaranteed to be immediate. This CPU can be used to consolidate and preprocess log information and statistics, if necessary, thus freeing the application CPUs for flow processing.

One very important function performed by the management CPU is general monitoring of system health. It watches hardware, the system environment (such as temperature), and the running applications. When a failure is detected, the criticality of the failure will be assessed, and corrective action will be taken immediately. This action may be to restart a failed application, shutdown a non-functioning CPU, signal a secondary (active or standby) system to take over, reroute traffic on a failed link, or failopen interfaces (for an IPS application, for example) to take the system offline due to non-recoverable errors.

Thus, the importance of a clean separation of management functions from applications cannot be overemphasized.

Application Maintenance and Portability

Common sense dictates that this is an important consideration for commercial ventures. The architecture we have proposed ensures this by providing a full Linux environment for applications, requiring only minimal changes, if any, for functioning in this environment. Indeed, we have ported open source applications to this architecture with only a recompilation. A beneficial side-effect is that applications do not get locked in to a particular architecture.

Some Additional Scaling Considerations

We said before that scaling using this architecture is linear. In this section, we expand on this statement.

Scaling is a term that applies to multiple features of a system. It can apply, for example, to network throughput, or to compute power. This architecture promotes scaling in both dimensions.

In order to increase the network throughput of the base system, an additional NPU can be added to the extended backplane. In practice, the additional NPU would be part of a board that might contain additional CPUs. The effect of the additional NPU is to increase the IO capability of the system. We have done this on a similar architecture with very little change to the base code. The main requirement is that the system be aware of the NPUs and their addresses on the backplane.

To increase the available compute power, additional CPUs can be added to an extended backplane. This enables more processing for each flow, for a given maximum system throughput. With enough compute power, wire-speed can be attained for any type of flow processing within reason (it is always possible to create pathological examples that are hard to accommodate).

In an SMP configuration of CPUs, we already saw how memory bandwidth affects throughput. Inherently, scaling using SMP configuration is not linear, because of the effects memory contention, and use of locking to arbitrate contention for critical resources.

In the loosely coupled CPU architecture we have described, scaling can be linear if the NPU is used appropriately. Essentially, by recognizing and grouping associations into sets of related flows that are processed by the same CPU, the need for synchronization between CPUs is drastically minimized. Synchronization may still need to be done for slow-path control functions such as configuration, but these do not affect the fast-path data processing. Aggregation functions such as statistics over the full set of flows (not visible to a particular CPU in its entirety), detecting certain types of anomalies, QoS, etc. are good candidates for implementation in the NPU. Not requiring CPUs to synchronize for these features makes CPU scaling linear.

Summary

In this paper, we have described an architecture which we believe is most suitable for general flow processing. Requirements for flow processing were enumerated, and an architecture for implementing these requirements was proposed. Each aspect of the architecture was discussed qualitatively, and shown to be of benefit in realizing flow processing requirements.

We have implemented this architecture and it has been deployed for processing flows commercially.