Analytical Performance Analysis of Network-Processor-Based Application Designs

Jie Lu
BMC Software Inc.
Waltham, MA

Jie Wang
University of Massachusetts
Lowell, MA

***Abstract*—Network processors (NP) are designed to provide both performance and flexibility through parallel and programmable architecture, making them superior to general-purpose processors on performance and to hardware-based solutions on flexibility. But NPs also introduce new challenges. It is important to study the limitations of NP architectures so that one can take full advantage of NP resources to achieve the required performance for a given application. It is therefore desirable to develop a general framework for analyzing performance of NP-based applications. This paper presents an analytical method for solving this problem. In particular, we devise a queuing network to model NP resources and application work flows. We then use queuing theory and operational analysis to obtain performance metrics on throughput and response time, among other things, at the component level as well as at the system level. We apply our performance model to SpliceNP, a TCP Splicing implementation of content-aware switches on network processors presented in [10], and show that the analytical results using our models match the experimental results from actual implementation.**

performance model; network processor

I. INTRODUCTION

A network processor is a programmable packet processing device that combines the advantages of low cost and flexibility of a RISC processor and scalability of custom silicon (i.e., ASIC chips) [2]. Specifically designed to store, process, and forward large volumes of data packets at wire speed through parallel and pipelining architectures, NPs are desirable building blocks for constructing network systems that can process data packets of any form. They do so through software, providing a flexible platform for implementing different network applications without the need to make new hardware. Moreover, software modules can easily be reused. Thus, NPs allows users to create and add, through software, the latest and best network services, and in the same time reduce development cost and provide quick-time-to-market products.

Programming NPs is challenging. This challenge is on top of the general issues that all software developing would face. For example, since several choices of designs for solving the same problem often exist, how do we know which design of data flow architecture would provide the best performance? To answer this question we would need to obtain quantitative analysis results.

Packet processors do not have operating systems. This means that software designers need to explicitly allocate NP resources when designing NP-based applications, including processor cycles, threads and memory units. Take memory units as an example, it is obvious that data structures that are accessed infrequently, such as packet payloads, should be placed in DRAM, while data structures that are accessed frequently, such as lookup tables, should be placed in SRAM. Some NPs, however, have multiple channels for the same memory unit, making it difficult to determine how to allocate an individual data structure to which memory channel to obtain the best performance. The decision would depend on the accessing pattern in the particular application we are trying to solve. Other functional units in NPs have the similar problem.

Most network applications have specific performance requirements, including requirements on throughput and delay latency. How can one know whether the target NP would meet the requirements? If not, would an alternate NP, or an array of NPs, provide the required performance?

These questions call for performance analysis tools that provide quantitative results. Bounding calculations and discrete event simulations are two common analysis methods. Bounding calculations, also called "back of the envelope" calculations, are often used to quickly assess the maximum throughput of a single system component. But this method has several serious limitations. For example, bounding calculations tend to yield optimistic predictions that are unrealistic, for the predicted performance would likely decrease when more details about the system are taken into consideration. For another example, an NP has multiple components, each of which has requests queued up waiting for service. Simple bound calculations cannot predict latency, nor can they model any interaction between components.

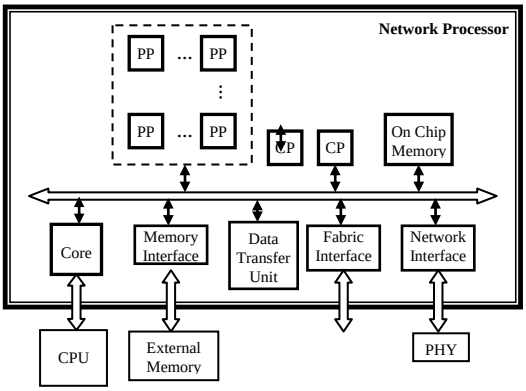
Discrete event simulations use a global time and an event scheduler to measure performance of a design. Transitions are represented by different objects of an event. All events have associated timestamps. The simulation program executes the events one at a time in the order of increasing timestamps. The global time jumps from one event timestamp to the next. In addition to simulating the logic of the system being modeled, events have to update the counters used for statistics, providing detailed performance characteristics of the modeled system. Most NP vendors provide development tool sets, including a simulator, which can be used to provide more accurate performance analysis of an application. But most of the simulators can only perform the performance analysis after the application programs are implemented. We note that it is often

