A QoS-enabled Packet Scheduling Algorithm for IPSec Multi-Accelerator Based Systems

Alberto Ferrante
Department of Information Technologies
University of Milan
Milano, Italy
ferrante@dti.unimi.it

Vincenzo Piuri
Department of Information Technologies
University of Milan
Milano, Italy
piuri@dti.unimi.it

Fabien Castanier
Advanced System Technology
ST Microelectronics
Agrate Brianza
Milano, Italy
fabien.castanier@st.com

ABSTRACT

IPSec is a suite of protocols that adds security to communications at the IP level. Protocols within the IPSec suite make extensive use of cryptographic algorithms. Since these algorithms are computationally very intensive, some hardware acceleration is needed to support high throughput. In this paper we discuss a scheduling algorithm for distributing IPSec packet processing over the CPU with a software implementation of the cryptographic algorithms considered and multiple cryptographic accelerators. This algorithm also provides support for quality of service. High-level simulations and the related results are provided to show the properties of the algorithm. Some architectural improvements suitable to better exploit this scheduling algorithm are also presented.

Categories and Subject Descriptors

C.3 [**Computer System Organization**]: Special-Purpose and Application-based Systems

General Terms

Algorithms

Keywords

Quality of Service, QoS, IPSec, cryptographic accelerators, scheduling.

1. INTRODUCTION

IPSec is a suite of protocols that adds security to communications at the IP level. This suite of protocols is becoming more and more important as it is included as mandatory security mechanism in IPv6. IPSec is mainly composed of two protocols, Authentication Header (AH) and Encapsulating Security Payload (ESP). The former allows authentication of each IP datagram's selected header fields or – depending on the operational mode that has been selected – of the entire IP datagram. The latter allows encryption – and optionally authentication – of the entire IP datagram or of the IP payload, depending on the operational mode that has been selected, namely the transport and the tunnel modes. The former was designed for being used in host machines, while the latter is for secure gateways. In tunnel mode the entire original IP datagram is processed; the result becoming the data payload of a new IP datagram with a new IP header. In transport mode only parts of the original IP datagram are processed (e.g., the data payload for the ESP protocol) and the original IP header is kept with some small modifications. Through encryption, authentication, and other security mechanisms included in IPSec (e.g., anti-reply), data confidentiality, data authentication, and peer's identity authentication can be provided [1, 2, 3, 4]. IPComp, a protocol for data payload compression, is also included in the IPSec suite of protocols [5].

IPSec is often used to create Virtual Private Networks (VPNs). A VPN is an extension of a private network on a public network (e.g., the Internet) [6, 7]. The extended part of the network logically behaves like a private one. Typical usage scenarios for VPNs are: remote user access to a private LAN over the Internet and connection of two private networks. In these cases a virtual secure channel needs to be created, respectively, from the user's PC to the LAN public access point or from one LAN to the other. Private network public access points are called *secure gateways*. A secure gateway is a router or a router/firewall also running a VPN-enabled software (e.g., an IPSec implementation). All the traffic inside the LAN is usually not protected, while the traffic going out or coming in the LAN through the secure gateway is protected by some security mechanisms.

IPSec has proved to be computationally very intensive [8, 9, 10]. Thus some hardware acceleration is needed to support large network bandwidths, as may be required even in small secure gateways. Usage of mixed hardware-software solutions for this, especially for low-end systems, has become a common practice in the last years [11, 12], since it provides flexibility and performance.

CF'05, May 4–6, 2005, Ischia, Italy.

Supporting quality of service (QoS) has become very important and it will be more in the incoming years. QoS is the ability to provide different levels of service to different fluxes of data. There exist two kinds of QoS: the hard and the soft ones [13] [14]. The former allows to negotiate some network parameters (throughput, maximum latency, ...) and guarantees that the constraints that have been negotiated are respected. The latter tries to provide different quality of treatment to different data fluxes. This is usually obtained by assigning different priority levels to these fluxes. The priority levels are decided at network level and in IPv4 they can be associated to each IP datagram by using a 3-bit header field. In IPv6 the size of this field has been increased to 12 bits, thus allowing to support more priority levels [15]. Priorities can be managed in different ways. The simplest one is to use a FIFO policy on the incoming packets, but other more effective ways exist: Priority Queuing, Custom Queuing, Flow-based Weighted Fair Queuing, and Class-based Weighted Fair Queuing are the most used ones. It is important to note that QoS is only useful in congestion management. When no congestion is experienced on the system, QoS does not introduce any benefice over the flux management and over the performance. While the concept of QoS has to be supported at system level, some support is also needed at accelerator level to enforce it.

While using multiple accelerators for the same task is not a very common practice especially for low end systems, it may allow for expandability at reduced cost. The scheduling algorithm presented in this paper allows to schedule packets to be processed either by the CPU with a software implementation of the considered cryptographic algorithms or by the accelerators. This also enhances scalability: one may use an accelerator tailored for the bandwidth normally required for VPNs and use the CPU to have a further processing capability when a higher bandwidth is required (the secure gateway may also provide support for non-encrypted traffic, thus the total network bandwidth that is available may be larger than the VPNs one). This may be convenient so that the short and not too high peaks over the estimated VPNs bandwidth can be supported at no additional hardware cost. While this may not fit for high-end systems, it may do so for network edge ones, like, for example, secure gateways run by small companies.

