Université de Montréal

The Co-Design Methodologies On Click Router Application System

Présenté par:
Dan Li

Département d`informatique et de recherche opérationnelle
Faculté des arts et des sciences

Mémoire présenté à la Faculté des études supérieures
en vue de l`obtention du grade de
Maître en Informatique

Décembre, 2004

Université de Montréal
Faculté des études supérieures

Ce mémoire intitulé :

The Co-Design Methodologies on Click Router Application System

Présenté par:

Dan Li

a éte évalué par un jury composé des personnes suivantes:

Jean Pierre David	Président-rapporteur
El Mostapha Aboulhamid	Directeur de recherche
Gabriela Nicolescu	Co-directeur
Claude Frasson	Membre du jury

Mémoire accepté 7 juin 2005

Résumé

Afin d'obtenir des meilleurs performances pour les applications actuels de plus en plus complexes, aujourd'hui le domaine de co-conception logiciel/matériel a pris une grande ampleur. En même temps, plusieurs architectures multiprocesseurs efficaces ont été proposées récemment et elles sont devenue la solution la plus populaire dans le domaine des systèmes embarques logiciels/matériels.

Des méthodologies innovateurs de conception sont nécessaires actuellement pour respecter les contraintes fortes de coût, performance et temps de mise sur le marché. Un des points clé de ces méthodologies et un des défies actuelles des concepteurs est donné par l'étape de partitionnement logiciel/matériel en partant d'une spécification de haut niveau pour l'application à implanter.

Dans ce contexte, les contributions de notre travail de recherche sont :

(1) La proposition d'une méthodologie de partitionnement logiciel/matériel. Nous avons utilisé comme application le routeur IPV4 Click fourni par l'Université Berkeley et nous avons analyse l'impact de la méthodologie sur les performances de ce système.

(2) Le développement et l'évaluation d'une architecture multiprocesseur intégrant 3 processeurs, dans le cas du routeur IPV4 à l'aide de l'outil StepNP de la compagnie STMicroelectronics. Cette approche peut être utilisée pour la conception de nouvelles architectures, intégrant plus de 3 processeurs.

Notre évaluation montre un gain en performances dans le cas des architectures logicielles/matériels (notre systeme a été deux fois plus rapide quand une partie de fonctions logiciels ont été implantées en matériel). L'architecture multiprocesseur présente une vitesse de traitement des paquets deux fois plus grande qu'une architecture monoprocesseur.

Keyword : Conception logiciel/matériel, partitionnement, systèmes multiprocesseur, routeur Click, StepNP, SystemC

Abstract

To improve the overall performance of the current increasingly complex applications, the combination of hardware and software solutions must be valued in co-design area. In addition, multiple processors architectures has been introduced as a generic model to design many specific application and they became the most efficient and popular solution in the field of hardware/software systems.

In order to respect the straight constraints of cost, performances and time-to-market imposed for multiprocessor embedded systems designers, a mew generation of methodologies is required. One of the key points and current important challenge in such methodologies is to perform hardware/software partitioning starting from a high level specification of the application to be implemented.

In this context, the contributions of our work are:

(1) Proposing a methodology for hardware/software partitioning. We use the Click IPv4 software router (provided by Berkeley University) as an application and analyze the methodology impact on the packets processing performance of Click IPv4 router.

(2) Developing and evaluating an architecture implementing Click IPV4 software router on three ARM CPUs with the help of StepNP platform provided by STMcirolectronics Company. This approach may be extended to implementing Click software on more than three multi-processors. As a result, the speedup of IPv4 Click router application system almost doubles in the help of combination of hardware/software co-design comparing with using only software implementation on Click router. In multiprocessors architecture for Click router integrating three processors, the overall speed to process a packet is about 1.84 faster than that of the Click router single processor architecture.

Keyword : Hardware/Software co-design, Partitioning, Multiprocessors, Click router, StepNP, SystemC

Table of Contents

List of Figures

List of Tables

Acknowledgement

This thesis work has been a challenge in my whole study period. Throughout this period, I received much support and help from the professors and friends.

Firstly, I would like to appreciate Prof. El Mostapha Aboulhamid, my director, for his continued guidance, his encouragement, his care and his support through the whole course of this thesis work. He has constantly given much precious advice on the thesis work by bringing up relevant accomplishment in this research area. I also owe my gratitude to Prof. Gabriela Nicolescu, my co-director, for her continued support, her encouragement, her thoughtful advice and her meticulous work for this thesis work.

Great appreciation is also given to David Quinn, and Mortimer Hubin, who have given me generous help on this thesis work. Based on their work on the hardware/software co-design of Click router application system, I can continue to complete this thesis work. Especially I received a lot of benefit from the discussion with David Quinn, who gave his precious time and invaluable effort for this thesis work.

Many thanks to all of my friends Qin Lisheng and Zhang Hong for their encouragement, and their valuable comments on my thesis report.

Finally, I would like to give a special acknowledgement to my sister, Li Wei, for her consistently patience, faith on me. I also would like to give such acknowledgement to my parents for their understanding, and making all of this come true.

Chapter 1

Introduction

1.1 Context and motivations

Due to the popularity of the Internet, the traffic on the Internet increases rapidly for the last ten to fifteen years. The performance of router, a core part of network, is considered to be an important factor to affect the overall performance of the whole network. Typically, a router accepts data-grams and relays them on toward their destination and is responsible for determining the route[13].

Ever-increasing requirements from network applications, however, drive routers to supply more new network services such as packet tagging, application-level proxies, applications-specific packet dropping, performance monitoring, intrusion detections, and various filter and firewalls rapidly[4]. While most backbone routers improve their forwarding process of packets using applications-specific integrated circuits (ASIC's), software based edged routers[2][5] become attractive due to high-performance network processors.

Thus, modular components routers architecture take advantage over that of conventional routers on flexibility and extension. Such a typical system is MIT's Click router[1] which is a modular architecture built from different software modular components(elements). Component-based routers make software networking easier to program.

However, component techniques, which split a complicated, big computation task into some small, simple computation units(components) and build the linkage to pass the messages(changes) between component possessing reuse and flexibility, suffer inefficiencies that monolithic software can avoid [11]. One way to avoid modularity overhead is to consider applying hardware/software co-design methodology to increase the computation in a component and to offset the cost of the communications among components. This methodology is implemented on StepNP[6] platform that we use in our work .

1.2 Objective

In the presented context, the main objectives of our work are:

1) Bring a new solution for the application partitioning onto hardware/software architecture using high-level simulation in co-design.

2) Exploring how to implement the Click router application system (see **Chapter 4.3**) on multiprocessors and evaluating the performance of such an architecture.

1.3 Contribution

In order to reach our objective the proposed contributions are:

1) Proposing a methodology for applications partitioning. We use the Click IPv4 software router as an application and analyze the methodology impact on the packets processing performance of Click IPv4 router.

2) Developing an executable application-specific architecture (see **Chapter 5.1.1**) of implementing Click software router on three ARM CPUs with the help of StepNP platform and giving the evaluation of the method. This approach may be extended to implementing Click software on more than three multi-processors.

1.4 Thesis structure

In the next chapters of the thesis, we introduce basic knowledge of HW/SW codesign in **Chapter 2** and multiprocessors in **Chapter 3.** Then, we present a co-design methodology on how to partition Click router into software parts and hardware parts, and on how to implement those parts respectively on StepNP platform (**Chapter 4**), and a methodology implementing Click on three processors in **Chapter 5**. **Chapter 6** will discuss the evaluation of those two methodologies by simulation means. In the final section, we will discuss future work as well.

Chapter 2

Hardware/Software Co-Design

2.1 Definition & motivation for hardware/software co-design

Hardware/software co-design refers to the collaborative design of hardware and software, considering the hardware and software concurrently to design computing systems that meet all performance requirements and minimize the amount of hardware resources[14]. Two main considerations stimulate the need for hardware/software co-design in digital systems design.

From the hardware aspect, the fact that dedicated computing systems need to react external input instantly and to deal with more complicated computation of digital systems raises higher demands on the performance to today's digital systems. With the help of ASICs processing data in parallel with processors, hardware solution may shorten execution time, improving the overall performance of digital systems. However, ASICs often are not high-volume produced. They are often designed to meet the performance need for the application systems. Thus, hardware solution may be more expensive than software solution [15] due to the development cost of ASICs.

From the software aspect, software is likely reused on various processors and modified more easily. Such possibility may reduce design efforts and shorten time-to-market. Therefore, while considering cost constraints, designers may values the re-programmability and flexibility of software solutions. However, software often performs tasks in serial order and may not complete some operations while respecting time constrains[15].

In summary, both software and hardware components are interesting for complex systems implementations. Currently, all systems include both types of components, and the design of these types of systems requires new methodologies to reduce the design efforts and to shorten the time-to-market.

2.2 The Hardware/software co-design flow

The hardware/software co-design process may contain creating hardware/software modeling, validating to meet the desired design, partitioning and implementing software (through compilation) and physical hardware(through synthesis). The overall design flow is depicted in **Figure 2-1**.

The first step in HW/SW co-design is system-level specification, defining the functionality of the system at high abstract level. Then, the process covers some major steps such as HW/SW partitioning stage, which will be discussed in more detail in section 2.4.

The next specification refinement step is actually what we call modeling, putting abstract specification into program implementation. Software may be coded in C language and hardware may be described in VHDL language. The interface between hardware and software is built at this step. System validation may be done at this phase, using formal verification, simulation and emulation methods. Formal verification is the techniques allowing designers to proof the properties of their design [14, 22]. This is done using mathematical and logical equations. Simulation refers to check whether the functions of designs match up the initial specification by limited input data. By loading hardware units into programmable hardware (e.g. FPGA) and simulating in the more practical testing environment, emulation helps to provide more practical prototypes, making system designs closer to final products.

After co-synthesis, a set of ASICs are synthesized and software are compiled into the executable codes targeted at specific processors. In fact, validation can be used in any abstract levels in design process. Partitioning may be done in a design process to fulfill performance and cost constrains.

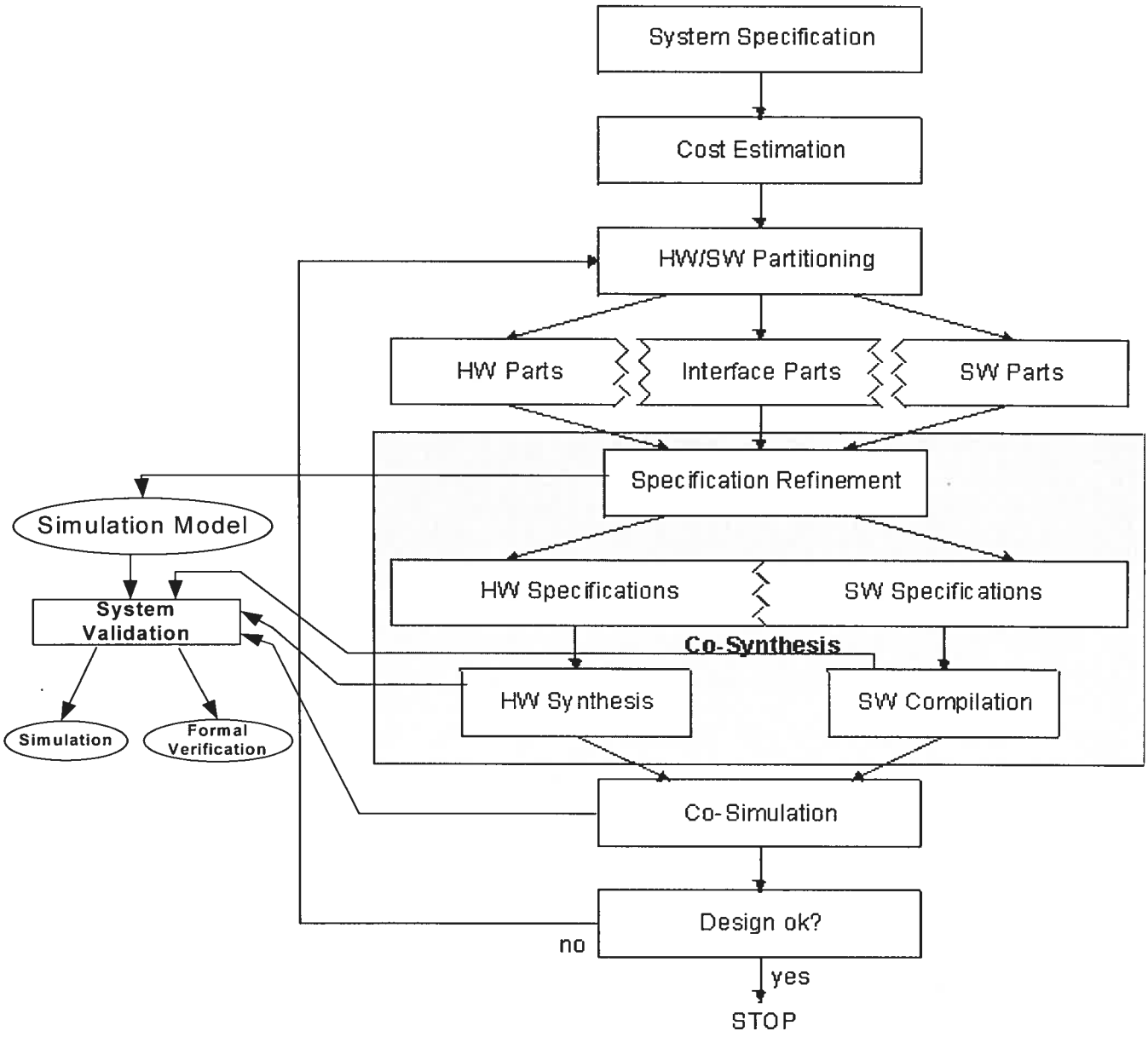


Figure 2-1 Hardware/software co-design framework[14, 22]

2.3 Target architectures

Target system architectures[27] may be classified into two categories:

Single-Processor architectures with one or more than one ASIC and Multi-Processors Architectures [14].

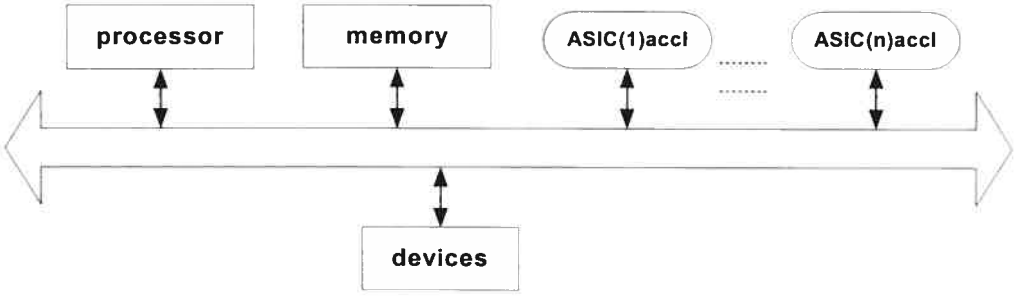


Figure 2-2 a single-processor architecture [14]

The overall time of a program executed on a single-Processor may be shortened by adding ASICs to the system (called also coprocessors) [14]. The coprocessors are dedicated to implement some particular functions in hardware in parallel with the software components running in the processor. Some Co-design tools (e.g. Cosyma, Mickey, and Vulcan [14]) support such architectures. **Figure 2-2** describes a typical frame concerning such target architectures.

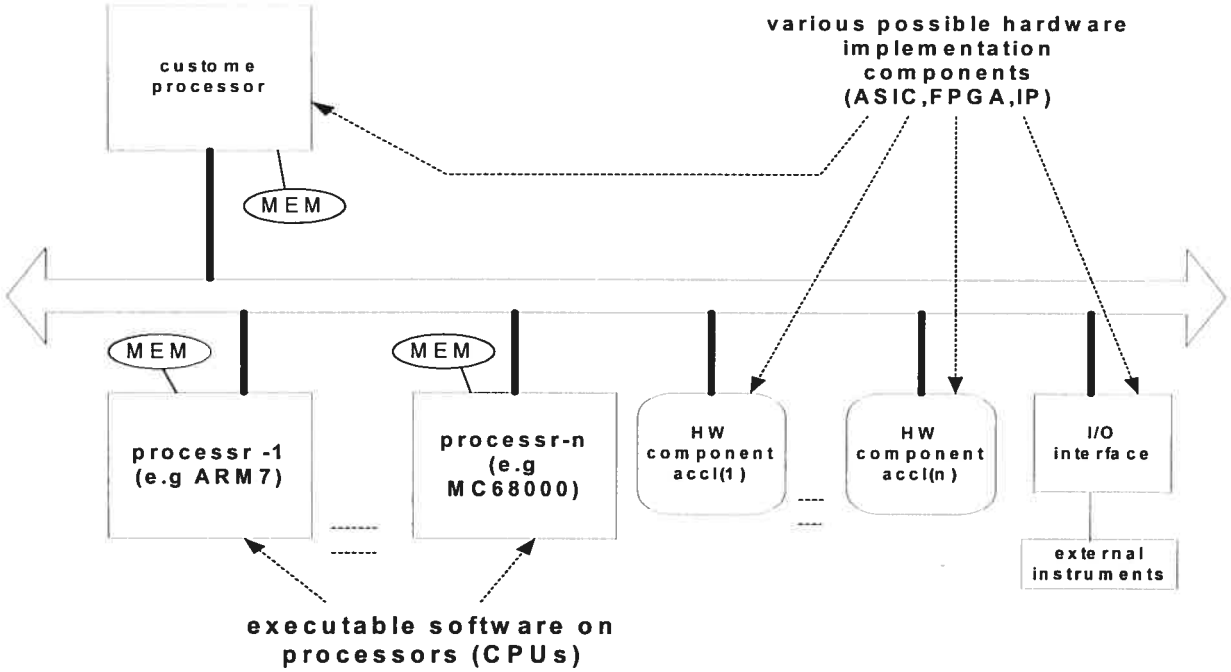


Figure 2-3 a typical multiprocessor architecture [24]

Multiprocessor architectures include multiple processors and several ASICs (or coprocessors) to improve the overall performance of the application programs running on such architectures. Some Co-design tools (e.g StepNP [6] and CoWare [14]) provide a way to build the interface between hardware and software and to create a simulation model, supporting the Co-design on Multi-Processor architectures. Designers are also able to select different kinds of processors (e.g. ARM 7 and MC68000 processors) and to build heterogeneous platforms.

2.4 Partitioning: definition and problems

As shown in previous section, the partitioning plays a crucial role for a successful hardware/software co-design. Based on mainly on the work presented in [14], we introduce in this section the principal partitioning problems. Hardware/software partitioning refers to dividing the application system into software components and hardware units, and bridging the software components and hardware units according to the system specification[14].

2.4.1 Recent problems on hardware/software co-design

While looking at the co-design, we observe several relevant problems remained in software and hardware.

- embedded system
- partitioning
- software compiler
- software modularity
- parallel computation
- other

Problems in embedded system

Embedded systems have two categories: embedded control systems and embedded data-processing systems [14]. Embedded control systems are also called real-time control systems which react to external events, execute corresponding functions to deal with incoming data and produce the result within restricted time. On the other hand, the demand for lower cost control forms huge

pressure in market place. How to balance the constraints of cost and time encourages design engineers to develop specific design methodologies that deploy software and hardware components of HW/SW co-design.

Meanwhile, embedded systems for telecommunication applications need to deal with more complex data processing, including data receiving, data compression/decompression and routing. Such complex computing systems require more powerful processors (e.g. DSPs or ASIPs), with ASICs. Therefore, researchers are developing advanced approaches related to specific HW/SW partition, performance analysis and evaluation.

Partitioning

Certain segments in a software program may be bottlenecks, downgrading the performance in the execution of the software programs [15]. In this case, the crucial issue of ***partitioning*** is to develop new methodologies to identify those critical segments efficiently in system level at the aid of CAD tools including modeling, verification and simulation. Thus, some interactive and iterative approaches on how to map on hardware/software draw the concern of many researchers in this area.

Some of these approaches are implemented automatically based on some partitioning algorithm to identify the critical operations in software code. The others are considered and guided by human designers by the aid of CAD tools manually. In addition, resources constraints may force tasks executed in serial order, resulting in difficult scheduling in partitioning issue [14].

Software compilers

Since embedded data-processing systems applications often execute dedicated software programs for telecommunication, applications-specific instruction processors(ASIPs), which select particular instruction set (IS) to match the specific application software, come to support specific programs with high-performance and to make programmability more feasible compared with ASICs [15] . Given the performance, power, flexibility and development time, ASIPs are between general-purpose processor and ASICs . For example, the design time of ASIPs is lower than that of ASICs, and performance of ASICs is higher than that of ASIPs [15]. The specific IS may support fast execution of particular programs, improving the performance of applications and making ASIPs

more competitive. However, ASIPs solution introduces a ***software*** problem related to instruction-set selection (ISS) and code generation [15]. How to select an instruction set for an applications may affect the underlying hardware structure of ASIP, and the design of the corresponding compilers, which generate efficient codes to perform desirable tasks on such ASIP [15]. Although re-targetable compilers may generate high-quality binary codes oriented to desired instruction sets and specific hardware architectures, some technology challenges in the development of such compilers have existed and have not been solved completely until now [16], requiring great efforts and much time in co-design environment.

Software Modularity

software modularity makes it possible for designers to reuse of existing modules [27]. Such software architectures allow to separate complicate functions into many simpler modules, reducing the design of the complexity of software. However, dividing a complicated task into too many processes may involve in many exchanging data among processes, reducing the overall performance to complete the whole task. As a result, meeting performance constraints presents a huge challenge to such architectures. Thus, the exploration of a methodology to shorten the execution time of modular software has special meaning while re-use is valued in commercial profitability.

Parallel computation

Since most applications for embedded computers require to respond external stimuli instantly, computations distributed physically on ASICs may be performed with the computations in CPUs concurrently to satisfy the performance constraints. This aspect increases the difficulty of co-design when designers must consider relevant scheduling and interfacing problems in several ASICs units to meet the performance constraints of the applications.

Other

As networks require much faster communications, network on chips (NoCs) have developed more complex systems, more stressing the speed of packets processing. How to evaluate and analyze the performance of NoCs on codesign becomes a demanding problem [20].

In CAD field, co-simulation methods have to be explored. Better tools need to be developed to support the conception of better system-level models in hardware/software co-design [15].

2.4.2 Partitioning concerns

Designers may use partitioning algorithms to select functionality, to balance the hardware and software components, and to meet the requirement of computing system automatically. The tools including those partitioning algorithms will finish the whole partitioning process automatically with as little designer's intervention as possible. Besides using partitioning algorithms, design engineers may complete partitioning phase semi-automatically with CAD tools(such as simulation tools). Several partitioning concerns related to hardware/software co-design must be taken into account [14]:

- hardware/software mapping
- hardware sharing
- interfacing
- scheduling

Hardware/software Mapping

Given the system specifications, hardware/software mapping refers to choosing some particular components to be executed on microprocessors as software and to determine the other parts to be implemented on a set of logic circuits as hardware.

The **Figure 2-4** shows that multiple functions (*v1,v2*) in a program cannot be executed on a processor concurrently. When these functions are not data dependency and are mapped onto different hardware units (processor *p1, h2*) respectively, these units may execute those functions (*v1,v2*) in parallel.

Hardware sharing

Designers usually can analyze the relationship among different functions of a program, place those functions into a single hardware unit through the consideration of performance and cost constraints. As a result, many functions can share the same hardware resources and a hardware unit may cover and implement several different software functions. This process is called hardware sharing, whose

goal is to use a minimum amount of hardware resource to meet the cost constraints(see **Figure 2-5**). In addition, while considering hardware sharing, design engineers have to think of the scheduling problem to achieve better outcome of hardware sharing.