impossible to implement every possible design to choose the best one. Thus, much time and effort will be wasted at this stage if it turns out that the chosen architectural design does not meet the performance requirements. We also note that simulators are designed for a specific NP model or an NP product line. To the best of our knowledge there has been no tool that allows users to compare (even if just roughly compare) the performance across different types of NPs.

To overcome the limitations in bounding calculations and discrete event simulations, we introduce in this paper an analytical method that provides a general framework for analyzing NP-based applications without implementing them.

The rest of the paper is structured as follows. In Section II we describe the base-line architecture common in any type of network processors. In Section III, we construct a queuing-network for modeling NP computations. In Section IV, we compare analytical results obtained from our theoretical models with an actual implementation of a TCP splicing content-aware switch called SliceNP. We show that our analytical results are consistent with the actual performance analysis.

Figure 1. General architecture of network processor



II. General architecture of network processor

No industry consensus exists at this point regarding what hardware components should be included in a network processor and how they should be organized on a chip. NP architectures from different vendors vary considerably, but they share the same base-line concepts and structures. In general, a typical NP consists of an array of programmable packet processors (PP) in a highly parallel architecture, a programmable control processor (a.k.a. core processor), hardware coprocessors (CP) or accelerators for common networking operations, high-speed memory interfaces, and high-speed network interfaces. Figure 1 shows the general NP architecture of a network processor.

Packet processors

Packet processors are RISC-based processors, with the advantage of being small, fast, inexpensive, easy to integrate with other hardware, and easy to program. PPs perform data-plane tasks and provide fast-path data processing at wire speed. Most packets are processed by PPs. PPs use an instruction set that is optimized for packet processing.

Memory I/O latencies affect performance a great deal. To hide memory latencies most PPs employ multi-threading technology on hardware to process multiple packets on a single PP concurrently. It minimizes the overhead of context switching, thus significantly increasing the overall throughput.

Data-plane tasks include packet classification, forwarding, filtering, header manipulating, protocol conversion and policing. Most processing in network applications occurs in data planes.

Control processor

The control processor is a general-purpose processor that runs an embedded operating system. The control processor provides overall control, performs configuration management, and processes exception packets. Exception packets could be control-plane-related, or data-plane-related that may require extra processing such as IP packets with options.

Coprocessors

The coprocessors are special-purpose hardware, providing specific functions for carrying out common network tasks, including pattern matching, table lookup, buffer management, queue management, hashing, checksum computation, and encryption/decryption. Since these functions are commonly used in packet processing regardless which protocols are used, implementing them via hardware speeds up execution. Coprocessors can be used to simplify software creation, for they provide a single-instruction access to complex operations.

Network and fabric interfaces

The fabric interfaces handle interaction between processors and fabric switch, and network interfaces handle interaction between processors and the physical layer of the external network. Most network processors also include data transfer units that are responsible for moving packets between MAC devices and memory directly.

Memory

High speed memory is expensive. Regular computer systems often use different types of memories in a hierarchical manner to balance between cost and speed. For example, an on-chip level 1 cache has the fastest speed, but with the smallest capacity (i.e., the number of bytes it can store). Level 2 and level 3 caches each provide lower speed with larger capacity than the previous level. The main memory has the largest capacity but with the lowest speed. To achieve good performance, data that are more frequently accessed are stored in faster memories.