This work is based on [16]. In this paper a scheduling algorithm for IPSec multi-accelerator based systems without QoS support is presented. In [17] an overview of multiprocessor scheduling algorithms is given. None of these algorithms fully exploits the following two characteristics which are typical in IPSec packet processing (see Section 3): processing time a priori known and no data dependency exists between different packets. [18] discusses the problem of load balancing (both static and dynamic) on multiple processors. While this could be interesting for many problems, our goal was different: to minimize latency and, possibly, maximize throughput. Furthermore, none of the algorithms discussed above provides QoS support.

In this paper we present a new scheduling algorithm that is suitable to tackle the envisioned application and fully exploits the two characteristics mentioned above. Section 2 describes the architecture of the considered system. Section 3 describes the scheduling algorithm, section 4 presents the model for the simulations and finally the obtained results.

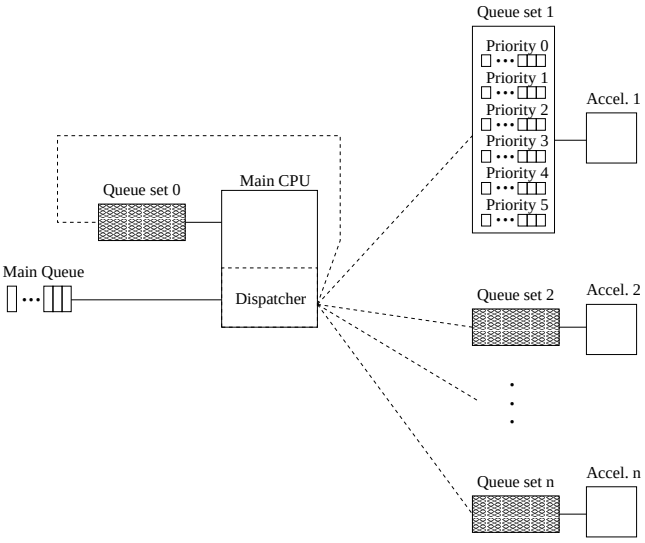

Figure 1: Reference scheme for the scheduling algorithm.

2. SYSTEM ARCHITECTURE

The system considered here is composed by a host computer and N cryptographic accelerators connected to the normal system bus; a PCI bus (32bit, 66MHz) [19] is adopted here as an example. We consider heterogeneous accelerators, i.e., accelerators implementing different cryptographic algorithms and allowing different processing speeds. CPU-memory communication is performed on a faster bus, as in most modern personal computers. The network card is also connected to the faster CPU bus. Only cryptography-related operations are offloaded to the accelerator(s). This means that all the IPSec header processing is done by the CPU. Pieces of data to be processed are stored in main memory and each processor loads them in its local memory by using DMA.

This is only a sample architecture we have used to test the properties of our scheduling algorithm. Higher throughput systems should use different system architectures.

3. THE SCHEDULING ALGORITHM

In the envisioned architecture each accelerator can support different sets of algorithms and different processing speeds. A common interface (i.e., an API) is therefore needed to allow for uniformly accessing to all the cryptographic devices. This common interface should also allow for accessing the software implementations of the cryptographic algorithms.

In this section we present the assumptions on which our scheduling algorithm is based and their motivations, the scheduling algorithm, and some limitations of our approach. In this paper we use the word *job* to identify a set of cryptographic operations that have to be performed on a packet.

3.1 Assumptions

Our algorithm is based on two fundamental characteristics: the first one is that the processing time for packets is known (at least approximately) in advance. This is true for symmetric-key cryptographic algorithms which are normally used within the IPSec context: their processing time only depends on the number of data blocks to be processed. The only exception is for the software implementations of these algorithms: in this case the computation time may vary depending on the current CPU load. The second characteristic is that each packet can be processed independently from the others (i.e., there are no data dependencies between different packets). This comes from IPSec specifications: each packet must carry any data required for its processing [3]. In [20] it is explained how to obtain data independency among packets for AES. Our approach fully exploits these characteristics to achieve high performance.

3.2 Description of the Algorithm

The goal of our algorithm is to minimize the average waiting time of the packets, thus giving the best throughput over average processing latency ratio. Since scheduling jobs on multiple processors is a NP-complete problem, we adopted a heuristic approach. This is the first attempt published in the literature to tackle this problem within a comprehensive design framework. Exhaustive evaluation of heuristic scheduling algorithms to identify the optimal one is behind the scope of this paper.

The main idea underlying our algorithm is to allocate the packets to be processed on the processor (i.e., either one of the accelerators or the CPU) which can provide the shortest processing time for the considered priority level. Our scheduling algorithm processes each packet as follows:

- a set of processors that are able to perform the cryptographic algorithm(s) required by the considered packet is selected,
- the *finishing time* for each of these processors is estimated, being the *finishing time* defined as the sum of the *waiting time* and of the *processing time* of the packet scheduled on the considered processor,
- the packet is scheduled to the processor with the lowest finishing time. When all of the queues are empty new tasks are scheduled to the first fastest accelerator. Each processor processes the assigned packets, stored in 6 priority queues, following a modified version of the flow-based Weighted Fair Queuing (WFQ) priority management mechanism.

