Run-Time Adaptive Processor Allocation of Self-Configurable Intel IXP2400 Network Processor

A.Satheesh vbsatheesh@yahoo.com
Department of Computer Science and Engineering
Periyar Maniammai University
Thanjavur-613 403, Tamil Nadu, India

Dr.D.Kumar kumar_durai@yahoo.com
Department of Electronics and Communication Engineering
Periyar Maniammai University
Thanjavur-613 403, Tamil Nadu, India

Dr.A.Vincent Jeyakumar avjeyakumar2004@yahoo.com
Department of Mathematics
Periyar Maniammai University
Thanjavur-613 403, Tamil Nadu, India

Abstract

An ideal Network Processor, that is, a programmable multi-processor device must be capable of offering both the flexibility and speed required for packet processing. But current Network Processor systems generally fall short of the above benchmarks due to traffic fluctuations inherent in packet networks, and the resulting workload variation on individual pipeline stage over a period of time ultimately affects the overall performance of even an otherwise sound system. One potential solution would be to change the code running at these stages so as to adapt to the fluctuations; a near robust system with standing traffic fluctuations is the dynamic adaptive processor, reconfiguring the entire system, which we introduce and study to some extent in this paper. We achieve this by using a crucial decision making model, transferring the binary code to the processor through the SOAP protocol.

Keywords: Network Processor, Reconfiguration, Runtime adaptation, dynamically adapting processor, Active Network, Self-Configurable, SOAP, IXP2400

1. INTRODUCTION

Traditionally most of the network core components have been implemented using Application Specific Integrated Chips (ASICs). We first recapitulate some of the earlier related works. *Kevin Lee and Geoffrey Coulson* in [14] analyse the exact position that runtime reconfiguration occupies in Network Processor (NP), such as dynamically extendable services, network resource management, configurable network based encryption , offload processing etc. Dynamic deployment of resources to different flows in NPs has been known to Kind, Pletka and Waldvogel (see [1]). Implementations of NPs as system-on-a chip multiprocessor, involving multiple multithreaded processing engines and on-and-off chip memory, was the contribution of Tilman

Wolf [18], [19]. However, these devices lack flexibility and speed and also consume more energy. They therefore require replacement of physical components in the network core whenever there is a protocol change or update. These replacements still need some fine tuning which we provide by the use of programmable processors in the reconfigurable environment. These Neo Network Processors are multiprocessor devices designed for the efficient data-plane and control-plane by processing in networking applications. They are programmed to offer the required flexibility in packet processing and at the same time they appropriately provide the necessary computing resources to meet the speed requirement constraints of the network protocols. Intel's IXA architecture provides the basis of a family of such NPs, of which the IXP2400 Network Processor is being used in this paper for the implementation of run-time adaptive processor allocation of self-configurable systems. An adaptive processor allocation to pipeline stages of a packet processing application at run-time can improve robustness of the system to traffic fluctuations, can reduce processor provisioning requirement of the system and can conserve energy.

2. THE SYSTEM ARCHITECTURE

The Intel IXP2400 scores over many other processors due to its high programming flexibility, code reuse, and faster deployment capabilities and many other advantages like supporting a wide variety of LAN and WAN applications. We therefore choose this Network Processor for our study.

2.1 Intel IXP2400 Network Processor

The IXP2400 is an integrated Network Processor, comprised of a single X-Scale Core processor, eight Micro engines, standard memory interfaces, and high-speed bus interfaces. It is targeted at networking applications requiring a high degree of flexibility, programmability, scalability, performance, and low power consumption. The unique architecture of the IXP2400 affords the user a highly concurrent packet processing model, while keeping the programming model simple. This is accomplished by providing many features in hardware that simplify the programming model. It allows the designer to implement the software, what was previously implemented in custom ASICs. This flexible, reprogrammable approach makes development time faster, facilitates easy bug-fixing, adds features to products after deployment in the field while conforming to standards that are not yet finalized. The micro engines are custom processors implemented specifically for networking applications. They are especially well suited to high-speed data manipulation and movement. The micro engines being fully programmable processors are able to examine packet contents at all levels of the networking stack. This makes them suitable not only for layer 2 and 3 switching/forwarding, but also for applications that require deeper inspection and manipulation of packet contents. The key features of IXP2400 NP are scrupulously discussed in [7], [15], [17] whose standard block diagram of Intel IXP2400 NP architecture as shown in Figure.1. This shows the six functional units, which are traditionally known as Intel X-Scale Core [3], Micro-engines [5], control store, contexts, Data path Registers [9], local memory, SRAM and DRAM controller [10] , [11], Media and switch Fabric Interface, PCI controller, X-Scale core Peripheral, Performance monitor [8], Scratchpad memory and Hash unit.

