Optimizing Energy-Efficiency for Multi-Core Packet Processing Systems in a Compiler Framework

by

Jing Huang, B.Eng.

Submitted in partial fulfilment of the requirements
for the Degree of MEng

Dublin City University
School of Electronic Engineering
Supervisor: Dr. Xiaojun Wang and Prof. Bin Liu
September 2012

DECLARATION

I hereby certify that this material, which I now submit for assessment on the programme of study leading to the award of MEng is entirely my own work, that I have exercised reasonable care to ensure that the work is original, and does not to the best of my knowledge breach any law of copyright, and has not been taken from the work of others save and to the extent that such work has been cited and acknowledged within the text of my work.

Signed: 

Jing Huang (Candidate)

ID No.: 57125597

Date: 7 September 2012

ACKNOWLEDGMENTS

I would like to thank my supervisor Dr. Xiaojun Wang for giving me the opportunity to undertake this research programme and helping me heartfully throughout the past years. My gratitude also goes to my colleagues in Network Processing Group for the enlightening advices they gave and the pleasant environment they created. I would also like to thank Prof. Bin Liu in Tsinghua University who also supervised me during my stay in his group and Dr. Olga Ormond, the research officer in Network Innovation Centre in RINCE, who presented me with numerous academic advices here and there.

I am also extremely grateful to my parents who always support me with their endless love. Whenever I am depressed, their encouragement is the best remedy to pull me back on track.

TABLE OF CONTENTS

ABSTRACT

Optimizing Energy-Efficiency for Multi-Core Packet Processing Systems in a Compiler Framework

Jing Huang

Network applications become increasingly computation-intensive and the amount of traffic soars unprecedentedly nowadays. Multi-core and multi-threaded techniques are thus widely employed in packet processing system to meet the changing requirement. However, the processing power cannot be fully utilized without a suitable programming environment. The compilation procedure is decisive for the quality of the code. It can largely determine the overall system performance in terms of packet throughput, individual packet latency, core utilization and energy efficiency.

The thesis investigated compilation issues in networking domain first, particularly on energy consumption. And as a cornerstone for any compiler optimizations, a code analysis module for collecting program dependency is presented and incorporated into a compiler framework. With that dependency information, a strategy based on graph bi-partitioning and mapping is proposed to search for an optimal configuration in a parallel-pipeline fashion. The energy-aware extension is specifically effective in enhancing the energy-efficiency of the whole system. Finally, a generic evaluation framework for simulating the performance and energy consumption of a packet processing system is given. It accepts flexible architectural configuration and is capable of performing arbitrary code mapping. The simulation time is extremely short compared to full-fledged simulators. A set of our optimization results is gathered using the framework.

LIST OF FIGURES

LIST OF TABLES

LIST OF ACRONYMS

ADAG – Annotated Directed Acyclic Graph

ASIC – Application Specific Integrated Chip

ASIP – Application Specific Instruction-set Processors

AST – Abstract Syntax Tree

BB – Basic Block

Bi-Par – Bi-Partitioning

CD – Control Dependence

CDG – Control Dependence Graph

CFG – Control Flow Graph

CFK – Compiler Known Functions

DAG – Directed Acyclic Graph

DD – Data Dependence

DDG – Data Dependence Graph

DF – Dominance Frontier

DiffServ – Differentiated Service

DPI – Deep Packet Inspection

DRAM – Dynamic Random Access Memory

GPP – General Purpose Processors

IDPS – Intrusion Detection and Prevention System

IPTV – Internet Protocol Television

ILP – Integer Linear Programming

IP – Internet Protocol

IR – Intermediate Representation

ISA – Instruction Set Architecture

LAN – Local Area Network

ME – Micro-Engines

Mpps – Million packets per second

MPSoC – Multiprocessor System-on-Chip

MTU – Maximum Transmission Unit

NAT – Network Address Translation

NP – Network Processors

NPU – Network Processing Unit

OSI – Open System Interconnect

PDG – Program Dependence Graph

PDT – Post-Dominator Tree

PE – Processing Engine

QoS – Quality of Services

RAM – Random Access Memory

RISC – Reduced Instruction Set Computer

SDK – Software Development Kits

SSA – Static Single Assignment

TCP/IP – Transmission Control Protocol/Internet Protocol

TTL – Time To Live

UDP – User Datagram Protocol

VLIW – Very Long Instruction Word

VoIP – Voice over Internet Protocol

VPN – Virtual Private Network

WAN – Wide Area Network

LIST OF PEER-REVIEWED PUBLICATIONS

Jing Huang, Xiaojun Wang and Bin Liu, "Energy-aware Compilation for Network Processors: Frameworks, Techniques and Trend", *China-Ireland International Conference on Information and Communication Technologies*, 26th-28th Sep., 2008

Jing Huang, Xiaojun Wang, "Program Dependence Graph Generation and Its Use in Network Application Analysis", *CIICT 2009*, Aug. 2009.

Jing Huang, Olga Ormond, Di Ma and Xiaojun Wang, "Optimizing Energy-Efficiency for Program Partitioning and Mapping onto Multi-Core Packet Processing Systems," *the Journal of China University of Posts and Telecommunications*, June 2012, 19(Suppl. 1), pp. 79-86.

Chapter 1 - Introduction

Since the world steps into the Information Age, information, the subject it is named after, becomes increasingly valuable to the society. As such, networks grow to be indispensable nowadays. Most significantly, the past decade witnessed the explosive growth of the "global network of networks", i.e. the Internet. At the beginning, most networks are based on variations of simple store-and-forward packet switching architecture [1]. The interconnection nodes known as routers usually just forward the packets without further processing. With the advent of service-centric networks [2], a large portion of the computation and processing workload is handed over from end hosts to the edge networks and access networks. Unlike traditional routers, devices in such an environment should not only simply deliver packets, but also process them at the same time. To meet the changing requirements of the services, these devices have to be easily programmable and configurable; and to keep pace with the soaring line speed, they should be well powerful to process the packets within an extremely short time scale. Packet processing systems are therefore specifically proposed to perform this type of tasks. Packet processing systems usually employ multiple processing cores run in parallel in data plane to satisfy the computation demand; and the cores are typically variations of Reduced Instruction Set Computer (RISC) processors that can be easily programmed via specific software development toolsets. Like General-Purpose Processors (GPP), the computational power can only be effectively utilized with well-written software. It implicates the vital importance of the complier and relevant runtime management tools in packet processing systems, which translate the high-level code and deploy them onto the underlying heterogeneous architectures. On the other hand, network processor has distinctive requirements from general processors. Not all compilation optimizations derived

from general processing techniques would be still valid; and more attention should be paid with regard to the high parallelism in multi-core architecture and the soaring energy consumption.

The thesis covers the literature review on this topic and the concrete work on optimizations for a multi-core packet processing system in a compiler framework, especially with the awareness of energy-efficiency.

1.1 Motivation

The role of a compiler is always a bridge between the programmers and the underlying system hardware. It is not an exception in the network processing domain. A compiler designed for the packet processing system should provide a decent interface for network application developers and map the high-level user codes onto the complex Application Specific Instruction-set Processors (ASIPs). Fig. 1.1 illustrates a commercial network processor from Intel. It has 16 Micro-Engines, a nickname for processing cores given by Intel, running in parallel [3]. The whole system is a comprehensive solution to process soaring line-rate traffic as well as to develop applications more flexibly. The features concluded from the diagram are,

- Parallel fast processing units
- Application-specific instruction set processors
- Heterogeneous system-level architectures
- Hierarchical memory sub-systems

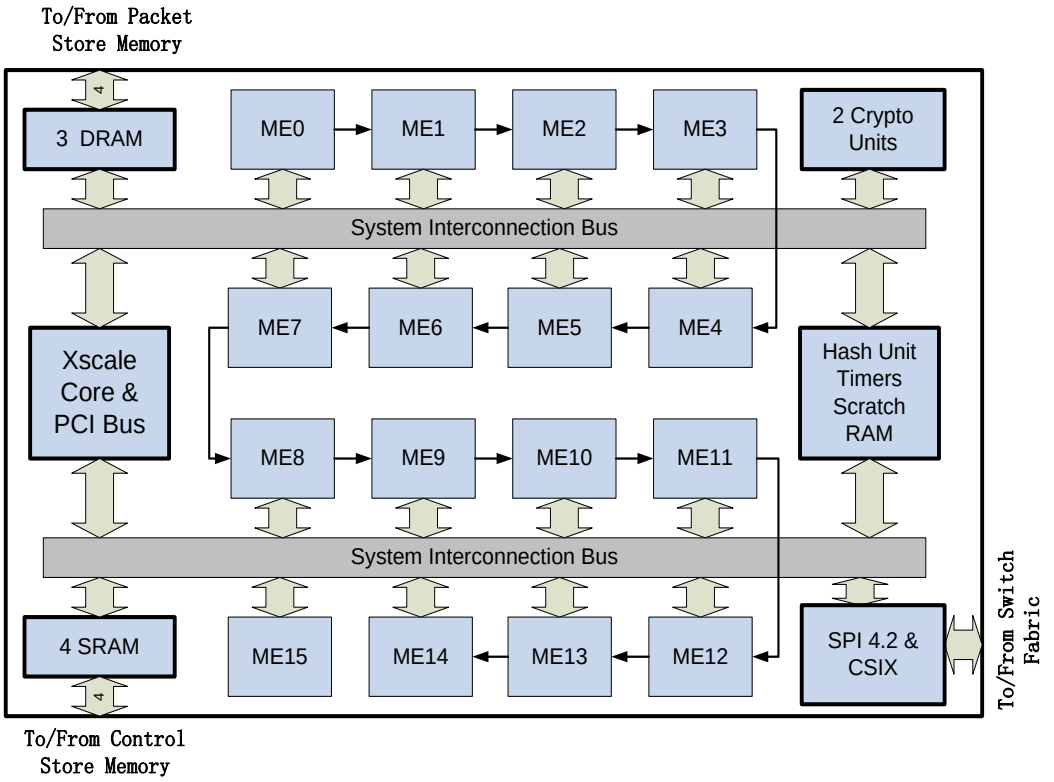

Fig. 1.1. Intel IXP2805 network processor architecture [3]

Compiler technique is an aged research topic, dating back to the 1960s. Yet it is still very active since both the programming languages and hardware platforms incessantly evolve. For example, the problems about re-targetable compilers, just-in-time compilation and inter-procedural pointer analysis were hardly envisioned twenty years ago. A classical compiler would execute a sequence of tasks in sequence, namely pre-processing, lexical analysis, syntax analysis and validation, semantic transformation, Intermediate Representation (IR) optimization, code generation and machine code optimization etc. Given the features of packet processing systems, the compilers in this domain have the following distinctions,

- Defining user-friendly interface that ease packet-processing applications programming
- Partitioning parallel tasks and mapping them onto heterogeneous processing elements
- Bit-stream data (packets) management in multi-level memory subsystems

Fig. 1.2 depicts a typical workflow of the compilers for packet processing systems.

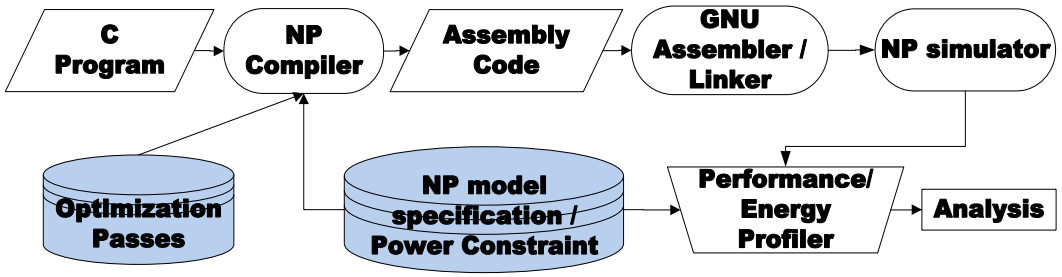

Fig. 1.2. Typical design flow of network processor compiler

The large number of multi-cores in packet processing system resembles the hardware platform of Multiprocessor System-on-Chip (MPSoC), whose architecture is also being heavily researched these days. In a recent paper [4], Leupters et al. indicated that new methodologies, tools and description languages are still required to fill in the gap for the following missing tasks,

- optimal task partitioning
- code generation tools for software production and software maintenance
- simulation and debugging environment

The heterogeneity of both MPSoCs and packet processing systems asks for optimized choices regarding code partitioning, task-to-processor assignment and on-demand task migration.

Energy-efficiency is another heated issue in designing next-generation network. Previous researches on the greenhouse impact of the switching and data storage equipment on the Internet have revealed great potential for power-aware optimizations [5]. A compiler can also play a part in such a process by either providing the system with instruction trace information or interacting with the runtime component, such as power-gating the functional units [6]. The optimization from the point of energy-efficiency would vary from existing techniques that are solely performance-oriented. That said, the space for design exploration in energy-efficient compilation is quite vast.

Bearing these in mind, the author chose the topic of energy–aware compiler optimizations in multi-core processing systems. The ultimate goal is to develop an energy-efficient compiler optimization framework for packet processing systems.

1.2 Contributions

In this work, a comprehensive solution for network application code analysis, task partitioning, task-to-core mapping and a simulation environment is proposed and validated. The major contributions of the thesis are listed as below.

- Task Partitioning and Mapping

A recursive Bi-Partitioning based algorithm is proposed that consider both computational cost and energy consumption. In the partitioning and mapping stage, additional optimization and refinement steps are taken to specifically enhance the energy-efficiency, which is not investigated before in literature to the best of the author's knowledge. The simulation results show that the method is particularly effective in improving the energy-efficiency compared to the existing solutions.

- Energy-Aware Simulation Framework for Network Processing System

The thesis provides the design of a simulation tool to benchmark the performance and energy-consumption of a generic network processing system. State-of-the-art tools are either execution-driven, too complicated to run the simulation fast enough in a large search space, or too simple to provide the energy-related data. The proposed framework is built on an analytical model and takes both computation and energy parameters into consideration. Its effectiveness and validity are carefully examined and verified in this work.

- Implementation of a Program Dependence Analysis Tool

The program dependence information is of vital importance in deciding the quality of task partitioning and mapping. In this work the author implemented a code analysis tool to collect the program dependence information of a code block within a full-fledged compiler framework. The tool is built into that framework as a plugin and has well defined interfaces interacting with partitioning and mapping modules. All the modules together can work as a complete tool chain.

1.3 Structure

The structure of the thesis is as follows. In chapter 2, the packet processing system at which the work targets is briefly reviewed. The compilation in packet processing domain is reviewed in literature from two aspects in this chapter as well, namely packet processing support and energy-efficiency related issues. The author briefly described some challenges in network processor compiler design and implementations. In chapter 3, the work on program dependence analysis is presented. A compiler module is implemented to generate the program dependence graph. It is the footstone for further compiler optimizations. Chapter 4 presents an energy-aware approach for program partitioning and mapping the author used to explore the system at the architecture level. Detailed results and analyses are given as well. Chapter 5 describes the performance and energy evaluation model that the author uses to simulate a generic multi-core packet processing system. A set of simulation results are provided. Finally in chapter 6, the conclusion and future research fields are introduced as a closure to the thesis.

Chapter 2 - Background on Packet Processing System and its Compilation

2.1 Introduction

As introduced in chapter 1, the network, especially the Internet, has already been an essential infrastructure for the modern world. Yet the scalability and complexity of today's Internet still evolves rapidly. It was estimated that more than a quarter of the world population use Internet services as of 2011. Beyond traditional text-based network applications, versatile services come into common use such as Voice over Internet Protocol (VoIP), web conferencing, Internet banking and Virtual Private Network (VPN) etc. With the decreasing cost of computation power and the advancement in distributed computing and virtualization, a lot more applications are becoming web-based and making use of the computation power on the network [7]. For instance, ten years ago, when two authors collaborate on a book, one of them might open a new Microsoft Word document, fill it in and send it via emails. The other collaborator would have to write down the reviews on that particular document and send it back and forth. It is easily messed up with multiple versions of the same file. Fortunately now with the maturity of online platform such as Google Docs and Microsoft Live, people can work on the same document simultaneously without any local storage. Recognizing such a trend, the research on the next generation Internet emphasizes the importance of programmable components of the network [8].

The packet processing system is such a programmable platform designed to meet the explosively growing need for higher line-rate processing. The core of the system is Network Processors (NPs), featuring specially tailored RISC Instruction Set Architecture (ISA) and parallel multi-core architecture. In this chapter, a brief

review on the packet processing system embedded with network processors will be given.

Like any other programmable hardware device, the performance and efficiency would largely depend on the quality of the software running upon it. The focus of this research is to optimize the system from a compiler's perspective. In this chapter, the author also presents a literature review on the compilation techniques for packet processing systems. Finally the challenges in compiler design and implementation are explained and a conclusion is given.

2.2 Multi-Core Packet Processing System

As stated above, the packet processing system is specifically designed for dealing with the network traffic. Most networks, such as the Internet, are distributed and layered systems composed of hosts, workstations, switches and routers etc. Bits of information are encapsulated in packets and flow in the network. And for packet routing and processing, a protocol is specified describing the packet format. Take the Internet for example once again. Internet Protocol (IP) is the core for manipulating data transmission in it. Any type of network, no matter it is Local Area Network (LAN), Wide Area Network (WAN) or even LAN Wireless which is not envisioned when Internet first came into being decades ago, can be all integrated into the Internet as long as they comply with IP (currently version 4 or 6) in packet encapsulation.

Today's network can be generally divided into three layers, namely Core Network, Edge Network and Access Network [9]. The topology of the Internet is illustrated in Fig. 2.1. Core Network consists of gigabit and terabit routers that are backbone of the Internet. The line-rate of network traffic routing is highest in core network. Edge network sits at the boundary of one network to others. The processing speed of edge equipment falls behind those in core network. Finally the access network connects the terminals of a customer endpoint. And usually the bandwidth and line-rate requirement is lowest among the three.

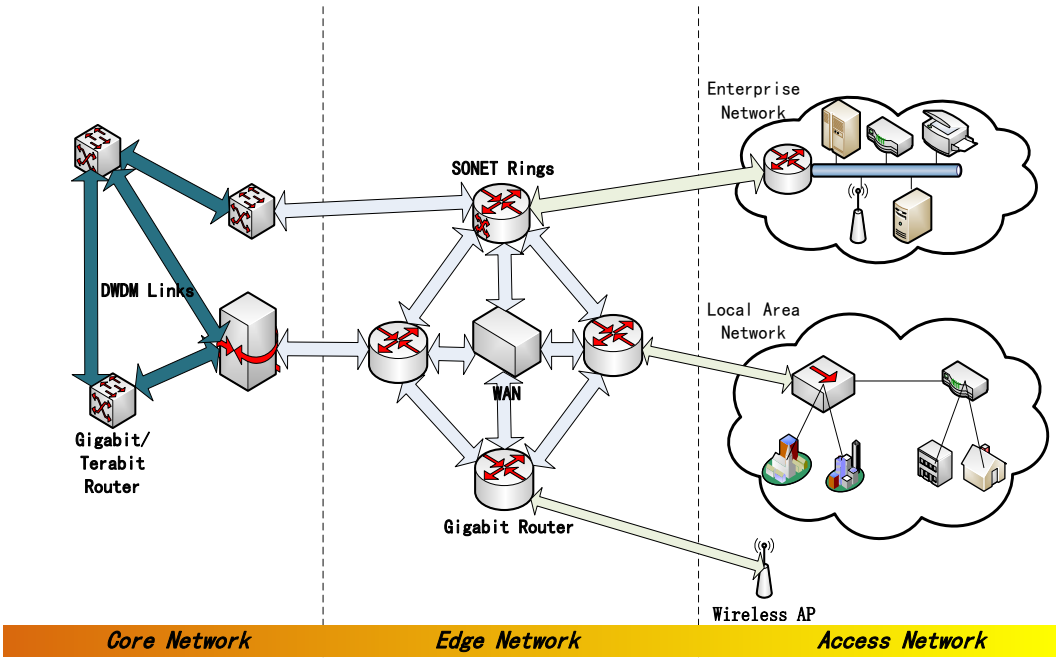

Fig. 2.1. Internet topology

The performance requirement and flexibility vary among equipment in different layers [10]. Table 2.1 gives a qualitative analysis in this regard. The core equipment, like terabit router, may route packets at 39,813 Mbps line-rate within OC-768 SONET link for example [11]. Yet for the edge network device, the line-rate falls slower but the diversity of applications increases. Control-plane and

Table 2.1. Comparison of the three network layers

	Performance	**Flexibility**	**Example**
Core	High	Low	Terabit Packet Routing
Edge	Medium	Medium	Load Balancing
Access	Low	High	Wireless Access Point

even management-plane applications are common to see at this layer. And the complexity of network applications culminates at access layer, such as packet encryption, load balancing in access router, packet inspection and network address translation etc. Fortunately, the traffic in this layer tends to be much lighter in most cases since the total endpoints are limited. Access equipment could trade the processing speed for code complexity to meet the stringent packet latency requirement.

The packet processing system can be equipped in any layer of the network, either in the high-end core routers or in the LAN switches. The flexibility of the system comes from the programmable elements within it, i.e. NPs. And a series of stacked network protocols guarantee its capability to achieve the performance specification.

2.2.1 Network Applications & Network Processors

In order to obtain further understanding of the interaction between the software and hardware of the packet processing system, a profile of the network applications and the network processors is given below.

As mentioned before, the Internet is built upon a stack of rigidly defined protocols, especially TCP/IP. Network applications process the workload in the way specified by the protocol. TCP/IP model defined in RFC1122 [12] describes a five-layer framework for computer network protocols, which has been continuously employed in Internet from its very origin. The International Organization for Standardization formally proposes a more prescriptive model, i.e. *Open Systems Interconnection* model (OSI model). Both models divide the networks into layers, with each layer utilizing the data from the layer immediately beneath and providing service for the layer directly above. The layered architecture exemplifies the principles of the modern network design — end-to-end communication and robustness in implementation. Fig. 2.2 represents both models and correlations in between.

- Physical layer defines the medium over which signals travel, e.g. electrical or optical fibre
- Data link layer provides point-to-point link between two network nodes and protects against data corruption
- Network layer enables transmission of data packets by routing through intermediate network device
- Transport layer provides end-to-end communication services for upper layers, like connection-oriented data stream support, reliability and flow control.

