國 立 交 通 大 學
資 訊 學 院

資 訊 科 學 與 工 程 研 究 所

博 士 論 文

多執行緒多處理器網路處理器之資源分配--針對計算密集及記憶體存取密集的網路應用程式

Resource Allocation in Multithreaded Multiprocessor Network Processors for Computational Intensive and Memory Access Intensive Network Applications

研 究 生：林義能

指導教授：林盈達　博士

中 華 民 國 九 十 六 年 七 月

多執行緒多處理器網路處理器之資源分配--針對計算密集及記憶體存取密集的網路應用程式

Resource Allocation in Multithreaded Multiprocessor Network Processors for Computational Intensive and Memory Access Intensive Network Applications

研 究 生：林義能　　　　Student：Yi-Neng Lin

指導教授：林盈達　　　　Advisor：Ying-Dar Lin

國 立 交 通 大 學

資 訊 科 學 與 工 程 研 究 所

博 士 論 文

A Dissertation Submitted to

Department of Computer Science

College of Computer Science

National Chiao Tung University

for the Degree of

Doctor of Philosophy

in

Computer Science

July 2007

Hsinchu, Taiwan, Republic of China

中華民國九十六年七月

多執行緒多處理器網路處理器之資源分配--針對計算密集及記憶體存取密集的網路應用程式

學生：林義能　　　　　　　　　　　　指導教授：林盈達

國立交通大學資訊科學與工程研究所博士班

摘　　要

今日網路應用程式之處理需要強大的硬體平台以應付日益龐大的計算量以及記憶體存取。此平台亦必須能夠隨著協定或產品規格之變動而作有效的調整。沿用已久的多用途處理器架構，其效能往往被“核心-使用者程式”間的溝通以及執行緒轉換的負擔拖累;而常用的 ASIC 解決方式則受限於開發時程過久且調整不易的缺陷而無法滿足需求。

本篇論文主要探討(1)應用日益盛行的網路處理器架構來加速網路網路封包處理的可行性，此網路處理器包含多個處理器且每個處理器包含多個硬體執行緒，具有豐富硬體資源、較小的執行緒轉換負擔以及可調整性等優點，和(2)用此平台來處理不同計算或記憶體存取量的網路應用程式時硬體資源的分配。我們首先檢視各

種不同的網路處理器並將其分成“助理處理器為主”和“核心處理器為主”兩大類。就前者而言，助理處理器負責占封包處理主要工作的資料面象部分，而後者則是由核心處理器兼顧所有的控制面象和大部分的資料面象的處理。之後我們針對計算密集以及記憶體存取密集的網路應用程式分別用“助理處理器為主”和“核心處理器為主”的兩種網路處理器來實作並評估其效能。最後，根據實作的經驗我們進一步設計出其數學模型以及模擬環境，以期能找出設計、使用此二種架構時的參考。

Resource Allocation in Multithreaded Multiprocessor Network Processors for Computational Intensive and Memory Access Intensive Network Applications

Student: Yi-Neng Lin　　　　　　Advisor: Dr. Ying-Dar Lin

Institute of Computer Science and Engineering
College of Computer Science
National Chiao Tung University

Abstract

Networking applications today demand a hardware platform with stronger computational or memory access capabilities as well as the ability to efficiently adapt to changes of protocols or product specifications. Being the ordinary options, however, neither a general purpose processor architecture, which is usually slowed down by kernel-user space communications and context switches, nor an ASIC, which lacks the flexibility and requires much development period, measures up.

In this thesis, we discuss (1) the feasibility of applying the emerging alternative, network processors featuring the multithreaded multiprocessor architecture, rich resources, minor context switch overhead, and flexibility, to solve the problem, and (2) the ways of exploiting those resources when dealing with applications of different

computational and memory access requirements. We start by surveying network processors which are then categorized into two types, the coprocessors-centric and the core-centric ones. For the former, the coprocessors take care of the data plane manipulation whose load is usually much heavier than the one of the control plane, while in the latter the core processor handles the most part of packet processing, including the control plane and data plane. After that we evaluate real implementations of computational intensive and memory access intensive applications over the coprocessors-centric and core-centric platforms, respectively, aiming to unveil the bottlenecks of the implementations as well as the allocation measures. Finally, based on the evaluations, analytical models are formalized and simulation environments are built to observe possible design implications for these two types of network processors.

致　　謝

回首研究生生涯，從剛開始的懵懂，到後來終於培養出自己的一套做事方法與態度，這都必須歸功於我的指導老師 林盈達教授。也正是因為他的教誨、包容與鼓勵，遇到困難的時候總能夠一路披荊斬棘，也及時導正了方向。在此，我要誠摯地表達出對林老師的感謝。

高速網路實驗室學長學弟們的討論與扶持也是這幾年來重要的支撐。謝謝賴源正學長、尹維銘學長、曹世強學長、林柏青、曾國坤以及學弟們，讓我能夠感受到滿滿的情誼。

最後要感謝家人（爸爸媽媽妹妹弟弟）這幾年來精神上的支持。融洽的家庭氣氛對於我情緒與壓力的排解有著非常正面的幫助。謝謝子晴、筱玲、和佳筠，你們的陪伴與傾聽讓我有堅持到最後的力量。你們是我永遠的家人。

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Challenges of Hardware Platforms for Modern Networking Applications

Increasing link bandwidth demands faster nodal processing, especially of data-plane traffic. Nodal data-plane processing ranges from routing table lookup to various classifications for firewall, DiffServ and Web switching. The traditional general-purpose processor architecture is no longer sufficiently scalable for wire-speed processing, and some ASIC components or co-processors are commonly used to *offload* the data-plane processing, while leaving only control-plane processing to the original processor.

Several ASIC-driven products have been announced in the market, such as the acceleration cards for encryption/decryption, VPN gateways, Layer 3 switches, DiffServ routers and Web switches. While accelerating the data-plane packet processing with special hardware blocks, much wider memory buses, and faster execution processes, these ASICs lack the flexibility of *reprogrammability* and have a long development cycle usually of months or even years. The cost of possible design failures is also high.

Network processors are emerging as an alternative solution to ASICs for providing re-programmability while retaining scalability for data-plane packet processing. A network processor typically consists of one core processor and a number of coprocessors, so that developers can embed the control-plane and data-plane traffic management modules into the core and coprocessors, respectively. Scalability concerns due to the computational and memory access overhead, in data-plane packet processing could be satisfied with the hardware contexts of minor context switching overhead in each of the coprocessors as well as the instructions specifically for networking.

1.2 The Importance of Resource Allocation for Network Processors

Though network processor is promising in its scalability and extensibility [LLP02, LLY$^+$03, BH95], the determination of architectural parameters such as numbers of processors, threads in a processor, and memory banks, respectively, is not trivial given a specific application and hardware platform combination. Furthermore, since one proper configuration today may not be suitable tomorrow due to different evolving speeds of manufacturing technologies of the functional units, some general guidelines may be demanded for efficient and appropriate parameter determination.

1.3 Coprocessors-centric and Core-centric Network Processors

Two types of network processors, the coprocessors-centric and core-centric ones, are classified and addressed in the thesis. In the former, a number of coprocessors are used to take care of the data plane manipulation whose load is usually much heavier than the one of the control plane. In the latter the core processor handles the most part of packet processing, including the control plane and data plane; only few coprocessors are required to offload some computational intensive processing.

Since the coprocessors-centric model is used mostly to offload the data plane, especially the memory access intensive processing, for its multithreading architecture, we investigate the resource allocation by implementing the Intrusion Detection and Prevention (IDP) system over the IXP2400 network processor.

As for the core-processor centric model, we implement the Virtual Private Network (VPN) gateway, which needs to offload limited portion of computational intensive operations to the coprocessors, over the IXP425. For both types we also investigate the effect of different architectural parameters through mathematical modeling.

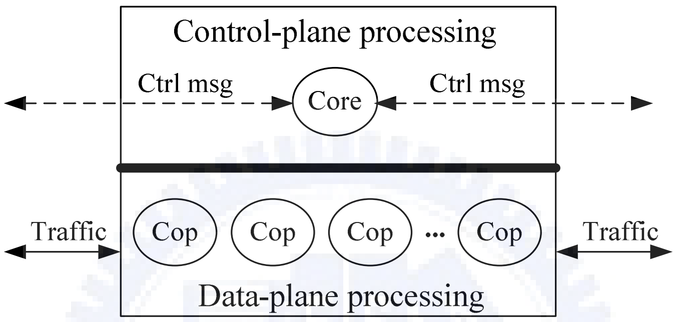


Fig. 1.1. Coprocessors-centric network processors.

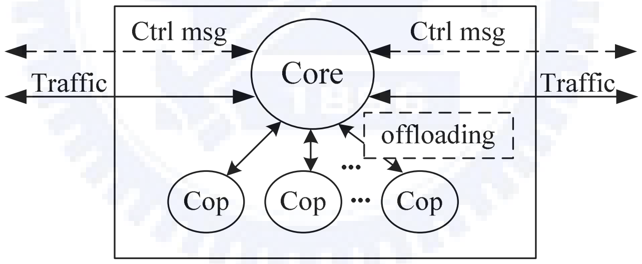


Fig. 1.2. Core-centric network processors.

1.4 Related Works

In this chapter, we present some prior groundwork for our thesis. To comply with our research directions mentioned previously, we discuss the related works in two aspects: (1) application implementation and (2) mathematical modeling and simulation. The following is summarized in Fig. 1.3.

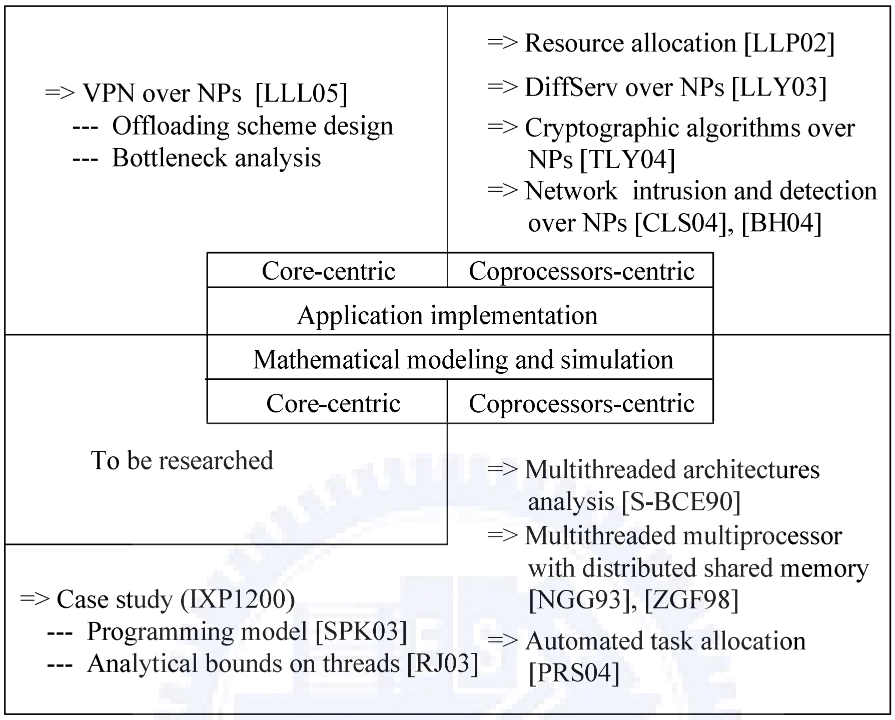

=> VPN over NPs [LLL05]
--- Offloading scheme design
--- Bottleneck analysis
=> Resource allocation [LLP02]
=> DiffServ over NPs [LLY03]
=> Cryptographic algorithms over NPs [TLY04]
=> Network intrusion and detection over NPs [CLS04], [BH04]
Core-centric
Coprocessors-centric
Application implementation
Mathematical modeling and simulation
Core-centric
Coprocessors-centric
To be researched
=> Multithreaded architectures analysis [S-BCE90]
=> Multithreaded multiprocessor with distributed shared memory [NGG93], [ZGF98]
=> Automated task allocation [PRS04]
=> Case study (IXP1200)
--- Programming model [SPK03]
--- Analytical bounds on threads [RJ03]

Fig. 1.3. Related works on the network processor resource allocation problem.

1.4.1 Application Design and Implementation

For memory access intensive applications, some researches have focused on improving the throughput by the deployment of network processors. Bos and Huang [BH04] implemented an NIDS over the Intel IXP1200 [INT]. The prototype comprises only the receiver and packet processing using the Aho-Corasick [AC75] algorithm, but it does not support inspection of patterns across more than two packets as well as multiple flows. Clark et al. [CLS$^+$04], designed a Network Intrusion Detection and Prevention System (NIDP) utilizing an IXP1200 and an FPGA. The former is for header processing and the latter serves as the signature matching engine, and the bottleneck is found to be the bus connecting them. Nevertheless, those researches did not discuss in detail on proper resource allocations.

As for the computational intensive application over network processors, to date only one can be found in the literature [TLY+04]. The authors implement various cryptographic algorithms over the IXP2800 network processor, analyze the instruction mix and compare it with other header processing applications, and finally propose implementation and optimization principles to improve overall performance. They find that the ALU operations occupy a significant share, 79.9%, of the total instruction mix, compared to the 58% of the Commenbench [WF00] PPA (Payload Processing Applciations), 53.5% of the NpBench [LJ03], and 41% of the Commonbench HPA (Header Processing Applications).

The implementation principles, besides some minor techniques, include the flow-level and intra-block-level parallelisms. In the flow-level parallelism in which each thread is allocated to a flow, it is observed that incorporating multiple threads does not necessarily improve the performance but depends on the algorithms. Another reason for the limited improvement is that multithreading is found only help consume more of the stalled cycles rather than the idle ones. To utilize the idle cycles, they use the intra-block level parallelism, in which one main ME (namely processor) and a helper ME are involved in processing a certain block of instructions. The helper ME pre-fetches the data from memory for the later use of the main ME. Some principles are also proposed for optimization such as (1) increasing the cache size on MEs to hold tables, (2) enlarging the memory and command queues and (3) organizing the MEs into a smaller cluster for fast shared-bus performance.

1.4.2 Mathematical Modeling and Simulation

Analytical approaches have been favored in many researches for its capability of fast evaluation of the systems under investigation [SMA03]. However, limited researches have devoted to the modeling of multithreaded multiprocessors. Rafael et al. [S-BCE90] proposed a model to obtain the performance, in terms of processor efficiency, of a multithreaded architecture with varying number of threads. The effect of multiprocessor can be mimicked by adjusting the memory access latency which is assumed geometrically distributed. This model possesses good abstraction of the architecture; however, the interaction between the processing elements and the memory subsystem is disregarded.

This problem was remedied in [NGG93] by including the memory subsystem in their model, in which the processing elements as well as the memory are distributed and shared. Each thread is capable of a complete packet processing, and has a rate to access local/remote memory modules during processing. Nevertheless, the model is not feasible since the queuing network adopted was a closed one, and thus does not consider the packet arrivals and departures of real networking applications.

A number of recent works concerning the modeling of NPs can be found in [FW02, WT01, GKS03, CFB01, CB02]. Though detailed parameters are included and programming paradigms are analyzed in their models, the discussion and consideration of thread allocation are substantially ignored. Lakshmanamurthy et al. proposed a methodology for analyzing the performance of the Intel IXP2400 [LLP02]. But they focused only on the validation of the system performance, while the processor and memory utilizations are not addressed and no design guidelines are suggested. In [SPK03] and [RJ03], the authors propose a programming model and an analytical method, respectively, for the IXP1200 as a case study. The former considerably accelerates the process of the application implementation and verification; the latter delivers the analytical bounds on the optimum number of threads. Moreover, Gries et al. in [GKS03] uses Network Calculus to model the IPv4 forwarding on the IXP1200. In [PRS04], the authors utilize the Linear Programming to achieve automated task allocation on multithreaded multiprocessor systems.

1.5 Thesis Objective and Dissertation Road Map

As mentioned in the previously, to leverage network processors for networking applications, we may need to arrange well the hardware resources. Further, some design implications may also be demanded for future network processors. The objective of this thesis is therefore:

to investigate resource allocation measure and design implications for network processors.

The roadmap of the dissertation is organized as follows. Chapter 2 declares the methodologies to the problem. Chapter 3 and chapter 4 present the investigation on resource allocation for coprocessors-centric model by implementing the IDP over IXP2400 and by mathematically modeling the similar architecture, respectively. Chapter 5 and chapter 6 discuss the implementation and modeling for the core-centric model. The results summary of the dissertation is mentioned in chapter 7.

Chapter 2

Research Methodologies

2.1 Application Design and Implementation

Since NPs are used to leverage the processing of networking applications, we need to verify the feasibility of doing so, namely by implementing those applications over NPs. We then try to identify possible bottlenecks after prototyping. The benefits from the identifications are two-fold: serve as (1) the implications for future NPs design, and (2) the foundation for further investigation on the optimal resource allocation. Before implementation, we need to understand the software architecture of the platforms. We also mention the environment and the tools for external and internal benchmarks.

2.2.1 Software Architecture of IXP425

The software architecture of IXP425 shown in Fig. 2.1 can be divided into two portions, namely the platform independent (applications and some higher level components such as networking protocol stacks in OS) and dependent parts (mainly device drivers). This design is favorable especially when an OS migration from a certain H/W platform to another is demanded, that is, the developers need to focus only on the dependent part, namely the development of drivers. When implementing device drivers, a set of software libraries collectively referred to as AccessLibrary can be used to drive devices such as NPEs, coprocessors, peripherals, etc. The AccessLibrary also provides utilities, such as OSSL and IxOSServices to implement some OS-related functions such as mutual exclusion.

The software processing flow is described as follows with library functions adopted from the AccessLibrary. During the boot time a function named IxNpeDl is called to download the corresponding code image into the instruction cache of each NPE. Then two functions, IxQmgr and IxNpeMh, are called to initialize the queue manager as well as the message handler responsible for the communications between NPEs and XScale. The Ethernet-related functions, IxEthAcc and IxEthDB, are used to receive and transmit Ethernet frames, while the IxCryptoAcc function is incorporated for possible cryptographic operations during packet processing.

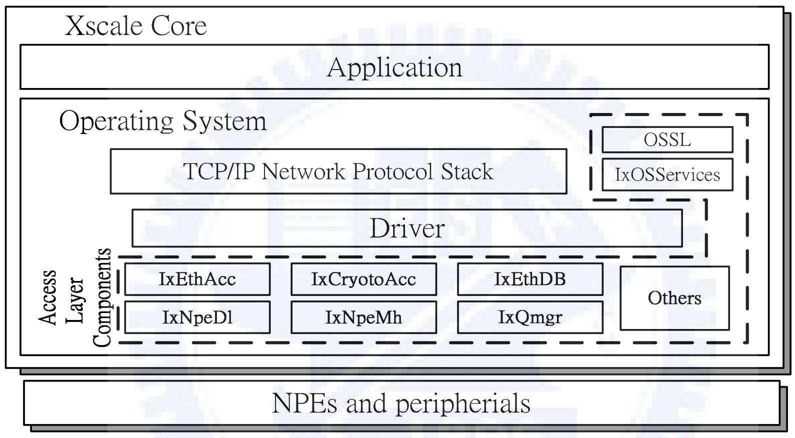


Fig. 2.1. Software architecture of IXP425.

2.2.2 Software Architecture of IXP2400

Figure 2.2 elaborates the development environment. The IXP2400 programming can be divided into the XScale programming and the microengine programming. While XScale programs are written in C/C++ under Tornado, microengine programs are written in assembly under Workbench for low-level packet processing capability. The compiled XScale executable is linked with object microcode compiled by the assembler, and loaded into the IXP2400 SRAM from which XScale initializes and loads microcode into the Control Store of microengines. The linked program can also be executed by the Transactor for pure software simulation. Besides, the XScale is little-endian and byte-addressable while microengines are little-endian but longword-addressable.

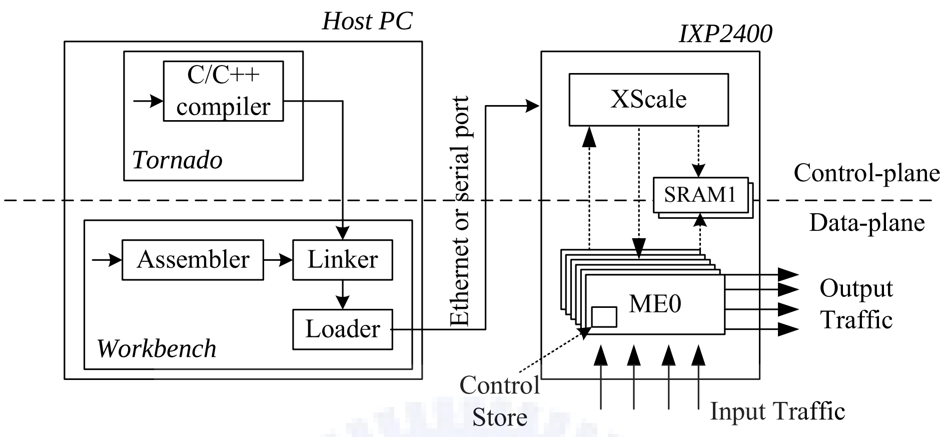


Fig. 2.2. Software architecture of IXP2400.

2.2.3 Performance Benchmark

Figure 2.3 illustrates the external benchmark environments, for packet forwarding and IPsec. We use SmartBits, which is a networking traffic generator and a performance analyzer, to generate the input traffic and collect and analyze the performance results. For internal tests, some system utilities, such as *vmstat*, *top* and *GProf*, are employed to obtain the system state and other internal behaviors such as CPU utilization and memory usage.

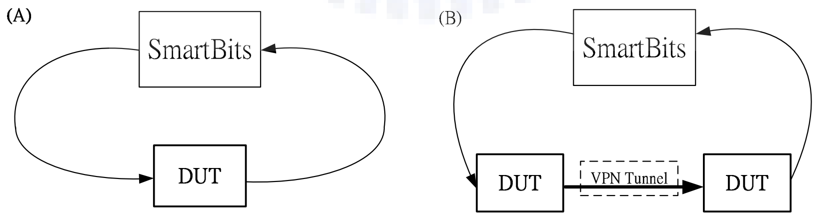


Fig. 2.3. Benchmark environments for (a) packet forwarding and (b) IPsec.

We also conduct a number of internal benchmarks, namely board-level simulations using the Transactor within the Workbench, in order to have detailed observations on the hardware utilizations.