In Figure 1 the reference scheme of the algorithm is presented. The scheduler runs on the CPU. As said above, a modified version of the WFQ algorithm is used to manage the priority queues of each processor. The original version of this algorithm assigns processor time based on byte proportion. In our case each packet must be processed as an atomic unit, therefore processor time is assigned to a packet if its processing time is smaller than the available one for the considered priority. If not the remaining processing time is accumulated and the packet is processed in one of the next round robin cycles, as soon as the accumulated time is enough for processing the considered packet. Considering P priority levels, the base time portion (F_p) of the round robin cycle that is allocated to each priority level p ($p \in \mathbb{N}$, $0 \leq p < P$)) can be computed as follows:

$$F_p = \frac{p+1}{\sum_{l=1}^{P} l} \tag{1}$$

Therefore considering a system with 6 priority levels (2 levels among the 8 that are available in IPv4 are not to be used), the portions of the round robin cycle can be obtained with the following formula:

$$F_p = \frac{p+1}{21} \tag{2}$$

The *waiting time*, $w_i(t)$, for the i-th processor at time t is defined as the sum of the processing time of all the packets contained in the related queues and of the *residual processing time* of the packet being processed at that time by the processor. The packets contained in all priority queues must be considered during this computation. An exact computation of this value may lead to confusing results, due to the fact that highest priority packets that may come in the future may change the waiting time for lowest priority ones. Thus waiting time can be instead estimated by using statistics on previously arrived packets. In this paper we identify the CPU as the processor 0.

Let us call $R_{i,p}$ the number of round robin cycles needed before a priority p packet can be processed on the i-th processor. An estimation of this value, $\widehat{R_{i,p}}$, can be computed as:

$$\widehat{R_{i,p}} = \left\lceil \frac{q_{i,p}}{F_p T_{RRi}} \right\rceil \tag{3}$$

where T_{RRi} is the round robin cycle time and $q_{i,p}$ is the sum of the processing times of the packet stored in the priority queue p of the i-th processor. The value obtained here is only an estimation of $R_{i,p}$ since the previous formula does not take into account that each packet has to be considered as a discrete unit as explained earlier in this paper. For each of the forecasted processing cycles we can compute an approximated scheduling. Considering $x_{i,p}^c$ to be 1 if there is a packet to be processed in the queue at priority p at round robin cycle c, and 0 otherwise. At each cycle the scheduling is given by these values. By definition, each $x_{i,p}^c$ is 1 when $c \leq \widehat{R_{i,p}}$ and 0 otherwise. Therefore we can estimate the waiting time for a priority p packet on the i-th processor as:

$$\widehat{w_{i,p}} = T_{RRi} \sum_{c=0}^{C} \sum_{l=0}^{P-1} F_l x_{i,p}^c \tag{4}$$

where C is the number of cycles needed to finish the processing of the packet to be scheduled if it was put in the priority ρ queue, as required. C can be estimated as follows:

$$C = \widehat{Rc_{i,\rho}} \tag{5}$$

Defining for each of the priority queues (i.e., $\forall l \in \mathbb{N}$, $0 \leq l < P$)

$$Cm_{i,l} = min\left\{\widehat{Rc_{i,\rho}}, \widehat{Rc_{i,l}}\right\} = min\left\{C, \widehat{Rc_{i,l}}\right\} \tag{6}$$

we have:

$$\widehat{w_{i,p}} = T_{RRi} \sum_{l=0}^{P-1} F_l \times Cm_{i,l} \tag{7}$$

Finishing time can be computed, by definition, as sum of waiting time ($w_{i,p}$) and processing time ($t_{i,p}^{proc}$) of the packet computed taking into account the proportional round robin policy.

$$f_{i,p} = w_{i,p} + t_{i,p}^{proc} \tag{8}$$

Considering for each of the queues

$$Cp_{i,p} = \left\lceil \frac{t_i^{proc}}{F_p T_{RRi}} \right\rceil \tag{9}$$

where t_i^{proc} is the processing time of the packet on the i-th processor if there was no priority management. Let us also define:

$$Cz_{i,p} = min\left\{Cp_{i,p}, \widetilde{R_{i,p}} - C\right\} \tag{10}$$

$t_{i,p}^{proc}$ can be estimated as:

$$\widetilde{t_{i,p}^{proc}} = T_{RRi} \sum_{l=0}^{P-1} F_l \times Cz_{i,l} + F_p \times Cp_{i,p} \tag{11}$$

By using $\widetilde{w_{i,p}}$ and the $\widetilde{t_{i,p}^{proc}}$ we can estimate the finishing time ($f_{i,p}$) as follows:

$$\widetilde{f_{i,p}} = \widetilde{w_{i,p}} + \widetilde{t_{i,p}^{proc}} \tag{12}$$

That is

$$\widetilde{f_{i,p}} = T_{RRi} \sum_{l=0}^{P-1} F_l \left(Cm_{i,l} + Cz_{i,l}\right) + F_p \times Cp_{i,p} \tag{13}$$

An additive and a multiplicative constants may be used in computing the finishing time of each of the processors. In particular, this can be done on the main CPU's finishing time to control its usage. The previous formula, also considering the two constants becomes:

$$\widetilde{f'_{i,p}} = \alpha_i \left[\beta_i + T_{RRi} \sum_{l=0}^{P-1} F_l \left(Cm_{i,l} + Cz_{i,l}\right) + F_p \times Cp_{i,p}\right] \tag{14}$$

When a packet is put or taken out from one of the queues, just the total waiting time of the corresponding queue needs to be updated. The other values need to be recomputed each time a packet need to be scheduled. These computations do not include any special operation (only addictions, multiplications, and divisions are used). Even though a simplification of this method might be studied for obtaining more efficient implementations.