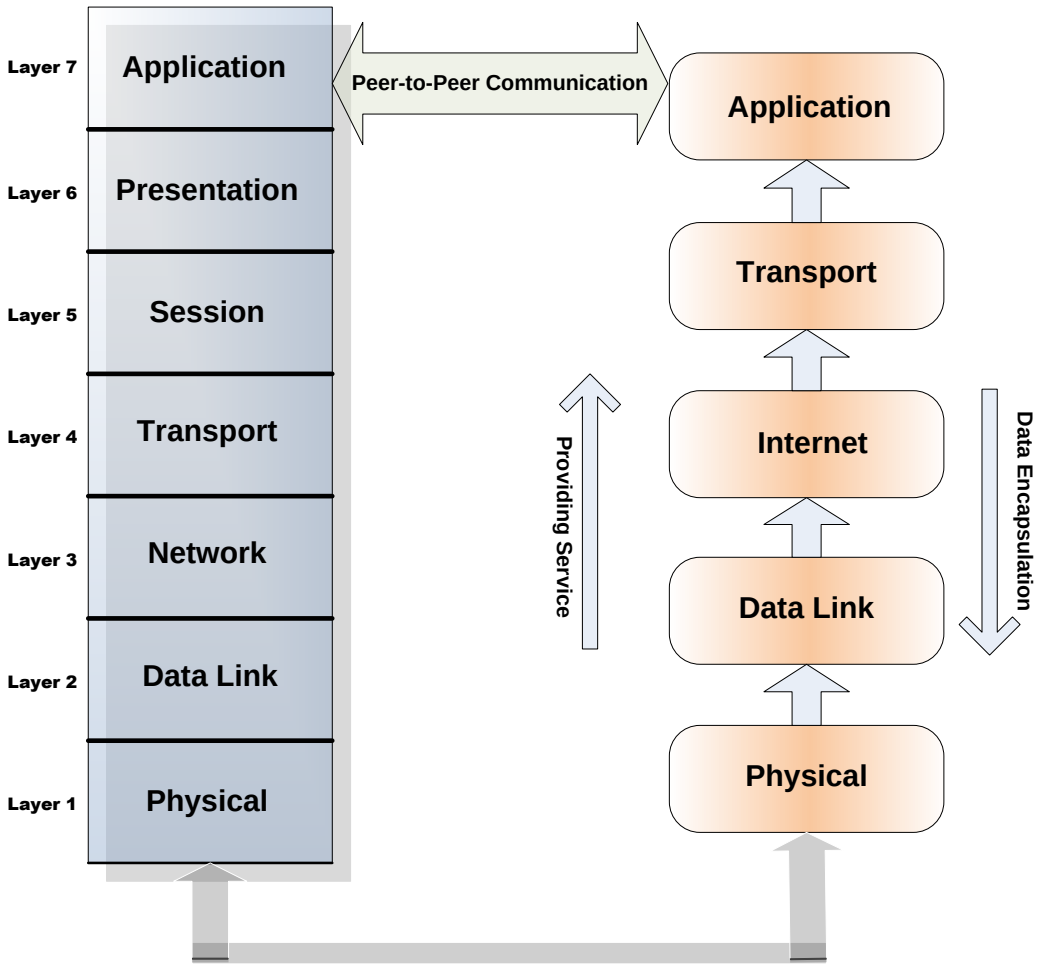

Fig. 2.2. OSI model & TCP/IP model

The most distinguished difference between the two models is that the top three layers defined in OSI – the Application Layer, the Presentation Layer and the Session Layer are not separate but combined in a single layer in TCP/IP. Because TCP/IP model evolves in line with the practice of the Internet, it is less attached to strict layering.

A number of protocols are specified to facilitate data encapsulation and transmission in a specific layer, e.g. TCP in transport layer and IP in network layer. Network processing system usually accesses layer 3- 4 information and process it based on protocol standards. One example application is the IPv4 router in core network. The protocol stack is presented in Figure 2.3. In a core router, the data-plane network processor inspects the IP packet header for destination address and performs a table lookup to determine which output port the packet should be sent to. This is a layer 3 application which features very high packet rates but essentially little inter-packet dependency. Another instance of network

applications is a packet classifier in access network devices. The classifier maps a packet to one of a finite set of flows using a 5-tuple, i.e. source and destination IP addresses, source and destination port numbers, protocol number. It makes use of Layer 3-4 header information, specifically IP header and TCP header that are illustrated in Fig. 2.3.

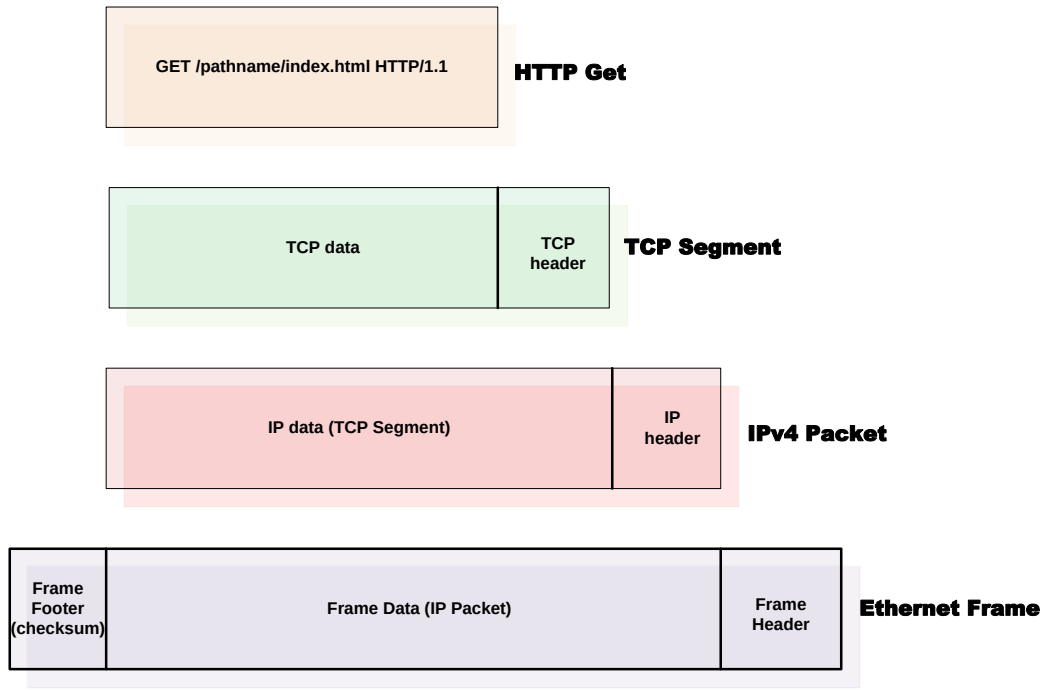

Fig. 2.3. Protocol stack through TCP/IP layers

Other typical network applications deployed in a network processing system include gateway applications such as *Network Address Translation* (NAT), *Quality of Service* (QoS) applications like Usage-based Accounting and *Differentiated Service* (DiffServ), *Intrusion Detection and Prevention System* (IDPS) and layer 7 peer-to-peer networking.

2.2.2 Advantages of Packet Processing System

To fit in with high line-rate data processing, Packet processing systems are specifically tailored in architecture level and instruction-set level, ensuring certain advantages in network processing system. This section outlines a brief analysis of the benefits.

Parallel Processing

A parallel architecture is ideally suited for high speed packet processing. Traffic stream in most network applications render little or no inter-packet dependency.

Hence, the line-rate can be linearly increased to a certain extent by loading parallel processing streams.

Two configurations are prevalent in parallel processing: pipelined and parallel. They are both used in commercial products [13][14][15]. The detailed comparison and explanation of the two configurations will be given in the next section together with compiler specific issues.

Flexibility

Packet processing system incorporates a flexible architecture where heterogeneous hardware components can be easily interfaced. Based upon the RISC-like processing cores, new data-plane applications can be quickly developed, which is infeasible using pure *Application Specific Integrated Circuits* (ASIC) solutions. And to accelerate common computation-hungry networking tasks, special hardware is extensively used in packet processing system. For example, a co-processor specialised in packet classification can be niched in the pipeline before performing any core applications. A special functional unit like checksum and hash unit is also available in commercial products like Intel IXP network processors.

Cost Effectiveness

Traditional network devices using ASIC solutions suffered greatly when it comes to the issue of cost. Firstly, the services that ASIC-based system can provide are pre-defined and difficult to change. If one manufacturer plans to release a series of products from low-end routers excelling in simple Network Address Translation to high-end systems carrying out complex *Deep Packet Inspection* (DPI), a number of circuits have to be synthesized, costing tremendous human efforts. However, in a network processor based packet processing system, the principal design remains intact while only software-level configurations and modifications are required. The shortened time to market also implicates the enhancement in cost effectiveness for network processors. Secondly, RISC-like processor cores cost much less than the complex ASIC design [10]. The expense to test and verify an ASIC design is predominant whereas for RISC-like processors the cost is largely amortised by the mass production.

2.2.3 Trends in Packet Processing Systems

Since the influence of the Internet grows exponentially, the complexity of packet processing system increases accordingly. Several trends were observed in this area.

Firstly, the functionality of network applications becomes multi-fold and ranges across layers. Take the application of deep packet analysis and inspection in a network security system for example. The application scrutinizes not only traditional layer 3-4 header information like source/destination IP addresses pair and port numbers, but also high-level data known as layer 7 in OSI model. The multi-layer applications of this kind are expected to be common in the future [16], which requires more flexibility from the network processing system end as well as added computational power.

Secondly, the heterogeneous parallel architecture is becoming a standard configuration for network processing systems to deal with the fast exploding data rates. Multi-core processor technology is becoming mature for GPP core manufacturing in the past decade and consumers are already benefiting from it [17]. Network processing system can utilize multi-core RISC cores to perform a highly paralleled processing in data-plane applications. For instance, very recently NetLogic Microsystems announced its flagship multi-core processor which has 128 CPUs capable of 240 million packets per second [18]. Besides, the progress on the development of ASIC-based co-processors, i.e. hardware accelerators has enabled the adoption of heterogeneous elements in the system to execute either algorithm-specific or task-specific functions.

Last but not least, the flexibility of network processors could never be realized without well-rounded software development kits (SDK). A typical SDK usually includes a patched operation system (e.g. Linux kernel), compiler tool-chain (pre-processor, interpreter, linker and loader etc.), debugger and documentations. Current solution in system level design embodies defining an application domain first, and then architecting the platform using software-hardware co-design techniques, e.g. LISAtek SDK and MAPS described in [19]. It is an open question that how the partitioning and mapping of applications can be best achieved. Besides, when programming in such a highly parallel multi-processor

environment, the problem that how the application code should be mapped onto the heterogeneous hardware platform is yet to be answered as well. This particular issue is the research interest of the author and the focus of the thesis.

2.3 Compilers for Packet Processing Systems

This section presents the literature review on the compilation techniques for the packet processing systems. As noted in previous sections, the research on compiler techniques for network processors has been a hotspot to tackle with the flexibility requirement in network application development and the computational needs for processing rapidly increasing line-rate data. Bit-stream-oriented programming, multiple processing units and heterogeneous architectures all make the job of an NP compiler complicated. Meanwhile, energy efficiency has also become a heated issue while parallel NP system is becoming more powerful and power-hungry. The text below elaborates on the issues of packet processing support and the energy-aware optimizations among all compilation stages.

Similar to most traditional GPP compilers, a NP compiler is generally partitioned into front-end and back-end modules, and linked by one generic *Intermediate Representation* (IR). To a great extent, it is possible to port and utilize various re-targetable compilation frameworks, such as SUIF [20]. The design of re-targetable compiler frameworks could be illustrated as in Fig. 2.4. Yet the implementation of NP compiler distinguishes itself by domain specific features, i.e. bit-stream-oriented packet manipulation and parallel task processing, both in software (source code) and hardware ends (code generator). Hence a review of several approaches that take these features into account is conducted.

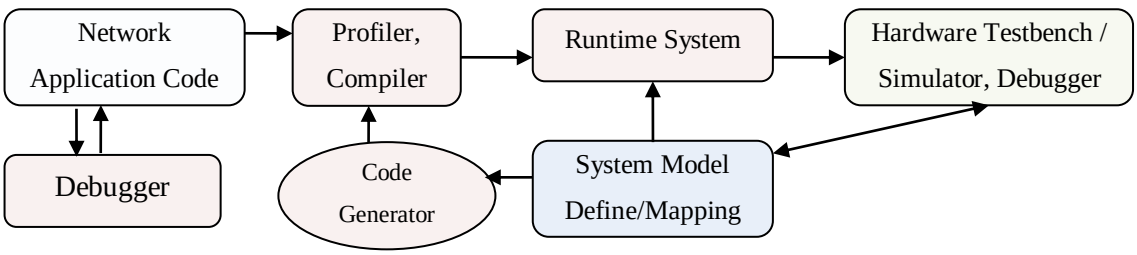

Fig. 2.4. Typical design flows of re-targetable compilation

2.3.1 Support for Packet Processing

The instruction sets of the network processor cores are often specially tailored to provide bit-field operations, e.g. finding the first bit set in a register instruction in Intel IXP ISA. In search of a solution to map high level programming language code into bit-field operations, generally two directions could be taken. One is to extend the capability of a compiler for an existing programming language, and the other is to revisit the language itself as well as to build a domain specific compiler for this language extension. Both have been explored primitively in academia and in industry.

In [21], Wagner and Leupers described the implementation of a C language compiler for an industrial NP, Infineon NP whose architecture includes special register arrays and extensions for bit-level data access. In order to fully exploit this processor's ISA feature, the C compiler employs the use of *Compiler Known Functions* (CKF) and renewed register allocation methods for efficient bit addressing. The main idea of using CKF is to make bit-level packet manipulation visible to programmers, similar to the #PRAGMA directive in ANSI C. The compiler is then responsible for mapping function calls to CKFs into a fixed instruction or a sequence of instructions. As such, the code complexity and programming difficulty in writing lots of bit operations is hidden. Yet unlike conventional C function calls and in-line assembly, there is little overhead on either hardware or programmer's side.

However, CKF is not quite portable with the existence of machine-specific compiler intrinsics. More advanced code selection techniques are yet needed. Budiu and Goldstein presented a compiler algorithm on exact bit-level data-flow analysis in [22] by using a bit-value lattice. Following this bit-value inference analysis approach, Wagner and Leupers evolved their C compiler by replacing the CKF with tree-pattern matching grammar to handle bit-packed addressing in C [23]. The code generation employs the bit-level data-flow analysis information of a basic block labelled by a lattice-string and detects bit-level packet operations in a tree-pattern matching grammar. And with the aid of mature code-generator generator tools, dynamic cost-functions could be modelled as well. Though the code quality is not totally comparable to hand-optimized code in this approach, it

did extricate NP programmers from diverse architectural complexities. And by taking this path it is able to generate a primitive but integral “compiler” in terms of its classical definition.

Instead of conforming to ANSI C language rigidly, in [24, 25] a novel programming language Baker was proposed as part of the NP compilation framework named Shangri-la. The framework consists of a hierarchical compiler and an adaptive run-time system in addition to Baker. Baker is designed to be a modular, packet-processing-oriented, C-like interface, and Shangri-la takes the responsibility to automatically target Baker programs at heterogeneous platforms via the specially optimized compiler and runtime resource management. The compiler itself leverages a lot of code base from the ORC project [26], originally targeting Itanium family but innately re-targetable with well-defined annotated IR named WHIRL. Most of the Shangri-la components are therefore independent of specific hardware. The front-end Baker, and the profiler inside it, and a full-fledged pipeline module are all portable at the expense of slight modifications. As for the back-end, i.e. the code generator and the runtime system, needs to be manually ported by taking the system model as parameters. This framework has already been validated, and functioning on IXP 2XXX NP family. Though it is not applicable to test the performance of Shangri-la directly, it is believed that the re-targetable model and the general ideas behind it are enlightening when dealing with the difficulties of packet processing tasks and managing heterogeneous NP architectures.

2.3.2 Support for Parallel Processing

Besides specialized processing cores dealing with the packet data, an NP, as stated in the introductory section, needs to process large traffic in short time intervals, and the architectures are hence preferably to be of a high parallel structure [27]. For instance in Intel IXP 2400 NP [28], 8 *micro-engines* (ME) are installed for packet processing, each ME supports 4 or 8 threads and low-overhead context switching. Additional co-processors, e.g. Packet Classification Unit, as well as multiple hierarchical memory modules are usually included to reinforce parallel packets processing in network processing system. The compiler is therefore left with the intricate job to manage the heterogeneous resources. A high-level

trade-off in allocating the fixed number of processing cores is on choosing either parallel or pipelining model, as shown in Fig. 2.5. Note that due to the intrinsic parallelism of network applications, or rather independent packets data and/or traffic flows, the same task (sub-program) could be run in parallel. It implies an optimal scheduling would possibly be a hybrid of the two configurations. Briefly NP compilers should take on the burden of program partitioning, resource scheduling and data mapping etc. to support parallel packets processing.

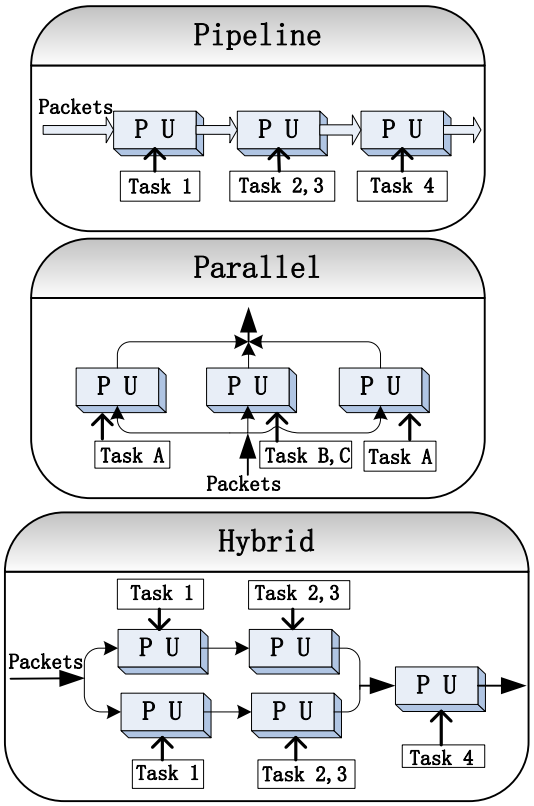

Fig. 2.5. Topology of architectural configurations

Several heuristic approaches have been reported to solve the partitioning and mapping problem in a multi-threaded multi-processor environment. Jia Yu proposed the *Resource Balanced Bi-partitioning* algorithm for the program mapping to achieve higher peak throughput [29]. Its top objective is to balance pipelining through setting appropriate stage numbers. The task graph of a program is first partitioned into two by calling *r-Balanced Min-Cut* procedure, where *r* is the estimated *cut_ratio* between two partitions. Processing elements are allocated in accordance with the execution and communication cost of that stage. A recursive partition is then performed until the code size can be fit in one instruction memory. Finally, a local refinement is performed in order to migrate

tasks from bottleneck stage to non-bottleneck stage. In summary, the time complexity of this heuristic is reduced to O(x3). In [30] Ramamurthi et al. also adopted a divide-and-conquer approach but particularly addressing memory layout and data mapping problem. The sub-tasks of an application are first mapped to *Processing Engines* (PE) following procedures similar to *Integer Linear Programming* (ILP) formulations. Then optimization is performed based on the calculated priority of a data item in a task process. Three situations in regard to the number of threads supported, data spill and idle time are tackled with specific strategies, aiming at maximizing throughput. Another heuristic presented by Weng [31] is called randomized mapping, which basically means choosing a valid mapping by random and compare its performance to previous results. It is testified to be possible to find near-optimal solutions for mapping parallel tasks, but naturally the algorithm takes up too much search space and thus low-efficient. The similarity shared by all the above approaches is the inclusion of real traffic-based profiling phase when building the IR (e.g. task graph) of a program. Either analysis of dynamic instruction traces [30, 31] or combination of static and dynamic analysis [29] is used for profiling programs.

2.3.3 Re-targetable Compilers

Re-targetable compiler contains an adaptive back-end that could be easily modified to interface with heterogeneous processors. Such a re-targetable compiler framework could be specifically tuned to construct specific NP compilers. The Shangri-la compiler [24, 25] is a good example in point. Below in this section four other models are reviewed and compared. General information about the compilers/platform solutions is given in Table 2.2 below.

Table 2.2. Comparison of the re-targetable compilers

	DSL Level	**ASIP target**	**Partitioning**	**Scheduling**
Shangri-la	Baker	IXP series	Automatic	Compiler-Assisted
PacLang	PacLang	IXP2xxx	Manual	Manual
NP-Click	Click & extension	IXP series	Automatic	Automatic
NEPAL	C & modular template	Cisco & IXP	Manual	Manual
TejaNP	C, C++ & extension	Many NPs	Manual	Manual

PacLang [32] made effort to abstract away the architecture complexities in source code, where application description could be given in a customized, linear, strong-typed language. By denoting linear type packets in this language, the syntax could be statically checked to ensure that no packet is referenced by more than one thread. If such a kind of unique ownership property is enforced, no additional locking mechanism is needed anymore. Finally all the linear typed blocks are connected to each other through queues and freely mapped to low-level architectures. Compiling the PacLang code has been demonstrated on IXP 2400 with an IPv4 packet forwarding case study, and the program is more lucid in comparison with traditional C language.

Plishker and Shah managed to evolve a programming model "NP-Click" [33]. Unlike PacLang or Baker, the programming model is based on Click modular router [34] rather than a brand-new grammar and syntax. Click software router has been proved competent in industrial practice for efficiently describing network applications in a modular way. NP-Click fortifies its capability in data-memory mapping, elements-threads mapping and shared resources management. The NP-Click itself is implemented in Intel Microengine C [35], thus the framework is essentially an extension of Intel C compiler. But the framework of the modular elements facilitates the extraction of parallelism; and the elements are automatically transformable to task graph. Those innovations could be applied to various NP compiler platforms.

Also featuring modular programming approach, NEPAL [36] was proposed as another runtime system for extracting modularization from sequential codes and mapping the modules into a variety of Network Processing Units (NPUs). The NEPAL converter and optimizer analyse C programs, C++ binaries etc. to generate modular codes that could be executed in parallel. The dynamic module manager, essentially a runtime environment, then is responsible for controlling the overall execution under different underlying architectures and maximizing parallelism. This framework has been validated using an ARM simulator where two different systems are simulated.

TejaNP is yet another software platform focusing on portability, performance, and ease of use [37]. The model is based on the C language with a minor C extension

and empowers users' capabilities in expressing both software and hardware architectures and managing application mapping. The modular elements are stitched together then to completely describe the functionality of the application.

2.4 Energy-efficient Compiler Techniques

The compiler of a network processor plays a vital role in ensuring machine code efficiency. A set of energy-efficient NP compilation techniques is reviewed in this section.