2.2 Mathematical Modeling and Simulation

Real implementations reveal precise observations for specific software/hardware combinations; however, they can hardly reflect generalized implications because of the difficulty in adapting architectural parameters. To remedy this shortcoming, we incorporate mathematical modeling as well as simulations. The former has the best flexibility and efficiency in altering parameters; nonetheless, it often suffers from the problem of state-space explosion. Though being less flexible and efficient than the mathematical modeling, the latter captures well the behaviors of a certain system.

Since our goal is to consider *I* processors, each of which contains *J* threads, and then capture the behaviors of processors, threads and memory, we use the Continuous Time Markov chain to mimic a multithreaded multiprocessor network processor. Figure 2.4 exemplifies the transition diagram of a thread. In this example, a thread could be *idle*, *active* in processing, accessing *memory*, *ready* if not permitted to run, and *finished* if the packet processing is completed. Based on this concept we can have further extension to support the modeling of multithreaded multiprocessor architecture.

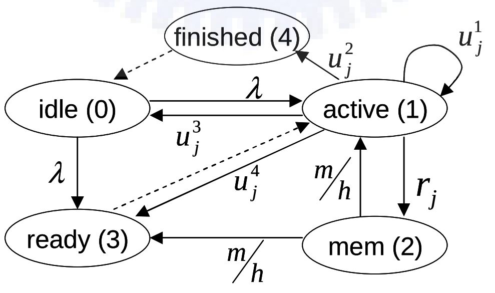


Fig. 2.4. Transition diagram of a thread in a multithreaded multiprocessor environment.

As for the simulation, we adopt the CPN Tools [RWL+03] to employee the timed and colored Petri nets [Mur89] that capture well component-level activities.

The features it supports, including the colored tokens, stochastic functions and hierarchical editing, provide efficiency in the construction of timed, colored Petri nets corresponding to both coprocessors-centric and core-centric models. Figure 2.5 shows an example Petri net describing a multithreaded processor.

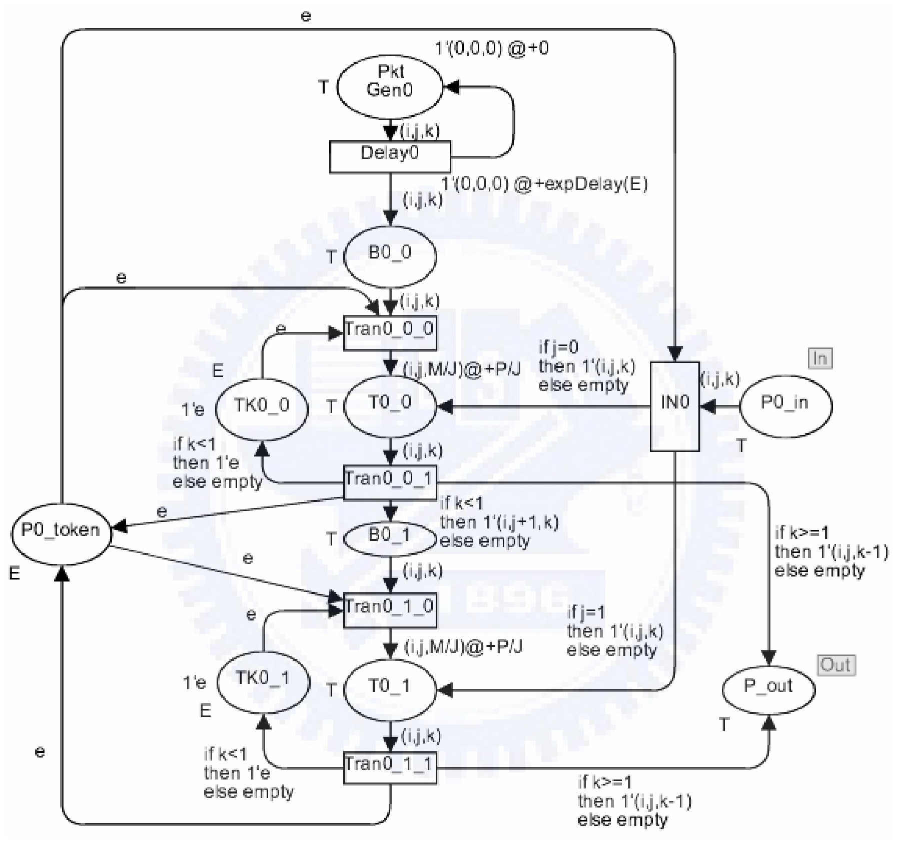


Fig. 2.5. Petri net of a multithreaded processor.

Chapter 3
Resource Allocation of the Coprocessors-centric Network Processors for Memory Access Intensive Applications

3.1 Introduction

Networking applications offering extra security and content-aware processing features demand much powerful hardware platforms to achieve high performance. For memory-access intensive applications such as the Network Intrusion Detection Systems (NIDSs) [Roe], general purpose processors with high speed memory banks are often adopted; however, the cost is considerable while the throughput is not satisfactory for that the processors' utilization is low because of the heavy memory-access overhead. Rather, the Application-Specific Integrated Circuits (ASICs) [JS97] can meet the performance requirement with a circuitry designed for strict guarantees on memory-access latency using pipelined architecture and embedded memory. Nonetheless, the lack of flexibility and long development cycle make it less appealing.

In this work, we implemented a memory-access intensive application, NIDS, over the Intel IXP2400 [INT] whose architecture is similar to most network processors, evaluated the effect of different resource allocations, and finally investigated the allocation measures. Two signature matching algorithms, the Aho-Corasick and Wu-Manber [WM94], were incorporated for their popularity in many security-related implementations, for example, Snort. Several software components referred to as *processing stages* [ARB02] were characterized, in which a tentative processor/thread allocation was applied. After implementation, we then conducted both external and internal benchmarks. The former unveiled the throughput of the implementation while the latter analyzed the utilizations of the hardware components for observing potential bottlenecks. According to the benchmark result, the effect of the ME/thread allocation is reviewed and methodologies for the optimal revision of the allocation were subsequently proposed. Finally, since extra memory banks are often exploited to shorten the memory access latency, the feasibility and effectiveness of adopting multiple banks for string-matching applications are discussed.

3.2 Hardware Platform (IXP2400)

As depicted in Fig. 3.1, the IXP2400 consists of several components that are categorized as following.

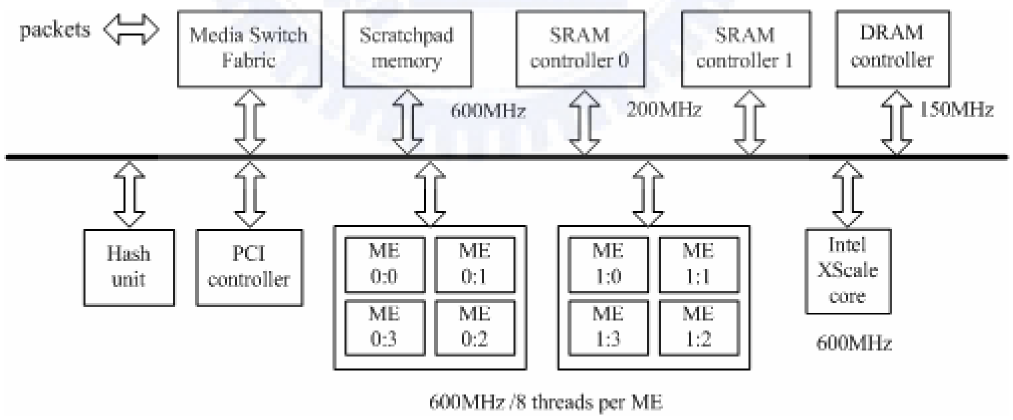


Fig. 3.1. Hardware architecture of IXP2400.

Multithreaded multiprocessor architecture

The IXP2400 features nine programmable processors: one Intel XScale core [INT] and eight microengines (MEs), operating at 600MHz. The Intel XScale core is responsible for housekeeping functions such as table initialization and exception handling for control-plane packets such as ICMP unreachable packets. Data-plane processing, which accounts for the most part in packet processing, is implemented on MEs. Every ME has eight hardware threads, each of which having its own register set and program counter to support fast context switch when memory accesses occur.

Hierarchical memory structure

To ease the memory-access overhead, IXP2400 exploits four types of memories, DRAM, SRAM, scratchpad, and local memory in an ME, given tradeoffs between size and latency. IXP2400 has one channel of DDR running at 150MHz. The channel can support up to 2GB of DRAM, yielding enough capacity for storing packets. Two channels of Quad Data Rate (QDR) SRAM running at 200MHz are also provided, and up to 16MB can be populated on each channel. The SRAM is primary for accommodating packet descriptors for locating packets in DRAM, queue descriptors, and other data structures frequently used. The on-chip 16KB scratchpad memory operates in the form of rings and provides similar capability to SRAM, while the 2560-words local memory is frequently used as a cache for smaller data structures.

Flexible external interface

The Media Switch Fabric (MSF) is an external interface used to connect the Intel IXP2400 to a physical layer device and/or a switch fabric. The MSF consists of receiving and transmitting interfaces which can be configured for different protocols such as POS PHY Level 3 [POS] and CSIX-L1 [CSI]. Incoming packets are received into the Receive Buffer (RBUF) and outgoing packets are held in the Transmit Buffer (TBUF), which are both 8KB in size. The MEs can move data from RBUF to DRAM and from DRAM to TBUF using the DRAM[rbuf_rd] and

DRAM[tbuf_wr] instructions directly, greatly avoiding packet duplications and unnecessary memory accesses.

Coprocessors

Two kinds of hardware coprocessors, including a hash unit shared by all MEs and a Cyclic Redundancy Code (CRC) unit inside each ME, are incorporated in the system. The hash unit is capable of 48-bit, 64-bit and 128-bit polynomial divisions. A high quality hash alleviates the probability of hash collisions, contributing to fewer memory accesses; however, performing a high-quality hash in software, which occurs frequently in packet classification, is cycle-consuming, and thus should be offloaded to the coprocessor. Similarly, the CRC unit is used to offloading the CRC computation.

Detailed Packet Flow in IXP2400

The processing flow of an ordinary packet is elaborated below referring to Fig. 3.1. Upon the arrival of a packet at the MSF of IXP2400, the MSF partitions the packet into several smaller chunks called mpackets, which can be configured to 64, 128, and 256 bytes in size, and places them into the RBUF elements. The threads of the MEs dedicated for packet receiving in turn perform the reassembly of mpackets, and move them directly from the RBUF into DRAM, in which MEs and the Intel XScale core carry out further operations. The packet processing typically consists of packet classification followed by packet modification. During packet processing at MEs, chances are that some exception handling and housekeeping are manipulated by the Intel XScale core through the interrupt and message queue mechanism. In the later scenario of packet flow, the transmission process is just the reverse of the reception process, namely the packet is segmented into several mpackets by the threads dedicated for packet transmission, and then placed into the TBUF.

3.3 Problem Statements

In addition to the implementation and evaluation of an NIDS, this work focuses on the impact of the processor, thread and memory bank allocations. Some problem statements are discussed below.

Task Allocation and Bottleneck Observation

Before implementing an NIDS, some functional blocks referred to as processing stages need to be identified and then mapped to the platform. During the mapping process, we try to exploit the hardware features such as the hierarchical memory structure and the multithreaded multiprocessor architecture. This involves mainly the assignment of memories to store different data structures, as well as the allocation of threads and MEs. After the system is implemented, we will try to identify possible bottlenecks through the internal and external benchmarks.

Effect of Improper ME/Thread Allocations

The performance of an application is affected by two factors, the computing power and the memory-access latency. The former is determined by the number of processors used referred to as *I*, while the latter can be alleviated by adjusting the total number of threads employed, namely $I \times J$ [LLP02], where *J* represents number of threads per processor. Observing that the number of processors is fixed to the hardware platform, it is interesting to see how an allocation (*I*, *J*), especially an improper one, affects the system performance.

Optimal *I* and *J*

It is known that memory-access intensive applications benefit directly from increasing the total number of threads, namely $I \times J$, rather than individual *I* and *J*, because of its ability of hiding memory-access latency. Nonetheless, how to determine a fitting $I \times J$, given a certain hardware spec such as clock rate and memory service rate, remains unanswered. In addition, we are also interested in finding an optimal (I, J) combination, regardless of the limit on the numbers of MEs and threads per ME of the platform. A (I, J) is considered optimal when the utilizations of both ME and memory are *cost-effectively* high, as will be explained in section 3.5.

Effectiveness of Employing Multiple Memory Banks

Multiple memory banks reduce the average memory access latency. For memory-access intensive applications, more memory banks are supposed to improve the performance. Nonetheless, the effectiveness could be influenced by whether the accesses are evenly distributed into memory banks. Some experiments are therefore designed to investigate the effectiveness of adding memory banks.

3.4 Design and Implementation

In this section, we introduce basic operations of an NIDS, characterize the operations into processing stages, and finally implement the NIDS by associating the MEs and threads to the stages. Some design issues are discussed to ensure proper inspections.

3.4.1 NIDS Briefing

The processing of an NIDS, for example, Snort [Roe], mainly consists of three phases (1) the packet decoding phase which sets up pointers to packet data at different layers and stores them into data structures for later analysis by the detection engine; (2) the detecting phase, in which a group of rules matched against a packet header are applied for further signature matching, and (3) the alert phase, in which some alert or logging routines are carried out. Although later versions of Snort include the preprocessing phase performing the IP de-fragmentation and TCP stream reassembly, it is optional and thus excluded in the implementation for simplicity.

3.4.2 Design Issues

According to the above-mentioned characteristic of an NIDS, it is clear that we can implement an NIDS over the IXP2400 by dividing the packet processing into a series of stages, namely the receiver, packet inspector and transmitter, and mapping them onto the MEs. The preprocessing phase is excluded in the mapping since oftentimes it is not done in the fast path [NSH02], but by the XScale. Moreover, packets can be distributed to a pool of MEs, and thus threads, in the packet inspector to exploit high parallelism. Nevertheless, two problems including *packet ordering* and *flow interleaving* arise.

Packet ordering

The issue of packet ordering occurs in a processing stage when multiple threads are dispatched to process the packets of a flow simultaneously. Oftentimes the amount of time to process a packet is not constant due to context switching, and thus the packet ordering may not be guaranteed, as shown in Fig. 3.2(a). To tackle this problem, a mechanism called *ordered threads* [JK03], is adopted requiring that threads handle packets in order in a processing stage of several functions, as presented in Fig. 3.2(b). For example, thread 1 is allowed to execute function 1 for a packet only after thread 0 completes the same function for another packet. When thread 0 completes function 1, it notifies thread 1 using inter-thread signaling. However, the effectiveness of multithreading could be greatly degraded if the function contains much memory accesses. The executing thread may not be able to context switch to other threads when performing memory accesses.

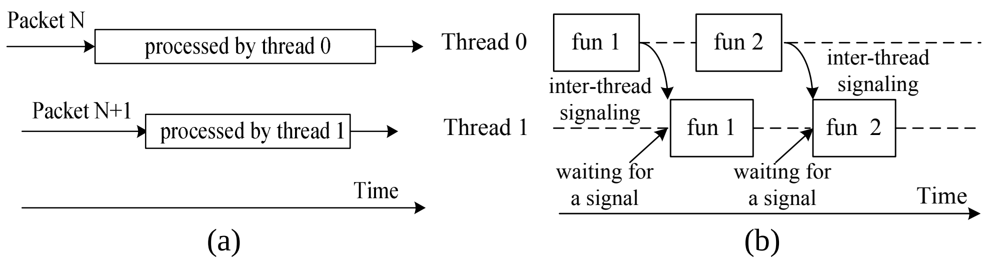


Fig. 3.2. Timeline showing two consecutive packets (a) being out of order, and (b) being ordered in a processing stage.

Flow interleaving

In packet inspection, a pattern may stretch across multiple packets. If flows are interleaved, it is not guaranteed that two consecutively processed packets belong to the same flow, meaning that patterns across multiple packets can not be inspected appropriately.

To fix these two problems, we refine our design by adding two processing stages, the *flow classifier* and *thread dispatcher*, supporting packet ordering. The main idea behind is to classify packets into different flow queues associated with a corresponding flow context, such that flows are no longer interleaved. The flow context comprises the SRAM address of the flow queue keeping the packet descriptors, state of inspection and some status flags. Further, each thread in the packet inspector stage is dispatched by the dispatcher to serve one flow queue. After finishing the inspection of a packet, the packet inspector thread stores the final state of the inspection for later reference by another thread serving the same queue. The implementation of the thread dispatcher will be detailed later in section 3.4.4.

3.4.3 Mapping Processing Stages to the Hardware Platform

Fig. 3.3 shows the processing stages of an NIDS, as well as the task and resource allocation for IXP2400. The NIDS processing is elaborated as follows. Upon receiving a packet from an input port, the packet data is moved from RBUF to DRAM; the corresponding packet descriptor is stored in SRAM while a duplicate is passed to the next stage through the receiving scratch ring. Subsequent the flow classifier retrieves a packet descriptor for flow classification which operates as following. First, the IP and port pairs in the packet are used to calculate a hash key for indexing in the hash table in SRAM in order to verify whether the flow which the packet belongs to exists. Since the task requires much computing power, the hash unit is adopted to offload the overhead. If a hash hit occurs, the hash entry pointing to a flow context in SRAM is referred to enqueue the packet descriptor for inspection; otherwise an entry for the new flow is created in the hash table.

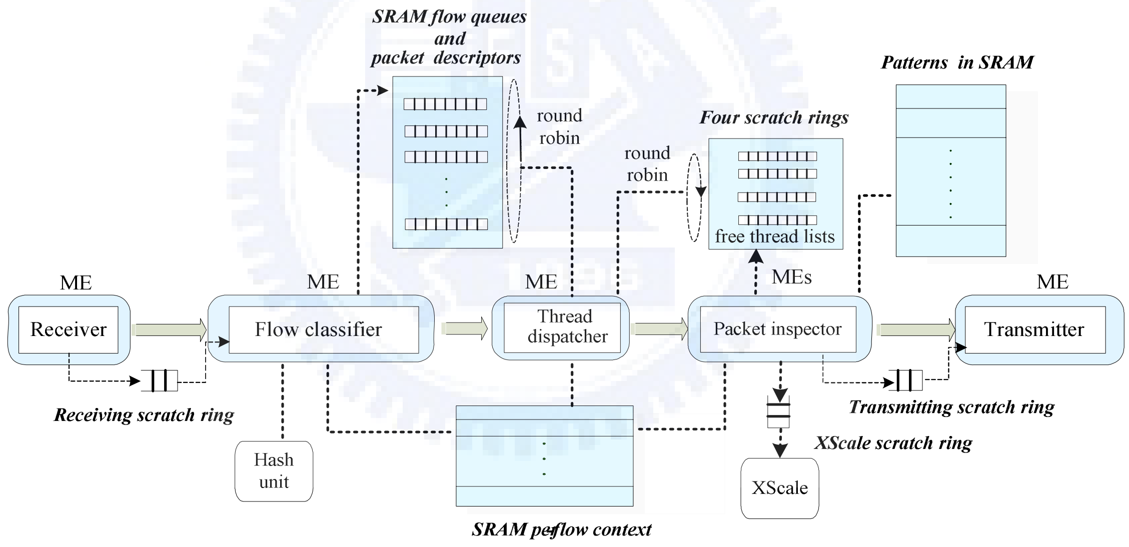


Fig. 3.3. The processing stages of an NIDS on IXP2400.

The dispatcher thread then round-robinly chooses a flow queue and dispatch an inspector thread to handle the first packet in the queue. Once a packet payload is matched against a pattern, a message is delivered to the XScale through the XScale scratch ring to signal an alert. Finally, the transmitter thread examines the transmitting scratch ring to determine whether a packet is waiting to be sent. If yes, it fetches the packet descriptor in SRAM and sends the entire packet in

DRAM to TBUF for output.

In our implementation, a tentative allocation of MEs and threads is determined based on the processing stages and the benchmark result of Snort, which argues that at least 31% of total processing time is consumed by the detecting phase [FV02]. So, each processing stage is allocated one ME except the packet inspector, which is given four MEs. That gives us totally four MEs, namely thirty-two threads for later adjustment and analysis. For thread allocation in the receiver, eight threads are evenly divided into four groups corresponding to four gigabit ports. Each port is served by two ordered threads to keep packets in order. As for the transmitter, eight ordered threads are assigned to one gigabit port. We adopt eight ordered threads in both classifier and dispatcher stages for the following two reasons leading to out-of-order packets: (1) classifying packets could take vastly different amount of time due to hash collisions, and (2) serving flow queues round-robinly needs that the round-robin counter be accessed by one thread at a time. In the packet inspector, it is manipulated that a flow queue is served by a thread at ay instance, in which ordinary thread scheduling mechanism,, rather than the ordered thread, is employed for better benefit from multithreading. Since a flow queue is served by one thread at a time, packets of a flow will never get out of order. Interaction between the thread dispatcher and packet inspector will be detailed in section 3.4.4.

3.4.4 Algorithms Adopted and Packet Inspection

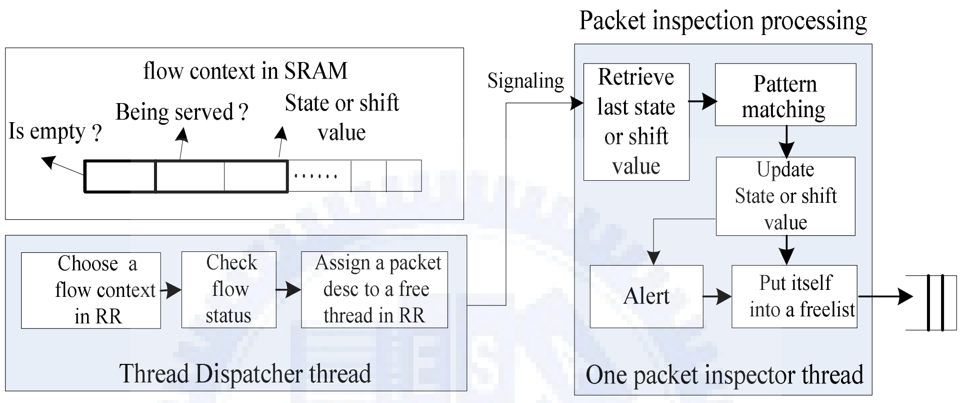
3.4.4.1 String Matching Algorithms

Packet inspection is a critical stage that influences the performance of an NIDS. Several string matching algorithms were proposed for improvement. However, coding microcode is difficult, since it depends heavily on the hardware characteristics. Two popular algorithms, Aho-Corasick referred to as A-C and Wu-Manber referred to as W-M, are thus used because they are easy to implement and popular in many applications such as Snort. The two algorithms consist of two common phases: a pre-processing phase, which computes and builds necessary data structures in memory from the input patterns, and an inspection phase, in which patterns are looked up against the packet payload. Nevertheless, the pre-processing phase is time-consuming and typically done by the XSacle. In our implementation, we store the data structures in SRAM for fast retrieval. Since the operation of the A-C involves state transitions, we record the final state immediately after the processing of a packet for later inspection of the succeeding packet in the same flow queue. Similarly, we keep the shift value for the W-M so that patterns across multiple packets can be inspected.