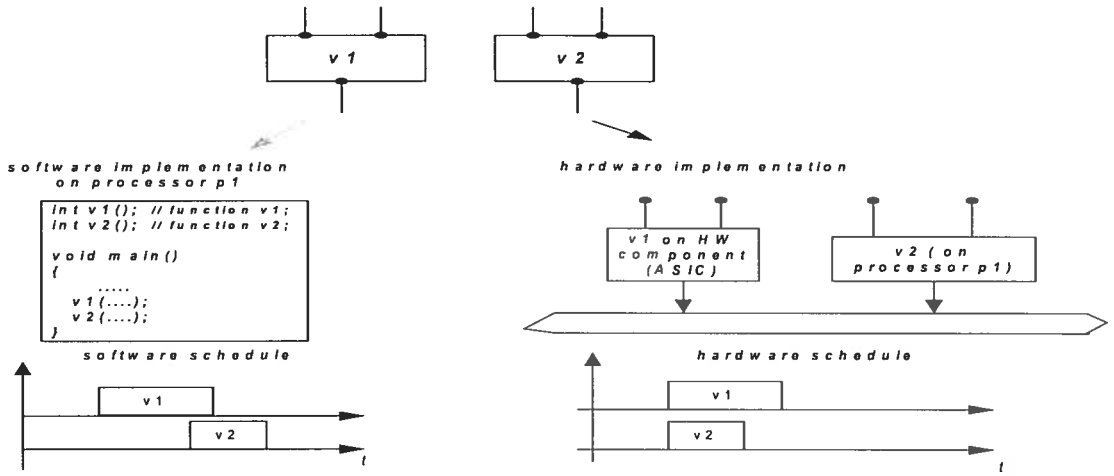


Figure 2-4 Hardware/software mapping [14]

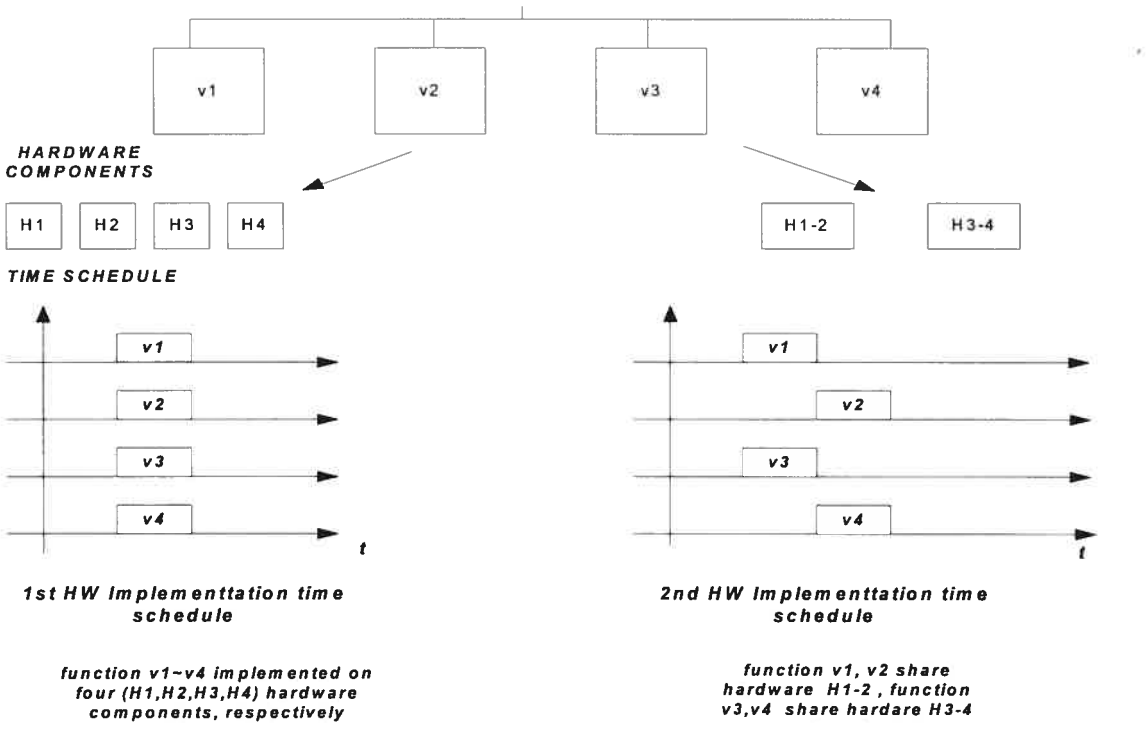


Figure 2-5 Hardware sharing [14]

Scheduling

Scheduling decides the start time at which different hardware (HW) units begin to execute a particular function. Given the allocation of hardware resource, this process is vital, especially for the hardware sharing. Through scheduling, designers have to consider the relationship such as dependencies, priorities of hardware units to prevent the conflict of those operations. Effectively scheduling limited hardware resource makes it possible to fulfill overall performance of computing system and to decrease the cost of resource at the same time. Functional pipelining is a typical example of scheduling [14]. **Figure 2-6** depicted this idea.

After transferring outputs to next HW unit v1, HW unit v1 starts to deal with new incoming data again. HW unit v2 depend on the result of HW unit v1. There is a data dependency between v1 and v2. Therefore, unit v2 must wait for v1 until v1 unit completes operations. The functions of whole system are scheduled to run concurrently or sequentially according to their relationship of data dependency and priority [14].

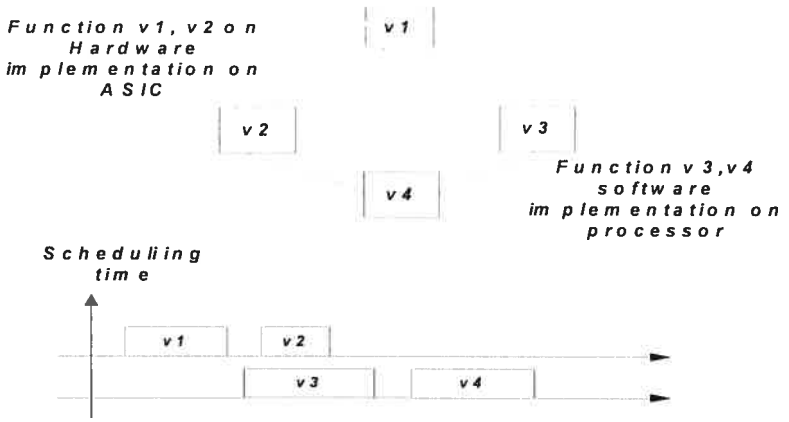


Figure 2-6 Functional scheduling [14]

Interfacing

Since the mapping phase separates computing functions onto different processing units, either software or hardware, to execute the particular functions in a program, interfacing focus on how to

communicate data between those units (HW and processor). Usually such communications are mapped as some specific channels between HW units (or ASICs) and processor. In addition, the external time for transferring data has to be considered and scheduled to prevent the data conflicts between HW units outputting the data and software functions waiting for the data as input.

Figure 2-7 explain the work what should be considered in interfacing phase.

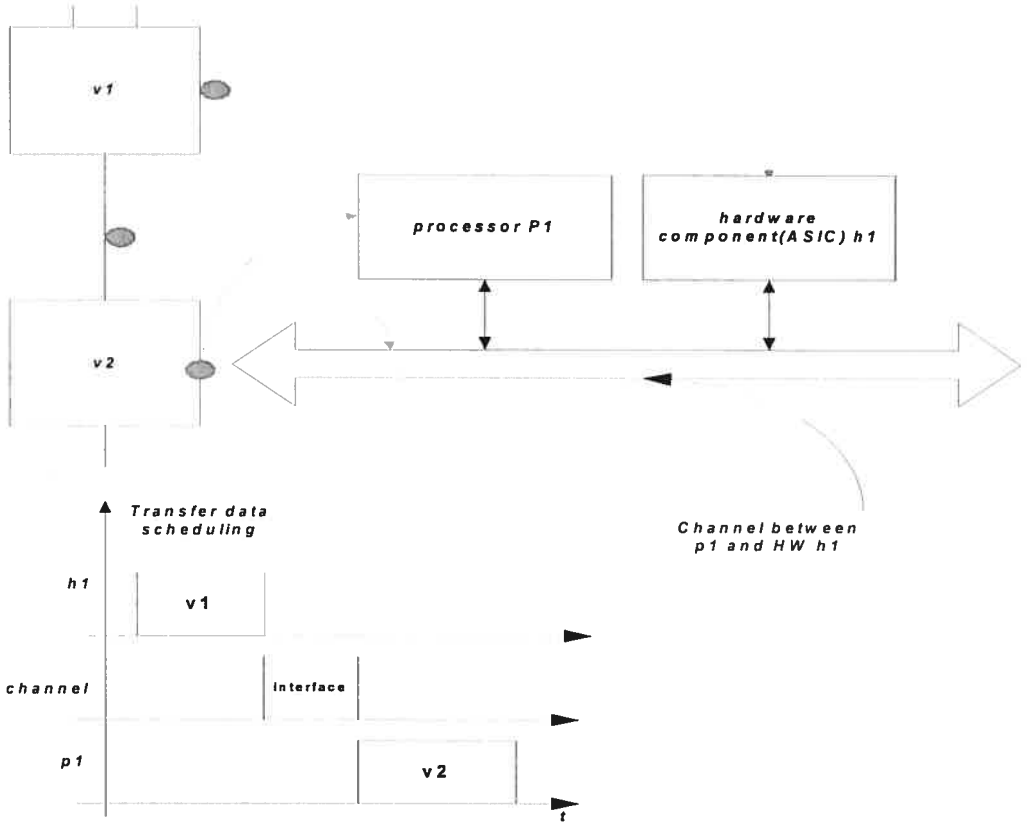


Figure 2-7 Interfacing [14]

2.4.3 The strategies of partitioning

Various system-level partitioning approaches have been developed to divide the behaviour specification into a set of software processes running on CPUs and co-processors in hardware/software co-design. Two approaches stimulate the interest of researchers: software-oriented and hardware-oriented [14].

The Cosyma synthesis tool suite (launched at the University of Braunschweit, Germany)[15] use software-oriented algorithm to partition a computing system into the components running on the processor and the components implemented in ASIC hardware, shortening the executing time of

overall system. Software-oriented approaches start with a compiled software program, determines the bottleneck of software and move the bottleneck segments into dedicated hardware coprocessors. The approach is a loop partitioning process to shift critical functions implemented in hardware until all timing constrains of a computing system are fulfilled [14].

A hardware-oriented algorithm has been introduced by Gupa and Demicheli and applied in the Vulcan synthesis tool suite [14, 15]. The approach starts the initial step to set all functions onto hardware and meets the performance constraints of a computing system by iteratively moving the functions executed on hardware coprocessors onto the software components running on the processor [20]. Thus, the functions remaining on hardware circuits for synthesis may be implemented on a minimum amount of ASICs hardware with lower cost of the system design. One benefit of the approach is to reduce the cost of hardware resource, fulfilling the overall performance constraints of the computing system [14].

2.5 Our work on the hardware/software co-design

Considering commercial profit and time-to-market shortening, design engineers must pay more attention on the reuse of modular software. The research on how to identify the crucial segments in modular software and implement these parts on hardware draws more attention in hardware/software co-design. New approaches dealing with partitioning of oriented-object systems are needed in order that such system can keep the flexibility of modular programming and gain high performance with the combination of hardware and software. Our work will be concentrated on this aspect of hardware/software co-design, exploiting the object-oriented paradigm.

We are using the Click router, which inherits the strengths of oriented-object technology such as flexibility and reuse. However, the transferring cost among objects may lower the overall performance of the Click system. Software optimization may not eliminate such cost due to the way in which Click routers process packets (see **Chapter 4.3**). Therefore, the alternative way of using hardware to implement some function plays a much more important role in accelerating packets processing in network. Thanks to the previous research that D.Quinn, M.Hubin[12] has completed a method on how to build the interface between Click router running on a processor and ASIC

components on StepNP platform (described in Chapter 4.2) has been proposed, we are able to develop a system-level approach on how to identify the bottleneck fragments in Click code and move those fragments into hardware components and how to evaluate the performance of network-on-chip (fixed Click inside) in co-design. In fact, D.Quinn and M.Hubin[12] developed a way to create connection between hardware parts and software parts in StepNP environment without discussing how to partition and evaluate the whole performance in the whole IPv4 router. We used their way to do the simulation and to finish the evaluation on our partitioning approach for the whole Click IPv4 software router in StepNP platform. Meanwhile, since D.Quinn and M.Hubin choose a specific Click element (CheckIPheader) on a crucial path to implement their method without implementing their method on the other Click elements on different downstream routers, their way may cause deadlock in simulation, we did some small modification on their method (the detail is discussed in **Chapter 4.5** section 1).

2.6 Our partitioning methodology

Our partitioning methodology is created based on model-based co-design (MBC) performed in higher abstract level. Model-based co-design (MBC) [24] partitioning refers to using simulation modeling approaches to explore the way of hardware/software partitioning in co-design. My partitioning methodology is created based on such idea. Although more and more CAD tools have been developed, we may not search for a solution without human users' guidance in most hardware/software co-design problems[15].

Based on simulation modeling technique as well as analyzing Click code manually, we explored a system-level methodology to partition Click application onto hardware/software, implementing a Click IPv4 router on one processor SoC architecture. Given the methodology, we did an evaluation for the performance of the entire IPv4 router architecture in hardware/software co-design.

Chapter 3

Multiprocessor Concepts

As the computing for embedded systems gets more complicated, those complex computations are distributed onto multiple processes and performed on several processors, with inter-communications links between the processors[21]. **Figure 3-1** is a example of the **distributed system** which includes both a DSP (digital signal processor) handling signal processing and a microcontroller dealing with external user interaction [21].

Furthermore, many embedded systems may perform time-critical tasks, such as real-time transactions. Due to the technique on semiconductor process, the yield of the small size chip is larger than that of the big size chip out of a wafer[30]. Therefore, using a couple of small processors including simple operations may cost lower than does using a large processor which contains complex operations[21]. Due to the constraints for higher performance and lower cost, the requirements for distributed computing motivate to search for new design approaches oriented to multiprocessor architectures in hardware/software co-design[27].

Also, while designers must handle the more difficult problem about processor parallelism caused by the distributed computing, the current multiprocessor architectures makes the flexibility to add (or cut) the number of processors possible and improves the performance of software by distributing computation among multiple processors[27].

This chapter will present the following sections. **Section 3.1** will concern some current problems on multiprocessor. **Section 3.2** will introduce some existing researches on the design of multiprocessor.

3.1 The basic knowledge on multiprocessors

A multiple processors architecture is an architecture defined with the following major features[31]:

1) Two or more central processing units (CPUs)

2) Shared memory and shared I/O

3) Hardware and software interact at all levels (including hardware and operation level)

4) Operation system for such architectures

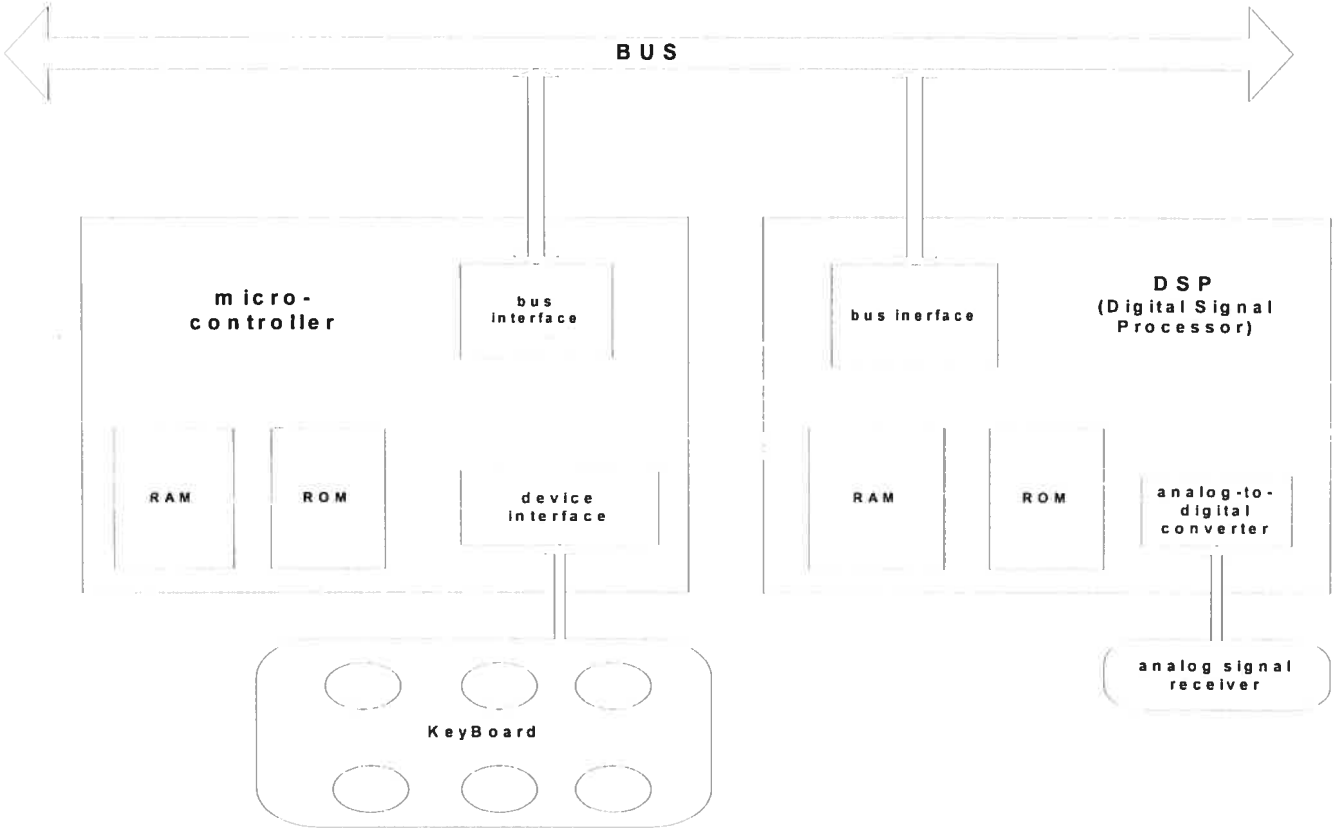


Figure 3-1 A distributed system on embedded chip [21]

Given the multiprocessor architecture, some existing problems should be considered:

- Basic organization
- Synchronization
- distributed system scheduling
- processes partitioning
- distributed process allocation

Basic organization

Based on different physical connections, the basic architectures of multiple processors may be classified into two typical categories: a single-bus multiprocessor and network multiprocessor[30]. The architecture of a single-bus multiprocessor is depicted in the **Figure 3-2.**

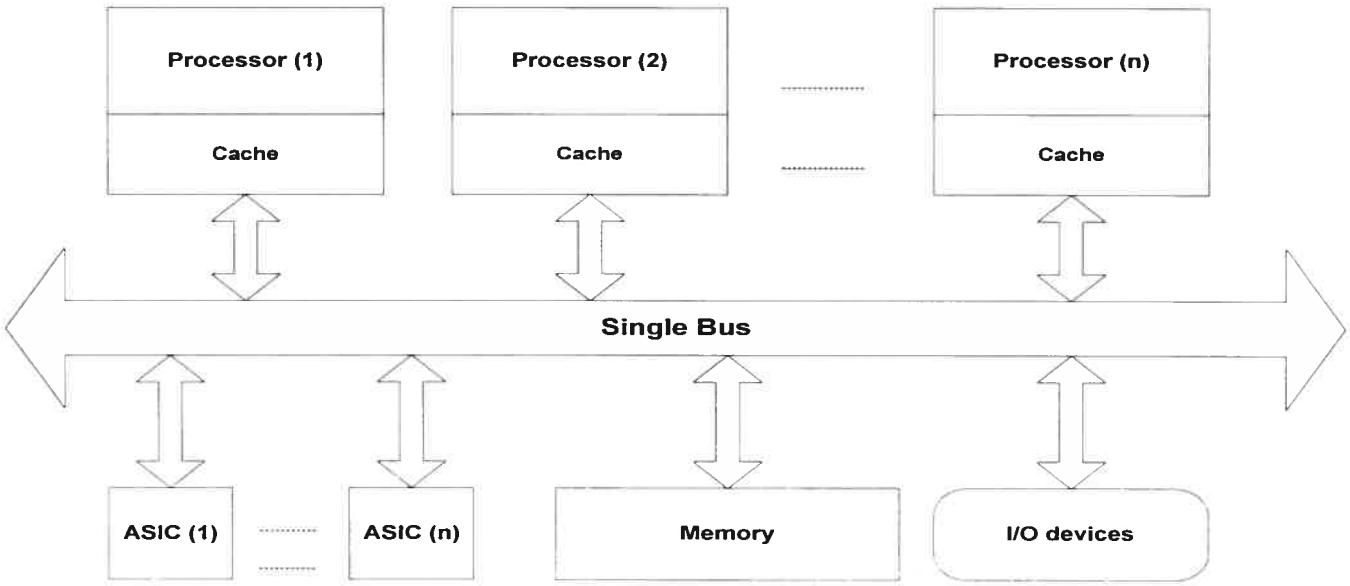


Figure 3-2 A single-bus multiprocessor [30]

By replicating the data in the caches, the caches in the close-processor structure may help to reduce traffic between memory and processors, lowering the communication pressure on the single bus. Single-bus organizations are useful and attractive when the number of processor is not quite large, usually between 2 and 32[30].

Network multiprocessor: processors are connected by a network [30] depicted **in Figure 3-3.**

Such structure overcomes the limitation of single-bus designs and may s upport more processors running in parallel, from 8 to 256[30]. Network multiprocessors may use message passing method to exchange data among processors[30].

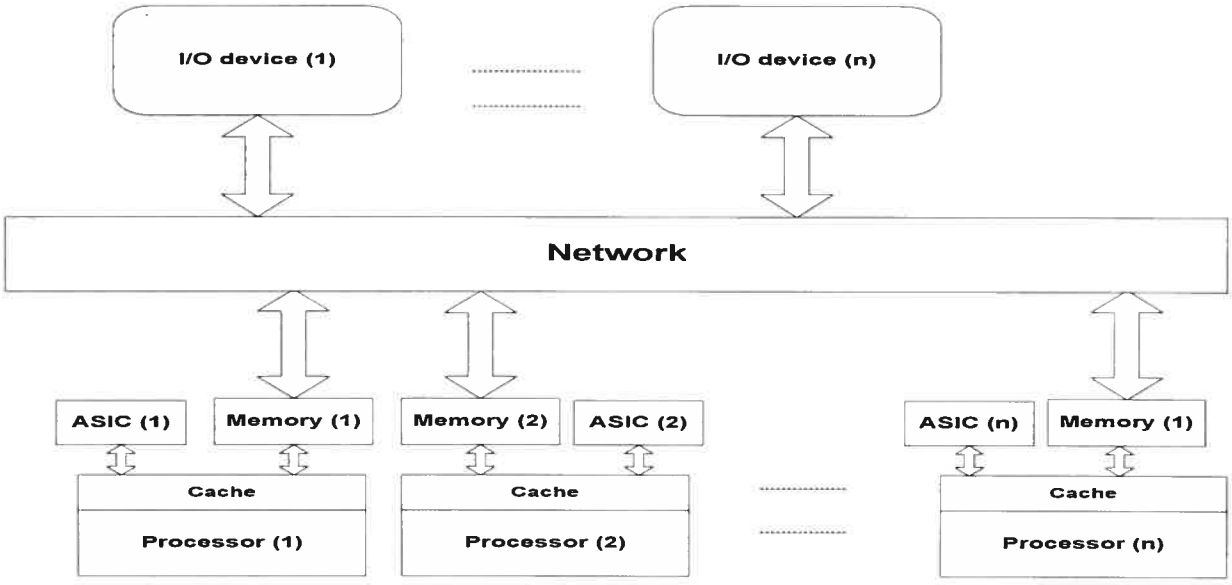


Figure 3-3 multiprocessor architecture with Network connection [30]

Synchronization

When processors are running concurrently, they may require exchanging or sharing data. Thus, the communication needs to be done to handle the shared resources between the processors. This process is called synchronization or coordination[29, 30]. Some efficient techniques are widely used to support this behaviour.

1) Locking

When a processor is working on the shared memory at a time, it locks the memory and other processors that require using the same memory have to wait at that moment until the previous processor releases the shared memory. This process is called locking and unlocking [29, 30].

The goal of locking techniques is preventing write/read conflict in the shared memory. Otherwise, a processor may work on the old data before the previous processor finish updating the data.

2) Message passing

Message passing is an alternative approach to handle how to share the data. This process is similar to the communication on local area network. When a processor as a sender sends a message to the processor as a receiver, the sender notifies the receiver that the message has been sent out. In

response, when the message gets to the receiver, the receiving processor will acknowledge the sending processor that the message has been received. This process is called message passing technique [30]. By this way, the processors may communicate the data between each other.

Distributed system scheduling

Process scheduling determines when hardware engines(e.g. processors) start to perform a process and how the hardware engines use available time efficiently[29]. Process Scheduling definitely affects the overall cost and performance of the distributed system. While making scheduling strategy, designers must consider two major points: the time required to run the process on available processors and the number of processors required to meet time constraints[21]. Usually, scheduling policies may be divided onto two categories: static scheduling and dynamic scheduling [29].

Static scheduling determine on which processor each process will execute according to existing processors and even the time to run each process when a program is designed. Thus, every process may run on a certain, fixed processor, preventing high overhead if scheduling is performed during system execution. However, this method may not suit the case that the number of parallel processes is likely to change at the runtime[29].

Dynamic scheduling can solve the above mentioned problem since such policy may load or unload processes running on a processor according to the changes of processes at runtime. Although this policy increases the cost for rescheduling process, it can get better performance when the number of processes in parallel changes frequently and rapidly and the cost for swapping the processes is low.

Until now, some important scheduling algorithms have been developed such as single shared ready queue, co-scheduling and dynamic partitioning.

Single share ready queue (SSRQ)

Based on some rules such as FCFS(firs come first served), SJF(shortest job first or RR(round robin), designers use a queue shared by all processors to allocate processes onto processors in

SSRQ policy[29].

Co-scheduling

Coscheduling arranges processes to be executed concurrently on different processors. Such policy may be suitable to the situation in which processes require frequent data exchange[29].