A non-parametric prediction model can be used for estimating the number of packets that will be in the queues and the average processing time for these packets. The method used here is to compute an k-step moving average each time one of the queues are modified or, in alternative, each l round robin cycles. The moving average value this way obtained is then used as one-step prediction in computing the value of the waiting time of each queue in the above formulas.

Once the cryptographic algorithm(s) to be applied to a packet is known, the *processing time* for this packet can be computed by using formulas corresponding to the considered cryptographic algorithm, while parameters depend on the characteristics of each processor. These characteristics can be provided by the manufacturer or determined through some simple speed tests. An example of a formula used to compute processing time is provided in Section 4.1.

3.2.1 Algorithm Limitations

As already said, this algorithm has been designed to process IPSec packets only with the assumptions given in Section 3.1. For a heterogeneous protocol environment our algorithm may have to be modified. This would be the case if, for example, the TLS protocol [21] also needed to be supported. In fact different TLS packets may exhibit data interdependency. Since our target system is a secure gateway, the situation in which IPSec and TLS are required to coexist should not be very common. TLS works above TCP and it would not make much sense having it on a gateway. Anyway, should this situation or other similar ones happen, two different solutions can be adopted: the first one consists of scheduling the packets that show interdependencies always on the same accelerator. In this way precedence problems are automatically resolved. The second, simpler, solution is to schedule a packet only when all data that are needed are available. If many packets exhibit data interdependencies, the scheduling algorithm may need to be radically changed.

The algorithm presented above cannot guarantee the processing order for the packets, but this is not a problem for IPSec per se.

4. SIMULATIONS

To validate our algorithm and evaluate its performance, we developed a queue-based model. Model simulations and results are here presented. The SystemC language [22] was selected to describe our model since it allows for specifying hardware-software systems. Delays associated with the performed operations can be easily modeled with this language.

4.1 Description of the Model

The SystemC model that can be used to describe our algorithm represents the queues of the cryptographic architecture and the flow of cryptographic requests of the IPSec packets. We are therefore not interested in specific operations performed on data, but only on their delays.

Among all possible conflicts and delays due to communication between the architectural components, we only considered bus contention. RAM contention and other necessary communications between the CPU and the accelerators have been ignored. Providing a highly accurate performance estimation of the considered system is in fact beyond the scope of this work; our main goal is to prove that our algorithm works as desired. Ignoring RAM contention should not introduce a too coarse approximation anyway, since in the considered system the RAM access is much faster than the access to system bus.

A bus contention mechanism, simplified with respect to the PCI standard, has been modeled here. Access to the bus is given to each of the processors which have requested it, one at a time, in the same order as it has been requested. Bus transfer time (that is also the bus lock time) is computed as:

$$t_{bus} = t_{bus\_cycle}(c_{address} + \lceil l_{data}/b \rceil) \tag{15}$$

where: t_{bus_cycle} is the bus cycle period (i.e., $10^{-6}/66$); $c_{address}$ is the time needed to assert the address (1 bus cycle for PCI in DMA mode). l_{data} is the length of the piece of data to be transferred (measured in bytes); b is the number

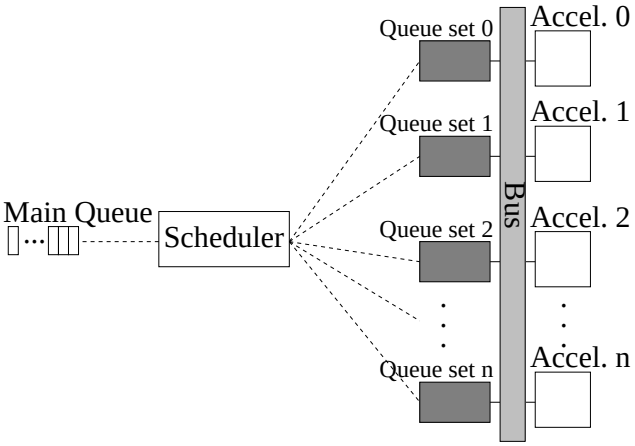

Figure 2: Simulation scheme.

of bytes that are transferred in each cycle (here we consider $b = 4$).

In our model we decided to have AES encryption as the only available operation both in hardware and in software. Time for decryption, depending on the selected algorithm, may differ; for AES encryption and decryption time can be very similar, depending on the implementation. To compute the delays of the AES encryption operation we considered a formula that can be applied to all cipher-block algorithms working on 16-byte blocks (like, for example, AES):

$$t_{enc} = t_{init} + t_{block} \times \lceil b_{data}/16 \rceil \qquad (16)$$

where: t_{init} is the algorithm initialization time; t_{block} is the time needed to encrypt a 16-byte data block.

