Towards a Compilation Infrastructure for Network Processors

by

Martin Labrecque

A Thesis submitted in conformity with the requirements
for the Degree of Master of Applied Science in the
Department of Electrical and Computer Engineering
University of Toronto

Document Version January 25, 2006.

Towards a Compilation Infrastructure for Network Processors

Martin Labrecque
Master of Applied Science,
2006
Department of Electrical and Computer Engineering
University of Toronto

Abstract

Modern network processors (NPs) typically resemble a highly-multithreaded multiprocessor-on-a-chip, supporting a wide variety of mechanisms for on-chip storage and inter-task communication. NP applications are themselves composed of many threads that share memory and other resources, and synchronize and communicate frequently. In contrast, studies of new NP architectures and features are often performed by benchmarking a simulation model of the new NP using independent kernel programs that neither communicate nor share memory. In this paper we present a NP simulation infrastructure that (i) uses realistic NP applications that are multithreaded, share memory, synchronize, and communicate; and (ii) automatically maps these applications to a variety of NP architectures and features. We use our infrastructure to evaluate threading and scaling, on-chip storage and communication, and to suggest future techniques for automated compilation for NPs.

Acknowledgements

First, I would like to thank my advisor, Gregory Steffan, for his constructive comments throughout this project and especially for his patience in showing me how to improve my writing.

I also aknowledge my labmates and groupmates for their camaradery. I also thank the UTKC, my sempais and Tominaga Sensei for introducing me to the ’basics’.

Special thanks go to my relatives for their encouragements and support. Most of the credit goes to my parents, who are coaches and unconditional fans of mine. A special mention goes to my brother for helping me move in Toronto. My beloved Mayrose also deserves a place of choice on this page for making the sun shine even on rainy days.

Finally, the financial support from the Natural Sciences and Engineering Research Council made this work possible.

Contents

List of Figures

List of Tables

Glossary

API	Application Program Interface
ASIC	Application Specific Integrated Circuit
CAM	Content Addressable Memory
CRC	Cyclic Redundancy Check
FIFO	First In First Out
Gbps	Gigabits per second
LAN	Local Area Network
MESI	Modified/Exclusive/Shared/Invalid
NP	Network Processor
OC-1	51.84 Mbps fiber optic link
OC-3	155.52 Mbps fiber optic link
OC-12	622.08 Mbps fiber optic link
OC-24	1.244 Gbps fiber optic link
OC-48	2.488 Gbps fiber optic link
OC-192	10 Gbps fiber optic link
OC-256	13.271 Gbps fiber optic link
OC-768	40 Gbps fiber optic link
PE	Processing Engine/Element
RISC	Reduced Instruction Set Computing/Computer
RFC	Request for Comments
SDK	Software Development Kit
VLIW	Very Long Instruction Word

1 Introduction

With the advent of e-commerce and the spread of broadband fiber-to-the-premises (FTTP) connectivity, the traffic of new IP services is expected to grow by over 100% per year [5] from 2004 to 2006. High speed network links require computers, or *network nodes*, to share the link capacity among many clients and to route traffic efficiently. Consequently, the pressure on those network nodes to process greater packet rates is bound to increase in the near future.

Until recently, network nodes were exclusively made out of fixed ASICs that were performing increasingly complex tasks. With the wide range of requirements for network nodes and the speed at which the needs of Internet users are changing, it is now very expensive for service providers to develop custom solutions in hardware to cope with each of their customers' needs. For this reason, the trend has been to put more and more programmability inside packet processors, even at line rates of 40-Gbps [13], in order to make the most out of the investment of hardware in network nodes. This programmability not only allows conforming to new requirements of processing, but also to develop more input dependent processing. The deployment of *network processors* (NPs), those programmable network nodes, has become increasingly common as networking applications continue to push more processing into the network.

Modern NP architectures are typically organized as a highly-multithreaded multiprocessor-on-a-chip, supporting a wide variety of mechanisms for on-chip storage and inter-task communication. In turn, NP applications are typically composed of many threads that share memory and other resources, as well as synchronize and communicate frequently. Furthermore, these applications are usually programmed in assembly code by hand to ensure the most efficient code possible, and to fully exploit the wide variety of instructions for synchronization and communication. Because of the complexity involved, the programmer must typically revert to modifying sample applications and use library code that takes over pre-determined shared resources on the chip.

1.1 Research Objective: An Integrated Approach to NP Architecture Simulation

Recently, a wide variety of network processors has emerged, presenting drastically different architectures and programming paradigms. Open questions still remain in deciding what should be the architecture of a programmable packet processor (or network processor) and how it should be programmed. In “Programming Challenges in Network Processor Deployment” [44], the authors name three central compilation challenges to the success of NPs that we address in our work: (i) partitioning an application in tasks over threads and processors, (ii) scheduling the resulting tasks and arbitrating the NP resources between them, and (iii) managing the task data transfers. In this work, we propose an infrastructure to realistically compare various network processor architectures and to evaluate how we can adapt applications for each of those NPs.

Our goal is to evaluate powerful network processors that can guarantee line rate performance, while being programmable and configurable. We also want to be able to quantify the headroom for further software features and the bottlenecks to direct NP development efforts. Our work focuses on system level exploration to automatically transform realistic network applications and simulate them accurately on widely-varying realistic NP architectures. In this thesis, we present a *Network Processor Infrastructure for Research and Evaluation* (NPIRE), which is composed of an integrated compiler and simulator for a wide design space of network processor architectures.

The influence of our work on NP design is to present that network processors can be programmed from a high-level language, assuming a certain organization in the application description, as explained in Chapter 3. We support this proposition by an evaluation of automated compiler transformations to scale the throughput of an application to the underlying hardware. To identify the performance-limiting factors, we perform a systematic bottleneck identification. Finally, we provide a methodology and a parametric architectural simulation environment for evaluating NP architectural features.

1.2 Thesis Organization

This dissertation is organized as follows. Chapter 2 gives a background in network processor architecture, and programming/compilation techniques, as well as summarizes the relevant research fields and research projects. In the next chapter, Chapter 3, we describe a compilation framework and techniques for transforming high-level NP applications, including managing both memory and tasks. We explain in more detail the compilation flow from the raw application to the target network processor in Chapter 4. Chapter 5 describes our simulation infrastructure, and our algorithm for mapping tasks to processing resources. In Section 6, we evaluate the impact of our compilation techniques on the scalability of selected NP applications, and we conclude in Section 7.

2 Background

ASIC designs for line-cards are protocol and line rate-specific: they have high acquisition and maintenance costs and are feature limited to what is provisioned in hardware. Network processors have emerged as more flexible programmable solutions. However, because of their architecture, they present several challenges for automated compilation. In this chapter, we first present an overview of the network processing industry and of network processor architectures. We then present different published research works that share common compilation objectives with our study. Next, we introduce benchmarks that are commonly used to evaluate the performance of network processor systems. Finally, we describe a building block of our infrastructure: the Click Modular Router, that we use to build benchmarks.

2.1 Overview of the Network Processing Industry

Computers connected directly to high speed links have a strategic position to perform packet processing tasks. Those tasks are typically at the Network (3rd) and Transport (4th) layers of the Open System Interconnection (OSI) model. One important function of the network nodes is to check the integrity of the packets in transit to prevent, in particular, packet headers to be mis-interpreted. In enterprise-scale networks, certain types of traffic must be re-directed to specific server machines; other types of traffic may simply be banned. For Internet Service Providers, traffic to and from certain clients might be prioritized, requiring the enforcement of quality of service policies. Network nodes will also perform packet accounting to provide billing information proportional to bandwidth usage. Finally, network nodes may alter every packet to encapsulated them inside another protocol. In summary, network computers were traditionally performing only low-level and low-complexity

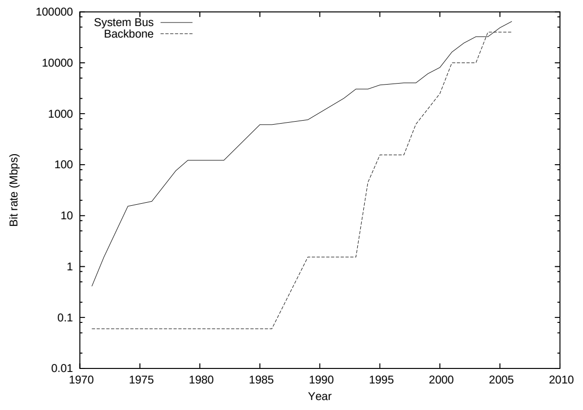


Figure 2.1: Evolution of computer memory data bus and backbone network link bit rates Data sources include [91], [90], [28], [52].

tasks.

With the increased Internet traffic and the diversity of web services, the industry is progressively taking advantage of the dedicated network nodes to also provide services at the higher layers of the OSI model: Session, Presentation and Application. For example, it is common now for network devices to *masquerade* all the traffic of a small or home office under a unique network address, so that there is only one link required to the Internet Service Provider. Also, there is a lot of interest in offloading expensive web server machines from handling the connection aspects of packet transfers (*TCP termination*) and the encryption of packets for secured transactions. Other network processing applications include the need to parse/alter the content of web messages to, for example, hide server-side changes in data layout or to load balance servers: a technique called *URL switching*. Multimedia applications of network nodes include *media transcoding* to, for example, allow a client with a portable screen and a slow connection to view, in a lower resolution, a large picture file. Finally, with the spread of viruses causing costly downtimes, it is vital for certain enterprises to inspect each packet and detect any irregularity in the data coming inside their network.

The architecture of network nodes has evolved considerably in the recent past. As explained by Roberts [71], one of the pioneers of Internet, 1997 was a turning point year for the designers of routers and switches. In 1997, the focus has changed from delay engineering to capacity engineering, i.e. since 1997, the router technology limits the maximum wire speed. Figure 2.1 shows the

Table 2.1: Network processor cycle budget assuming a RISC core at 400 Mhz and minimum packet size of 64B.

Rate	Speed (Gbps)	Packet Inter-Arrival (ns)	Cycles per Packet
OC-48	2.5	204.8	81
OC-192	10	51.2	20
OC-768	40	12.8	5
OC-3072	160	3.2	1

evolution of the data rate available on the memory bus (also known as the “front side bus”) of a high-end consumer computer system compared to the evolution of the bit rate of backbone links. The figure shows that in the mid-2000, those two data rates are close to meeting. Considering that a packet buffered in memory must travel twice on the memory bus (in and out), we can clearly see that the architecture of a conventional computer is unsuited to handle peak network traffic. Another view of the same reality is depicted in Table 2.1. A moderately clocked RISC processor only has 5 cycles to process a stream of minimum sized packets (the most dense and hence stressful traffic) at OC-768. This cycle budget given by conventional processors is truly insufficient.