NPs adopt a similar memory hierarchy. Since NPs are used to process a large volume of network packet data that demonstrates almost no locality, most NPs do not provide cache to packet processors. Some NPs provide on-chip memory for fast accessing. All NPs provide high-speed memory interface for various levels of external memory, where

the Static RAM (SRAM) provides faster speed and the Dynamic RAM (DRAM) provides large storage with lower accessing speed.

Unlike conventional computer systems, NP programmers need to explicitly choose which memory to store which data. Normally, SRAM is used to store configuration and status information, or packet headers in some cases, which needs to be accessed frequently. DRAM provides large space to buffer the payload data that are less frequently accessed.

A number of chip makers manufacture various types of network processors. The most popular models include AMCC nPcore family [1] and Intel IXP family [5].

III. QUEUING NETWORK MODEL

Any NP-based application would typically go through three pipelining stages: packet receiving, packet processing, and packet transmitting. Packet receiving and packet transmitting can each be implemented on a single packet processor or on multiple packet processors in parallel. Packet processing can be implemented on multiple packet processors in parallel or in pipeline.

We model packet flows on a network processor as a queuing network, illustrated in Figure 2. We then devise a performance model from it.

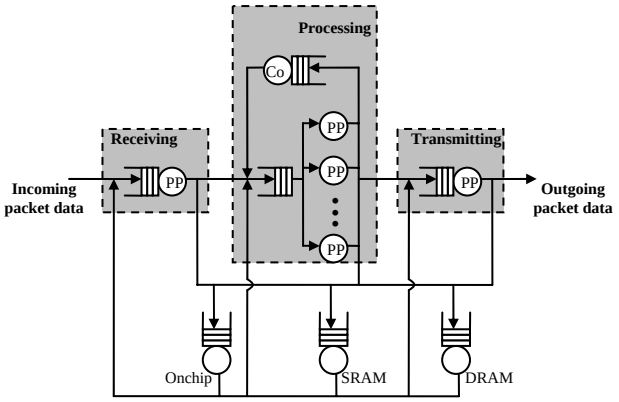


Figure 2. Queuing network model of network processor

In this queuing network model we supply a separate input queue to each packet processor, coprocessor, and memory unit. We assume three levels in the memory hierarchy: on-chip memory, SRAM, and DRAM. Since the control processor is normally used to manage configurations and handle exceptions, rather than processing packets on the fast data path, it has less impact on the overall performance of an application and so we do not include it in the model. We assume that packet receiving and packet transmitting are each handled by a packet processor, and packet processing is handled by multiple packet processors in parallel. Using the multithreading mechanism, packet processors do not need to be held waiting for response from coprocessor, thus the coprocessors are modeled in the same way as packet processors do. There are several ways to allocate resources. For simplicity, we present here a simpler model, which can be extended with easy modifications to meet other configurations.

Critical parameters

We identify the following parameters that are critical for evaluating analytical models for each resource as well as for the entire NP.

- Arrival rate: It is the number of requests or packets that arrive per second.
- Throughput: It is the number of requests or the number of packets completed per second.
- Resource utilization: It is the percentage of time that the resource is busy processing requests.
- Average response time: It is the average time duration that each request or packet spends inside the NP.

The arrival rate can be easily measured externally. It can also be specified explicitly. If the analysis interval is large enough, the system throughput is, according to the flow equilibrium principle, the same as the system arrival rate.

In a conventional computer system, the operating system measures accurately the utilization of resources, such as utilization of processors the utilization of memories. Naturally, NP-based applications should always try to achieve optimal performance and eliminate unnecessary overheads. There is usually no measurement on packet processor utilizations. While the measurement of queue length at each packet processor is relatively easy to obtain, we note that frequent sampling would cause significant negative performance impact. On the other hand, coarse sampling would cause big distortion. Thus, we use the service demand method. Service demands of each resource may be calculated via pseudo code analysis.