Dynamic partitioning

In dynamic partitioning policy, the processes of a program are dynamically scheduled onto a set of processors chosen from available processors. Then the program periodically checks the number of processes from scheduling server. This number is called "ideal number", which indicates the processes number with which a program may be executed in an ideal condition . If the ideal number is more than that of processes which are running, the scheduler actives previously suspended processes again. If the number of running processes is more than ideal number, some processes will be suspended to meet the most ideal situation in which the program is executed best [29].

Process partitioning [21]:

Process partitioning refers to dividing functions without data dependency (one running has to wait for another's output as input) into several smaller processes and running those processes on different processors. Good partitioning might ensure hardware resources (e.g. processors) to be used efficiently and prevent the phenomenon that some processors are very busy at computing and the others are often idle [21]. Such partitioning may shorten the idle time of a processor, speed up the executing time of processes in parallel on different processors and improve the overall performance of software architectures [21].

For example (see **Figure 3-4**), a process p1 includes two functions, which are likely to be partitioned onto two process x and process y before partitioning. Since two functions are executed sequentially, the process p3 that depend on the two values from two functions has to wait a long time, increasing the idle time of processor 3. After process partitioning, process x and process y may run on two different processors concurrently. As a result, the computing outcome of two processes x and y may be sent the process p3 earlier than before the partitioning, shortening the idle time of processor p3 [21].

Some typical contributions in this research area includes the process partitioning algorithm developed by Huang [21] to minimize the delay between inter-process communication [21], and Ma et al. branch-and-bound algorithm to reduce the number of processes and inter-process communication cost.[21].

Distributed process allocation [21]

Distributed process allocation considers how to assign a group of processes running onto a few processors. For example, in **Figure 3-5 (a)**, processes f1, f2 and f3 require exchanging data (messages) frequently. Process f4 has no or a little data exchange with the other processors [21].

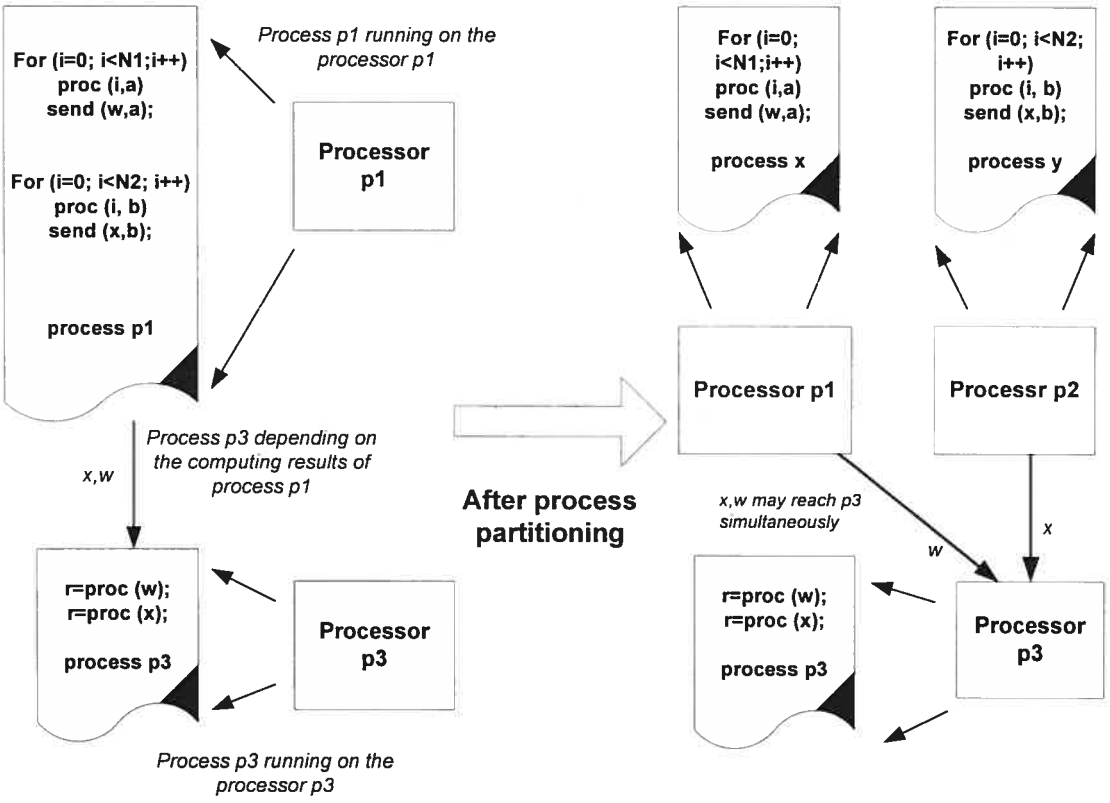


Figure 3-4 An example to describe the influence of process partitioning on distributed system performance [21]

Thus, we may consider to arrange f1, f2, and f3 running on the same processor(P1), and f4 running on the processor P2 as depicted in **Figure 3-5 (b).** Since f1,f2 and f3 are running on the same processor, we may use much cheaper, faster shared memory as a way of exchanging their messages. Such allocation may minimize the bandwidth of communication on the link (e.g. bus)

between processor P1 and P2, reducing bus conflict and improving the whole performance of distributed process allocation[21].

The process allocation can be done either under the designer's guidance or in automated algorithm. The first automated algorithm was developed by Stone[21], but might not suit more than two processors allocation. After the birth of the algorithm, many researchers have continued to developed more approaches on the process allocation. For example, with the help of graph-matching heuristic technology [21], Shen and Tsai's algorithm may effectively allocate the processes running onto the available processors, lowering inter-processor communication traffic to

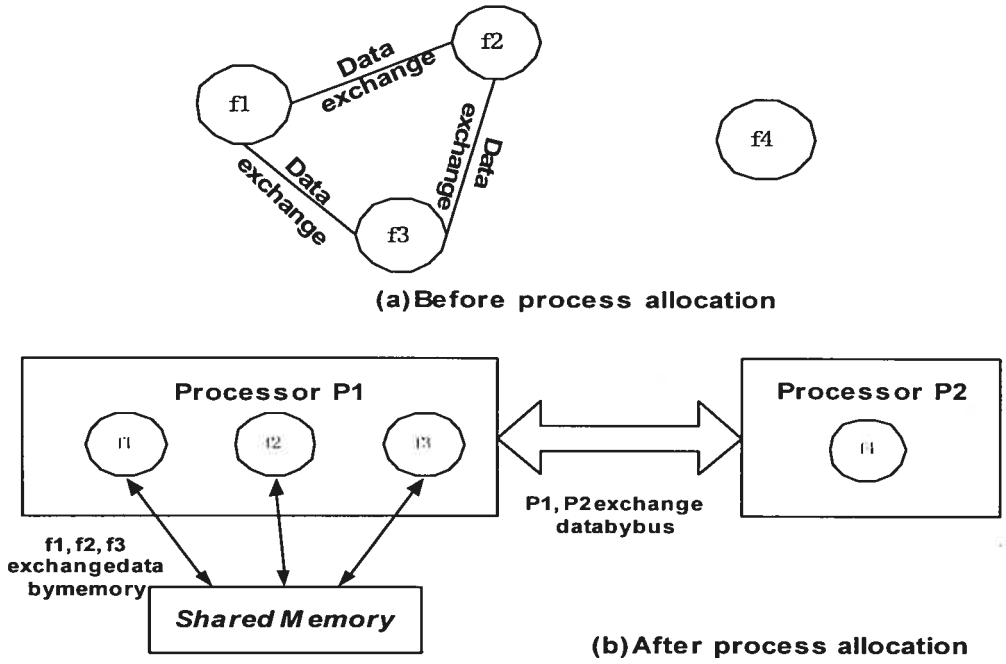


Figure 3-5 how process allocation affects the performance of distributed computing system [21]

a minimum amount and balancing overall system load [21].

3.2 The related researches on multiprocessors

Nowadays, System-on-chip stimulates much interest of researchers in multiprocessors architecture design. Several existing approaches have presented various model to support the design related to

applications-specific (dedicated applications like embedded systems) multiprocessor system-on-chip[27]. Among those research projects, POLIS[14] introduces a target architecture with the combination of general-purpose processors and other possible components such as DSPs and ASICs. COSY[14] has developed a layered communication model based on the POLIS approach [27]. CoWare[14] presents a more generic architecture and design flow to help applications-specific multiprocessor systems-on-chip design[27]. However, above mentioned approaches have at least two limitations to support various applications-specific multiprocessor systems-on-chip design:

1) Those approaches give the restriction on the use of components, only supporting some specific components. For example, designers can choose only DSP or products with a particular type(MC68000, or ARM 7, etc.) [27] in those approaches.
2) Their protocol channel library may offer limited protocol types for communication among components. Only some specific components can be connected with the specific communication link[27].

Given those situations, Amer.B. and Damie.L introduce a generic model and a methodology to support heterogeneous processors in applications-specific SoC design.

Compared with the above mentioned methods, this approach comes up with an architecture model on the higher level, and gives the more abstract architecture model, which may be the guide of building more generic platforms and may be used in much wider application fields without the limitation of components and communication models. For example, StepNP platform is such a practical application project based on the generic model[27], helping network-on-chip design. **Figure 3-6** depicts such a generic architectural model.

In multiprocessor architectural model, designer must be concerned with the following three aspects:

Modularity is the nature of oriented-object technology allowing to divide a complex system into some small simple components, which encapsulate specific behaviours. The internal behaviours may not be modified by the outsider directly. Due to the flexibility of modules, components may be reused and assembled to support various applications according to different requirements [27].

Components may be software, hardware, and communication network (e.g. channel, or bus). Hardware components include processors, memories or particular peripherals. Components inherit the quality of modularity. In this model, processors components may choose various brands of CPU such as ARM7 or Mc68000 to construct heterogeneous systems, meeting various practical SoC designing[27].

A multiprocessor architecture model should be **scalable**. In such architecture, designers should be able to increase or reduce the number of components to support various application-specific

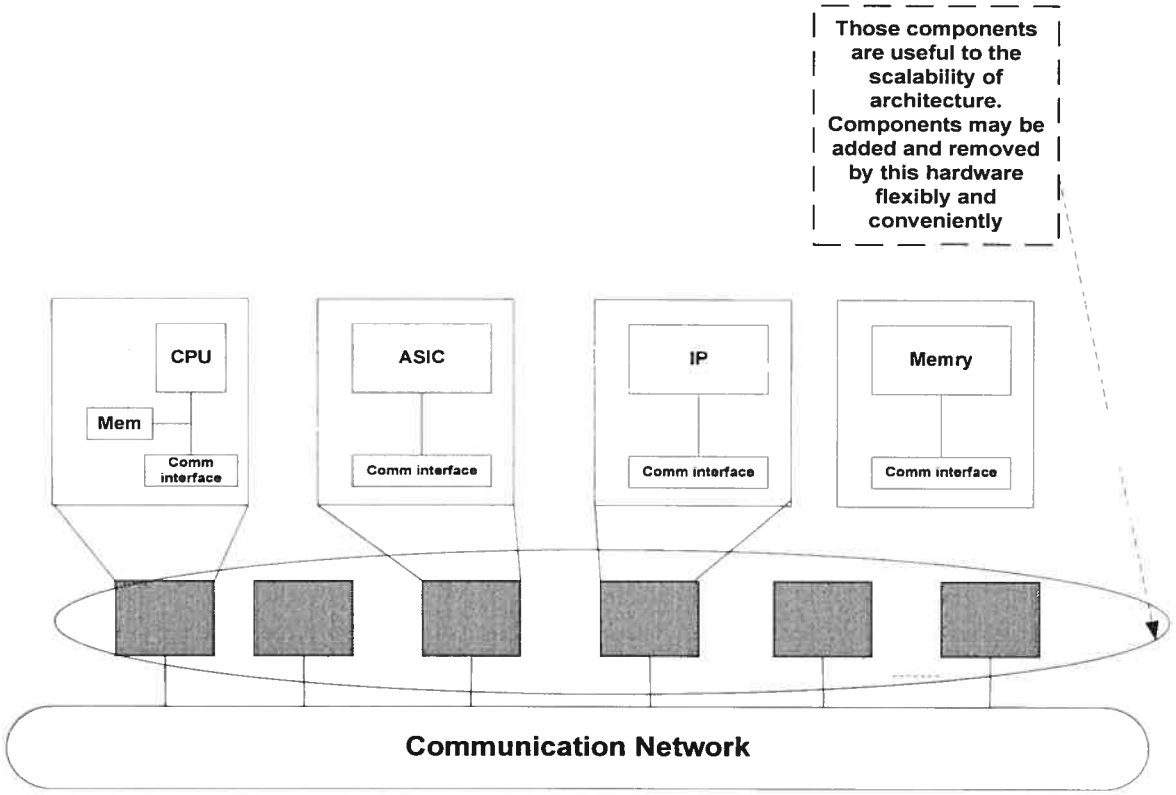


Figure 3-6 Generic architecture model [27]

systems [27]. Now components can be CPU or communication channel [27].

In the **Figure 5-2** of chapter **5**, the architecture for click application on StepNP platform may be regarded as an instance of the generic multiprocessor architecture model. The detail is depicted in chapter **5**. Meanwhile, Amer.B. and Damie.L present a generic methodology on multiprocessor SoC design. The overall design flow is descried in **Figure 3-7**.

The major steps are introduced as the following:

1. Determine hardware architecture with fixed parameters including CPU types, such as ARM7, MC6800 etc, from the hardware aspect. Consider which process of the application-specific should be executed concurrently on the processors from the software aspect. These two actions may be taken concurrently.
2 Select parameters such as the number of CPUs, communication protocol for creating high-level model.

Furthermore, all parameters and next step has been presented in **Figure 3-7**.

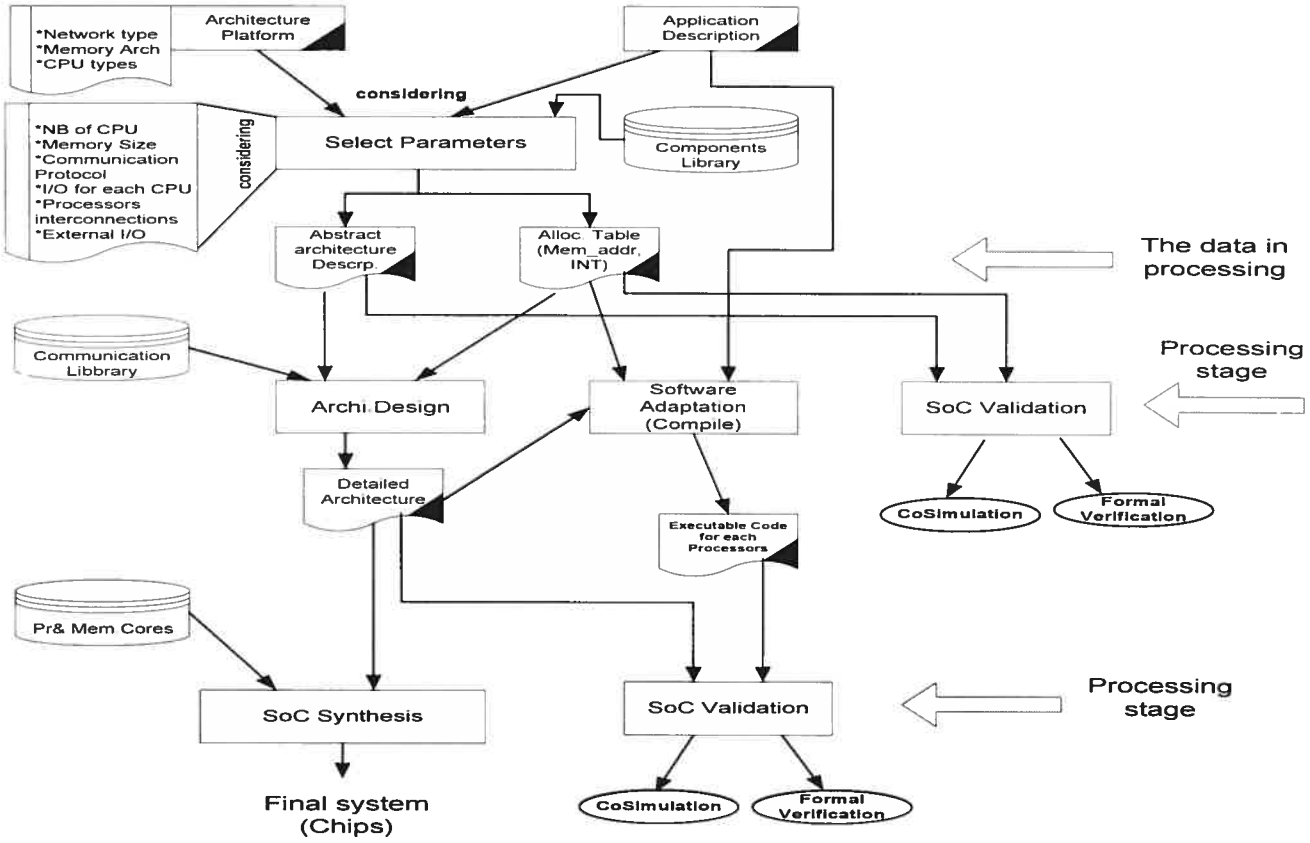


Figure 3-7 the generic design methodology of a multiprocessor SoC architecture [27]

The Select Parameters process(see **Figure 3-7**), which focuses on hardware design based on applications-specific parameters, generates abstract architecture description and allocation table.

The abstract architecture is a rough layout of the architecture platform. Allocation table consists of detailed information about architecture, including interruption levels reserved for each CPU [27]. An allocation table may be used to refine the rough layout into further detailed architecture for SoC synthesis and to determine various interfaces including the connection between processors [27].

The work of software design mainly focuses on compiling programs and generating the binary code targeted on each CPU [27].
After the validation on CAD simulators, the final results of design are generated by SoC synthesis process.

This approach introduces a general guide on how to build an architecture on SoC efficiently and supports the development of various application-specific systems. Designers might be able to follow the design flow depicted in **Figure 3-7** to complete embedded chip design[27].

3.3 Our work on the multiprocessor

As we mentioned in the Chapter introduction, modular components router has benefit on flexibility and extension compared with conventional router. Click router is a new architecture to build flexible and configurable router with high performance [1]. When we analyse the structure of Click software router, we know that Click configurations are modular and easy to extend [1]. Such structure has potential parallelism, make it more possible to running Click router on multiprocessor. In addition, Click router modularity has its own flaws and limitation (see the explanation in the opening of **Chapter 4**). Our work on the multiprocessor focuses on exploring a way to implement the Click router applications system (coded in C++) on multiprocessor.

While considering the hardware design of multiprocessor architecture, researcher must consider whether software applications are available to run in parallel and whether programs can run faster on multiprocessor architecture. In fact, it is even harder to find applications that can perform well on multiple processors, taking advantage of such hardware architectures efficiently[30]. Also, designers should consider whether programs can be reused and whether the cost to modify the software for such architecture can be minimized as the number of processors increases[30]. Given

the reuse of software, an oriented-object language program presents such strength due to its inherent nature that " describes a highly mobile concurrent system in which new objects are created as computation proceeds and the linkage between components changes as references to objects are passed in communications "[29].

A particular example of the relevant applications is SMP Click router, originated from Click router [1] (the detail in **Chapter 4.3**) and designed for running on multiprocessor architectures. The nature of modularity of SMP Click allows users to write separated configurations, achieving the goal of configuration-level parallelism[28]. Configuration-level parallelism refers to splitting a configuration into smaller sub-configurations and executing the sub-configurations on multiple processors concurrently.

Due to the cost of passing between modules, the Click router may get speed up either with the help of coprocessors or by running on multiprocessors in hardware/software co-design. However, it may be difficult to implement SMP Click on multiprocessor network on chip (NoP) co-design for the following reasons:

1) The expensive cost for CPU scheduling on the centralized task queue of elements makes the Click designers choose the private a task queue of elements for each separated threads running on different processors. However, since SMP Click inherits the CPU scheduling of Click (see the feature 3 in **Chapter 4.3.2**), such scheduling scheme may it difficult to determine the running time for the element on different processors. Thus, it may take place that one processor is very busy, and others are idle. How to balance processors resource becomes a problem that need to be solved [28] .
2) Now that SMP Click use separated threads for different processors, designers must consider how to create shared locking to synchronize the data exchange between processors. That increases the complex of handling the packets communication between different processors [28] .

Therefore, above mentioned problems leave much room to explore an approach on how to execute

Click router on multiprocessor SoC architecture. Meanwhile, The structure of modularity contributes large flexibility to Click router, making it possible to implement the approach.

3.4. Our methodology for Click implementation on multiprocessor

STMicroelectronic has developed a simple multi-threaded processor architecture on which a hardware thread executes the overall Click application with the different or same configuration files. In their case, there are very few data dependency and data exchanges between the different threads executing the Click application. This approach is called inter-packet processing level[26]. Our approach focuses on using multiple ARM processors to process a single packet with possible high inter-block dependencies [26], implementing the configuration-level parallelism [28]. In other words, the Click application will be partitioned on the different processors of the architecture.

With the guidance of the generic architectural model, we have explored a system-level design approach on how to implement Click router with multiple processors and evaluated the method on the StepNP simulator. The detail of the method will be introduced in **Chapter 5**. The approach may be considered as a contribution on how to design a suitable multiple processors SoC architecture to run programs written in oriented-object language (e.g. C++).

Chapter 4

Hardware/Software Co-Design Methodology on Partitioning

Although the Click element (the detailed in section 4.3) has flexibility and scalability for various router requirements, such modular element causes the cost of performance on processing packets in two-folds:

1) the modularity of elements brings about the expense for passing packets between elements in flow chart of Click router configuration [1]. For example, one of the three virtual functions such as simple_action(Packet *p), pull(int *port_number*) or push(int *port_number*, Packet *p) in each element (or C++ object) must be called when a packet is forwarded from one element to next element along the processing path in a Click configuration. A virtual function call must cost certain CPU time. According to a experimental result on Click performance, the cost to pass a packet between two elements is about 70 ns[1] and the passing the sixteen elements in a regular Click IP configuration costs about 1 ms[1] in total.
2) the modular structure may inevitably increase the cost of passing packet since the Click router has to call some elements with the general functionality [1]. A typical example is the element named "Classifier", which includes the generic functions that may not be used for IP Click router. Some generic functions are not tailored to some specific applications of IP Click router when software writers may consider its multiple usages for various Click router applications. Thus, given the usage in Click IP router, such general element codes may cause unnecessary overhead, costing more CPU time[1].

Those features are inherent in Click application software and the overheads may be very difficult to be prevented in Click software. To counteract these costs, using H/W co-design methodology is a feasible choice. The Methodology is implemented on StepNP platform.

4.1 The general presentation for methodology

Now that my research concentrates on system-level partitioning process with the help of using simulation tools and manually analyzing source code together, my work focuses on the approach from three aspects: hardware, software, and simulation platform.

Simulation Platform:

We choose StepNP platform as our simulation platform. **Section 4.2** presents the structure of StepNP platform in detail.

Hardware:

Based on StepNP simulation platform, we use systemC[1] modeling language to design the hardware units for computational system, to implement the system-level partitioning, the performance analysis and the evaluation on Click IPv4 router.

Software:

Our software mainly concentrates on an open source code: Click router application. STMicroelectronic company has modified the source code targeted to ARM7 processor SoC architecture.

Section 4.3 will introduce the working principle of Click IPv4 router and the basic knowledge of Click router application in detail. **Section 4.4** introduces how to move crucial software segments onto hardware implementation partitioning. After creating a hardware architecture in SystemC, we may analyze the modeling result of simulation and identify the bottleneck code segments for partitioning. **Section 4.5** describes the hardware algorithm in SystemC and the way on how to

[1] SystemC is a modeling language that is derivative from C++ libraries and may be used to design both hardware and software. If hardware and software are designed based on oriented-object technology, the system-level co-simulation of HW/SW may be completed more easily in the environment to support SystemC. In addition, some synthesis tools such as CoCentric(R) SystemC compiler also can implement hardware synthesis in the lower level.

communicate between the SystemC hardware modules and the software modules in C++ oriented-object language.

In our research, we modify Click software, design hardware, build simulation platform and adjust these three aspects iteratively until the constrains for the application system are reached. Since the executable specification of Click application has been defined and completed in oriented-object C++ language, the co-simulation between Click and the SystemC hardware units may be done more easily on high-level. **Figure 4-1** depicts the general outline of the methodology.