On an NP, maximum efficiency is required to process packets at their incoming rate, even if the complexity is high. If packet losses due to contention in an NP may be acceptable in a distributed video game, they can only be marginally tolerated in a high-performance network. Network processors are data-driven machines that address completely different challenges than traditional computers. The software handling the quasi totality of the packets, or the forwarding software, must service packets at the line rate, no matter what it is. This aspect makes it very complicated to write and maintain a tightly written program (or a library) when the requirements change frequently. To exploit to the maximum the resources on an NP, the programmer must match the application to the chip architecture while conversely, the NP architecture must be matched to the application.

2.2 Overview of Network Processor Architectures

Because network processing is a relatively new field, we will first present in this section a recapitulation of the origins of network processors (NPs). Since an understanding of the architectural features of NPs is needed to evaluate how to program them, we next present the main categories of network processor organizations. Finally, for each of these categories, we present the state of the art processors made available by the industry.

2.2.1 A Brief History of Network Processors

The origins of network processors can be traced back to the late 1990s. In 1998, IBM started its network processor activities at the IBM Research Triangle Park Laboratory. In 1999, Intel's acquisition of Level One Communications, Inc., later propelled the IXP1200 network processor to be one of today's most well known re-programmable network engines. By releasing a developer's tool kit and an academic program (the Intel IXA University Program, created in late 2000), Intel became a strong supporter of the shift from the ASIC process to programmable architectures. In 2002, the PowerNP from IBM was the first network processor verified to operate at 10 Gbps. A lot of companies tried to make a name for themselves in the early days of network processors. A short list of the current survivors includes: Agere, AMCC, Bay Microsystems, Blue Steel, Broadcom, Cisco, ClearWater, Conexant, Cognigine, Ericsson, EZ-chip, Fast-Chip, Hifn, IBM, IDT, Intel, IP Infusion, Lucent, Mindspeed, Motorola, Nortel Networks, Pixel fusion, PMC Sierra, Silicon Access, Switch ON, Vitesse, Xelerated and Xilinx. According to the Worldwide Datacom/Telecom Semiconductor 2004 Vendor Analysis (IDC #33483), AMCC and Intel are leading the network processor market.

2.2.2 High-Level NP Architecture

As explained in the PowerNP paper [3], the system architecture of network processors is divided in two main paradigms: the *run-to-completion* (RTC) and pipeline models. The RTC label encompasses single stage models where a single processor takes care of the bulk of the packet processing. As shown in Figure 2.2(a), the input stream of packets may be divided upstream among several processors working in parallel. The programming model is one of a single thread, with a global view of all shared resources (for example: hardware co-processors, memory resources and busses).

The alternative, the pipeline model, consists of dividing the processing of a single packet into several processor stages, where each processor is specialized to perform a certain task (Figure 2.2(b)). Often, different processor stages have access to different hardware resources, such as memory channels. A key characteristic of this model is that the pipeline will function at the minimum rate of its constituting stages. So programming such a processor is usually either complex or heavily constrained to partition the work evenly among the pipeline stages. Program maintenance may lead to serious difficulties for programmers. One the other hand, one advantage of this model is a strict ordering between operations that facilitates synchronization and arbitration. The pipeline model can also exist in the form of multiple parallel pipelines, where each stage has the same view of on-chip shared resources.

Figure 2.2(c) presents a third alternative that is the most flexible. The hybrid RTC-pipeline model exists when packets do not necessarily flow on ordered isolated pipelines: packet processing can be viewed as distributed on a matrix of processors having a common view of on-chip resources. However, if the programmer desires it, he can design his application according to the pipelined or run-to-completion model (with possibly more arbitration required between shared resources).

2.2.2.1 A Brief Survey of Available Commercial NPs

Before listing some of the main features of major network processors, we explain what components are especially relevant in this presentation.

In general network processing, the greater the difference between the designed service rate and the peak rate of the network, the more buffering is required. Packets buffers and large routing tables require external memory storage, a common feature of network processors. Large buffers are commonly implemented in DRAM (rather than SRAM) for budgetary reasons. The wide busses of DRAMs reduce the number of data transfer cycles but not the latency of accesses, so there is a tradeoff between latency overhead and granularity overhead. Also relevant to packet processing in the chips listed below are SRAM buffers both on-chip and off-chip. They have a quick bus turn-

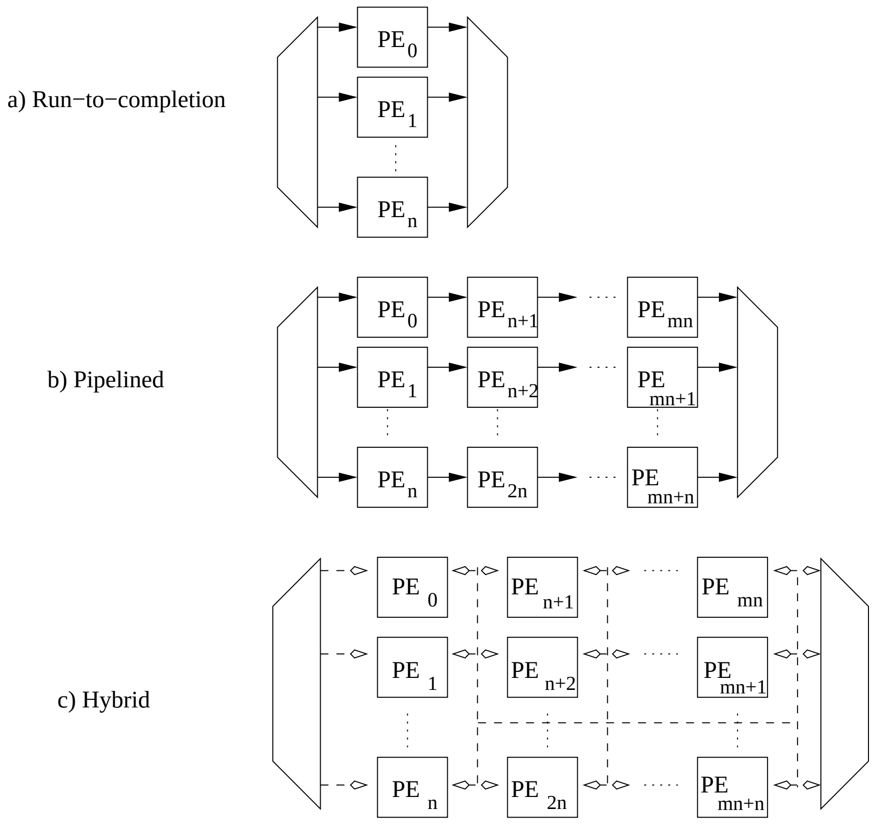


Figure 2.2: The three main system architectures of network processors.

around (read to write and vice-versa) and have a high frequency of operation. Their low density is the reason for their higher price. Aside from off-chip memory channels, most of the processors have means of caching data or at least generating an indexed view of data structures. However, caches are not common due to the weak temporal and spatial locality of the data touched. In summary, the organization of memory resources in a network processor is important in deciding how to program it.

Several processors make use of multi-threading to hide the latency of memory operations by overlapping that latency with some computation related to another packet. Also, there is a convergence on the idea to use many processing elements to exploit as much parallelism as possible. Hence, the number of threads of execution in a processor is of interest to us.

In the NPs, most *processing elements* (PEs) have some internal instruction and data memory.

The instruction set is a blend of conventional RISC instructions with additional features specifically tailored for network processing. Several processing elements can also harness the power of a CAM or a CRC unit. Most PEs also have interfaces to neighbouring PEs and some have access to a shared bus, where applicable. We do not intend to delve in the details of the organization of processing elements because their design is somewhat orthogonal to the system-level design of the network processor. We rather intend on outlining the functions that they should support.

As we will show, a number of processors have exclusive features: this motivates our work in finding the key architectural components of NPs. For example, some NPs have on-chip accelerators often consisting of a variant of a hash unit designed to make a lookup based on an *n-tuple* (a set of “n” ordered values). We limit the scope of this document to only include those on-chip resources that are directly relevant in the execution of the application. For this reason, media interfaces that handle different electrical or optical signaling protocols will not be considered.

We divide our description of the processors between the run-to-completion, pipelined and hybrid high-level architectures (as defined in section 2.2.2). Some processors have a very short description since they have little publicly available information.

Run-to-Completion Architectures In this processor model, packets are processed by a single-stage, ”run-to-completion” program on a single core.

Vitesse IQ2200 This 2.5 Gbps network processor, released in late 2001, has 4, 5-threaded processing elements. It has a classification and queue management engine.

AMCC nP7510 Released in 2002, this 10 Gbps network processor is composed of six multi-threaded processing elements. The chip is equipped with several hardware coprocessors, such as a search coprocessor, a statistics engine and specialized counters.

Broadcom BCM1480 This chip has four 64-bit MIPS CPUs scalable from 800 MHz to 1.2 GHz, a shared bus, a shared L2 cache, a memory controller, and I/O bridges. Coherence across processing elements is ensured using a MESI protocol. This NP can handle 10 Gbps and it supersedes a similar architecture with only 2 CPUs.

Mindspeed M27483 TSP3 The TSP3 architecture is based on two programmable processor cores tightly coupled with several co-processing engines. The processor itself is clocked at 333MHz and supports packet rates at up to 2.5 Gbps.

IBM PowerNP NP4GX This network processor, equipped with a PowerPC 400 supervisor processor core, can support links up to OC-48. In the PowerNP [3], hard to predict branches are eliminated because all ALU instructions support predicated execution. Its processing elements, shown in Figure 2.3(a), function at 500 MHz. Each cluster of 4 threads is equipped with a tree search engine co-processor.

Pipelined Architectures In this processing model, packets flow in parallel arrays of processing elements.

Cisco Toaster The Cisco Toasters, as seen in Figure 2.3(b), are used in Cisco's high-end routers, for which they were specifically designed. In those routers, they can be encountered connected as a pipeline of 4 chips, thus agglomerating 64 processors (8 rows of 8 processors) running at 154 Mhz. Programming them is challenging because of the pipeline model used: contention on the column memory is manually avoided by programming with a tight control on the ordering of all the processors requests. Packets can be re-circulated through the pipeline of processors as needed.

EZ-chip NP2 Released in 2004, this processor (Figure 2.3(c)) is an incremental build on its predecessor, the NP-1c, and uses a simple single-image programming model with no parallel programming nor multi-threading. This chip operates at 240MHz and can process packets at 5 Gbps full-duplex. Synchronization among the processor's internal resources and maintaining frame ordering is performed in hardware and is transparent to the programmer. The programmer only needs to provide four functions to program the matching replicated pipelined engines.