3.4.4.2 Thread Dispatcher and Packet Inspector

Fig. 3.4 details the interactions between thread dispatcher and packet inspector. As mentioned in section 3.4.3, a flow queue is round-robinly selected and the first packet descriptor in that flow is passed to an inspector thread chosen from the free thread list of the ME. This process involves some operations. First, two flags, *isEmpty* and *beingServed*, of a flow context are checked in each round. The former indicates if the corresponding flow is empty while the latter denotes whether that flow is being served by a thread. If the flow is not empty and not being served, a packet descriptor is assigned to an inspector thread followed by the corresponding modifications of the two flags. This ensures that a flow is served by only one inspector thread at a time, by which preventing the state (for the A-C) or shift value (for the W-M) from being altered by other threads. The inspector thread then examines a packet payload against the patterns in SRAM and updates accordingly the state or shift value in the flow context. If no pattern is matched, the packet is passed to the transmitter thread to be sent out; otherwise the XScale is notified of a match. Finally, the packet inspector thread puts itself into the free thread list, waiting for the next signal from the dispatcher. The four free thread lists implemented using four scratch rings correspond to the four MEs.

The inspector threads are dispatched round-robinly among the MEs for better load balancing. To avoid the system resource being exhausted by excess idle flows, a timeout counter maintained by the XSacle is associated with each flow. Once the counter turns to zero, the flow queue as well as the flow context and hash entry are removed.



Packet inspection processing
flow context in SRAM
Being served ?
State or shift value
Is empty ?
……
Signaling
Retrieve last state or shift value
Pattern matching
Update State or shift value
Alert
Put itself into a freelist
Choose a flow context in RR
Check flow status
Assign a packet desc to a free thread in RR
Thread Dispatcher thread
One packet inspector thread

Fig. 3.4. Interaction between the thread dispatcher and packet inspector.

3.5 System Benchmark and Bottleneck Analysis

In this section, we evaluate the performance by externally and internally benchmarking the system implemented using two string matching algorithms. To have both MEs and memory, namely SRAM, well utilized, we investigate the appropriate numbers of *I* and *J* for the application. Since the memory access overhead accounts for a considerable portion in the packet processing, the feasibility of exploiting multiple memory banks for load balance is exploited.

3.5.1 Benchmark Setup

The XScale core in our design is responsible simply for the preprocessing and alerting; therefore, in this section we focus mainly on the performance of the MEs which are the main component that handles the most part of packet processing.

Since the performance statistics including the ME and memory utilizations can only be obtained by the simulator, we evaluate the performance through simulations. The preprocessing phase originally done by the XScale is shifted to the receiver ME since the simulator does not comprise the XScale. Notably two MEs from two processing stages, the flow classifier and thread dispatcher, respectively, are borrowed in the analysis due to the dearth of MEs.

3.5.1.1 Patterns for Packet Inspection

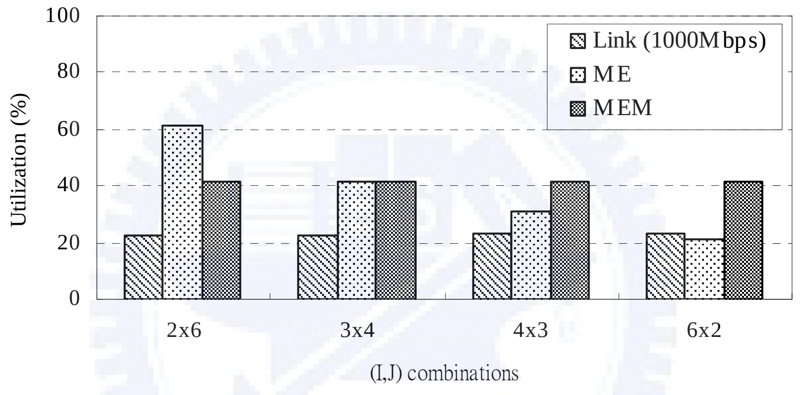
Observing that 2475 patterns are used in the current Snort, we employ 2000 random patterns in which characters are generated uniformly according to the guidelines discovered in [AAP04]. The *shortest pattern length*, LSP, which is known as a major factor on the performance of string matching algorithms such as W-M, is set to four [LHC04].

3.5.1.2 Simulator Setup

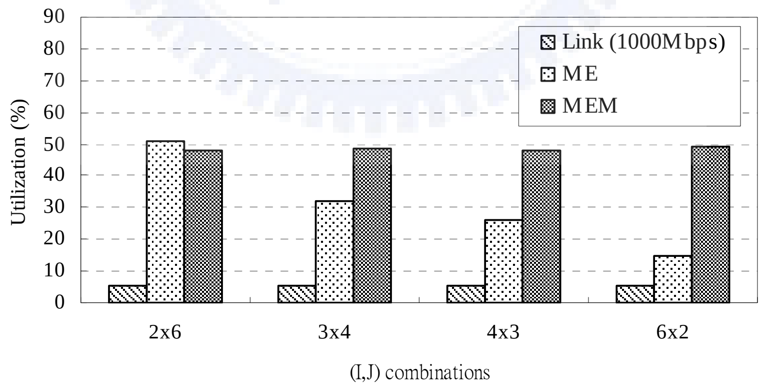
The IXP2400 Developer Workbench simulator provides tools for compiling the microC into microcode and a simulator called Transactor, for evaluating the performance. The simulator allows users to configure parameters. In our experiment, the clock of the ME is 600 MHz. The input interface of the MSF is divided into four gigabit ports, while the output interface is a four-gigabit one. The transmitter and receiver buffers are both 256 bytes. Four data streams of 64-byte TCP/IP packets with randomly generated payload are injected. All simulations last for 50000 packets.

3.5.2 Effect of Improper ME/Thread Allocations

To investigate the effect of improper ME/thread allocations, we compare the performance, in terms of utilization, of the A-C for different (I,J) combinations. As shown in Fig. 3.5(a), I and J can be configured while the total number of threads, $I \times J$, is fixed to 12. Some observations are made. First, the throughput is influenced mostly by $I \times J$, rather than *I*, as the throughput remains unchanged for the (*I*,*J*) combinations. Second, the average ME utilization degrades while increasing *I*. This is because the same traffic load is balanced by more MEs. The same explanation applies to the results of the W-M in Fig. 3.5(b). Third, the throughput of the W-M is only one-fourth of the one of A-C. This is due to the relatively high processing overhead of the W-M, as clarified in Fig. 6.



(a)



(b)

Fig. 3.5. Performance of the (a) A-C and (b) W-M for different (*I*,*J*) combinations. Total number of threads is fixed at 12.

Figure 3.6 profiles the total memory-access cycles referred to as *P*, as well as the computational cycles referred to as *M*, required by the A-C and W-M for handling a 64-byte packet. From the figure we can see that the sum of *P* and *M* of the W-M is approximately 4 times of the one of A-C. This explains the relative low throughput of the W-M compared with the A-C. Further, the memory access overhead dominates the processing time of a packet, namely 94% for A-C and 98% for W-M. Fortunately this un-balance situation is tolerated by multithreading, which makes the utilizations of MEs and memory much closer to each other than what otherwise will be.

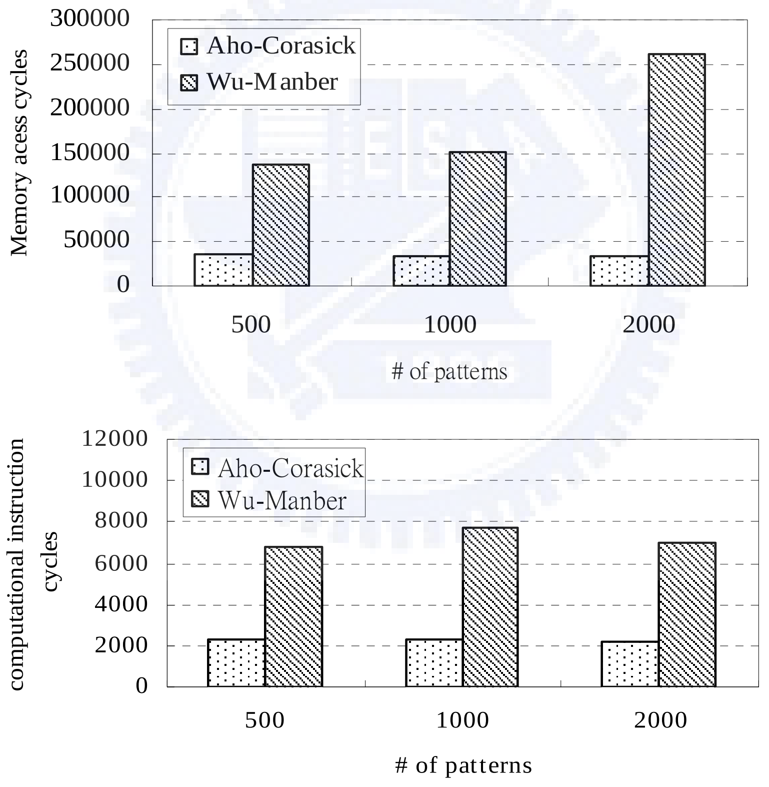


Fig. 3.6. Profiling of the total (a) memory access cycles and (b) computational cycles for processing a 64-byte packet.

3.5.3 Estimating the Optimal (*I*,*J*) Pair

Figure 3.7 depicts the performance of the two implementations by increasing number of MEs and therefore the total number of threads. Some observations can be made. First, the throughput of A-C is better due to less computational and memory-access overhead. Second, for number of MEs being from one to four, the ME utilizations of both implementations are almost the same, implying that the number of threads per ME is insufficient. Third, initially, the throughputs of both implementations increase with a direct ratio to $I \times J$. Nevertheless, the throughput increases slightly as *I* = 5 for W-M and *I* = 6 for A-C, respectively, because memory is almost fully utilized. Fourth, as *I* increases and memory utilization approaches 90%, the average ME utilization degrades, because the load making memory saturated is diluted by large *I*.

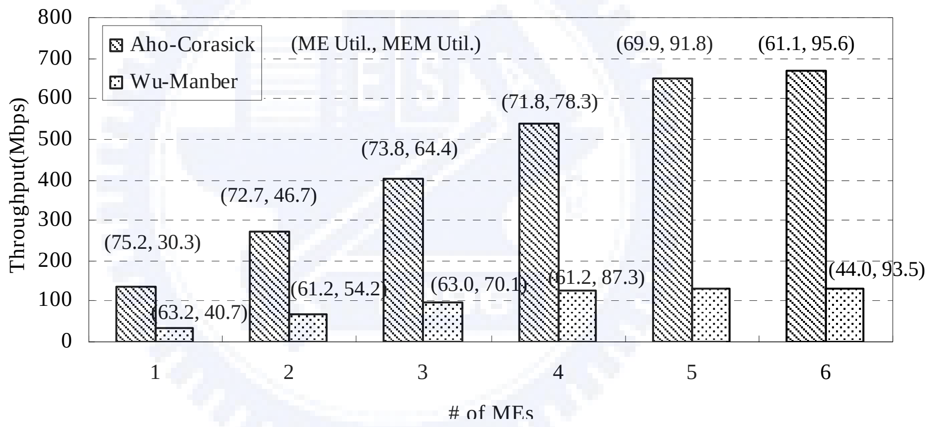


Fig. 3.7. The performance of A-C and W-M with different numbers of MEs (eight threads per ME).

We can also estimate a combination of (*I*,*J*) such that both ME and memory are best utilized. As we learn from Fig. 3.7, when memory utilization is above 90%, increasing *I*, and therefore total number of threads contributes slightly to the performance and is not cost-effective. For example, the improvement of memory utilization from incorporating the sixth processor is about $95.6-91.8 \approx 3.8\%$ and $93.5-91.9 \approx 1.6\%$ for A-C and W-M, respectively. Hence, $5 \times 8 = 40$ threads should be cost-effectively enough for both algorithms to well utilize the memory. Nonetheless, the ME utilization is low when *I* = 5, meaning that the computing power is unnecessarily much and should be further reduced. We fix this problem by employing four MEs, rather than five, so that the average utilization of MEs shall become $\frac{69.9\%\times 5}{4} \cong 87.4\%$ (since $\frac{69.9\%\times 5}{3} \cong 116.5\% > 100\%$), and J can thus be estimated to $\frac{40}{4} = 10$. Similarly, a combination of $(3,13)$ can be derived for the W-M.

3.5.4 Effectiveness of Multiple Memory Banks

One of the solutions to the memory bottleneck is to add more memory banks. To evaluate the benefit, we adopt two SRAM banks to store the data structures of the string matching algorithms. Table 3.1(a) shows that only minor improvement can be gained due to the difficulty of splitting the data structure, namely *goto* table, of A-C evenly into different memory banks. The W-M, on the contrary, benefits substantially (about 43.7%) from two banks as presented in Table 3.1(b). This is credited to the use of several tables which make the distribution of data a lot easier and more efficient to memory banks.

Table. 3.1. Performance of (a) A-C and (b) W-M with one and two memory banks, respectively. (I,J) = (6,8).

(a)

	One memory bank	Two memory banks
Avg. ME util. (%)	61.1	63.2
MEM util. (%)	95.6	95.2/1.8
Throughput (Mbps)	670.6	674.4

(b)

	One memory bank	Two memory banks
Avg. ME util. (%)	44.0	63.2
MEM util. (%)	93.5	70.0/57.2
Throughput (Mbps)	133.2	191.4

3.6 Summary

In this work, we elaborate the implementation of a memory-access intensive application, NIDS, over the IXP2400 network processor. We introduce the hardware platform, briefing the NIDS processing flow, and identify necessary processing stages to be mapped to the platform. Among those processing stages, the packet inspection is implemented with the Aho-Corasick and the Wu-Manber algorithm. Some design issues including packet ordering and flow interleaving, which may cause incorrect inspection results for patterns across multiple packets, are discussed and solved. After implementation, we externally and internally benchmark the system aiming to observe the effect of the allocations of processors, threads, and memory banks, as well as possible bottlenecks.

The benchmark result shows that the system can support up to 670 Mbps when implemented using the Aho-Corasick and 133Mbps using the Wu-Manber. It is also observed that given a certain application and algorithm, the throughput is influenced mostly by the total number of threads as long as the ME utilizations do not exceed 100%. Although enlarging the total number of threads by adding more processors benefits the throughput, the ME utilization suffers. This is because the load saturating memory is diluted by the increased *I*, meaning that *J* instead should be extended.

The bottleneck is then found to be the SRAM as the $I \times J$ exceeds the upperbound *k* that *cost-effectively* utilizes the memory. With the upper-bound, we can estimate an optimal (*I*, *J*) combination, i.e. (4, 10) for the Aho-Corasic and (3, 13) for the Wu-Manber, respectively. In fact, supposed an application, algorithm and *k*, an optimal (*I*, *J*) can always be derived. Two workarounds are suggested to solve the SRAM bottleneck, namely when $I \times J > k$. The first is to use multiple memory banks. Our result indicates that the performance gains a 43.7% improvement from two banks for Wu-Manber since the data structure itself makes it easy to be evenly distributed among banks. The other is to adopt a multi-port memory which allows multiple simultaneous memory accesses. This is helpful especially to algorithms, such as the Aho-Corasick, having data structures difficult to be uniformly split.

Two issues are to be investigated in the future. First, real traffic, rather than the synthetic one, should be adopted. The second is to investigate the allocation measures for computational-intensive applications.

Chapter 4

Coprocessors-centric Network Processors: Analysis, Simulation, and Design Implications

4.1 Introduction

In this work, we aim to unveil possible hints, especially the thread allocation, for future NP design in two directions: (1) develop a preliminary analytical model using the Continuous Time Markov Chain, and (2) build a Petri net simulation environment which is also used for model validation. Our approach considered both memory and ready queuing effects that are often ignored in other works, and involves two important networking applications, Simple Forwarding and DiffServ, which have different computational and memory access requirements. We propose a concept named P-M ratio and discover that a large *I* , or *J*, is needed for high, or low, P-M ratio, and further that when processor overhead (P) is similar to the memory's (M), the most appropriate number of threads is shown to be 5. Notably the core processor was not included in our model since the control-plane processing accounts for only a minor portion in the packet processing.

Another concern in our approach is the selection of a thread allocation scheme. Thread allocation schemes decide how threads in a processor are arranged for processing packets; adopting an improper scheme could result in un-balanced load distribution among processors. We compared and discussed four possible allocation schemes, and chose the most appropriate one as the base assumption throughout this work. Factors influencing the selection include the amount of hardware resources, design complexity, and flexibility in processing.

The rest of this article is organized as follows. Section 4.2 introduces the concept of thread allocation schemes. Section 4.3 elaborates the analytical model. Section 4.4 details the construction of the Petri net simulation environment, validates our analytical model, and presents some interesting simulation results. Conclusive remarks and future work are given and discussed in section 4.5.

4.2 Effect of Different Thread Allocation Schemes

Thread allocations should be carefully involved and studied before analyzing the M-M architecture. Four thread allocation schemes are common in real implementations, in which at most one thread is active in a processor. The first is that a thread is assigned to process a complete packet. Nonetheless, this scheme may require intricate inter-thread communications in order to maintain the packet ordering in a flow.

Another two schemes, which are shown in Fig. 4.1, are called homogeneous and heterogeneous thread allocations, respectively. In the homogeneous allocation, all threads in a processor belong to the same type, e.g., receiver, scheduler, transmitter, etc. Each thread in a processor deals with only part of the packet processing and after that, it signals a certain thread in the succeeding processor for further processing. A thread in a processor may have either fixed or dynamic task assignment, namely it may stick to a certain input port or it may support other ports whenever necessary. Notably, since all threads in a processor are of the same type, this scheme has a more relaxed requirement for the size of the instruction memory while exhibiting desirable data locality in cache. Nonetheless, in the homogeneous scheme, processing load can hardly be distributed to processors evenly, and packet ordering is unlikely to be maintained.

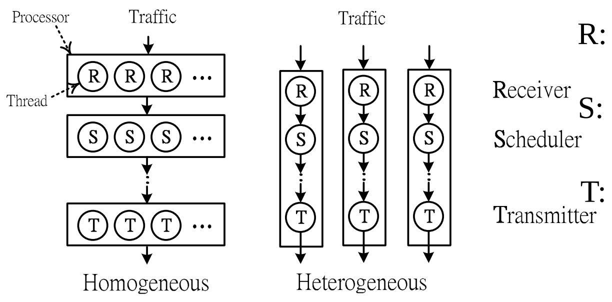


Fig. 4.1. Homogeneous and heterogeneous thread allocations. At most one thread is active per processor.

This situation can be avoided with the heterogeneous allocation, where the traffic can be assigned to a processor with a lighter load by some load-balancing hardware and mechanisms [BDE01]. In this scheme, each thread in a processor belongs to different types and is supposed to take charge of an equal overhead in the packet processing. The requirement for a larger instruction memory will not be a problem because less than 5K of it is needed by general header processing applications [RW03], and that requirement has already been supported in many commercial products such as the Intel IXP2400 and Motorola C-5 [MOT]. Another edge of the scheme is the minor synchronization overhead, since the inter-thread communication is done using global registers in the processor. A comparison between these two strategies is shown in Table 4.1. For the reasons discussed above, we take the heterogeneous allocation as the basic assumption in our model throughout this work.

Table 4.1. Comparison between the homogeneous and heterogeneous schemes.

Allocation strategy	**Threads in a processor**	**Packet processing**	**Instruction memory**	**Data locality**	**Load balancing**	**Sync. overhead**
Homogeneous	Same type	Partially	Small	High	Hard	High
Heterogeneous	Diff. types	Completely	Large	Low	Easy	Low

It is also possible to use the *hybrid* allocation scheme, in which processors of homogeneous or heterogeneous allocations are incorporated. This scheme preserves the strength of large instruction memory and high data locality, which can be achieved by assigning homogeneous processors to tasks exhibiting high data locality. However, the load balancing and packet ordering originally supported by the heterogeneous scheme no longer exist.

4.3 Overview of the Analytical Model

In this section we present an approximate analysis of the multithreaded multiprocessor network processor using a Continuous Time Markov chain. We define the state space of the model, derive the transition rates and solve the model. In addition to the heterogeneous allocation determined in the previous section, we proceed with the assumption of blocking processing, as shown in Fig. 4.2. The blocking processing contrasts with the non-blocking processing, which is also shown in Fig. 4.2 in that no buffer exists between two adjacent threads of a processor. That is, a thread cannot pass the processing result to its successor and accept another packet if the successor is busy with a packet. Since normally the packet processing overhead, including computation and memory access, is fairly distributed among threads, this simplified assumption has limited influence on the correctness of the model while considerably reducing the state space.

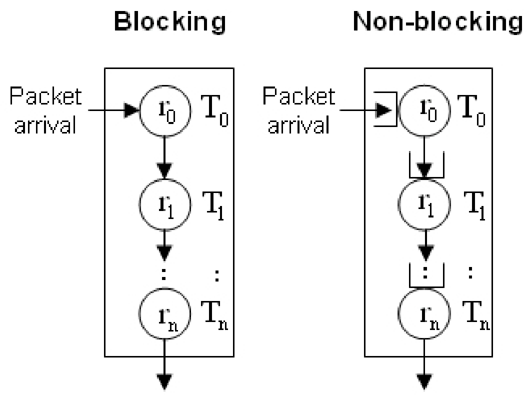


Fig. 4.2. The blocking and non-blocking packet processing schemes. A thread T_t accesses memory with rate r_t during the processing.

4.4 Markov Chain Formalization

4.4.1 State Definition and State Space Determination

Our model considers *I* processors, each of which contains *J* threads, and aims to characterize the behaviors of processors, threads and memory. To do that, we need to clarify possible activities, i.e. *statuses transitions*, of a thread. They are depicted in Fig. 4.3 and elaborated below. When a packet arrives at an *idle* thread, the thread either enters the *ready* queue of the processor waiting for execution, or enters the *active* status if no thread is currently *active*. Sometimes it issues a *memory access* to, for instance, perform table lookups and manipulate packet descriptors. Once serviced it re-enters the ready queue, or goes directly back to execution if the ready queue is empty. Normally, the thread becomes idle again after the packet is processed and passed to the succeeding thread. Nonetheless, it may get stuck and enter the *finished* status if the succeeding thread also has a packet under processing.