We first measure service rate, throughput, and response time at the component level using queuing network analysis. We then use these measures to measure throughput and response time at the system level by treating the whole system as a black box.

Component level modeling

The queuing network model shown in Figure 2 is a closed model. We use it to analyze performance at the component level. There are a fixed number of requests in the system.

Mean Value Analysis

We choose the Mean Value Analysis (MVA) to solve closed queuing network model. MVA is intuitive and is widely used.

The detailed description and derivation of the MVA algorithm can be found, e.g., in [7]. The algorithm can be simplified as a procedure of recursively applying three equations: the residence-time equation, the throughput equation, and the queue length equation, as shown in Equation (3 – 1) to (3 – 3).

- Residence time equation:

$$R_i'(n) = \begin{cases} D_i & \textit{delay resource} \\ D_i\left[1 + Q_i(n-1)\right] & \textit{queuing resource} \end{cases} \qquad (3-1)$$

- Throughput equation:

$$X_0(n) = \frac{n}{\sum_{i=1}^{K} R_i'(n)} \qquad (3-2)$$

- Queue length equation:

$$Q_i(n) = X_0(n) \times R_i'(n) \qquad (3-3)$$

Here *i* denotes the index of the resource, *K* the total number of resources, and *n* the total number of requests reside in the queuing network. We assume that the service demand at each resource, D_i, is known by deriving from pseudo code analysis.

Residence time is the total amount of time that a request stays in a resource. It is the sum of the service time and the queuing time that the request waits for service. The residence time at a delay resource is the same as the service time, since there is no waiting queue at delay resources. The residence time equations for the queuing resources means that the time a request spends waiting in the queue is the accumulated service time of all requests in front of it in the queue.

The throughput equation is derived from Little's Law (see, e.g., [3]). The end-to-end response time of a request going through the queuing network is the summation of the time it spends at each resource, which is the residence time.

The average queue length at resource *i* when there are *n* requests in the system, $Q_i(n)$, is the average number of requests at the resource. Thus, the queue length equation can be derived from Little's Law and the Forced Flow Law [3].

Throughput bound

To make the MVA algorithm converge, we need to find the bound of the system throughput. The maximum throughput of a system is determined by the bottlenecked device.

According to the Service Demand Law [3], we have $D_i=U_i/X_0$, where U_i is the utilization of resource *i*, and X_0 is the throughput of the entire system. Since the utilization of any resource can never exceed 100%, we have $X_0 \le 1/D_i$ for any resource *i*. Then we have,

$$X_0 \le \frac{1}{\max_{i=1}^{K} D_i} \qquad (3-4)$$

Intuitively, the maximum throughput the system can ever achieve is bounded by the resource with largest service demand. Therefore the resource with the largest service demand is the bottlenecked resource in the queuing network.

Parallel processor modeling

In the queuing network model shown in Figure 2, there are multiple packet processors handling data in parallel in the packet processing stage. In this case, there are multiple resources serving requests from a single queue. The basic operational analysis equations does not account for this case.

Seidmann [9] proposed an approximation method for analyzing parallel server stations with a single server under medium to heavy utilizations. The idea is to convert the configuration of parallel servers into the configuration of serialized servers. Assume that there are *m* parallel resources taking requests from a single queue, and the service demand on each individual resource is *D*. Then all *m* resources can be replaced with a single resource that is *m* times faster than each individual's original resource. Therefore, the service demand on this new resource is *D/m*. In this way, the waiting time on the queue is close to the waiting time in the system with *m* parallel resources. A second resource is added in tandem to rectify the mean-time-in-station estimates by assuring that the total service remains the same. Hence, the second resource can be viewed as a delay resource without waiting queues. The service demand on the delay resource is *D*(m-1)/m*.

Requests under light load spend no waiting time in the queue. Thus, the average response time for a request in the new configuration is *D/m + D*(m-1)/m = D*. This matches the average response time for a request in the original parallel configuration.