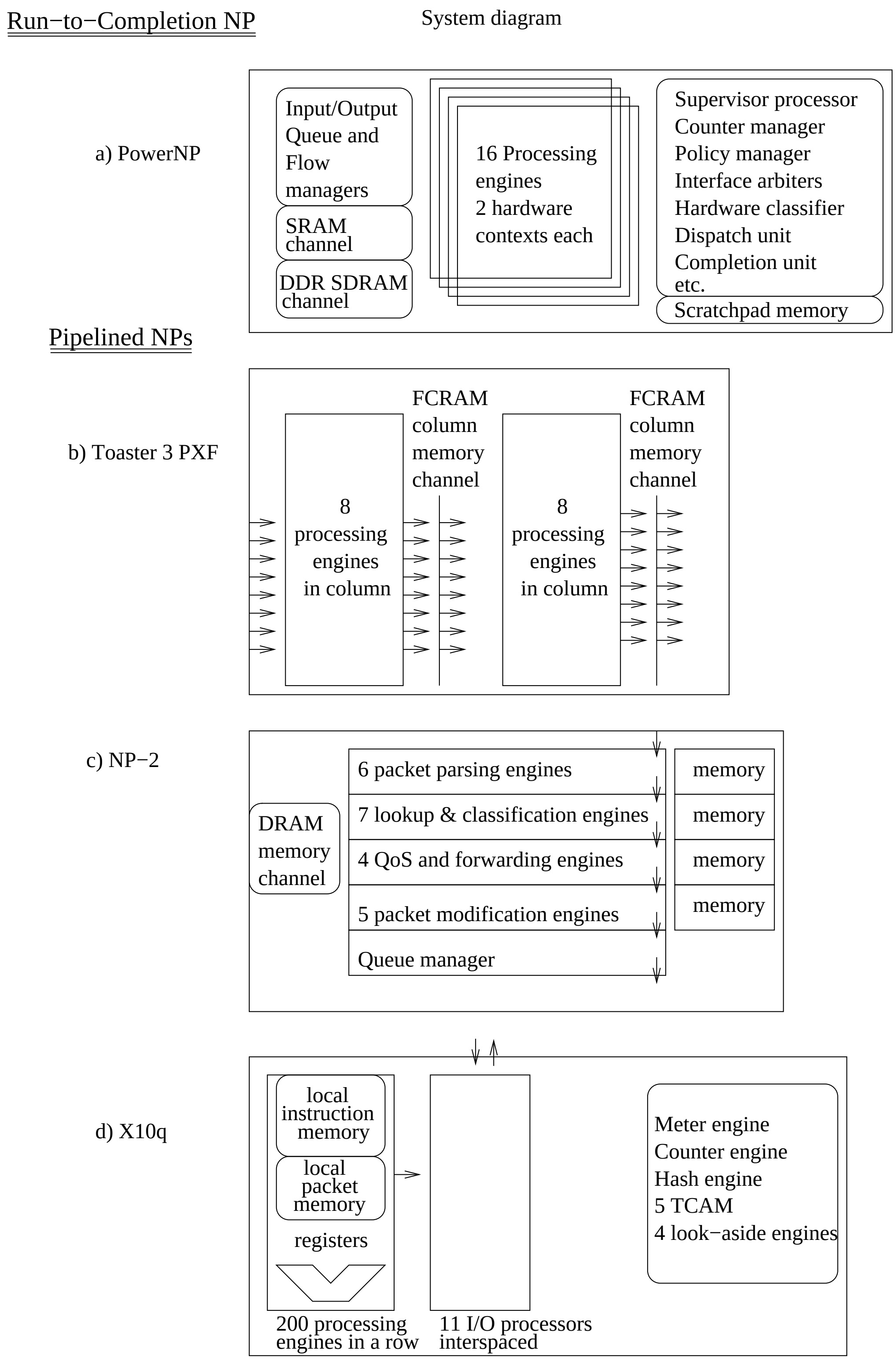


Figure 2.3: A brief survey of network processor organizations.

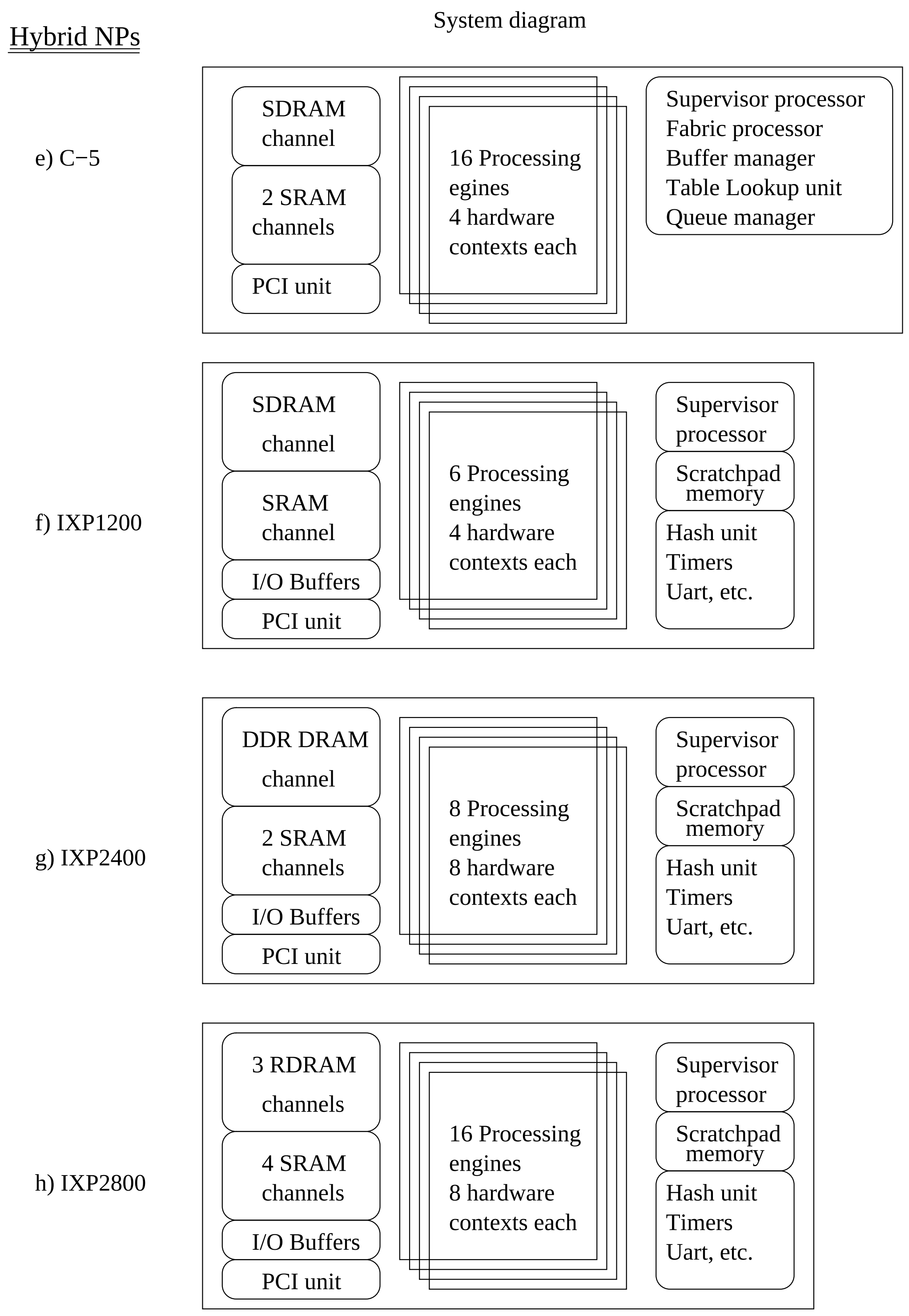


Figure 2.3: A brief survey of network processor organizations (continued).

Xelerated X10 The X10q family of processors [36] offers a deterministic execution through a deep pipeline of VLIW processing elements. Figure 2.3(d) shows only a compressed view of the pipeline. In the X10q, an initial I/O processor is followed by 20 processing elements. The pattern is repeated 10 times and terminated by an additional I/O processor. With processing elements running at 200MHz, the chip can support data rates of 20 Gbps full-duplex.

The Xelerated X11, released in 2005, is a 20 Gbps network processor. In the X11 data flow pipeline, each packet passes through 360 processing engines, leveraging the same architecture as the X10. The amount of logic gates that can be placed inside a network processor, along with latency concerns, are the main limitation so far of data flow architectures for NPs.

Agere PayloadPlus APP540 Released in 2003, this processor, not shown, is composed of a pipeline of a pattern processor for classification and a routing/traffic management processor that executes VLIW instructions. This last processor is programmed using a functional programming language in a single-threaded model. Another processor outside of the packet stream collects statistics for traffic management.

Hybrid Architectures In this model, packets are processed successively by different processing elements that have access to shared resources.

Motorola C-5e This processor [8] can function in 3 modes with hardware support: processing elements can function independently in single stage run-to-completion mode. They can also work in a pipeline, while being fed by a single data stream. This allows to harness the maximum processing power independently of the input rate. The 16 processing elements (Figure 2.3(e)) can finally be aggregated in four even clusters. In that case, the channel processors in a cluster share their instruction and local data memories and can work as a parallel group to handle the one physical network interface, for higher speed interfaces. The C-5 runs at 200MHz and can support rates up to Gigabit Ethernet (1000 Mbps).

The C-5 is organized around three main busses: a ring bus (for inter-processor communications with a bounded latency), a global bus (shared, arbitrated bus) and a payload bus (carrying the payload data and payload descriptors between the engines). The I/O serial data processors contain small processor features plus a CAM and a CRC block.

Intel IXP1200 The IXP1200 is the first of Intel's IXP family of processors [26] that provides a general interconnection of processing elements that share memory. Figure 2.3(f) shows an IXP1200, comprised of a StrongARM processor core and six multithreaded programmable RISC packet processing engines. The processing elements of this processor function at 232MHz and are designed for OC-3 to OC-12 applications. Hardware context switching is controlled in software. Next neighbor register structures allow fast communications across processing elements.

The processing engines share a SRAM and a SDRAM bus. The controllers for these memory channels do optimizations on the order of the accesses to memory, unless the programmer manually specifies otherwise in the individual instructions. Large data transfers of up to 64 bytes can be made in a single reference from the microengines due to large transfer register spaces. The SRAM controller is equipped with a CAM that allows it to create synchronization by allowing one context to access a memory location and putting other requests in a waiting queue. Mutual exclusion can also be accomplished by passing tokens across threads. One optimization that the programmers can exploit in that situation is avoiding to write-back a value to external memory until a group of threads have finished modifying data (this programming strategy is called *thread folding*). Additionally, atomic test-and-set operations are provided inside the SRAM.