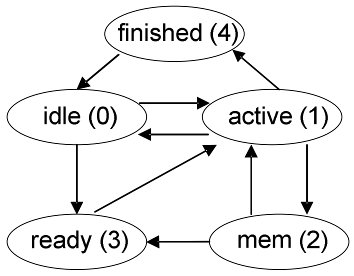


Fig. 4.3. Status transitions of a thread.

According to the above descriptions we can formally define a state of the system as

$$S = (s_{0,0} \ldots s_{0,j} \ldots s_{i,j}),\ 0 \le i < I \text{ and } 0 \le j < J\ ,$$

where $s_{i,j} \in \{0: idle, 1: active, 2: mem, 3: ready, 4: finished\}$ represents the status of $T_{i,j}$, the *j*th thread in processor *i*. Furthermore we define $S(k) = \{s_{i,j} \mid s_{i,j} = k\}$, so that the number of executing processors and number of accesses in the memory system equal to $|S(1)|$ and $|S(2)|$, respectively. We also define $h(i) = \{s_{i,j} \mid s_{i,j} = 2\}$ so that the number of queued memory accesses of processor *i* is denoted by $|h(i)|$. Besides, the *RSS* (Random Selection for Service), rather than the FIFO, is assumed as the queuing discipline for both memory and ready queues. This assumption further diminishes the state space by disregarding the ordering information in the queues, and is proven not to affect the correctness of the analytical result in section 4.5. Taking (*I*,*J*)=(2,2) as an example, the state space can be derived by excluding exceptional states exhibiting the following properties:

1. A processor has more than one active thread. For instance, $(1,1,0,0)$.
2. At least one ready thread but no active thread, such as $(2,3,0,0)$. One of the ready threads must enter the active status as long as the previous active thread completes its processing.
3. $s_{i,j} = 4$ while $s_{i,j+1} = 0,\ 0 \le j < J$. In this case, $T_{i,j}$ must pass the packet

immediately to the succeeding one.

4. $s_{i,J-1} = 4$; the same reason as the one in 3.

4.4.2 Determination of the Status Transition Diagram and State Transition Matrix

We will need the state transition matrix in order to solve the model. To derive the matrix, however, we have to deal with the status transition rate diagram of threads since a state change occurs when one or more threads alter its status. By assuming the packet arrival rate for processor *i* as λ_i, memory access rate and service time of the *j*th thread in that processor as $r_{i,j}$ and $1/\mu_{i,j}$, memory service rate as *m*, and number of queued memory accesses from the processor as *h*, we can have the status transition rate diagram shown in Fig. 4.4. Notably the service rates, as well as the memory access rates, of threads having same thread index in all processors are set the same because of the homogeneity among those threads. That is, $\mu_{i,j} = \mu_j$ and $r_{i,j} = r_j$.

Notice that some status transitions in Fig. 4.4 do not have a rate because of being a *follower* transition. A transition is regarded as a follower if it does not initiate a status transition but follow a certain *activator* transition which actively launches a transition. For example, a finished thread (follower) blocked by its successor can enter the idle status only after the successor (activator) finishes processing and passes down the packet. Another example is that a ready thread (follower) will never enter the active status unless a thread switches out from active.

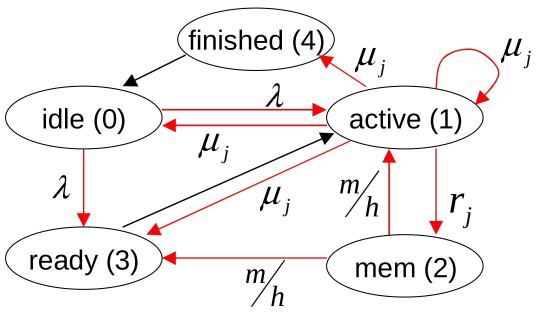


Fig. 4.4. Status transition rate diagram of $T_{i,j}$.

Observing the relationship between activator and follower, two additional transitions can be discovered out of Fig. 4.3 and shown in Fig. 4.4, the active to active and active to ready transitions. The former occurs when an active thread switches out and is then chosen again to execute for the packet passed by its finished predecessor; the latter is similar except that it is not chosen for execution but put into the ready queue.

The state transitions and transition matrix can therefore be determined according to the status transition diagram. More specifically, a state transition is considered valid if there exists only one *activation event* containing an activator transition and possibly a number of corresponding follower transitions. Figure 4.5 shows four example state transitions, assuming (*I*,*J*)=(1,6). The detailed matrix derivation is described in the following section.

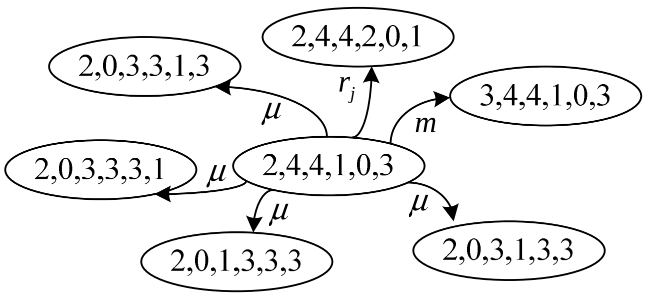


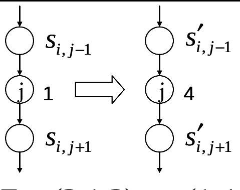
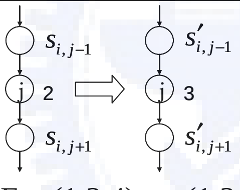
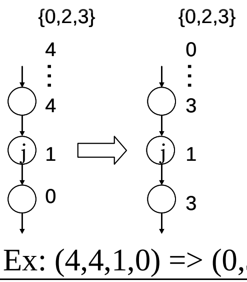
Fig. 4.5. Example state transitions.

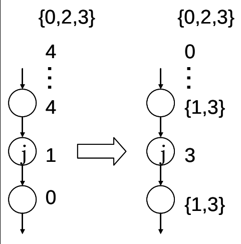
4.4.3 Determination of the State Transition Matrix

A state transition of a non-zero rate consists of one activation event containing an activator transition and possibly a number of corresponding follower transitions. To verify a state transition, we need to characterize the activation event, namely the activator and follower transitions. Obviously, a transition initiated by a thread in the active(1) or the memory access(2) status is always an activator transition, whereas a transition performed by a thread in the idle(0), ready(3) or finished(4) status is a follower transition with two exceptions. The exceptions occur when the transitioning thread is the first one in a processor, in which idle-to-active or idle-to-ready transitions are possible because of the packet arrival.

With the observations above and the conditions defining the status of threads other than the activator thread, all activation events can be identified as summarized in Table 4.2. An activation event is considered valid if the corresponding conditions of the activator transition are satisfied. For instance, before recognizing an activation event with the activator transition being from active to finished, namely the thread is finishing the processing of a packet but getting blocked by its successor, two conditions need to be met. First, $j < J - 1$ and $s_{i,j+1} \in \{2,3,4\}$, since if *j* equals *J*-1 or $s_{i,j+1} = 0$, the thread would have been able to send out the packet. Second, for threads other than $T_{i,j}$ in processor *i*, their statuses remain unchanged if none of them is in the ready status; otherwise one thread shall be chosen for execution. Take (*I*,*J*)=(1,3) as an example, the activation events (2,2,1)=>(2,2,4), (2,1,0)=>(2,4,0), and (3,1,2)=>(3,4,2) are all invalid.

Table 4.2. Activation events initiated by $T_{i,j}$, and the corresponding examples ($I=1, J\in\{3,4\}$) and conditions. $s_{i,j}$ and $s'_{i,j}$ denote the source and destination status of $T_{i,j}$, respectively. The status transition rates are shown in Fig. 4.4.

Activator	Example	Condition
act(1) => fin(4)	 $s_{i,j-1}$ j 1 $s_{i,j+1}$ $s'_{i,j-1}$ j 4 $s'_{i,j+1}$ Ex: (3,1,2) => (1,4,2)	1. $j<J-1, s_{i,j+1}\in\{2,3,4\}$ 2. if ($^{\forall} j'\neq j, s_{i,j'}\neq 3$) then $s_{i,j'}=s_{i,j'}$ else $^{\exists!} j'\neq j \ni s_{i,j'}=3,\ s'_{i,j'}=1$
act(1) => mem(2)	Ex: (3,1,2) => (1,2,2)	The same with (2) in 1=>4.
act(1) => idle(0)	Ex: (3,1,0) => (3,0,4)	1. if $j<J-1$ then $s_{i,j+1}=0, s'_{i,j+1}\in\{1,3\}$ 2. if $j>0$ then $s_{i,j-1}\neq 4, s'_{i,j-1}\neq 4$ 3. The same with (2) in 1=>4 except $j'\notin\{j,j+1\}$.
mem(2) => rdy(3)	 $s_{i,j-1}$ j 2 $s_{i,j+1}$ $s'_{i,j-1}$ j 3 $s'_{i,j+1}$ Ex: (1,2,4) => (1,3,4)	1. $^{\forall} j'\neq j, s_{i,j'}=s'_{i,j'}$ 2. There exists an active thread.
mem(2) => act(1)	Ex: (2,2,4) => (2,1,4)	$^{\forall} j'\neq j, s_{i,j'}=s'_{i,j'}$
idle(0) => act(1)	Ex: (0,2,4) => (1,2,4)	$j=0;\ ^{\forall} j'\neq 0, s_{i,j'}=s'_{i,j'}$
idle(0) => rdy(3)	Ex: (0,1,4) => (3,1,4)	$j=0;\ ^{\forall} j'\neq 0, s_{i,j'}=s'_{i,j'}$
act(1) => act(1)	 {0,2,3} 4 4 j 1 0 {0,2,3} 0 3 j 1 3 Ex: (4,4,1,0) => (0,3,1,3)	1. $s_{i,j-1}=4,\ s_{i,j+1}=0,\ s'_{i,j+1}=3$ 2. while n>0 { # *n*=*j* -1 $s'_{i,n}=f(s_{i,n-1})$, where $f(4)=3, f(0)=f(2)=f(3)=0$ if $s'_{i,n}=0$ then break else $n=n-1$ }

act(1) => rdy(3)	 {0,2,3} 4 4 j 1 0 {0,2,3} 0 {1,3} j 3 {1,3} Ex: (4,4,1,0) => (0,1,3,3)	1. $j > 0, s_{i,j-1} = 4$ 2. if $j = 1$ then $s'_{i,j-1} = 0$ 3. if $j < J - 1$ then $s_{i,j+1} = 0, s'_{i,j+1} \in \{1,3\}$ 4. The same with 1=>1 except $f(4) \in \{1,3\}$.

4.4.4 Performance Estimation for the Analytical Model

The performance metrics that we are interested in obtaining from the analytical model include the processor and memory efficiencies. We can compute these measures from the stationary probability vector, π, for the Markov chain. The mean number of executing processors, which we call processing power (P_{power}), and the processor utilization, which we call processor efficiency ($P_{efficiency}$), are then calculated from the vector as

$$P_{power} = \sum_{S} (\pi(S) \times |S(1)|) \quad \text{, and} \tag{1}$$

$$P_{efficiency} = P_{power} / I \ . \tag{2}$$

Memory utilization, which we call memory efficiency ($M_{efficiency}$), number of memory accesses in memory system ($M_{accesses}$), and ready queue length of a processor (R_{length}) can be calculated as

$$M_{efficiency} = \sum_{S:\exists s_{i,j}=2} \pi(S) \, , \tag{3}$$

$$M_{access} = \sum_{S} \pi(S) \times |S(2)| \text{, and} \tag{4}$$

$$R_{length} = \left(\sum_{S} \pi(S) \times |S(3)| \right) / I \; . \qquad (5)$$

4.5 Simulation and Analytical Model validation

In this section, we describe the construction of a simulation environment based on timed, colored Petri nets (CPNs) [Mur89]. It is used to validate the analytical model discussed in the previous section as well as to observe possible hints for future NP design.

4.5.1 Design of the Petri Net Based Simulation Environment

The key challenge in simulating memory queuing effect is that an outgoing memory access must go back to the thread where it is issued. For that purpose, we adopt the event-driven CPN-Tools [RWL+03] as our simulator. The features it supports, including the colored tokens, stochastic functions and hierarchical editing, provide efficiency in the construction of timed, colored Petri nets corresponding to our model. To give a general idea of the design of the Petri net based model, we use an example whose configuration of (*I*,*J*) is (1,2) shown in Fig. 4.6. Simulations for larger *I* and *J* are constructed in a similar way.

The sample Petri net implements the processor and memory subsystems shown in Fig 4.6(a) and 4.6(b), respectively, and works as following. A token is added in places such as the P0_token (for processor 0), TK0_0 and TK0_1 (for thread 0 and 1), Pkt_Gen0 (for packet generator), and Init (for memory). Among those tokens the one in Pkt_Gen0 is designed to be a colored token, which represents a packet and carries information about the processor index (*i*), thread index (*j*), and the number of memory accesses (*k*) the thread is obligated to perform to process the packet. The tokens of the others are simply non-colored ones.

In the processor subsystem, the inter-arrival time of packets is exponentially distributed with mean *E* using the function *expDelay*, and the availability of a thread depends on whether a token is in places of the processor and thread. When a packet arrives at B0_0, namely a colored token is fired by the transition Delay0, and if there is a token in both P0_token and TK0_0, the packet is admitted by consuming those three tokens and firing the transition Tran0_0_0. After that, the packet is processed for *P/J* computation cycles (active state) and *M/J* memory accesses are assigned to the thread by setting *k*= *M/J*, where *P* and *M* denote the numbers of computational instructions and memory accesses required to process a packet, respectively. The CPI is assumed to be 1.

The memory access takes place by firing transitions Tran0_0_1 and S1, and then enters the queue (M_buf) of the memory subsystem and gets serviced if no other access is present. After a service time of *L* cycles (memory access state), the packet is passed back to the place T0_0 where it is issued according to the *i* and *j* in the token. The same procedure executes repeatedly until *k* becomes 0. The packet is passed to B0_1, waiting to be admitted by the next thread where operations similar to the above are carried out before leaving the system.

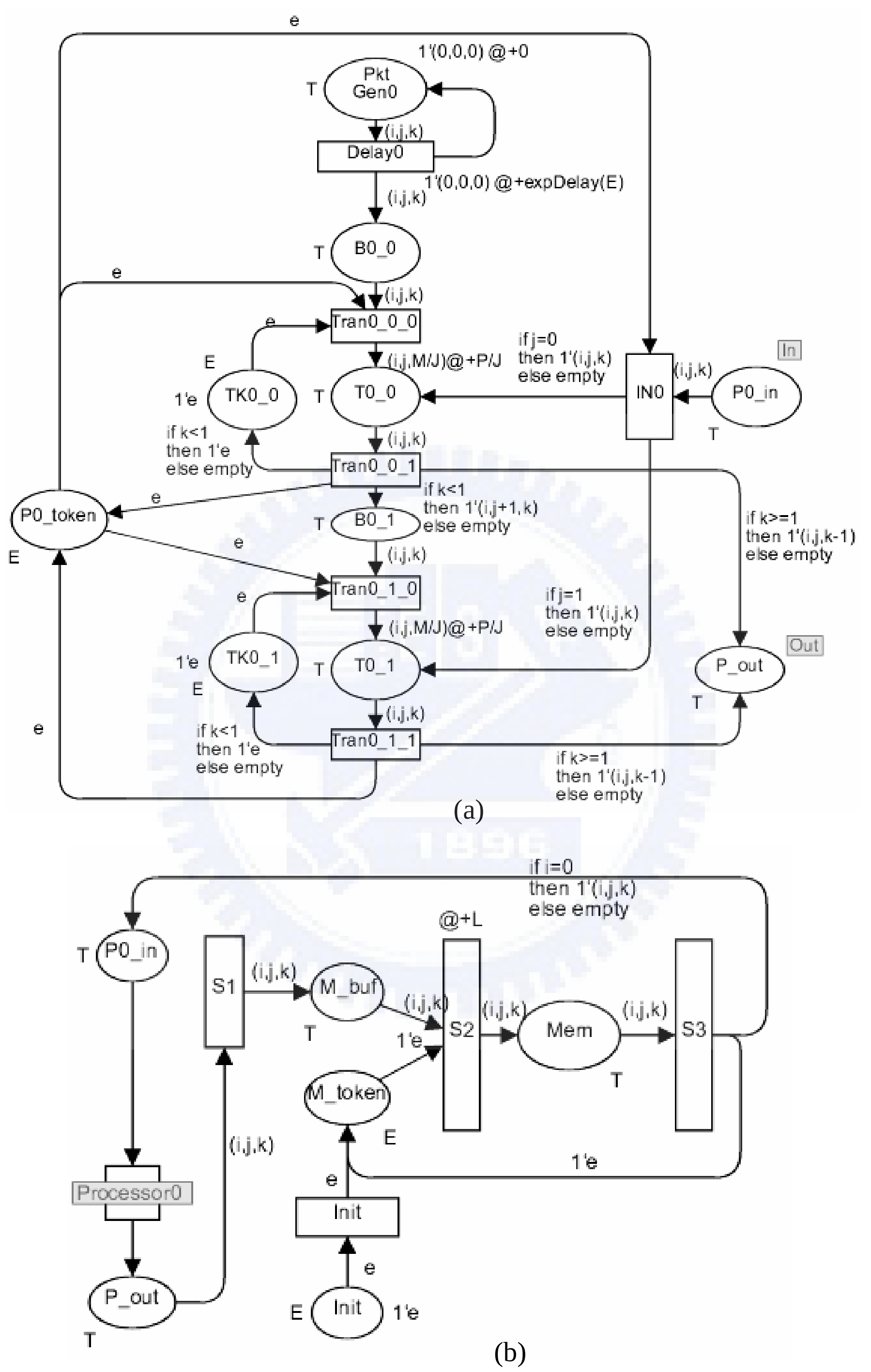


Fig. 4.6. An example hierarchical CPN describing (a) a processor containing two threads, and (b) the memory subsystem.

The simulation design differs from the analytical model in that the memory access rate and thread service rate are fixed according to the requirement of the application. The memory queue not shown in the above example is implemented in the M_buf using utilities of the CPN-Tools.

4.5.2 Model Validation By the Simulation

The analytical model is validated by simulations. Parameters for the analytical model as well as the simulation are listed in Table 4.3.

Table 4.3. The setup of parameters setup in the model validation. *P*=555 and *M*=30, and the system clock rate is denoted by *C*.

	Simulation	**Analysis**
Packet arrival	*E* = 7300 (cyc/pkt)	$\lambda = C \times \frac{1}{E}$ (pkt/sec)
Instruction processing capability of a thread	*P/J* (cyc/pkt)	$\mu_i = C \times J/P$ (pkt/sec)
Memory access intensity of a thread	*M/J* (acc/pkt)	$r_i = \mu_i \times \frac{M}{J}$ (acc/sec)
Memory service time	*L*=90 (cyc/acc)	$m = C \times \frac{1}{L}$ (acc/sec)

Our first observation is that, as presented in Table 4.4, the analytical results are mostly within 10% of the blocking simulation results. The discrepancy comes from the different assumptions between the model and the simulation. The former assumes non-deterministic behaviors in the instruction processing, memory access rate and memory service time, while the latter uses deterministic ones. In fact, the discrepancy can be reduced to be less than 3% if all activities are presumed to be non-deterministic in the simulation. Second, the deviation further extends to be within 5-25% when comparing the blocking against the non-blocking simulation, meaning that the existence of buffer fairly influences the precision of the model. Tough the results of the three cases have similar behaviors; we focus on the non-blocking scheme which resembles the real implementation to unveil possible design implications for network processors.

Table 4.4. Validation of the analytical model against the blocking and non-blocking cases. The non-blocking case resembles the real implementation.

(I, J)	Processor Utilization (%)				
	Ana.	Blocking	Non-Blocking	%(ana-B)	%(B-NB)
(1,4)	5.35	6.17	7.58	13.29	18.6
(2,2)	5.31	5.78	7.76	8.13	25.5
(2,3)	6.84	7.13	7.8	4.06	8.6
(2,4)	6.97	7.21	7.75	3.33	6.97
(3,2)	4.82	5.2	6.85	7.31	24.1
(4,2)	4.57	4.79	5.11	4.59	6.26

(a)

(I, J)	Memory Utilization (%)				
	Ana.	Blocking	Non-Blocking	%(ana-B)	%(B-NB)
(1,4)	26.56	29.31	36.56	9.38	19.83
(2,2)	51.1	56.57	74.15	9.67	23.7
(2,3)	67.77	70.63	74.92	4.05	5.72
(2,4)	68.45	71.26	74.58	3.94	4.45
(3,2)	68.78	77.11	99.84	10.8	22.76
(4,2)	87.43	99.84	99.99	4.14	8.78

(b)

4.5.3 Simulation Setup

Two networking applications, Simple Forwarding (SF) and DiffServ (DS), are involved in the simulations, in which the numbers of computational cycles and memory accesses for handling a packet are configured according to [LLP02]. For simplicity, we assume that all memory accesses are of the same type, so the corresponding (P, M)s are configured as (235, 12) and (555, 30). Besides, in order to be realistic, we adopt the non-blocking scheme for the following simulations, in which buffer is provided for packets processed by a thread.

Our goal is to investigate the relationship among processors, threads and memory banks. To do this, a term named *P-M ratio* is defined as

$$\frac{\textit{computational overhead}}{\textit{memory access overhead}} = \frac{\#\ \textit{of computational instructions}}{\#\ \textit{of memory accesses} \times \textit{latency per access}},$$

and three sets of simulations are conducted: simulations with P-M ratio smaller than 1, close to 1, and larger than 1, respectively. A large (small) P-M ratio means the processor overhead is relatively higher (lower) than the memory's and is thought to be an unbalanced combination of the processor and memory, while a P-M ratio close to 1 is considered as a sensible combination. Table 4.5 details the configurations of three different P-M ratios for the SF and the DF. The Intel IXP1200 and IXP2400 are considered in the simulation by setting the memory service time to 20 and 90 cycles, respectively [Com04].

Table 4.5. Different kinds P-M ratios: (a) smaller than 1, (b) close to 1, and (c) larger than 1. SF and DF are included and the memory access latencies are configured as the one of the IXP1200 and IXP2400.