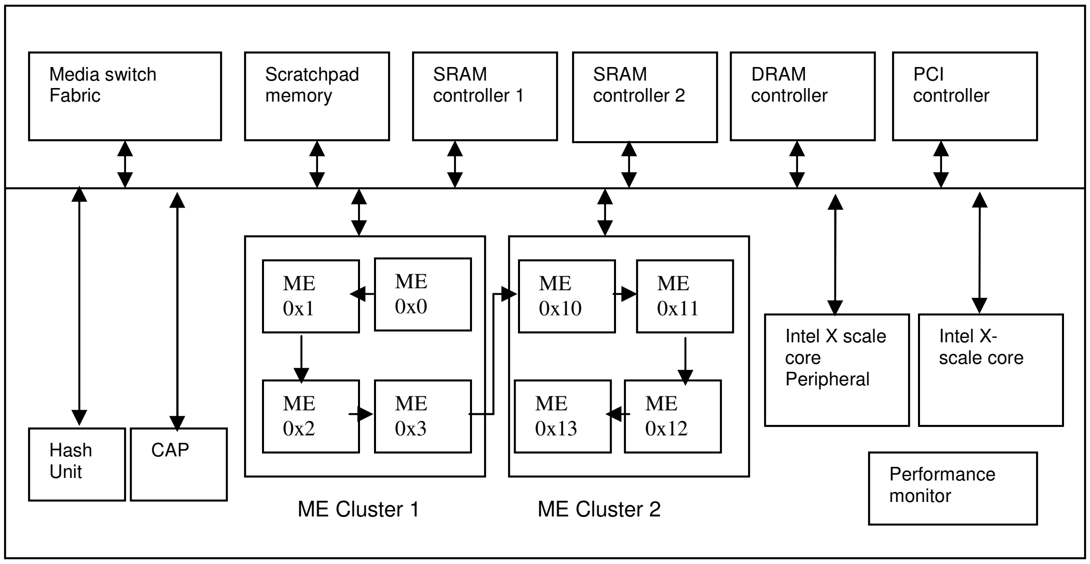


FIGURE 1: Simplified block diagram of the IXP2400

2.2 Workload Model

Network processing in Intel IXA is essentially a series of tasks that are applied to a constant stream of packet or cell data. With the multi-processor/multi-threaded architecture of the IXP 2400 network processor, these tasks are distributed over several micro-engines, each of which is programmed to perform specific tasks. When a micro-engine completes its tasks, it passes the context to the next micro-engine so that it can continue processing the data.

2.3 Performance Metrics

In our analysis, we focus on the following performance metrics, which are key indicators of the performance of a runtime system:

- **Processor utilization (ρ):** This metric is defined as the fraction of time that the processor is busy. Processor utilization indicates the efficiency at which the system operates.
- **Packets in the system K**. This metric indicates the extent to which the queues and processors are utilized in the system. A large value indicates that many packets are queued and that packets experience a large delay when traversing the system.

2.4 Organization of the paper

The remainder of this paper is organized as follows: Section 3 discusses related works. The overall methodology is described in Section 4. In Section 5 we draw the state diagram of the proposed system. In Section 6 we describe the model which we propose to introduce. The implementation details will be discussed in section 7. The results will be analyzed in Section 8. We conclude the paper in section 9.

3. RELATED WORK

Recently, several studies have been initiated in reconfiguration of network processors. We brief a few here. Xin Huang and Tilman Wolf [22],[23] in their work present a methodology for evaluating runtime systems for NPs by defining workload models, queuing discipline and improving existing mapping algorithm . In paper [12] J. Allen et al have used Hifn PowerNp, wherein they discussed the challenges and demands posed by next generation networks and have described how network processors can address these issues by performing highly sophisticated packet processing at line speed. *Kevin Lee and Geoffrey Coulson* [13], [14] in their work demonstrated the importance of the specialized software, to support runtime reconfiguration that exploits the potential of NPs. They have focused mainly on the generic mechanism that can be potentially applied in all areas. They have used Intel IXP 2400 as a representative of the state-of-the-art current generation of NPs. In fact, they discover new approaches that present a runtime component based approach to programming NPs. The approach promotes conceptual uniformity and design portability across a wide variety of NP types while simultaneously exploiting hardware assists that are specific to individual NPs. *Troxel, et al* [6] have demonstrated the superior performance of enhanced NP over baseline NP for prioritized traffic that is non uniform. In the baseline experiments, the ME pipelines were not reconfigurable. This type of system mimics the behavior of today's NPs. *Ravi kokku, et al* [16], [17] have presented a new approach of delay-conscious processor allocation algorithm (PAL) for packet processing systems. And they analyzed the benefits and challenges of adapting allocations of processors to packet types in the above systems and also they demonstrated that, for all the applications and traces considered, run-time adaptation can reduce energy consumption and processor provisioning level. On the other hand, *Vinod Balakrishnan, et al* [21] concentrate on balancing two requirements in packet-processing applications on multi-core processors. *Arun Raghunath, et al* [2] present yet another approach to support network processor platforms, which are increasingly required to support a rich set of services. These multi-service systems are also subjected to widely varying and unpredictable traffic. According to them, current network processor systems do not simultaneously deal well with a variety of services and fluctuating workloads. They have implemented an adaptive system that automatically changes the mapping of services to processors, and handles migration of services between different processor core types to match the current workload.

4. THE INTEL IXP2400 NP CONFIGURABLE ENVIRONMENT

This section gives the complete high level design details of the Intel IXP2400 NP that is needed in our work, by developing a self-configurable environment that would dynamically reconfigure its resources based on the results of monitoring traffic flows. For this, flow statistic is gathered by runtime-mapping-technique. In contrast, the unused hardware resources such as micro-engines are also quantified. These statistics will be used for choosing the appropriate resources for the services being offered, i.e., dynamic deployment reconfiguration.

The proposed system and its architecture are depicted in Figure.2. This will provide the resources based on the network traffic for the purpose of reconfiguration. The network traffic is analyzed using the monitoring module. The monitoring module will scrutinize the number of packets coming in and getting out of packet processing system, using a counter. Based on the number of packets, the arrival rate and departure rate of the packets can be determined.

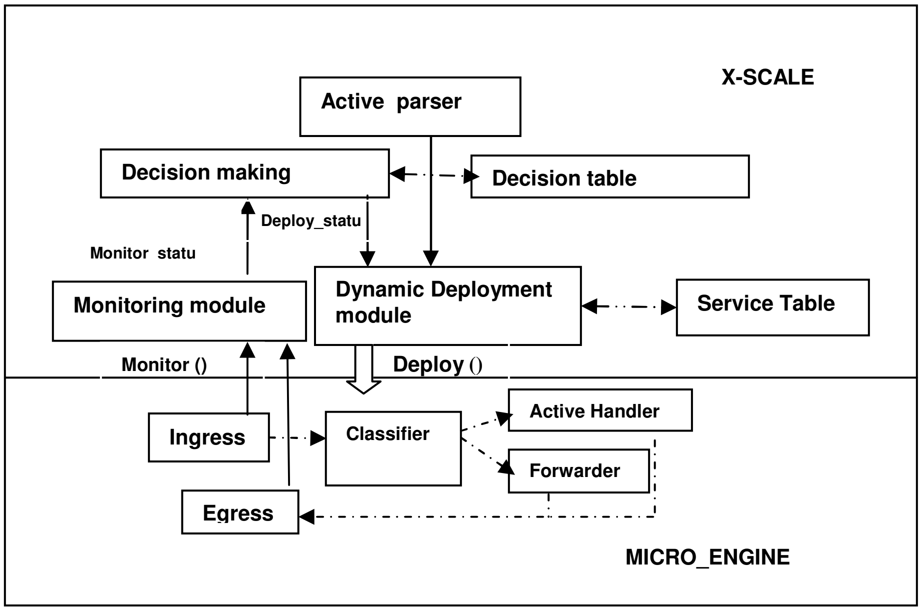


FIGURE 2: Functional block diagram of NP reconfiguration

The result of this monitoring module will be used for decision making wherein the need for extra processor is decided. This is done using the decision table and service table (see Table.1 and Table 2) to be explained in section 4.2.