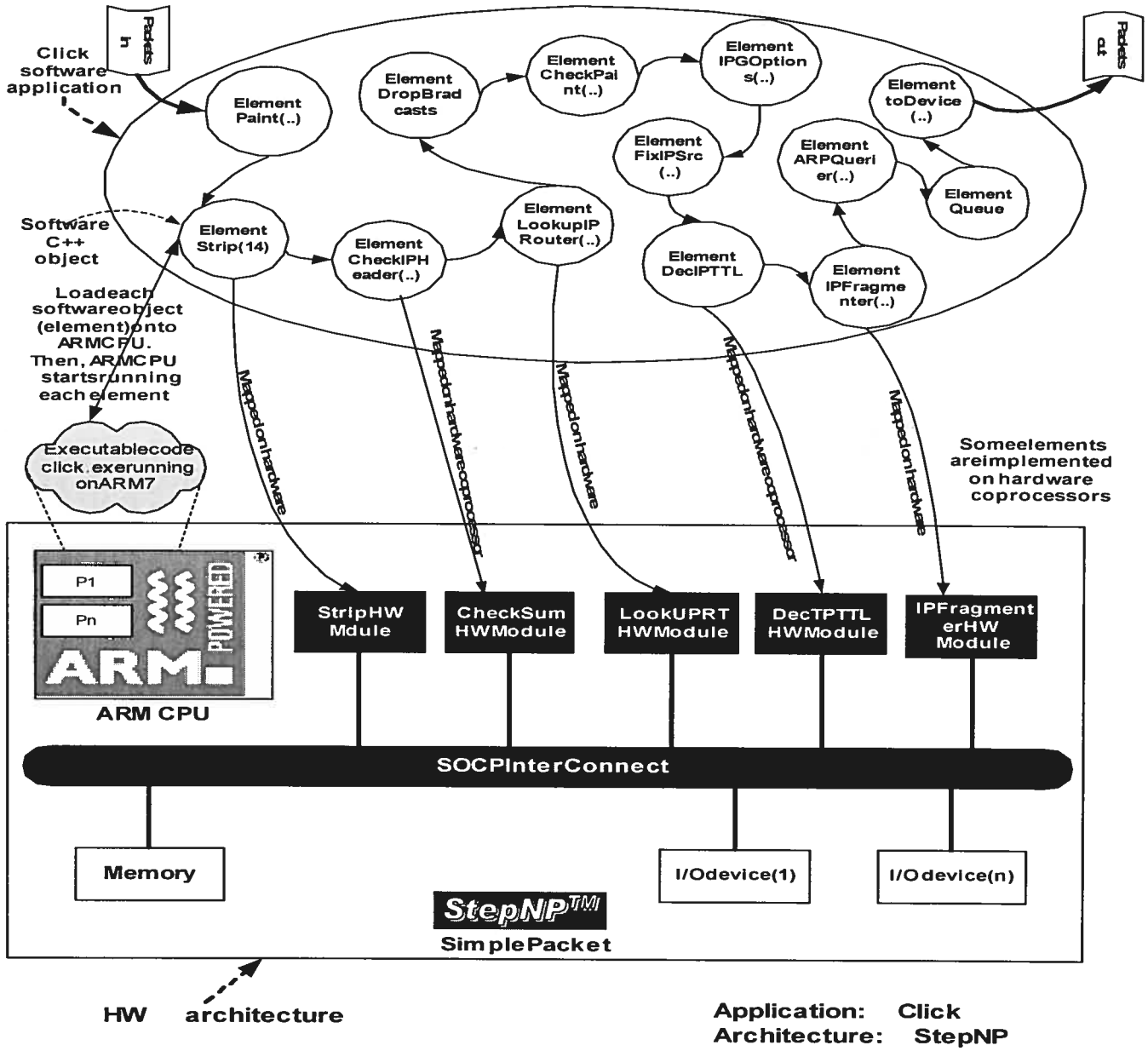


Figure 4-1 the general framework of the approach on how to map software onto hardware

First, we created the cycle-accurate executable architecture in **Figure4-1** by modifying the existing simulation platform in StepNP tool kits. Then, running each element of IPv4 router configuration in the ARM processor of the architecture, we applied simulation to count the executing cycles of each element, which show the complexity of each part of Click IPV4 code. Thus, we were able to identify bottleneck fragments that cost many cycles and map those critical operations onto co-processors tailored for the given architecture (here is ARM) iteratively [19].

With the support of StepNP platform, the approach may build the executable application-specific architecture easily and complete partition process efficiently based on the combination of user guide and simulation evaluation, shortening the time and efforts in hardware/software co-design.

4.2 StepNP platform introduction

First, we introduce the principle of StepNP platform, which is a System-level Telecom Experimental Platform[6] running on Unix-like OS for Network Processing. StepNP modeling supplies an approach to support top-down co-design verifications methodologies and to reduce the time and efforts spent on system-level partitioning stage. StepNP offers existing components libraries for designers to model hardware components and to create a system-level model frame for simulation validation and evaluation. StepNP platform contains three components relevant to ARM RISC processor.

1) MIT Click Router Platform in StepNP

StepNP has developed some patches to MIT Click router (user-model) for ARM simulator, and supplied some new elements as the interfaces to support Click's running on ARM simulator.

Thus, the user-module configuration for MIT Click router can running on StepNP's ARM simulator. Two new added elements are FwidlSink which injects packets from Unix platform into the address in StepNP, and FwidlSource which reads packets from the address in StepNP platform. These two elements establish a bridge between the internal environment of StepNP platform and external processing system or operation system platform.

2) SoC tools platform

SoC tools platform is classified two categories: 1) tools for developing embedded system on a single processor, including an instruction-set simulator , a compiler, etc. 2) tools for developing computing system over multiprocessors, including controlling, debugging and analyzing functionality[6].

3) NPU architecture simulation platform

The three major components of the NPU architecture simulation platform include modeling language(using SystemC 2.0), multithreaded processor model and a SOCP(SystemC Open Core Protocol) communication channel interface[6].

Using an existing ARM processor components as a master and SOCP communication channel, we can build a simple master-slave testing architecture (called simplePacket platform shown in **Figure 4.2**) to support the simulation of Click router for ARM CPU. Then, we choose a collection of the elements in Click and test it on the simplePacket platform.

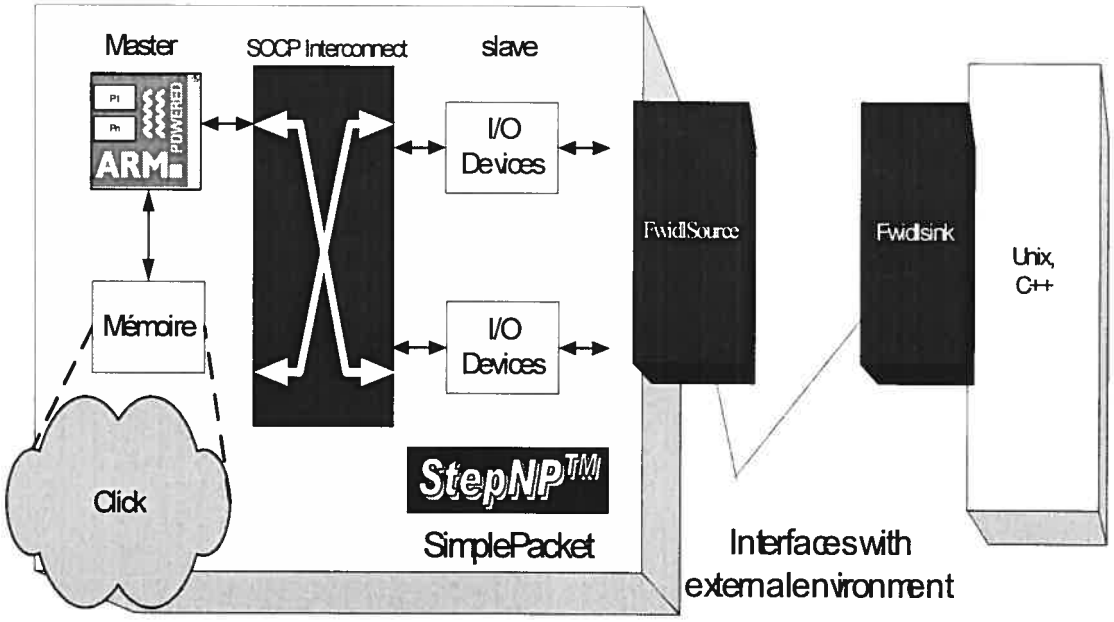


Figure 4-2 SimplePacket platform in StepNP [12]

4.3 Click router

Click router is a typical oriented-object application system developed by Eddie Kohler in MIT. Most router systems have predetermined, fixed functions often implemented on hardware(or ASIC's) before they are launched into the market. Thus, such routers leave little room for an administrator to configure routers flexibly to meet some particular requirements and to add new functions in router's configuration to support new protocol or new network development [1]. The software based [4] (means implementing most functions in software) Click router is a configurable architecture for packet forwarding processing to prevent those disadvantages. Click router has about 60 major elements to support various router configurations. Each element implements specific functions in packet processing. Deploying the Click router elements with connections (explained in section 4.3.1 in detail), the users may build a router configuration- the collection of the elements- to support the desired behaviors.

In fact, the Click **"element"** is the object of the C++ class. Some of the elements are virtual classes. According to the inheritance attributes of C++ class, Click router has two typical features about router configuration:

1. Based on existing elements or virtual classes, users may develop some new elements to extend Click functionality. After they write a new C++ object with the new functions, users are able to add the new object into Click router conveniently according to the approached taught by Click router developers. Thus, the new elements may combine with the old ones to build various configurations, supporting desired complex applications[1].

2. An element implements various functions based on the concept of C++ polymorphism. for example, RED(Random Early Detection) can behave as RED over multiple queues, weighted RED, or drop-from-front RED according to the different requirements of packet processing[1].

4.3.1 Click router element connection

The graphic edges among elements in a configuration diagram are called connections used to forward packet between the elements. Each element may have the multiple ports for exit to connect itself to the other element's port for entrance[1].

Click elements have three types of ports, pull, push and agnostic. Connections on push ports will pass the packet from one push port downstream to other push ports. Connections on pull ports will have downstream pull ports get packets by triggering upstream pull ports upstream in forwarding path. The agnostic ports may be used as push if connected to push ports or pull if connected to pull ports. The possible connections among elements are either push ports connected to push ports or pull ports connected to pull ports[1].

The types of connections among the elements are determined during the configuration initialization phase. Then, a packet will be forwarded through the established connections from entrance ports, which receives incoming packets such as FromDevice element, to exit ports, which sends packets out of router such as ToDevice element, when Click router starts to process packets. The connections are actually implemented in the way which Click scheduling modules call some virtual functions such as push(int *port_number*, Packet *p) in each element[1]. All elements in Click router include one of the three virtual functions such as simple_action(Packet *p), pull(int *port_number*) and push(int *port_number*, Packet *p).

4.3.2 Click router features

According to packets processing functionality, Click router elements may be classified into several major groups such as Network Devices, Classification, Checking Validity, Storage, Dropping, Packet Scheduling and Duplication[1]. Click router elements have all the features of C++ objects. The forwarding actions for a packet are encapsulated in each element. In addition, Click elements have the following typical characteristics.

1. First, Click router has an explicit Queue element as its storage to hold the packet. The benefit of this feature has designers handle how to store a forwarded packet[1] in a direct way.

2. Secondly, the routing table in conventional routers is shared by general CPU, interface card and any other entities in the router. Click router's routing table, however, is encapsulated in one single element involving in the packet forwarding path[1]. Due to this property, Click router may more suitably be used as a distributed router in different network card, processing packets independently.

3. Although the forwarding flow in a Click router may have many branch as the result of routing, Click router always runs in a single thread and follows the running of each element on a single processor [1]. Click router loads and schedules the element's running through the path in configuration until a packet is sent into an explicit store such as Queue. Then, Click router will continue to process other incoming packets. Using this feature, we are able to implement Click on multi-processors platform (described in Chapter **5**).

4.3.3 Click router general forwarding principle

To explain the general forwarding principle, we choose the example that Internet Protocol (IP) is used over Ethernet. When an incoming packet is waiting at the Click router, packet source drivers (e.g FromDevice element) read it from network and push it to the next elements on the path of packet processing. Thus, the packet is forwarded through a series of elements in configuration until the packet reaches an explicit storage (e.g. Queue). Once a packet is stored in the queue, the packet will stay in the queue until the packet sink (e.g. ToDevice) get ready to dump it to the network. After ARQuerier element finds the MAC address for the packet, the downstream packet sink elements (e.g. ToDevice) will pull (or move) the packet from the explicit storage and put the packet on to the output device and transmit the packet to the next website in network[4].

4.3.4 Click configuration for IPv4 router

It is well known that Internet Protocol (IP) is a major part of TCP/IP suite and is the most widely used in the packet processing route of an IP router. Thus, the capability of IP packet processing plays an important role to determine the performance of a router[3]. Therefore, we use IP Click router as an example to explain the co-design methodology in our report. We choose sixteen Click elements to construct an IP router configuration depicted in **Figure 4-3.**

Given the behaviors of each element, the setup of Click IP router must arrange the order of elements on the forwarding path in IP configuration to follow IP protocol standard, guaranteeing IP packets processed correctly[1]. The behavior for the elements in IPv4 Click router is realized based on the work presented in[1] as follows:

FromDevice(eth0): Reads the packets from external network devices (e.g. port eth0), working on the Linux kernel.

Classifer(pattern1…patternN): checks and compares incoming packets with the pre-set patterns including ARP queries, ARP response , IP packet and others. Then, forwards the packets to the

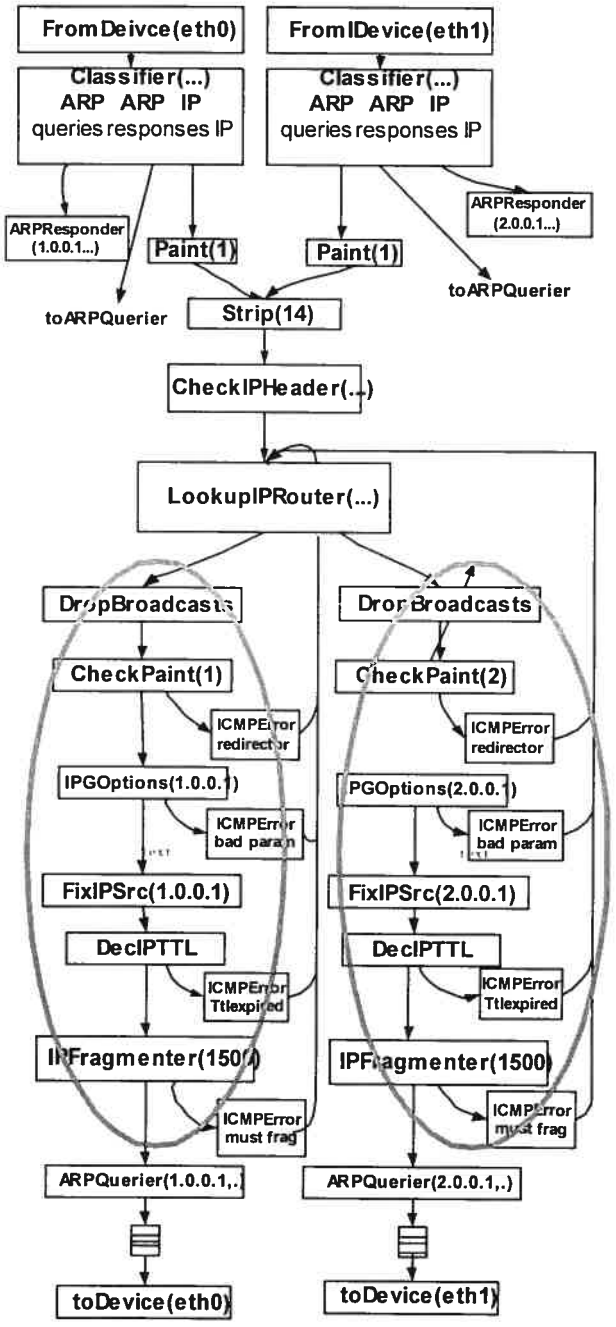


Figure 4-3 Click Configuration for Ipv4 Router[1]

corresponding output ports respectively according to the result of pattern comparing.

Paint(colorX): Sets a mark on a packet to ensure that the packet is coming to and leaving from the same interface identified by the color X.

Strip(N): Remove N bytes before a packet header. The element is usually use to remove Ethernet frame head from IP packet header.

CheckIPHeader(...): Validates IP packet header information including IP version, header length, packet length and *CHECKSUM* field [1]. The operations of this element focus on IP header field mainly. If any header information of an incoming packet is invalid, the packet will be dropped or forwarded to another element depending on the arrangement in Click IP configuration.

LookupIPRouter(...): Defines a forwarding table, matches a route in the table for an incoming IP packet and direct the packet to relevant out-port. The forwarding table here is a linear routing table.

DropBroadcasts: Drop the packet with broadcast address. This function gives router function for isolating broadcasting in network.

IPGWOptions(Myaddress): Focuses on the Options segment (if IP header length > 5 32 bit). Checks some parameters including route recording and IP Timestamping in optional field [1]. If optional field is modified, re-compute the *CheckSum* field. Parameters Myaddress is the address of current eth0 port.

FixIPScr(Myaddress): This element is for ICMPError packets. If the passing packet is an ICMPError message which uses current Ethernet port address(Myaddress) as source address, IP Source annotation (or flag bit) should be set in this element, and the *CheckSum* field will be fixed in this element [1]. Then , the ICMPError packet will be sent back to the original source address.

DecIPTTL: This element decrements the numbers in Time-To-Live field of IP packet header, and update *Checksum* field.

IPFragmenter(MTU): Checks the length of an incoming packet. If the packet size is larger than MTU, then this element fragments the large packet into two pieces whose size is smaller than MTU and re-compute *Checksum* in two separated fragments.

Queue(buffer): Stores the passing packets until the moment when the downstream elements pull the packet from queue.

ARPQuerier(...): Sends an Ethernet frame following ARP protocol, gets appropriate frame address to the destination address of the host packet from an ARP response, and encapsulates the IP packet into an Ethernet frame.

4.4 Partitioning methodology

As shown in **Figure 4.2**, StepNP platform is split into two parts. The left part executes IP packets, and the right is responsible for injecting and outputting the packets. In order to perform simulation we modified click application, we add functionality required for :

- read external packets injection on stepNP platform(FwidlSource(...))
- communication for the simulation experiment(In **Figure 4-3**, both ARPQuerier(...) and ToDevice(...) elements were replaced to modify distant communication into local communication)

4.4.1 Partitioning – implementation considerations

When we choose the parts in Click elements to be implemented by hardware module, we consider several factors which influence hardware module efficiency in terms of hardware algorithm complex, resources (gates, addition...) and time constraints. Based on those thoughts, we consider the following three points:

A) Module speedup

The hardware usually performs an algorithm faster than the software to meet time constraints. We consider using hardware to implement the code segments with computational complexity in Click elements. For example, the hardware solution with a shift register and some X-OR gates may

perform the modulo 2 operation of Checksum to meet time constraints while the operation may cost more time if implemented on software solution.

B) Shared modules

By analyzing all elements in IPv4 configuration, we find that Checksum algorithm is shared by most elements such as CheckIPHeader, IPGWoptionst, FixIpSrc, DecIPTTL, IPFragmenter. Designers may get better overall performance by shortening the executing time of the frequently used modules [30]. Considering this factor, we choose CheckSum to be implemented by hardware circuits.

C) Module functionality

According to the way of Click's CPU scheduling, the following path is the bottleneck on the forwarding route of Click IPv4 router described in **Figure 4-4**. Especially, as a multi-gigabit IP router which may handle multiple protocols and forward the packets from different direction link at higher speed, the longest prefix matching of LookupIProuter is on the must-passing path and has serious impact on lowering the overall performance of router [7, 8].

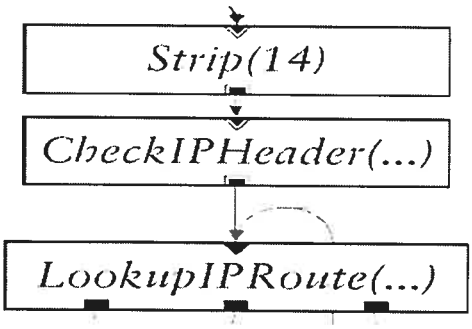


Figure 4-4. The crucial path in Click IPv4 router [12]

Considering above-mentioned points, we have as much as possible code fragment implemented in hardware circuit except the memory transfer operation along click element in IPv4 diagram. The main idea for the methodology I can be described as the following diagram **Figure 4-5**, which gives an outline structure about SimplePacket internal core part.

4.4.2 Partitioning - simulation considerations

Our partitioning methodology is based on the following two considerations:

1) We obtain all timing information from the StepNP simulator. The accuracy of our methodology depends on the simulator accuracy.

We use the simulation results when making partitioning. For the computational complexity

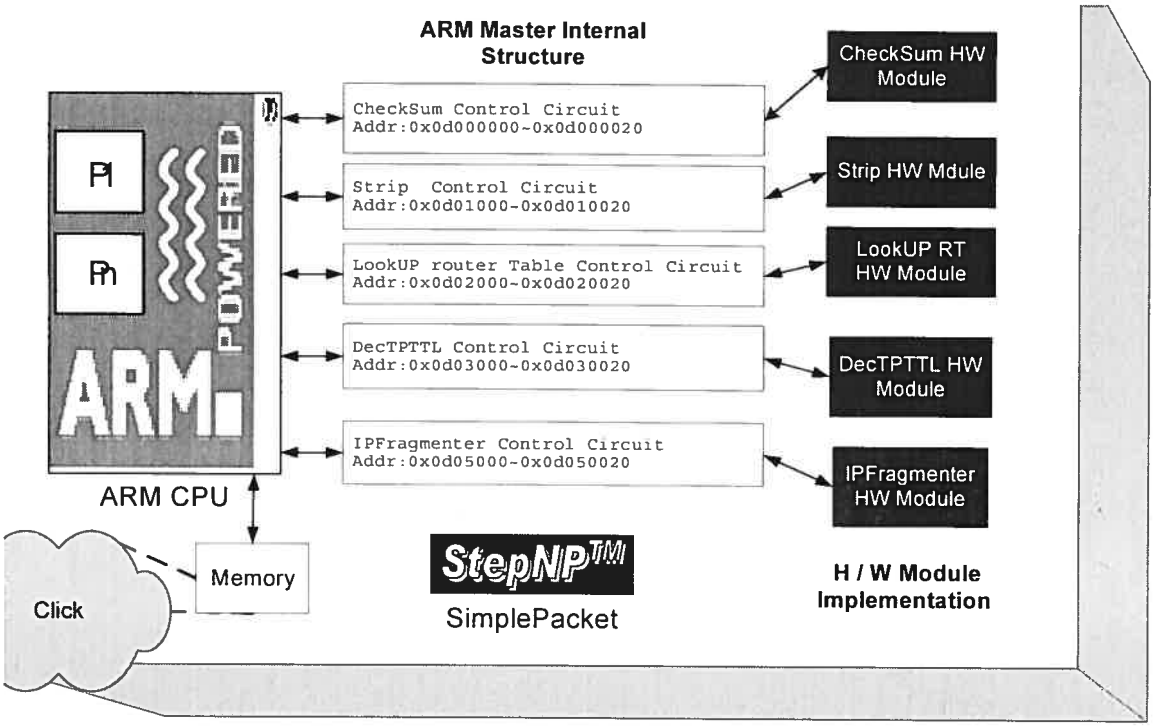


Figure 4-5 Hard/software Co-design for Click router on SimplePacket Platform [12]

2) and memory considerations, the possible parts to move onto the hardware implementation are those that contain much arithmetical computation, occupy large proportion on the whole software elements, not just memory access . For instance, although some parts do occupy a large execution cycles(e.g. ICMPError), we don't move this parts into co-processor for the major operation of this part is just memory transfer (e.g. "memcpy(....)"). Moving this part into co-processor may not lose the advantages of hardware implementation that greatly speeds up complicated mathematical computation.

The **Table 4-1** presents computational complexity of the different elements of the Click IPv4 router, and the complexity is based on and reflected at the rate of each element executing time to

the total time of all elements in the Click IPv4 router. For each element, the executing cycles of operations are counted during simulation. Meanwhile, we use the worst-case executing time for Click IPv4 router to simulate packets processing. For example, the first incoming packet often costs more time to pass the path flow of packets processing than any other packets incoming later and passes as possible as maximum elements inside the IPv4 router. The result of simulation also shows

	Software(each element) cycles	CS cycles	The running cycles of each element / Total running cycles of entire IPv4 router
Classier	356		2,44%
Paint	16		0,11%
Strip	56	44	0,38%
CheckIPHeader	688	320	4,71%
LookupIPRouter	1324	1040	9,06%
DropBroadcasts	32		0,22%
PaintTee	184		1,26%
IPGWOptions	2108	320	14,42%
FixIPSrc	464	324	3,17%
DecIPTTL	232	152	1,59%
IPFragmenter	4228	840	28,92%
EtherEncap	476		3,26%
queue	112		0,77%
ICMPError	4344	640	29,71%
Total elements cycles number	14620		

Table 4-1 the click element computational cycles from simulation

the executable cycles of each element in Click IPv4 router configuration from Strip to ICMPError. Although ICMPError spends the most execution cycles in the entire Click IPv4 router, ICMPError is not on the crucial path of forwarding packets in IPv4 Click router. Only some packets with errors may need to be processed by this element. Moreover, its major function is copying data in memory. Considering the above-mentioned consideration of partitioning, we put our design efforts on the lookup router element (**LookupIPRouter**)as shown in the **Table 4-1**. This part may be accelerated greatly once it is implemented on the coprocessor (see **Table 6-6**).