2.4.1 Dynamic Energy Reduction

Energy-oriented optimization could start from front-end, even at the highest source-level to translate tool-generated code or newest algorithm implementation written for better readability into energy efficient counterparts. In [38] Yang et al. experimented front-end loop transformation, such as loop permutation, loop fusion and tiling, and evaluated to find their significant contribution to energy reduction in *SimpleScalar*, a cycle-accurate architectural simulator [39].

Inside the back-end part, specific energy-efficient optimizations could be performed at nearly all stages. During the *code selection* stage, one way is to attach higher priority to low energy consumption instructions. An example is shown in [40] where an algorithm is tailored for laying out local variables based on *access patterns* to take advantage of auto-increment / auto-decrement addressing modes available on a commercial NP. Instruction optimizations for efficient register files access have been exploited as well in [41], especially in the code generation that are capable of optimizing address instructions. Besides, loop transformations like unrolling are still tempting at this stage by maximizing instruction level parallelism that result in energy reduction [38].

During the instruction scheduling phase, special algorithms could be implemented aiming at reducing the energy contribution due to the change of operations on the datapath. In [42] Bona et al. proposed a spatial scheduling algorithm for embedded *Very Long Instruction Word* (VLIW) processors, based on a low-power reordering of the parallel operations within the same long instruction by considering each basic block of the generated code and rescheduling operations within the same bundle(spatially) to minimise the cost function:

$$\sum_{k} V(w_n^k | w_{n-1}^k),\ \forall (w_n^k | w_{n-1}^k) \in \textit{Basic Block} \tag{2.1}$$

where $V(w_n^k | w_{n-1}^k)$ is the additional energy contribution due to the change of operation $(w_{n-1}^k \longrightarrow w_n^k)$ on the same lane k. Analogous approach is described and validated by Yun and Kim [43] to reduce the step power and peak power consumption from performance-critical loop bodies using a power-aware modulo scheduling algorithm. In this sense, the step power is defined as the difference in the average power between consecutive clock cycles while the peak power means the maximum power dissipation during the execution of a whole program. Another relevant scheduling technique is Lee's greedy bipartite-matching scheme for horizontal scheduling and a heuristic method for vertical scheduling for VLIW architectures [44], especially for achieving the optimal switching activities of the instruction bus.

2.4.2 Leakage Power Control

Unlike dynamic power consumption, leakage consumption derives from leakage current as long as the circuit is on regardless of the switching activities. Thus the methods for dynamic power control cannot be applied. A common approach nowadays to reduce leakage power is shutting down inactive hardware units, though the turn-on and turn-off certainly demand additional hardware-based built-in support. Compiler optimization is a key to improve the leakage power control benefits, because compiler-based techniques are in charge of data and task mapping as well as system resource scheduling. If code and data executing on hardware are optimally scheduled to concentrate the workload on a limited number of PE, more space can be earned to perform hardware turn-on/off. During code analysis phase of compilation, an optimized compiler could find the code region where one or some PEs can be possibly shut down at some stage.

An example following this strategy is illustrated in [45]. Zhang et al. proposed a technique that first detects the idling functional units based upon a data-flow analysis along the paths in control flow graph. Then functional unit activation/deactivation instructions are inserted with regard to estimated basic block execution cycles and leakage control strategies. In their approach, two leakage control strategies were taken into account, namely input vector control

and supply gating. Jia Yu also considered and validated the use of power gating strategy to reduce leakage power in NP [29]. When zero overhead is assumed to turn on and off a PE, it is quantified that power gating can save much more power than clock gating (i.e. reducing dynamic energy solely) for four representative network applications.

2.5 Challenges in Compiler Design and Implementation

Though many implementations of NP compilers are available at present as discussed above, several notable challenging issues are still to be addressed.

- *Easy programming interface for programmers, together with packets-oriented code generators for PE*

It is common to encompass specific bit-wise operations in instruction set design of PE for most NP units nowadays, like in Intel IXP NP [28]. An efficient compiler should optimize its code generation by taking advantage of those domain specific instructions like bit-packet addressing, and those arithmetic operations such as comparing and modifying fields in the packet header. Needless to say, the compiler is even better off to support new high-level languages that are easy to learn and use, preferably tuned for network applications i.e. tailored for processing packet data.

- *Intelligent program mapping in managing parallel processing resources*

As the NP system tends to employ more PEs on chip to provide parallel processing power to keep pace with the increasing network traffic and computation complexity, compiler techniques need to be consistently evolved to go with the trend. At task level, the strategy of merging and replicating applications and mapping them onto mixed pipeline and parallel hybrid architecture (i.e. Fig. 2.5) still demands investigation and exploration. Furthermore, the efficient utilization and arbitration of heterogeneous resources of a NP system, such as co-processors and special hardware accelerators, also needs to be further studied. The compiler should be more sensitive or intelligent to manage all the processing units. Intelligence and flexibility support are also inadequate at high level currently, e.g. it is not an easy job to add/remove applications in a multiple application system. For now, usually a complete

re-compilation and scheduling have to be done every time the configuration changes, which is neither time-efficient nor energy-efficient.

- *Management of packet data in NP memory hierarchy*

Just like compilers for general processors, NP compiler is responsible for program data placement in the NP memory hierarchy. But in the absence of a run-time system or OS support, NP compiler is even more influential in deciding the overall memory performance. Packet data usually exhibit low inter-dependence. And for the sake of achieving higher throughput and lower latency, NP seldom includes hardware cache mechanism as opposed to general processors. Memory management in a packet data-aware approach is certainly a challenge anyhow, especially when the NPs need to support various network application functions as of now.

- *Energy efficiency in compilation*

Intense power consumption is observed nowadays in NP since more processing cores are assembled on a chip and the complexity of packet processing tasks is increased in advanced network applications (e.g. Firewall, instruction detection/prevention). Innovation in energy-aware compilation techniques is deemed promising. Trade-off should be properly addressed among conflicting metrics, e.g. throughput, latency and power-consumption, as it is observed that optimization solely for one metric is often realized at the expense of others [29].

2.6 Conclusions

Network Processing System family is an example of heterogeneous on-chip system with different processor instruction-set architectures, memory hierarchies, cooperative hardware accelerators and interconnections etc. Network processing application itself exhibits distinct characteristics in relation to packet processing as well. The job of programming NPs is thus not wholly complying with traditional compilation scenarios, asking for innovative approaches tackling NP-specific difficulties. In this chapter, specific techniques for tackling with these problems were investigated, from aspects of packet-oriented processing support and parallel architecture support respectively [46]. Several challenging issues in NP compilation are also pointed out. The compiler could be tuned more efficient

by applying specific register allocation algorithm and special code selection techniques. Given such a heated issue in embedded system on power efficiency currently, novel compilation techniques on energy saving were also reviewed in the text. It was observed that though traditional performance enhancement methods might not always be beneficial for energy-oriented optimizations, classical heuristics and methodologies can be referred to and extended at all stages during compilation.

As energy-efficient compilation and compiler techniques for packet processing further develop, the following trends are worth attention:

- A more flexible compilation model in front-end. It would provide a programming interface tuned for packet processing and bit-level operations, so easing programmers' job. The interface could be modular-based for conveniently adding/removing applications.
- A more flexible compilation model in machine-specific back-end. An intelligent compiler could choose best code generation strategies among different candidate solutions, and/or re-adjust program mapping reflecting the changing requirement of performance, code size, power consumption and traffic volume etc.
- The idea of incremental compilation would be incorporated. When new applications are added into network processing system, the compiler analyses all the tasks and incrementally re-schedules the hardware resources and re-maps the program and data. The results should be up to specified metrics.
- Energy efficient compilation for network processing system will receive more attention. Traditional optimization hotspots are still where energy-aware techniques can play a part, while constraints are with respect to power dissipation instead of performance only.
- The interaction and trade-off between energy-efficient optimizations and those grounded on classical metrics, such as high throughput and low latency, awaits further exploration. A balanced point needs to be found.
- Energy-aware optimizations for NP compiler will be more interesting at task-level (besides instruction- or basic-block level reviewed in the thesis),

where energy classification of code/data blocks could be examined.

Chapter 3 - Analysis of Network Applications

3.1 Introduction

In last chapter, it is pointed out that task level allocation of code and data blocks is worth investigating, especially for parallel processing optimizations in a multi-core network processing system. In order to perform any further optimization, it is a prerequisite to obtain a comprehensive profile of the application to be deployed. This chapter presents the work on developing a compiler tool to characterize the application dependences. The dependences information is vital in resolving application partitioning and mapping problem which is a sub-domain of task scheduling. The focus of this chapter is on the network application analysis in the eyes of a compiler.

In the internal work flow of compilers, an *Intermediate Representation* (IR) is a data structure used to collect the input information, e.g. the semantics of C code. Most of the compiler optimizations would conduct upon a specific kind of IR. Classical examples of IRs include the *Control Flow Graph* (CFG) built for flow analysis, *Abstract Syntax Tree* (AST) employed in syntax-directed translation etc. As said earlier, in the context of task allocation problem for network processor systems [47], the compiler needs to characterize the dependence profile of an application. Previous researches have employed *Annotated Directed Acyclic Graph* (ADAG) [48], basic-block based task graphs [49], general analytical model [50], etc. to abstract the applications. However, these representations are generally generated from runtime traces of the network applications. From the compiler's perspective, they are not directly applicable. Rather, efficient representations of static profiling results are required during compilation. In this regard, *Program Dependence Graph* (PDG) was developed by Ferrante et al. [51] around 1990s;

and in [52], it was used as the IR to statically characterize the applications and was fed into the task partitioning algorithm. The PDG explicitly expresses the dependences of a given program in a graph, and implicitly indicates the opportunities for code parallelization. Given its prominent features as an IR, it can be used extensively in compiler optimizations for parallel systems like network processors systems. A compiler pass was implemented to efficiently generate the PDG in Machine SUIF [53] compiler infrastructure. The sections below summarize the work of the PDG pass implementation and demonstrate its use in network applications analysis.

3.2 Dependence Graph

In essence, PDG is a form of *Directed Acyclic Graph* (DAG). It consists of two sub-graphs, namely *Control Dependence Graph* (CDG) and *Data Dependence Graph* (DDG). CDG expresses the *Control Dependence* while DDG depicts the *Data Dependence*. Its application extends to not only the program dependence analysis but also subsequent optimizations on top of it. In this section, the terminologies of the graph and definitions used during the dependence analysis are introduced before any design and implementation issues are elaborated.

By definition, program dependence comprises CDG and DDG. One can always regard PDG as an integration of those two sub-graphs, each being a self-contained component on its own. CDG summarizes control dependence information while DDG holds data-dependence links. So in the following text, the relevant background of each sub-graph will be dealt with separately. Note as well that, PDG can be applied to different levels of the code, i.e. nodes in a PDG may be *basic blocks* (BB), statements or individual expressions (operators). This report concentrates on the BB-level since it exposes more program-level features that could be utilized in parallelization optimizations, which is much of current interest.

3.2.1 Control Dependence Graph

A *Control Dependence (CD)* is a constraint that is relevant to the control flow of the program. For example in the three-address code block showed in Fig. 3.1, statement S2 and S3 will be executed only when S1 is evaluated to be false. And in this case, statements S2 and S3 are control-dependent on statement S1. It is

analogous in the BB-level CFG. Conceptually, in the BB-level CFG, node *Y* is control-dependent on *X* if *X* has two paths to *exit*, one through node *Y* while the other does not. Fig. 3.2 depicts such a relationship.

```
if a>b goto L1      (S1)
    d = a + b       (S2)
    e = a * b       (S3)
L1:
    d = a * b       (S4)
```

Fig. 3.1. Three-Address code block

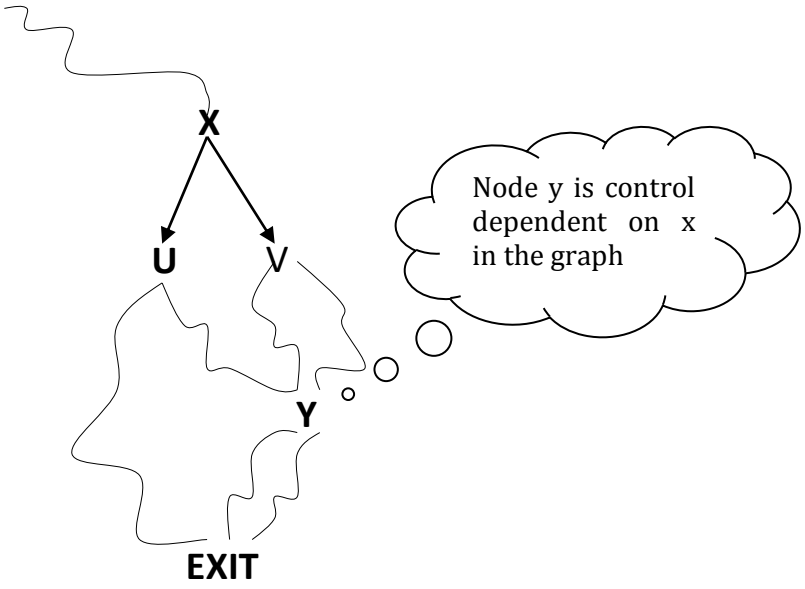

Fig. 3.2. Control dependency relations

The formal definition of Control Dependence is literally given in the following procedures. The definition of the *Post Dominance* in a CFG is given below first.

Definition 3.1: in a Control Flow Graph *G*, node *V* is *post-dominated* by node *W*, if every directed path from node *V* to node *EXIT* contains *W*.

With the post dominance given, now *control dependence* can be formally defined as follows.

Definition 3.2: in a Control Flow Graph *G*, node *Y* is control-dependent on node *X* when

(1) A directed path P from *X* to *Y* exists with any nodes *Z* in P post-dominated by Y (P does not include *X* and *Y*);

(2) *Y* does not post-dominate *X*.

In other words, a node *Y* is control-dependent on *X* if from *X* there is a branch to *U* and *V*; from *U* there is a path to exit that avoids *Y*, and from *V* every path to *EXIT* (including *V*) hits *Y* (e.g. Figure 3.2).

The formal definition may seem a little obscure. It is helpful to keep in mind the semantic meaning of Control Dependence though. *Y* being dependent on *X* essentially means that the execution of node *Y* depends on the result of the conditional statement in node *X*.

Interestingly, the control-dependence relationship is strongly connected with the concept of *Dominance Frontier* (DF). It is not accidental though, as both theories are proposed by the same group of researchers when they are working on application dependence analysis [51][54]. Conceptually, DF of one particular node is the border between the dominated and un-dominated nodes. It is commonly used in construction of Static Single Assignment (SSA) [54], another type of compiler IR. The "interesting" link between control dependence and dominance frontier is that *Y* is control-dependent on *X* if and only if *Y* is in *X*'s *Reverse DF* (N.B. Reverse DF is the nodes' DF in the Reverse Control Flow Graph). The theoretical explanation of the relationship is out of the scope of this report, so only the conclusion is given here.

At this stage all the control dependence of one program can be captured in a graph representation, i.e. the CDG. The following is the formal definition of CDG.

Definition 3.3: CDG has an edge from *X* to *Y* whenever *Y* is control-dependent on *X*.

CDG is built on top of the program CFG, and particularly at BB-level in the context of this report. So obviously, the nodes in CDG are the same of the nodes in *CFG*, i.e. Basic Blocks of the program. Additionally, CDG also contains special nodes to summarize control conditions, as detailed below.

Four kinds of nodes make up the CDG, namely *start* (i.e. *root*), *region*, *predicate* and *statement* nodes.

Start represents the entry point of the program. It is a dummy node facilitating the creation of CDG, usually empty. In the CFG, it is also the starting node but with

two edges. The true edge goes to the "real" entry point of the program (BB containing instructions) while false edge linking the exit point of the program;

Predicate nodes usually end with control transfer instructions. They represent the True or False conditions to be selected among different control dependence edges;

Statement nodes are those ones comprising pure computations, excluding any control information within;

Region nodes summarize the set of control conditions for subsequent nodes.

Observation here (and could be verified in theory) is that the *statement* nodes have only one exact parent node and no children in the CDG (i.e. nobody is control dependent on them), while *predicate* nodes have one parent and two children linked by edges marked "T (True)" and "F (False)" respectively. As for the *region* nodes, they could have multiple children and multiple parents in the CDG. *Start* node has no parent but could have multiple children. Finally by definition, the targets of edges originating from *predicate* nodes must be region nodes.

3.2.2 Data Dependence Graph

The definition of *Data Dependence (DD)* is not that straightforward as for CD, and it is varied in different contexts and applications. Hence, the *Data Dependence Graph (DDG)* would not be formally defined in a way that any kind of DD edges is incorporated as part of the DDG. Usually, different forms of dependency graph comprise assorted combinations of *load* and *store* dependencies, *def/use* dependencies, *loop* dependencies etc.

In a larger sense, typical data dependence information includes *def-use* (*flow-dependence* or *true dependence*), *anti-dependence*, and *output-dependence*, as depicted in Fig. 3.3. Only the DD that is meaningful to the work will be elaborated. With regard to the problem of analyzing the communication cost between tasks, the *def-use* data dependence is of particular interest.

X := := X True-Dependence	:= X X := Anti-Dependence	X := X := Output-Dependence

Fig. 3.3. Classification of Data Dependence

Def-use DD (i.e. the *true dependence*) denotes the data flow from node M to N by assignment at M and use at N. The way to capture the DD in a graph representation is to make use of the SSA form.

SSA is an intermediate representation introduced by Cytron [54]. Three forms of SSA are described in literature, namely *Minimal SSA*, *Pruned SSA* and *Semi-Pruned SSA* [55].

The minimal approach is named in terms of its time consumption, as it requires less time to compute. But it may insert lots of dead *phi*-nodes, i.e., one that define names that aren't used later;

The *pruned* approach corrects that flaw by doing liveness analysis to avoid inserting dead *phi*-nodes. It therefore saves space, while at the expense of doing data-flow analysis that might not otherwise be needed;

The *semi-pruned* approach is based on the observation that many compiler-generated temporaries are never live across a control-flow edge, i.e. local to a single basic block. Thus, they never require *phi*-nodes. *Semi-pruned* SSA form uses this observation to eliminate many of the *phi*-nodes that exist in minimal form and avoid performing the liveness analysis needed for *pruned* SSA form.

The SSA form greatly simplifies the *def-use* analysis for symbols at BB-level since each variable has only one definition. The space and time to compute the *def-use* chain in normal CFG would be a quadratic blow-up. But in SSA, the *def-use* is essentially a link rather than a chain. For nearly all realistic programs, the size of SSA is just in linear relation to the original CFG.

3.2.3 Program Dependence Graph

PDG is a graph IR that is strongly related to the concept of CFG, the classical graph IR. As the same in CFG, the instructions are grouped together at the *Basic-Block* (BB) level. In a BB, the first instruction is the only entry point in the control flow, while the last instruction is the only exit point. Thus CFG represents the control flow with its nodes being BBs and its edges being the path of the control flow.

The very basic kind of CDG is also composed of BBs. However, its edges now represent the Control Dependences. As explained earlier in the section about CDG, *control dependence* (CD) is an abstraction of the execution order. For example, node *x* in the CFG (i.e. BB *x*) ends with a branch instruction and hence has two paths at the exit point of the node. If node *y* (i.e. BB *y*) will be executed only when the control flow goes through the *true* path at the exit of node *x*, it is said that the node *y* is Control Dependent on node *x* on the *true* edge. Correspondingly in the CDG, a directed edge is added from node *x* to *y*, labeled with a control condition, e.g. *true* in this example. After this initial generation of CDG, the *region nodes* are inserted in the second phase to represent a set of control dependences. For instance, if node *y* is control-dependent on node *a* on the *true* edge and on node *b* on the *false* edge, a region node *R1* is created to hold the control dependences of <*a*T, *b*F>. The node *y* is made to be control dependent on the newly created region node *R1* only.

As for the DDG, its nodes are still BBs. The edges now represent the data dependences. If an instruction in basic block *y* uses a variable that is defined in basic block *x* (i.e. a *def-use* chain exists between different basic blocks), it is defined that there is a data dependence edge between *x* and *y*. It is also assumed that the weight of the edge is same with the number of such *def-use* chains across two basic blocks. In the DDG, the edges are labeled with the *weight*.

The PDG can then be easily constructed by combining the CDG and DDG together. Since the nodes of both graphs are largely the same (i.e. BBs), with a few additional region nodes in CDG, the merge process is straightforward.

3.3 Design of PDG Generators

As explained in introduction, the PDG has two sub-graphs CDG and DDG. In this section, the algorithms for constructing CDG and DDG, the specification of the PDG class and the explanation of its fields and methods will be given.

3.3.1 Graph Construction Algorithms

Firstly the algorithm for the construction of the CDG is introduced. There are two options here, namely,

Method 3.1*:* Follow Ferrante's canonical algorithm described in [51].

Method 3.2: Calculate the reverse dominance frontier, and CDG has edge x -> y (i.e. y is control-dependent on x) whenever $x \in DF_{G'}[y]$ (i.e. x lies in the reverse dominance frontier of y).

Table 3.1. CDG construction algorithm 1

INPUT	**Control Flow Graph**
STEP 1	Augment the CFG with an empty (dummy) starting node.
STEP 2	Compute the post-dominators of each node and construct the Post-Dominator Tree (PDT) of the CFG.
STEP 3	Define an edge set S. Each edge E=(A,B) in the CFG and B is not an ancestor of A in the PDT is added to the set S.
STEP 4	For each edge in the set S, find least common ancestor L of A and B in the PDT. L should be either A or A's parent in PDT.
STEP 5	All the nodes on the path from L to B (except L if L is A's parent) are control-dependent on A.
STEP 6	Add the region nodes.
OUTPUT	**Control Dependence Graph**

Table 3.2. CDG Construction Algorithm 2

INPUT	**Control Flow Graph (*G*)**
STEP 1	Add a new dummy predicate entry-node start to *G*, with its "T" edge running to the original entry and its "F" edge to exit.
STEP 2	Let *G'* be the reverse control-flow graph.
STEP 3	Construct the dominator tree of *G'*.
STEP 4	Calculate the dominance frontier $DF_{G'}$ of the nodes of G'.
STEP 5	The CDG has edge x->y whenever $x \in DF_{G'}[y]$
OUTPUT	**Control Dependence Graph**

Note that, these two methods are theoretically identical as it was explained in last section. Under Machine SUIF infrastructure, method 2 tends to be more straightforward because it has provided a *Control Flow Analysis* (CFA) library

with reverse dominance frontier calculated off the shelf. Both methods are included in my generator though, to best illustrate the complete algorithms.