We list below the salient functions of the above reconfigurable diagram,
1. *Monitoring module* monitors the incoming and outgoing packets.
2. *Decision-making module* decides whether to deploy the code or to stop the Micro-engine.
3. *Dynamic deployment module* dynamically deploys the code in the Micro-engine.
4. *Active code transfer module* parses the SOAP packet that comes from administrator. This packet contains the binary file for deployment.

4.1 Monitoring Module

The counter is set at the micro-engine level. The counters at ingress will keep track of the incoming packets to the queue. And the last counter, which is at egress, will keep track of the outgoing packets from the packet processing system that is to be processed by the egress. These two counters help to determine the number of packets in the intermediate stage. So, with the help of Arrival and Departure rate from and to the packet processing system, the traffic intensity is determined as shown in Figure 3. Since there is need for accessing the counter by both the processors, the counter values must be maintained in common to both the X-scale processor and Micro-engine for easy access to it.

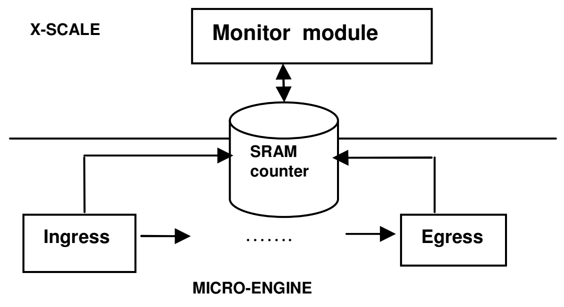


FIGURE 3: Role of X scale and ME in the Monitoring Process

In this module the following two functions are performed,
1: Counting of incoming packets to the ingress and
2: Counting of outgoing packets from the egress

We calculate the service rate in this module using the following equation (1),

$$\text{Overall service rate} \quad X(t) = N/t, \tag{1}$$

where N = Number of packets (packet count) transmitted
t = Time interval in seconds.

We also generalize these ideas to k server by iteration process as shown below.

At the initial level, we are assuming the traffic level is low. In this situation the system is used in single micro-engine for packet processing. So the service rate is

$$X_1(t) = \frac{E}{t}. \tag{2}$$

We are assuming in the second level that the system is in moderate traffic, so that it can be provisioned by additional resources. The service rate of the second processor is,

$$X_2(t) = \frac{E - X_1(t)}{t}. \tag{3}$$

And when the system is in heavy traffic, more number of processors can be provisioned. The service rate of additional resource is,

$$X_3(t) = \frac{E - [(X_1(t) + X_2(t)]}{t}. \tag{4}$$

We can generalize the service rate of the j th server as

$$X_j(t) = \frac{E - \sum_{i=1}^{j-1}[X_1(t) + X_2(t) + \dots + X_{j-1}(t)]}{t} \qquad (1 \le j \le k). \tag{5}$$

where,

E = total count of egress value.

4.2 Decision making module

Before we go into the various parameters of this module, we must explain the most important aspect of this module, which is perhaps the most important concept of this paper itself. The basic idea is to optimize the resources and avoid the packet loss when the activated micro-engines are at full throttle. We fix two parameters T_{q_1} and T_{q_2}, for the queue lengths where $T_{q_1} < T_{q_2}$.

The decision making module decides whether to dynamically deploy the code in a Micro-engine or relieve the load on a currently utilized Micro-engine, based on the packet arrival rate (λ) and the lagging packets in the intermediate queue (k). When the arrival rate is less, we call T_{q_1} the minimum threshold (T_{min}) and when the arrival rate is at its peak, we call T_{q_2} the maximum threshold (T_{max}) and these threshold values are fixed based on the network trace obtained by monitoring module. This module employs the decision table (Table. 1) and the service table (Table. 2) for dynamic deployment.

Traffic	Threshold	Processor
Low	$< T_{q_1}$	P1
Medium	$T_{q_1} \leq M < T_{q_2}$	P2
High	$\geq T_{q_2}$	P3

TABLE 1: Decision Table

Processor	Process	Flag	Service
ME0	SRAM1.uof	8107	Ingress
ME1	x.uof	8107	Dynamic deployment
ME2	SRAM2.uof	8107	Egress
ME3	x.uof	8107	Dynamic deployment
ME4	-	8106	inactive
ME5	-	8106	inactive
ME6	-	8106	inactive
ME7	-	8106	inactive

TABLE 2: An example of Service and Resource table (Moderate Traffic)

where,

Flag value 8106 and 8107 indicate that the micro-engine is inactive and active,
Service – is the functionality of the instance.

The pseudo code for the rules adopted by the decision-making module is given below Algorithm.1 and 2.

Algorithm: 1. ***Allocation Rule***

```
1. int i ← $T_{min}$
2. int j ← $T_{max}$
3. int q1 ← $T_{q_1}$
4. int q2 ← $T_{q_2}$
5. if ( $\lambda$ > i) and (k >q1) then
6.         deploy_code_for_moderate_traffic
7. if ( $\lambda$> j) and (k >q2) then
8.         deploy_code_for_maximum_traffic
```

Algorithm: 2. ***Deallocation Rules***

```
1. int i ← $T_{min}$
2. int j ← $T_{max}$
3. if  ( $\lambda$ < i) and current_code == moderate_traffic then
4.                 stop_code_for_moderate_traffic
5. if ( $\lambda$< j) and current_code == maximum_traffic then
6.                 stop_code_for_maximum_traffic
```

In case the traffic is low, the threshold value T_{q_1} = T_{min} will be fixed in such a way that the arrival rate is less than $T_{min.}$, the system will activate the first micro-engine S1 (see State diagram S1). T_{q_1} is fixed in such way that arrival rate is above 70% of the service rate of the micro-engine. Incidentally our blanket assumption is that all micro-engines in the system have equal capacity of service rate.