Resources under heavy load are busy most of the time. The dominant part of the average response time is the waiting time spent in the queue. The time delay from the delay resource becomes negligible. The average waiting time in the queue of the single resource is the same as the average waiting time in the queue of the *m* parallel resources.

Using this approximation, the parallel packet processors in Figure 2 are replaced with a single queuing resource plus a delay resource, as illustrated in Figure 3.

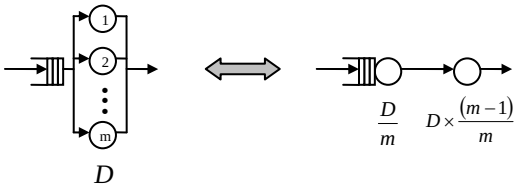


Figure 3. Approximate modeling on parallel processors

System level modeling

From the viewpoint of an outside observer, the whole queuing network in Figure 2 can be treated as a black box. It takes requests, or packets, one by one from its input queue and completes them as output. This black box presents no difference from a queuing resource discussed before. It is therefore a perfect example of an open queuing network model. Unlike most other resources, the service rate varies depending on the number of requests in the system. Thus the service rate it can be treated as a load-dependent resource.

Using analysis at the component level, we may obtain an array of throughput based on different numbers of requests, *k*, in the system. Then the variable service rate can be derived from them as below.

$$\mu_k = \begin{cases} X(k) & k < J \\ X(J) & k \ge J \end{cases} \qquad (3-5)$$

Here J is the number of requests in the system when it reaches the maximum throughput. It can be found using the MVA algorithm.

Assume that the requests arrive at a constant rate, λ, and the input queue of the network processor system is unbounded. The equilibrium probability [8] can be rewritten as follows:

$$p_0 = \frac{1}{1+\sum_{k=1}^{J-1}\frac{\lambda^k}{\beta(k)}+\frac{\rho\lambda^J}{\beta(J-1)(1-\rho)}} \quad (3-6)$$

$$p_k = \begin{cases} p_0\dfrac{\lambda^k}{\beta(k)} & k < J \\ p_0\dfrac{X(J)^J\rho^k}{\beta(J)} & k \geq J \end{cases} \quad (3-7)$$

Here $\beta(k) = X(1)\times X(2)\times\cdots\times X(k)$ and $\rho = \lambda / X(J)$.

Then the average number of requests in the system can be further derived as below.

$$\overline{N} = p_0\left(\sum_{k=0}^{J-1}\frac{k\lambda^k}{\beta(k)}+\frac{\rho\lambda^J(1+J-J\rho)}{\beta(J)(1-\rho)^2}\right) \quad (3-8)$$

In the open queuing network model, the arrival rate has to be less than the maximum throughput, that is, $\lambda < X(J)$. Otherwise, the waiting queue may grow to infinity. So does the average response time. According to the equilibrium principle of open queuing network, the throughput is equal to the arrival rate, $X = \lambda$. Applying Little's Law we may obtain the average response time as follows:

$$R = \frac{\overline{N}}{X} = \frac{\overline{N}}{\lambda} \quad (3-9)$$

Bounded input queue length

The assumption of unbounded queue length in previous section is unrealistic in a real network system, for the size of the input buffer of any system is bounded. Once the input buffer is full, any new incoming packets will be dropped.

In many cases, the network system utilization is not too high, or the average service rate is much greater than the average arrival rate. Then the waiting queue would not grow too long. If the input buffer is reasonably large, dropping packets is not an issue. The analysis model with unbounded queue length is sufficient for modeling such systems.

In designing NP-based applications, it is important to find a good balance between the input buffer size and the packet dropping rate. If certain packet dropping rate is acceptable, it would be reasonable to limit the queue length for achieving better average response time.

For network systems that expect to reach the queue length bound, the analysis model with unbounded queue length is not sufficient. The queue length bound has to be factored in.