The elaboration of the first method (building from scratch) is given in the Table 3.1. A sequence of data structures are generated throughout the process. First of all the *Post-Dominator Tree* (PDT) is derived from the CFG by computing post-dominators. Then a candidate edge set is identified in step 4. The control-dependence is calculated by finding least common ancestors in PDT for the nodes in candidate edge set. The definitions of the data structures used in this algorithm have been given in section 3.2.1.

The procedure of *Method 3.2* is outlined in the Table 3.2. This method finds the control dependence based on the Reverse DF. The data structures used in this algorithm sequentially includes reverse control flow graph, dominator tree, and dominance frontier tree. The two methods can converge to generate the same CDG because if a node *y* is control-dependent on node *x*, then *y* is in *x*'s reverse DF, vice versa.

```
  nleft = len;
  w = addr;
  sum = csum;
  while (nleft > 1)
    {
      sum += *w++;
      nleft -= 2;
    }
  if (nleft == 1)
    sum += ((*w<<8) & 0xff << 8 | (*w<<8) & 0xff00 >> 8);

  /* add hi 16 to low 16 */
  sum = (sum >> 16) + (sum & 0xffff);
  sum += (sum >> 16);               /* add carry */
  answer = ~sum;                    /* truncate to 16 bits */
  return (answer);
}
```

Fig. 3.4. IPv4 forwarding application code snippet

To exemplify the CDG construction, the corresponding CFG, PDT and CDG for an IPv4 code snippet are showed below. The C code provided in Fig. 3.4 is extracted from a radix-based IP forwarding application.

Each basic block is numbered in sequence by its appearance in the CFG (i.e. Fig. 3.5). Node *exit* post dominates all nodes since any node has to pass through it in execution. So *exit* always acts as the root node in PDT. PDT exhibits the hierarchical post-dominance relations. For instance, BB6 immediately post dominate BB5 and BB4. BB4 post dominates BB2 which post dominates BB3 and BB1 in turn. PDT showed in Fig. 3.6 clearly illustrates such relational hierarchy. Further, edge 4->5 would be added into the candidate edge set as BB5 is not an ancestor of BB4 in the PDT. Next BB6 is found to be the least common ancestor of BB5 and BB4 in the PDT. Finally it is known that BB5 is control dependent on BB4 by the step 5 in Table 3.1, and hence the edge 4->5 in Fig. 3.7 labelled T for true. Other edges in the CDG would be identified through identical procedures listed above.

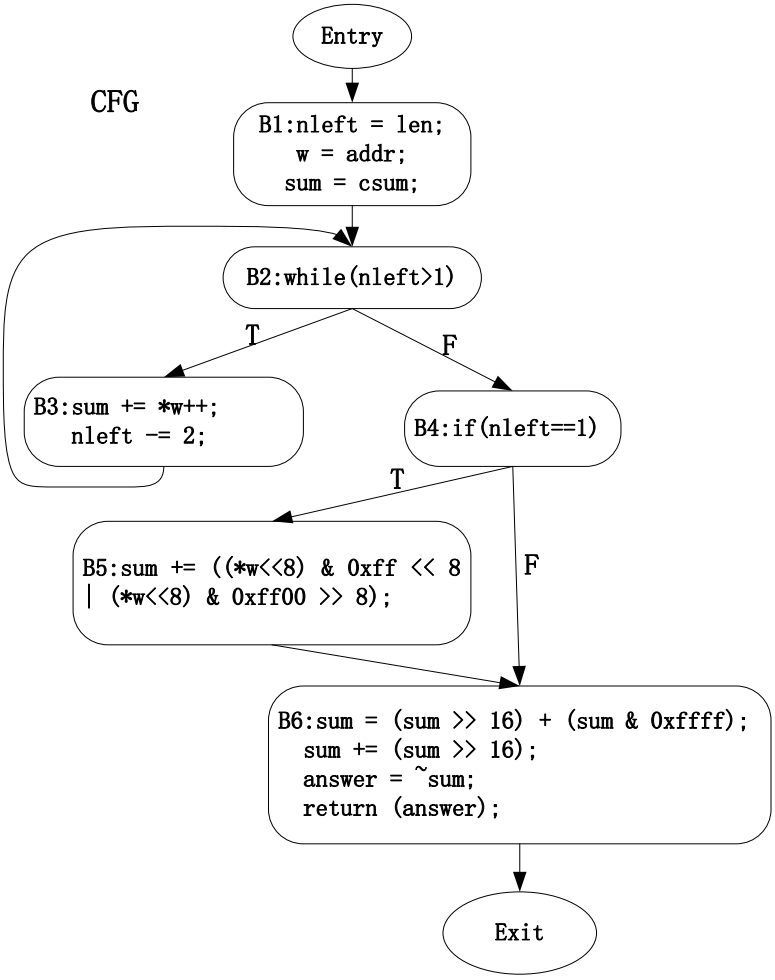

Fig. 3.5. CFG of the IPv4 code snippet

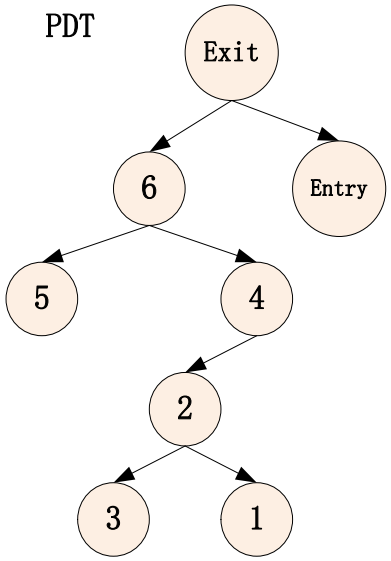

Fig. 3.6. PDT of the IPv4 code snippet

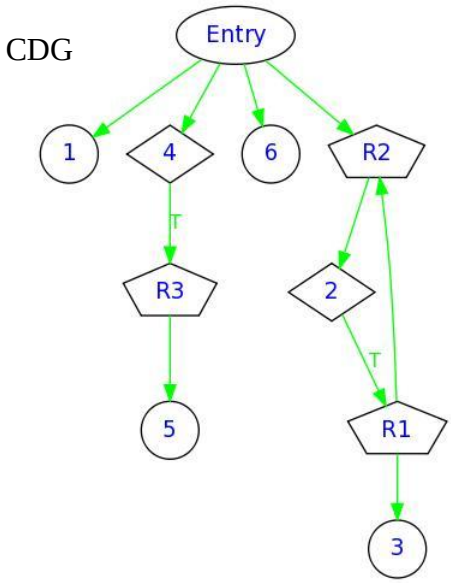

Fig. 3.7. CDG of the IPv4 code snippet

After the construction of CDG, the method for calculating the data dependence and the construction of the DDG is briefly introduced. As Machine SUIF provides a SSA library and helping routines to transform the compiler IR to and back from SSA, the library would be effectively used to gather the data dependence.

To construct the DDG, the CFG is transformed first to SSA in pruned-form. Pruned-form SSA is chosen since it compresses the number of *phi*-nodes and the number of *def-use* chain as well. The reported data dependence across basic blocks in pruned-form would be more accurate than those in the other two SSA

forms, i.e. no “artificial” dependence brought by value re-numbering and phi-nodes insertion.

In the SSA form, all the *def-use* chain across basic blocks can be found then. The total amount of inter-BB *def-use* chain represents the data dependence *weight*.

Finally the SSA form should be converted back to the original CFG form.

3.3.2 Classes Design of PDG Pass

In this section, the design of a PDG pass under Machine SUIF infrastructure is explained. As Machine SUIF recommends using its *Optimization Programming Interface* (OPI) model programming, the PDG class is wrapped in a general SUIF pass employing the OPI to maximize the substrate-independence.

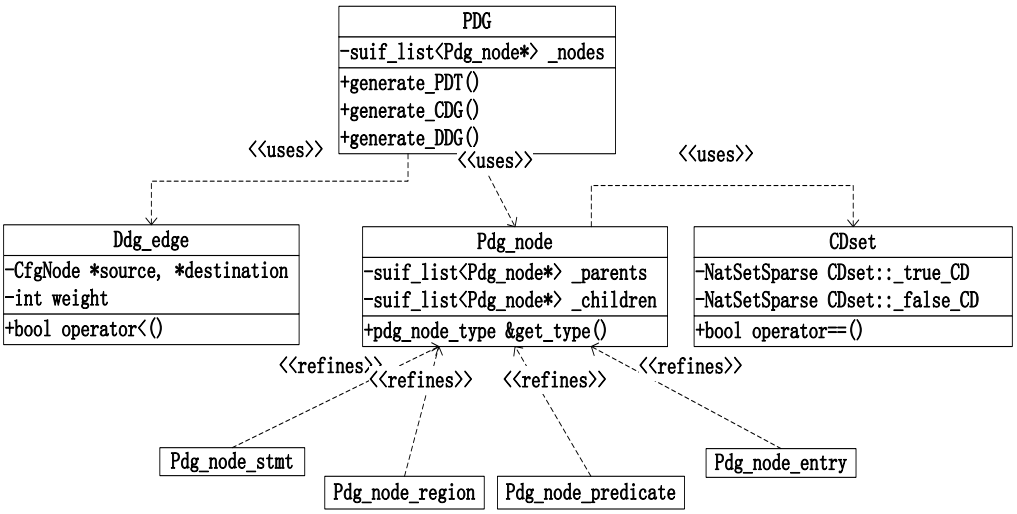

Fig. 3.8. Class design of PDG pass

Consistent with other built-in SUIF passes, the PDG pass is designed in an object-oriented pattern. The design model of the classes is given by Fig. 3.8. By construction, the PDG nodes could be classified further into types of *entry* node, *statement* node, *predicate* node and *region* node. Each of them is modelled in an inherited sub-class of the parent class, i.e. *Pdg_node*. The dashed lines labelled with “refines” in the figure represent this relationship. The *CDset* class models an arbitrary set of control dependences and is used by the *Pdg_node* class to represent the control dependences of a given node. Finally the *Ddg_edge* class gives the data dependence information between PDG nodes. It has a *weight* property to indicate the number of *def-use* chains, as explained earlier. There is no need to have a class to model the CDG edges since the control dependences are

implicitly included in the *Pdg_node* class, specifically by the properties of *_parents* and *_children* in the class. The anatomy of each class design is included in the appendix A.

3.4 Implementation of PDG Pass

The algorithm of PDG generation is adopted from [51] and was implemented in Machine SUIF compiler infrastructure.

The class methods of PDG (see Figure 3.8), namely *generate_CDG* and *generate_DDG*, output the graph results in both a pure text and a graph description formats (i.e. *.dot* files). The description formats files can be fed into *Graphviz* [56] to generate the actual image files, e.g. in JPEG or GIF format.

The detailed description of the issues in implementation is included in appendix B.

3.4.1 Lessons Learned

SUIF defines an *Optimization Programming Interface* for developers to add their own passes. Abiding by these OPI, it is possible to separate the algorithm details from the substrate IR (i.e. SUIF IR); thus the portability of code and productivity of coding are both enhanced. SUIF is also packaged with several built-in libraries facilitating control-flow and data-flow analysis. Making use of these library functions greatly reduced the workload of implementations. For example in data-dependence analysis, the Single Static Form library was used to directly give the *def-use* chains and the only work left is to assemble that information in the PDG form.

3.5 Results

The PDG generator pass was run on a set of network application benchmarks to testify the validity of the pass and to collect the program dependence information.

A code segment for checking the packets' integrity, namely the *check_sum* function is analysed first. It is one of the most common operations in packet processing systems. The procedure of *check_sum* is to calculate the 1's complement sum over the packet header octets. It returns true if the results are all 1 bits. The CFG of the function and its corresponding CDG and PDG output by

the generator are given in Fig. 3.9. In the PDG, the edges in solid lines are CDG edges while those dashed lines are DDG edges. The round vertices represent the *statement* nodes and diamonds stand for *predicate* nodes. These two types of nodes are also the BBs derived from CFG nodes containing instructions. The pentagonal vertices are *region* nodes that summarize a set of control dependences as explained earlier. By the nature of CDG, the set of nodes that are control-dependent on the same node, such as node 1 and node 6 in the Fig. 3.9, could be executed in parallel, as long as they do not entail any data dependences.

A set of tests consisting of several sample code snippets were also conducted. The results were validated by comparing the generated PDG against those reported in [51] [52] and some in compiler textbooks. These tests are not necessarily all relevant to the network applications, but the comparison results ensured the validity of the developed PDG pass in general.

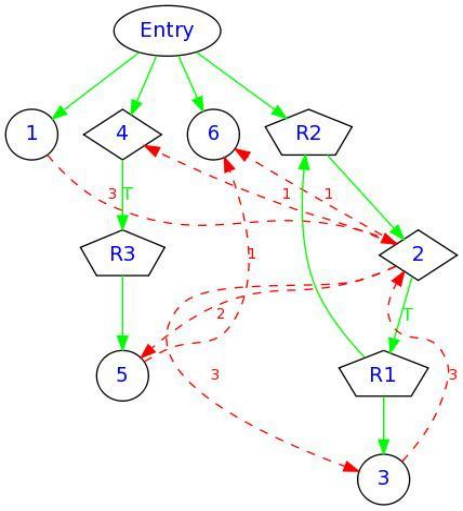

Fig. 3.9. An example of PDG

3.5.1 Example Application IPv4 Forwarding

As a concrete example the pass is run for a trie-based IPv4-packet forwarding application. The IPv4 forwarding code was adopted from Packetbench [57]. In order to generate the PDG of the whole IPv4-packet forwarding application, all the functions are inined. It is common to do so for network applications, since the applications themselves are usually small in C code size.

Firstly the PDG of the code snippet presented earlier in Fig. 3.4 is illustrated below in Fig. 3.9. This is the standard output interfacing with backend Graphviz.

Round vertex indicates *statement* nodes, then diamond for *predicate* nodes, and pentagon for *region* nodes. Edges in solid lines represent the control dependence, while dashed ones are for data dependence.

The major procedures of IPv4 forwarding include building a route table during system initialization; checking the packet type (dropping non-IP packet); validating the integrity of the packet; checking *Time To Live* (TTL) field and decrementing it; updating the checksum; and finally looking up the destination address in the route table to determine the next-hop port. In the experiment, after inlining all the major functions, the C code is lowered down to SUIF IR and then transformed to CFG IR. And then the PDG generator pass takes the CFG IR as the input and generate the PDG of the whole application as the output. Fig. 3.10 captures the steps through the whole process in Machine SUIF.

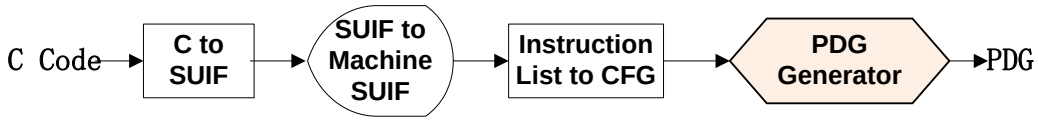

Fig. 3.10. Steps for running PDG pass

Fig. 3.11 illustrates the generated PDG of the whole packet forwarding application. Note that the nodes in round and diamond vertices are basic blocks in CFG. Their numbers are consecutive and consistent with their CFG numbers. The graph exposes clear hierarchy of control dependences. For example, predicate nodes 4, 6, 9, 12, 15, 18, 21, 24 and their respective children nodes are all control-dependent on entry node, and have no remaining entangling control dependence edges among each other. It means the paths (e.g. from node 9 down to node 11 in the figure) could be well grouped together and run independently on one processor. The communication cost, though, is given by the data dependences edges (i.e. the dashed lines) that connecting any nodes on the path.

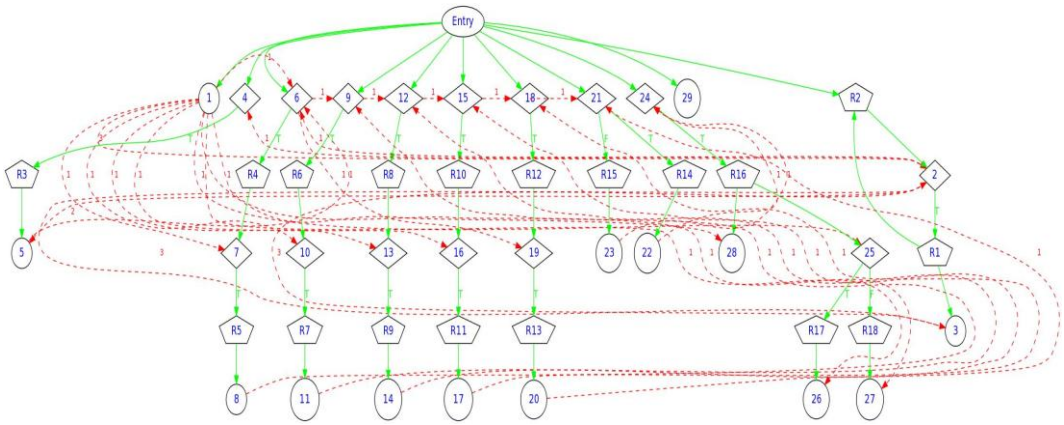

Fig. 3.11. PDG of IPv4 packet forwarding

3.6 Practical Use of Dependence Graph

Previous researchers have employed PDG in various ways in static program analysis. In [58], Gong et al. also constructed PDG in SUIF compiler to facilitate the logic synthesis. Due to their special application domain, their PDG data structure was different from the one presented here, with the SSA form incorporated. Rather in my approach, SSA is directly used to collect data dependences. The Linda Compiler, a precursor in developing language support for parallel systems, also explored SUIF to generate PDG for its internal work flow [59]. Their approach is close to mine except that their intended use of PDG was for message communication in distributed-memory systems. Moreover, the Linda Compiler was based on the old SUIF1 that is superseded by the newer SUIF2 employed in this work. The two compiler frameworks are not compatible and according to SUIF group's documentation [60], SUIF1 is less flexible in modular design and code-reuse etc. The contribution here should be more applicable for today's use.

In the next chapter, the work on network application partitioning and mapping for the network processors systems makes extensive use of the PDG generated by this SUIF pass. In [52] an algorithm adopted from Min-Cut Max-Flow problem was implemented to take the PDG as the input graph and regard the weight of the edges as the flow capacities in the Max-Flow problem. It aimed to minimize the communication cost (including both control dependence and data dependence) among the partitions and balance the resource utilization of the network processors. Indeed, other heuristics solving the partitioning and mapping

problems for the network applications should be extensively investigated, and other performance metrics may be taken into consideration.

Besides, the PDG could be used in other compiler optimizations such as efficient data mapping in presence of cache system, branch speculation and loop optimizations. Experiment will be carried out to verify the validity of the optimizations in network processors systems.

3.7 Conclusions

In order to perform certain analysis and optimizations in compilers, an efficient representation that explicitly captures the control-flow and data-flow dependence information of the source code is needed. *Program Dependence Graph* is an example of such representation. The design and implementation of a compiler pass in Machine SUIF infrastructure that generates the *Program Dependence Graph* IR is described [61][62]. Taking advantage of the Optimization Programming Interface programming in SUIF, most part of the pass is largely independent of the concrete compiler substrate and thus of highly portability. The PDG generated is made up of two sub-graphs, *Control Dependence Graph* and *Data Dependence Graph*, each summarizing the dependence information regarding control and data respectively.

The generated PDG was used to analyze the dependence hierarchy of network application benchmarks. The output of the pass could also be fed into *Graphviz* to get visualized image. In the next chapter, the PDG will be input into the application partitioning and mapping algorithms to evaluate the performance of different partitioning and mapping heuristics.

Chapter 4 - Energy-Aware Program Bi-Partitioning and Mapping for Packet Processing System

4.1 Introduction

In chapter 3, a compiler module has been introduced for extracting program dependence information. Dependence information is vitally important for further code analysis. This chapter explores an energy-aware approach for program partitioning and mapping on to multi-core packet processing systems based on the results obtained from the PDG generator.

As introduced in chapter 2, the main function of a packet processing system is to perform packets processing tasks at the network level. The popularity of bandwidth-consuming services and real-time web applications (e.g. VoIP, virtual world and Internet of things etc.) has already made the traditional routers with simple store-and-forward structures obsolete. To meet the market demands, the multi-core platform has grown to be the de facto standard today, in terms of both the vendors' choices and researchers' focuses. The system architecture can be built upon general purpose processors such as the Intel x86-64 Xeon [63], or RISC-based network processors like Cavium's OCTEON [64] and NetLogic's XLR processors [65], or FPGA-based chips, for example, the NetFPGA project [66]. It is a natural choice for deploying packet processing applications on multi-core system since the packet parallelism can be easily exploited by core replication. Moreover, as the processing cores can handle a number of varied tasks at the same time, task-level parallelism can be better achieved in a multi-core environment.

Programming in a multi-core platform however implicates several daunting issues that are not obvious or are non-existent in a single-core processor [67]. This chapter looks into two of the most prominent, yet correlated, problems. The first challenge is how to schedule the miscellaneous tasks in the parallel processing cores; the second correlated challenge is how to control the overall system energy consumption under a reasonable budget. State-of-the-art network packet processing cores, such as OCTEON CN58XX, feature fast parallel processing units and hierarchical memory sub-systems. When developing applications on such a platform, either the programmer or the compiler has to know how to partition the parallel tasks and map them onto the processing cores. In theory, multi-core architectures can be configured into one of three topologies, namely pipeline, parallel or a hybrid of the two [49]. Fig. 4.1 illustrates a hybrid scheduling topology, where in stage 2 the cores are run in parallel and the three stages are run in pipeline connected by FIFO queues. The task mapping is flexible enough; however, how to obtain an optimal solution for a given set of applications, limited processing cores and performance / latency metrics is still an open question.