The SystemC model is shown in Figure 2 and is mainly composed of 6 classes:

- the *accelerator* class describes the functionalities of an accelerator,
- the *scheduler* class implements the scheduling algorithm,
- the *fifo* class implements the FIFOs that are needed for interblock communications,
- the *fifo_accel* class implements the FIFOs used as priority queues,
- the *prio_fifo_accel* class implements the queue sets,
- the *bus* class models the PCI system bus.

To create the model for simulation, the following objects need to be instantiated:

- one *fifo* object to store the input data,
- one *scheduler* object,
- $N + 1$ *accelerator* objects (one represents the CPU),
- one *bus* object,

Table 1: β_0 and corresponding number of bytes that can be processed in such time by one of the accelerators.

β_0 [ns]	Number of bytes
0	0
1.76×10^4	640
6.6×10^5	24,000
6.6×10^6	240,000

- $N+1$ *prio_fifo_accel* objects, used to connect the *scheduler* object to the $N + 1$ *accelerator* ones through the bus,
- 6 *fifo_accel* objects inside each *prio_fifo_accel* object.

Inputs used for the simulations are taken from network trace files, e.g., the ones provided on the Internet Traffic Archive [23] website. These files contain long traces obtained by using the *tcpdump* tool [24] on various networks. We consider a trace taken from a 2Mbit/s gateway and containing about 3 million TCP packets. For our simulations we decided to use only 1 million of these packets to avoid having too long simulations. The only parameter we took from the considered tracefile was packet dimensions, while we ignored the timestamps. Being our target a gateway-like machine, in our simulation we processed the packets as if ESP in tunnel mode was being used. We incremented the packet size read from the tracefile by 40 bytes (i.e the size of a normal TCP/IPv4 packet header) since the sizes contained in ITA files are the ones of data payloads only. Priorities are arbitrarily assigned to the packets based on the source IP address.

In these simulations we considered the throughput for cryptography on the CPU to be of 110Mbit/s. This value is similar to the one that can be obtained on a Pentium III [10]. Even if a more powerful CPU is used, it also has to execute, among other operations, IPSec header processing and TCP/IP packet processing, and to run the packet scheduler. Thus only a fraction of the overall computational capacity can be dedicated to running cryptographic algorithms. We should also remember that multi-GHz Pentium class CPUs are not able to process normal TCP/IP multi-Gbit/s traffic by themselves [25]. Therefore other form of hardware acceleration may be needed in large bandwidth systems.

When multiple accelerators were used, all of them were considered to have the same performance and to support the same cryptographic algorithm. We chose to have accelerators capable of processing a data traffic of about 230Mbit/s.

Values considered for the α_0 parameter are 1, 5, 10, 50, 100, and 500. Table 1 gives some values for the parameter β_0 (in nanoseconds) and the corresponding number of bytes that can be processed by one of the accelerators in such a time.

4.2 Simulation Results

Two different implementations of the algorithm have been considered during the simulations. In the first one, named *RR Average*, the moving average value is cyclically computed each l round robin cycles. In the second one, named

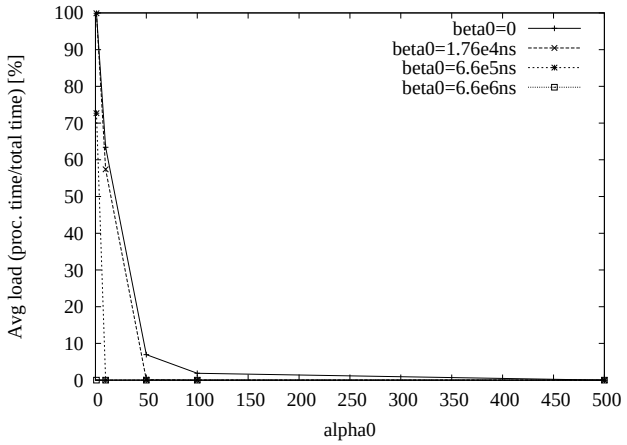

Figure 3: CPU load due to cryptography for a 2-accelerator system in the *Packet Average* case.

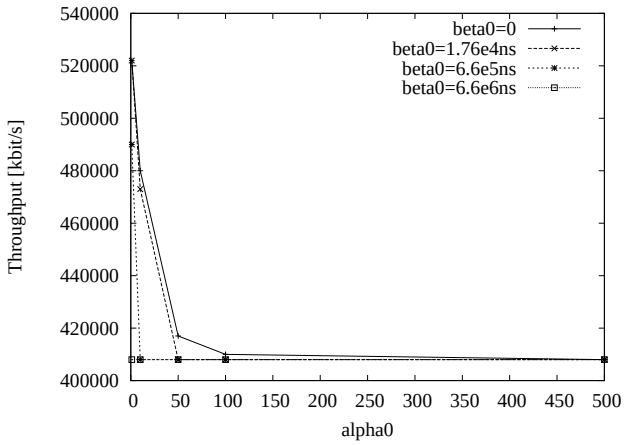

Figure 4: Throughput for a 2-accelerator system in the *Packet Average* case.