Bounded queue length means that there are a bounded number of states in the state transition diagram. Suppose K is the maximum number of requests allowed in the system, the variable service rate in equations (3 – 5), (3 – 6) and (3 – 7) can be rewritten separately for when $K \geq J$ and when $K < J$.

When $K < J$,

$$\mu_k = X(k) \qquad k = 1,\cdots,K \quad (3-10)$$

$$p_0 = \frac{1}{1+\sum_{k=1}^{K}\frac{\lambda^k}{\beta(k)}} \quad (3-11)$$

$$p_k = p_0\frac{\lambda^k}{\beta(k)} \quad (3-12)$$

When $K \geq J$,

$$\mu_k = \begin{cases} X(k) & k = 1,\cdots,J-1 \\ X(J) & k = J,\cdots,K \end{cases} \quad (3-13)$$

$$p_0 = \frac{1}{1+\sum_{k=1}^{J-1}\frac{\lambda^k}{\beta(k)}+\frac{\lambda^J\left(1-\rho^{K-J+1}\right)}{\beta(J-1)(1-\rho)}} \quad (3-14)$$

$$p_k = \begin{cases} p_0\dfrac{\lambda^k}{\beta(k)} & k = 1,\cdots,J-1 \\ p_0\dfrac{X(J)^J\rho^k}{\beta(J)} & k = J,\cdots K \end{cases} \quad (3-15)$$

Then, the average number of requests in the system can be derived by applying the following equation:

$$\overline{N} = \sum_{k=0}^{K} k\times p_k \quad (3-16)$$

In systems with bounded queue length, the performance modeling concerns not only the throughput and average response time, but also the packet dropping rate. The proportion of the dropped packets is the fraction of time when there are K requests in the system. Based on equations (3 – 12) and (3 – 15), we have

$$p_K = \begin{cases} p_0\dfrac{\lambda^K}{\beta(K)} & K < J \\ p_0\dfrac{X(J)^J\rho^K}{\beta(J)} & K \geq J \end{cases} \quad (3-17)$$

Multiple class workloads

The analysis model described in previous sections deals with a single workload class only. Unlike a standard computer system, a network processor is dedicated to a specific application in many cases. There is only one workload running on it. The single class analysis model is sufficient for such systems. However, the analysis model can be easily extended to handle multiple workload classes, for the situations that multiple applications run on a single NP. The detail analysis for multiple-class workload is omitted in this paper.

For simplicity, our analysis model currently considers only the simple case of NP being a "multiplexer". We leave the extension to "concentrator" function to future work.

IV. APPLICATION ANALYSIS

SpliceNP

To validate our analytical model we apply it to actual NP-based applications. In particular, we choose SpliceNP [10] as an example. SpliceNP implements TCP Splicing for a content aware switch using an Intel IXP2400 network processor. It processes data using four components: packet receiving, packet transmitting, processing of packet from client, and processing of packet from server. Each component is assigned with a dedicated packet processor called a microengine (ME) in the IXP technology.

When the switch receives a connection start request (SYN) from the client, it establishes a TCP connection with the client using the handshake protocol. Once receiving the HTTP request, it parses the request and matches it with preset policy to find the target server. After the target server is identified, it establishes another TCP connection with the server using the handshake protocol again. Then the two connections are spliced together. Unlike most Layer-7 switches, SpliceNP creates a brand new TCP connection with target server for each connection from the client. Table 1 summarizes the packet interaction sequence for each HTTP request.

From client	From server	To client	To server
SYN		SYN/ACK	
ACK/Request		ACK	SYN
	SYN/ACK	ACK/Request	
	Response	Response	
ACK			ACK
	…	…	
…			…
FIN			FIN
	FIN/ACK	FIN/ACK	
ACK			ACK

Table 1. Packet sequence in switch for each HTTP request

The shaded entries are packet interactions after the two TCP connections are spliced. If the request file size is larger than the MTU, the file is broken into multiple packets, and so there are multiple pairs of response and ACK packets for that file.