Intel IXP2400 This processor is an incremental build on the IXP1200. Its processing elements can function at 600MHz and are designed for OC-48 network access and edge applications. On each processing element of this processor (Figure 2.3(g)), a 16 entry CAM (Content Addressable Memory) complements the register structures to act as a distributed cache unit. In fact, a software controlled cache can be created if a data structure in the local memory is bound to each entry of the CAM. This artificial cache can be used to minimize the latency associated with external memory references. All the threads on a processing element share the CAM, so it can also be used in a coherence scheme, to manage multiple writers to shared data. New features are also introduced with this processor: a pseudo-random number generation, time stamps, hardware support for multiplications and automation for packet byte alignment.

Intel IXP2800/2850 Over the IXP1200, this processor shows a large increase in the number of hardware contexts (from 64 to 256) and of memory channels (Figure 2.3(h)). Each memory channel has a push (read data), pull (written data) and command bus attached to it. There is a set of each of these busses for each cluster of 8 processing engines. The processing elements of this processor [35] can function at 1.4 GHz and are designed for OC-192 network edge and core applications. The IXP2850 is similar to the IXP2800 but with on-chip hardware cryptographic engines.

2.2.2.2 Observations on the surveyed processors

The Motorola C-5 and the IBM PowerNP mentioned previously are at the end of their life cycle and their manufacturers have not released upgrades or direct replacements products. Those processors, backed by semiconductor leaders, are the ones who offered the most powerful and diverse hardware features. One reason for their obsolescence is that they traded flexibility and programmer control: in their system, the programmer had to make a lot more decisions at a global scope versus at a local (processing element) scope, which makes his work much harder.

Dilemmas between ease of programming and hardware architectural features make it hard for a designer to choose any one of the platforms when the tradeoffs are not clear. Also, when presented with a variety of co-processors, it is not clear what are the challenges in programming for performance and what is achievable by automated compilation tools. For most of the chips reported above, it is not possible to compute a cycle budget unless we know the amount of parallelism utilized by the program. This parallelism can be at the thread, instruction or memory level. The maximum packet processing rate supported also depends on the physical media connected (half or full duplex), the number of input ports, and the amount of processing performed on each packets. Interestingly, we can see that from the IXP1200 to the IXP2800, chips with a similar programming model, the cycle budget for the maximum packet rate has decreased from 182 to 67 cycles. So the advertised maximum bandwidth of the processor is in fact a reflect of the workloads provisioned by the chip manufacturer.

In spite of the high degree of parallelism inside the chips, some suppliers have heard and responded to the need of programmers to write single-threaded programs. Agere [59] considers that there is a “huge credibility gap” in network processor programming ease. Indeed, various NPs will have very different programming models and software development kits offering from bare-bone assembly to high level abstractions. Software integration is one of the enabling technologies for network processor wide spread because programming them and maintaining their software is a large part of their total cost of ownership.

2.3 Overview of Previous Research Infrastructures

In this section, we list the major works in network processor systems, that is, the ones that evaluate the performance of an application in the context of network processor architectures.

One of the early works of system research on network processors is from Crowley et al. [15]. In that paper, the authors show that chip multiprocessors and simultaneous multithreaded processors out-perform super-scalar and fine-grained multithreaded processors. The chip multiprocessor that they evaluated had very simple cores each with an instruction issue width of 1. The paper also shows that an operating system over a chip multiprocessor has a negative impact on performance because of the architecture’s inability to execute this sequential code in parallel. Their evaluation was made with microbenchmarks on an architecture with caches, which are not common in modern NPs (as explained in Section 2.2.2.1).

In our framework, we adhere to the trend of having fast and simple cores: we do not attempt to extract instruction-level parallelism or simultaneous multithreading from workloads. Most NPs have adopted multi-core architectures (for example, Nepal [55]). Benefits are in data locality, less contention on shared resources that are also typically slower, smaller instruction stores, and more packet parallelism (thus achieving a better throughput).

Thiele et al. [83] define network processing as a constraint problem where a service curve has to meet a packet arrival curve. Because of several components in our infrastructure that have arrival and service rates, our approach has similarities with this analytical modeling. We borrow from their exploration that is organized around a double loop: an inner one that maximizes the throughput of the network processor under given memory and delay constraints, and an outer loop that performs a design space exploration. However, they do not make architectural conclusions that can be generalized: they try to bind a limited set of architectures, with no memory hierarchy, to applications. We take the opposite approach by trying first to characterize applications and then trying to derive the architectural implications.

The three following framework for the evaluation of NP architectures—along with this work—are based on the Click Modular Router [41]. StepNP [64] uses Click to program individual processors for the purpose of prototyping multiprocessor systems-on-chip (SoCs). While StepNP facilitates a detailed hardware evaluation, it does not easily allow transformation of the input task graph. Crowley and Baer [14] provide a framework to investigate queueing, synchronization, and packet rate control on a single general-purpose processor. We will investigate similar issues in the context of network processor architecture. Finally, Nepal [55] offers the possibility of dynamically mapping an automated fine task decomposition to processing elements. In their work, only one module (part of a task) is active at a time and the others execute speculatively while buffering writes and snooping the memory addresses on the bus, looking for violated dependences. Unlike our study, their work does not evaluate architectural bottlenecks. Also, Nepal's centralized support for task control and speculation makes some assumptions on the hardware that may be considered overly aggressive with regards to contemporary processors.

Two other environments for NP evaluation rely on a very specific architecture. First, the Intel IXP SDK and Architecture Tool [26] are targeted at Intel's IXP family of NPs, and hence provide limited flexibility in varying the underlying architecture. Nepsim [50], being the open-source version of the the Intel SDK simulator, has the same drawbacks. Second, the "Design Space Exploration" paper [27] discusses a low-level particular implementation based on a 'network-on-chip' design where processing is very deterministic. It presents mapping, scheduling and identifies sources of load imbalance. In contrast with their chip that does not support external memory, our high-level programming model allows widely varying latencies.

NP-Click [75] provides a programming model to help bridge the gap between an application description and the low-level use of the capabilities of a specific NP architecture. NP-Click provides manually-coded Click elements that are targeted to the features of the specific underlying NP. As opposed to other works using or proposing a new or architecture specific programming language [25], [10], [22] to simplify the compilation process, we aim for automation and ease of integration. While the authors of Shangri-La [88] introduce a new programming language and try to exploit several low-level well known instruction optimizations, they make a case for compiler support for automation, retargettability, and performance. This idea of integrated, profile-driven, compilation is central in our work, that can be easily extended by additions such as code generation (such as in Wagner et al. [89]) and custom instructions (Wolf [95]).

Also relevant to network processor characterization, power estimations of an NP system are available in such work as Luo et al. [51]. While power consumption is also an interesting computer challenge, our infrastructure focuses on measuring throughput and latency.

Experience with IXA SDK To further understand network processors, we have studied sample programs from the Intel IXA Software Development Kit, in particular running on the IXP2800. For the reader's benefit, we explain some implementations aspects and challenges encountered that can be generalized to other processor families.

On the IXP2800 network processor, each processing element has 8 hardware contexts. Each context has its own register set, program counter, and context specific local registers. Any context can access the whole register file and context switching is completely under software control. A processing element is in one of the following states: ready (a signal has arrived and the processing element is ready to handle it), sleeping (the thread is waiting for an event to occur), executing (or computing), inactive (or disabled). Event signals indicate to each thread that selected events have occurred. This is especially useful to react to the completion of memory reads that are non-blocking, and to detect the arrival of new packets. The programmer must be very careful in not modifying the source (for writes) or destination (for reads) of the memory operations until they actually complete. In the IXP2800, the minimum DRAM physical access length is 16 bytes.

When executing code, the hardware contexts have a thread identifier that allows for thread dependent behaviours. Contexts can implicitly be referred as 'next' when using inter-thread communication. In the SDK, the compiler is in charge of generating code versions for each hardware context. This is useful when, for example, using a C-like language, some code requires per-thread memory allocation. The SDK compiler provides no support for either a data stack or a subroutine call stack. The compiler must enforce a hardware requirement that an instruction can only read or write one register for each of the two register banks, thus occasionally inserting register moves. The on-chip memory controllers are in charge of distributing the memory accesses to the attached memory banks (also known as *striping*) to load balance them.

Intel has recently introduced the concept of structuring applications into ’microblocks’: independent pieces of code that allow the developer to build modular applications. However, communication uniformity and orthogonality are still challenges because of the number of heterogeneous means of on-chip communication. In fact, library code often uses global, shared resources on chip which is conflicting with the goal of having ’independent’ modules.

2.4 Benchmarks for NPs

Network processing is typically performed at three levels, as explained by Ehliar and Liu [20]: (i) core routers demand high throughput such that they usually have few features, (ii) network (or access) terminals, where the traffic rate is much slower but the tasks to execute are more elaborate, and (iii) edge routers, which are a middle ground. To accurately estimate the performance of a processor, computer architects usually rely on measuring the throughput on representative workloads for their chip: benchmarks. Finding the right set of applications to compare or evaluate network processors has been the object of recent research.

Most NP architecture evaluations to date have been based on typical tasks taken individually: *microbenchmarks*. NetBench [56], NPBench [48] and CommBench [96] provide test programs ranging from MD5 message digest to media transcoding. Those suites sometimes emulate packets by simply reading relevant packet fields from a file: their aim is to characterize certain algorithms that are used in the realm of network processing. While microbenchmarks are useful when designing an individual PE or examining memory behavior, they are not representative of the orchestration of an entire NP application. As explained by Tsai et al. [84], kernels may not expose bottlenecks. One reason for that is that they do not exploit any form of parallelism.

In this work, we opt for application-level benchmarks and use the Click Modular Router [41] as a building block (described in section 2.5). Click has been widely used as a base application for performance evaluation in such works as [14], [64], [75], [73]. However, we are the first to provide automated compiler analysis and transformations for it. Our work could be generalized to other router frameworks that are similar in that their applications are a composition of an extensible suite of modules (examples include VERA [40], PromethOS [72] and Router Plugins [18]).

2.5 The Click Modular Router

Click [41] is a modular software architecture for creating routers. Click is part of the XORP [29] project that has for mission to develop an extensible router platform by addressing the challenges of making open-APIs for routers and allowing researchers to prototype and deploy experimental protocols in realistic environments. Click acts as the packet forwarding path: i.e. the software component that handles the packets. Over other research software for routers, Click was designed with four major concerns in mind: long feature list, extensibility, performance and robustness. By design, Click [98] is not limited to be run on commodity PC hardware but could run on a PC augmented with network processors doing the bulk of the packet processing, or, in the future, in high-performance traditionally ASIC based, core routers. In the original evaluation of Click [41], it was shown that Click could compete advantageously against the Linux operating system for routing.