App.	Comp. overhead	Mem. access overhead	P-M ratio
SF	235	$12\times 90 = 1080$	(a) $235/1080 = 0.217$
		$12\times 20 = 240$	(b) $235/240 = 0.98 \cong 1$
		$12\times 5 = 60$	(c) $235/60 = 3.92$
DF	555	$30\times 90 = 2700$	(a) $555/2700 = 0.205$
		$30\times 20 = 600$	(b) $555/600 = 0.925 \cong 1$
		$30\times 5 = 150$	(c) $555/150 = 3.7$

4.5.4 Effect of the RSS Memory Queuing Discipline

Before proceeding with the issues mentioned above, we need to justify the use of the RSS queuing discipline in memory and ready queues. As mentioned in section 4.4, the RSS is assumed to be the queuing discipline for both memory and ready queues without affecting the correctness of the result. For the blocking case, according to Fig. 4.7, it is proven that the processor utilizations using RSS are very close to the corresponding ones using FIFO. Similar observation is seen for the non-blocking case. This is because of the power of *averaging*, namely memory accesses, from a thread, having higher priorities in the queue this time could have lower ones next time. The explanation applies to the memory queue, and is believed to hold for the ready queue.

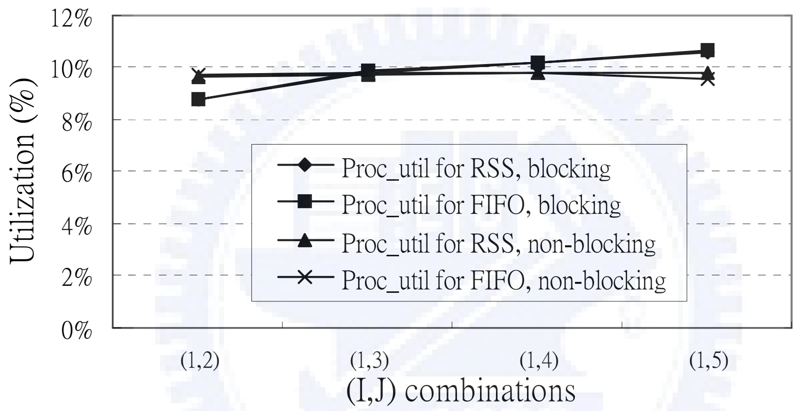


Fig. 4.7. Effect of different memory queuing disciplines for SF.

4.5.5 Unbalanced Load among Threads

Another concern is the resilience of the heterogeneous thread allocation against the unbalanced load distribution. We evaluate the impact by involving the unbalance ratios, in which a ratio of *n* means the load of a thread is *n* times of the one of its predecessor. Figure 4.8 depicts the number of packets in the system for two ratios after executing 3×10^7 cycles. From the figure it is clear that for ratio=2, the number of packets in system increases notably as *J* increases. Nonetheless, only a slight raise is seen when ratio=1.5, meaning that as far as a sensible P-M ratio, which is close to 1, is considered, the system is quite resilient to the unbalanced load among threads.

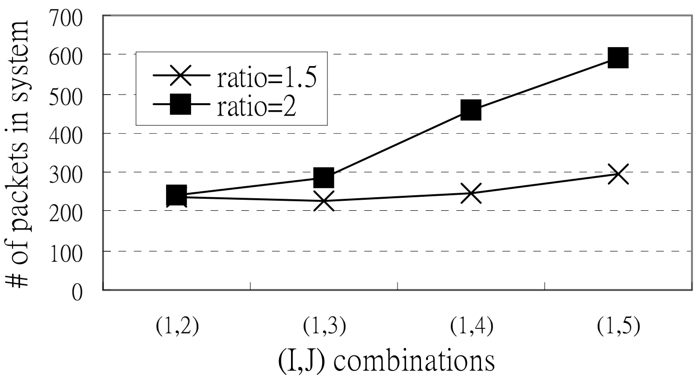


Fig. 4.8. No. of packets in system under different unbalance ratios and no. of threads.

4.5.6 Simulations with Three P-M Ratios

Simulations with a P-M Ratio Larger Than One

Figure 4.9 shows the results of the simulations with a P-M ratio larger than 1. Apparently the memory access overhead is relatively so large that the processor efficiency is low and only two threads are enough to utilize the memory. The SF and DS have similar processor and memory utilizations because their P-M ratios are similar.

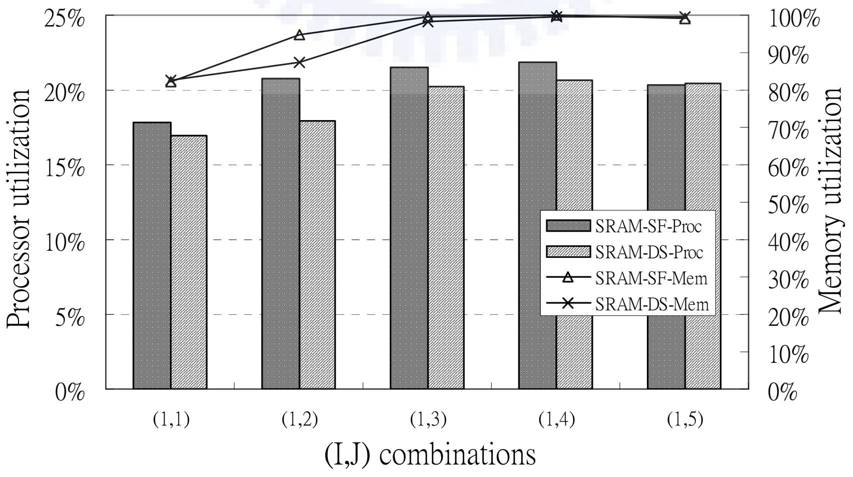


Fig. 4.9. Processor and memory utilizations for the DS and SF with different numbers of threads. The memory service time is 90 cycles.

Simulations with a P-M Ratio Close to 1

Figure 4.10 shows the results of the simulations with a P-M ratio close to 1. The SDRAM, in addition to the SRAM, with a service time of 40 cycles is involved for comparison. From the figure we can see that for SRAM-SF and SRAM-DS the utilizations of both processor and memory are similar because the ratios are close to 1. Moreover, the benefit of utilizing memory from adding threads, taking the SDRAM-SF as an example, becomes less obvious as the memory utilization exceeds 90%. This observation also suggests that *J*=5 is best for applications with a P-M ratio close to 1, since the memory utilization of the SRAM-SF has reached 90% when *J* is 5, implying that adding the sixth thread can have merely limited gain.

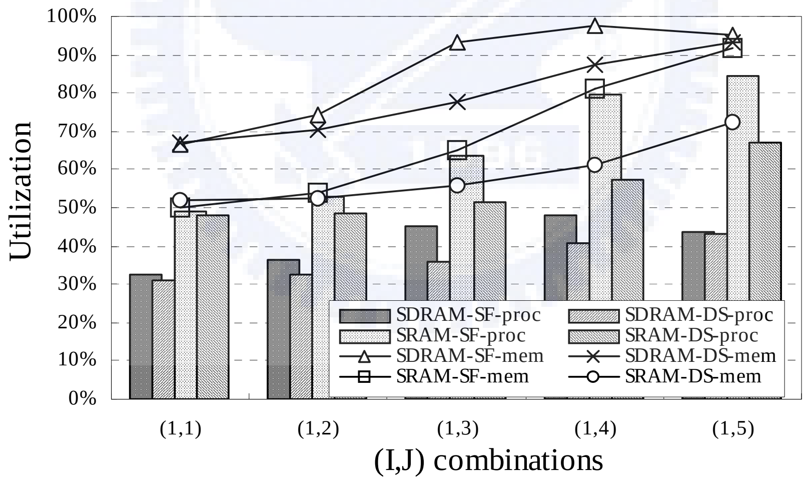


Fig. 4.10. Memory access latency and utilization of various numbers of threads.

Simulations with a P-M Ratio Less Than 1

Figure 4.11 shows the performance improvement by increasing the number of processors. The memory service time is assumed to be 5 cycles, indicating that the memory overhead is less than the one of the processor. The memory sustains the access load until four processors are incorporated for both SF and DS. Interestingly, though memory is apparently not a bottleneck when *I*=1 and 2, the processor is not fully utilized as shown in Fig. 4.12. This suggests that the *J*, which could lead to the low processor utilization, must be carefully estimated before using a fast memory module. Another observation from Fig. 4.12 is that, the fifth processor contributes limitedly in utilizing the memory while resulting in low processor efficiency, implying that *J*, rather than *I*, should be increased when (*I*,*J*)=(4,3).

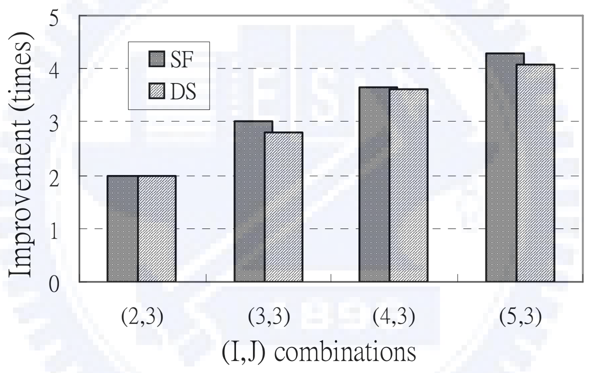


Fig. 4.11. Performance relative to (1,3).

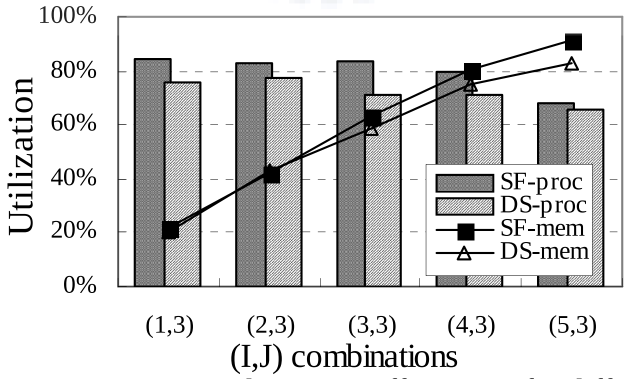


Fig. 4.12. Processor and memory efficiencies for different *I*s.

4.5.7 Solutions for the Memory Bottleneck

Memory usually becomes the bottleneck not only because of the nature of the application but because of the speed gap between processor and memory. To tackle the problem, three common solutions are investigated and compared: enlarging the cache size for better hit ratio; adopting a memory access efficient algorithm, and adding more memory banks. Figure 4.13 compares the effectiveness of the solutions for the DS when (I,J)=(5,5) and L=20. The hit ratio is assumed to be 16.6% and 33.3%, respectively, by reducing the number of memory accesses from 30 to 25 and 20. As for the memory access efficient algorithm, we proceed by supposing a *classification algorithm*, which is part of the packet processing, having memory accesses 50% less (from 10 to 5 accesses) while computational instructions 100% more (from 160 to 320 instructions) than the original algorithm, i.e. (P,M) from (555,30) to (715,25). The idea is that more computational instructions are usually traded for less memory accesses. We consider the effect of multiple banks by employing two banks, looking into two situations in which memory accesses are (1) evenly distributed and (2) distributed with ratios of 1:2 and 1:4. The cause of the second situation is the *data structure* and the nature of the application or the algorithm. An example would be the pattern matching application using the classic Aho-Corasick algorithm [AC75]. It is hard to split the goto table evenly into memory banks, resulting in unbalanced memory access locality. Even if it is possible, the locality problem remains since the matching frequently returns to the root state stored in a certain bank.

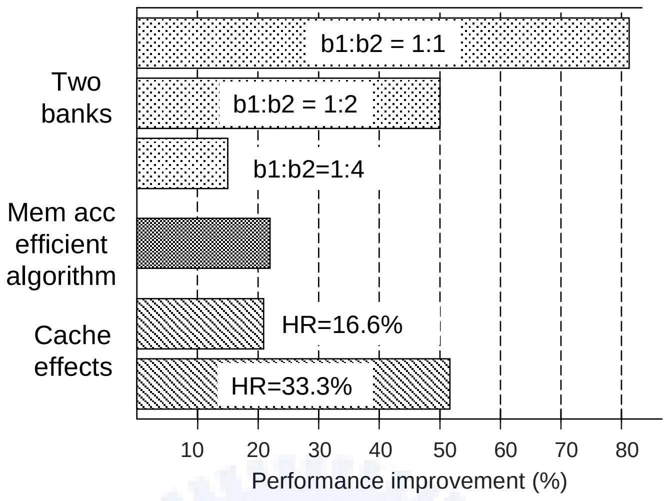


Fig. 4.13. Performance improvement from the three solutions with respect to (I,J)=(5,5) performing the DS. The hit ratio of 16.6% and 33.3% are simulated by using (P,M)=(555,25) and (555,20), while (715,25) is designed to mimic a system with a memory access efficient classification algorithm. Ratios of 1:1, 1:2 and 1:4 are investigated for the two banks case.

From the figure we can see that with a hit ratio of 16.6%, an improvement of 21% can be obtained. The improvement advances to 51.5%, 2.5 times of the one of 16.6% ratio, for a hit ratio of 33.3%. The benefit from a memory access efficient algorithm is 21.5%, similar to the one with 16.6% hit ratio, despite the increased number of computational instructions. The performance gain is best when introducing another memory bank. However, it degrades from 81% to 50% and 15% as the distribution of memory accesses becomes unbalanced.

4.6 Summary

In this work, we try to derive possible design implications, especially the thread allocation, for network processors by developing a preliminary analytical model as well as simulations based on the timed, colored Petri net. Two real networking applications, the Simple Forwarding (SF) and DiffServ (DS), are involved. To date, this work is the first research that adopts the heterogeneous thread allocation scheme and considers the queuing effects in memory and ready queues by practically modeling *I* processors and *J* threads per processor.

Although the analytical model is verified to have similar behavior with the non-blocking simulation which quite resembles the real implementation, we focus on the latter in order to have precise observations. Key observations from the simulation results include (1) the *Random Selection for Service* (RSS) has similar effect with the FIFO when serving as the queuing discipline for both memory and ready queues; (2) the heterogeneous allocation is better than other schemes, and is resilient to the unbalanced load among threads for unbalance ratios smaller than 1.5; (3) for a sensible P-M ratio, i.e. a ratio close to 1 as in the SF/DS over the IXP1200, the most appropriate number of threads is 5, and should be increased/decreased as the ratio decreases/increases, and (4) for solving the memory bottleneck, if any, adding memory banks best improves the performance, though the effectiveness depends heavily on the data structure of the application/algorithm. The observation (1) can be used for further state-space reduction while (2)~(4) serve as implications for the design and implementation of multithreaded multiprocessor network processors. Moreover, by applying the observation (3) assuming the IXP1200 as the hardware platform, we can assert that $J < 5$ is appropriate for the VPN while $J > 5$ for the Intrusion Detection and Prevention as well as the Anti-Virus.

Some issues are to be investigated in the future. First, the analytical model should be revised for large (*I*, *J*)’s. Our model is currently limited to 8 threads in total, for example (2,4) and (4,2), due to the state-space explosion problem. Second, the simulation environment could be enhanced to support (*I*, *J*)’s larger than (5, 5). This is for identifying the *I* and *J* needed for application-platform combinations whose P-M ratio is much larger than 1, namely more computational overhead, and much smaller than 1, i.e. more memory access overhead, respectively. Though increasing *I* in the simulation is doable, *J* is currently confined to 5 because of the user interface of the tool. Finally, since the ordinary multi-bank memory suffers from the difficulty of splitting the data structure of certain applications/algorithms, a multi-port memory, which services multiple memory accesses at once, may be incorporated and considered in our model.

Chapter 5

Resource Allocation of the Core-centric Network Processor for Computational Intensive Applications

5.1 Introduction

Today's networking applications, such as virtual private network (VPN) [BGK99] and content filtering that offer extra security and application-aware processing, have demanded more powerful hardware devices to achieve high performance. The most straight-forward way to tackle this problem is to increase the clock rate of a general purpose processor, though some disadvantages, such as the cost and the technology limit, accompany. Moreover, the low efficiency is also expected since the processor, as its name suggests, is not specifically designed for the processing of networking packets.

Another solution to this problem is to employ the concept of *offloading*, that is, to shift the computing-intensive tasks from the core processor to a number of additional processors. The Application-Specific Integrated Circuit (ASIC) [JS99] has been a possible candidate to serve as an additional processor. Nonetheless, this workaround might not be preferred in two aspects. First, since the functionalities are fixed once tapped out, it needs to be redesigned for any modifications. Second, the development period is so time-consuming that the time-to-market requirement may not be met.

In this work, we explored the feasibility of implementing VPN, which is a computation intensive application, over the Intel IXP425 [INT] network processor featuring an XScale core, *multiple hardware contexts* and coprocessors, and tried to figure out the performance and possible bottlenecks of the implementation. The VPN mechanism, which is usually based on the IPSec [Atk95], comprises several processing stages such as packet reception (Rx) and transmission (Tx), encryption and decryption, authentication and table lookups, each of which needs a certain amount of processing. We analyzed the detailed packet flow and decided to offload packet transferring and cryptographic calculation to coprocessors. Some efforts have also been done to port the VPN application from ordinary PC to IXP425 in the meantime. We then externally and internally benchmarked the resulting prototype. The former characterized performance figures of the implementation, while the latter carried out the in-depth analysis of the observations which were left unexplained in the external benchmarks such as system bottlenecks. The Xscale is identified to be the bottleneck for IPSec processing.

Some related works researching the bottlenecks of network processors can also be found in the literature: Spalink et al. [SKP01] presented the results of simple IP forwarding and Lin et al. [LLY$^+$03] implemented DiffServ, both over Intel IXP1200. Nevertheless, our work differs from theirs in that (1) no coprocessor was involved in their implementations; (2) both the control-plane and part of the data-plane processing were handled in the core processor of IXP425 while the core of IXP1200 took care of the control-plane packets only, and (3) computation intensive VPN application was considered, as compared with simple forwarding and memory intensive classification of these two studies.

This work is organized as follows. We first describe the hardware and software architectures of IXP425. Next, we elaborate the details of the design and implementation of VPN over IXP425. Then we present the results and observations from the external and internal benchmarks. Some conclusive remarks of this article are made finally.

5.2 Hardware Platform (IXP425)

5.2.1 Hardware Architecture of IXP425

The hardware block diagram of IXP425 is depicted in Fig. 5.1. The core of IXP425 is a 533MHz XScale processor handling system initialization and software objects execution. Three buses interconnected by two bridges provide the connectivity among components on IXP425.

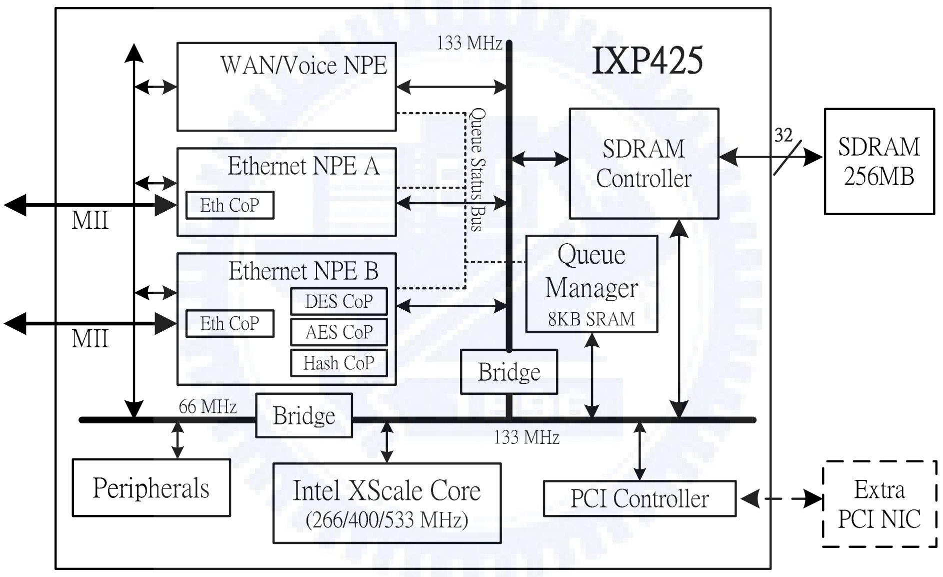


Fig. 5.1. Hardware architecture of IXP425.

To assist the XScale core in processing networking packets, three 133MHz programmable network processor engines (NPEs) are used to execute in parallel the code image stored in internal memory for providing functions such as MAC, CRC checking/generation, AAL2, AES, DES, SHA-1 and MD5, in cooperate with a number of application-specific coprocessors. The support of hardware multithreading with single cycle context switch overhead further makes NPEs more tolerant to long memory accesses and thus reduces the number of processor stalls. The communication between the XScale core and NPEs is handled by a hardware queue manager using interrupt and message queue mechanisms. The queue manager also contains 8KB SRAM divided into 64 independent queues manipulated as circular buffers for allocating free memory space to incoming packets and for locating packets in the memory. The SDRAM can be expanded up to 256MB for storing tables, policies and OS applications in addition to packets. A PCI interface is available for an additional PCI NIC. Some peripheral controllers, like USB and UART controllers, are also equipped into IXP425 for better extensibility.

5.2.2 Detailed Packet Flow in IXP425

The processing flow of an ordinary packet is elaborated below referring to Fig. 5.1. Upon the arrival of a packet at the interface of an NPE, it is partitioned into several 32byte segments and stored at the Receive FIFO of an Ethernet coprocessor which in turn performs MAC-related operations. The NPE then moves those segments into corresponding addresses in SDRAM allocated by the queue manager, which then interrupts the XScale of the reception for further processing. During normal processing procedures such as IP and other higher layer protocol stacks at XScale, chances are that some authentic and cryptographic operations are needed. The XScale core may handle them either by itself or by *offloading* the computation overhead to appropriate coprocessors residing in NPE B. In the latter scenario, the coprocessors are directly invoked by NPE B, requested by the XScale, to process a certain data segment in SDRAM, where a message queue implemented in the queue manager is exploited to pass the request. The queue manager is informed by NPE B upon the completion of the operations and then interrupts the XScale.

5.2.3 Software Architecture of IXP425

The software architecture shown in Fig. 5.2 is divided into two portions, namely the platform independent (applications and some higher level components such as networking protocol stacks in OS) and dependent parts (mainly device drivers). This design is favorable especially when an OS migration from a certain H/W platform to another is demanded, that is, the developers need to focus only on the dependent part, namely the development of drivers. When implementing device drivers, a set of software libraries collectively referred to as *AccessLibrary* can be used to drive devices such as NPEs, coprocessors, peripherals, etc. The AccessLibrary also provides utilities, such as *OSSL* and *IxOSServices* to implement some OS-related functions such as mutual exclusion.