Analysis method

For a Layer-7 switch, it is easy to obtain the request arrival rate and the average packet size. But it is difficult to measure the average service demand for each request or the number of packets in the queue at each resource. In our study we estimate service demand by analyzing the pseudo code of each application. MEs in IXP2400 are RISC processors, and so most instructions only take one clock cycle. Thus, service demand for an ME can be obtained based on the number of instructions for processing each request. For memory access, the service demand can be obtained based on the average access latency and the number of memory reference made for each request.

$$D_{ME} = \frac{instruction\_count}{clock\_rate} \quad (4-1)$$

$$D_{SRAM} = \frac{reference\_count \times SRAM\_latency}{clock\_rate} \quad (4-2)$$

$$D_{DRAM} = \frac{reference\_count \times DRAM\_latency}{clock\_rate} \quad (4-3)$$

$$D_{SHaC} = \frac{reference\_count \times SHaC\_latency}{clock\_rate} \quad (4-4)$$

When using multiple MEs in parallel at the processing stage, the MEs are replaced with two resources in the model shown in Figure 3. Therefore, the service demands for the aggregated resource and delay resource are different from those of a single ME.

$$D_{aME} = \frac{instruction\_count}{clock\_rate \times ME\_count} \quad (4-5)$$

$$D_{dME} = \frac{instruction\_count \times (ME\_count - 1)}{clock\_rate \times ME\_count} \quad (4-6)$$

The related hardware parameters for an IXP2400 network processor are listed in Table 2 [4].

ME Clock Rate (MHz)	ME Threrads	Latency (processor clock cycles)		
		SHaC	SRAM	DRAM
600	8	16	90	120

Table 2. Intel IXP2400 parameters

SHaC is a functional unit providing on-chip memory (called scratchpad), hashing, and control status registers.

Pseudo code analysis

According to the design description of SpliceNP, we derive pseudo code modules and summarize estimated service demands in Table 3.

We obtain estimated service demands for the modules with * from the standard microblocks that come with the Intel IXA SDK.

Not every packet goes through all modules. The modules in the shaded area are only executed for certain specific packet types. All packets are processed by Data Forward after the two connections are spliced.

Based on Tables 1 and 3 we can derive the service demand on each resource for one HTTP request. The number of packets for HTTP request and response vary, depending on the request file size. In our experiment, we assume that the request is small enough to fit into one packet, and the MTU for the response packet is 1500 bytes for Ethernet.

Module	Inst. Cycles	SHaC Ref.	SRAM Ref.	DRAM Ref.
Packet Rx*	86	1	1	1
DL Source*	33	1		1
Ethernet Decap*	20			
IP Validate	100			
Ctrl Block Lookup	64	1	3	
TCP Validate	100			
Client SYN	31		16	
Client ACK/Request	120		72	1

Server SYN/ACK	35		8	
Data Forward	28			
Client FIN	1		2	
Server FIN/ACK	2		8	
Ethernet Encap*	60			
DL Sink*	54	1		1
Packet Tx*	92	1	2	1

Table 3. Pseudo code summary of SpliceNP design

In order to compare the analytical results obtained from our performance model with the actual measured results from SpliceNP implementation, we choose the same set of request file sizes to obtain the maximum throughput. We then convert the throughput results in terms of requests into bytes, based on Table 1. The left chart of Figure 4 shows the results. Our analytical results match the measured results from actual implementations presented in [10], which is shown in the right chart of Figure 4 (The only discrepancy is when request file size is small). This validates our approach.