Click is built from fine-grained software components called *elements*. These elements have a common interface allowing for initialization, user interaction and mainly packet handling. Elements are linked using *connections* that represent possible packet paths. Packet processing on a connection can be initiated by the source end of the connection (*push processing*) or by the destination end (*pull processing*). The motivation for the pull action is to let an element (for example, a transmitting interface) decide when it is ready to receive a packet so that it is not overflown, and looses the ability to control the buffering. Each connection end must be used as a push or pull input or output interface

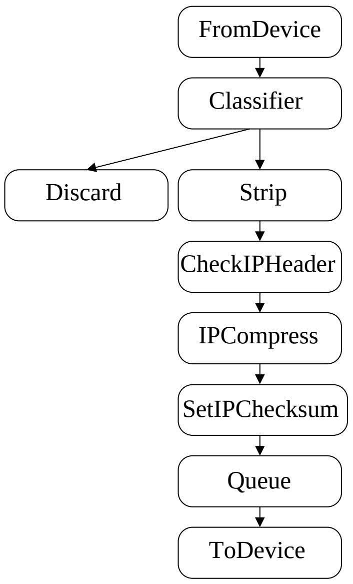
```
c0 :: Classifier( 12/0800, -);

c0[1]
      -> Discard;

FromDevice(DEVICE_A, 0)
      -> c0;

c0[0]
      -> Strip(14)
      -> CheckIPHeader
      -> IPCompress(LEVEL 1)
      -> SetIPChecksum
      -> Queue(2048)
      -> ToDevice( DEVICE_B );
```

(a) Configuration Script



(b) Result Task Graph

Figure 2.4: Very simple IP compression application in Click.

exclusively. In Click, any packet transfer routine must return to its caller before another task can begin. After being processed inside an element, packets must either be handed to the next element, stored or destroyed. Packets should not be used after being passed along to another element.

Figure 2.4 exemplifies a very simple configuration that compresses valid IP packets. First, we define a `Classifier` to distinguish IP packets from the others (IP packets have the pattern 0800 starting at byte offset 12, inside the Ethernet header). Non IP packets are sent to a `Discard` element. Others follow a chain of `CheckIPHeader` (that implicitly discards corrupted IP packets), `IPCompress`, `SetIPChecksum` (the checksum has to be recomputed as the payload changes) and `Queue`. The `Queue` buffers packets for the transmitting interface DEVICE_B: queuing is explicit inside Click. To simplify experimentation, packets can flow in an out of trace files using the elements `FromDump` and `ToDump` (instead of `FromDevice` and `ToDevice`).

Click was implemented in order to run efficiently on uni-processor as well as shared memory systems [11]. For this reason, packet descriptors can only exist on a straight-line, sequential chain of elements: branches in the task graph require a call to a function that creates a new packet descriptor.

As well, all packet writes are guarded by a method that uniqueifies the packet buffer (in case more than one descriptor would share the packet buffer).

2.6 Summary

In this section, we have presented several network processors currently on the market. In most of them, because of the large number of packets in treatment at the same time, the programmer must specify, for example, when memory operations need to be ordered, allocate and handle signals and make use of a plethora of optional tokens in the assembly language, for example, to hint branch prediction. Because of the complexity involved, the programmer must typically revert to modifying sample applications and use library code that takes over pre-determined shared resources on the chip.

The key benefit of programmability is conditional processing: being able to ask a processor to accomplish unconstrained applications is central to making these multi-processor ASIPs successful. Our work will illustrate the need for deep application understanding for a correct and efficient resource and task allocation. To avoid the common pitfall over-biasing application characterization towards certain architectures, we will select the main characteristics out of the surveyed processors and ally the flexibility to simulate arbitrary architectures. The architectural components of interest are both at a fine granularity (for example, non blocking instructions and synchronization) and at a coarse granularity (for example, processing elements, memory and interconnect). We also showed in this section that application-level benchmarks are best to explore compilation in the realm of network processors. We presented the Click Modular Router that we use as a building block. In the next chapter, we will describe how we plan to transform our Click benchmarks to execute efficiently on a network processor.

3 Towards a Compilation Infrastructure for NPs

As NP architectures become more complex and contain a greater variety of computation and storage resources, the task of efficiently programming them by hand becomes intractable. This chapter addresses our aim at making network processors more accessible by making them easier to program. To make this possible, the programmer should work in a popular high-level language that is automatically transformed to use the low-level intrinsics of NPs. To give a structure to this endeavor, we use as a starting point a basic programming model that minimizes constraints on the programmer.

Instead of using low-level assembly routines, we would rather the programmer be able to express the application as a **graph of tasks** (that can contain branches and cycles), written in a high-level language. The compiler infrastructure would then map tasks to processing elements (PEs) and memory resources in the underlying NP, identifying memory types and increasing the parallelism specified in the original task graph through transformations. Ideally, the compiler would also automatically insert all synchronization, signaling, and manage memory, allowing the high-level application to scale up to the available resources in the NP.

Task graphs are a well accepted way of representing parallel applications: examples include the Cilk project [6] for multithreaded applications and the POEM project [2] targeted at distributed message passing systems. However, one of the differences with these projects is that our task graph describes the sequential processing of a packet and the programming model allows us to parallelize the application using the techniques described in this chapter. We take advantage of the fact that the Click Modular Router, introduced in Section 2.5, provides a large library of predefined network processing tasks, called elements, that are meant to be connected in a task graph. This modular property of Click allows us to create a wide variety of applications on which we can directly apply the techniques that we present in this work.

In this chapter, we first consider packet ordering and task dependence issues. Then we describe in more detail the task transformations that we intend to perform on a task graph and the implementation techniques involved both in a compiler and in hardware. After this description of concepts that are more general than our particular implementation, the last part of this chapter will introduce the actual composition of our infrastructure.

3.1 Packet Processing Order

Before introducing parallelism in sequential tasks, we need to make sure that we are not changing the behavior of the network processor in a manner that is incompatible with its initial purpose. More specifically, to ensure that our compilation infrastructure makes viable transformations, we need to adequately answer the following questions:

1. Must the packet ordering be identical at the input and at the output of the NP?
2. Does the result of a task processing a packet depend on the order that this task has seen the packets arrive?

As an answer to the first question, *RFC 1812 - Requirements for IP Version 4 Routers* states that the Internet was designed to tolerate packet reordering but that ordering should be preserved as much as possible. RFC3366 makes the distinction between global packet order and per-flow packet order. A *flow* is a set of packets having the same characteristics, usually, the same origin and destination. Packets belonging to different flows are often re-ordered when the router performs some kind of policy-based sharing of a link. On the other hand, intra-flow reordering may incur retransmission of packets if a network protocol layer interprets that some packets have been lost. Intra-flow reordering may also increase the amount of buffering required for the clients and will increase the jitter for real-time applications (for example, voice or video). In fact, packet ordering requirements mostly depend on the application. For example, in the the IPComp standard packet compression scheme (RFC3173) the compression task for each packet is independent, i.e. there is no persistent state across packets. Conversely, there are several applications that benefit from the

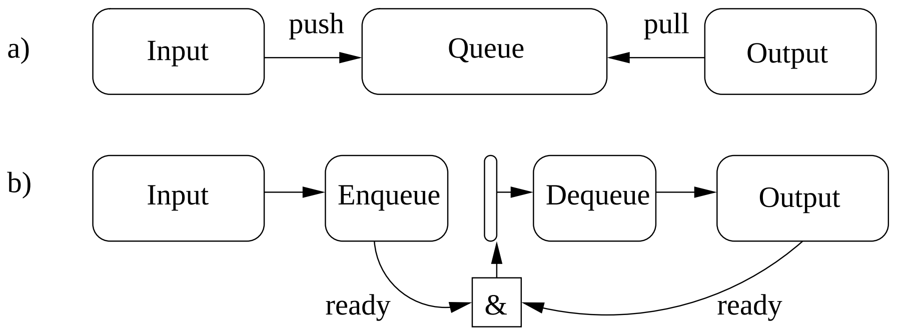


Figure 3.1: Minimal task graph showing (a) the original application diagram with push and pull (b) the transformed application with push and rate controlled pull.

preservation of packet ordering, such as web client that interprets an hypertext page from the first to the last line.

To preserve packet ordering while not being overly conservative in the common case, we assume a mechanism for the application developer to specify that ordering be preserved at a given point in the task graph. In Click, the `Queue` element is used to manage buffering by having a *push* handler to enqueue packets and a *pull* handler to dequeue packets as shown in Figure 3.1(a). In our infrastructure, we take the convention that the `Queue` element can enforce the packet ordering, in which case, we postpone the ready signal from the `Enqueue` operation until the next packet in order is enqueued, as illustrated in Figure 3.1(b). As in most real network processor applications, our applications precede all their output ports by a `Queue` element. At the simulation level, we model a structure similar to a jitter buffer [79] that implements packet ordering with a complexity of O(1). Sorting packets on a per-output interface basis, while decentralized, is complex because not all packets entering the NP will be sent to an output interface of the NP, and some may also be discarded. We implement a “best effort” ordering by inserting signaling in the application code. Those signals create a sorting place-holder on a `Queue` as soon that it is determined on which interface the application will output a packet. Hence, the point from which the order is guaranteed for a packet is given by simulation feedback of the used paths in the task graph.

The answer to the second question, pertaining to the requirement of tasks in the NP to process packets in order or not, is also difficult. Taking for example a common classification engine, classification is typically based on the source and destination addresses of a packet as well as the source and destination ports. Packets in this case have no dependence on each other. In a Network Address Translation application, typically centered around a classification engine, TCP header fields of packets are modified according to a per-connection port remapping. A mapping is created for every flow, every time a new flow identifier is needed to characterize incoming packets. The only aspect of this application that depends on packet order is that the destruction of a mapping is usually scheduled a short time after the packets indicating the end of a TCP connection are observed. In fact, it is common for the application to delay the destruction of per-flow state to ensure that all packets in the flow have been processed.

Thus, we have observed in our applications that most elements have a behavior independent of the order of packets processed. Hence, we advocate that as long as sequential task dependences are respected and application semantics preserved through the special meaning of the `Queue` element, packet ordering in the processing is not required. Other work simulating packet level parallelism (such as ILP and SMT studies in Crowley et al. [16]) also assume that packets can be processed in parallel usually without regard for their ordering.

At the simulation level, reordering packets may not exactly reproduce the same sequence of events that we could observe in the original application: for example, a threshold condition could be reached on a different packet. Since we process the same number of packets as the original application, we can assume that the overall performance evaluation is correct as well as the overall code semantics.