The application code for the moderate traffic occupies four micro-engines (moderate resource provisioned) while that for heavy traffic uses all available resources in order to tolerate the maximum traffic.

4.3 Active code Transfer module

In Figure.4, shows the working principle of Active code transfer module. All the packet processing would be carried out in data plane (micro-engine level).Hence when the packet comes in; the packet is classified based on the destination IP and Port by classifier. The active code will be passed on to the active code handler and the other traffic to normal packet handler. The active code handler will check for more flag bit set and fragmentation ID, then extract the packet content and copy an image to SRAM. Later the X-scale will monitor the SRAM for active code and get the content to form a binary file.

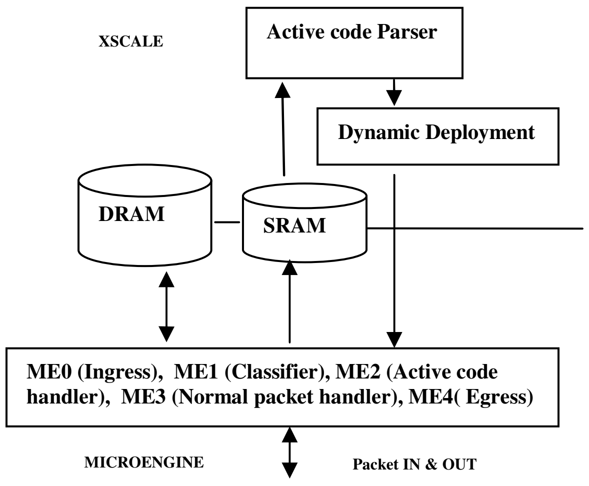

XSCALE
Active code Parser
Dynamic Deployment
DRAM
SRAM
ME0 (Ingress), ME1 (Classifier), ME2 (Active code handler), ME3 (Normal packet handler), ME4(Egress)
MICROENGINE
Packet IN & OUT

FIGURE 4: Functional Block Diagram

The algorithm for identifying and extracting the active packet from the rest of the traffic is shown below,

Algorithm : 3. *Identifying and Extracting the Active packet*

```
1.  If (Destination_IP = NP_IP && Destination_port=port)
    //Classifier
    {
            //Active code Handler
2.  Check for More Flag bit set;
3.          Extract Fragmentation ID;
            Extract Total length of the packet;
4.  Set the flag in SRAM with the offset of packet content;
5.          Move the packet content from DRAM to SRAM;
6.  Next SRAM_LOC;
    }
```

4.4 Active code Parser module

The XML Code which comes by SOAP as active code will be parsed in this module. SRAM location is monitored in order to check whether the flag is set. If the flag is set, from the offset detail, the packet content is parsed. The header and the binary image are separated from the packet content. The Micro-engine number and file name of the active code can be identified by parsing the content header between the XML tag **<ME></ME>** and <**IMAGE></IMAGE>.b** The Binary UOF file will be extracted by parsing the active code between **–MIME-BOUNDARY**. The Figure 5. shows the X-Scale module function.

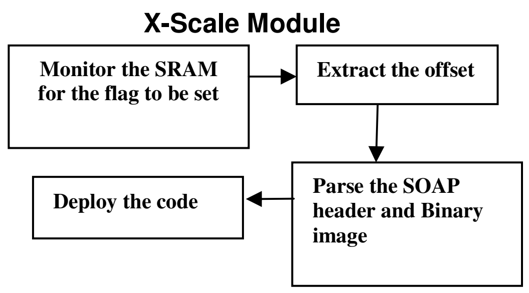

X-Scale Module
Monitor the SRAM for the flag to be set
Extract the offset
Parse the SOAP header and Binary image
Deploy the code

FIGURE 5: X Scale module

4.5 The SOAP Format for Active code

Simple Object Access Protocol (SOAP) is a way to structure data so that any computer program in any language can read SOAP and send messages in SOAP. SOAP provides the answer to two main requirements. More precisely, XML provides a standard way to represent data, and SOAP provides an extensible message format ("extensible" essentially means you can make up your own tags; for example, HTML is not extensible). So this helps to create an active packet with information of the code by extensible message format (see Figure.6), where the binary file can be carried by the attachment part.

```
--MIME_boundary
Content-Type: text/xml; charset=UTF-8
                     Content-Transfer-Encoding: 8bit
Content-ID: networkprocessor.xml@annauniv.com

<?xml version='1.0'?>
<SOAP-ENV:Envelope xmlns:SOAP-
ENV="http://schemas.xmlsoap.org/soap/envelope/">
                           <SOAP-ENV:Body>
<ME>
<MENO>17</MENO>
<IMAGE>packet5</IMAGE>
 <theAttachment href="cid:packet5.uof@annauniv.com"/>
</ME>
</SOAP-ENV: Body>
</SOAP-ENV: Envelope>

--MIME_boundary
Content-Type: text/plain
Content-Transfer-Encoding:binary
Content-ID: packet5.uof@annauniv.com
Binary Code……………………………………….
--MIME_boundary
```

FIGURE 6: Format for Active packet

5. THE STATE-DIAGRAM

At this stage, we introduce queuing theory as the primary systematic network analysis network delay. The conventional method of taking transmission delay L/C has been improved in the present day contexts to Little's Law N = λT and its infinitesimal modification $N_t = \lambda_t T_t$, which we follow in our situation (for details see [4]).