Packet Average, the moving average is computed each time a packet goes in or out from the priority queues. Different values for the round robin cycle period, for the moving average window size, and for the moving average computation frequency have been considered in simulations. Results here presented are obtained considering a moving average window size of 1 and a computation frequency equal to one round robin cycle (where it applies). For each processor we defined the round robin cycle time as a multiple of the time needed to process a 40-byte packet on the same processor. Results here shown were obtained by considering a multiplicative factor of 10.

Various figures of merit have been analyzed by simulation to evaluate the performance of our algorithm considering both implementations. The CPU usage (in percent) due to cryptography for the *Packet Average* case is shown in Figure 3. The CPU usage can be controlled through the α_0 and β_0 parameters. The higher β_0, the lower the number of packets processed by the CPU, even for low values of α_0. The higher α_0, the lower the CPU load (and thus the number of packets processed by the CPU itself). These two parameters may

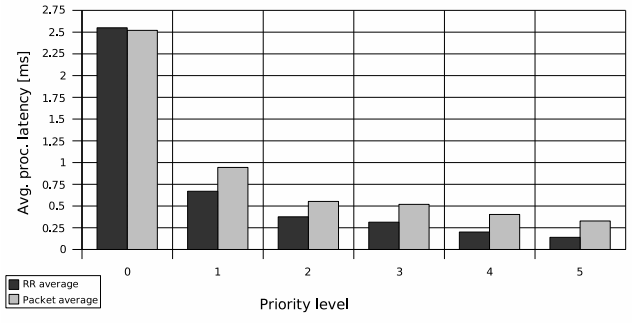

Figure 5: Average processing latency for the *RR Average* and the *Packet Average* cases. $\alpha_0 = 50$**;** $\beta_0 = 1.76 \times 10^4 ns$

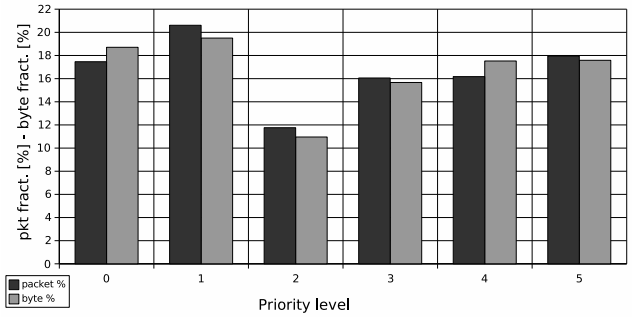

Figure 6: Packet distribution among different priority levels for the network trace we have considered.

even be used to dynamically control the main CPU load: when main CPU load grows, α_0 can be augmented (up to a certain upper limit) to try to avoid having more packets being scheduled on the CPU.

The throughput (in kbit/s) achieved by the whole system for the *Packet Average* case in performing cryptographic operations is shown in Figure 4. Throughput changes accordingly with the CPU load. This is due to the fact that an overloaded system is considered (systems maximum throughput is slightly more than 500Mbit/s, while the required bandwidth would be of 1Gbit/s).

The average processing latency for cryptography in the *Packet Average* case is shown in Figure 7. This parameter represents the average time between when a packet is scheduled and when its processing is completed. Except for low values of the α_0 parameter, processing latency decreases accordingly to CPU load. The CPU is in fact considerably slower (in this case) than accelerators and queuing packets on it enlarges the processing time. Thorughput is anyway higher when the CPU is used. Therefore a trade-off between having high throughput and too high latency can be found. Similar results are shown in figure 8 for the *RR Average* case.

Average processing latency has been evaluated for each of the priority levels. Figure 5 shows a comparison of the average processing latencies we have obtained for the *RR Average* and the *Packet Average* cases. Packets are distributed among different priority levels as shown in Figure

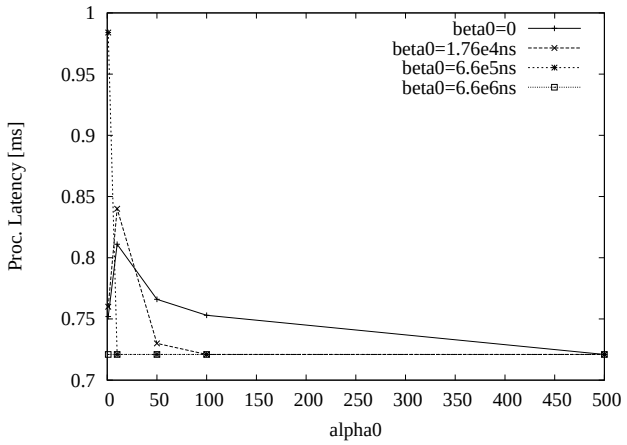

Figure 7: Average processing latency for a 2-accelerator system in the *Packet Average* case.

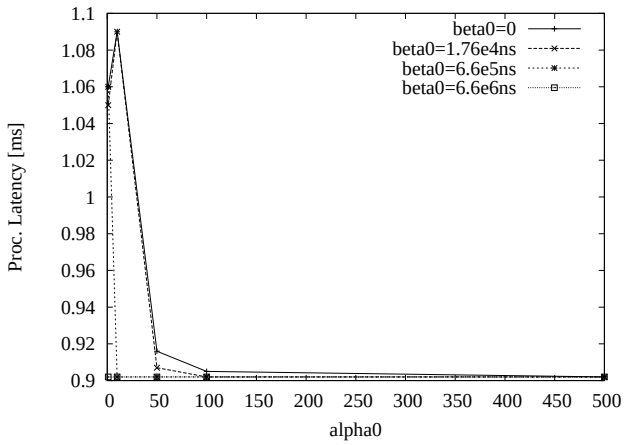

Figure 8: Average processing latency for a 2-accelerator system in the *RR Average* case.