The software processing flow is described as follows with library functions adopted from the AccessLibrary. During the boot time a function named *IxNpeDl* is called to download the corresponding code image into the instruction cache of each NPE. Then two functions, *IxQmgr* and *IxNpeMh*, are called to initialize the queue manager as well as the message handler responsible for the communications between NPEs and XScale. The Ethernet-related functions, *IxEthAcc* and *IxEthDB*, are used to receive and transmit Ethernet frames, while the *IxCryptoAcc* function is incorporated for possible cryptographic operations during packet processing.

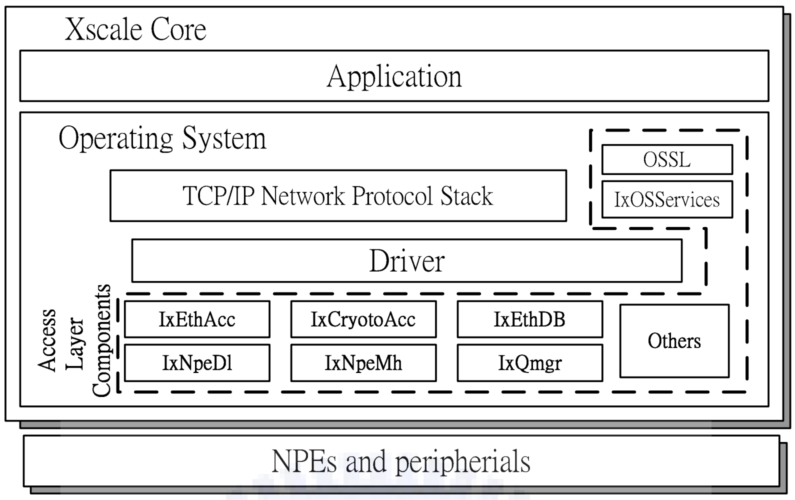


Fig. 5.2. Software architecture of IXP425.

5.3 Processing Stages Analysis and Offloading Schemes Design

In this section, we first introduce basic operations in a VPN environment and then analyze its packet processing flow in order to identify possible bottlenecks as offloading candidates. Finally, we describe how to implement a VPN gateway over IXP425.

5.3.1 VPN Briefing

Virtual Private Network (VPN) provides secure transmission over un-trusted networks. Normally the IPSec protocol is adopted as the underlying technique due to the popularity of the Internet Protocol. It supports data authentication, integrity and confidentiality, in which two gateways are employed as endpoints constructing a VPN tunnel for secure data transmission. Improving the performance of the gateways is decisive to the VPN throughput.

5.3.2 Identifying Offloading Candidates

To resolve the performance issue, we analyze the VPN packet processing flow in order to identify possible candidates to be offloaded to coprocessors. A detailed inbound IPSec packet flow was displayed in Fig. 5.3. It consists of three main blocks, namely the packet reception, IPSec processing, and packet transmission. Their operations are elaborated below.

Once an Ethernet frame is received by the physical interface, checking for frame check sequence screens out broken frames and the remaining frames are examined in accordance with possible MAC address filtering configurations. Reception is accomplished after the frame is moved into memory, followed by a classification recognizing it as an IPSec packet. At this time, some table lookups for processing rules and cryptographic parameters are performed and payload of this IPSec packet is decrypted or checked for authentication. Finally, a new packet decrypted from the original IP payload is further processed by higher-level protocols, or is transmitted according to the routing table.

Tasks suitable to be offloaded to coprocessors can be identified by two characteristics: whether those tasks are repeated routines or computation intensive ones. As mentioned earlier this section, we know that IPSec processing, especially the cryptographic operation, is computation intensive. Hence, we decide to pick the cryptographic processing as an offloading candidate. Another candidate comprises the packet transfer, CRC checking/generation, MAC filtering, and packet movement between NPE and memory, since the procedures are precisely the same for every packet. From the hardware block diagram in section 5.2, it is obvious that the IXP425 has the hardware components for the identified candidates.

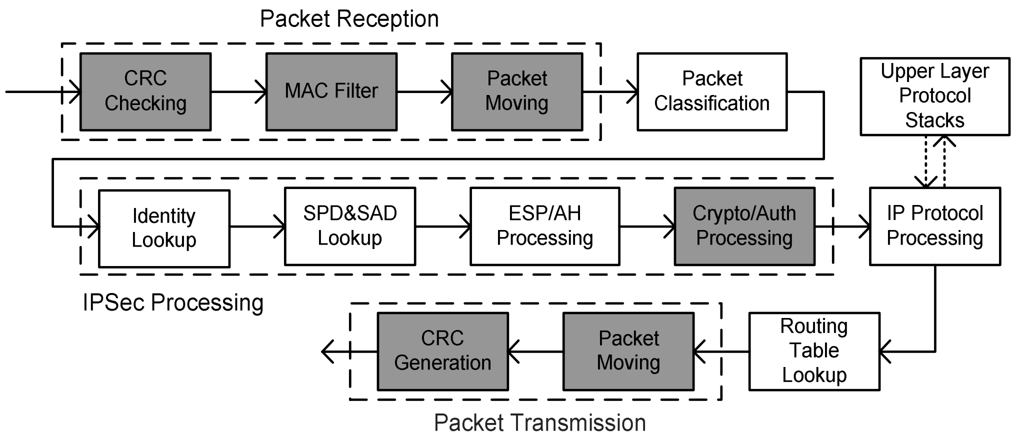


Fig. 5.3. Processing flow of an inbound IPSec packet. Shaded blocks are candidates to be offloaded.

5.3.3 Implementation

We adopt the NetBSD [Net], a secure, highly portable and open-source OS derived from 4.4BSD, as our operating system. Clean design between platform dependent and independent parts makes it a good implementation target for new hardware platform. Following relates three major components in prototyping a security gateway over IXP425.

Operating System Porting. The most efficient way to porting an OS to a new platform is refer to the port of another similar platform and then implement drivers for the target platform based on that port [Kes95]. To port NetBSD over IXP425, therefore, we adopt the "EvbARM" port in NetBSD. It supports various evaluation boards that equip with XScale or other ARM-based core processors, so that only system-level modifications have to be done to enable normal operations of IXP425. Example modifications include the CPU identification, setup of board-specific memory map, and system initialization procedures.

Driver Development. A number of drivers for devices such as UART, NPEs and coprocessors need to be implemented for communication between the operating system and those devices. This effort can be alleviated with the help of the AccessLibrary introduced in section 5.2. Besides drivers, we have to modify two OS dependent modules, namely OSSL and IxOSServices, in AccessLibrary to ensure proper operations of the OS-related services.

Offloading the Cryptographic Operations. The last modification to kernel concerns the offloading of in-kernel IPSec cryptographic computations from XScale to coprocessor. Ordinary method requires that the kernel performs and subsequently *waits* on the encryption/decryption operations carried out by the coprocessor. However, NetBSD provides another option named *FAST_IPSec* that makes use of the Open Crypto Framework (OCF) for offloading. In OCF, the cryptographic operations can be handled by a *registered* function. The FAST_IPSec prevails over the original offloading technique in that the XScale would not suspend during cryptographic operations. We exploit this technique by pre-registering the crypto driver, which drives the crypto coprocessor using functions in AccessLibrary, to the OCF.

5.4 Benchmark and Bottleneck Observations

In this section, we investigate the benefits from offloading by externally benchmarking the implementation using various offloading schemes. A number of internal tests are also conducted in order to observe what cannot be obtained in the external benchmarks.

5.4.1 System Benchmark Setup

To have a better understanding of the improvement from the network processor architecture as well as the offloading mechanisms, we design and benchmark systems of different offloading schemes, and compare their performance results.

Four offloading schemes are adopted: (1) offload both crypto operations and packet Rx/Tx to the corresponding coprocessors; (2) offload crypto operations only; (3) offload Tx/Rx only, and (4) no offloading. Figure 5.4 diagrams the corresponding data paths for the four schemes.

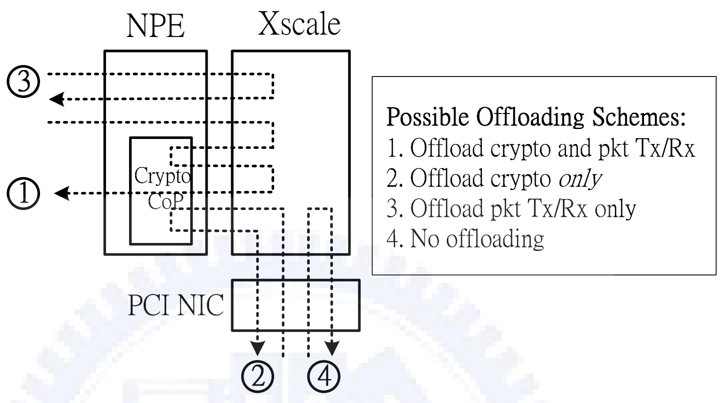

NPE
Xscale
③
①
Crypto CoP
PCI NIC
②
④
Possible Offloading Schemes:
1. Offload crypto and pkt Tx/Rx
2. Offload crypto *only*
3. Offload pkt Tx/Rx only
4. No offloading

Fig. 5.4. Data paths of the four offloading schemes.

As for the external benchmark environments for packet forwarding and IPSec, we use *SmartBits* to generate the input traffic and to collect and analyze the performance results. For internal tests, some system utilities such as *vmstat*, top and *GProf*, are employed to obtain the system state as well as other internal behaviors such as CPU and memory utilizations.

5.4.2 Scalability Test

Scalability tests aim to derive the maximum throughput of the prototypes of different offloading schemes. Another gateway implementation using Pentium III 1GHz processor and 256MB SDRAM is also included for comparison between IXP425 and x86-based systems.

Packet Forwarding. Figure 5.5 shows the performance results of 1-to-1 packet forwarding under the condition of zero packet loss. From the figure we can see that throughput of the IXP425 offloaded by two NPEs parallels the one of Pentium III 1GHz. Both of them can support wired speed for packet lengths larger than 512 bytes. Besides, a performance improvement of up to 60% contributed by

NPEs can also be gained. We also observed that the maximum throughput occurs when the packet length is 1024 bytes, rather than other larger lengths. This is because the longer processing time of larger packets counteracts the benefit from their reduced header processing overhead.

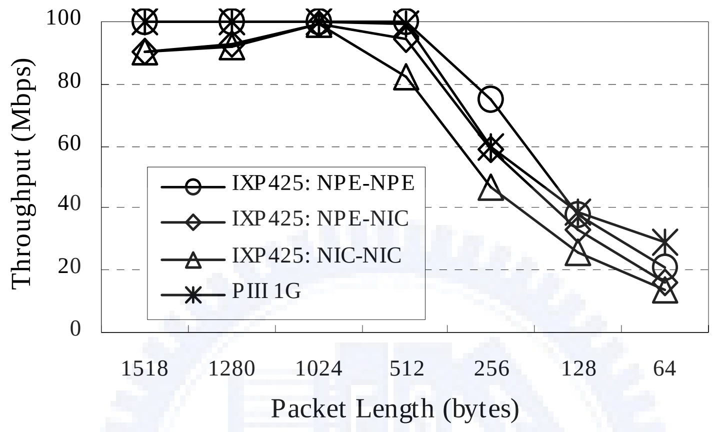


Fig. 5.5. Throughput of packet forwarding when different numbers of NPEs are used for offloading.

IPSec Processing. Figure 5.6 depicts the throughput of DES for different packet lengths. Some observations can be made. First, offloading IPSec processing to coprocessors in NPE B improves the performance by 350%; in some cases IXP425 even outperforms the Pentium III 1GHz. Second, the maximum throughput occurs when the packet length is 1450 bytes, instead of 1518 bytes. This is because 1450 bytes is the largest length for a packet not to be fragmented when being encapsulated into an IPSec one. Third, the throughput of 3DES on IXP425, as shown in Fig. 5.7, is similar to the one of DES whereas the computation requirement of the former is almost triple of the later. The reason is that it is the XScale, not the coprocessors, that becomes the bottleneck.

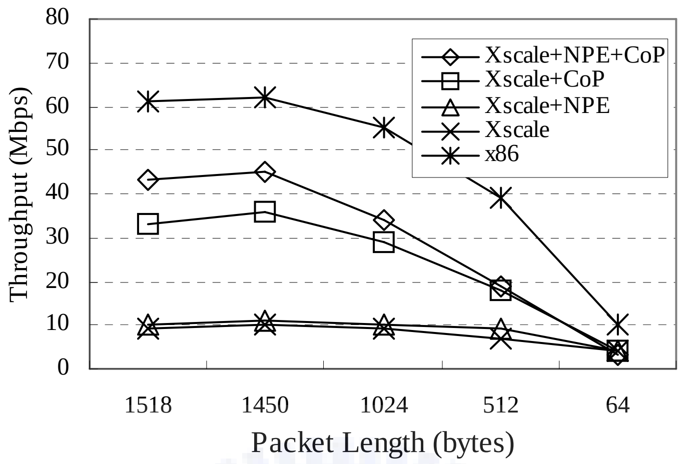


Fig. 5.6. IPSec Throughput: the DES case.

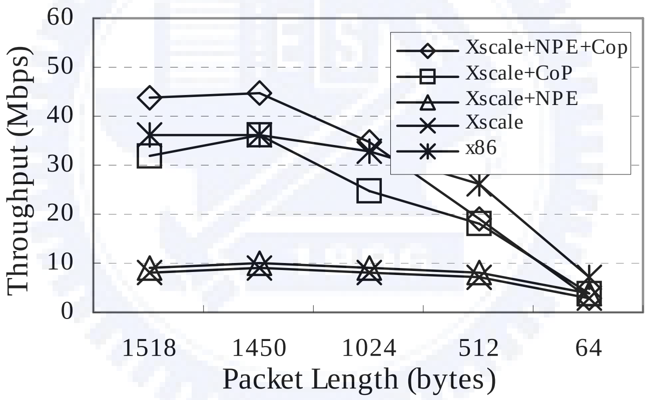


Fig. 5.7. IPSec Throughput: the 3DES case.

5.4.3 Bottleneck Analysis

Bottleneck of Packet Rx/Tx. To proceed the bottleneck analysis, we considered four main functional units likely to affect system performance: bus, memory system, NPE and XScale. It is obvious that neither the bus nor the memory is a bottleneck because wired speed can be achieved for some larger packet lengths. The NPE is not a bottleneck either, since, as observed by the *netstat* utility, all packets are received and stored at the memory. The bottleneck can therefore be identified as the XScale since the packet processing is carried out mostly by it.

Figure 5.8 shows that the utilization of the XScale linearly advances as the traffic load increases.

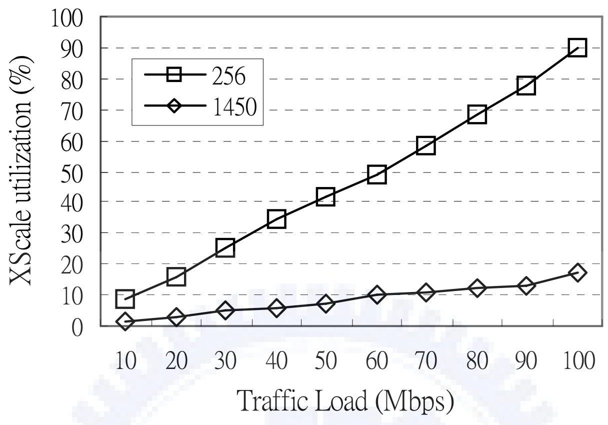


Fig. 5.8. Input traffic load vs. XScale utilization for two packet lengths (bytes).

Bottleneck of IPSec Processing. The bottleneck in the IPSec processing is known to be the XScale before offloading is applied, since the cryptographic calculation demands much computing power. However, the XScale is again found to be the bottleneck even after offloaded by the crypto coprocessors. Figure 5.9 shows that when traffic load is 50Mbps exceeding the maximum system throughput of 46Mbps, the utilization of XScale approaches 100% and the success ratio of IPSec packets significantly drops to 22%. This is because the processor is so busy that incoming packets are dropped due to limited buffer space.

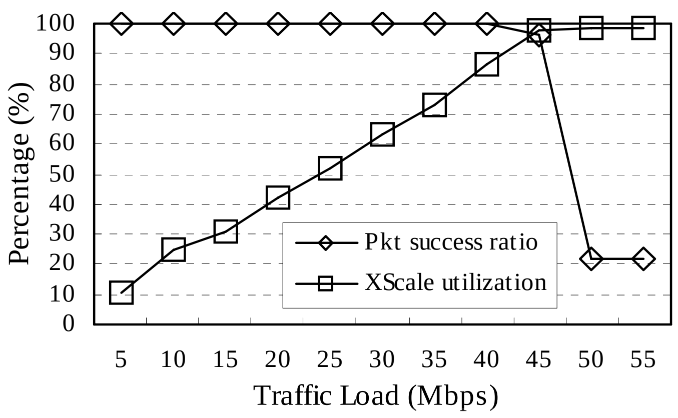


Fig. 5.9. IPSec packet success ratio vs. XScale utilization.

The XScale bottleneck can be further confirmed with the turnaround times of the DES and 3DES requests, respectively, as shown in Fig. 5.10. The turnaround time means the duration from the time a request of cryptographic operations is issued by XScale to the queue manager, to the time the XScale is notified of the completion. As mentioned previously, the throughputs of DES and 3DES are similar, indicating that their turnaround times should also be the same. However, this contradicts the results in Fig. 5.10 in which the turnaround times of DES and 3DES are different, justifying that the XScale, rather than the crypto coprocessor, is the bottleneck when performing DES and 3DES. The throughputs of DES and 3DES are the same because they are bound by XScale.

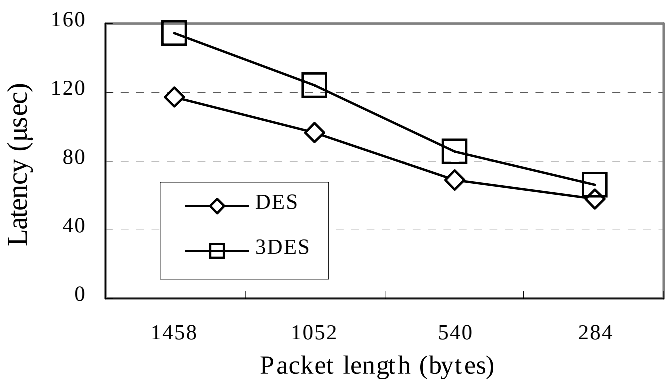


Fig. 5.10. Turnaround time of a cryptographic request for a packet.
Packet size may vary.

We can also estimate the maximum throughput of the crypto coprocessor as the processing times of encryption and decryption are proportional to the data length. The estimated performances can be computed by △s/△t, where △s and △t represent the differences of two packet lengths and two latencies, respectively. Therefore, the crypto coprocessor is estimated to scale approximately to $\frac{1458-1052}{117-97} \cong 20.3(bytes/\mathrm{u\,sec}) = 162.4(Mb/\sec)$ for DES, and to 101Mbps for 3DES likewise.

5.4.4 Turnaround Time Analysis of Functional Blocks

Figure 5.11 depicts the turnaround time analysis of the functional blocks when processing DES and 3DES packets. Functional blocks considered consist of the IP processing, IPSec preprocessing including identity and SAD/SPD lookups, and IPSec encryption. Three kinds of testbed configurations are conducted for testing DES and 3DES: IXP425 with the cryptographic operations offloaded to the coprocessor; IXP425 without offloading, namely XScale only; and PIII processor.

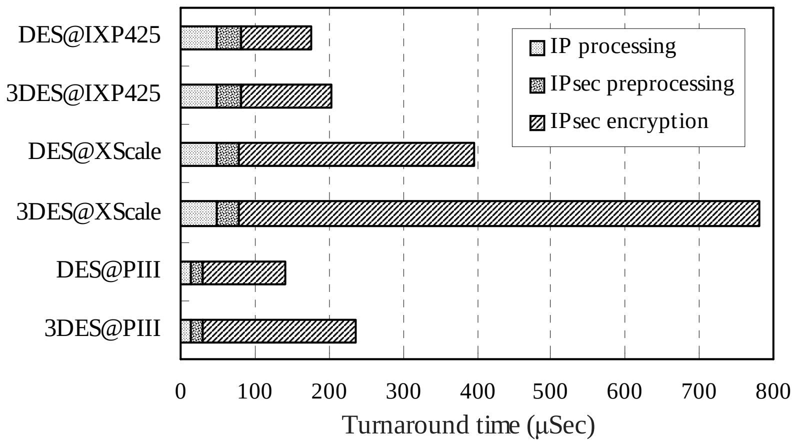


Fig. 5.11. Turnaround time of functional blocks.

From the figure we can see that cryptographic calculation accounts for a major portion, from 80% to 90%, in the packet processing time before offloading. After offloading to the coprocessor, the time for cryptographic calculation is reduced from 700 us to 100 us. Notably both the IXP425 and single XScale configurations have the same IP processing and IP preprocessing periods because those tasks are executed only by XScale.

5.5 Summary

In this work, we elaborate the implementation of a VPN gateway over the IXP425 network processor, where a number of coprocessors are provided for offloading computation intensive tasks from the Xscale core. We introduce the hardware and software architectures of the platform, analyze the VPN, i.e. IPSec, processing flow, and then identify the packet Rx/Tx as well as encryption/decryption as the ones to be offloaded to coprocessors. We realize the offloading design by implementing a number of drivers in NetBSD, and finally externally and internally benchmark the system in order to find possible performance bottlenecks.

The benchmark results show that the throughputs of packet Rx/Tx and IPSec processing are improved by *60%* and *350%*, respectively, after offloading. However, the Xscale is again found to be the bottleneck for both packet Rx/Tx and IPSec processing.

Two issues are to be investigated in the future. First, more tasks may be offloaded to NPEs or to coprocessors. An example of this is the IPSec database lookup, which determines the policy to be applied to a certain IPSec packet. Second, the performance may be further improved if we call the related functions in the AccessLibrary directly for cryptographic operations, instead of going through the Open Crypto Framework.

Chapter 6

Core-centric Network Processors: Analysis, Simulation, and Design Implications

6.1 Introduction

Networking applications offering extra security and content-aware processing features demand much powerful hardware platforms to achieve high performance. For computational intensive applications such as the Virtual Private Network (VPN) [BGK^{+}99], general purpose processors are often adopted; however, the cost is considerable while the throughput is not satisfactory because of heavy cryptographic operations. Rather, the Application-Specific Integrated Circuits (ASICs) [JS97] can meet the performance requirement with a circuitry designed for both networking and cryptographic processing. Nonetheless, the lack of adaptability makes it less appealing.