Let,

N(t) = Number of packets in the queue at time t

α(t) = Number of arrival in the interval (0,t)

β(t) = Number of departure in the interval (0,t)

So that,

$$N(t) = \alpha(t) - \beta(t) \tag{6}$$

Let t_i and T_i be the time of arrival and the time spent in the system respectively, by the i^{th} customer. Using elementary integration theory as a limit of summation, we obtained the following identity:

$$\int_0^t N(\tau)d\tau = \sum_{i=1}^{\beta(t)} T_i + \sum_{i=\beta(t)+1}^{\alpha(t)} (t - t_i) \tag{7}$$

Dividing throughout by t , we obtained the modified Little's Law

$$N_t = \lambda_t T_t ,$$

where,

$N_t = \frac{\int_0^t N(\tau)d\tau}{t}$ = Time average of the number of customers in the system in the interval (0,t)

$\lambda_t = \frac{\alpha(t)}{t}$ = Time average of the customer arrival rate in the interval (0,t).

$T_t = \frac{\sum_{i=1}^{\beta(t)} T_i + \sum_{i=\beta(t)+1}^{\alpha(t)} (t-t_i)}{\alpha(t)}$ = Time average of the time a customer spends in the system in the interval (0,t)

We explain the situation in the state-diagram Figure.7. Here $S_1, S_2, \ldots, S_c$ denote the active servers and S_{c+1} and S_{c+2} denote respectively the ingress and egress. Since, the total number of servers = k, we have k = c + 2. Suppose, we use minimum resource (less number of micro-engines) with high traffic flow, we incur packets loss and this can be represented by dotted lines in the state diagram. The T_{q_1} and T_{q_2} are the position of minimum and maximum of queue length. The packets arrive in *Poisson* distribution and service of each server is exponential. The mean arrival rate is λ and mean service rate is $\frac{1}{\mu}$. The incoming packets are distributed from the dispatcher in FCFS discipline. The queuing model is *M/M/1/* or in *M/M/c* as a generalized version.

It is surprising to note that the dotted line representing the 'critical line' where the packet loss occurs (before our innovation of resource optimization) resembles the 'critical line' ***Res = ½*** , where the zeros of the classical Rieman-zeta function $\sum \frac{1}{\xi^s}$ in complex analysis are conjectured to fall. This emphasis the natural connection between classical mathematics, probability theory and modern computer analysis.

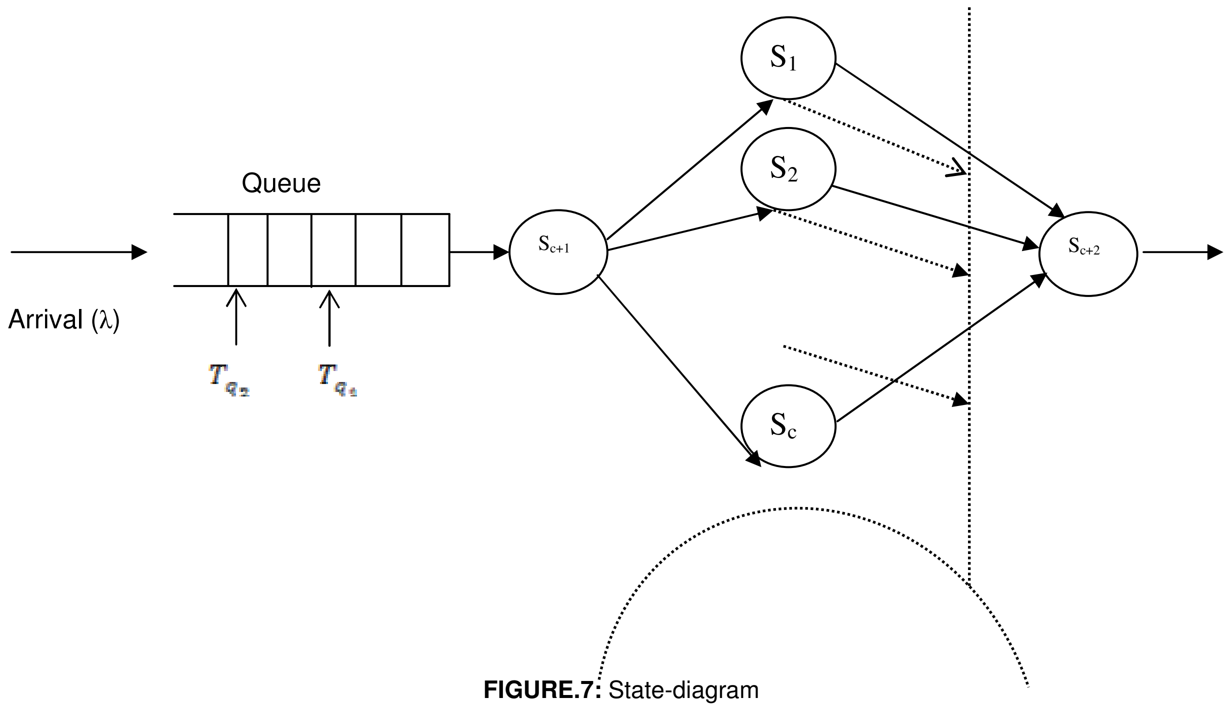


FIGURE.7: State-diagram

We now enter into the actual methodology for the proposed system model.

6. PROPOSED SYSTEM MODEL

In this proposed system we adopt the principles of Adaptive load balancing model. This application is all about sharing the load directed upon a single server across multiple redundant severs thereby reducing the per head load. The per-head load is reduced by sharing the load equally by all the redundant servers. The scenario is as shown in Figure 8.

6.1 Assumptions

It is assumed that the system has only five replicated servers all the time and the service each provides is identified by the respective port number only. If two different redundant servers provide a service in the same port number it is assured that the two services are the same *i.e.* if a client requests for a service providing a port number all redundant servers (if they have such a service) have the same service at that requested port number. All the packets that flow through the NP are assumed to be TCP packets running on IP.