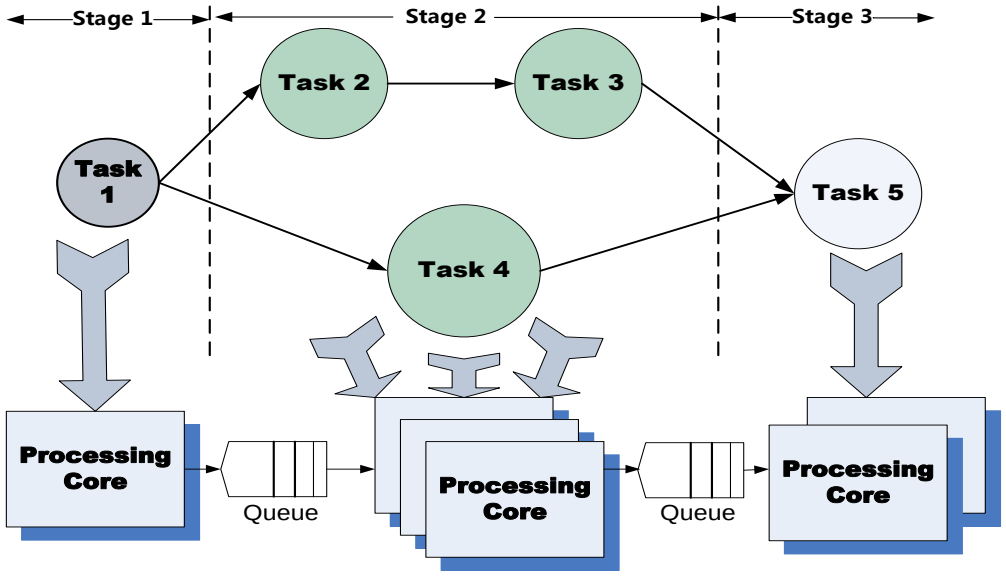

Fig. 4.1. Overview of multi-core packet processing system

Another prominent issue accompanying the wide adoption of multi-core systems is their greater hunger for processing power [68]. When deciding the architectural topology and scheduling the tasks, it is important to find a comprehensive method

that includes both consideration of the system energy consumption and throughput. While it is easy to scale up the number of cores and hence the productivity, it is sometimes a self-contradictory goal to increase both the power-efficiency and the overall multi-core performance.

This chapter proposes an integrated approach by extending the traditional bi-partitioning algorithm (Bi-Par) [69] in program partitioning and mapping to consider the trade-off between energy consumption and system scalability and versatility. The specific contributions the author makes include:

1. The author proposes methods for deploying multiple network applications on a multi-core network processing system based on program partitioning and task-to-core mapping. The algorithm takes both performance and energy-efficiency related metrics;
2. The author develops a generic framework with performance and power models to evaluate the multi-core packet processing system. The system can be configured in parallel, pipeline or hybrid mode in a flexible way;
3. The author gives the analysis of the proposed approach in respect of energy-consumption and system throughput;
4. A comparison with other related work is also presented.

The focus of the chapter is on its branch of Bi-Par. To the best of the author's knowledge, this is the first work on extending Bi-Par and program mapping with energy-saving considerations. The remainder of the chapter is organized as follows. Section 4.2 explains the application model and formally defines the problem it is solving. Section 4.3 describes the Bi-Par and task mapping algorithm for task allocation and scheduling in a multi-core packet processing system, together with a discussion of related approaches. Section 4.4 gives the results of comparison between the Bi-Par branch and other approaches. Finally section 4.5 concludes the work on this topic.

4.2 Preliminaries

4.2.1 Problem Statement

The PDG detailed in chapter 3 is used as the task graph to characterize the network applications. The instructions of a program are grouped together to form a *task* by consolidating those instructions within the same Basic Blocks (BB). The control-flow of instructions and data-flow of variables are both categorized as *dependency* among the tasks. Besides dependence information, it is possible to augment PDG with runtime profiling statistics. Fig. 4.2 shows an example of the augmented PDG generated the compiler module the author develops. The additional portfolio it possesses is block execution time, instruction sizes and branch frequency.

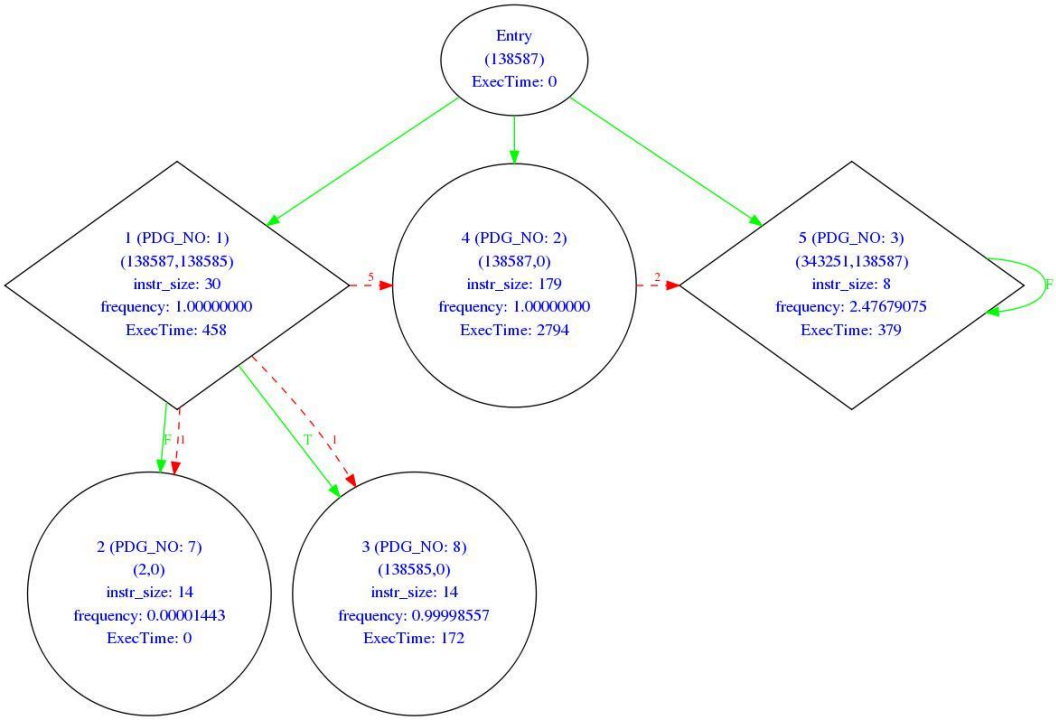

Fig. 4.2. Augmented PDG

As said in last chapter, the round nodes contain only non-branch statements, while diamond nodes have branch instructions at the exit. Node weight (as depicted by *instr_size* in the Fig. 4.2) is equal to the number of instructions each node contains. As for the edges, green ones depict control-flow dependency and red ones show data-flow dependency. Green edges can be labelled as "*True*" or "*False*" and the

red edges labelled with number of data transmits. The weight of the edge is equal to the communication cost to transmit the dependency.

Now it is to define a generic multi-core packet processing system that the application model (PDG) will be mapped onto. Let *N* be the number of available processing cores and each core's instruction store size is I_{max}. *N* cores can be configured freely in pipeline or parallel fashion like in Fig. 4.1. Suppose the pipeline has *T* stages, and in stage *i* the number of cores used is PE_i, then

$$\sum_{i=1}^{T} PE_i = N \tag{4.1}$$

In a stage, the packet latency will be determined by the sum of three factors, namely computation time, communication time between two stages, and memory access time of each stage. In this work the performance is measured from a system's viewpoint first, i.e. the system throughput.

If a task is mapped by duplication into *M* cores in one stage, one can take the *effective* computation time as a division of *actual* stage time by *M*. Multiple tasks can be mapped onto different cores in one stage, so the overall stage computation time and memory access time is subject to the slowest task. Suppose there are *W* tasks mapped onto one stage, then the effective stage time will be

$$\tau_{stage} = max_{i=1}^{W}\left(\tau_{comp_i}^{e}\right) + max_{i=1}^{W}\left(\tau_{mem_i}\right) + \tau_{comm} \tag{4.2}$$

where

$$\tau_{comp_i}^{e} = \frac{\tau_{comp_i}}{M} \tag{4.3}$$

The system throughput is decided by the slowest stage in the pipeline, so

$$\text{Thoughput} = \frac{1}{\max_{i=1}^{D}\left(\tau_{\text{stage}}\right)} \tag{4.4}$$

and *D* is the pipeline length.

As for the energy consumption (E), consider the classical equation

$$E = C \cdot K_a \cdot V^2 \tag{4.5}$$

for the computational cost. K_a is a task-processor dependent factor and *V* is the voltage neither of which are considered within this paper. But the cycle runtime C is relevant here. And the energy efficiency (*Eff*) is measured as

$$Eff = \frac{Throughput}{Energy\ Consumption} \tag{4.6}$$

Instead of reducing the computational energy cost directly, this chapter focuses on improving the energy efficiency. Due to scheduling constraints (dependency) and inter-task communication delays among the cores, it is not straightforward to simply raise the ratio of packets per cycle. The energy consumption of memory interfaces and inter-stage communication should be taken into account also. The details will be visited when discussing the simulation model in next chapter when the evaluation model is discussed.

The formal definition of the problem the author is solving is as follows. Given M_{app} network applications described by a PDG task graph and *N* processing cores that can be configured in a hybrid pipeline and parallel topology (subject to above constraints and equations), find an optimal task allocation and mapping approach that will increase the throughput rate while keeping the power consumption under control, resulting in increased energy efficiency.

4.2.2 Case Study

For the case study, this section took a typical packet processing system scenario with 8 cores, representing the mid-range market product, i.e. the OCTEON CN5840, and ran two network applications on the system, namely radix-based IP-forwarding (*IP-radix*) and AES-based IP packet encryption (*IPsec*). IP-radix is a header processing application while IPsec works on the payload. Because there is no dependency between these two applications, they can run in parallel in the system.

The simplest configuration by intuition would be two pipelines in parallel with one application mapped to each pipeline. Within each pipeline, four cores run in parallel. Theoretically, the throughput could be 8 times higher compared to a single core solution. However, this straightforward task scheduling and mapping is far from optimal. The following issues will constrain the overall system performance dramatically:

1. The computational need for the two applications varies considerably. A profiling run with a single core simulator [39] showed that the total

execution time for IPsec is 15000+ CPU cycles, whereas IP-radix counts for only 4000+ cycles. If configured as above (with 2 pipelines and 4 cores per application per pipeline), the output interface has to wait for the payload processing to finish, so the throughput will be much undermined;

2. Many network processors have a limited instruction memory for each processing core. The code size of IPsec is 3833 and IP-radix is 1551.It is likely that for some systems (e.g. Agere APP550) one core cannot hold the entire instruction base and the task has to be divided into pipelines;
3. In the pipeline configuration, it is desirable that each stage has approximately the same processing time so that very few core cycles would be wasted. But how to evenly distribute the processing time is not explicit without any profiling analysis. It is easy to fall into the trap of simply greedily feeding each core's instruction store.

Table 4.1 shows a partitioning and mapping example for running the combined IPsec and the IP-radix applications with the system resources as described. The configuration described in Table I produces the highest throughput as indicated by the proposed Bi-Par and also by manual tuning. *PE* is the number of processing cores, *I* denotes the number of instructions mapping to the stage (with I_{max} restricted to 2000, simulating conventional network processors) and *C* means the *effective core cycles* each stage would take. As explained previously IPSec is a computation-consuming application and is accordingly allocated 4 parallel cores in the first stage to reduce the effective core cycles.

Table 4.1. A partitioning and mapping example

Stages	**Resources**	**Applications**		**Parallel Cores**	
		IPsec	***IP-radix***	***IPsec***	***IP-radix***
1	I_{max}=2000 *PE*=5	*I*=1973 *C*=2645	*I*=812 *C*=2121	4	1
2	I_{max}=2000 *PE*=3	*I*=1860 *C*=2245	*I*=699 *C*=1986	2	1

4.3 Program Bi-Partitioning and Task Mapping

4.3.1 Base Algorithm

The decision problem formulated in section 4.2 is NP-complete [70]. To solve it the author adopted a divide-and-conquer heuristic, namely program bi-partitioning and recursive task mapping. The base algorithm is an application of the classical max-flow min-cut problem from network flow study [71]. The PDG is augmented as described in Fig. 4.2 to be a flow network with dummy entry and exit nodes. A min-cut will partition the graph into two sub-sets where the connecting edges would incur minimum flow values. In the case of PDG, this means that the edges with lowest dependency *weight* between two sub-tasks will be chosen. The workflow is given in Fig. 4.3. A detailed explanation of each step is summarized in Table 4.2.

Recall the equations that were deduced in section 4.2. The system throughput is determined by three factors, i.e. communication cost, computation cost and memory access time. The min-cut ensures that the algorithm always tries to minimize the communication cost. The balanced-weight property guaranteed by the step 3 in Fig. 4.3 ensures that the pipeline is evenly loaded so that very little overhead would be wasted in synchronization. There is of course certain trade-offs between finding minimum communication cost and balancing the pipeline. A deviation factor ϵ is adopted to allow a flexible exploration between the two goals, as detailed in Table 4.2. The cutting ratio γ is measured by the weights between two cuts, and can be used to find an arbitrary number of cuts of the original program by recursively running the Bi-Par.

After allocating the sub-tasks as indicated by the PDG cuts, one can assign each task with appropriate computation resources. In the proposed model, the nodes weight represents the computational needs (in terms of core cycles) and the edges weight labels the communication needs (interconnects between cores). So the author assigns each task with the number of cores in proportion to its nodes weight and the number communication interconnects in scale with the PDG edges weight.

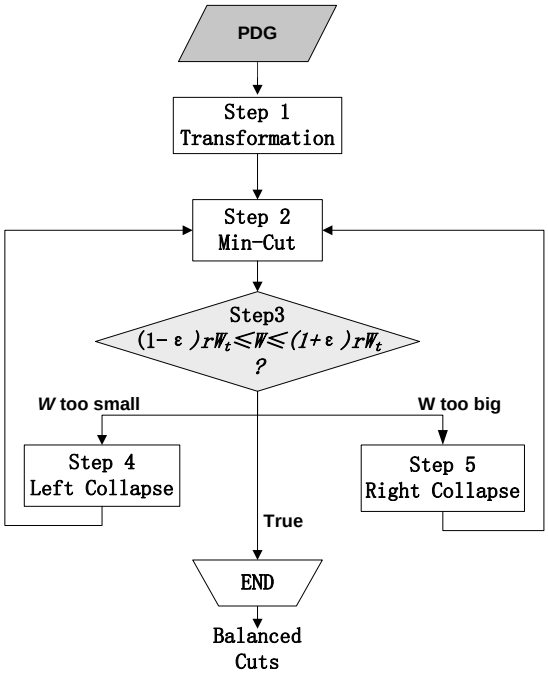

Fig. 4.3. Base recursive bi-partition algorithm

Table 4.2. Steps in recursive bi-partition

INPUT	**Flow Graph, ϵ,γ**
STEP 1	Identify the start and terminal node
STEP 2	Find a min-cut that bi-partitions the network into X and X'. Let W denotes the weights of X, and W' for X'
STEP 3	If $(1-\epsilon)\,\gamma W_t \leq W \leq (1+\epsilon)\gamma W_t$, then terminate
STEP 4.1	If $W < (1-\epsilon)\gamma W_t$, then collapse all nodes in X to start node
STEP 4.2	Select a node in X' and collapse it to the start node as well
STEP 4.3	go back to step 2
STEP 5.1	If $W > (1+\epsilon)\gamma W_t$, then collapse all nodes in X' to terminal node
STEP 5.2	Select a node in X and collapse it to the terminal node as well
STEP 5.3	go back to step 2
OUTPUT	**Two balanced cuts**

4.3.2 Energy-Aware Extension

The algorithm described in Fig. 3 only takes throughput performance into consideration and aims solely at increasing throughput. However, as discussed before, the energy consumption cannot be overlooked nowadays especially with the increasing number of cores on chip [72]. So the author extended the original algorithm with refinement steps using power-related data to increase the energy-efficiency. The data I profiled mainly contains:

1. **The average energy consumption on each processing core -** Recall that: $\mathrm{E} = \mathrm{C} \cdot K_a \cdot V^2$. Since V is constant here and K_a is not modifiable, its number of cycles (C) for a given task is profiled together with the respective energy consumption on each core;
2. **The energy consumption on interconnect**s **-** It comprises two parts, i.e. leakage energy as a function of running cycles and dynamic power related to the number of dependences between tasks on different cores;
3. **Energy consumption in memory interfaces.**

Energy-Aware Bi-Par Algorithm

Input: task graph $G(V, E, W_v, W_e)$, list of possible cores numbers
Output: task mapping matrixes;

```
1: for each number of cores N
2:     Bi-Par (G, N)
3:       Compute stage time and energy consumption for
             two cuts respectively, T1, C1, T2, C2
4:     for each boundary nodes Bi
5:         try migrate Bi to the neighbour cut
6:         re-compute T1', C1', T2', C2'
7:         if (T1+T2)/(T1'+T2') > (C1'+C2')/(C1+C2) then
8:             update the cut
9:             C1=C1', C2=C2', T1=T1', T2=T2'
10:        end if
11:    end for
12:    allocate cores based on cut_ratio γ
13:    if pipeline not even or code size > limit
14:        Bi-Par (Gi, Ni)   /* recursive bi-par*/
15:        same migration trials in recursive bi-par
16:    end if
17:    for the number of stages S, record the task
       mapping in a matrix M[S,N]
18: end for
19: return (M1[S1,N1], M2[S2,N2]…Mk[Sk,Nk])
```

During the task partitioning, each node's weight is collected in terms of both execution time and total energy. In the task mapping, the algorithm iterates over the sub-tasks residing at the edges of the graph between cuts, migrate each of them to neighbouring cores and find out which migration would reduce the product of stage time (in cycles) and energy consumption (in Joules) the most, thus improving the energy-efficiency as given in Equation (6) (line 4 to 11).

The intuition behind the refinement heuristic is that by migrating boundary nodes, a large search scope is available for optimizing energy-efficiency at the cost of a small throughput sacrifice. The proposed technique tries to identify any groupings of nodes with uniform memory accesses in order to minimize memory interface leakage. Interconnects leakage power is saved by turning off interconnects within un-balanced pipeline.

4.3.3 Other Approaches

A vast array of literature exists in the area of task allocation and mapping for multi-threaded and/or multi-core system [73][74][75][76]. As the focus of this work is on network processing applications, this chapter compares the proposed approach mainly with the studies in the networking area.

The early work proposed by Weng [31] employed randomization in program mapping. The tasks are randomly allocated to processing cores without violating dependency constraints. All valid mappings are recorded and the one with best throughput is filtered out in the second phase of the strategy. Near-optimal mapping is not guaranteed especially when the iteration time is limited.

Another heuristic described in [49] is based on greedy algorithm. It packs the task by filling one processing core with basic blocks until the instruction store is full. However, it does not take communication cost into consideration; so the mapping quality could be sub-optimal.

The work described here resembles the approach discussed in [52] most. Yu et al. also adapted Bi-Par for network processors. Their refinement focuses on throughput optimization and does not include energy awareness. In the experiments, the results are compared against these three approaches [31][49][52] and give a comprehensive comparison analysis.

In [77] Kuang and Bhuyan took power budget into consideration for task scheduling in packet processing system. However, their approach is based on Dynamical Voltage and Frequency Scaling (DVFS) which needs hardware support. Additionally, their method reduces power by extending the computation time, rather than optimizing energy-efficiency. In this regard, it is not fair to compare with their approach in this chapter.

4.4 Performance and Energy-Consumption Evaluation

To validate the proposed solution, the author implemented a simulation framework to allow easy and large design space exploration. It has the performance and energy models respectively. In this section the experiments will be described and the collected results using the proposed models will be discussed.

4.4.1 Testbench Framework

The author extended the SUIF/Machsuif compiler [78] with new passes that perform code analysis, PDG generation and Bi-Par mapping. Fig. 4.4 depicts the brief components and workflow of the test-bench. The application is first profiled with Halt passes provided by Machsuif [78] and the task graph with profiling analysis is fed into the PDG generation pass. The PDG module will collect all the information in an internal augmented PDG. Then program partitioning and mapping is carried out over the PDG. Task mapping results are input to the simulator to give performance and energy results. This process can be recursively executed to conduct comparison and optimization for a given application or a set of applications.

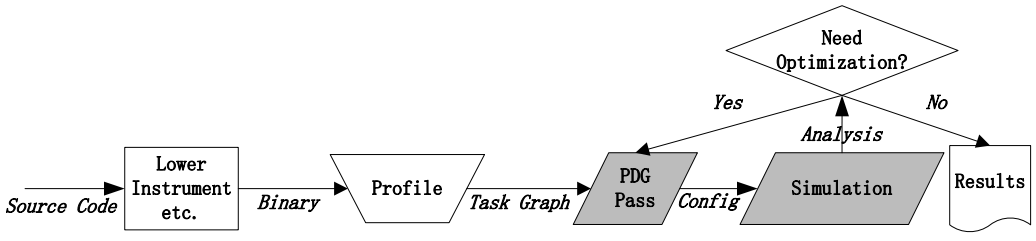

Fig. 4.4. Experiment Framework

4.4.2 Performance Results

In the system-level, the total throughput of a network processing system is the decisive measurement of the performance. However, the individual packet latency is also an important factor in many applications, e.g. real-time streaming). For comparison, the framework is used to evaluate three other approaches from the literature described in section 4.3. The benchmark applications are *LC-Trie* IP-forwarding, IP packet encryption (*IPsec*) and *Port-Scan* adopted from PacketBench [57]. The core frequency was set to 2GHz. Both memory access time and interconnects transmission time are assumed to be one unit of clock cycle.

Table 4.3 shows the throughput measurements for different combinations of the three applications with different numbers of processing cores. Since the code size limit is seldom a bottleneck for modern multi-core network systems, the number of pipeline stages in the evaluation were short. Thus the even number of cores is preferred to enable parallel processing across pipelines. In Table 4.3, application I is LC-Trie, II is IPSec, and III is Port-Scan. For all of the applications, the proposed energy-aware Bi-Par exhibits good scalability as the number of cores increase. The throughput gain is greater than double when cores are added from 8 to 16 and upwards. This is due to the free migration of tasks that have high communications cost between pipelines when processing resources are abundant. Bi-Par favours communication-heavy applications over computation-heavy ones since the base algorithm minimizes inter-stage communication cost. The throughput increase from 16 to 32 cores for PortScan is 269% while for IPsec is 244% in this case.

Table 4.3. Throughput for combinations of three applications on multiple cores

	Ⅰ	Ⅱ	Ⅲ	Ⅰ+Ⅱ	Ⅱ+Ⅲ	Ⅰ+Ⅲ	Ⅰ+Ⅱ+Ⅲ
2 Cores	0.56	0.18	0.11	0.11	0.05	0.07	N/A
4 Cores	0.91	0.33	0.28	0.18	0.11	0.12	0.04
8 Cores	1.65	0.75	0.6	0.41	0.39	0.41	0.12
16 Cores	3.78	1.98	1.43	1.12	0.88	0.9	0.41
32 Cores	8.75	4.85	3.85	3.1	2.12	2.43	1.45

To avoid any potential bias, LC-Trie plus IPsec were used in the performance comparison experiments. Fig. 4.5 illustrates the results when 16 cores are used for

mapping the two applications. By varying the number of stages, it is simulating different requirement for task code sizes. The proposed approach (BiPar-E) shows 33.1% throughput improvement over greedy in a 2-stage pipeline and 50.7% over randomization in a 4-stage pipeline. Randomization requires very large search space as discussed. When the pipeline is longer (i.e. more applications) and search time is predefined, it is hard to reach a good mapping. The energy-aware extension brings an average of 10% throughput decrease compared to Bi-Par without migration. It will be revisited with energy consumption data to validate if the efficiency is improved.