Given the above two considerations, we are able to identify the critical fragments(presented in **section 4.4** Algorithm analysis) inside the element (highlighted in **Table 4-1**) of Click router, obtain large speedup gain while moving Click application into hardware (see **Chapter 6 Table 6-6**).

4.5 Algorithm analysis

We choose SystemC as a hardware modeling language. The hardware components are implemented according to the algorithms described in the following sections.

1) Build address connection on StepNP platform

First, we need to allocate an address in element class definition. We use DecIPTTL element as an example to explain how to build an address connection between click and simplePacket platform and ensure click to access the DecIPTTL hardware module.

In click we have:

```
#define ADDR_DecIPTTL   0x0d030000

class CSDecIPTTL : public Element { public:

  CSDecIPTTL();
  ~CSDecIPTTL();
 ..............
 ..............

 private:

  uatomic32_t _drops;

(1) static uint* AddrCS = ADDR_DecIPTTL;

(2) void simple_action()
  {
    *(AddrCS+4)=(uint)ip->ip_ttl;    // TTL address in Click IP header
     *AddrCS = (uint)(ip->ip_sum);   // Checksum address in Click IP header
     ip->ip_ttl = *(AddrCS + 4);     // Read from hardware module
     ip->ip_sum = *(AddrCS);         // Read from hardware module

  }
  };
```

Statement (1)(underlined) is a pointer which we add into the old DecIPTTL class. The pointer address is assigned according to the address allocation shown in **Figure 4-5**.

Then, in **statement(2)** we assign the pointer a value with start address of IP header information, thus click can access to HW DecIPTTL HW module in this way [12]. In particular, we observe that the passing packets can choose more than two branches from lookupIPRouter element by routing selection in **Figure 3-3**. We may consider using more than one processors to process the packets out of different branches.

Secondly, the statement(1) in Class CSDecIPTTL must have ***static***, because the CSDecIPTTL elements in two branches are the objects belonging to CSDecIPTTL Class, respectively. Click creates the objects for each branch when configuration initialization. All these objects share the same Class. ***Static*** indicates that this pointer variable is shared by the whole Class. Thus, the address pointer value wont get changed while another new CSDecIPTTL object is defined. Otherwise, if Click is running, and the address pointer value is changed, simulation will produce Click router running error and deadlock.

2) Element Algorithm Analysis and Implementation

CheckIPHeader: The detailed algorithm is described in D.Quinn and M.Hubin's report [12].

DecIPTTL: Click router adopt RFC1624 to update CheckSum described as **Figure 4-6**. DecIPTTL hardware implementation uses the algorithm directly with SystemC modeling language. The detail is shown in **Appendix -J**.

```
        The following "C" code algorithm re-computes the checksum when
                              TTL field has been changed
{     ip->ip_ttl--;
    // 19.Aug.1999 - incrementally update IP checksum as suggested by
    // SOSP reviewers, according to RFC1141, as updated by RFC1624.
    // new_sum = ~(~old_sum + ~old_halfword + new_halfword)
    //         = ~(~old_sum + ~old_halfword + (old_halfword - 0x0100))
    //         = ~(~old_sum + ~old_halfword + old_halfword + ~0x0100)
    //         = ~(~old_sum + ~0 + ~0x0100)
    //         = ~(~old_sum + 0xFEFF)
    unsigned long sum = (~ntohs(ip->ip_sum) & 0xFFFF) + 0xFEFF;
                ip->ip_sum = ~htons(sum + (sum >> 16));
}
```

Figure 4-6 The code description of DecIPTTL element [1]

IPFragmenter: The **Figure 4-7** describes IPFragmenter Algorithm.

From **Figure 4-7** , we can see CheckSum algorithm is called twice respectively. Thus we can replace CheckSum function with HW Checksum module in the following way.

```
……
…..
ip1->ip_sum = 0;
*(AddrCS_CHK + 4) = (uint) ip1->ip_hl;  //
*AddrCS_CHK = (uint) (unsigned char *)(p1->data());
 …..
 …..
 ip1->ip_sum = *AddrCS_CHK;
 …..
 …..
qip->ip_sum = 0;
*(AddrCS_CHK + 4) = (uint) qip->ip_hl;

 …..
 …..

qip->ip_sum = *AddrCS_CHK;
```

In addition, the codes between (1) end (2) in **Figure 4-7** can be implemented by HW IPFragmenter module directly. The **Appendix -I** gives SystemC codes about detail algorithm about IPFragmenter module.

The "C" code algorithm IPFragmenter

```
{
// This algorithm check whether the length of an IP packet is over
//pre-set length by the IP head info. If so, the algorithm will
//separate the IP packet into two fragments within the fixed length.
   (1) IPFragmenter_begin:

    ip1->ip_hl = h1len >> 2;        //assign IP packet length
    ip1->ip_off = ((off - hlen) >> 3) + (ipoff & IP_OFFMASK);
    if(ipoff & IP_MF)    // assign IP fragmentation offset
      ip1->ip_off |= IP_MF;
    if(off + p1datalen < plen)
      ip1->ip_off |= IP_MF;
    ip1->ip_off = htons(ip1->ip_off);
    ip1->ip_len = htons(p1len);
   (2) IPFragmenter end:

    ip1->ip_sum = 0;
    ip1->ip_sum = click_in_cksum(p1->data(), h1len); //re-calculate
    ……                              // the checksum of fragmented packet
    qip->ip_sum = 0;
    qip->ip_sum = click_in_cksum(reinterpret_cast<unsigned char
                    *>(qip), hlen);
}
```

Figure 4-7 The code description of IPFragmenter Element [1]

Strip(N): This element is usually used to remove Ethernet frame header before IP header when N=14 -> Strip(14). The algorithm is described in **Figure 4-8.**

The following "C" code algorithm re-locate IP header's starting position by plus 14 bytes

```
// the following algorithm cut the bytes by the argument, and can be
// used to cut an Ethernet frame head into an IP packet head
static  uint* AddrCS_Strip = (uint*) ADDR_STRIP;
static  uint* AddrTimer_Strip = (uint*) 0x0dd00040;

Packet::pull(uint32_t nbytes)
{
  if (nbytes > length()) {
    nbytes = length();
  } ….
  (1) _data += nbytes;
}
```

Figure 4-8 The code description of Strip element [1]

The algorithm in **Figure 4-8** is implemented by hardware Strip module directly. Statement (1) can be replaced by the following code fragment.

```
*(AddrCS_Strip + 4) = (uint) nbytes;
*AddrCS_Strip = (uint) (unsigned char *)_data;

(*AddrTimer_Strip )=1;      // waiting for calculating result from HW module

_data = (unsigned char *) (*AddrCS_Strip);
```

It should be aware that the algorithm in **Figure 4-8** is running in the Packet Class to protect the passing packets. Due to the encapsulation of C++ objects, we must modify the codes in Packet.cc file directly rather than that in Strip element Class.

LookupIPRouter: Click router uses Linear searching algorithm described in **Figure 4-9**. LinearIPLookup uses a linear search algorithm that may look up every route on each packet. It is therefore most suitable for small routing tables in edged routers.

The linear algorithm was applied in our hardware module design directly. We made some modifications in our hardware module. The algorithm in **Figure 4-9** is separated into three parts,

searching, appending and deleting on the shared routing table. Software click can allocate some spaces for routing table in the memory. In our hardware design, we put the routing table in a hardware component. That means that reading/writing the routing table is concentrated on one chip. We must pay attention on the signal for the synchronization among the processes (e.g. add(..), lookup(..) and del(..)) , when we do the simulation in SystemC. Otherwise, the processor wont find the correct route from the hardware chip. The following **Figure 4-9** illustrates our hardware design methodology about longest matching prefix in routing table.

The following "C" code algorithm describe searching, appending and deleting routing entry in linear routing table

```
bool
IPTable::lookup(IPAddress dst, IPAddress &gw, int &index) const
{
  int best = -1;
  // longest prefix match
  // check in the routing table by comparing destination address
  for (int i = 0; i < _v.size(); i++)
    if (dst.matches_prefix(_v[i].dst, _v[i].mask)) {
      if (best < 0 || _v[i].mask.mask_more_specific(_v[best].mask))
        best = i;
    }

  if (best < 0)         // if not found, return "false"
    return false;
  else {
    gw = _v[best].gw;
    index = _v[best].index;
    return true;
  }
}

void IPTable::add(IPAddress dst, IPAddress mask, IPAddress gw, int index)
{
  // add an IP destination address and related address
  // into the routing table, usually for initializing
  // a routing table of a router
  dst &= mask;
  struct Entry e;
  e.dst = dst;
  e.mask = mask;
  e.gw = gw;
  e.index = index;

  for (int i = 0; i < _v.size(); i++)
    if (!_v[i].valid()) {
      _v[i] = e;
      return;
    }
  _v.push_back(e);
}

void IPTable::del(IPAddress dst, IPAddress mask)
{
  // delete an IP destination address and related
  // address out of the routing table
  for (int i = 0; i < _v.size(); i++)
    if (_v[i].dst == dst && _v[i].mask == mask) {
      _v[i].dst = IPAddress(1);    _v[i].mask = IPAddress(0);
```

Figure 4-9 The code description of lookupIPRouter Element [1]

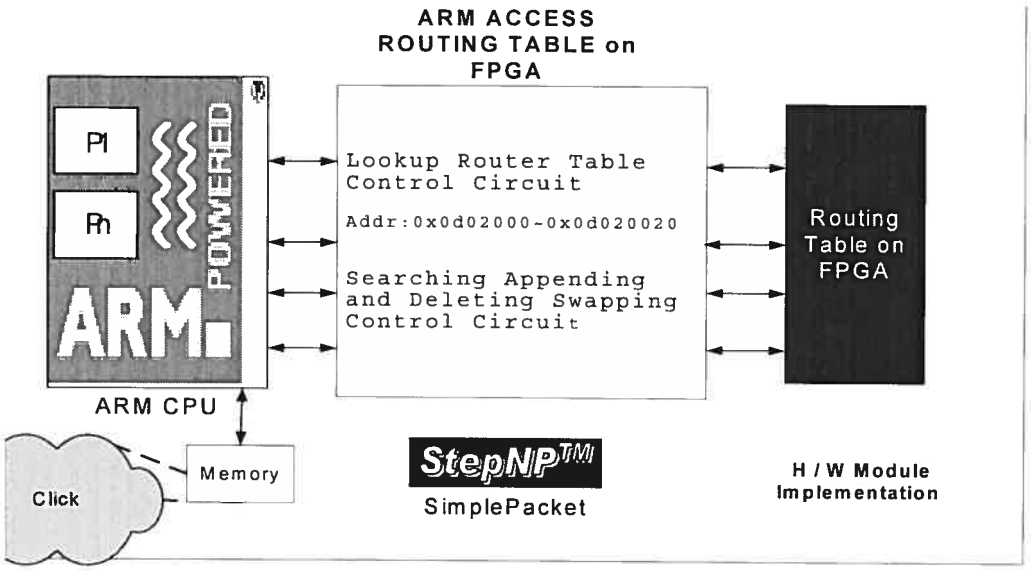


Figure 4-10 The methodology about longest matching prefix in routing table [12]

The detailed steps about searching routing table algorithm are:

1. The IP address entry is appended into the routing table(or forwarding table) on the synthesized component (shown in **Figure 4-10**). when Click initializes configuration with a loop. Then, FPGA module outputs the actual capacity of the routing table to lookup router table control Circuit (shown in **Figure 4-10**). Meanwhile, FPGA module gives a signal indicating the FPGA is ready for searching.
2. When a packet is passing the lookupIPRouter element, ARM CPU sends a startlookup signal to control circuit(CC). When the CC receives the signal, it starts the signal to communicate with the synthesized component and search the network number in the routing `table` on the synthesized compnent in the loop circuit, until finding the next hop.
3. The loop times is determined by the IP network address's location in the synthesized component(shown in **Figure 4-10**) routing table, and capacity of routing table. The component(shown in **Figure 4-10**) output the next hop number(port) to register in ARM kernel.
4. When Click executes read instruction in lookupIPRouter element, the result is return to Click.

Based on the consideration in section **4.4.1 (B),** we can have the left elements - FixIPSrc(), IPGWOptions() and IPFragmenter() - share the same CheckSum hardware module. Thus, the whole IPV4 router configuration may perform the packet processing with the combination of hardware and software components in co-design.

Chapter 5

The Methodology to Implement Click Router on Multiprocessor

5.1 General presentation of the methodology

This section presents an overall picture of the methodology about how to put Click router onto the multiprocessor and the advantage of this methodology.

The methodology involves in three parts: "Click.exe" source code, StepNP components and Click elements

1) By modifying on "Click.exe" , we ensure that multiple processors may perform the different task of processing packets concurrently. Section 5.2 will explain this issue in detail.

2) By improving the StepNP existing processor components, we implemented that three ARM processors can access(read/write) the shared lock for transferring packets data between the three processors. Section 5.3.1 will introduced how to modify the processor components in detail.

3) By developing a few new Click elements, we build the connections between different configuration files executed on different processors. Section 5.3.2 will present such implementation in detail.

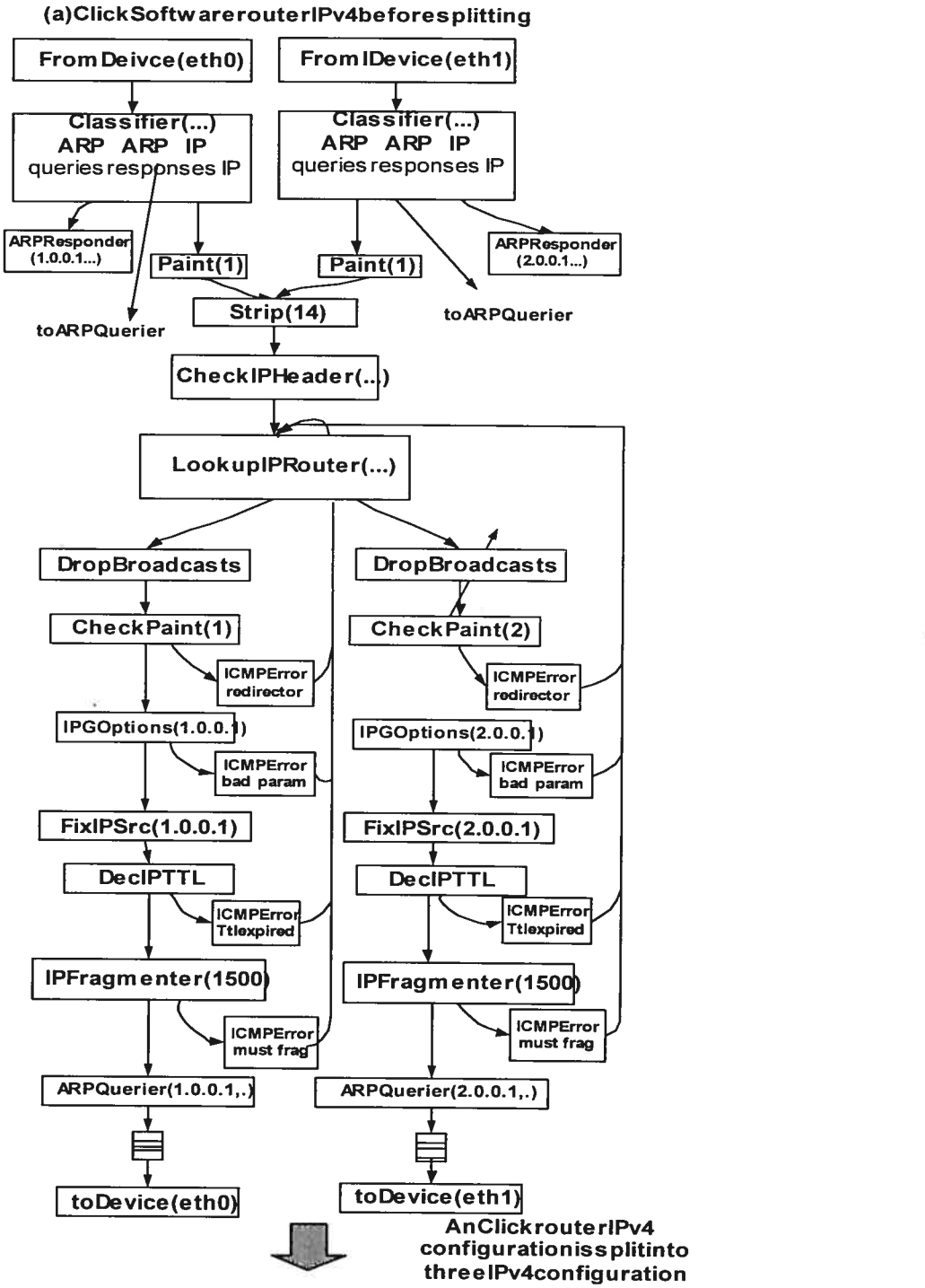
5.1.1 The architecture description

Click modularity allows to process packets ***pipelining*** over several configurations that are executed on multiple processors, implementing **configuration-level pipeline** (defined in **Chapter 3.3**) [28].Thus, we may take advantage of multiple processors to process packets in parallel without modifying Click source code. The methodology also may be flexible to split or merge passing flow by the number of processors and use the processors resources efficiently.

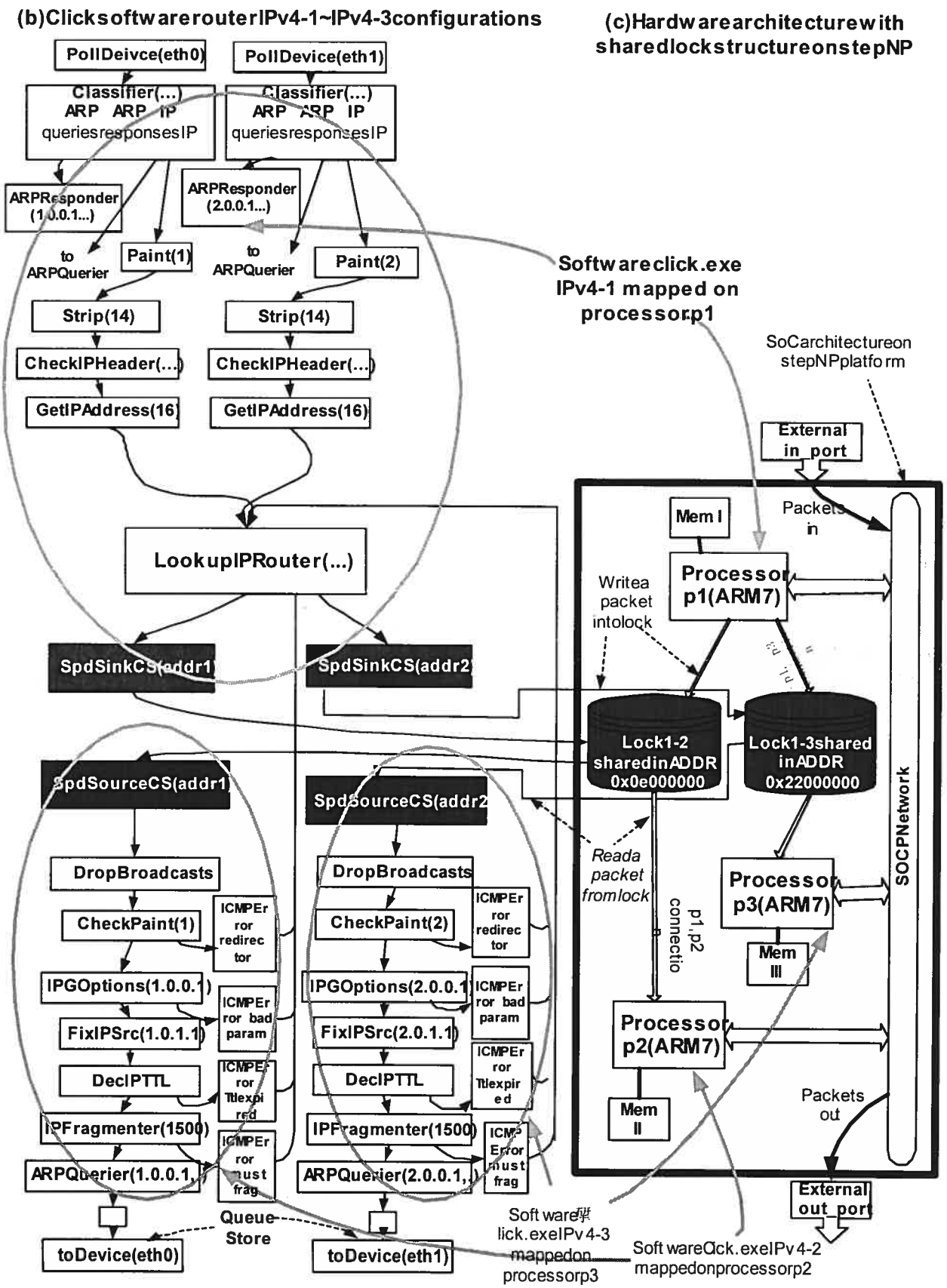
We use Click configuration for IPv4 Router as an example to present how the system-level design methodology works on multiprocessor architecture. **Figure 5-1** is the overall picture of an

executable application-specific multiprocessor architecture including hardware components and software applications

Figure 5-1 the general description on how Click works on three ARM processors



Inmymethodology,Clicksoftwarerouterrunningonthree ARM7processors respectivelywithdifferentIPv4configurationparametersdepictedasfollows:


(b)Clicksoftwarerouter IPv4-1~IPv4-3configurations
(c)Hardwarearchitecturewith sharedlockstructureonstepNP
PollDeivce(eth0)
PollDevice(eth1)
Classifier(...) ARP ARP IP queriesresponsesIP
Classifier(...) ARP ARP IP queriesresponsesIP
ARPResponder (1.0.0.1...)
ARPResponder (2.0.0.1...)
to ARPQuerier
Paint(1)
to ARPQuerier
Paint(2)
Strip(14)
Strip(14)
CheckIPHeader(...)
CheckIPHeader(...)
GetIPAddress(16)
GetIPAddress(16)
LookupIPRouter(...)
SpdSinkCS(addr1)
SpdSinkCS(addr2)
SpdSourceCS(addr1)
SpdSourceCS(addr2)
DropBroadcasts
DropBroadcasts
CheckPaint(1)
CheckPaint(2)
ICMPError redirector
IPGOptions(1.0.0.1)
IPGOptions(2.0.0.1)
ICMPError bad param
FixIPSrc(1.0.1.1)
FixIPSrc(2.0.1.1)
DecIPTTL
DecIPTTL
ICMPError Ttlexpired
IPFragmenter(1500)
IPFragmenter(1500)
ICMPError must frag
ARPQuerier(1.0.0.1,...)
ARPQuerier(2.0.0.1,...)
Queue
Store
toDevice(eth0)
toDevice(eth1)
Softwareclick.exe IPv4-1 mapped on processorp1
SoCarchitectureon stepNPplatform
External in port
Packets in
Mem I
Processor p1(ARM7)
Writea packet intolock
Lock1-2 sharedinADDR 0x0e000000
Lock1-3shared inADDR 0x22000000
Reada packet fromlock
p1,p2 connectio
Processor p3(ARM7)
Mem III
Processor p2(ARM7)
Mem II
Packets out
SOCPNetwork
External out_port
Softwareclick.exeIPv4-3 mappedon processorp3
Softwareclick.exeIPv4-2 mappedonprocessorp2

In **Figure 5-1**, we set configuration file ipv4-1 on processor p1, configuration file ipv4-2 on processor p2, configuration file ipv4-3 on processor p3. Thus, each configuration can process packets on *p1, p2*, and *p3* concurrently. By setting two shared memories among three processors, we have created the dedicated hardware components to support transferring packets. In addition, we have designed an element to write incoming packets to the shared memories and two elements to read the packets out of the shared memories. The internal operations of three processors have been modified to perform the task concurrently and to implement pipelining on multiple processors. The method may be considered to implement configuration in pipeline[28].The system-level design of the methodology is simulated and evaluated on StepNP platform.

5.1.2 The benefit of the methodology

The model of the methodology has two advantages on implementing Click on multiprocessors.

1) the model reduce the latency for incoming packets to wait for process. Given the Click processing mechanism (see the feature 3 in **Chapter 4.3.2**), only one packet is being processing in Click router each time on one processor mechanism. This methodology may process three (or more) packets simultaneously in three processors, shortening the waiting time for incoming packets. Considering the availability of hardware resources, we may take the methodology for a feasible way to improve the performance of Click router.

2) The architecture model is scalable meaning that users may easily increase or reduce the number of processors to packet processing path, implementing multiple Click configurations in parallel. From the point of LookupIProuter operation, users may easily add or modify Click configuration files, changing the number of the path to process packets according to the number of Ethernet card in the router system.