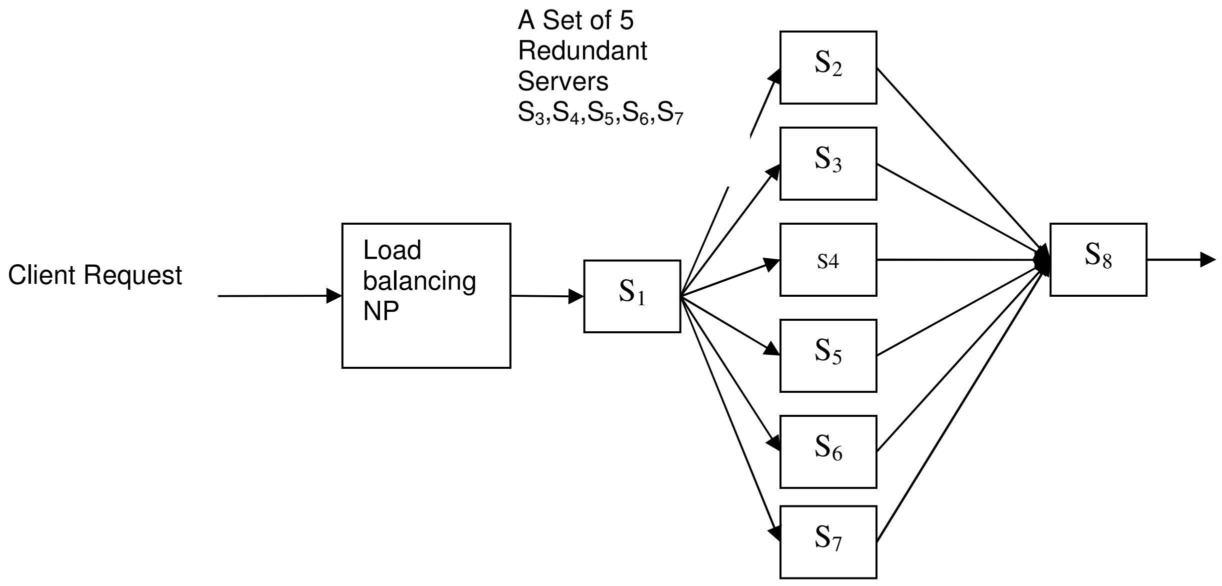


FIGURE 8: Adaptive Load Sharing- Context diagram

6.2 Runtime Mapping

6.2.1 Mapping I: Min_service_upon_less_traffic

First, we consider the minimum traffic in the system. At that time, the system uses only single ALS instance. This is a basic pipeline consisting of Ingress, processing ALS operations (i.e., ALS instance) and egress as shown in Figure 9.

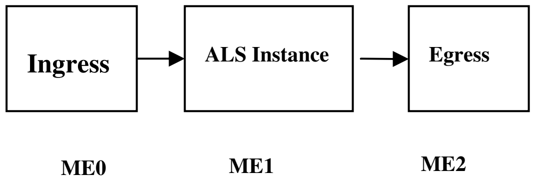


FIGURE 9: Base line Configuration

6.2.2 Mapping II: Min_service_upon_moderate_traffic

Second, we consider the moderate traffic flow. A copy of the previous instance is replicated in order to accommodate the moderate traffic as shown in Figure 10.

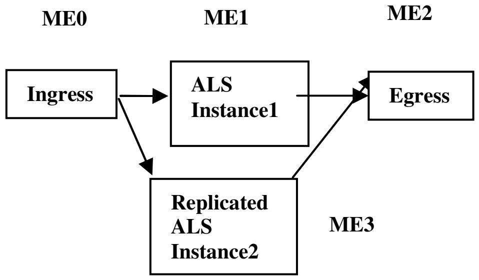


FIGURE10. Replication of the same instance

6.2.3 Mapping III: Max_service_upon_high_traffic

In order to overcome the heavy traffic, a new ‘substituted’ instance of the ALS application is deployed (Figure. 11). In the substituted new instance, the ALS function is split to different micro-engines, whereas in the replicated instance all the functions are carried out in a single micro-engine.

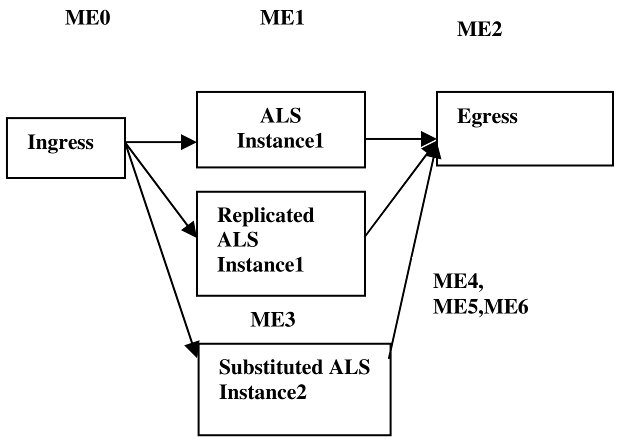


FIGURE 11: Substituting new instances

We use the following setup to evaluate the runtime systems

7. IMPLEMENTATION - HARDWARE PLATFORM

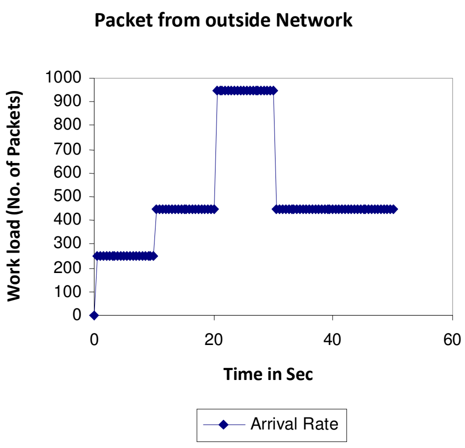
Our setup contains a PC hosting the Radisys ENP-2611 board with Intel® 600MHZ network processor. IXP2400 contains one XScale^{TM} core and eight micro-engines. Each micro-engine has an instruction store to hold 4K-40 bit instructions that are optimized for fast-path packet processing. In Intel IXP2400 Developers Workbench, Microcode Assembly language is used to implement the system and X-scale implementation incorporated Embedded C programming language for implementation. Intel IXP2400 Developers Workbench in a simulation tool which helps us to execute the micro engine code on it and give the performance measures for the application program. Later the same application is ported onto the hardware. The execution of the code can be traced on an instruction-by-instruction basis using the simulation tool. This helps in debugging the code as well as in providing means to validate the code. There are provisions to watch the runtime values getting stored in various memory storage units like SDRAM, SRAM, Scratch Pad Memory, Local Memory and Micro -engine Registers.