When deciding if parts of the packet processing can be re-ordered, we find in Click's source code that this is not always possible because some data can be communicated between tasks in the meta-data accompanying the packet. We could try using the commutativity test [69] to experimentally determine if processing tasks out-of-order on packets leads to, for example, the same resulting routings or packet annotations. However, for this work, we assume that when there is communication that no re-ordering is allowed.

In this section, we explained how our infrastructure can respect ordering requirements inside and at the edges of the NP as long as it is specified by the programmer. Preserving the input packet

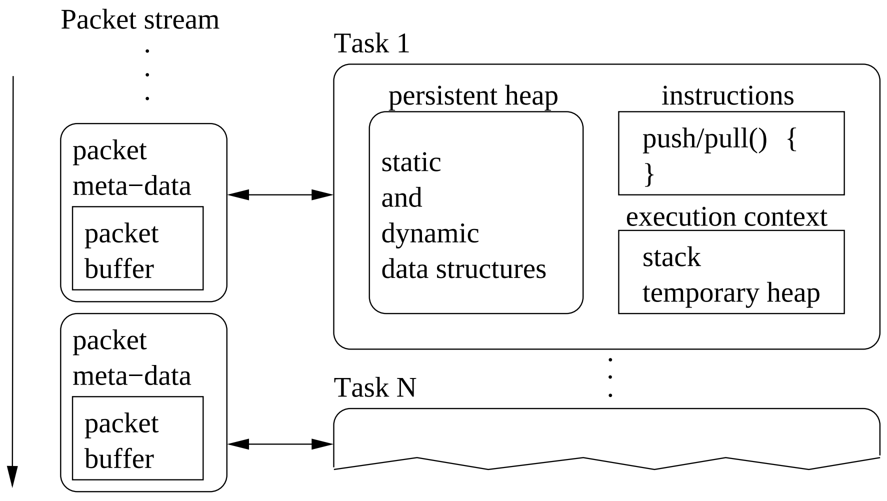


Figure 3.2: Classification and separation of different types of memory, assuming a Click-like programming model.

ordering at the output of an NP can be beneficial to certain client hosts that have little buffering capabilities to reorder packets. The tasks that we consider in this work are packet ordering agnostic. Finally, the order of the tasks that process a packet must be respected where inter-task communication exists. The next section expands on the types of transformations that can be performed on the task graphs that we consider and describes the compiler support that enables these optimizations.

3.2 Managing Memory

A complete compilation system for NPs requires automated memory management. In particular, the compiler must map the different variables and data structures used by the application to the various types of storage available in the NP architecture. Ennals et al. [23] agree that memory typing eases parallelization and that automated task optimizations are essential to map an application efficiently to a specific network processor architecture.

Figure 3.2 illustrates the different types of memory accessed when using a programming model similar to Click's, for which we define four categories. First, there are the instructions that comprise a task, which are typically read-only. Second is the *execution context*, the data which is private to a

Table 3.1: Example storage types available in Intel IXP NPs.

	Per PE				**Chip-Wide**			
Processor	**# Contexts**	**Registers**	**Local Mem.**	**# Instructions**	# PEs	**Scratch**	**SRAM**	**DRAM**
IXP1200	4	512B	0	2K	6	4KB	8MB	256MB
IXP2800	8	2400B	2560B	8K	16	16KB	256MB	2048MB

task such as its execution stack, registers, and any temporary heap storage. Third is *persistent heap data*, which is maintained across instances of a distinct task. Fourth is packet data, including the actual packet payload as well as any meta-data attached to the packet by tasks. In a programming model such as Click's, the only way for two distinct tasks to communicate is through this packet meta-data.

The challenge is to map each of these types of memory to a memory unit available in the target NP architecture. Examples of different storage types and capacities for two Intel IXP processors are given in Table 3.1, of which there is evidently a large variety. The mapping of application storage to architected storage is described in further detail in Section 6.1. Given this typing of the memory storage of our applications, we next present two optimizations on packet processing: one intra-task and one inter-task. We will later refer to both together as the *locality transformations*.

3.2.1 Improving Locality Through Batching Memory Requests

The idea of improving an application's locality by limiting the number of long latency accesses to memory has been examined in the "Data Filtering" paper [54]. In this project, the authors propose a coprocessor physically adjacent to the off-chip memory interface. This coprocessor offloads PEs from instructions that access data with low locality (history based), thus limiting the accesses on the bus between the PEs and the off-chip memory. The authors made their experiments with a different methodology and different goals from ours (they did not have any memory typing and they were trying to optimize power) and they were using specialized hardware support. Sherwood et al. [76] shows the benefits in using wide word memory transfers but perform their evaluation on a novel memory controller. This work presents task-level optimizations in an integrated compiler/simulator context with realistic evaluation on an NP, where our first concern is packet throughput. We next present in what context limiting the number of memory accesses is useful and how we implement this technique.

In a typical mapping of a network application to an NP, the packet data is mapped to SDRAM, a memory with a large latency but high throughput. To help tolerate this latency and to reduce request traffic, NPs such as the Intel IXP typically support wide memory operations. The programmer is expected to create large memory requests and accesses data at a finer granularity once the data is brought closer to the processor. Compiler support for managing memory must be aware of this ability and automatically target wide memory operations when accessing a large data structure, or when accessing several small but consecutive memory words. Once a block of memory is transferred to local storage, a processing element (PE) can have more fine-grain access to the data. In a way, batching memory requests implements a form of software-managed prefetching. A good example where this applies is the IP header (20 bytes on average) that needs to be fetched for the checksum validation, for processors with no CRC hardware support. Tan et al. [81] measure an improvement of over 50% in throughput for different compression algorithms by manually transforming the code to issue wide memory requests (called “memory bursts”).

When implementing batching of memory accesses, we first identify tasks that consistently access memory locations that are not local to the processing element. Our approach is to group all those memory locations and send batched requests when the task starts execution. The results of the batched memory accesses are stored locally to the PE. When the task later performs fine grained accesses to the recurrent memory locations, the accesses are remapped to the local storage. Because persistent heap (data local to a task) should not be accessed outside of a synchronized section, we avoid any prefetching of it.

Studies with no memory management nor compiler support would be limited to perform individual memory loads. Batching is an automated transformation that makes our simulation more realistic by using the available NP hardware to issue wide word bursts of memory accesses. Those transfers are done in a non-blocking fashion. This means that the processor stalls until the whole buffer arrives only if the prefetch operation does not complete before a finer access to the buffer is issued. We next present another optimization, enabled by batching, that launches memory operations earlier in the packet processing.

3.2.2 Memory Forwarding

To further capitalize on batched requests, if two tasks make batch accesses to the same portions of a packet's meta-data or payload then that data can be forwarded directly to the second task from the first—potentially saving on memory traffic and latency. To save time, the data is forwarded before it is even requested; otherwise, there is no guarantee that it will stay stored in the processing element where the data is currently available. If the destination processor cannot be determined because the next work unit remains to be scheduled, then the data is saved in a shared on-chip memory, also known as the scratch-pad. We implement batching and forwarding using profiling in the simulator; more details are given in Section 5.1.5. The process consists of first identifying memory accesses that can be batched, and second deciding which tasks in sequence make use of that same data.

3.3 Managing Tasks

Because we propose that the NP application be described in a task graph, the program specification does not present any form of parallelism in its initial format. Giving the illusion of programming a machine where everything happens in sequential order greatly simplifies the programmer's work by removing the need to perform any dependence management. Based on the dependences between tasks and the memory typing that we present next, it is possible for the compiler to re-organize the task graph automatically and insert appropriate synchronization to exploit available parallelism in the network processor. In this section, before giving details on our approach, we first present how it improves on related studies discussing NP task-level management.

3.3.1 Contrast with Related Work

One problem we have to address in the management of tasks is the imbalance in latency of different tasks. This imbalance can create large idle gaps in the task schedule. For example, if a task assigned to a single processing element (PE) has a disproportionately long latency with respect to the other tasks, it can hold back the processing of multiple packets. In that case, a limited number of processing elements will appear to be busy when there is contention on a particular task. This contention can be the result of three factors: the task breakdown, called *task partitioning*, the dependence management scheme, and finally, the *task scheduling* that describes the temporal order of execution. We next give some background on those three concepts and relate it to our work.

3.3.1.1 Task partitioning

Other work targeted at transforming tasks automatically is found in Weng et al. [92] and Nepal [55]. In both papers, applications written in a high-level programming language are partitioned into modules, i.e. groups of instructions or basic blocks. Weng at al. present a greedy algorithm to create modules based on a maximization of the computation over the communication ratio of the application instructions. Next, the authors use a trial and error algorithm to assign modules to processing engines. Only one module is mapped to each PE, meaning that a large number of processing elements can be required. It is questionable how much additional inter-modules communications overheads an implementation on real hardware of this partitioning would incur. In fact, it is often not possible to determine statically if there can be a dependence between two memory accesses, thus limiting the authors to conservative assumptions.

In this work, we favor applications that are derived from realistic network processing applications such as the ones found in the Packetbench suite [67]. Click's programming model allows us to have a realistic breakdown of tasks and accurately model accesses to dynamic data structures. Instead of approximately reverse-engineering dependences created by the loose usage of global data structures in an application, often found in `C` programs, we start from tasks that follow general requirements (explained in Section 3.2). We exploit this task modularity in the form of Click elements to investigate other task transformations.

3.3.1.2 Dependence management

There are three known ways to handle dependences between tasks: (i) use speculation and dependence violation detection hardware as in Nepal [55]; (ii) insert synchronization in the task instructions; and (iii) perform pipelining. Pipelining an arbitrary program, i.e. breaking it into components that run in isolation, is very restrictive. Figures 4.3 and 4.4 show examples of code where almost the totality of a task has to be executed in sequence because of required synchronization. This synchronization constraint greatly limits the pipelining possible.

3.3.1.3 Task Scheduling

The problem of scheduling in the presence of variable paths, and possibly heterogenous processing elements, does not have a lot of theoretical background. An example approach, called “minimum makespan scheduling” [87], consists of assigning jobs to machines so that the completion time, also called the makespan, is minimized. Most static scheduling papers, as surveyed by Kwok et al. [45], consider task graphs having tasks with fixed or predictable latencies and no conditional branching. Hence, static scheduling usually does not consider task graphs with loops. Authors also usually rely on the fact that tasks start after their predecessor tasks complete, a requirement known as the “frame separation property” [80]. Because Click elements can take dramatically different latencies, the usual method for schedulability analysis, i.e. approximating a piece of code by its worst case behavior, does not work. In conclusion, the scheduling of our task graphs is best addressed by dynamic scheduling, i.e. by scheduling a task as soon as the required data and execution resources become available.