To ensure three processes (e.g. Click.exe) work, we must consider two questions:

A. How to map the Click .exe program on three ARM processors, and ensure the three processors execute the same Click.exe program simultaneously.

B. How to communicate between the three ARM processors.

5.2 Mapping the program on three processors

First, according generic multiprocessors SoC design, the application-specific is determined as Click router software named "click.exe" binary code for ARM processors.

The Click elements performing subtasks of packet processing make up a packet path in Click configuration router. It is "click.exe" program that loads each element and executing the element code along the packet path, processing incoming packets.

Then, like the process partitioning (defined in **Chapter 3**), the IPv4 configuration is divided into three sub configurations since Click elements of configurations may be split and re-assembled into various configurations according to different packets operations. In **Figure 5-1**, sub configurations are *ipv4-1, ipv4-2, ipv4-3*.

The configuration file is the parameter of "click.exe" according to Click router [1]. Since original "click.exe" program does not support more than one configuration files running on different processors, we have to modify "Click.exe" program so that the same "click.exe" may run on the multiple processors and load different configuration files for different processors. Thus, three sub-configurations (ipv4-1, ipv4-2, ipv4-3) perform different IP packets processing tasks according to the elements collection in the three configuration files. The modified code segment in Click router is described in **Figure 5-2.**

In addition, we have to do some modification on StepNP components to support the building of three processors architecture. With the guide of application-specific multiprocessor SoC design flow, we choose major following components from StepNP library to construct the architecture with three processors. Three ARM7 processor components (p1, p2, p3) is rewritten according to user's operation requirements to processors in StepNP platform. The three improved ARM7 processors are modified as **Figure 5-3-1**.

The Modified "Click.exe" program

1) Click.exe modification

```
int main(int a, char **av) {
#endif
// Ask to the coprocessor

 uint* AddrCS;
 AddrCS = (uint*) 0x0d000000;
 unsigned result=(unsigned)(*AddrCS);
 printf("result== %d\n",result);

 int argc = 2;
 char *argv[] = {
    "click",
    "armrouter.click",
    ""
  };
// else
// change the configuration file

 if (result == 1) {
  argv[0] = "click";
  argv[1] = "sidlPush-1.click";     // configuration file name -1
  argv[2] = "";
  };
 if (result ==2){
  argv[0] = "click";
  argv[1] = "sidlPush-2.click";     // configuration file name -2
  argv[2] = "";
    };

 if (result ==3){
  argv[0] = "click";
  argv[1] = "sidlPush-3.click";     // configuration file name -3
  argv[2] = "";
   };
 printf("starting click main...\n");

....
....
....
```

Figure 5-2 Click router code modification for multiprocessor

The p1,p2 and p3 ARM Processors modification

```
// the following code tells ARM processors to run different Click
//routers in simulation on StepNP platform

class MyArm1 : public MT_ARM {
ArmData memRead(ArmAddr address){

if(address >= 0x0d000000 && address < 0x0d000020) {
         printf("Demande la reponse du AEM processor-1 \n");
            return (uint)1;
}
}

class MyArm2 : public MT_ARM {
ArmData memRead(ArmAddr address){

if(address >= 0x0d000000 && address < 0x0d000020) {
         printf("Demande la reponse du AEM processor-2 \n");
            return (uint)2;
}
}

class MyArm3 : public MT_ARM {
ArmData memRead(ArmAddr address){

if(address >= 0x0d000000 && address < 0x0d000020) {
         printf("Demande la reponse du AEM processor-3 \n");
            return (uint)3;
}

}
....
```

Figure 5-3-1 ARM Processor modification in StepNP

A single bus with SOCP protocol connects three processors. The instance of architecture is depicted in **Figure 5-3-2.**

After modifying "Click.exe" and ARM processor components, we succeeded in mapping ipv4-1 on p1, ipv4-2 on p2, ipv4-3 on p3 (shown in **Figure 5-3-2)**. Each configuration can process packets on *p1*, *p2*, and *p3* concurrently.

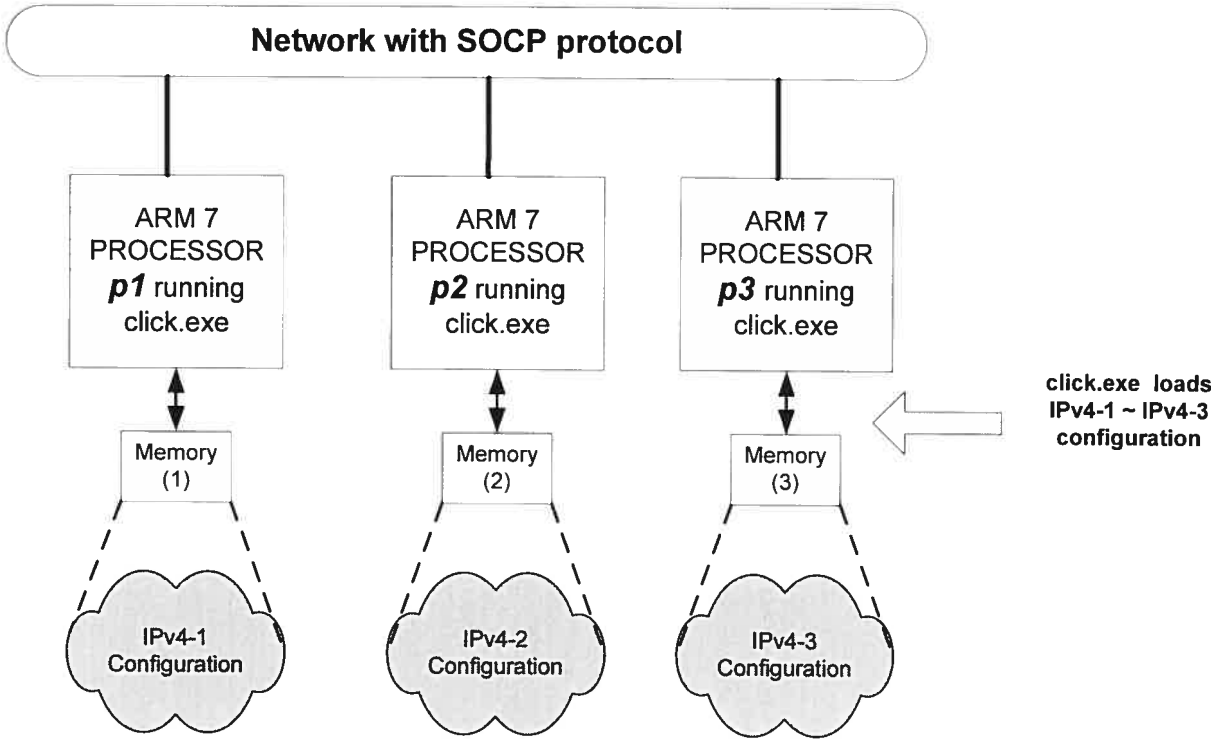


Figure 5-3-2 the instance of Click on three ARM7 Processors architecture

5.3 Exchanging data between three processors

After a processor completes processing incoming packets processing, it will transfer the packets to one of its two down flows (depicted **Figure 5-1**) along the passing path depending on the routing result. How to synchronize the communication between two processors raises concern in Click users. For this question, we can find a solution from ***hardware*** and ***software*** aspects respectively.

5.3.1 Hardware architecture building

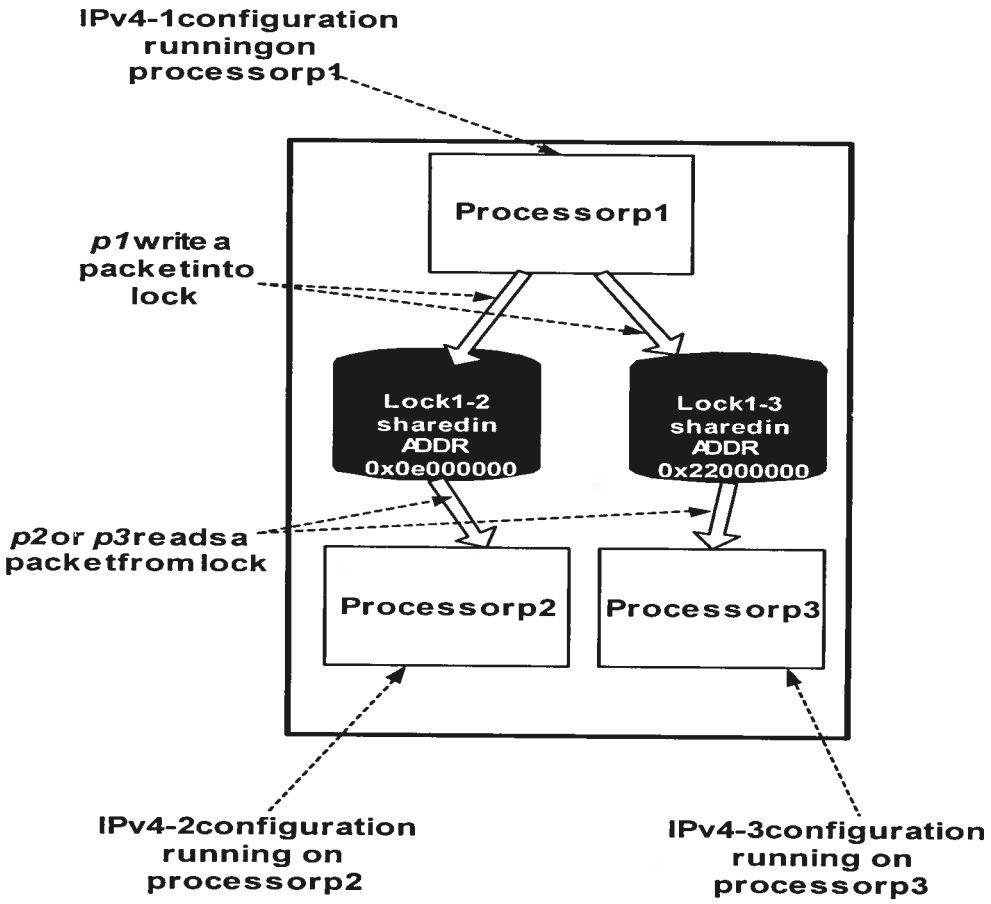
The hardware solution is to set two individual shared structures called locks to buffer the packet between two processors. The capacity of a lock is big enough to hold a packet. Each lock is set up a single address space that programmers should be offered for designing the architecture. In my experiments, we set the two locks` address with 0x0e000000 and 0x0e0000020 respectively, depending on the available address spaces to users on StepNP platform.

For example,

Address(lock-1-2) = 0x0e000000; Address(lock-1-3)=0x22000000;

Figure 5-4 describes the detail structure.

Figure5-4 Hardware architecture with shared lock structure on stepNP



In addition, we must modify the internal write/read operations of processor components. Thus, processors *p1, p2* may access the lock structure to perform data exchange. Because existing StepNP does not contain such components to support the behaviour of processor *p1, p2* to finish the data transfer described in Figure 5-4, we did some modification on StepNP platform code as follows,

The processor p1 is modified as in **Figure 5-5-1**;

The processor p2 is modified as in **Figure 5-5-2**;

The processor p3 is similar to p2 (refer to **Figure 5-5-2**);

After rewriting component, we can see that processor p1 will check whether the shared hardware structure is lock by read() function. If not, write() function in processor p1 will lock the shared structure and write the packet into the hardware structure.

The relevant "C" code algorithm Component Processor 1

```
{
   // the following shows how to access(R/W) the shared lock structure
   // on the multiprocessor architecture in the simulation on StepNP
    ....

   (1) Key segment begin_begin:

     ArmData memRead(ArmAddr address){
      if(address >= 0x0d010000 && address < 0x0d010020) {

          printf("Read ARM-1 SPD_Flag --1 \n");
           return (uint)spd_ARM1_ARM2.inReady;

       }
      if(address >= 0x0e000000 && address < 0x0e000020) {

          printf("Read ARM-1 SPD dataIn and lenIn --1 \n");

          if (address == 0x0e000000) {
              return (uint)spd_ARM1_ARM2.dataIn[0];
          }
       else
            return (uint)spd_ARM1_ARM2.lenIn;
       }
     void memWrite(ArmAddr address, ArmData data) {
        if(address >= 0x0e000000 && address < 0x0e000020) {

          printf("writing SPD DataIn and lenIn --1 \n");

          if (address == 0x0e000000) {
            spd_ARM1_ARM2.dataIn[hLen] = data;
           else {
            if (address == 0x0e000004) spd_ARM1_ARM2.lenIn = data;
            if (address == 0x0e000008) hLen = data;
           } }
       else if(address >= 0x0d010000 && address < 0x0d010020) {

         printf("locking/unlocking SPD shared structure \n");

         spd_ARM1_ARM2.inReady = data;
       }
(2) The key segment end:
    ....
    ....
}
```

Figure 5-5-1 The internal operations of an ARM processor

The relevant "C" code algorithm Component Processor 2

```
{

   // the following shows how to access(R/W) the shared lock structure
   // on the multiprocessor architecture in the simulation on StepNP
   ....

   (2) Key segment begin_begin:
    ArmData memRead(ArmAddr address){

       if(address >= 0x0e000000 && address < 0x0e000020) {

         if (address == 0x0e000000) {
            len =lenIn-hLen;
            hLen--;
       //   printf("Read ARM-2 SPD dataIn --2\n ");
            return (uint)spd_ARM1_ARM2.dataIn[len];

          }
        else {
            //printf("Read ARM-2 SPD lenIn --2 \n");
            hLen = (uint)spd_ARM1_ARM2.lenIn;
            lenIn = hLen;
            return (uint)spd_ARM1_ARM2.lenIn;
          }
       }
....

       else if(address == 0x0d010000) { // && address < 0x0d010020) {

        // printf("Demande la reponse du SPD_Flag lock1-2 \n");

           return (uint)spd_ARM1_ARM2.inReady;
         }
    }

    void memWrite(ArmAddr address, ArmData data) {

      // printf("wrote flag to address\n");

        if(address == 0x0d010000) && (address < 0x0d010020) {
        //    printf("writing SPD inReady Flag lock1-2 \n");

              spd_ARM1_ARM2.inReady = data;

        }
(2) The key segment end:
    ....
}
```

Figure 5-5-2 The internal operations of an ARM processor

5.3.2 Software (Element)

To prevent the write/read conflict to the lock, accessing the lock must respect the locking policy. Since all storages are explicit in Click, those two locks are actually underlying hardware and Click applications doesn't provide elements to access those two lock structures explicitly. Therefore, we must write two new specific elements to push packets into the lock and to pull packets from the lock, supporting locking policy.

Two new elements are the following:

SpdSinkCS: contain the implicit behaviour to write a packet into shared lock structure.

SpdSourceCS: contain the implicit behaviour to read a packet from shared lock structure.

SpdSinkCS:

SpdSinkCS(...)is a push element("push" is explained in **Chapter 4.2.1**).

The key segment of SpdSinkCS for writing is described in **Figure 5-6**:

```
(1) AddrCSChip1 = (uint*)0x0e000000
(2) AddrCSFlag = (uint*) 0x0d010000
```

In line (1), address point AddrCSChip1 holds the address for lock-1-2;

In line (2), address point AddrFlag holds a value to indicate whether lock-1-2 is occupied or released;

SpdSinkCS lock or unlock the shared memory by setting the flag with `1` or `0`.

If lock-1-2 is being occupied, processor p1 won't write a packet into to the shared structure.

SpdSourceCS:

SpdSourceCS(...)is pull element("pull" is explained in chapter 4.2.1).

The key segment of SpdSourceCS for writing is described in **Figure 5-7**:

```
(1) AddrCSChip1 = (uint*)0x0e000000
(2) AddrCSFlag = (uint*) 0x0d010000
```

The relevant "C" code algorithm SpdSinkCS

```
{
   ....

   (1) Key segment begin_begin:

  AddrCSFlag = (uint*) 0x0d010000
  AddrCSChip1 = (uint*)0x0e000000
 *AddrCSFlag = 0;

  void SpdSinkCS::push(int, Packet * p)
 {

  if (p != 0) {

    unsigned flag_push = (unsigned) (*AddrCSFlag);

   //check whether shared lock structure is available. if not, wait..
    while (flag_push == 1){
      printf("AddrCSFlag-1: %d\n",flag_push);
      flag_push =(unsigned) (*AddrCSFlag);
    };

  // fill the packet into the shared lock structure
  if ( flag_push == 0) {
    int len = (int)p->length();
    *(AddrCSChip1+1)= len;
    unsigned char* dataIn_push = (unsigned char*)p->data();

    for (int i =0; i<len;i++){
      *(AddrCSChip1+2) = i;
      *AddrCSChip1 = *dataIn_push;
      dataIn_push ++;
    };

    *AddrCSFlag = 1;       // set up inReady =1 telling shared locked
                           // structure is not available now
   ....
   }

    p->kill();
  }
}

(2) The key segment end:
    ....
    ....
}
```

Figure 5-6 the segment code to write packets into the shared hardware on StepNP platform

In line (1), address point AddrCSChip1 holds the address for lock-1-2;

In line (2), address point AddrFlag holds a value to indicate whether lock-1-2 is occupied or released;

SpdSourceCS lock or unlock the shared memeory by setting the flag with `1` or `0`.

Processor p2 won`t take reading operation to shared structure until lock-1-2 is released.

The relevant "C" code algorithm SpdsourceCS

```
{
   ....
   (1) Key segment begin_begin:

   AddrCSFlag = (uint*) 0x0d010000
   AddrCSChip1 = (uint*)0x0e000000
  *AddrCSFlag = 0;

   Packet * SpdSourceCS::pull(int)
{
   struct SPD &spd = *(SPD *) _addr;
   spd.inReady = (int)(*AddrCSFlag);

   //check whether shared lock structure is available. if not, wait..
   while (spd.inReady != 1){
      printf("AddrCSFlag-spdsource: %d\n",spd.inReady);
      spd.inReady =(int) (*AddrCSFlag);
   };

   // read the packet out of the shared lock structure
   if (spd.inReady ==1 ){
      spd.lenIn = (int)*(AddrCSChip2+1);
      if(spd.lenIn < 0 || spd.lenIn >= SPD_MTU) return 0;

      for (int i=0; i<spd.lenIn;i++){
        spd.dataIn[i]=(unsigned char) *AddrCSChip2;

      };

      WritablePacket *p = Packet::make(spd.dataIn, spd.lenIn);
      memcpy(p->data(), spd.dataIn, spd.lenIn);

                *AddrCSFlag = 0; // release lock structure
   ....
      return p;
   }
}
(2) The key segment end:
     ....
}
```

Figure 5-7 the segment code to read packets from the shared hardware on StepNP platform

After completing the design of software and some hardware components, the original IPv4 router may be changed into new one that described in **Figure 5-8**.

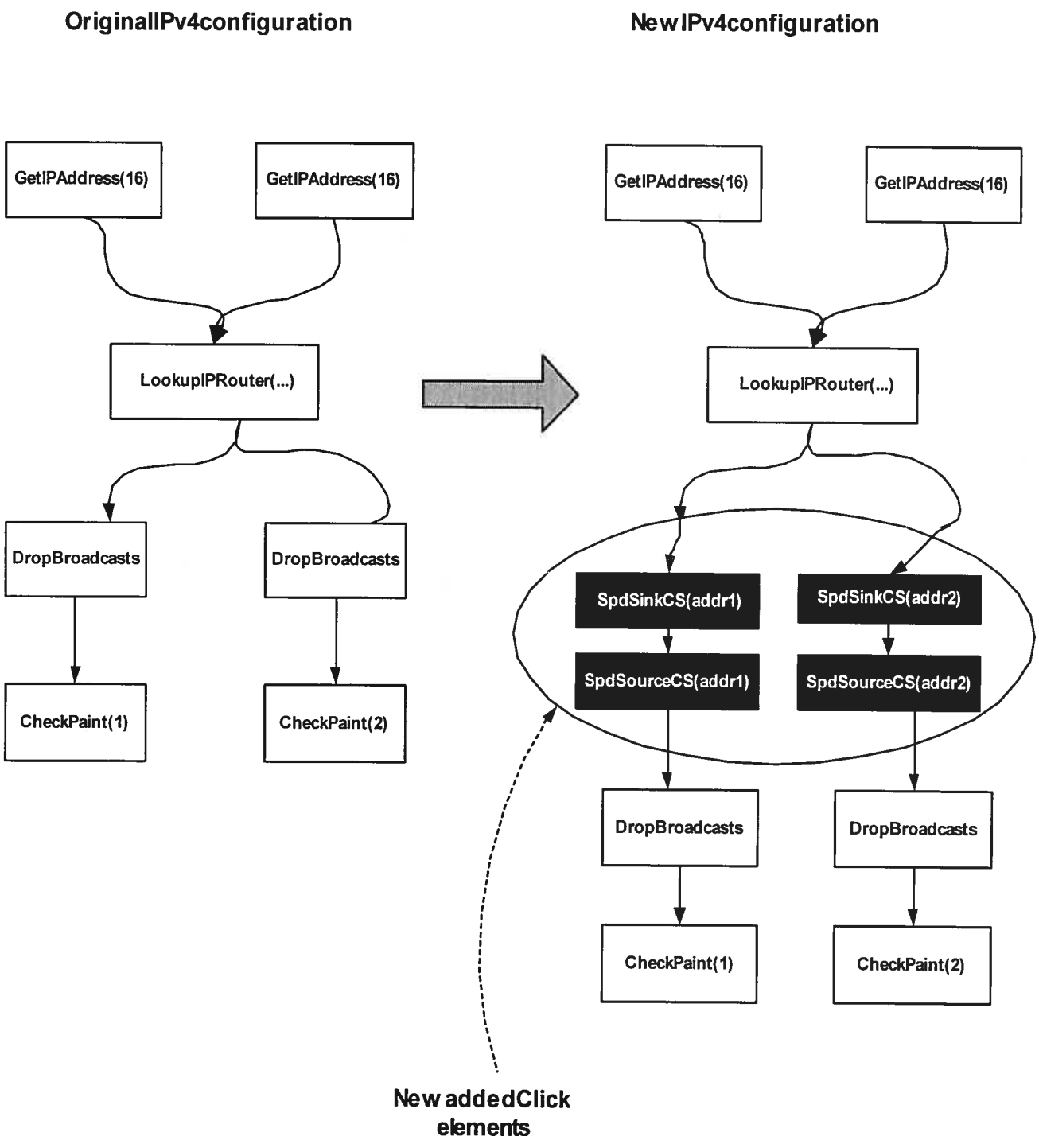


Figure 5-8 IPv4 original file and IPv4 configuration drawing with two new elements

5.4 Overall description of communication between processors

In general, **Figure 5-9** introduces how to exchange data (e.g. packets) between two processors(*p1* to *p2*, or *p1* to *p3*).

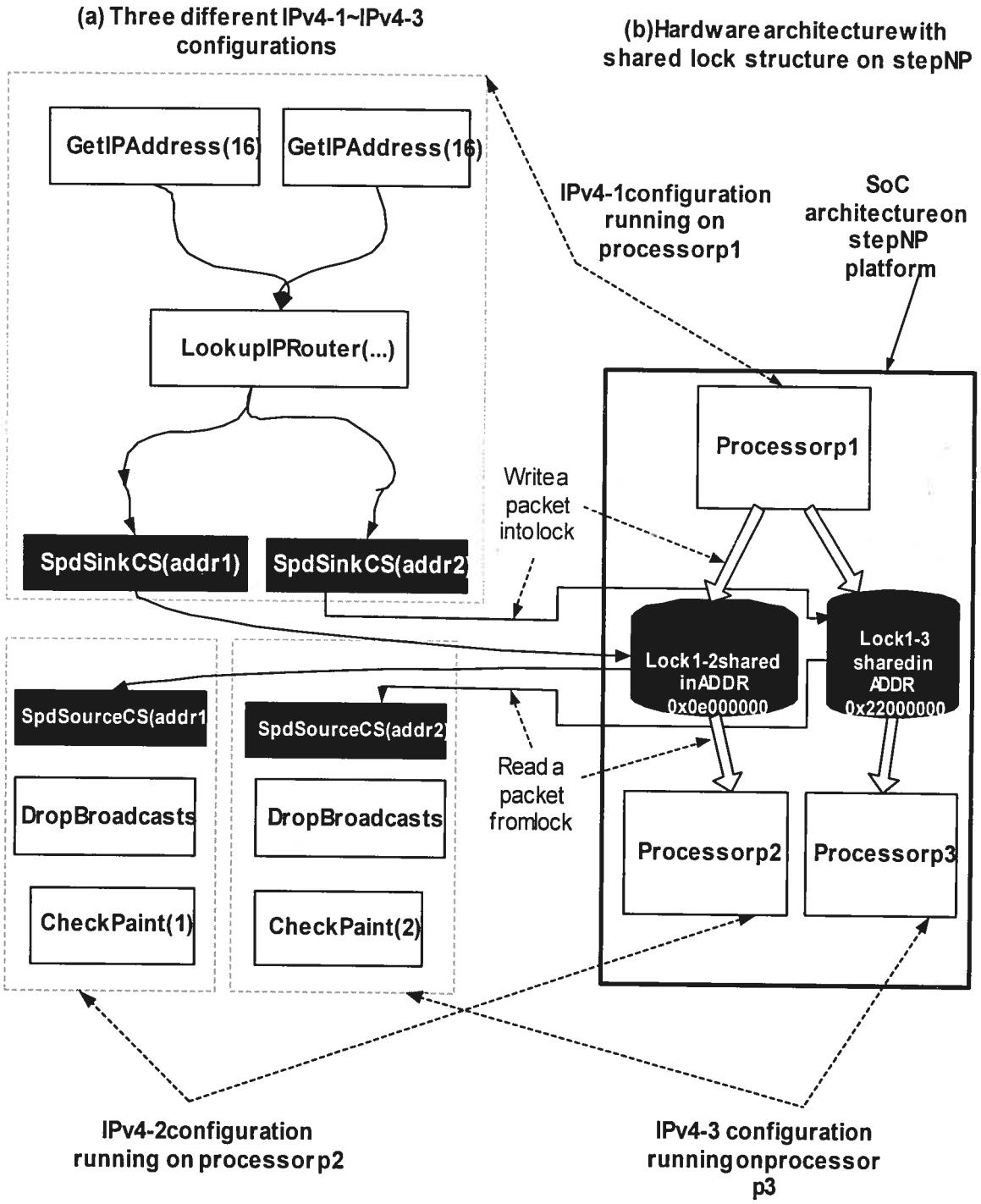


Figure 5-9 The communication between three processors with the help of software & hardware

In **Figure 5-9** after the routing by element "lookuprouter", the packet chooses a destination address to go. Thus, a packet is transferred to element SpdSinkCS element as an incoming packet. There are two choices at this moment:

1) If the packet chooses "addr1" path, element SpdSinkCS(addr1) running on processor p1 must check whether the hardware lock-1-2 is occupied by reading operation. processor p1 must check and wait repeatedly until lock-1-2 is released. Once unlocking, processor p1 writes the packet into the lock and continue to process other incoming packets by executing the element such as "Classfier(...)" in ipv1 flow. At the moment, if processor p2 is performing the SpdSource element code and the lock has the packet, p2 will read the packet out of lock-1-2 and continue to transfer the packet into element "DropBroadcasets". Thus, processor p1 and p2 may perform packets processing in pipeline, executing different configuration ipv1, ipv2.