8. RESULT ANALYSIS

The following section details the manner in which the baseline configurations (a) and (b) are compared to the dynamic configuration (c) described by us so far.

a. Traffic from outside Network:

The Traffic trace taken from the network are classified into low (250 pkts/sec), moderate (450 pkts/sec) and high (950 pkts/sec) and the traffic mixture taken are low, moderate and high. In Figure.12, the traffic mixture with low traffic was maintained for the first 10 seconds and then the traffic was increased to moderate for the next 10 seconds and finally the traffic was increased to high profile.



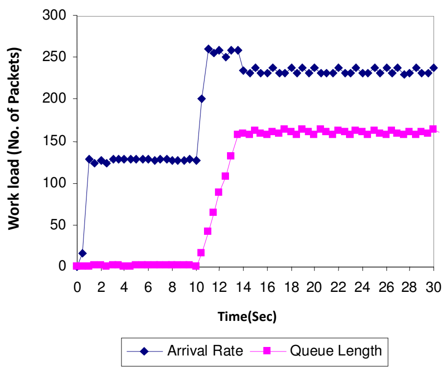


FIGURE 12: Snapshot of Traffic from outside network

FIGURE 13: Arrival rate and lagging packet in Mapping I

b. NP without Dynamic Reconfiguration:

Mapping I

At the initial stage, the system uses minimum resources so that (3 micro-engines) have been provisioned for NP. And the monitoring interval is 0.5 second. In Figure 13., the traffic mixture taken is low, moderate and high. So, when the workload gets increased (moderate traffic), the micro engine will not be able to process all the incoming packets, hence the queue size increases abruptly and queue overflow takes place, because in this mapping-I, the system uses only three micro-engines, which are ingress and egress. Hence the system itself with a single micro-engine, so it will tolerate only moderate traffic.

Mapping II

In the second level, the system is in moderate traffic. The Figure.14 shows the mapping of second level. At this moderate traffic, four micro engines have been used. And the monitoring interval is 0.5 second. The traffic mixture taken is low, moderate and high. So when the workload is increased (high traffic), the micro engine will not be able to process all the incoming packets, hence the queue size increases abruptly and as before queue overflow takes place.

Mapping III

At the third level, the system is in high traffic; here all the eight micro engines have been provisioned for NP. And the monitoring interval is 0.5 second. The traffic mixture taken is low, moderate and high. So when the workload increases, the micro engine will be able to process all the incoming packets, hence the queue overflow will not take place (see Figure. 15). So, in this mapping-III the system used, maximum capacity in high level traffic.

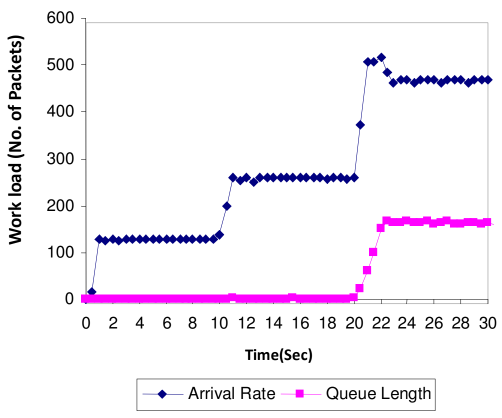


FIGURE 14: Arrival rate and lagging packet in Mapping II

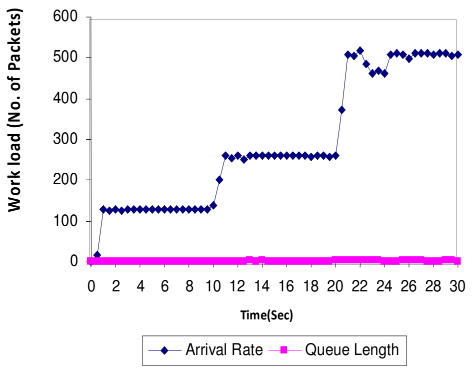


FIGURE 15: Arrival rate and lagging packet in Mapping III

c. NP with Dynamic Reconfiguration

In this dynamic reconfiguration, at initial stage minimum resource will be provisioned for NP. When the traffic get increased from low to medium, the minimum resource will not be able to manage the workload, so the queue length get increases in time for example 10-12 sec and 20-22 sec (see Figure.16) .In this stage, the system will balance this situation to activate additional resources. The monitoring module which monitors the packet rate from X-scale core, deploy the code using runtime environment module in order to tolerate the moderate workload. Once again the heavy traffic is injected into the NP and the maximum resource will be provisioned in order to overcome the network traffic.