6. There are values of the parameters for which the processing latencies do not respect the priority levels. In most of the cases this is due to the fact that data are not uniformly distributed between different priorities. By using a different data set that gives a uniform distribution of the packets to be processed, we verified that the algorithm works in a proper way. In some other cases wrong algorithm parameters (such as too long moving average computation times) lead to a wrong priority management.

Results for a similar system with 4 accelerators were also obtained. In this case throughput was really limited by bus contention.

By considering a non overloaded system (200Mbit/s of required bandwidth), results are different: the CPU is barely used and both latency and throughput do not change with α_0 and β_0. In fact, in this situation, all data traffic is always processed by the accelerators. QoS is also not useful in this situation as packets can be processed as they arrive.

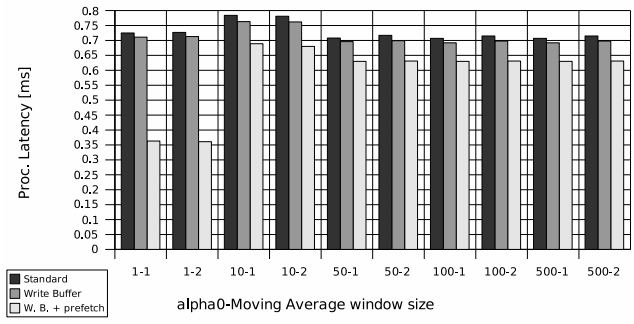

Figure 9: Average processing latency comparison for a 4-accelerator system.

4.3 Enhanced System Architecture

The previous results showed that, especially when the throughput approaches 1Gbit/s, access to the system's bus really limits the performance. Coprocessors got idle in many cases waiting for the opportunity to write back the results in memory or to fetch data to be processed. We therefore propose two enhancements for the system architecture in order to avoid these problems and increase the overall system performance. The first one consists of introducing a write buffer in each accelerator. This allows for decoupling data output from packet processing; the accelerator can therefore continue to process incoming data while processed data are written back to the memory. The second enhancement consists of using data prefetch on each accelerator. Packet in the priority queues are fetched on the accelerators in advance to their processing. This can mess things with priorities when a packet with lower priority is fetched and another one with highest priority comes before the first one is processed. Prefetch is anyway very convenient, since its cost is only due to some additional memory and control logic on the accelerators. Data prefetch allows for decoupling the data fetch phase from the data processing, thus allowing the accelerators to perform data processing, while data for the subsequent computation(s) are fetched.

Comparisons of the results achieved with the enhanced architecture in terms of average processing latency, throughput, and CPU usage are shown in Figure 9, 10, and 11, respectively. A 1Gbit/s required bandwidth and a 4-accelerator system are considered. The β_0 parameter was set to $1.76 \times 10^4 ns$.

A performance improvement can be obtained by adopting the write buffer on the coprocessors. By adopting both enhancements together we can achieve even bigger benefits. By adopting data prefetch and write buffer we are able to exploit all the processing capabilities of the system, thus allowing to reach higher performance.

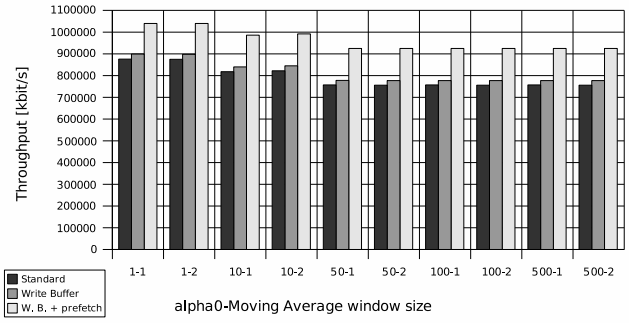

Figure 10: Throughput comparison for a 4-accelerator system.

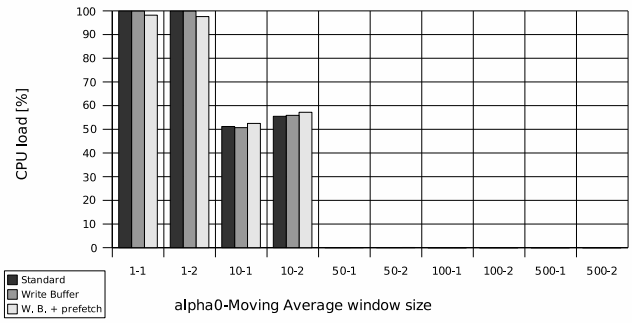

Figure 11: CPU usage comparison for a 4-accelerator system.

5. CONCLUSIONS AND FUTURE WORK

We developed a scheduling algorithm which allows for distributing IPSec packet processing over the CPU and multiple accelerators and to support soft QoS. We also provided some high-level simulations to prove that the algorithm works as desired and that it can provide a performance enhancement especially when the system is overloaded. Some optimizations based on dimensions of the packet is being carried out. A comparison between the algorithm we here presented and some other suitable scheduling algorithms will also be carried out.