Network processors [Lek03] have been embraced as an alternative to tackle the above-mentioned problems for their *core-processor/coprocessors* -based architecture, on which control and data -plane processing can be separated for efficiency, and the re-programmability for functional adaptations. The core processor can perform complicated operations and is thus responsible for control messages, while a number of coprocessors, having specifically designed instructions for networking purpose, are employed for mass data-plane processing. This kind of architectures, referred to as the *coprocessors-centric* model, is frequently applied as a core device which requires low configurability but high scalability [LLY$^+$03][CLS$^+$04][CM06][LCL$^+$07][TLY$^+$04]. When acting as an edge device that deals with relatively mild traffic volume, both control and data –plane packets are processed by the core processor. This is referred to as the *core-centric* model. Nonetheless, computational intensive tasks such as receiving, transmission and en/de-cyption can still be offloaded to certain application-specific coprocessors [LLL$^+$05].

Several studies have acknowledged the feasibility of adopting these models in packet processing for applications such as DiffServ, VPN, Cryptographic algorithms, Intrusion Detection and Prevention (IDP). In addition to evaluation through implementations of both models to discover system bottlenecks [LLY$^+$03][CLS$^+$04][CM06][LCL$^+$07][TLY$^+$04] [LLL$^+$05], mathematical modeling [CB02][WF06][LW06] is favored in order to unveil possible design implications which are unlikely to observe through real benchmarking. Though, analytical resort for the emerging core-centric model is yet unattempted.

In this work, we analyze the untapped core-centric network processors by modeling the IXP425 performing Virtual Private Network (VPN) application. The IXP425 [INTb] employs an XScale core processor in charge of general packet processing and coprocessors executing receiving, transmission and cryptographic operations. The task allocation and important parameters are obtained from real implementation [LLL$^+$05], providing sufficient ground for model accuracy. Two analytical models are developed using Continuous Time Markov Chain, a method widely adopted for capturing system behaviors. The first is a *busy-waiting* model in which the core hands over the intermediate results to the coprocessor for certain processing, and keeps polling whether the coprocessor finishes. This primitive approach is used by some operating systems, for instance NetBSD, when certain coprocessors are incorporated. The busy-waiting model is then extended and compared to an *interrupt-driven* model in which the core switches to another process while waiting for a signal indicating the completion of previously offloaded job. This technique is realized in NetBSD by enabling the OCF (Open Crypto Framework) option.

Aside validation on the analytical model, the simulation is developed for inspecting internal characteristics of the system, which oftentimes cannot be obtained from real implementations and from mathematical analysis due to enormous state space. With these established analytical and simulation models, we aim at revealing design implications from system and IC vendors' perspectives, respectively. The former includes the effects from processor run length, context switch overhead, while the latter covers the benefits from offloading and influence of the buffer size.

Results of the analytical model prove to be closely inline with those of Petri net simulations and system benchmark. Though, context switch delay considered in the model is then found to be ineffective, implying minor, if not zero, switching overhead in the real implementation. The model is thus revised and shown to retain accuracy.

This article is organized as follows. Section 2 briefs the overview of the core-centric IXP425 network processor system and our modeling approach. We develop the analytical models and simulation design in section 3 and section 4, respectively. Section 5 presents the results and observations. Some conclusive remarks of this article are made in section 6.

6.2 Background

6.2.1 Performance Model Overview

The core of IXP425 is an XScale processor handling system initialization and software objects execution. Three buses interconnected by two bridges provide the connectivity among components on IXP425. To assist the XScale core in processing networking packets, three coprocessors named network processor engines (NPEs) are used for providing functions such as MAC, CRC checking/generation, AAL2, AES, DES, SHA-1 and MD5, in cooperate with a number of application-specific coprocessors. Our analytical models for the processing flow are based on the implementation of VPN over the IXP425 network processor. As shown in Fig. 1, the processing flow can be summarized into five tasks, namely (1) receiving, (2) IPSec preprocessing, (3) en/de-cryption, (4) IP processing, and finally (5) transmission. Notably the shadowed blocks, i.e. tasks #1, #3 and #5, are offloaded to corresponding coprocessors namely receiving coprocessor, computational coprocessor and transmission coprocessor, whereas tasks #2 and #4 are handled by the core through context switching.

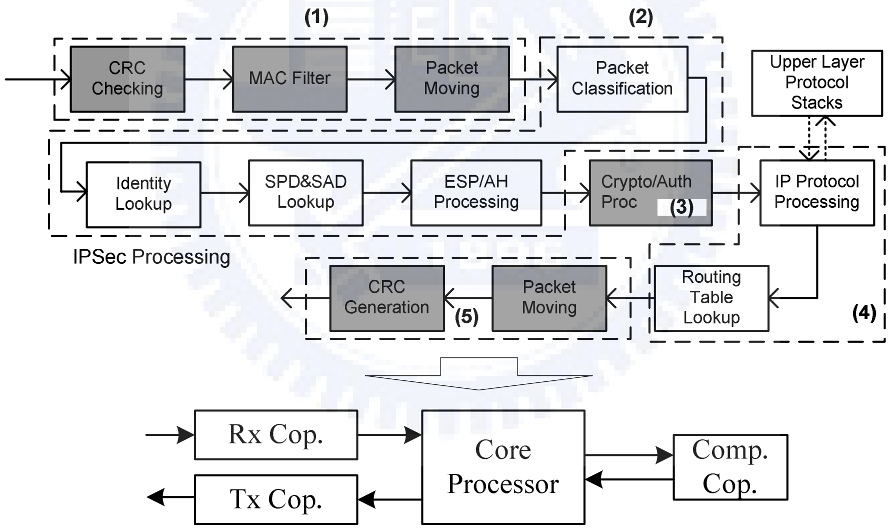


Fig. 6.1. Processing flow and task allocation of the VPN application over IXP425: physical and logical views.

6.2.2 Architectural Assumptions

Some coprocessors may incorporate multiple hardware threads [INT04] to alleviate memory access latency by switching out the processor control to another thread when issuing a memory access. Nevertheless, hardware multithreading requires duplicate register sets which suggest an increased cost, and is helpful for only memory-access intensive applications such as DiffServ, Intrusion Detection and Prevention (IDS). Therefore, in this work we assume single thread in each coprocessor since VPN is computational intensive, rather than memory-access intensive. Buffer for each processing stage which is frequently involved practically are also encompassed, except for the busy-waiting model which needs no buffer between the core and the computational coprocessor.

6.3 Analytical Model

6.3.1 The Busy-waiting Model

In this model, the core does not have buffer between and the computational coprocessor and therefore has to wait on the signal from the coprocessor. For example, when the core finishes the IPSec preprocessing, the result is passed to the computational coprocessor for en/de –cryption and is then again handed over to the core for IP processing. In this regards, the core and the computational coprocessor can be seen as different processes in a logical CORE processor, since only one of them can be active anytime. The scheme can further be simplified as three series queues, as shown in Fig. 2, in which all components are independent $M/M/1/\infty$ models and the departure-time distribution from a queue is identical to the interarrival-time distribution of another. The utilizations of the receiving and transmission coprocessors are trivial, whereas for CORE it can be obtained as

$$\mu_{CORE} = \frac{\lambda}{T_{core\_A} + T_{cop} + T_{core\_B}}, \tag{1}$$

where λ denotes the arrival process at the CORE and T_{core_A}, T_{cop} and

T_{core_B} represent the processing time for IPSec preprocessing, en/de –cryption and IP processing, respectively. Finally we can have the utilizations for core and computational coprocessor as

$$\mu_{core} = \mu_{CORE} \times \frac{T_{core\_A} + T_{core\_B}}{T_{core\_A} + T_{cop} + T_{core\_B}}, \text{ and} \tag{2}$$

$$\mu_{cop} = 1 - \mu_{core}. \tag{3}$$

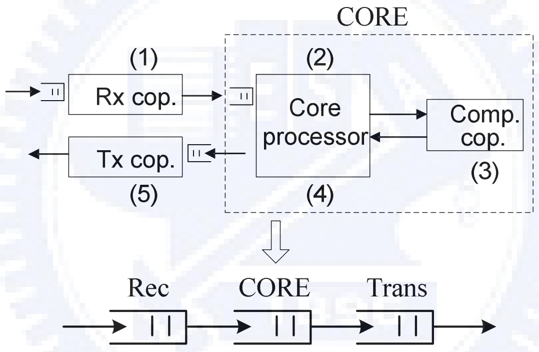


Fig. 6.2. The busy-waiting model.

6.3.2 The Interrupt-driven Model

Contrasted with busy-waiting, in this model the core passes the result of IPSec preprocessing to the computational coprocessor and resumes without being blocked. To realize this concept, two processes need to be forked in the core for IPSec preprocessing and IP processing, respectively, and buffer is required between the core and coprocessor. When the IPSec preprocessing is done and the packet is passed to the coprocessor's buffer, the context is switched to the other process, with certain switching delay T_D, for performing IP-related operations so that the core is not stalled. To reflect this enhancement, a *processor control switch* referred to as *PCS* is adopted to capture behaviors of the two processes. According to the above descriptions we can formally define a state of the system as

$$ST = (R, A, C, B, T, S),$$

where *R*, *A*, *C*, *B and T* denote the queue lengths for the five task stages, namely receiving, IPSec preprocessing indicated as Core_A, en/de –cryption indicated as Cop, IP processing indicated as Core_B, and transmission, while *S* denotes PCS. As shown in Fig. 3, *S*=0/*S*=1 means the core is processing packets at Core_A/Core_B. Notably the core could still be busy-waiting for (1) packet arrivals from its predecessor or (2) available buffer slots in its successor for passing the result. The PCS should be manipulated well to avoid these situations by setting (1) appropriate run lengths T_{S1} at Core_A and T_{S2} at Core_B so that the processing resource is reasonably distributed, and (2) correct transitions so as to ensure that context switches are performed upon those situations. Parameters used in the analytical model are described in Table 6.1.

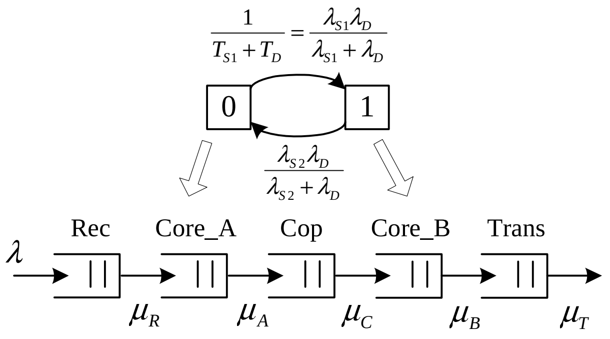

$\frac{1}{T_{S1}+T_D} = \frac{\lambda_{S1}\lambda_D}{\lambda_{S1}+\lambda_D}$
0
1
$\frac{\lambda_{S2}\lambda_D}{\lambda_{S2}+\lambda_D}$
Rec
Core_A
Cop
Core_B
Trans
λ
μ_R
μ_A
μ_C
μ_B
μ_T

Fig. 6.3. The interrupt-driven model.

Table 6.1. Notations for the analytical models.

- λ denotes packet arrival rate.
- T_{S1} denotes the run length of PCS at Core_A.
- T_{S2} denotes the run length of PCS at Core_B.
- λ_{S1} denotes the switching rate of PCS from 0 to 1. $\lambda_{S1} = 1/T_{S1}$.
- λ_{S2} denotes the switching rate of PCS from 1 to 0. $\lambda_{S2} = 1/T_{S2}$.
- T_D denotes the context switch delay.
- λ_D denotes $1/T_D$
- μ_X denotes the service rate of processing stage *X*.

6.4 Simulation Environment

Some tools have been available for simulating architectures similar to network processors [NFS04][DFL05]. Though accurate, they focus mainly on the low-level configuration such as cache structure and lack flexibility in task allocation. In this section, we describe the construction of the simulation environment based on timed, colored Petri nets (CPNs) [Mur89][ZGF98] which captures well component-level activities. It is used to validate the analytical model discussed in the previous section as well as to observe possible hints for future design.

We adopt the event-driven CPN-Tools [RWL$^+$03] as our simulator. The features it supports, including the colored tokens, stochastic functions and hierarchical editing, provide efficiency in the construction of timed, colored Petri nets corresponding to our model. As shown in Fig. 4, the net contains five *transitions* representing task stages, each of which associated with a control token indicating the availability of the processing resource, and equipped with a *place* representing buffers, namely B0, B1, IF_out, IF_in, and B2. The size of the buffers is configured in other five places, i.e. B0’, B1’, IF_out’, IF_in’ and B2’, respectively, by marking them with a number of initial tokens. The following description exemplifies a sample processing flow.

When a packet arrives at the receiving coprocessor, B0, with the inter-arrival time being exponentially distributed with mean λ, one token in B0' is consumed indicating the occupation of a buffer slot. Once the receiving coprocessor is available (the R_tok place contains a token), the packet is processed for P_R *u*sec and then passed to the Core_A stage, if room (B1'>0), while the tokens go back to the R_tok and B0'. If the token in P_tok is available, that is the Core_B is not executing, the Core_A starts to process the packet for P_A *u*sec and then offloads en/de –cryption operations to the computational coprocessor which take for P_C *u*sec. Notably the token returning to P_tok costs additional T_D *u*sec for context switch overhead. Similar procedures apply to the Core_B and the transmission coprocessor which last for P_B and P_T *u*sec, respectively.

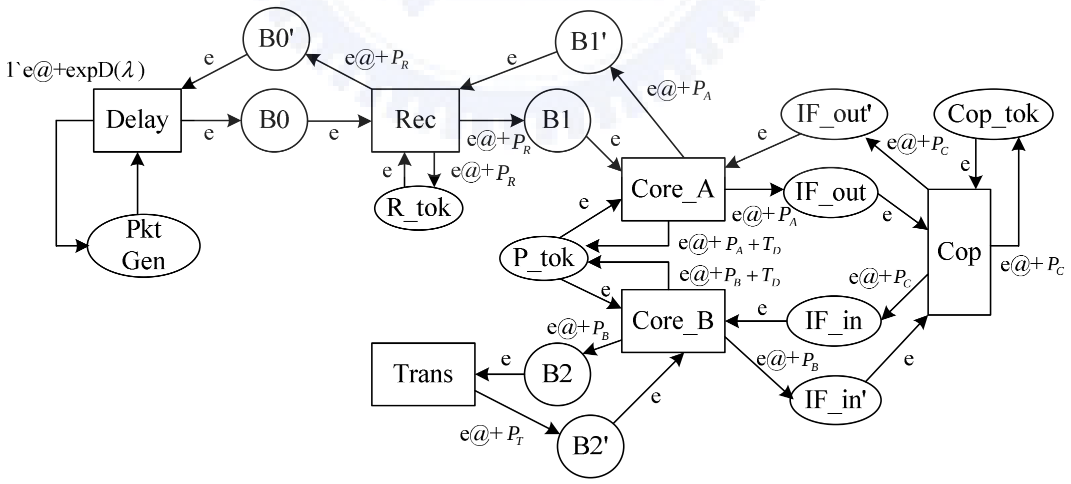


Fig. 6.4. The Petri net simulation model.

6.5 Evaluation

In this section we first validate the analytical model with simulations and real implementation, through which the model is revised to be much precise. We then evaluate and analyze core-centric network processors from both system and IC vendors' perspectives, and disclose possible design implications.

6.5.1 Validation of the Analytical Model

The analytical model is validated by simulations. Parameter settings for the analytical model as well as the simulation are listed in Table 2.

Table 6.2. Processing time of the tasks evaluated in a real implementation.

Task	Processing time
(1) Receive	27.3 *u*s/pkt
(2) IPSec prep	31 *u*s/pkt
(3) Crypto	12.6 *u*s/pkt
(4) IP processing	49 *u*s/pkt
(5) Transmit	27.3 *u*s/pkt

We first try to find the most appropriate transition rate for PCS. As Fig. 5 presents, compared to the normal run length of 6666 *u*sec [20], when choosing 100 *u*sec we can have 2.26 times improvement on the effective core utilization while consuming 20.5% less computational resource. Busy-waiting period, the difference of the utilization and effective utilization, is significantly alleviated.

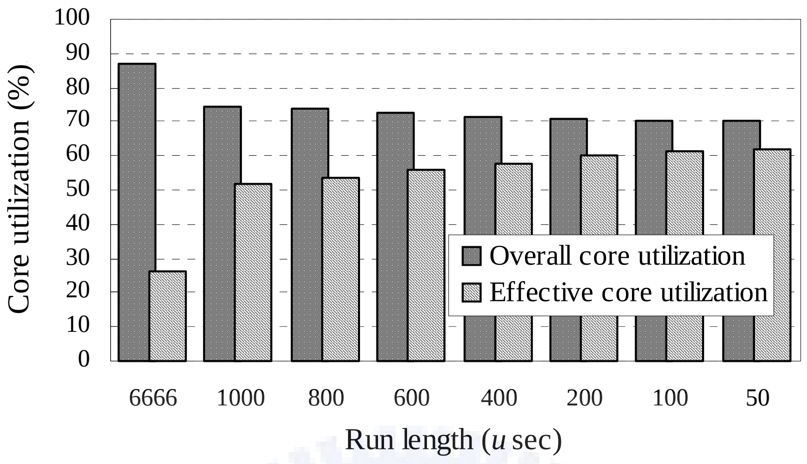


Fig. 6.5. Run length vs. core utilization.

In simulations, the context switch delay, T_D, has been decreasingly reduced in order to have results inline with those of the analytical model. We finally find that, with T_D being very close to 0 the analytical results are mostly within 1% of the simulation, as presented in Fig. 6. The discrepancy comes from different assumptions between the model and simulation. The former assumes non-deterministic behaviors in the packet arrival and instruction processing, while the latter uses deterministic ones in order to be realistic. What can be further implied is that the context switch delay is minor in the implementation, which is quite unreasonable, suggesting that only one process in the core is employed for both IPSec preprocessing and IP processing. The utilization of the implementation is slightly higher (3%-4%) than the analytical model when lightly loaded because of the operating system overhead. The discrepancy noticeably increases when overloaded. It is also surprisingly learned that the limited buffer size, which is configured to 3, does not influence the accuracy of the model. We will discuss it later in this section.

One observation concerning us is, as the validation proceeds, T_D in the analytical model does not have much effect than it should. We soon realize that the context switch overhead is actually not effective in the model, since the PCS transits with no delay.

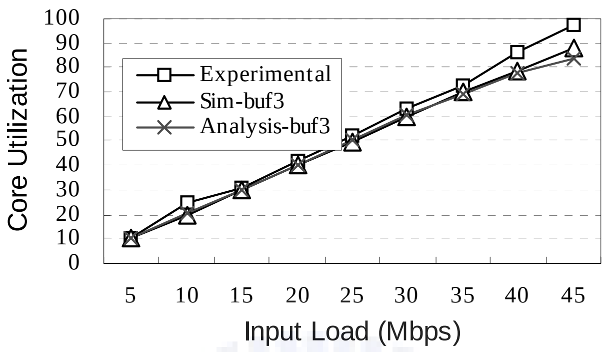


Fig. 6.6. Analytical model validation against the simulation and real implementation.

The model is then revised by adding two statuses for the PCS and again proves to have results inline with those in Fig. 6. As Fig. 7 shows, the overhead is considered (2=>1 and 3=>0) after PCS decides to switch (0=>2 and 1=>3). Fig. 8 elaborates five sample transitions, among which four of them are performing certain tasks and one is receiving packets. Since the buffer size is configured to 3, Core_B cannot pass the result to the transmission coprocessor whose buffer is already full. Similarly, PCS does not change from 0 (Core_A) to 1 (Core_B) to refrain from busy-waiting.

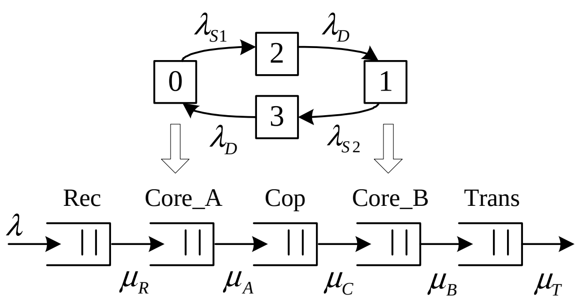


Fig. 6.7. The revised analytical

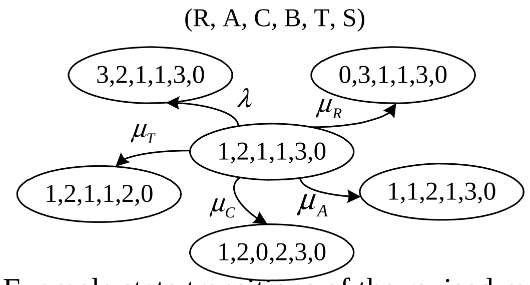


Fig. 6.8. Example state transitions of the revised model.

6.5.2 Differentiated Run Lengths

Run lengths have been shown to be influent on the system performance. Rather than having same run length for Core_A and Core_B whose processing times are different, it is sensible to differentiate them so as to balance the load. As presented in Fig. 9, when T_{S1} is configured as $100\mu\text{sec}$ which is found appropriate previously, the system performance improves as T_{S2} increases, in which largest advance occurs when $T_{S2} = 200$. Nevertheless, given that the processing time for Core_A and Core_B is $31\mu\text{sec}$ and $49\mu\text{sec}$, respectively, the results do not necessarily suggest possible relationship between T_{S1} and T_{S2}.

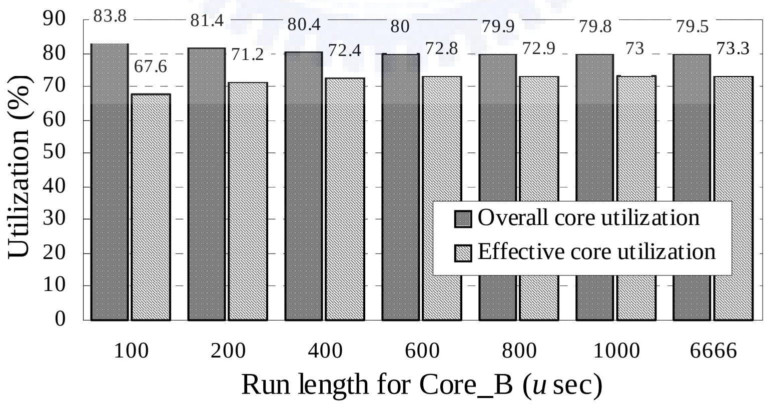


Fig. 6.9. Benefits from differentiated run lengths for Core_A and Core_B. T_{S1} is configured as $100\mu\text{sec}$.