2) If the packet chooses "addr2" path, element SpdSinkCS(addr2) running on processor p1 must check whether the hardware lock-1-3 is occupied by reading operation. processor p1 must check and wait repeatedly until lock-1-3 is released. Once unlocking, processor p1 writes the packet into the lock and continue to process other incoming packets by executing the element such as "Classfier(...)" in ipv1 flow. At the moment, if processor p3 is performing the SpdSource element code and the lock has the packet, p3 will read the packet out of lock-1-3 and continue to transfer the packet into element "DropBroadcasets". Thus, processor p1 and p3 may perform packets processing in pipeline, executing different configuration ipv1, ipv3.

By this way, processor p1, p2 and p3 may execute their own task respectively in parallel, and transfer the packet in pipeline.

In addition to the above-mentioned supports, StepNP platform can execute software code on cycle-*accurate instruction-set-simulator(ISS)* [27] based on standard RSIC processors and use *SystemC Open Protocol(SOCP)* as communication [6] between components. All the code is written in SystemC language (defined in **Chapter 4.1**). Thus, after designing the software for multiprocessor architecture and building a *cycle-accurate executable architecture* [27], we can start system-level simulation to verify whether the processors on architecture can be running in parallel correctly. While hardware parts (e.g. ARM processor, memory, channel, etc.) of the architecture can be

modeled in SystemC, Click router software parts may be executed in StepNP (ISS) core to support SoC validation on system-level [27] and to get the evaluation result of performance for the overall multiple processors architecture.

Chapter 6

Performance Evaluation

With the support of StepNP simulation platform, we get the evaluation result to estimate the cost of the design quality such as execution time by cycle, to validate whether my methodology is possible, and to calculate the overall Click IPv4 acceleration with the help of co-processors, shortening the design process in HW/SW co-design.

Performance evaluations mainly consist of two sections:
6.1 hardware/software partitioning on one processor with ASIC hardware (described in **Chapter 4**);
6.2 the evaluation of IPv4 router in application-specific SoC three processors SoC architecture (described in **Chapter 5**).

6.1 Evaluation of the hardware/software co-design methodology in Chapter 4

This section introduces the result of hardware/software partitioning methodology implemented on Click IPv4 router and the evaluation way on how to get such result.

6.1.1 Synthesis tools

This section introduces the tools for simulating and synthesizing the HW/SW design.
1) Software simulation: we choose STMicroelectronics StepNP platform for developing and testing our design.
2)Hardware Module: we choose Synopsys CoCentric SystemC Compiler to synthesize a behavior design.

6.1.2 Initial constraints for synthesis

```
bc_enable_analysis_info  =  "true"
bc_multicycle  = "false"
cycle period = "25ns"
IO module = "superstate"
effort module = "low"
```

6.1.3 Synthesis result

CoCentric SystemC Compiler presents the reports about resources, time and FSM status to four hardware module. (CheckSum module see D.Quinn and M.Hubin `s report [12]).

bc_shell> create_clock clk –p 25

Cycles period is 25 ns.

bc_shell> reprt_schedule –abstrac_fsm > csdecttl.fsm

We used the above command to get FSM report.(see Appendix A-J).

The statistics results are described in **Table 6-1.** The detail information is shown in the Appendix A-J.

Hardware Module	Cycles	Status (FSM)	Estimated Area			Resource	
			Combination	Sequential	Totale	ADD/SUB/CMP	Operation (1bits)
ccstrip	1	3	188	733	921	1/0/0	32
csipfragp	2	4	2427	1614	4042	1/1/1	32
csdecipttl	1	3	247	565	812	3	8-16
csiptable	1	6	6396	55882	62278	1	32

Table 6-1 Result from CoCentric SystemC Complier [12]

6.1.4 Performance analysis by result

We use the evaluation methodology which is described in D.Quinn and M.Hubin `s report [12]. We set breaking points in the elements and calculated the Cycles number aided by StepNP platform. Thus, we obtained a group of figures to illustrate the running cycles comparison between software and hardware as follows (except CheckSum element explained in D.Quinn and M.Hubin`s report [12]) from the simulation.

Then, we supposed Click software running on ARM CPU 150MHZ.

Therefore, the period of a cycle for HW and SW is different.

Software period/cycle = 6.6 ns

Hardware period/cycle = 25 ns

Meanwhile, we suppose that an instruction executes in one cycle. Thus, we get the following result about software/hardware for different HW module.

6.1.5 Software/hardware analysis

	Strip Module	Total(including call fonctions)	Actual in Element
Hardware	2	6	4
Software	21	44	44

Table 6-2 Cycle Number in Strip module execution[12]

	DecIPTTL Module	Total(including call fonctions)	Actual in Element
Hardware	2	5	3
Software	133	152	152

Table 6-3 Cycle Number in DecIPTTL module execution[12]

	IPFragmenter Module	Total(including call fonctions)	Actual in Element
Hardware	3	13	10
Software	93	200	200

Table 6-4 Cycle Number in IPFragmenter module execution[12]

In **Table 6-2~ Table 6-4,** the hardware cycles for DecIPTTL, IPFragmenter and Strip get from CoCentric SystemC Complier.

From above figure we must take notice two points:

1. The period for HW cycle is more than that for Software. Now We must consider how to make HW module and SW code to run in parallel. To avoid software code's waiting for reading result from hardware, we can bring writing data into hardware forward and keep write/read HW some code length, then let CPU execute other necessary codes (CPU must execute them) in the same element. After a CPU finishes executing other codes, it begins to read the results from hardware module. In this way, we can avoid latency for waiting the results of hardware module.

2. When we evaluate the hardware performance, we must consider the cycles for writing instruction to HW, we suppose one cycle for this step. **Figure 6-1** illustrates the two above-mentioned points:

```
/*
    *(AddrCS + 1) = hllen;
    *(AddrCS + 2) = off;
    *(AddrCS + 3) = hlen;
    *(AddrCS + 4) = ipoff;
    *(AddrCS + 5) = plen;
    *(AddrCS + 6) = pldatalen;
    *(AddrCS) = pllen;

  .......                  // Between this code segment, other
 ......                   // Click router instructions are executed, implementing
 ....                     // running with Hardware component(or ASICs) in parallel

    ip1->ip_sum = 0;

    ip1->ip_hl = *(AddrCS + 1);
    ip1->ip_off = *(AddrCS + 2);
    ip1->ip_len = *(AddrCS + 3);
   */
```

Figure 6-1 The code segment in IPFragment element

As for the routing table, one cycle is for searching, the capacity of routing table array is defined to 20 in our testing.

We suppose the worst case is to parse the whole data array and the first packet always cost the most time to look up the output port in a routing table. Thus we get the hardware cycle for searching routing table is 25 cycles.

	Routing table Module	Total(including call fonctions)	Actual in Element
Hardware	25	30	5
Software	926	1040	1040

Table 6-5 Cycle Number in Lookup Routing Table module execution[12]

6.1.6 Click IPv4 router performance in co-design

We use Amdhal formula [12] to calculate our acceleration by HW/SW co-design. This formula is applied to measure to extent which an co-design methodology improve the software.

$$Gain = \frac{1}{(1-p) + p/acc}$$

p : Instructions implemented by HW / Instructions in Total Elements (1%)

acc : Instructions before using HW / Instructions after using HW *(100)*

Gain: Evaluation parameters for Performance Improved

Normally, the more improved proportion in software, the more the Gain. The bigger the Gain, better the performance for the software.

For example, from **Table 6-5** (LookupIPRouter), we can get :

```
Acc = 1040/5 = 208
Gain = 4.58
```

The **Table 6-6** table is the speedup statistic and overall speedup for the whole IPv4 Router.

Elements in IPV4 Router	Gain
Classier	1.00
Paint	1.00
Strip	3.50
CheckIPHeader	1.85
LookupIPRouter	4.58
DropBroadcasts	1.00
PaintTee	1.00
IPGWOptions	1.18
FixIPSrc	3.24
DecIPTTL	2.76
IPFragmenter	1.05
EtherEncap	1.00
queue	1.00
ICMPError	1.17
Average	1.82~1.87

Table 6-6 Improved status for each element in IPv4 Click router

Therefore, we can see that the performance for overall Click IPv4 router has been improved 1.82-1.87 times as much as the old one after using the methodology (mentioned in **Chapter 3**). The actual performance is more than that shown by data normally. Because Click router can be applied as the NPU in each network card. This may be called Distributed Router Architecture[3]. An normal packet is usually passing time-critical which includes Strip, CheckIPHeader, LookupIPRouter, DecIPTTl,ARPQuerier and Queue elements. In this path, routing table searching is an time-consuming packet forwarding process which is a bottleneck to seriously lower the performance of an router. By co-design methodology, we can see searching routing table(RB) get the most improvement from the *Gain* for LookupIPRouter element in **Table 6-6**.

Non-time critical path includes IPGWoptions, FixISrc and IPFragmenter elements [3] which cost much instruction cycles to transfer data in memory. If a packet doesn't pass these elements, the Gain will increase. Thus the performance of IPv4 Click router shows much higher in most time. The methodology has disadvantage, however, as the lookupIPRouter elements, the hardware consuming time for searching in routing table is too long(25cycles). The cycles between write and read hardware module is about 10 hardware cycles(25ns/cycles). **Table 6-6** shows that hardware cost 25 hardware cycles for searching in routing table by our co-design methodology. Therefore, the methodology needs to be improved by using better data structure and search algorithm. In addition, applying multiple NPUs is an possible way to improve the performance for Click router.

6.2 Evaluation of the methodology explained in Chapter 5

This section presents the result of the methodology on how to implement oriented object program on multiple processors using Click IPv4 router as an example. Meanwhile, the section give the method how to evaluate this methodology to obtain the result.

6.2.1 Experimental tools and approach

Experimental tools consist of StepNP platform that offers a simulation environment and is developed by STMicroelectronics. The purpose of the experimental approach is to estimate how much the time to process packets may be shorten by comparing Click executed on three ARM processors with Click running on a single ARM processors and to verify whether the methodology

(described in **Chapter 5**) may improve the Click router overall performance on three ARM processors. We choose "cycles per packet" as execution time for a packet, meaning that how many cycles should be required to process a packet in a IPv4 Click router configuration..

Based on StepNP tool, we designed two experimental architectures as follows:

(1) An experimental architecture with three multi-threaded ARM processors.

(2) An experimental architecture with a single multi-threaded ARM processor.

In the architecture (1), three processors respectively execute the same Click program with three different router configuration parameters. As a whole, such router architecture undertakes the equivalent task of processing incoming IP packets as architecture (2) does (the reason is explained in Chapter 5). Then, we set up a check point in "Print" and "Strip" elements (C++ objects) to count the executable cycle number in StepNP simulation. In the following step, we use "InfiniteSource" offered by Click software router to generate two streams of packets to pass the two above mentioned router architectures. To make experimental gathered realistic, the two streams of packets have different route while passing the two router architectures. By increasing the packets to pass those router architectures gradually, we may gather the experimental result for analysis.

6.2.2 Experimental result

(a) The cycles required for packets to pass the Click router on a single ARM processor.

Packets Number	The time getting to the Click router (cycles)	The time leaving the Click router (cycles)	Cycles/packet
1	20 679 838	20 857 502	177664
2	20 679 862	21 187 070	253604
3	20 679 862	21 512 638	277592
5	20 679 862	22 165 366	297101
7	20 679 862	22 818 094	305462
10	20 679 862	23 798 166	311830
12	20 679 862	24 452 030	314347
14	20 699 358	25 132 798	316674
18	20 699 358	26 432 126	318487
20	20 710 622	27 100 966	319517
35	20 710 622	32 003 638	322658
50	20 710 622	36 919 744	324182
80	20 710 622	46 726 054	325193
100	20 710 622	53 277 630	325670
250	20 681 646	102 292 734	326444
500	20 681 646	184 043 526	326724
902	20 681 646	315 520 334	326872

2000	20 701 846	674 883 886	327091
3000	20 701 846	1 002 177 166	327158
4000	20 681 646	1 329 470 446	327197
5 000	20 701 846	1 656 764 054	327212
10000	20 701 846	3 293 231 310	327253
50 000	20 685 046	16 373 670 990	327060
100000	20 685 046	32 728 341 502	327077

(b) The cycles required for packets to pass the Click router on three multiple ARM processor.

Packets Number	The time getting to the Click router (cycles)	The time leaving the Click router (cycles)	Cycles/packet
1	9 488 038	9 665 638	177600
2	9 488 062	9 848 174	180056
3	9 488 062	10 022 022	177987
5	9 488 062	10 375 942	177576
7	9 488 062	10 729 910	177407
10	9 488 062	11 263 694	177563
12	9 488 062	11 616 230	177347
14	9 488 062	11 972 318	177447
18	9 493 870	12 678 422	176920
20	9 496 742	13 037 374	177032
35	9 496 742	15 686 758	176858
50	9 496 742	18 342 214	176909
80	9 496 742	23 645 446	176859
100	9 496 742	27 183 790	176870
250	9 478 486	53 799 942	177286
500	9 478 486	98 131 614	177306
902	9 478 486	169 417 782	177316
2 000	9 498 582	364 391 870	177447
3 000	9 498 582	541 965 150	177489
4 000	9 498 582	719 538 430	177510
5 000	9 498 582	897 112 038	177523
10 000	9 498 582	1 784 979 294	177548
50 000	9 493 174	8 893 030 366	177671
100 000	9 493 174	17 778 100 878	177686

Table 6-7 The cycles counted for packets passing the Click router on two architectures respectively

The original data about performance estimation is shown in **Table 6-7**. We should note that a packet does not require so many cycles to pass the Click router in real router setup. We only want

to observe the change of performance about the two Click router architectures by comparing the cycle number per packet. Therefore, whether the data is real time for a packet to pass the Click router wont affect the accuracy of the performance analysis according such experimental result.

6.2.3 The discussion of results

Based on the data depicted in **Table 6-7**, we work out the comparison about the different performance on the two Click router architecture shown in **Table 6-8**.

Packets Number	One processor Cycles/packet	Three processors Cycles/packet	Performance(three processors) / Performance(one processor)
1	177664	177600	1,000
2	253604	180056	1,408
3	277592	177987	1,560
5	297101	177576	1,673
7	305462	177407	1,722
10	311830	177563	1,756
12	314347	177347	1,772
14	316674	177447	1,785
18	318487	176920	1,800
20	319517	177032	1,805
35	322658	176858	1,824
50	324182	176909	1,832
80	325193	176859	1,839
100	325670	176870	1,841
250	326444	177286	1,841
500	326724	177306	1,843
902	326872	177316	1,843
2 000	327091	177447	1,843
3 000	327158	177489	1,843
4 000	327197	177510	1,843
5 000	327212	177523	1,843
10 000	327253	177548	1,843
50 000	*327060*	*177671*	1,841
100 000	*327077*	*177686*	1,841

Table 6-8 The comparison of the packet processing performances on the two Click router architectures

According to **Table 6-8**, we are able to draw the diagram described in **Figure 6-2**.

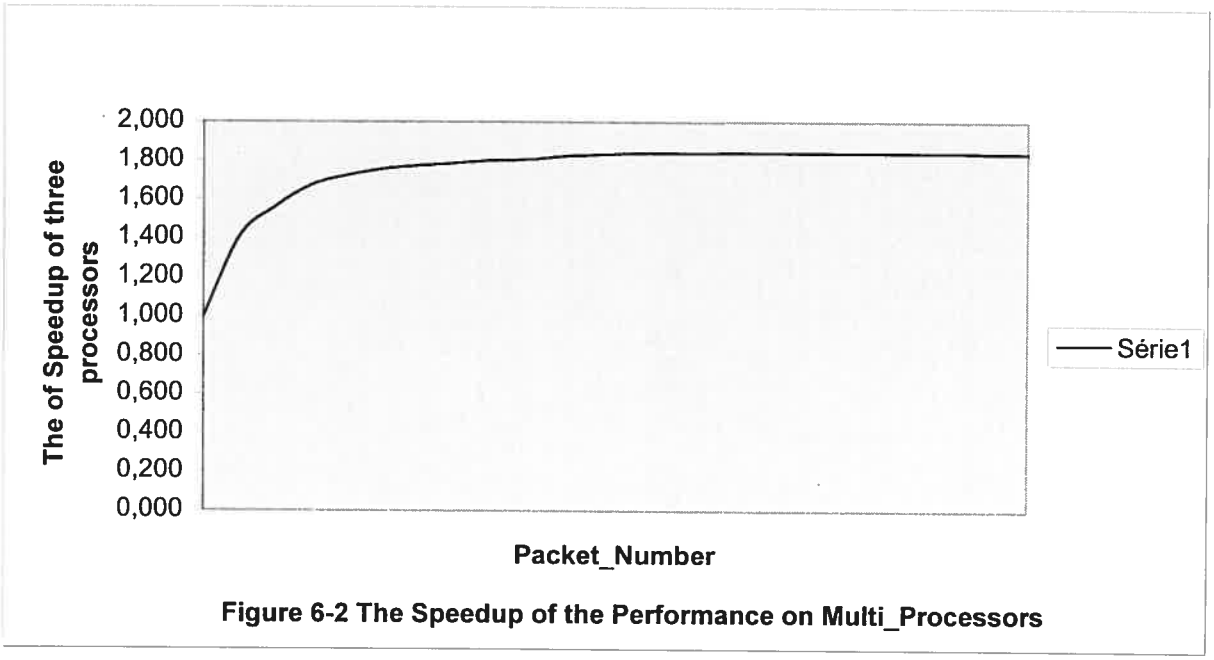


Figure 6-2 The Speedup of the Performance on Multi_Processors

In **Figure 6-2**, the curves line changes abruptly from start, and then gets a stable state as the packets processed increase. We may observe two points from this phenomenon.

1) As the number of incoming packets to Click router architecture increases, the speedup (cycles per packet on three processors % cycles per packet on one processor) increases gradually until getting to a stable point (***1.843*** in **Figure 6-2** from ***902*** packets on). This phenomenon shows that the Click router running on three processors may process the packets 1.843 faster than running on one processor, and the ability to process the packets may get improved as the number of incoming packets increases.
2) If the number of packets continues to increase to a specific point (as highlighted in **Table 6-8** from ***50000*** packets on), the speedup increases slightly. This indicates that the performance of process packet may be difficult to get improved, and even the performance may be reduced as incoming packets increase to a large amount.

Given the above analysis, we may draw a conclusion as follows:

The Click router implemented on three processors may process packets nearly double faster than a single processor does, proving that the methodology described in **Chapter 5** is feasible. If the number of processing packets is small, the advantage of Click multiprocessor architecture is not so noticeable. As the incoming packets increase, the overall execution time to process a packet may be shortened rapidly. Therefore, Click multiprocessor architecture fit to process large amount of

passing packets. However, such processing ability may be limited if the number of packets is beyond some constraints, meaning that a packet needs to wait longer time to pass the Click router. This situation shows that hardware resources get a saturated state at this moment. Under this circumstance, we may consider increasing processors or other methodologies if we want to improve the performance of processing packet on Click router. The methodology depicted in **Chapter 5** is scalable. Based on the methodology, we may increase processors to design a new multiprocessor architecture easily provided that we must consider the design constraints of the hardware resource.

Chapter 7

Conclusion and Future Work

We presented a co-design H/W methodology implemented in Click router. As a software based router, Click possesses flexible and extensible quality thanks to its modular architecture. The router architecture can construct different configuration to server different requirements. To avoid software disadvantage, we choose some key parts of Click element and implement them in hardware module. Using methodologies in co-design, we can increase the Click performance on packet forwarding process and improve performance overall Click router. The methodology was tested on multithread ARM CPU of StepNP platform. The time (cycle numbers) for processing an packet is 1.8 times as much as that before using the methodology.

In addition, we also found a methodology to implement different parts of Click IPv4 on multi-ARM processors. This way will reduce the overhead of passing packets in different elements. The current data structure about routing table determines Click is better to be applied in edged router.

Our future work may include implementing Click IPv4 with the combination of multiprocessors and HW/SW co-design. For large routing table, we will try to replace the current linear algorithm in lookupIPRouter element with binary search tree algorithm [7, 8] and test the performance. Some researches said such binary lookup tree algorithm may shorten the time of finding an route, especially in large network searching [8]. If such algorithm can be used in Click router and implemented in hardware, we may improve the performance of Click and keep the flexibility of Click over the conventional router. Furthermore, because Click router may be flexible to make up various router configurations according to different router functions or usages such as firewall, Ethernet switch, we will try to implement the practical element such as RED and IPFilter which are key elements [1] in those configuration, and evaluate the performance to overall router configuration. Those future work may cover more the parts of Click router and get higher performance of Click in network application.

References

[1] E.KOHLER,R.MORRIS,B.CHEN,J.JANNOTTI, and M.FRANS KAASHOEF, " The Click Modular Router ", *ACM Transactions on Computer Systems18(3), August 2000, pp. 263-297.* http://www.pdos.lcs.mit.edu/papers/click:tocs00/

[2] D.Decasper,Z.Dittia,G.Parulkar, and B.Plattner, " Router Plugins: A Software Architecture for next-Generation Routers", *IEEE/ACM Transactions on Networking, Feb. 2000, pp 2–15.*

[3] J.Aweya " IP Router Architectures: An Overview " CiteSeer.IST, 1999.38 http://citeseer.nj.nec.com/aweya99ip.html

[4] Y.Gouttlieb and L.Peterson, " A Comparative Study of Extensible Routers ", *Open Architectures and Network Programming Proceedings, 2002 IEEE, June 2002, pp. 51–62.*

[5]D.L.Mills. The Fuzzba. " In Proceedings of the SIGCOMM ",*88 Symposium, Stanford, CA, USA, Aug.*1998, *pp. 115-122.*

[6] P.Gpaulin,C.Pilkington, and E.Bensoudane, "StepNP: A System -Level Exploration Platform for Network Processors ", *Design & Test of Computers, IEEE, Nov.-Dec. 2002,pp. 17-26.*

[7] D.Pao,C.Liu,A.Wu,L.Yeung and K.S.Chan, "Efficient hardware Architecture for Fast IP Address Lookup", *Computers and Digital Techniques, IEE Proceedings- IEEE, Jan. 2003, pp. 43 - 52.*

[8] T.Harbaum,D.Meier,M.Zitterbart and D.Brokelmann, "Hardware-Assist for Ipv6 Routing Table Lookup", *International Symposium on Broadband European Networks (SYBEN). Zurich, Switzerland, May 1998,* http://citeseer.nj.nec.com/aweya99ip.html

[9] S.Floyd and V.Jacobson, " Random Early Detection Gateways for Congestion Avoidance", *IEEE/ACM Transaxtons on Networking, Aug. 1993, pp. 397 - 413.*

[10] A.Moestede,P.Sjodin,T.Khler, "Header Processing Requirements and Implementation Complexity for Ipv4Routers",*HP Labs Technical Reports,* 1998. http://www.hpl.hp.com/techreports/98/HPL-IRI-98-001.html

[11] E.Kohler,R.Morris,B.Chen, "Programming language optimizations for modular router configurations", *ACM SIGOPS Operating Systems Review, SESSION: Communication abstractions and optimizations Dec. 2002, pp. 251 - 263* .