6. REFERENCES

[1] S. Kent and R. Atkinson, "Security Architecture For the Internet Protocol – RFC2401," IETF RFC, 1998. [Online]. Available: http://www.ietf.org/rfc.html

[2] ——, "IP Authentication Header – RFC2402," IETF RFC, 1998. [Online]. Available: http://www.ietf.org/rfc.html

[3] ——, "IP Encapsulating Security Payload (ESP) – RFC2406," IETF RFC, 1998. [Online]. Available: http://www.ietf.org/rfc.html

[4] D. Harkins and D. Carrell, "The Internet Key Exchange (IKE) – RFC2409," IETF RFC, 1998. [Online]. Available: http://www.ietf.org/rfc.html

[5] A. Shacham, R. Monsour, R. Pereira, and M. Thomas, "IP Payload Compression Protocol (IPComp) – RFC2393," IETF RFC, 1998. [Online]. Available: http://www.ietf.org/rfc.html

[6] J. Feghhi and J. Feghhi, *Secure Networking with Windows 2000 and Trust Services*. Addison Wesley, 2001.

[7] R. Yuan and W. T. Strayer, *Virtual Private Networks*. Addison Wesley, 2001.

[8] S. Miltchev, S. Ioannidis, and A. D. Keromytis, "A Study Of the Relative Costs of Network Security Protocols." Monterey, CA: USENIX Annual Technical Program, June 2002.

[9] S. Ariga, K. Nagahashi, M. Minami, H. Esaki, and J. Murai, "Performance Evaluation Of Data Transmission Using IPSec Over IPv6 Networks," in *INET*, Yokohama, Japan, July 2000.

[10] Alberto Ferrante, Vincenzo Piuri, and Jeff Owen, "IPSec Hardware Resource Requirements Evaluation," in *NGI 2005*, IEEE, Ed. Rome, Italy: EuroNGI, 18 Apr. 2005.

[11] F.T. Hady, T. Bock, M. Cabot, J. Chu, J. Meinecke, K. Oliver, and W. Talarek, "Platform Level Support For High Throughput Edge Applications: the Twin Cities Prototype," *IEEE Network*, vol. 17, no. 4, pp. 22–27, July 2003.

[12] John Freeman, "An Industry Analyst's Perspective on Network Processors," in *Network Processor Design*, P. Crowley, M. A. Franklin, H. Hadimioglu, and P. Z. Onufryk, Eds. Morgan Kaufmann, 2003, vol. 1, ch. 9, pp. 191–218.

[13] Sean Convery, *Internetworking Technologies Handbook*. Cisco Press, 19 Apr. 2004, no. ISBN158705115X, ch. 49, pp. 49–1 – 49–32.

[14] S. Blake, D. Black, M. Carlson, E. Davies, Z. Wang, and W. Weiss, "An Architecture for Differentiated Services – RFC2475," IETF RFC, Dec. 1998. [Online]. Available: http://www.ietf.org/rfc.html

[15] S. Deering and R. Hinden, "Internet Protocol, Version 6 (IPv6) Specification – RFC2460," IETF RFC, Dec. 1998. [Online]. Available: http://www.ietf.org/rfc.html

[16] Fabien Castanier, Alberto Ferrante, and Vincenzo Piuri, "A Packet Scheduling Algorithm for IPSec Multi-Accelerator Based Systems," in *ASAP 2004*, IEEE Computer Society Press, Ed., Galveston (TX), USA, Sept. 2004, pp. 387–397.

[17] J. Carpenter, S. Funk, P. Holman, A. Srinivasan, J. Anderson, and S. Baruah, "A Categorization of Real-Time Multiprocessor Scheduling Problems and Algorithms," in *Hanbook of Scheduling: Algorithms, Models, and Performance Analysis*, Joseph Y. Leung, Ed. CRC Press, 2004, ch. 31.

[18] R. Rajaraman and S. Muthukrishnan, "An Adversarial Model for Distributed Dynamic Load Balancing," in *the 10th Annual ACM Symposium on Parallel Algorithms and Architectures*, June 1998, pp. 47–54.

[19] (2002) PCI comparison, 32 vs. 64-bit and 33MHz vs. 66MHz. [Online]. Available: http://www.buildorbuy.org/pdf/64bitpci.pdf

[20] S. Frankel, R. Glenn, and S. Kelly, "The AES-CBC Cipher Algorithm and Its Use with IPsec - RFC 3602," IETF RFC, Sept. 2003.

[21] T. Dierks and C. Allen, "The TLS Protocol Version 1.0 – RFC 2246," IETF RFC, Jan. 1999. [Online]. Available: http://www.ietf.org/rfc.html

[22] "SystemC Official Website." [Online]. Available: http:/www.systemc.org/
[23] (2000) The Internet Traffic Archive. [Online]. Available: http://ita.ee.lbl.gov/
[24] TCPDUMP Public Repository. [Online]. Available: http://www.tcpdump.org/
[25] Srihari Makineni and Ravi Iyer, "Architectural Characterization of TCP/IP Packet Processing on the Pentium M Microprocessor," in *Tenth International Symposium on High-Performance Computer Architecture*, Feb. 2004, pp. 152–162.