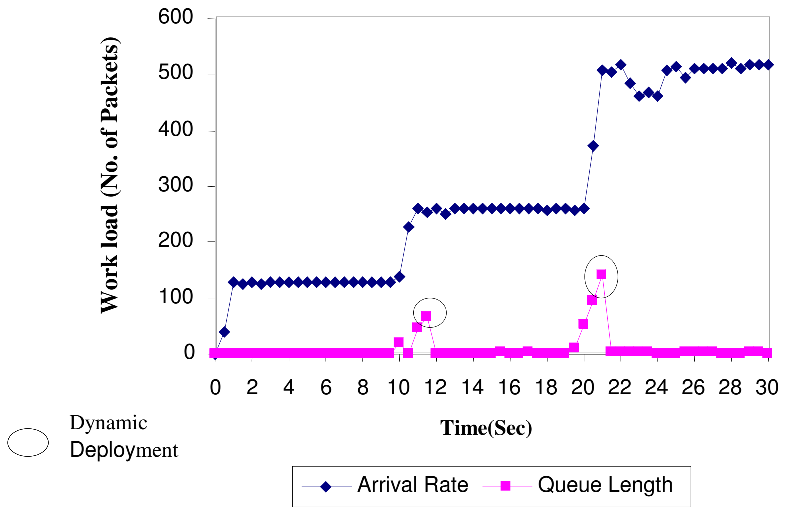


FIGURE 16: Arrival rate and lagging packet in Dynamic Reconfiguration

9. CONCLUSION AND FUTURE PLAN

To summarize, we have described the design and implementation of a dynamic reconfigurable system for Intel IXP2400 NP that can perform resource allocation at runtime. This allows maximum utilization of resources whenever possible in the steady state. In the baseline experiments (using the three mappings), the Micro-engine pipelines were not reconfigurable. The comparison of the baseline configuration over dynamically reconfigurable NP as described in the earlier sections shows the superior performance of Self-Configurable NPs over baseline NPs. Our future work is to develop an adaptive system for IPV6 services. We propose to implement an adaptive system that automatically changes the mapping of IPV6 services to processors, and handles migration of services between different processor core types to match the needs.

10. REFERENCES

1. A.Kind, R.Pletka, and M.Waldvogel *"The role of network processors in active networks"* In Proceedings of IWAN 2003, pages 18-29, Kyoto, Japan, December 2003.

2. Arun Raghunath, Aaron Kunze, Erik J. Johnson, Vinod Balakrishnan *"Framework For Supporting Multi-Service Edge Packet Processing On Network Processors*". Architecture for networking and communications systems. 26-28 Oct. 2005 Page(s):163 – 171.

3. Bill Carlson, "*Intel® Internet Exchange Architecture & Applications A Practical Guide to Intel's Network Processors",* Intel Press.

4. Dimitri Bertsekas, Robert Gallager , "*Data Networks*" , pp.115, PHI-1987.

5. Douglas E. Comer "*Network Systems Design using Network Processors*", Prentice Hall, Jan 2003.

6. A. Troxel, A. D. George, S. Oral, "*Design and Analysis of a Dynamically Reconfigurable Network Processor*," 27th Annual IEEE International Conference on Local Computer Networks (LCN'02), 2002, pp.0483.

7. Intel IXP2400/IXP2800 Network Processors, *"Intel XScale Core Support Libraries Reference Manual"* November 2003.

8. Intel® IXP2400 and IXP2800, Network Processor Programmer's Reference Manual, July 2005.

9. "*IXP2400 Hardware Reference Manual",* June 2001, Intel Corporation,

10. "IXP 2400 Development Tools User's Guide", June 2001, Intel Corporation.

11. J. Allen, B. Bass, C.Basso, R. Boivi, J.Calvignac, G.Davis, L.Frelechoux, M.Hedds, A. Herkersdorf, A.Kind, J.Logan, M.Peyravian, M.Rinaldi, R.Sabhikhi, M.Siegel, and M. Waldvogel, *"IBM PowerNP Network processor: Hardware, software, and applications", IBM* Journal of Research and Development , Volume (47), nos. 2/3 , pp.177-194,2003.

12. Kevin Lee, Geoff Coulson, Gordon Blair, Ackbar Joolia, Jo Ueyama *"Towards a Generic Programming Model for Network Processors*" In Proc IEEE International Conference on Networks (ICON04), Singapore, November 2004.

13. Kevin Lee, Geoffrey Coulson *"Supporting Runtime Reconfiguration on Network Processors",* Proceedings of the 20th International IEEE Conference on Advanced Information Networking and Applications (AINA'06) Volume 1, 18-20 April 2006 Page(s): 721 – 726.

14. L.Thiele, S.Chakraborty, M.Gries, and S.K¨unzli, *"Design Space Exploration of Network Processor Architectures"*, Proc. First Network Processor Workshop/Eighth IEEE Int'l symp. High Performance Computer Architecture (NP/HPCA'02), pp.30-41, Feb.2002.

15. Ravi Kokku, T.Rich´e, A. Kunze , J.Mudigonda , J.Jason and H.Vin, *"A Case for Run-time Adaptation in Packet Processing Systems"* Proc.Second Workshop Hot Topics in Networks (HOTNETS'03), Nov.2003.

16. Ravi Kokku, Upendra Shevade, Nishit Shah, Harrick M. Vin, Mike Dahlin *"Adaptive Processor Allocation in Packet Processing Systems"* University of Texas at Austin Technical Report # TR04-04.

17. A.Satheesh**,** S.Krishnaveni, S.Ponkarthick *"Self-Configurable Environment for the Intel IXP 2400 Network Processor"* International Journal of Computers and Applications, 31(4):268-273, 2009.

18. Tilman Wolf, "*Network Processors - Flexibility and Performance for Next-Generation Networks*", ACM SIGCOMM Computer Communication Review, Volume 32 , Issue 1 (January 2002) Pages: 65.

19. T.Wolf and M.Franklin, "*Performance Models for Network Processor Design", IEEE Trans. on Parallel and Distributed Systems*, vol. 17, no.6, Pages: 548 – 561, June 2006.

20. Tilman Wolf, Ning Weng, Chia-Hui Tai, "*Runtime Support for Multicore packet processing systems*", IEEE Network, Page(s).29-37, July/August-2007.

21. Vinod Balakrishnan, Ravi Kokku, Aaron Kunze, Harrick Vin, Erik J. Johnson *"Supporting Run-Time Adaptation in Packet Processing System"* Intel Research and Development University of Texas at Austin 2004, Technical Report.

22. Xin Huang, Tilman Wolf, "*A Methodology for Evaluating Runtime Support in Network Processors*", Architecture for Networking and Communications systems, ACM/IEEE Symposium on Volume , Issue , 3-5 Dec. 2006 Page(s):113 – 122.

23. Xin Huang, Tilman Wolf, *"Evaluating Dynamic Task Mapping in Network Processor Runtime Systems"* IEEE Transactions on Parallel and Distributed systems, vol. 19, no. 8, August 2008, Page(s).1086-1098.