[12] D.Quinn,M.Hubin, "Approche co-design pour des optimisations de haut viveau ", *IFT6221 Synthèse des systèmes numériques,*Dec. 2002.

[13] W.Stallings, "DATA AND COMPUTER COMMUNICATINS", *Prentice hall Upper Saddle River, N.J. c2004*

[14] R. Niemann, "Hardware/Software Co-Design for Data flow Dominated Embedded Systems", *Kluwer Academic Publishers, Boston, c1998.*

[15] G.De Micheli, "Computer-aided hardware-software codesign", *Micro, IEEE, Aug. 1994, pp. 10-16.*

[16] G.De.Micheli and K.Gupta, "Hardware/Software Co-Design", Poceedings of the *IEEE, MARCH, 1997, pp.349-365.*

[17] J.Levman, G.Khan, and J.Alirezaie, " Hardware-Software Co-Design of Embedded Systems: A Brief Survey ", Department of Electrical and Computer Engineering, Ryerson University.

[18] A. kalavade, and Edward A.Lee, "A Hardware-Software Codesign Methodology for DSP Applications", *Design & Test of Computers, IEEE, Sept. 1993, pp. 16 – 28.*

[19] A.Jantsch, P.Ellervee , J.Oberg, and A.Hemani, "A Case Study on Hardware/Software Partitioning ", *FPGAs for Custom Computing Machines, Proceedings. IEEE Workshop on, 10-13 April 1994, pp.111 – 118.*

[20] W.Wolf, "A Decade of hardware/Software Codesign ", *Computer, IEEE, April 2003, pp. 38- 43.*

[21] WAYNE H.WOLF, "Hardware-Software Co-Design of Embedded Systems ", Proceedings of the *IEEE. July,1994. pp. 967 - 989.*

[22] M.Chiodo, P.Giusto.H.Hsieh, A.Jurecska, L.Lavagno, and A.S-Vincentelli, "A Formal Methodology for Hardware/Software Co-design of Embedded Systems", *IEEE Micro, August, 1994, pp.26-36.* http://citeseer.ist.psu.edu/chiodo94formal.html

[23] Daniel D.Gajski, F.Vahid, and S.Narayan,"A System-Design Methodology: Executable-Specification Refinement", *European Design and Test Conference, 1994. EDAC, The European Conference on Design Automation. ETC European Test Conference. EUROASIC, The European Event in ASIC Design, Proceedings , IEEE, March 1994, pp. 458 – 463.*

[24] S.Schulz, J.W.Rozenblit, K.Buchenrieder, and M.Mrva,"Concepts for Model Compilation in Hardware/Software Codesign", *World Computing Congress, Beijing, 2000, pp. 413-420*

[25] R.Domer, A.Gerstlauer, P.Kritzinger, and M.Olivarez, "The SpecC System-Level Design language and Methodology, Part 2 Class 349", *Embedded Systems Conference San Francisco, March, 2002.* http://citeseer.ist.psu.edu/559949.html

[26] P.G. Paulin, C. Pilkington, E. Bensoudane, and M. Langevin, "Application of a Multi-Processor SoC platform to High-Speed packet Forwarding", *Design, Automation and Test in Europe Conference and Exhibition, 2004 Proceediings,* Feb. 2004 pp. 58 – 63.

[27] A. Baghdadi, D. Lyonnard, N. Zergainoh, and A. A. Jerrays, "An Efficient Architecture Model for Systematic design of Applications-Specific Multiprocessor SoC", *Design, Automation and Test in Europe, 2001. Conference and Exhibition 2001. Proceedings, March 2001, pp. 55 – 62.*

[28] B. Chen and R. Morris, "Flexible Control of parallelism in a Multiprocessor PC Router", *The USENIX 2001 Annual Technical Conference, June, 2001.*

[29] T.L. Casavant, P.Trdik, and F. Plasil, "Parallel Computers Theory and Practice", *IEEE computer Society Press, c1996.*

[30] J.L. Hennessy and D.A. Patterson, "Computer Organization and Design THE HARDWARE/SOFTWARE INTERFACE", *Morgan Kaufmann Publisher, Inc. San Francisco, California, c1998.*

[31] H. Philip and J. Enslow , " multiprocessors and parallel processing.", *N.Y., Toronto, Wiley, c1974.*

Appendix A

```
*******************************************************************************
               Date       : Thu Oct  9 17:56:59 2003
               Version    : 2003.06
               Design     : csdipttl
*******************************************************************************

*********************************************
*  State graph style report for process run:  *
*********************************************
===============================================================================
 present     next
 state      input  state            actions
-------------------------------------------------------------------------------
 s_0_0       c1    s_0_1          (no actions)
 s_0_1       c2    s_1_2        a_0   main_loop_DecIPTTL/ip_sum_46 (read)
                   a_1   main_loop_DecIPTTL/ip_ttl_45 (read)
                   a_3   main_loop_DecIPTTL/out_sum_55 (write)
                   a_5   main_loop_DecIPTTL/out_ttl_54 (write)
                   a_8   main_loop_DecIPTTL/add_48 (operation)
                   a_11  main_loop_DecIPTTL/add_50 (operation)
                   a_14  main_loop_DecIPTTL/add_52 (operation)
 s_0_1       c4    s_1_2        a_2   main_loop_DecIPTTL/start_44 (read)
 s_1_2       c5    s_1_2        a_0   main_loop_DecIPTTL/ip_sum_46 (read)
                   a_1   main_loop_DecIPTTL/ip_ttl_45 (read)
                   a_3   main_loop_DecIPTTL/out_sum_55 (write)
                   a_5   main_loop_DecIPTTL/out_ttl_54 (write)
                   a_8   main_loop_DecIPTTL/add_48 (operation)
                   a_11  main_loop_DecIPTTL/add_50 (operation)
                   a_14  main_loop_DecIPTTL/add_52 (operation)
 s_1_2       c6    s_1_2        a_2   main_loop_DecIPTTL/start_44 (read)
 +++++       c7    s_0_0          (no actions)
-------------------------------------------------------------------------------
       ***********      Branch Conditions      ***********
-------------------------------------------------------------------------------
 state       condition   source
-------------------------------------------------------------------------------
c1           true
c2           (and (branch 0 of conditional main_loop_DecIPTTL/BB5_SPLIT_L44_10)
                  true)
c4           true
c5           (branch 0 of conditional main_loop_DecIPTTL/BB5_SPLIT_L44_10)
c6           true
c7           true
-------------------------------------------------------------------------------
===============================================================================
```

Appendix B

```
*****************************************************************************
               Date       : Thu Oct  9 18:11:35 2003
               Version    : 2003.06
               Design     : csipfragp
*****************************************************************************

*********************************************
*  State graph style report for process run:  *
*********************************************
=============================================================================
 present     next
 state      input  state            actions
-----------------------------------------------------------------------------
 s_0_0       c1    s_0_1          (no actions)
 s_0_1       c2    s_1_2        a_1   loop_47/hlen_54 (read)
                   a_2   loop_47/ipoff_55 (read)
                   a_3   loop_47/off_53 (read)
                   a_4   loop_47/p1datalen_57 (read)
                   a_5   loop_47/plen_56 (read)
                   a_38  loop_47/sub_61 (operation)
 s_0_1       c4    s_1_2        a_6   loop_47/start_50 (read)
 s_0_1       c5    s_1_2        a_18  loop_47/add_64 (operation)
                   a_26  loop_47/lt_64 (operation)
 s_1_2       c6    s_1_3        a_0   loop_47/h1len_52 (read)
                   a_15  loop_47/add_61 (operation)
 s_1_2       c7    s_1_3        a_7   loop_47/out_ip_hl_67 (write)
                   a_9   loop_47/out_ip_off_68 (write)
 s_1_2       c8    s_1_3          (masked out)
 s_1_2       c10   s_1_3          (masked out)
 s_1_2       c12   s_1_3          (masked out)
 s_1_2       c14   s_1_3          (masked out)
 s_1_3       c16   s_1_2        a_1   loop_47/hlen_54 (read)
                   a_2   loop_47/ipoff_55 (read)
                   a_3   loop_47/off_53 (read)
                   a_4   loop_47/p1datalen_57 (read)
                   a_5   loop_47/plen_56 (read)
                   a_38  loop_47/sub_61 (operation)
 s_1_3       c17   s_1_2        a_6   loop_47/start_50 (read)
 s_1_3       c18   s_1_2        a_18  loop_47/add_64 (operation)
                   a_26  loop_47/lt_64 (operation)
 +++++       c19   s_0_0          (no actions)
-----------------------------------------------------------------------------
       ***********      Branch Conditions      ***********
-----------------------------------------------------------------------------
 state       condition   source
-----------------------------------------------------------------------------
c1          true
c2          (and (branch 0 of conditional loop_47/BB5_SPLIT_L50_10)
               true)
c4          true
c5          (and (branch 0 of conditional loop_47/BB5_SPLIT_L50_10)
               true)
c6          (branch 0 of conditional loop_47/BB5_SPLIT_L50_10)
```

```
c7          (branch 0 of conditional loop_47/BB5_SPLIT_L50_10)
c8          (branch 0 of conditional loop_47/BB6_SPLIT_L62_34)
c10         (branch 0 of conditional loop_47/BB9_SPLIT_L64_43)
c12         (branch 1 of conditional loop_47/BB6_SPLIT_L62_34)
c14         (branch 1 of conditional loop_47/BB9_SPLIT_L64_43)
c16         (branch 0 of conditional loop_47/BB5_SPLIT_L50_10)
c17         true
c18         (branch 0 of conditional loop_47/BB5_SPLIT_L50_10)
c19         true
------------------------------------------------------------------------
========================================================================
```

Appendix C

```
*******************************************************************************
               Date      : Thu Oct  9 18:28:19 2003
               Version   : 2003.06
               Design    : csstrip
*******************************************************************************

*********************************************
*  State graph style report for process run:  *
*********************************************
===============================================================================
 present     next
 state      input  state            actions
-------------------------------------------------------------------------------
 s_0_0       c1    s_0_1             (no actions)
 s_0_1       c2    s_1_2        a_0   process_loop/data_51 (read)
                   a_1   process_loop/in_50 (read)
                   a_3   process_loop/result_56 (write)
                   a_6   process_loop/add_54 (operation)
 s_0_1       c4    s_1_2        a_2   process_loop/start_47 (read)
 s_1_2       c5    s_1_2        a_0   process_loop/data_51 (read)
                   a_1   process_loop/in_50 (read)
                   a_3   process_loop/result_56 (write)
                   a_6   process_loop/add_54 (operation)
 s_1_2       c6    s_1_2        a_2   process_loop/start_47 (read)
 +++++       c7    s_0_0          (no actions)
-------------------------------------------------------------------------------
       ***********      Branch Conditions      ***********
-------------------------------------------------------------------------------
 state       condition   source
-------------------------------------------------------------------------------
c1          true
c2          (and (branch 0 of conditional process_loop/BB5_SPLIT_L47_10)
                 true)
c4          true
c5          (branch 0 of conditional process_loop/BB5_SPLIT_L47_10)
c6          true
c7          true
-------------------------------------------------------------------------------
===============================================================================
```

Appendix D

```
*****************************************************************************
               Date       : Thu Oct  9 18:30:41 2003
               Version    : 2003.06
               Design     : cs
*****************************************************************************

*********************************************
*  State graph style report for process run:  *
*********************************************
=============================================================================
 present     next
 state      input  state            actions
-----------------------------------------------------------------------------
 s_0_0       c1    s_0_1              (no actions)
 s_0_1       c2    s_1_2        a_2   loop_38/start_38 (read)
 s_0_1       c4    s_2_3        a_0   loop_40/in_46 (read)
                   a_3   loop_40/result_49 (write)
                   a_6   loop_40/add_47 (operation)
                   a_14  loop_40/add_47_2 (operation)
                   a_17  loop_40/add_48 (operation)
 s_0_1       c6    s_2_3        a_1   loop_40/start_43 (read)
 s_0_1       c7    s_2_3        a_2   loop_38/start_38 (read)
 s_0_1       c8    s_2_3          (masked out)
 s_0_1       c9    s_2_3          (masked out)
 s_1_2       c11   s_1_2        a_2   loop_38/start_38 (read)
 s_1_2       c12   s_2_3        a_0   loop_40/in_46 (read)
                   a_3   loop_40/result_49 (write)
                   a_6   loop_40/add_47 (operation)
                   a_14  loop_40/add_47_2 (operation)
                   a_17  loop_40/add_48 (operation)
 s_1_2       c13   s_2_3        a_1   loop_40/start_43 (read)
 s_1_2       c14   s_2_3        a_2   loop_38/start_38 (read)
 s_1_2       c15   s_2_3          (masked out)
 s_1_2       c16   s_2_3          (masked out)
 s_2_3       c17   s_2_3        a_0   loop_40/in_46 (read)
                   a_3   loop_40/result_49 (write)
                   a_6   loop_40/add_47 (operation)
                   a_14  loop_40/add_47_2 (operation)
                   a_17  loop_40/add_48 (operation)
 s_2_3       c18   s_2_3          (masked out)
 s_2_3       c19   s_2_3        a_1   loop_40/start_43 (read)
 s_2_3       c20   s_2_3          (masked out)
 s_2_3       c21   s_2_3          (masked out)
 +++++       c22   s_0_0          (no actions)
-----------------------------------------------------------------------------
       ***********      Branch Conditions      ***********
-----------------------------------------------------------------------------
 state       condition   source
-----------------------------------------------------------------------------
c1          true
c2          (and true
                 (not (branch 1 of conditional loop_38/BB4_SPLIT_L38_32)))
c4          (and (branch 0 of conditional loop_40/BB5_SPLIT_L43_12)
```

```
                true
                (branch 1 of conditional loop_38/BB4_SPLIT_L38_32))
c6          (and true
                (branch 1 of conditional loop_38/BB4_SPLIT_L38_32))
c7          (and true
                (branch 1 of conditional loop_38/BB4_SPLIT_L38_32))
c8          (and true
                (branch 1 of conditional loop_38/BB4_SPLIT_L38_32))
c9          (and (branch 1 of conditional loop_40/BB5_SPLIT_L43_12)
                true
                (branch 1 of conditional loop_38/BB4_SPLIT_L38_32))
c11         (not (branch 1 of conditional loop_38/BB4_SPLIT_L38_32))
c12         (and (branch 0 of conditional loop_40/BB5_SPLIT_L43_12)
                (branch 1 of conditional loop_38/BB4_SPLIT_L38_32))
c13         (branch 1 of conditional loop_38/BB4_SPLIT_L38_32)
c14         (branch 1 of conditional loop_38/BB4_SPLIT_L38_32)
c15         (branch 1 of conditional loop_38/BB4_SPLIT_L38_32)
c16         (and (branch 1 of conditional loop_40/BB5_SPLIT_L43_12)
                (branch 1 of conditional loop_38/BB4_SPLIT_L38_32))
c17         (branch 0 of conditional loop_40/BB5_SPLIT_L43_12)
c18         (branch 0 of conditional loop_40/BB5_SPLIT_L43_12)
c19         true
c20         (branch 1 of conditional loop_40/BB5_SPLIT_L43_12)
c21         (branch 1 of conditional loop_40/BB5_SPLIT_L43_12)
c22         true
--------------------------------------------------------------------------------
================================================================================
```

Appendix E

```
****************************************************************************
               Date       : Tue Sep 16 17:26:31 2003
               Version    : 2003.06
               Design     : csdipttl
****************************************************************************

************************************
*  Summary report for process run:  *
************************************
----------------------------------------------------------------------------
     Timing Summary
----------------------------------------------------------------------------
 Clock period 25.00
 Loop timing information:
    run.....................................2 cycles (cycles 0 - 2)
        main_loop_DecIPTTL..................1 cycle  (cycles 1 - 2)
----------------------------------------------------------------------------
     Area Summary
----------------------------------------------------------------------------
 Estimated combinational area    247.342819
 Estimated sequential area       565.640015
 TOTAL                           812.982834

 3 control states
 3 basic transitions
 2 control inputs
 2 control outputs

----------------------------------------------------------------------------
     Resource types
----------------------------------------------------------------------------
      Register Types
=======================================

      Operator Types
=======================================
      (8_8->8)-bit DW01_add.............1
      (16_16->17)-bit DW01_add..........2

      I/O Ports
=======================================
      1-bit input port..................1
      8-bit registered output port.....1
      16-bit registered output port.....1
      32-bit input port.................2

----------------------------------------------------------------------------
```

Appendix F

```
****************************************************************************
               Date       : Tue Sep 16 17:43:05 2003
               Version    : 2003.06
               Design     : csipfragp
****************************************************************************

************************************
*  Summary report for process run:  *
************************************
----------------------------------------------------------------------------
     Timing Summary
----------------------------------------------------------------------------
 Clock period 25.00
 Loop timing information:
    run.....................................3 cycles (cycles 0 - 3)
        loop_47.............................2 cycles (cycles 1 - 3)
----------------------------------------------------------------------------
     Area Summary
----------------------------------------------------------------------------
 Estimated combinational area    2427.434082
 Estimated sequential area       1614.905029
 TOTAL                           4042.339111

 4 control states
 4 basic transitions
 5 control inputs
 7 control outputs

----------------------------------------------------------------------------
     Resource types
----------------------------------------------------------------------------
      Register Types
=======================================
      1-bit register.....................3
      14-bit register....................1
      32-bit register....................1

      Operator Types
=======================================
      (32_32->1)-bit DW01_cmp2...........1
      (32_32->32)-bit DW01_add...........1
      (32_32->32)-bit DW01_sub...........1

      I/O Ports
=======================================
      1-bit input port...................1
      8-bit registered output port.......1
      16-bit registered output port......1
      32-bit input port..................6

----------------------------------------------------------------------------
```

Appendix G

```
********************************************************************************
               Date       : Tue Sep 16 18:35:21 2003
               Version    : 2003.06
               Design     : csiptable
********************************************************************************

****************************************
*  Summary report for process lookup:  *
****************************************
--------------------------------------------------------------------------------
     Timing Summary
--------------------------------------------------------------------------------
 Clock period 25.00
 Loop timing information:
    lookup.....................................5 cycles (cycles 0 - 5)
        reset_loop.............................4 cycles (cycles 1 - 5)
            loop_29............................1 cycle  (cycles 2 - 3)
                exit at line 29..................... (cycle 2)
            main_loop..........................1 cycle  (cycles 4 - 5)
--------------------------------------------------------------------------------
     Area Summary
--------------------------------------------------------------------------------
 Estimated combinational area    6396.606445
 Estimated sequential area       55882.113281
 TOTAL                           62278.719727

 6 control states
 8 basic transitions
 7 control inputs
 20 control outputs

--------------------------------------------------------------------------------
     Resource types
--------------------------------------------------------------------------------
      Register Types
========================================

      Operator Types
========================================
      (32->32)-bit DW01_inc.............1

      I/O Ports
========================================
      1-bit input port..................1
      1-bit registered output port.......1
      32-bit input port.................6
      32-bit registered output port......83

--------------------------------------------------------------------------------
```

Appendix H

```
****************************************************************************
               Date       : Tue Sep 16 18:11:14 2003
               Version    : 2003.06
               Design     : csstrip
****************************************************************************

************************************
*  Summary report for process run:  *
************************************
----------------------------------------------------------------------------
     Timing Summary
----------------------------------------------------------------------------
 Clock period 25.00
 Loop timing information:
    run.....................................2 cycles (cycles 0 - 2)
        process_loop........................1 cycle  (cycles 1 - 2)
----------------------------------------------------------------------------
     Area Summary
----------------------------------------------------------------------------
 Estimated combinational area    188.465607
 Estimated sequential area       733.520020
 TOTAL                           921.985626

 3 control states
 3 basic transitions
 2 control inputs
 2 control outputs

----------------------------------------------------------------------------
     Resource types
----------------------------------------------------------------------------
      Register Types
=======================================

      Operator Types
=======================================
      (32_32->32)-bit DW01_add...........1

      I/O Ports
=======================================
      1-bit input port...................1
      32-bit input port..................2
      32-bit registered output port......1

----------------------------------------------------------------------------
```

Appendix I

```
/*****************************************************************************

  CSFragp.h --

  Author: Dan Li
 *****************************************************************************/

/*****************************************************************************

  MODIFICATION LOG - modifiers, enter your name, affiliation, date and
  changes you are making here.

      Name, Affiliation, Date:
  Description of Modification:

 *****************************************************************************/

#define NB_BITS_HEADLEN  8
#define NB_BITS_OUT  16
#define NB_BITS_DATA 32
#define IP_OFFMASK 0x1fff
#define IP_RF 0x8000
#define IP_DF 0x4000
#define IP_MF 0x2000

SC_MODULE(csipfragp) {

  sc_in<bool>  reset;
  sc_in<bool>  start;

  sc_in<int>   hlen;
  sc_in<int>   off;
  sc_in<int>   hlen;
  sc_in<int>   ipoff;
  sc_in<int>   plen;
  sc_in<int>   p1datalen;

  sc_out<sc_uint<NB_BITS_HEADLEN> > out_ip_hl;
  sc_out<sc_uint <NB_BITS_OUT> > out_ip_off;

  sc_in_clk    clk;

  SC_CTOR(csipfragp){
      SC_CTHREAD(run, clk.pos());
      watching(reset.delayed() == true);
  }

 // csipfragp() { cs("IPfragP");};

  void run();
};
```

```
/*****************************************************************************

CS: IPFragmenter

*****************************************************************************/

/*****************************************************************************

  csipfragp.cpp --

  Author: Dan Li

 *****************************************************************************/
#innclude <systemc.h>
#include "csfragp.h"
#include <sys/param.h>

void csipfragp::run() {
 int   inh1len,inoff,inoff_ip;
 int   inhlen,inipoff;
 int   inplen,inp1datalen;

   while(1) {
     inh1len = 0;
     inoff = 0;
     inhlen = 0;
     inipoff = 0;
     inplen = 0;
     inp1datalen = 0;

     wait_until(start.delayed() == true);
     wait();
     while(1) {
       if (start == 1) {
            inh1len = h1len.read();
            inoff = off.read();
            inhlen = hlen.read();
            inipoff = ipoff.read();
            inplen = plen.read();
            inp1datalen = p1datalen.read();

          inh1len = inh1len >> 2;
            inoff_ip = ((inoff - inhlen) >> 3) +(inipoff & IP_OFFMASK);
            if(inipoff & IP_MF)
              inoff_ip |= IP_MF;
            if(inoff + inp1datalen < inplen)
              inoff_ip |= IP_MF;

          out_ip_hl.write(inh1len);
            out_ip_off.write(inoff_ip);

         }
         else inoff = 0;
         wait();
      };
   };
}
```

Appendix J

```
/*****************************************************************************

  CSDepttl.h --

  Author: Dan Li

 *****************************************************************************/

/*****************************************************************************

  MODIFICATION LOG - modifiers, enter your name, affiliation, date and
  changes you are making here.

      Name, Affiliation, Date:
  Description of Modification:

 *****************************************************************************/

//define NB_BITS_OUT  16
#define NB_BITS_DATA 32
#define NB_BITS_IP_TTL 8
#define NB_BITS_IP_SUM 16

SC_MODULE(csdipttl) {

  sc_in<bool>  reset;
  sc_in<bool>  start;

  sc_in<unsigned> ip_ttl;
  sc_in<unsigned> ip_sum;

  sc_out<sc_uint<NB_BITS_IP_TTL> > out_ttl;
  sc_out<sc_uint<NB_BITS_IP_SUM> > out_sum;


  sc_in_clk    clk;

  SC_CTOR(csdipttl){
      SC_CTHREAD(run, clk.pos());
      watching(reset.delayed() == true);
  }

 // csdipttl() { cs("decipttl");};

  void run();
};
```

```
/*****************************************************************************

CS: DecTPTTL

*****************************************************************************/

/*****************************************************************************

  cs.cpp --

  Author: Dan Li

 *****************************************************************************/
#include <systemc.h>
#include "csdipttl.h"

void csdipttl::run() {

sc_uint<NB_BITS_IP_TTL> inttl;
sc_uint<NB_BITS_DATA> insum32;


   reset_loop_DecIPTTL:while(1) {
      inttl = 0;

      insum32 = 0;

      wait_until(start.delayed() == true);
      wait();
      main_loop_DecIPTTL:while(1) {
          if (start == 1) {
            inttl=ip_ttl.read();
            insum32=~ip_sum.read();

            inttl--;
            insum32 = insum32.range(15,0)+0xFEFF;
            insum32 = insum32.range(15,0)+insum32.range(31,16);

            out_ttl.write(inttl);
            out_sum.write(~insum32.range(15,0));
         }
         else insum32 = 0;

         wait();
      };
   };
}
```