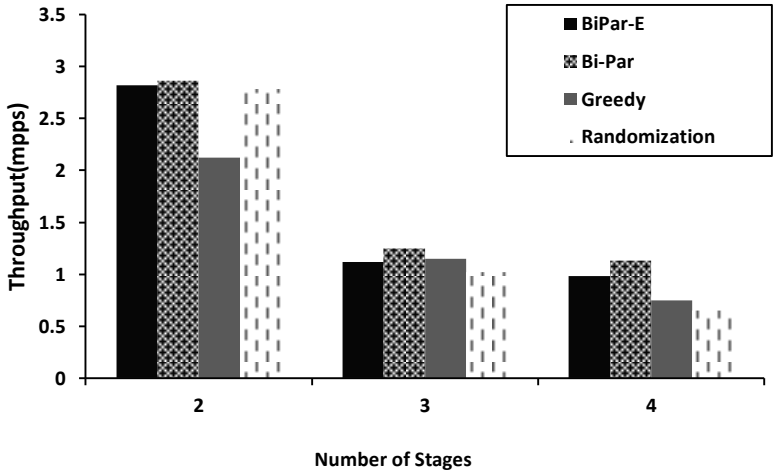

Fig. 4.5. Throughput comparison by number of stages

Fig. 4.6 summarizes the individual packet latency comparison for the three benchmark applications. For LC-Trie, four approaches generate similar results. For the other two applications, the latency difference is within 10% margin among the four approaches. And the proposed extension involved a slight 5% increase on average. A safe conclusion is that the energy-aware Bi-Par would not sabotage the individual packet latency even if system throughput is optimized for.

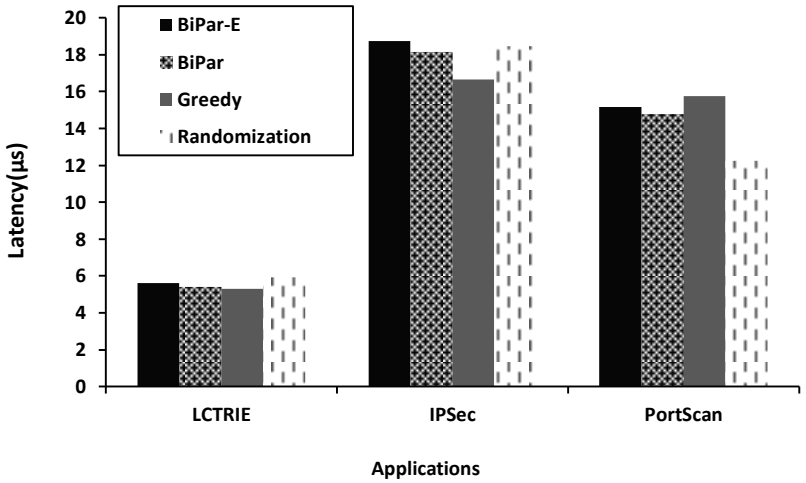

Fig. 4.6. Latency comparison by applications

4.4.3 Energy Results

The energy efficiency of the proposed algorithm is measured as the system throughput divided by total energy consumption (in Joules). The runtime power is usually an important indication of the energy-efficiency. However, traditional techniques such as DVFS just try to reduce power at the expense of longer runtime cycles. The total energy consumption could be well the same if not more in that case, implying that the energy-efficiency is not improved. Here the energy data from an efficiency perspective is organized as depicted in Fig. 4.7. The bar graph shows the total energy consumed by processing one million packets with three benchmarks respectively and in increasing order by the number of processing cores. The trend-line illustrates the energy-efficiency by graphing the throughput (in mpps) over energy consumption (in Joules). In all benchmarks, the energy-efficiency is clearly on the rise as the number of cores is scaled up. It proves the energy-aware Bi-Par to be particularly beneficial in a large system with dozens of processing cores. For LC-Trie, 25.4% increase of energy-efficiency is noted when cores are populated up from 2 to 16. The corresponding increase for IPsec is 168% and 29.4% for PortScan. The dramatic rise for IPsec is majorly attributed to the little heat overhead in interconnects and memory interface, especially the leakage power (which is considerably larger in LC-Trie).

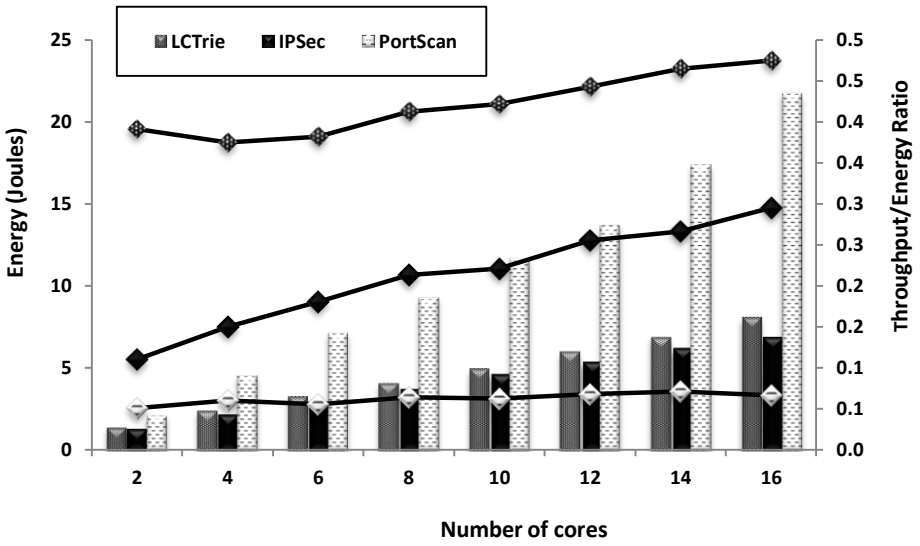

Fig. 4.7. Energy consumption comparison by applications

To explicitly demonstrate the energy-efficiency gains of the proposed Bi-Par and extension, the energy-related data of other three approaches are collected in the simulator as well. It used 8 processing cores and set each core's maximum code size to be 2000. The results are shown in Fig. 4.8. For all the three benchmarks, the proposed algorithm not only excels the original Bi-Par without energy-aware refinement, but also generates better mappings than greedy and randomization. In IPsec, BiPar-E gained 34% energy-efficiency increase by migrating tasks in the refinement step. The outstanding gain is mainly because the availability of many sub-tasks at edges and little back-dependency among them. The power on processing core is the decisive factor for IPsec so the migration can take considerable effect. By nature, migration refinement can have little impact on memory and interconnects energy consumption except for leakage power. Yet in LC-Trie and PortScan an average of 10% efficiency improvement is still observed after refinement step. Therefore, the proposed algorithm proves promising and advantageous both in terms of scalability and universality.

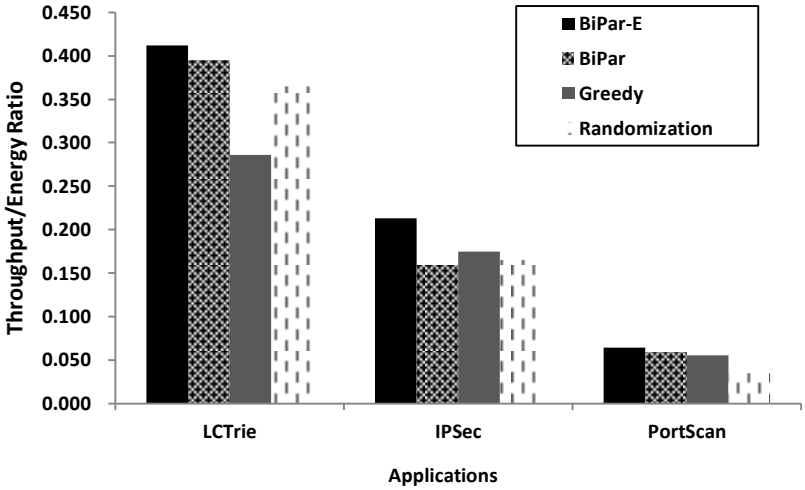

Fig. 4.8. Energy efficiency comparison by applications

4.5 Conclusions

The sharp increase in bandwidth requirements and versatility of network applications has prompted packet processing systems to widely adopt a multi-core multi-threaded architectural design. A challenging issue when programming such a system is how to fully utilize the processing power in a pipeline-parallel topology. As the power consumption increases, maintaining the energy-efficiency of the whole system also becomes delicate.

In this chapter, the author proposed an energy-efficient program partitioning and mapping algorithm for packet processing systems [79]. The approach is based on Bi-Par and built into a compiler suite. The algorithm searches for an optimal configuration of the pipeline depth and the width of each pipeline stage. Steps taken to optimize the performance include iterations over the sub-tasks at the pipeline edges, and performing migration of tasks between cores to improve energy-efficiency. The author also implemented an evaluation framework to simulate the multi-core network processing system in terms of performance and energy consumption. The simulation results show that the proposed approach improves the energy-efficiency in all three benchmarks by between 8.04% and 34%, with a marginal loss in throughput in comparison with three other partitioning and mapping algorithms, i.e. greedy, randomization and base Bi-Par.

Chapter 5 - Performance and Energy Evaluation Model

5.1 Introduction

As explained in chapter 4, optimal configuration of a multi-core packet processing system at the architecture level is the key to maximize the performance and to minimize the cost. The author has explored the problem of optimizing system-level topology as given in the last chapter. However, it is not straightforward to validate the methodology and present quantitative analysis of the results without a valid yet efficient simulation tool. The tool needs to feature at least two strengths,

1. It should be easily configurable at the architectural-level so that a large number of topologies for a given packet processing system can be simulated without much manual intervention;
2. The simulation speed needs to be fast enough to allow the search in a large space with acceptable margin of error. The simulation time should not grow exponentially with the number of architectural components (e.g. processing cores).

Existing tools such as NePsim [80] and SimpleScalar [39] are either too ISA-specific or time consuming. SimpleScalar does include a high-level profiler that is quick to execute; however, that profiler would not generate any energy-related statistics. With Sim-Panalyzer [81] add-on, it can collect detailed and various power data, while the simulation speed drops down to a few hundred target cycles per second. That means it would take even a day to simulate a typical network application's runtime that process just thousands of packets in a few milliseconds. So the author implemented an analytical simulation framework that

satisfies the needs. The framework incorporates both performance and energy models.

This chapter is organized as follows. Section 5.2 elaborates the analytic model of multi-core packet processing system in detail, classified by performance and energy models respectively. Section 5.3 presents the evaluation results and correctness validation results. The chapter concludes in section 4.5 with a summary of the work.

5.2 The Analytical Model

5.2.1 Motivation

In Fig. 2.5, three possible topologies of network processors configuration are illustrated, i.e. pipeline, parallel and a hybrid of the former two settings. Processors run in parallel can execute the same task to properly utilize the data-level parallelism or packet-level parallelism which is abundant in network applications. It can also be regarded as a type of task duplication [82]. Multiple cores can also run in a parallel mode with different tasks. Those tasks would not bear any inter-task dependence nor shared resources. In other words, they are independent tasks, with little coupling issues with other tasks. In both ways, the performance, especially the traffic throughput can be greatly increased by employing additional cores to run in parallel.

Pipeline configuration is also widely employed in network processing system for two reasons. One is that compared to data parallelism, the benefits of pipeline parallelism particularly apply to data-intensive applications, because it significantly reduces the contention for shared resource (e.g. bus, external RAM) in a multi-core system. Second reason is due to the rapidly growth of the network application complexity, the code size of a large network system usually exceeds the instruction store available in network processors. As discussed in chapter 2, the network processor is an evolution from ASIC and GPP design. Like many lightweight RISC embedded processors (e.g. ARMv5TE), the instruction memory on the die is very limited. Table 5.1 lists the memory size of some commercial network processors. It can be observed that most processors have instruction memory in the range of 1KB to 100KB. As a comparison, the code size

requirement of five typical packet processing tasks is listed in Table 5.2. The code was derived from PacketBench toolset [57] and manually inlined by the author. The code size denotes the number of instructions of that application. From the table, it can be seen that the code size of complex packet processing tasks such as Portscan has already exceeded the capacity of on chip instruction store. Yet, modern packet processing system usually has to take on a good many tasks at the same time. That would definitely worsen the problem. In this case, it is nature to employ a pipeline of processors to make up the deficiency. This method is similar to the idea of software decoupling [83], but not exactly the same in the context of network processors.

Table 5.1. Size of instruction memory

Network Processors	Instruction Memory Size (bytes)	Word Size (bits)	Frequency (Hz)	Number of Cores
Intel IXP2805	8,192	32	1400	16
Hifn 5NP4G	32,768	32	133	16
Agere APP550	256	128	266	3
AMCC NP3740	16,384	32	700	3

Table 5.2. Code size of packet process applications

Tasks	Description	Code Size
IPv4 Forwarding 1	Trie-based route table lookup	1548
IPv4 Forwarding 2	Radix-based route table lookup	1551
Flow ID	Flow hashing based on 5-tuple	3632
Portscan	Monitoring abnormal activity	6443
IPsec	AES encryption	3833

A distinct feature of a network processing system is the flexibility in architecture-level configuration, as was stressed in chapter 2. Ideally, given a network application, an optimal solution employing a hybrid of parallel and pipeline architectures can be found, like the one depicted in Fig. 4.1. In chapter 4,

the author has strived to optimize the topology taking both performance and energy results into consideration. Yet as explained in section 5.1, the researcher would need a simulation tool to quickly get a rough idea whether the optimization will really increase the performance, rather than in the opposite way. For such a tool, flexibility and speed come in the first place. Then it is for the correctness. Bearing that in mind, the author chooses an analytical model and implemented it in a mini simulator. The simulator works seamlessly with the compiler modules as illustrated in Fig. 4.4.

5.2.2 Performance Model

The model accepts PDG as the task graph. The PDG is augmented to bundle with profiling analysis information. Thus from the PDG, the simulator can extract a number of parameters, e.g. the number of instructions executed on each core, the runtime measured in cycles, communication time between stages and the number of memory accesses. These parameters can be input to the model together with the architectural configurations. The overview of the model is depicted in Fig. 5.1. In 0[84] Weng et al. described a similar model for evaluating their Annotated Directed Acyclic Graph (ADAG) mapping. The author adapted their model to fit the PDG partitioning and mapping environment.

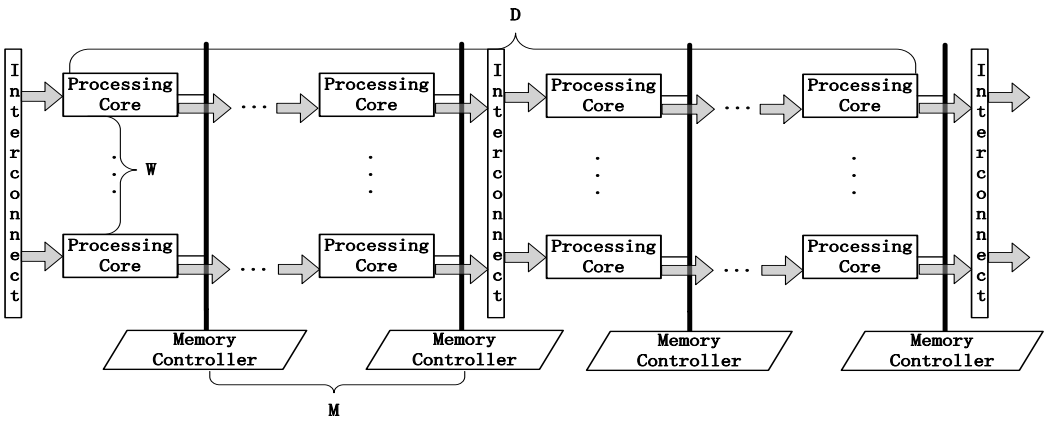

Fig. 5.1. Simulation model for performance and energy evaluation

In Fig. 5.1, processing cores represent generic processing units in the data plane; interconnects are FIFO-like buffer to transmit the tasks along the pipeline; and memory controllers are interfaces between cores and SRAM.

1) Architectural Parameters

The major parameters for architecting the system are: pipeline depth (*D*), width of each pipeline stage ($W_i, 1 \leq i \leq D$), number of memory channels shared by one stage of cores ($M_i, 1 \leq i \leq D$), the number of interconnects and the number of stages per communication interconnect (*I*). By setting different architectural parameters, the simulator can easily explore the effects of different topologies. For instance, given a fixed number of cores, whether a deep pipeline or high parallel configuration is favoured can be tested by adjusting D and W accordingly. When examining the PDG partitioning and mapping, the pipeline depth and width of each stage should also be setup based on the compiler's mapping knowledge.

To separate the system specific parameters, the simulator takes a standalone configuration file (default to config.cfg) for input.

2) Task Mapping Specification

In the performance model, the author re-used most of the information collected in the annotated PDG. Since the output of the mapping algorithm already gives the number of instructions and cycles of each stage, the compiler module pass those values in an array parameter. The number of memory access is also recorded here. The dependency across processing cores represents the communication cost and it is a bit tricky to adapt when the simulator duplicates a task mapping (making it parallel) in one stage. Instead of passing the dependency between stages directly, the author expands the array into a $\mathrm{M} \times \mathrm{N}$ matrix where inter-core communication between any two cores can be setup.

The task mapping specification file is generated automatically by the compiler's Bi-Par and mapping module.

3) Stage Time and Throughput

The key metrics of the performance model can be deduced by following the equations 4.2 – 4.4. The pipeline stage time is calculated by Equation 4.2 and τ_{comp_i} is the number of core cycles divided by the clock frequency (set in the architecture file). Memory access time is derived from a Machine Repairmen model [84] and communication time is a linear function of inter-core dependencies.

Finally the system throughput is given in Equation 4.4. Since the proposed model is capable of mapping multiple applications, the throughput of each application should be summed up to calculate the overall throughput in that case.

5.2.3 Energy Model

As the aim is to use the simulator to justify the task mapping quality, an analytical model for estimating the power consumption is included as well. The model uses a bottom-up method to evaluate the energy data of each component respectively and sum them up in the end.

1) Core Energy

Core power is dissipated both during idle time and job runtime. The author adopted the power data from Intel IXP2805 [3] to estimate the core power with respect to the number of active cores and each core's utilization. IXP2805 is a multicore network processor running at 1.4GHz/1.3V from which the proposed model can take sound samples. In Table 5.3 the power data for even number of cores (in many real cases even number of stages pipeline is optimal [49]) is summarized. "Typical" describes the average power consumption (*W*) measurement for 70% core utilization, while the worst case row is for 100% utilization. It is observed that a near-linear increase of dynamic power is in line with the growth of utilization for the applications given in Table 5.2 in Sim-Panalyzer [81]. Thus, when estimating the core energy, each core's dynamic power would be decided by its utilization multiplied by the worst case power. The total core power is the dynamic power added to the static power.

Table 5.3. Core power estimation

Number of Cores	**2**	**4**	**6**	**8**	**10**	**12**	**14**	**16**
Typical (W)	18.31	19.19	19.72	20.43	21.13	21.84	22.54	23.25
Worst Case (W)	21.73	22.59	23.45	24.31	25.17	26.30	26.89	27.75

2) Interconnects Energy

Interconnects can be viewed as FIFOs between stages for transmitting inter-core dependencies as depicted in Fig. 5.1. To compute the power consumed by this

component, the simulator also collected static leakage power and average dynamic power for conveying one-unit (4 bytes) dependency variable in SimpleScalar. With the number of inter-stage dependencies and runtime cycles from profiling analysis, it is able to figure out the dependency-related energy consumption.

3) Memory Interface Energy

The energy consumed on the memory interface is directly related to the number of memory accesses which is available from the profiled PDG task graph, as the sum of memory reads and writes. The data on leakage power and average energy per read / write is extracted from SimpleScalar samples. Briefly, the total data-related energy is given as

$$E_d = N_r * E_r + N_w * E_w + P * C \quad (5.1)$$

where N_r and N_w are the number of reads and writes on each core respectively; E_r and E_w are average read / write energy from sampling run; and P is leakage power per core cycle. C is the number of runtime core cycles.

5.3 Evaluation Results

5.3.1 Correctness Validation

Being an analytical tool, the flexibility in accepting a variable number of architectural parameters and the simulation speed are paramount. However, the model has to be validated to give sound results for generic multi-core packet processing systems. In the performance experiment, the author runs the network applications in Intel Architecture Tool [85] and compare its results against the analytical model (i.e. the Mini-Sim simulator). The benchmarking network applications are IPv4-trie, IPv4-radix and PortScan. Table 5.4 shows the hardware specification that both Intel AT and Mini-Sim are configured to. And the results are plotted in Fig. 5.2.

In the legend of Fig. 5.2, T-AT stands for system throughput collected from Intel AT while T-MiniSim is for the analytical simulator. L-AT depicts the individual packet latency generated by Intel AT and L-MiniSim is for the latency figures

from the analytical simulator. For IPv4-lctrie and IPv4-radix, all the figures that Mini-Sim generated are within 15% difference from Intel AT. The most significant difference lies in latency values of PortScan. Since PortScan occupies a large instruction store, it is likely that Intel AT has some internal thread (context) scheduling that results in a larger latency number. Most importantly, the trend and inclination between the two pairs of lines are nearly identical, meaning that the tool is valid in identifying the impact on the performance from the benchmarks.