The CUSP project [73] claims that supporting tasks with non-deterministic durations allows programmers to focus on functionality rather than complicated timing analysis. While supporting varying task latencies, we will provide measurements allowing to find bottleneck tasks. We limit the scope of this work to static task assignments to processing elements and a steady distribution of packets on the task graph (on the paths taken). With the following task transformations, we uncover a wide solution space where transformations can be combined. Simulation will uncover the different performance tradeoffs. We now investigate the implementation of each of these transformations

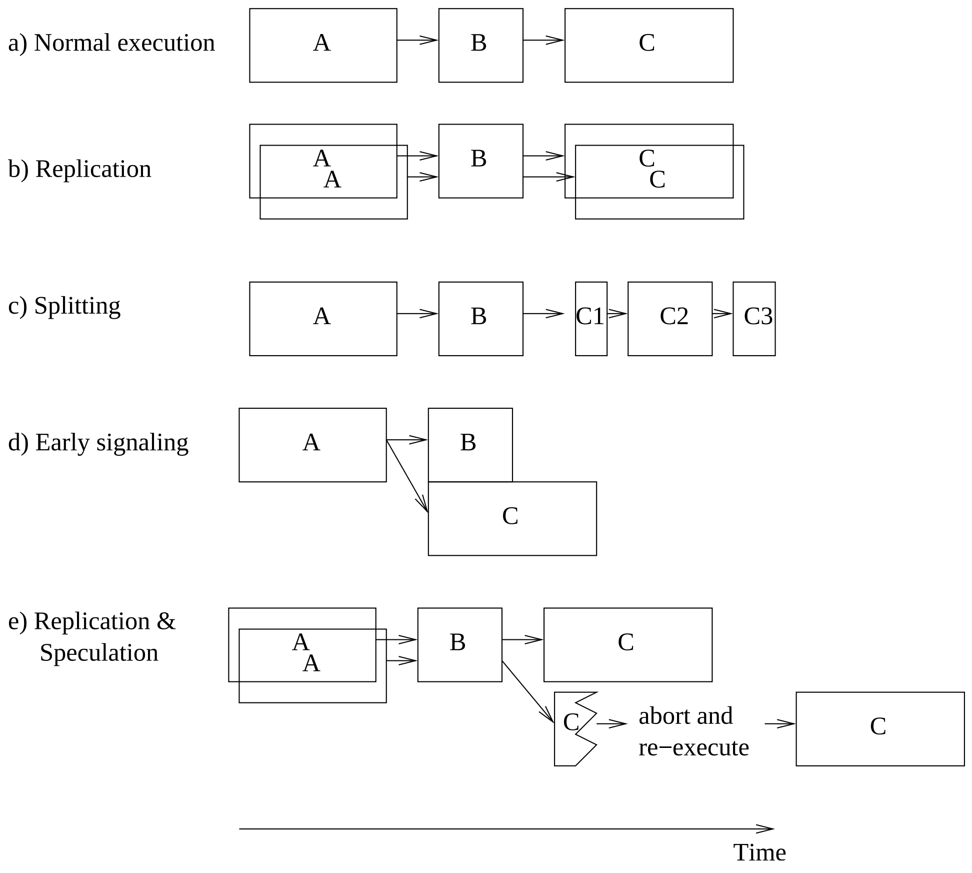


Figure 3.3: Task transformations to increase parallelism; `A`, `B`, and `C` are each distinct tasks.

in greater detail.

3.3.2 Proposed Transformations

The goal for automated task management in a compiler is to increase parallelism and hence throughput by (i) improving the potential for tolerating memory latency, and (ii) scaling the task graph to exploit all available processing and memory resources. Figure 3.3 illustrates four task transformations for increasing parallelism in the task graph of a network processing application. We first give a brief overview of the transformations to clarify the difference between them and we describe each of them in more detail below. Our work has led us to consider other possible task transformations that we present as future work in Section 7.2.1.

Figure 3.3(a) represents the normal, sequential execution of three distinct tasks `A`, `B`, and `C`. To increase parallelism, and to allow a minimal task graph to scale up to a larger number of PEs and hardware contexts, we employ *task replication* (Figure 3.3(b)): in this case, tasks `A` and `C` each have a replica. Replication can be used to increase the throughput of a bottleneck task, but can be limited by intra-task dependences.

We use *task splitting* to improve load balance by breaking a large task into smaller tasks, allowing the new task splits to be scheduled on multiple PEs. Figure 3.3(c) shows task splitting applied to task `C`. Splitting differs from pipelining because the task splits execute in order, with no temporal overlap. Next, we consider *early signaling* : when one task is guaranteed to be executed after another and the tasks have no dependence between them, their execution can be aggressively overlapped. In Figure 3.3(d), task `C` is signaled early by task `A`, permitting greater parallel overlap. Finally, Figure 3.3(e) shows that speculation can be used to schedule tasks with dependences, for example, two replicas of task `C`. Speculation ensures the correct execution of the two replicas by aborting and re-executing a task that would violate data dependences between the replicas.

Later in this section, we describe several compilation methods for addressing these challenges, which we later implement and evaluate in the NPIRE infrastructure.

3.3.3 Task Dependences

We now present the vocabulary used to describe different ways of organizing the tasks on the network processors. We also present new kinds of dependences that occur.

The typical NP-specific meaning of intra-packet and inter-packet dependences as well as examples of shared data are given in Henriksson [31]. Table 3.2 shows the impact of the application distributing work to processors at different granularities in a network processor. The conclusion is that extracting more parallelism generally leads to more shared data among threads of execution.

Before we can investigate methods for transforming tasks to increase parallelism, we must first understand the different forms of dependences between tasks and the memory locations where they occur, as summarized in Table 3.3. A benefit of using a programming model such as Click is that the only way for two distinct tasks to communicate is through packet meta-data, or potentially through

Table 3.2: Breakdown of dependences in a network processor. Source: [31]

Partition scheme	**Shared data**	**Drawback**	**Advantage**
Thread-per-Message	Connection state	Shared connection state	Flexibility, good load balancing
Thread-per-Connection	None	Bad utilization and saturation	No shared data
Thread-per-Protocol	Packet	Shared packet data	Specialization is possible
Thread-per-Task	Packet and connection state	Shared packet data and connection state	Possible latency reduction

Table 3.3: Potential dependences in memory.

Dependence Type	**Dependence Location**
Between distinct tasks	packet descriptor packet payload
Between task replicas	persistent heap
Within a task and between task splits	stack temporary heap persistent heap packet descriptor packet payload
With an early-signaled task	(none)

a modified packet payload. However, if we attempt to replicate a task, there will be potential dependences between replicas through the persistent heap (for example, if a task increments a persistent counter for every packet). Hence, at a point in the task graph when packet ordering does not matter, dependences between task replicas are *unordered*. In other words, the order of execution of the task replicas does not matter so long as they execute atomically with respect to the shared persistent heap. This atomic execution can be accomplished through the proper insertion of synchronization, as described below in section 3.4.1.1.

In contrast, when we attempt to *split* a task, i.e. partition a task into a number of sub-tasks, we

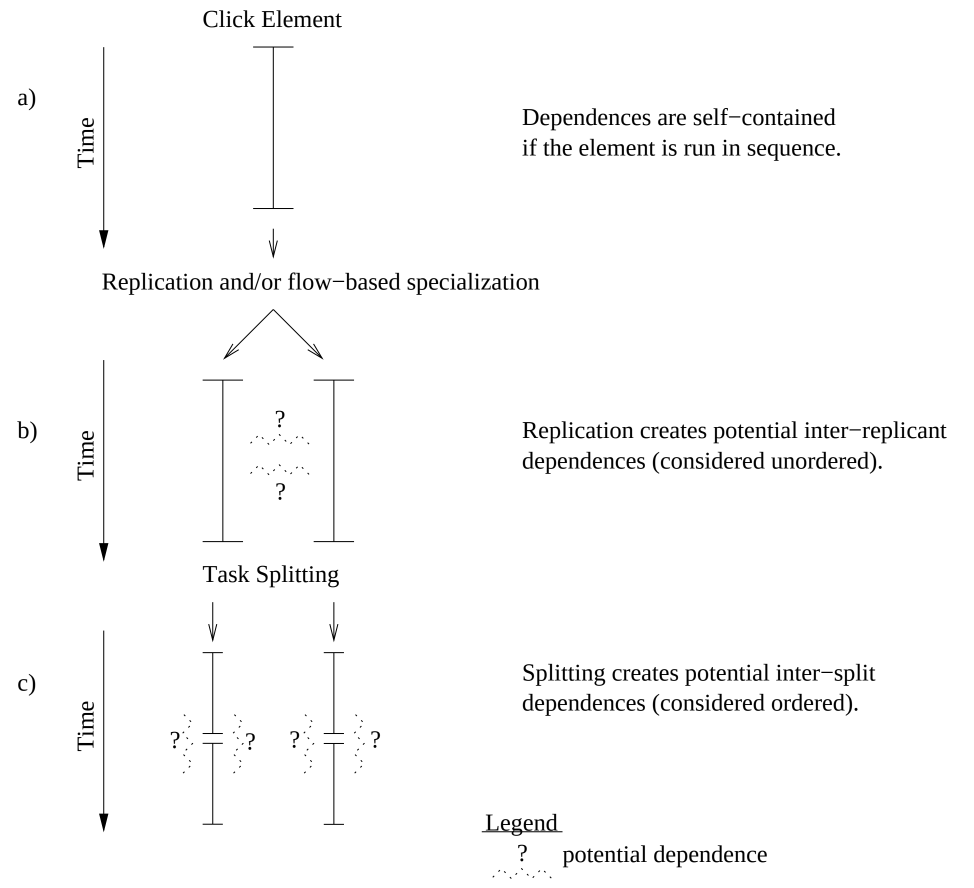

Click Element
a)
Time
Dependences are self−contained if the element is run in sequence.
Replication and/or flow−based specialization
b)
Time
?
?
Replication creates potential inter−replicant dependences (considered unordered).
Task Splitting
c)
Time
?
?
?
?
Splitting creates potential inter−split dependences (considered ordered).
Legend
?
potential dependence

Figure 3.4: Unordered and ordered dependences encountered.

must preserve any of the original dependences within the task that now cross task-split boundaries. As shown in Table 3.3, these dependences can exist in any of the storage locations used by a task, and are therefore much more difficult to manage. Also, these dependences are *ordered*, since the results of the first split must be forwarded to the second task split as input. Figure 3.4 shows the dependences that arise from replication and task splitting. Finally, it is possible to re-order tasks that have no dependences between each other. In summary, the synchronization cases can be categorized as follows:

- ordered operations require wait and signal;
- unordered operations require lock and unlock.

This classification of dependences based on memory typing determines how task transformations can be applied. We next explain our automation through compiler support to exploit parallelism, as shown in Figure 3.3.