6.5.3 Effect of the Context Switch Overhead

Though context switching is helpful in alleviating the memory access overhead, for computational-intensive applications it could jeopardize the performance, as Fig. 10 explains. From the figure we can learn that a delay of 300μsec leads to low effective utilization (12%) but considerable context switching and busy-waiting burdens (38% and 47%). As the delay reduces, not only does the core utilize effectively but also lessen the overhead. The burden from busy-waiting can even be annihilated when T_D=10 T_{S1} and T_{S2} are further configured to 100 and 200μsec, respectively. Since a context switch delay of 10μsec is quite unrealistic for current XScale core implementation (except for some coprocessors with hardware multithreads [INT04]), this result is also suggesting that system vendors adopt single process for multiple tasks in computational intensive applications.

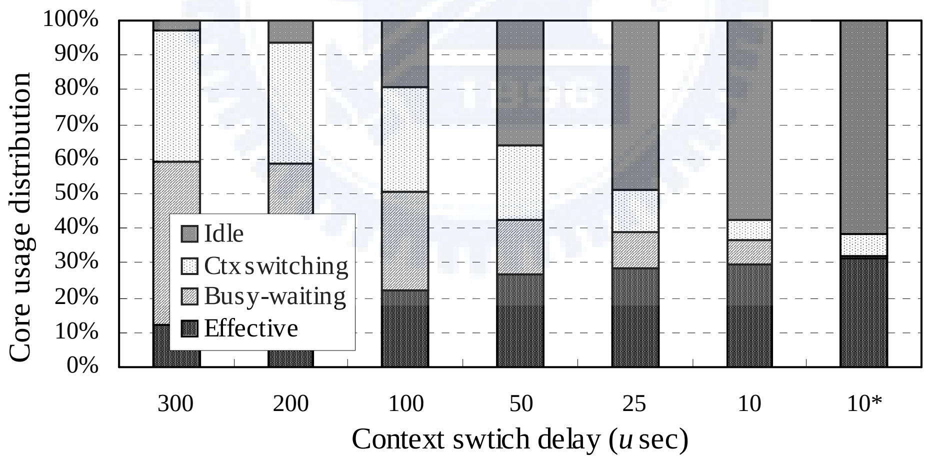


Fig. 6.10. Core usage distribution for different context switch delays. The asterisk means T_{S1} and T_{S2} are configured to 100 and 200μsec.

6.5.4 Benefit from Offloading

Offloading complex, routine tasks to specially design coprocessors has been an alternative to pure speeding up the core processor. However, the benefit from offloading is not well uninvestigated. Figure 11 demonstrates the gain of doing cryptographic operations, which is the most time-consuming task, by (1) multiplying the core clock rate, and (2) offloading to the computational coprocessor. The former includes (1) no speedup and (2) speedup for 2, 4 and 6 times for the core processor, while the latter involves both interrupt-driven and busy-waiting schemes. As revealed in the figure, the throughput increases in direct proportion to the speedups. Nonetheless, the interrupt-driven scheme still outperforms the un-offloaded one equipped with a core of 6-time speedup resembling a 3.2 GHz P4 processor. The busy-waiting scheme also parallels the core of 4-time speedup.

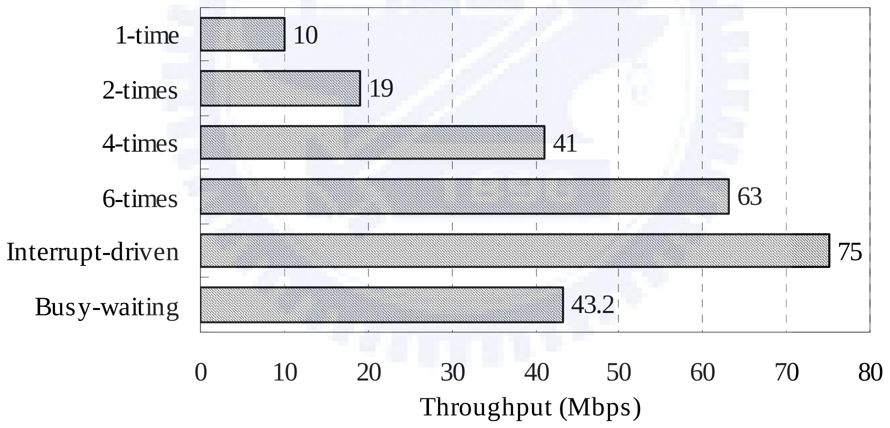


Fig. 6.11. Throughput of various offloading schemes. The clock rate of the XScale core in the implementation is 533MHz, as a reference for comparison.

The performance figures can even be validated as follows. Let the capability of the core be *m* cycles/sec, and the processing time for Core_A, en/de –cryption and Core_B be *x*, *y* and *z* cycles/Mbits, respectively, we can have

$$\frac{m}{x+y+z} = 10\,(\text{Mbps}), \tag{4}$$

since the throughput of an ordinary core without offloading is 10 Mbps. Moreover, because the core, namely XScale in the real implementation, is the performance bottleneck [LLL+05], we can also have

$$\frac{m}{x+z} = T \text{ (Mbps)}, \tag{5}$$

where T represents the throughput of the core executing Core_A and Core_B, and therefore the throughput of the interrupt-driven scheme as well. With (4) and (5) we can have

$$\frac{(5)}{(4)} : \frac{z+y+z}{x+z} = \frac{T}{10}. \tag{6}$$

Since $y:(x+z)=(31+49):12.6 \cong 6.4$, according to Table 2, the throughput T can finally derived as

$$T = 10 \times \frac{1+6.4}{1} = 74 \quad \text{(Mbps)}, \tag{7}$$

which is very much close to the one from the analytical model.

6.5.5 Effect of Limited Buffer Sizes

As pointed out earlier in this section, the limited buffer size does not impact much on the accuracy of the model. This is verified in Fig. 12 which compares two significantly different sizes, 3 and 1000. From the figure we can see that the core utilization is the same for both sizes when input load does not exceed the system capability. The queue length, which is not shown, for the two cases does not grow noticeably, implying that the system is quite tolerant to the variance of the packet inter-arrival time.

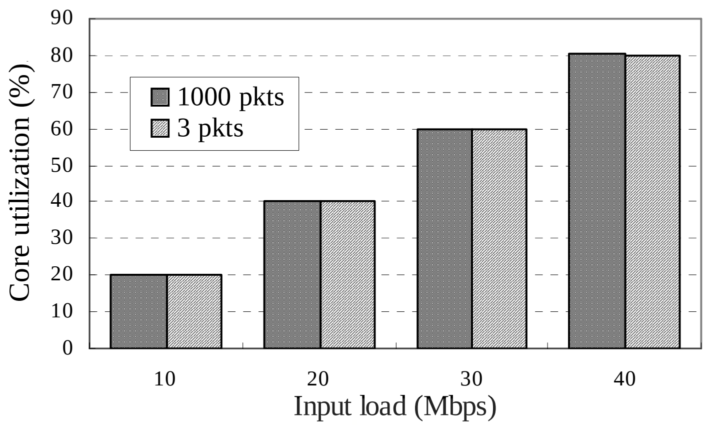


Fig. 6.12. Core utilization under two buffer sizes.

6.6. Summary

This work aims at deriving possible design implications for core-centric network processors by developing an analytical model as well as simulations based on the timed, colored Petri net. The computational intensive VPN application, which has some complex but routine tasks is adopted to explore the benefit from offloading to coprocessors. To date, this work is the first research that practically models the interrupt-driven and busy-waiting schemes over this emerging architecture.

The analytical model is verified to have behaviors quite inline with the simulation (within 1%) and the implementation (within 3%-4%), indicating a satisfactory accuracy for detailed investigation on architectural-level issues which are unlikely to perceive on real implementations. Through both analytical and simulation measures we observe that

- by adopting appropriate process run lengths, 2.26 times improvement on the effective core utilization and 20.5% less consumption on the computational resource can be achieved; better results can be have if run lengths are further differentiated according to the processing time;

- by reducing the context switch delay from 300μsec to 10μsec we can have 2.6 times advance on the effective core utilization, and the switching overhead and busy-waiting time can be alleviated by as much as 90%; this observation also strongly suggests the use of single process for multiple tasks since 10μsec delay is normally unfeasible for today's technology;
- by incorporating coprocessors for bottleneck task, namely the en/de -cryption, the throughput boosts 7.5 times compared to that of single processor;
- under Poisson arrival, the system is quite tolerant to limited buffer size.

We believe the first two findings are useful for system vendors while the others may interest IC vendors. Discovery concluded in this study should be applicable to network processors of similar architecture.

As future work, we plan to extend this approach by considering memory-access intensive applications such as IDP (Intrusion Detection and Prevention). In such extension, memory access operations can be offloaded to coprocessors specifically designed with wide memory bus. To further analyze the potential memory bottleneck, the model can also involve multiple memory modules or multi-port memory supporting concurrent accesses.

Chapter 7

Conclusions

The goals of this dissertation include (1) comparison of the thread allocation schemes in multithreading architecture; (2) design implications and (3) resource allocation strategies, for coprocessors-centric and core-centric network processors implementing different types of applications. For the first, we found that the heterogeneous thread allocation is the best scheme, since the load balance among processors is simple and effective, compared to the homogeneous and the hybrid schemes. It is also resilient to the unbalanced load among threads for unbalance ratios smaller than 1.5. Observations regarding others are categorized and stated as follows.

General NP Design Implications

1. *Number of threads per processor*: For a sensible P-M ratio, i.e. a ratio close to 1 as in the SF/DS over the IXP1200, the most appropriate number of threads is 5, and should be increased/decreased as the ratio decreases/increases.
2. *Solution to memory bottleneck*: For solving the memory bottleneck, if any, adding memory banks best improves the performance, though the effectiveness depends heavily on the data structure of the application/algorithm.

Resource Allocation for Coprocessors-centric NPs Implementing Memory Access Intensive Applications

1. *Most important architectural factor*: Given a certain application and algorithm, the throughput is influenced mostly by the total number of threads as long as the processor utilizations do not exceed 100%.
2. Although enlarging the total number of threads by adding more processors

benefits the throughput, the ME utilization suffers. This is because the load saturating memory is diluted by the increased *I*, meaning that *J*, rather than *I*, should be extended.

3. *Most appropriate (I,J) estimation through bottleneck identification.* The bottleneck is found to be the SRAM as the $I \times J$ exceeds the upperbound *k* that *cost-effectively* utilizes the memory. With the upper-bound, we can always estimate a most appropriate (*I*, *J*) configuration for the application.

Resource Allocation for Core-centric NPs Implementing Computational Intensive Applications

1. *Improvement from offloading*: Offloading from the core processor to the coprocessors improves the overall performance for 7.5 times. Moreover, offloading the crypto processing benefits the throughput more than offloading the Ethernet processing.
2. *Bottleneck observation*: The core tends to be the bottleneck even after offloading.
3. *Effect and implications from run length analysis*: By adopting appropriate process run lengths, 2.26 times improvement on the effective core utilization and 20.5% less consumption on the computational resource can be achieved; better results can be had if run lengths are further differentiated according to the processing time;
4. *Effect and implications from context switch overhead analysis*: By reducing the context switch delay from $300\mu\text{sec}$ to $10\mu\text{sec}$ we can have 2.6 times advance on the effective core utilization, and the switching overhead and busy-waiting time can be alleviated by as much as 90%; this observation also strongly suggests the use of single process for multiple tasks since $10\mu\text{sec}$ delay is normally unfeasible for today’s technology.

Bibliography

[AAP04] S. Antonatos, K. G. Anagnostakis, M. Polychronakis, and E. P. Markatos, "Performance Analysis of Content Matching Intrusion Detection Systems," Proc. of the *International Symposium on Applications and the Internet (SAINT2004)*, January 2004.

[AC75] A. Aho and M. Corasick, "Efficient string matching: An aid to bibliographic search," *Communications of the ACM*, vol. 18 issue 6, P.333-340, 1975.

[ARB02] M. Adiletta, et al., "The Next Generation of Intel IXP Network Processors," *Intel Technology Journal*, vol.6 issue 3, 2002.

[Atk95] R. Atkinson, "Security architecture for the Internet protocol," RFC1825, IETF Network Working Group, August 1995.

[BDE01] W. Bux, W. E. Denzel, T. Engbersen, A. Herkersdorf, and R. P. Luijten, "Technologies and Building Blocks for Fast Packet Forwarding," *IEEE Communications Magazine*, January 2001.

[BGK+99] T. Braun, M. Günter, M. Kasumi and I. Khalil, "Virtual Private Network Architecture," Technical Report IAM-99-001, CATI, April 1999.

[BH04] H. Bos and K. Huang, "A network instruction detection system on IXP1200 network processors with support for large rule sets," Leiden Univeristry Techical Report 2004-02.

[BH95] G. Byrd and M. Holliday, "Multithreaded Processor Architectures," *IEEE Spectrum*, vol. 32 issue 8, 1995.

[CB02] P. Crowley and J.-L. Baer, "A Modeling Framework for Network Processor Systems," Proc. of the *Network Processor Workshop in conjunction with Eighth International Symposium on High Performance Computer Architecture (HPCA-8)*, 2002.

[CFB01] P. Crowley, M. Fiuczynski, and J.-L. Baer, "On the Performance of

Multithreaded Architectures for Network Processors," *UW Technical Report*, October 2001.

[CLS$^+$04] C. Clark, et al., "A Hardware Platform for Network Intrusion Detection and Prevention," Proc. of *the 3rd Workshop on Network Processors and Applications (NP3),* Madrid, Spain, February 2004.

[CM06] D. Comer and M. Martynov, "Building Experimental Virtual Routers with Network Processors," Proc. of the *2nd International Conference on Testbeds and Research Infrastructures for the Development of Networks and Communities*, TRIDENTCOM'06, 2006.

[Com04] D. E. Comer, "Network Systems Design using Network Processors," p. 282, Prentice Hall, 2004.

[CSI] CSIX-L1: Common Witch Interface Specification, http://www.npforum.org/csixL1.pdf.

[DFL05] J.D. Davis, C. Fu, and J. Laudon, "The RASE (Rapid, Accurate Simulation Environment) for Chip Multiprocessors," Proc. of the *Workshop on Design, Architecture and Simulation of Chip Multiprocessors,* November 2005.

[FV02] M. Fisk and G. Varghese, "Applying Fast String Matching to Intrusion Detection," *SEP'02*, 2002.

[FW02] M. Franklin and T. Wolf, "A Network Processor Performance and Design Model with Benchmark Parameterization," in *Network Processor Workshop in conjunction with Eighth International Symposium on High Performance Computer Architecture (HPCA-8)*, February 2002.

[GKS03] M. Gries, C. Kulkarni, C. Sauer, and K. Keutzer, "Comparing Analytical Modeling with Simulation for Network Processors: A Case Study," in *Proc. of the Design, Automation, and Test in Europe (DATE)*, 2003.

[INTa] Intel IXP12XX Product Line of Network Processors, http://www.intel.com/ design/network/products/npfamily/ixp1200.htm.

[INTb] Intel IXP425 Network Processor, http://www.intel.com/design/network/ products/npfamily/ixp425.htm.

[INTc] Intel XScale Microarchitecture, http://www.intel.com/design/intelXScale.

[INT04] IXP2400 Data Sheet, Intel document number 301164-011, February 2004.

[JK03] E. J. Johnson and A. R. Kunze, "IXP2400/2800 Programming– The Complete Microengine Coding Guide," Intel Press, April 2003.

[JS97] M. John and S. Smith, "Application-Specific Integrated Circuits," Addison-Wesley Publishing Company, ISBN 0-201-50022-1, June 1997.

[JS99] M. John and S. Smith, "Application-Specific Integrated Circuits," Addison-Wesley Publishing Company, ISBN 0-201-50022-1, June 1997.

[Kes95] Lawrence Kesteloot, "Porting BSD UNIX to a New Platform," January 1995.

[LCL$^+$07] Y.-N. Lin, Y.-C. Chang, Y.-D. Lin, and Y.-C. Lai, "Resource Allocation in Network Processors for Memory Access Intensive Applications," to appear in the *Journal of Systems and Software*.

[Lek03] P. C. Lekkas, "Network Processors: Architectures, Protocols and Platforms (Telecom Engineering)," McGraw-Hill Professional, ISBN 0071409866, July 2003.

[LHC04] R.-T. Liu, N.-F. Huang, C.-H. Chen and C.-N. Kao, "a fast string-matching algorithm for network processor-based intrusion detection system," *ACM Transactions on Embedded Computing Systems*, vol 3 issue 3, P.614-633, August 2004.

[LJ03] B.K. Lee and L.K. John, "NpBench: A Benchmark Suite for Control Plane and Data Plane Applications for Network Processors," Proc. of the *IEEE Int'l Conf. Computer Design (ICCD 03)*, 2003, pp. 226-233.

[LLP02] S. Lakshmanamurthy, K. Y. Liu, Y. Pun, L. Huston, and U. Naik, “Network Processor Performance Analysis Methodology,” *Intel Technology Journal* vol. 6 issue 3, 2002.

[LLL$^+$05] Y.-N. Lin, C.-H. Lin, Y.-D. Lin and Y.-C. Lai, “VPN Gateways over Network Processors: Implementation and Evaluation,” Proc. of the *11th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'05),* San Francisco, March 2005.

[LLY$^+$03] Y. D. Lin, Y. N. Lin, S. C. Yang, and Y.S. Lin, “DiffServ Edge Routers over Network Processors: Implementation and Evaluation,” *IEEE Network, Special Issue on Network Processors*, July 2003.

[LW06] J. Lu and J. Wang, “Analytical performance analysis of network-processor-based application designs,” Proc. of the *15th International Conference on Computer Communications and Networks (IC3N06),* Arlington, VA, Oct. 2006. IEEE Press, Pages 33-39.

[MOT] Motorola C-5 network processor, http://e-www.motorola.com/.

[Mur89] T. Murata, “Petri Nets: Properties, Analysis and Applications,” *Proceedings of the IEEE,* vol. 77, no. 4, 1989.

[Net] The NetBSD Project, http://www.netbsd.org/.

[NFS04] D. Nussbaum, A. Fedorova, and C. Small, “*An Overview of the Sam CMT Simulator Kit*,” Technical Report of Sun microsystems, June 2004.

[NGG93] S. S. Nemawarkar, R. Govindarajan, G. R. Gao, and V. K. Agarwal, “Analysis of Multithreaded Multiprocessor Architectures with Distributed Shared Memory”, Proc. of *the Fifth IEEE Symposium on Parallel and Distributed Processing*, Dallas, pp.114-121, 1993.

[NSH02] U. Naik, et al., “IXA Portability Framework: Preserving Software Investment in Network Processor Applications,” *Intel Technology Journal*, vol.6 issue 3, 2002.

[POS] POS PHY Level 3 Link Reference Design, http://www.latticesemi.com/products/devtools/ip/refdesigns/pos_phy.cfm.

[PRS04] W. Plishker, K. Ravindran, N. Shah, and K. Keutzer, “Automated Task Allocation on Single Chip, Hardware Multithreaded, Multiprocessor Systems,” Proc. of the *Workshop on Embedded Parallel Architectures (WEPA-1)*, 2004.

[Roe] M. Roesh, “Snort: The open source network intrusion detection system,” http://www.snort.org.

[RJ03] S. T. G. S. Ramakrishna, H. S. Jamadagni, “Analytical Bounds on the Threads in IXP1200 Network Processor,” *Proc. of the Euromicro Symposium on Digital System Design (DSD’03)*, pp. 426-429, 2003.

[RW03] R. Ramaswamy and T. Wolf, “PacketBench: A Tool for Workload Characterization of Network Processing,” Proc. of the *6th IEEE Annual Workshop on Workload Characterization*, 2003.

[RWL$^+$03] A. V. Ratzer et al., “CPN Tools for Editing, Simulating, and Analysing Coloured Petri Nets,” Proc. of the *International Conference on Applications and Theory of Petri Nets*, 2003.

[S-BCE90] R. S-B, D. Culler, and T. Eicken, “Analysis of multithreaded architectures for parallel computing,” Proc. of the *2nd Annual ACM Symposium. on Parallel Algorithms and Architectures*, 1990.

[SKP01] T. Spalink, S. Karlin, L. Peterson, and Y. Gottlieb, “Building a Robust Software-Based Router Using Network Processors,” Proc. of the *18th ACM Symposium on Operating Systems Principles (SOSP)*, 2001.

[SMA03] K. Skadron, M. Martonosi, D. August, M. Hill, D. Lilja, and V. S. Pai, “Challenges in Computer Architecture Evaluation,” *IEEE Computer*, 2003.

[SPK 03] Niraj Shah, William Plishker, Kurt Keutzer, “NP-Click: A Programming Model for the Intel IXP1200,” Proc. of the *2nd Workshop on Network Processors (NP-2)*, held in conjuction with the *9th*

International Symposium on High Performance Computer Architecture (HPCA), 2003.

[TLY⁺04] Z. Tan, C. Lin, H. Yin, and B. Li, “Optimization and Benchmark of Cryptographic Algorithms on Network Processors,” *IEEE Micro*, vol. 24, no. 5, pp. 55-69, 2004.

[WF00] T. Wolf and M. Franklin, “CommBench: A Telecommunication Benchmark for Network Processors,” *Proc. IEEE Int’l Symp. Performance Analysis of Systems and Software (ISPASS 00),* IEEE Press, 2000, pp. 154-162.

[WF06] T. Wolf and M. K. Franklin, “Performance Models for Network Processor Design,” *IEEE Transactions on Parallel and Distributed Systems*, Vol. 17, No. 6, pp. 548-561, June 2006.

[WM94] S. Wu and U. Manber, “A fast algorithm for multi-pattern searching,” Technical Report TR94-17, Department of Computer Science, University of Arizona.

[WT01] T. Wolf and J. S. Turner, “Design Issues for High- Performance Active Routers,” *IEEE Journal on Selected Areas in Communications*, vol. 19, no. 3, 2001.

[ZGF98] W. M. Zuberek, R. Govindarajan, F. Suciu, “Timed Colored Petri net Models of Distributed Memory Multithreaded Multiprocessors,” Proc. of the *Workshop on Practical Use of Coloured Petri Nets and Design*, pages 253-270, Aarhus University, June 1998.