Table 5.4. Testbench configuration

Processor	IXP2800
Frequency	1.4GHz
SRAM (on chip)	32MB
DRAM (external)	512MB
Number of Cores	16

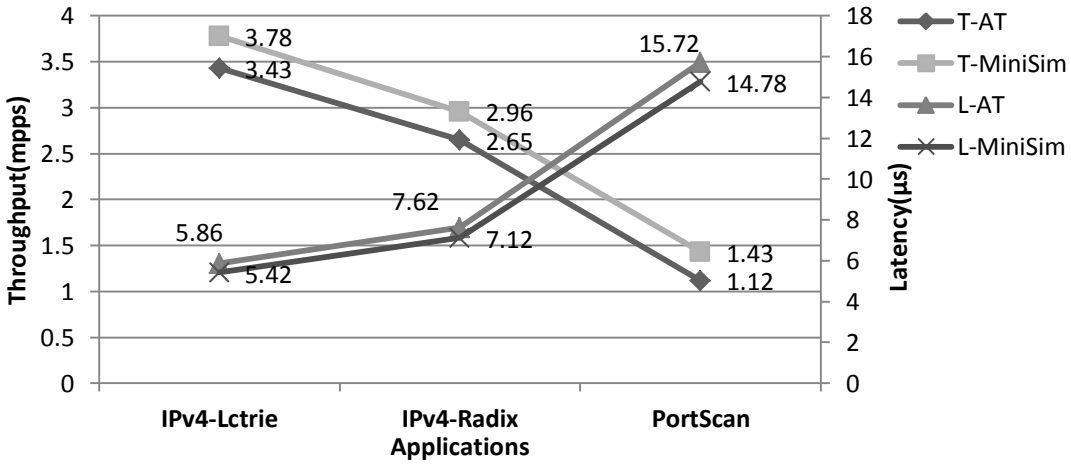

Fig. 5.2. Performance validation by varying the benchmarks

Fig. 5.3 gives the throughput and latency data from both tools by varying the number of cores for just one benchmark, i.e. IPv4-lctrie. The vertical bars covers 15% difference margin from Intel AT and it can be seen that except for one case in L-MiniSim, all other data Mini-Sim generated fall into that range. The larger difference in latency for 2 cores results from less detailed modelling of micro-architectural components in Mini-Sim. When the instructions memory is denser for 2 core scenario, the inter-instruction latency may have more direct

influence where Mini-Sim tends to ignore. However, as can be clearly seen in the figure, the trends between the two pairs of lines are again nearly the same. The researcher is thus able to use it to explore the impact of different architectural settings in a generic multi-core packet processing system.

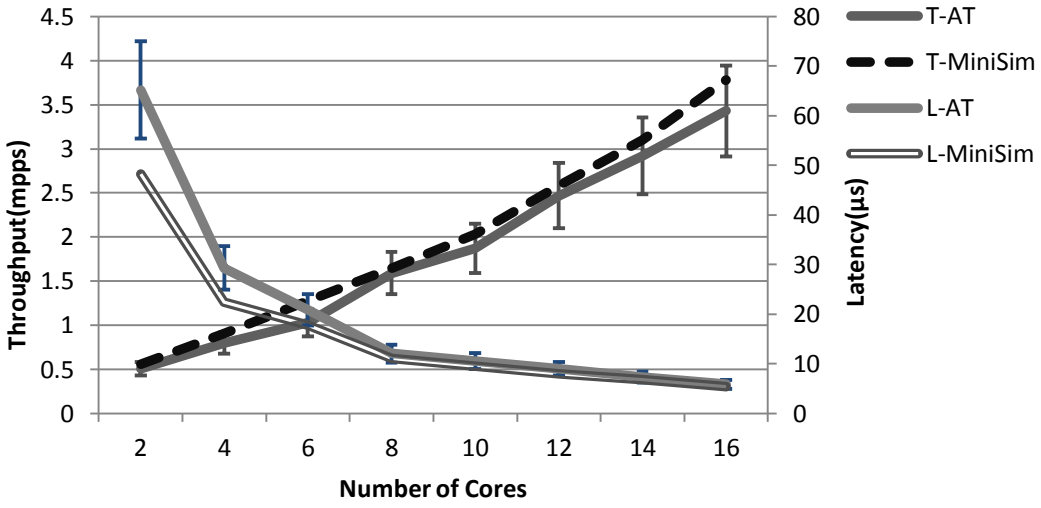

Fig. 5.3. Performance validation by varying the number of cores

Now it comes to the energy-related data. The energy consumption by processing cores (*core*), memory (*mem*) controller and interconnects (*ic*) are collected respectively. As a comparison, the author also implemented the benchmark IPv4-Lctrie in Sim-Panalyzer and collected the corresponding energy figures. All data has been normalized into SI unit Joule as the total energy consumption for a given application is what the researcher really cares.

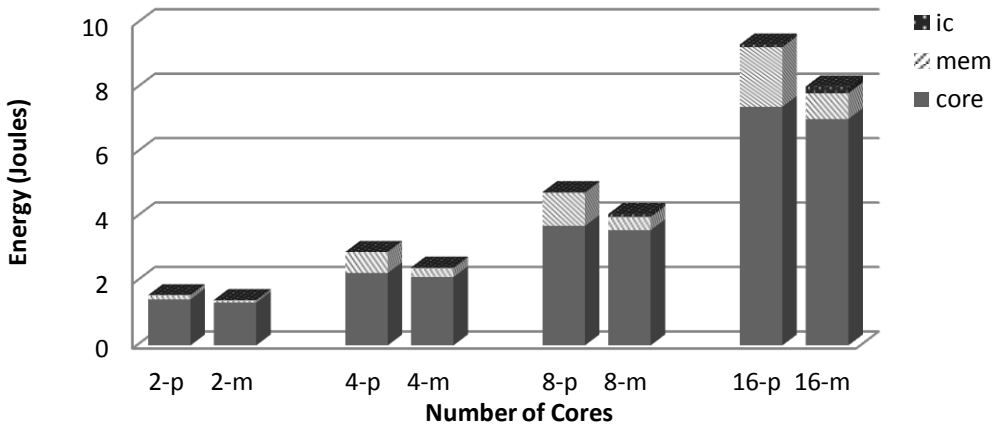

Fig. 5.4. Energy validation by varying the number of cores

Fig. 5.4 illustrates the results in a bar graph. In the x-axis, 2-p means the data is collected from Sim-Panalyzer by configuring 2 processing cores up running while 2-m meant the data is from Mini-Sim. The processing cores take about 90% of the total energy consumption in all cases while interconnects usually count for just 1%. The figures generated by Sim-Panalyzer are generally larger than those from Mini-Sim since the analytical model is coarser-grained. Yet again, it is observed that the difference is within 15% margin in all settings. As the tool itself is used to quickly validate the architectural optimization algorithms, this margin of error is acceptable in measuring the effectiveness of the optimizations.

5.3.2 Simulation Results

Section 4.4 gives some of the simulation results to compare energy-aware Bi-Par and three other approaches using the analytical model. In Table 5.5 it elaborates the data Mini-Sim is able to present and compares them to the figures from Intel AT and Sim-Panalyzer to run IPv4-lctrie for 10000 packets. Most significantly, the simulation time for running the Mini-Sim is on the order of a few milliseconds since it is inherently an analytical tool. The Sim-Panalyzer does not support multi-core simulation natively. As a comparison, the author adopted an extended

Table 5.5. Comparison of Mini-Sim and other simulators

Method	**Mini-Sim**				**Intel AT/Sim-P**			
Cores	2	4	8	16	2	4	8	16
Throughput (mpps)	0.56	0.91	1.65	3.78	0.51	0.8	1.59	3.43
Latency (μs)	48.2	22.45	11.12	5.42	65.2	29.3	12.1	5.86
Energy (Joule)	1.41	2.42	4.08	8.04	1.56	2.9	4.76	9.34
Sim Time (s)	0.41	0.34	0.38	0.37	1.14	17.17	172.3	N/A

version of simplescalar to collect the simulation time. Because simplescalar is an execution-driven simulator and records many fine-grained statistics, the runtime is soaring with the increasing number of architectural components. In the experiment, 8-core simulation already took nearly 3 minutes to finish. Considering that the actual application just took a few thousand cycles (i.e. a few

milliseconds) to finish, the simulation speed is far from fast enough to allow the researchers to perform generic topological optimization explorations.

So drawn from the simulation results, it can be concluded that the analytical simulation model and tool satisfy the requirement proposed in section 5.1, that is, fast and flexible enough with sound statistics.

5.4 Summary

This chapter has introduced an evaluation model for generic multi-core packet processing system. The model is analytical and can give both performance and energy data for various subsystem components. A mini simulator Mini-Sim is implemented based on the model for exploring architectural optimizations especially on choosing the best topology among pipeline, parallel and a hybrid settings. The advantages of the analytical model over heavy-weight simulators are: it is flexible to change the pipeline-parallel topology parameters so it is easy to use; the simulation time is extremely fast since it does not actually execute the code-path. Yet the profiled statistics is close to those figures that the real simulators generate.

In the experiment, the author validated the correctness of the analytical model by running three typical network applications on both heavy-weight simulators and Mini-Sim. The performance results are compared between Intel AT and Mini-Sim while the energy results are examined between Sim-Panalyzer and Mini-Sim. The number of processing cores and the number of pipeline stages are also varied in the experiment. In all cases, the relative differences are all within 15%. When varying the number of cores, the trend-lines of the output data from the heavy-weight simulator and Mini-Sim are parallel on the whole. The simulation time totally favours Mini-Sim since it can always finish within a few hundred milliseconds. All in all, it is proved that the model and the simulation tool are valid and efficient in exploring the architectural optimizations for generic multi-core packet processing systems.

Chapter 6 - Conclusions and Future Work

6.1 Summary of the Research

Packet processing system is a comprehensive solution specifically designed to provide the computation power required in today's computer networks. New applications could be written to extend the system capability and the number of processing cores can be scaled up to avail of the workload parallelism. However, such flexibility and processing power cannot be fully utilized without a suitable programming environment. The compilation toolset is important in mapping the handwritten application onto the multi-core platform. The quality of the generated machine code would largely determine the overall system performance in terms of packet throughput, individual packet latency, core utilization and energy efficiency.

This research focuses on the energy-aware optimization for packet processing systems in a compiler framework. The multi-core packet processing system and its major characteristics have been reviewed and the particular issues in networking domain are investigated. The work has been carried out on the program dependence analysis of network applications, as well as on program partitioning and mapping based on the dependence information. To fully achieve the computational potential of the multiple cores, the inherent modularity of the applications is carefully analysed and the generated modules are mapped onto the processing cores appropriately. The multi-core architecture can be configured to be parallel, pipelined or a hybrid of both. An extension of Bi-Par for optimizing energy-efficiency has also been proposed and carefully validated. In experiments, an analytical model which is able to quickly evaluate the effectiveness of the optimizations in a large search space is built and a mini simulator is implemented

based on it. To the best of the author's knowledge, this is a pioneering piece of work on extending Bi-Par and program mapping with energy-saving considerations without much performance loss.

6.2 Future Work

Chapter 4 describes the work on energy-aware recursive program bi-partitioning and mapping for network processing systems in detail. However, as reviewed in chapter 2, there is a vast area to explore in network applications domain for compiler optimizations. In this chapter, a couple of topics that could be the extension of the existing work are proposed.

6.2.1 Cooperation with Runtime Management

The work conducted in chapter 4 took advantage of a combination of static and dynamic profile-based analysis. The compiler module of the program dependence graph generator collects the communication cost by analysing the control flow graph of the program statically. Dynamic information such as the execution time and frequency are gathered by profiling the applications with a set of instruction traces. All the information is incorporated into an augmented program dependence graph. The subsequent optimizations, such as task scheduling and program mapping, are performed in turn availing of the graph. This framework is flexible enough to include additional program analysis. One way to extend such a framework is to interface it with the runtime management of the packet processing system.

By the nature of the network applications, many key profiling results are dynamic. For instance, in a level-compressed trie-based routing table lookup task, the number of comparisons cannot be determined until the destination address of the ingress packet is retrieved. The packet that finds its trie pointer in the tree within two branches would consume much less computation power than those traverse down to more than five levels. Besides, the traffic is usually dynamic in quantity as well over different periods of time. Enterprise network is busy in working hours while the volume of packets in home network is low. And the situation usually goes the other way around at night. So instead of generating a valid processor scheduling and program mapping off-line and configure the system

accordingly, to maintain the system's performance at its maximum, adequate management must be made during runtime. The program recursive bi-partitioning and mapping described in chapter 4 could be well employed in runtime for packet processing system. Nevertheless, the parameters of the model change so fast in dynamic traces that a given mapping result can lose its optimality very quickly. Moreover, unlike the compiler module executed off-line, very limited resources are available for runtime management while the time constraint is of first priority. If a refined mapping takes a long time to generate whereas the dynamic traces change considerably quicker, it will be pointless to make any runtime adjustment, since the new mapping is invalid before it is actually downloaded and up running. Further research should investigate the trade-off between time frames to perform runtime adjustment and the changing characteristics of the workload.

Another approach to cooperate with the runtime system is to monitor the power consumption online and aim to minimize it by adjusting the configuration dynamically. Like the procedures taken to collect the communication cost and execution time, given a packet processing system with N processing cores and a set of traffic traces, the traffic volume can be classified into several groups based on the power consumption. Such information can be recorded by the compiler. During runtime, the compiler can bestow the record upon runtime management module so that the latter can adjusts the power consumption of the whole system accordingly. When the traffic volume becomes low it can turn off processing cores to reduce energy consumption and as soon as it detects the increase of traffic line rate, additional cores are waken up and restored. In addition to the base algorithm presented in chapter 4, other heuristics can also be applied to preparing the group of mappings here, such as randomization and greedy duplication.

6.2.2 Data Mapping

In addition to the task mapping described in chapter 4, data mapping algorithm can also be designed to minimize the energy consumed in data communications. The problem of data mapping in a Network on Chip (NoC) has been formalized in [86]. A similar model may be constructed for packet processing systems. For example, the compiler might auto-detect where to place the route table in an IP forwarding application (SRAM or Scratchpad depending on the table size and the

available memory), and how to transmit the inter-processor variables in a pipelined application (via ring memory or message queue).

Apart from energy optimizations, the placement of application data also has direct and extensive effect on the system throughput and energy efficiency. The architecture of a packet processing system with homogeneous multiple processing cores and heterogeneous memories is similar to the Non-Uniform Memory Access (NUMA) multiprocessor system. In NUMA architecture, the multiple processors share a single on-chip bus to connect to the system memory and each processor has its own local data storage. The access latency to the global memory is usually an order slower than that to the local memory. But the capacity of local memory is much smaller than the shared memory. So the former is preferred for storing data that is used most often in a program. The compiler can aid on the decision of program data placement. Like profiling the control flow and dependence, a compiler module is able to record the access patterns of a processor to the memory within a given instruction trace. Optimizations could be made by migrating elements with high data reusability and similar lifetimes onto the local memory. The layout of the data elements can be also optimized in the local memory address space to minimize fragmentation. The local memory is also designed to take advantage of temporal locality and spatial locality available in the program. Within a network application, the loop iterations are not that abundant like those in digital processing or scientific computing. Therefore careful exploration is needed to evaluate the gains of locality optimizations. Moreover, in NUMA system, attention should also be paid to the data consistency in the distributed local memory image. Coherence across the local memory has to be enforced and maintained by the system at any point during runtime.

In [87], a framework for exploiting task, data and software pipeline parallelism comprehensively was proposed for stream programs. It is interesting to see how to combine the aforementioned techniques in a packet processing system as well. Currently, the task-parallelism and data-parallelism are extracted individually and corresponding partitioning and mapping heuristics are performed solely with the knowledge of one or another. The authors in [81] proposed an architecture independent high-level programming language to describe the program

parallelism explicitly. Other means of representations of the program could be utilized to expose the parallelism still, i.e. program dependence graph and task graphs. After extraction, the parallelism can be implemented by both hardware pipelining and software pipelining. Software pipelining is more flexible to use, e.g. it can conduct nearly arbitrary partitioning, at the expense of additional prologue overhead etc. This comprehensive framework of parallelism extraction, exploitation and implementation is far from complete and vast design space exploration is yet in need.

In the networking domain, Click modular router [78] provides a well-rounded framework for the development of new packet processing programs. It maximizes the use of modularity in essentially all network applications and assembles a router by a combination of packet processing elements. The rich built-in libraries with numerous elements empower Click's functionality and greatly reduce development complexity. The task-level parallelism is inherently visible in Click design; yet the parallelism has not been taken advantage of. There has been symmetric multiprocessing (SMP) version of Click run on Linux kernel, as well as an adaption of Click to network processors. Still, the framework could be extended to exploit task, data and pipeline parallelism at the same time. This is an interesting issue to be delved into.

6.2.3 Instruction Level Optimizations

The work conducted currently is at a relatively high abstraction level. In chapter 3, the program dependence is summarized based on the basic block of code. In chapter 4, the whole program is divided into sub-tasks, and the system schedules and maps them onto different processing cores. The coarse granularity in task creation is natural because most network applications are inherently modular, a feature that is also the basis for the success of Click software router [34]. However, the high level optimizations would in no way hinder the employment of fine-grained instrumentation. One processing core can be dealt with independently after the task creation and mapping. In such a case, traditional compiler optimization techniques could still be deployed.

An early work on instruction level optimizations for scientific applications was presented in [88]. It proposed an estimation model that can calculate the energy

consumption, code size and execution cycles of individual instructions. The model can be even tuned to give the energy estimation of each instruction in datapath, cache, memory, bus and clock components respectively. Though the ILP based optimizations they proposed are not very flexible, the estimation model is promising in further research on fine-grained optimizations, as long as the accuracy of the estimation results can be validated.

Network applications can be classified by functionality into two sub-groups, i.e. header processing and payload processing. Most applications fall into one group with a few going to both. The computational complexity of payload applications is usually much higher than that of header applications. Using profiling benchmarks like Packetbench, one can collect the instruction count and pattern for a suite of network applications. From previous analyses [57], it is observed that the complex payload applications, which require heavy data computation and transformation such as IPSec-AES packet encryption, generated considerably longer instruction sequences than simple header applications, such as 5-tuple based flow classification. However, header applications have much tighter latency constraint than payload applications in most cases. And in a network processor, certain types of instructions like the conditional branch and floating-point number arithmetic are expensive to implement while others may cost much less, such as the unconditional branch and bit-level comparison. Given a thorough instruction cost and power model, together with an instruction pattern for a given application, the compiler can trade some expensive instructions (in terms of instruction latency and energy cost) for lighter ones to reduce either latency or power consumption. In some extreme cases, the compiler might detect a sequence of self-contained costly operations. It may be rewarding to remove them from the code, and repack the operations in a separate task which can be dedicated to a hardware accelerator.

At instruction-level, the compiler could also make use of the special features of the instruction set provided by the processing cores. For example, Infineon network processors allow sub-word register access of packet bits. And Intel IXP micro-engines can find the first bit set in a register using just one instruction. This kind of packet-level address access not aligned at the processor word boundaries typically does not exist in general processors. Based on the instruction cost and

power model, an intelligent compiler would be able to identify the hot spot in the hand-written code where these instruction add-ons can be placed to optimize the code.

APPENDIX A - PDG PASS IN DETAIL

A.1 PDG Class

The PDG class is responsible for generating the PDG representation as introduced above. The input is a CFG form of the program provided by SUIF. The whole process of PDG generation is divided into three main phases, namely,

- Generate Post-Dominance Tree (void PDG :: generate_PDT(OptUnit*));
- Generate Control-Dependence Graph (void PDG :: generate_CDG());
- Generate Data-Dependence Graph (void PDG :: generate_DDG(OptUnit*));

And each phase is associated with a corresponding class method with the name given above.

To maintain the data structure of the PDG during the generation, the following properties should be kept within the PDG class,

- The set of PDG nodes (suif_list<Pdg_node*> PDG::_nodes);
- The data dependence edges (Set<Ddg_edge> PDG::_ddg_edges);
- The hash table used to map the set of control-dependence to the corresponding region node (RegionNodeHashMap *hash_table_region_nodes);
- The underlying CFG form. The PDG generation should avoid changing the original CFG by all means (Cfg PDG::*cfg);
- The Dominance Tree. Here the existing library in Machine SUIF is used (DominanceInfo PDG::*d; NatSetDense *_pdom_immd_children);

A.2 PDG Node Class

The control-dependence edges are not explicitly maintained in the PDG class. Rather, the CD edges are implicitly included as parent-child links of the PDG node. The class of `Pdg_node` is responsible for PDG nodes representation. It has three important properties,

- The parents of the PDG node (suif_list<Pdg_node*> _parents);
- The children of the PDG node (suif_list<Pdg_node*> _children);
- The set of control dependence of the PDG node (CDset _cd_set);

As for the four types of PDG nodes, i.e. *region*, *entry*, *predicate* and *statement*, a derived class of the PDG node class for each of them is defined. Properties and methods specific to that particular kind of node are contained in the sub-class.

To facilitate the *region* node insertion in the step 6 of the CDG construction (see Table 3.1), the *entry* node is especially treated as a combination of *predicate* node and *region* node to some extent (i.e. conceptually in design context but not really in theory). That is to say, the *entry* node behaves as control dependence predecessor (like *predicate* nodes) but has no TRUE/FALSE labelling on the edge and it could have multiple children (unlike *predicate* nodes). The object-design of the nodes is,

- class Pdg_node_stmt: public Pdg_node
- class Pdg_node_predicate: public Pdg_node{
  Pdg_node_region* true_child;
  Pdg_node_region* false_child;
  }
- class Pdg_node_region: public Pdg_node

Herein, the `true_child` and `false_child` implicitly indicates the control dependence edges in the CDG.

The *region* node does not include any underlying CFG node as its property (the *cfg property inherited from the parent node is set to `NULL` for *region* node). While other pdg nodes are simply labelled with underlying CFG node number, another property field is added for *region* node in the derived class to number it.

A.3 DDG Edge Class

The `Ddg_edge` class is a helper class for PDG class to store the data dependence information between CFG nodes explicitly. It has following properties,

- The source and destination nodes are the two end nodes of the directed edge (CfgNode *source, *destination);
- The weight of the edge (int weight);

The field of weight is an integer representing the edge *weight* of the data dependence between the two nodes, as it was defined in section 3.2.3.

To simplify the DDG graph, only one edge between any two nodes is needed with an amounted *weight* property. So in the PDG class, all the `Ddg_edge` instances are arranged in a C++ *set* container. C++ *set* container asks for a "less-than comparison" function to perform internal ordering and achieve item uniqueness storing. Therefore both "equality" and "less-than" comparison methods are defined for `Ddg_edge` class explicitly based upon the *weight* property of the DDG edge.

A.4 CDset Class

The `CDset` means a bunch of control dependence (may be just one though), so the `CDset` class is responsible for summarizing the control dependence for a specific node. For example, if node 2 is control-dependent on node 1's true edge, the CD is denoted as <1T>. A CD set is thus a sequence of the CD, like <1T, 3F, 7F…> and so on. Its properties include,

- A flag indicating whether the set is empty or not (bool CDset::_is_empty);
- A flag indicating whether the edge originates from entry node or not (bool CDset::_entry);
- A Natural Set containing control dependence on a node when the branch evaluates to be true (NatSetSparse CDset::_true_CD);
- A Natural Set containing control dependence on a node when the branch evaluates to be false (NatSetSparse CDset::_false_CD).