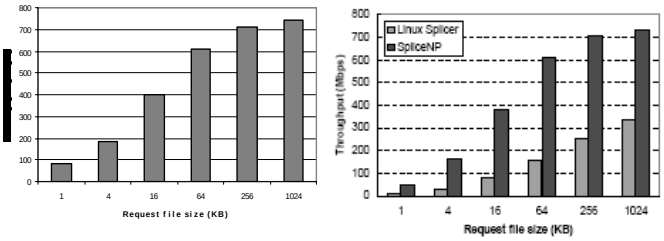


Figure 4. Throughput result comparison

When the switch reaches its maximum throughput, the average response time is often unacceptable. Figure 5 illustrates how the average response time changes according to the throughput. This could provide a guideline to the application designer. In our experiment we fix the request file size to 16KB. The response time reaches to 1.19 second when the throughput reaches the maximum of 22000 requests.

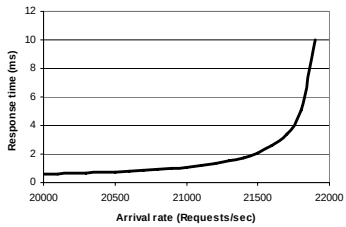


Figure 5. Response time vs. arrival rate

Bounding the input queue length is an easy way to significantly improve the response time at the price of dropping requests. The quantified results will help to determine if it is acceptable. Figure 6 shows that the response time is improved tremendously while the request dropping rate is still kept low.

We observe that adding additional MEs makes no significant performance improvement. This is because the bottleneck of the application is on SRAM access. In order to significantly improve performance, one has to either reduce I/O reference to SRAM or distribute SRAM to multiple channels if they are available.

V. CONCLUSION

In this paper, we propose and validate a general analytical framework for measuring the performance of NP-based application designs. It allows application designers to evaluate the performance of a design with accuracy to determine whether it meets the performance requirement, and thus it saves designers' time and effort from the need of actually implementing the design to obtain performance measurements from simulations.

ACKNOWLEDGEMENT

This work was supported in part by an Intel grant. The second author was also supported by NSF under grant CCF-0429906.

REFERENCES

[1] http://www.amcc.com/products/process.html

[2] D. Comer, "Network Systems Design Using Network Processors," Pearson Prentice Hall, 2004

[3] P. Denning and J. Buzen, "The Operational Analysis of Queueing Network Models," Computing Survey, Volume10, Number 3, September 1978

[4] Intel IXP2400 Network Processor Datasheet, February 2004

[5] http://developer.intel.com/design/network/products/npfamily/

[6] D. Menasce and V. Almeida, "Capacity Planning for Web Services," Prentice Hall PTR, 2002

[7] M. Reiser and S. Lavenburg, "Mean-Value Analysis of Closed Multichain Queuing Networks," Journal of the Association for Computing Machinery, Volume 27, Number 2, April 1980

[8] T. Robertazzi, "Computer Networks and Systems: Queueing Theory and Performance Evaluation," Springer-Verlag, 1990

[9] A. Seidmann, P. Schweitzer and S. Shalev-Oren, "Computerized Closed Queueing Network Models of Flexible Manufacturing Systems," Large Scale Systems, Volume 12, Number 4, 1987

[10] L. Zhao, Y. Luo, L. Bhuyan and R. Iyer, "SpliceNP: A TCP Splicer using A Network Processor," ACM Symposium on Architectures for Network and Communications System, Princeton, NJ, October 2005

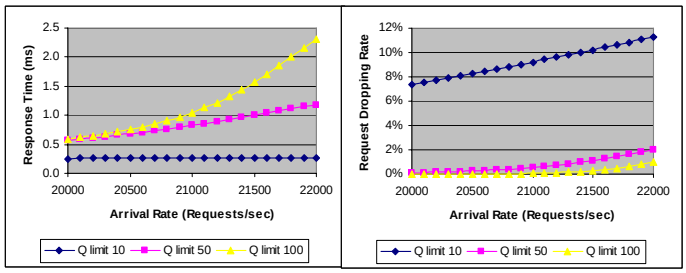


Figure 6. Response time and request dropping rate with limited Q length