The `_entry` Boolean tells whether the CD set includes the *entry* node or not. It is included because the entry node is treated in a special way in design as described above. When it's true, it means the `CDset` includes *entry* node, and otherwise false.

During the *region* node insertion, the `CDset` objects are stored in a hash table. So it is necessary to provide methods for comparing the object equality and calculating the hash value.

- Comparison methods (bool CDset::operator== (const CDset &cd_set) const);
- Hash function (size_t suif_hash(const CDset s)).

APPENDIX B – IMPLEMENTATION OF PDG PASS

In this section, particular implementation issues in writing the PDG pass are explained. The core algorithm, as introduced in the section 3.3, consists of PDT generation, CDG generation and DDG generation. But before they are presented in detail, the auxiliary classes, data structures and functions are explained first.

By design, the PDG nodes and DDG edges are the two most important classes that should be implemented. The CDG edges are implicit as part of the PDG, represented by the nodes' parent-child relationship so it is not implemented separately. An important property of PDG node is its control dependence set. Thus a `CDset` class is defined to represent it, and a hash table in the PDG node to store the existing `CDset` instances.

```
typedef suif_hash_map<CDset, Pdg_node_region*> RegionNodeHashMap;
RegionNodeHashMap Pdg:*hash_table_region_nodes;
```

The true dependencies and false dependencies are kept individually in two `NatSetSparse` objects. In essence, `NatSetSparse` collects a set of natural numbers, like 1, 2, 3, etc. The hash function implemented for `CDset` class is based on those numbers. The equality comparison is tested against the two sets as well. Both sets should be equal, together with the `_entry` Boolean, to satisfy the equality.

```
CDset& CDset::operator=(const CDset& other){
    if(this!=&other){
        _true_CD = other._true_CD;
        _false_CD = other._false_CD;
        _is_empty = other._is_empty;
        _entry = other._entry;
    }
    return *this;
}
```

Now the implementation of the three core methods in generating the PDG is gone through. First of all, it is to generate the Post-Dominance Tree of the input CFG. As PDT is directly relevant to CFG rather than PDG, tree information in the `Pdg_node` class does not need not to be stored. But instead, the tree structure in an array of *Natural Number Set* (`NatSetDense` provided by Machine SUIF) is kept in the top-level `Pdg` class. Each `NatSetDense` in the array holds children's index numbers of the parent node. Also in Machine SUIF CFA [60] library, OPI provides a class called `DominaceInfo` to capture the dominators, post-dominators, dominance frontier, and post-dominance frontier of a CFG. So in method generate_`PDT`, each node's immediate post dominator is found through iteration, and the parent-child link is put in the particular `NatSetDense` array. That is to say, each of the CFG node's PDT children is stored by visiting each one's PDT parent.

```
for(h = nodes_start(cfg); h!= nodes_end(cfg);h++){
    v = get_node(cfg,h);
    if (d->immediate_postdominator(v)){
        immediate_postdominator.insert(get_number(v));
    }
}
```

The next step is to generate the CDG based on the PDT in method `generate_CDG`. The algorithm introduced in section 3.3. The set of CFG edges S that destination node does not post-dominate source node is identified firstly. And then for every edge in the set S, one edge is visited at a time. If the edge is A->B, the method traverse backward from B in the PDT until it reaches A or A's

parent. All the nodes met during the traversal are marked as control dependent on A.

The tricky implementation lies in *region* node insertion. The basic function of *region* node is to summarize the control dependences and after insertion, the *predicate* nodes would have only two children, *true* and *false* respectively. And the PDG is organized hierarchically. Ferrante et al. describes the algorithms as a two-phase process [50]. The first pass is based on the post-order traversal of the PDT to insert any necessary *region* nodes. The second pass is a check on the output of the first pass to ensure that any *predicate* node has only two children.

For the post-order tree traversal, a separate class method is implemented to do the work. The method is recursively called on each `CfgNode` (starting from the exit node).

```
void Pdg::insert_region_postorder(CfgNode *cfg_node, NatSetDense
*flags);
```

Upon each call, the `CDset` of the visited node is checked in the hash table to see if any *region* node already exists. If so, the *region* node and the visited node are simply linked up. If not, a new region node is created; the visited node and the newly created *region* node are linked up, and the region node is put into the hash table. Next compute the intersection INT of CD, i.e. check if the set of control dependences for each immediate child of the visited node in the PDT overlaps or not. If the intersection INT equals CD, then the corresponding dependences are deleted from the child and replaced with a single dependence on the *region* node. If every control-dependence of the child is in the intersection INT, then the corresponding dependences are deleted and replaced with a single dependence edge on the child's control predecessor. The second pass of the region node insertion works on the *predicate* node P in the CDG having multiple control dependence successors with the same associated label L. For each P, a region node R is created. Each node in the graph that had control dependence predecessor P with the label L is made to have the single control dependence predecessor R. Finally, R is made to be the single control dependence graph successor of P with the same label L.

Finally for DDG generation method, the work is straightforward based on the Machine SUIF SSA library. After the CFG is transformed to SSA form, each BB in the procedure is visited. Upon each visit, for each definition that instruction and phi-nodes defines, mark the corresponding uses along the def-use chain. The C++ code of doing so is given below,

```
for (int i = 0; i < nodes_size;i++){
       CfgNode *node_block = get_node(i);
       MarkDefUseFilter mark_def_use;
       for(InstrHandle h =
           start(node_block); h != end(node_block);h++)
           {
               Instruction *instr = *h;
               map_opnds(instr,mark_def_use);
           }
       phi_node_list = get_phi_nodes();
       for(PhiHandle phid =
       phi_node_list.begin(); phid != phi_node_list.end();phid++)
       {
           PhiNode *phinode = *phid;
           map_opnds(phinode,mark_def_use);
       }
```

The `map_opnds` is an OPI function. In the `MarkDefUseFilter` class, the function `operator()` is overridden to do the edge insertion. The filter looks at each operand of the instruction, refers to the uses of each operand (by calling the get_def_use_chain method provided by SSA library). If the uses are in different BB from the defining instructions / phi-nodes, mark the def-use chain as a candidate DDG edge. The edges are inserted if they do not already exist between the two BBs. While if not, the candidate edge is not inserted, rather, the dependence weight is incremented by 1 to avoid multiple DDG edges between any two nodes.

BIBLIOGRAPHY

[1] D. Clark, "The design philosophy of the DARPA internet protocols," Symposium proceedings on Communications architectures and protocols, Stanford, California, United States: ACM, 1988, pp. 106-114.

[2] J. Rolia, R. Friedrich, and C. Patel, "Service centric computing - Next generation Internet computing," Performance Evaluation Of Complex Systems: Techniques And Tools -, vol. 2459, 2002, pp. 463-479.

[3] Intel Corp. "Intel IXP 2805 Network Processor, Hardware Reference Manual", April 2006.

[4] R. Leupers, L. Thiele, N. Xiaoning, B. Kienhuis, M. Weiss, and T. Isshiki, "Cool MPSoC programming," Design, Automation & Test in Europe Conference & Exhibition (DATE), 2010, pp. 1488-1493.

[5] R. Tucker, "A green internet," in IEEE Lasers and Electro-Optics Society, 2008. LEOS 2008. 21st Annual Meeting of the, pp. 4-5, 2008.

[6] S. Roy, S. Katkoori, and N. Ranganathan, "A compiler based leakage reduction technique by power-gating functional units in embedded microprocessors," in 20th International Conference on VLSI Design, 2007. Held jointly with 6th International Conference on Embedded Systems., pp. 215–220, 2007.

[7] A. Weiss, "Computing in the clouds," netWorker, vol. 11, 2007, pp. 16-25.

[8] A. Feldmann, "Internet clean-slate design: what and why?," SIGCOMM Comput. Commun. Rev., vol. 37, 2007, pp. 59-64.

[9] C. Rosewarne, “Network Processors Evaluating Architectures for Leading Edge Applications,” Calyptech White Paper Issue: 01, March 2004.

[10] N. Shah, “Understanding Network Processors,” 2001.

[11] A.S. Tanenbaum, Computer networks, Prentice Hall PTR, 2003.

[12] R. Braden, “Requirements for Internet Hosts - Communication Layers,” Oct. 1989.

[13] Cisco Systems. “Parallel eXpress Forwarding in the Cisco 10000 Edge Service Router.” White Paper. October 2000.

[14] Alchemy Semiconductor, Inc. “The Alchemy Au1000 Internet Edge Processor.” Product brief. 2000.

[15] EZchip Technologies. “EZchip Technologies Software Development Suite Now Available For Its 10-Gigabit 7-Layer Network Processor.” Press Release. January 17, 2001.

[16] P.G. Paulin, F. Karim, and P. Bromley, “Network Processors: A Perspective on Market Requirements, Processor Architectures and Embedded S/W Tools,” Design, Automation and Test in Europe Conference and Exhibition, Los Alamitos, CA, USA: IEEE Computer Society, 2001, p. 0420.

[17] K. Asanovic et al., “A view of the parallel computing landscape,” Commun. ACM, vol. 52, 2009, pp. 56-67.

[18] NetLogic Microsystems. “NetLogic Microsystems announces breakthrough multi-core processor solution which integrates 128 NXCPUs™.” Press Release. July 2010.

[19] J. Ceng et al., “MAPS: an integrated framework for MPSoC application parallelization,” Proceedings of the 45th annual Design Automation Conference, Anaheim, California: ACM, 2008, pp. 754-759.

[20] Stanford SUIF Compiler Infrastructure, “The SUIF 2 Compiler System”, Stanford University, Accessed May. 2011; http://suif.stanford.edu/suif/suif2/index.html.

[21] J. Wagner and R. Leupers, "C compiler design for a network processor," Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on, vol. 20, 2001, pp. 1302-1308.

[22] M. Budiu et al., "BitValue Inference: Detecting and Exploiting Narrow Bitwidth Computations," Lecture Notes in Computer Science, 2001, pp. 969-979.

[23] J. Wagner and R. Leupers, "Advanced Code Generation for Network Processors with Bit Packet Addressing," Proceedings of the 1st Workshop on Network Processors, 2002, pp. 91-115.

[24] M.K. Chen et al., "Shangri-La: achieving high performance from compiled network applications while enabling ease of programming," Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation, Chicago, IL, USA: ACM, 2005, pp. 224-236.

[25] H. Vin, J. Mudigonda, and J. JASON, "A Programming Environment for Packet-Processing Systems: Design Considerations," 3nd Workshop on Network Processors (NP-3), 10th Intl Symposiun on High Performance Computing Architectures (HPCA-10), 2004.

[26] J. Roy, C. Sun, and C.Y. Wu, Open Research Compiler for Itanium Processor Family (IPF)[A]. MICRO-34 Tutorial [C], Texas, USA: ACM Press, 2001.

[27] L Shi, Y Zhang, J Yu, B Xu, B Liu and J Li, "On the Extreme Parallelism Inside Next-Generation Network Processors", INFOCOM 2007. 26th IEEE International Conference on Computer Communications. IEEE, May 2007, Anchorage, Alaska, USA, pp. 1379-1387.

[28] Intel Corp. "Intel IXP 2400 Network Processor: Flexible, High-Performance Solution for Access and Edge Applications", white paper, Jan. 2003.

[29] Jia Yu, "Architectural and Compiler Optimization for Network Processors", Ph.D. Thesis, UC riverside, September 2007.

[30] V. Ramamurthi et al., "System level methodology for programming CMP based multi-threaded network processor architectures," VLSI, 2005.

Proceedings. IEEE Computer Society Annual Symposium on, 2005, pp. 110-116.

[31] N. Weng and T. Wolf, "Profiling and mapping of parallel workloads on network processors," Proceedings of the 2005 ACM symposium on Applied computing, Santa Fe, New Mexico: ACM, 2005, pp. 890-896.

[32] R. Ennals, R. Sharp, and A. Mycroft, "Linear types for packet processing," Lecture notes in computer science, 2004, pp. 204-218

[33] W. Plishker, "Automated Mapping of Domain Specific Languages to Application Specific Multiprocessors," Oct. 2006.

[34] E. Kohler et al., "The click modular router," ACM Trans. Comput. Syst., vol. 18, 2000, pp. 263-297.

[35] Intel Corp., "Intel Microengine C Compiler Language Support: Reference Manual," Nov. 2003.

[36] G. Memik and W.H. Mangione-Smith, "NEPAL: A Framework for Efficiently Structuring Applications for Network Processors," 2nd Workshop on Network Processors (NP-2) at the 9th International Symposium on High Performance Computer Architecture (HPCA-9), Anaheim, CA, 2003, pp. 203-226.

[37] K. Crozier, "A C-Based Programming Language for Multiprocessor Network SoC Architectures," Network Processor Design: Issues and Practices, Morgan Kaufmann, 2003, pp. 427-443.

[38] H. Yang et al., "Power and Energy Impact by Loop Transformations," Workshop on Compilers and Operating Systems for Low Power 2001, Parallel Architecture and Compilation Techniques, 2001.

[39] D. Burger and T. Austin, The simplescalar toolset, Version 2.0, Computer Sciences Dept, University of Wisconsin, 1997.

[40] B. Li and R. Gupta, "Simple offset assignment in presence of subword data," Proceedings of the 2003 international conference on Compilers, architecture and synthesis for embedded systems, San Jose, California, USA: ACM, 2003, pp. 12-23.

[41] F. Li et al., “Profile-driven energy reduction in network-on-chips,” Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation, San Diego, California, USA: ACM, 2007, pp. 394-404.

[42] A. Bona et al., “Energy Estimation and Optimization of Embedded VLIW Processors Based on Instruction Clustering,” Proceedings of the 39th Design Automation Conference DAC’02, 2002, p. 886–891.

[43] H. Yun and J. Kim, “Power-aware modulo scheduling for high-performance VLIW processors,” Proceedings of the 2001 international symposium on Low power electronics and design, Huntington Beach, California, United States: ACM, 2001, pp. 40-45.

[44] C. Lee et al., “Compiler optimization on VLIW instruction scheduling for low power,” ACM Trans. Des. Autom. Electron. Syst., vol. 8, 2003, pp. 252-268.

[45] W. Zhang et al., “Leakage-aware compilation for VLIW architectures,” Computers and Digital Techniques, IEE Proceedings-, vol. 152, 2005, pp. 251-260.

[46] Jing Huang, Xiaojun Wang and Bin Liu, “Energy-aware Compilation for Network Processors: Frameworks, Techniques and Trend”, China-Ireland International Conference on Information and Communication Technologies, 26th-28th Sep., 2008.

[47] Q. Wu and T. Wolf, “On runtime management in multi-core packet processing systems,” Proceedings of the 4th ACM/IEEE Symposium on Architectures for Networking and Communications Systems, San Jose, California: ACM, 2008, pp. 69-78.

[48] R. Ramaswamy, N. Weng, and T. Wolf, “Application analysis and resource mapping for heterogeneous network processor architectures,” Network Processor Design: Issues and Practices, vol. 3, 2005, pp. 277–306.

[49] J. Yao, Y. Luo, L. Bhuyan, and R. Iyer, “Optimal network processor topologies for efficient packet processing,” IEEE Global Telecommunications Conference, 2005. GLOBECOM'05.

[50] X. Huang and T. Wolf, “Evaluating Dynamic Task Mapping in Network Processor Runtime Systems,” IEEE Transactions on Parallel and Distributed Systems, vol. 19, 2008, pp. 1086-1098.

[51] J. Ferrante, K.J. Ottenstein, and J.D. Warren, “The program dependence graph and its use in optimization,” ACM Trans. Program. Lang. Syst., vol. 9, 1987, pp. 319-349.

[52] J. Yu, J. Yao, L. Bhuyan, and J. Yang, “Program mapping onto network processors by recursive bipartitioning and refining,” Proceedings of the 44th annual conference on Design automation, San Diego, California: ACM, 2007, pp. 805-810.

[53] M.D. Smith and G. Holloway, “An introduction to Machine SUIF and its portable libraries for analysis and optimization,” Division of Engineering and Applied Sciences, Harvard University, 2002.

[54] R. Cytron, J. Ferrante, B.K. Rosen, M.N. Wegman, and F.K. Zadeck, “An efficient method of computing static single assignment form,” Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, Austin, Texas, United States: ACM, 1989, pp. 25-35.

[55] G. Holloway, “The Machine-SUIF Static Single Assignment Library,” Division of Engineering and Applied Sciences, Harvard University, 2002. http://www.eecs.harvard.edu/hube/software/nci/ssa.pdf.

[56] J. Ellson, E. R. Gansner, E. Koutsofios, S. C. North, and G. Woodhull, “Graphviz and dynagraph–static and dynamic graph drawing tools,” Graph Drawing Software, pp. 127–148, 2003.

[57] R. Ramaswamy and T. Wolf, “PacketBench: a tool for workload characterization of network processing,” Workload Characterization, 2003. WWC-6. 2003 IEEE International Workshop on, 2003, pp. 42-50.

[58] W. Gong, G. Wang, and R. Kastner, “A High Performance Application Representation for Reconfigurable Systems,” Intl. Conf. on Engineering of Reconfigurable Systems and Algorithms (ERSA), Las Vegas, NEV, USA, 2004.

[59] J. Fenwick and L. Pollock, Implementing an optimizing linda compiler using suif, 1996.

[60] http://suif.stanford.edu/suif/suif2/doc-2.2.0-4

[61] Jing Huang, Xiaojun Wang, “Program Dependence Graph Generation and Its Use in Network Application Analysis”, CIICT 2009, Aug. 2009.

[62] Jing Huang, Xiaojun Wang, “Program Dependence Graph Generator in Machine SUIF”, Internal Technical Report, May 2009.

[63] G. Xia, B. Liu, "Accelerating network applications on X86-64 platforms," IEEE Symposium on Computers and Communications (ISCC), June 2010.

[64] J. Meng, X. Chen, Z. Chen, C. Lin, B. Mu, and L. Ruan, “Towards High-Performance IPsec on Cavium OCTEON Platform,” in Trusted Systems, vol. 6802, 2011.

[65] T. R. Halfhill, “Netlogic broadens XLP family,” Microprocessor Rep., vol. 24 , 2010.

[66] D. Yin, D. Unnikrishnan, Y. Liao, L. Gao, and R. Tessier, “Customizing virtual networks with partial FPGA reconfiguration,” ACM SIGCOMM workshop on Virtualized infrastructure systems and architectures (VISA), 2010.

[67] C. A. R. Hoare, “Communicating sequential processes,” Commun. ACM 26, 1, 100-106. Jan. 1983.

[68] P. Chang, I. Wu, J. J. Shann and C. Chung, "ETAHM: An energy-aware task allocation algorithm for heterogeneous multiprocessor," Design Automation Conference (DAC), June 2008.

[69] H. H. Yang and D. F. Wong, "Balanced partitioning," Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on, Dec 1996.

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

[71] H. Yang and D. F. Wong, “Efficient network flow based min-cut balanced partitioning,” *IEEE/ACM international conference on Computer-aided design*, 1994.

[72] R. Mishra, N. Rastogi, D. Zhu, D. Mosse and R. Melhem;, "Energy aware scheduling for distributed real-time systems," Parallel and Distributed Processing Symposium, Proceedings. International, Apr. 2003.

[73] Y. Zhang, K. Ootsu, T. Yokota and T. Baba, "Automatic Thread Decomposition for Pipelined Multithreading," Parallel and Distributed Systems (ICPADS), IEEE 16th International Conference on, Dec. 2010.

[74] J. Dai, B. Huang, L. Li, and L. Harrison, “Automatically partitioning packet processing applications for pipelined architectures,” ACM SIGPLAN Conference on Programming Language Design and Implementation, (PLDI), June 2005.

[75] A. Mallik, Y. Zhang, and G. Memik, “Automated task distribution in multicore network processors using statistical analysis,” in Proceedings of the 3rd ACM/IEEE Symposium on Architecture for networking and communications systems - ANCS, 2007.

[76] L. Li, B. Huang, J. Dai, and L. Harrison, “Automatic multithreading and multiprocessing of C programs for IXP,” in Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming - PPoPP, 2005.

[77] J. Kuang and L. Bhuyan, “Optimizing Throughput and Latency under Given Power Budget for Network Packet Processing,” IEEE Conference on Computer Communications, 2010.

[78] Monika Lam, “An Overview of the SUIF2 System”, ACM Conference on Programming Language Design and Implementation (SIGPLAN), 1999.

[79] Jing Huang, Olga Ormond, Di Ma and Xiaojun Wang, “Optimizing Energy-Efficiency for Program Partitioning and Mapping onto Multi-Core Packet Processing Systems,” the Journal of China University of Posts and Telecommunications, June 2012, 19(Suppl. 1), pp. 79-86.

[80] L. Bhuyan, Y. Luo, J. Yang and L. Zhao, "NePSim: A Network Processor Simulator with Power Evaluation Framework”. Sept/Oct 2004.

[81] T.M Austin, T Mudge, Sim-Panalyzer: the simplescalar-arm power modeling project, http://web.eecs.umich.edu/~panalyzer/

[82] Wu and T. Wolf, “Dynamic workload profiling and task allocation in packet processing systems,” in International Conference on High Performance Switching and Routing, 2008. HSPR 2008, pp. 123–130, 2008.

[83] J. Huang, A. Raman, T. B. Jablin, Y. Zhang, T. H. Hung, and D. I. August, “Decoupled software pipelining creates parallelization opportunities,” in Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization, pp. 121–130, 2010.

[84] N. Weng and T. Wolf, “Pipelining vs. Multiprocessors - Choosing the Right Network Processor System Topology,” in in Proc. of Advanced Networking and Communications Hardware Workshop (ANCHOR), 2004.

[85] Intel IXP2XXX Product Line of Network Processors, Intel Corporation.

[86] M. Kandemir, O. Ozturk, and V. S. R. Degalahal, “Enhancing Locality in Two-Dimensional Space through Integrated Computation and Data Mappings,” in 20th International Conference on VLSI Design, 2007. Held jointly with 6th International Conference on Embedded Systems., pp. 227–232, 2007.

[87] M. I. Gordon, W. Thies, and S. Amarasinghe, “Exploiting coarse-grained task, data, and pipeline parallelism in stream programs,” in Proceedings of the 12th international conference on Architectural support for programming languages and operating systems, pp. 151-162, 2006.

[88] I. Kadayif, M. Kandemir, G. Chen, N. Vijaykrishnan, M. J. Irwin, and A. Sivasubramaniam, "Compiler-directed high-level energy estimation and optimization," ACM Transactions on Embedded Computing Systems (TECS), vol. 4, no. 4, pp. 819-850, 2005.