3.4 Implementation of Task Management

In this section, we present the compiler and hardware support, where appropriate, to support the four task transformations proposed in Section 3.3.2: replication, splitting, early signaling and speculation.

3.4.1 Task Replication

For a task with only unordered dependences between instances (through persistent data structures), we can increase parallelism by replicating the task. A task and its replica(s) (i) can occupy two hardware contexts on the same PE or occupy two separate PEs, (ii) can share an instruction store (if on the same PE), and (iii) share memory for persistent data structures. The challenge for supporting replication is to automatically insert synchronization for accesses to shared persistent data structures, so that replication remains transparent to the application programmer.

3.4.1.1 Synchronized Sections

When having multiple replicas of a task running at the same time, we have to introduce either atomic operations or synchronized sections to preserve consistency on shared memory locations. Every time a task accesses a memory location that is potentially written to at some other point in the program, the task must acquire a lock to operate in isolation of other tasks. This is a requirement to preserve correctness of execution. As shown in Table 3.3, the only memory type that may require synchronization is the data that is local to a task (for example, a counter).

Dependence Identification The problem of identifying the dependences in the code refers to pinpointing the memory reads and writes that access shared memory locations. One way of proceeding would be to identify at simulation time the memory accesses that lead to dependences and feed them back to the compiler. Our approach is to take advantage of the exact memory types of data accesses that are fed back into the application being compiled. This analysis is hence more exhaustive because it allows us to see dependences even in code not reached by the particular packet trace(s) used for simulation. As we will explain, our compiler technique, that we call *buffer shape analysis*, finds all instructions that can access a given data location. Other than identifying potentially aliasing memory accesses, we envision that this compiler pass can have other future applications, such as directing data layout optimizations. Also, because this compiler analysis, as we explain next, is given the type of memory buffers accessed by instructions and puts the emphasis on understanding data structures, it is different from pointer analysis [32].

In our compiler pass, we start by discovering data structures that are accessed within a task. This discovery can be formulated as intra-procedural analysis (see Section 4.3.1.1). To identify pointer aliases and data access patterns, we attempt to understand the layout of buffers accessed inside a task. Our analysis discovers data members and pointers to other buffers inside data structures. Figure 3.5 shows a graph that would be created while analyzing the code of a simple linked list traversal. As shown in the figure, because this work is performed in the compiler, we have to handle numerous temporary variables. Pointers that access arrays are distinguished by their index inside the array. If the index is computed, then we conservatively assume that it may refer to any item of the array. We also add edges to account for recursive data structures and to show *boxing*, i.e. encapsulation of data within other data structures. Our compiler work only needs to consider a single task at a time. This work is based on the assumption that the top level buffers in the graph that we build do not alias. This is a valid assumption inside Click: packet buffers are guaranteed to be unique when they are written to.

In our analysis, the only variables considered are the ones bound to memory accesses to the “permanent heap” (see Figure 3.2). For each variable read and written to the permanent heap in a task, there is a possible unordered dependence with a replica of that task. To manage those dependences, the compiler needs to insert unordered synchronized primitives (lock/unlock), as we describe next.

```
struct datum {
  char* start;
  char test[16];
  struct datum* link;
  int a;
};

int main(void)
{
  struct datum* one, *head, *crit, *prior;
  prior = new struct datum;
  prior->link = NULL;
  one = new struct datum;
  one->link = prior;


  while(one->test[one->a+1] <= 10)
    {
      while(one)
        {
          one = one->link;
        }
      one = head;
    }
  printf("int %d\n",one->a);

  return 0;
}
```

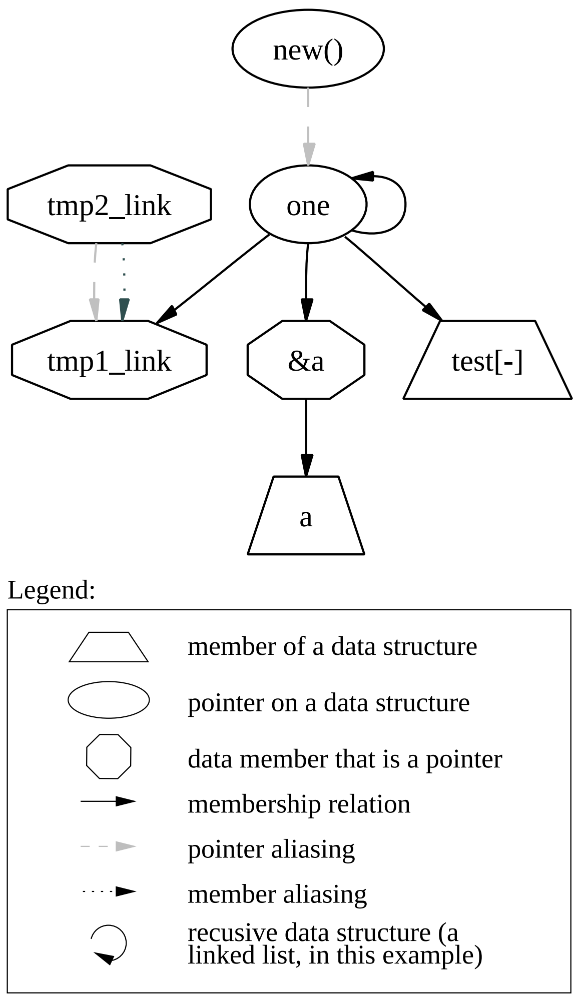

new()
tmp2_link
one
tmp1_link
&a
test[-]
a
Legend:
member of a data structure
pointer on a data structure
data member that is a pointer
membership relation
pointer aliasing
member aliasing
recusive data structure (a linked list, in this example)

Figure 3.5: Buffer analysis, simple test case.

Placing Synchronization Markers We use simple lock primitives that can be implemented by traditional atomic instructions. We insert synchronization such that:

1. the task acquires a lock before the first read or write to a given shared location;
2. the task releases the lock after the last read or write to that location;
3. if any critical section partially overlaps with another, both critical sections will be combined into one.

Because we may need to place a marker in a position that post-dominates all basic blocks, we need to unify all return paths of the function considered into a single basic block before attempting the synchronization placement.

With this scheme, it becomes mandatory at run time to ignore duplicate locks and spurious unlocks (otherwise we might, for example, lock the holder of a lock). No locking/unlocking pair should be put inside loops because this would not protect the variable across loop iterations. Finally, because a processing element can only hold one lock at any given time, this placement strategy is deadlock-free.

While more aggressive or more fine-grained locking strategies are possible, this method has the benefit of avoiding deadlock situations. More advanced approaches might attempt to decrease the size of the critical section through instruction scheduling [99], implementing thread folding [26], or possibly converting the code to non-blocking algorithms [57]; all of those three techniques are however beyond the scope of this work.

3.4.1.2 Context Eviction

Because of a potentially high level of task replication across all processing elements, there may be several tasks waiting to acquire a lock. All tasks waiting to acquire a lock occupy hardware contexts without performing any work. The solution we implemented was to preventively evict any context in a locked state when other packets are waiting to be processed. We save their register space and restore it upon acquisition of the lock. We impose no constraint on space to save the context's state but, in the cases observed, the number saved at any one time is bounded to a reasonable number.

3.4.2 Task Splitting

To improve load balance, we can break a large task into smaller tasks through *task splitting*, allowing the new task splits to be scheduled on multiple PEs. To split a task requires an analysis of all dependences between the two task splits, which we will refer to as the *producer* and the *consumer*. Data values for any true dependences (read-after-write) between the producer and consumer must be forwarded to the consumer task split. Furthermore, if a replicated task is also split, any locks held across the split point must also migrate from the producer context to the consumer context.

Splitting is expected to be useful to allow for finer scheduling of tasks. However, we also have

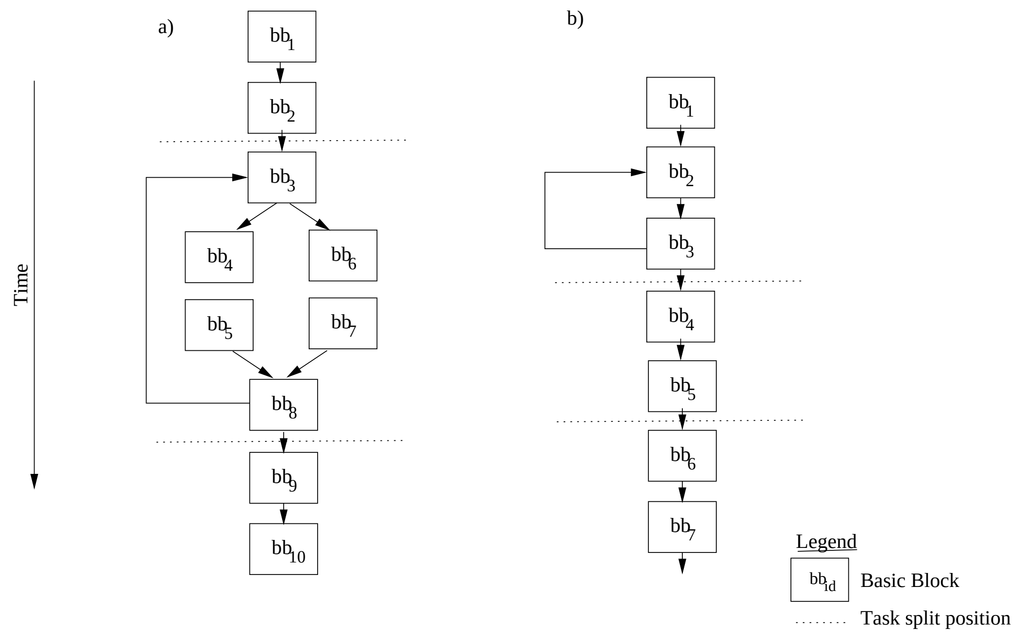


Figure 3.6: Examples of the compiler-driven positioning of task splits for two simple control graphs of basic blocks. The compiler avoids placing splits into loops.

to model the migration time of a task upon a split to the next execution context. This operation may also involve some latency to find an appropriate next execution context. For now, we implement communication between subtasks as a non-blocking data transfer on the same medium that transports the signal to launch the next split. The compiler computes the amount of data to be communicated as the union of all scalar values that are defined before the split and used after it. To simplify, we compute this set by looking at the basic blocks of the most frequent basic block trace. In our implementation, splits are bound to the same subset of PEs that the initial task can run on (i.e. the same mapping).

The split compiler operation can be performed iteratively. It is part of the post-simulation compiler pass (presented in Section 4.3.2) because it reuses several analyses from it. We have two algorithms to position the splits: when loops are found in unsplit regions, we split the function at loop boundaries; otherwise, we try and make three even splits out of an unsplit region. An illustration of the technique is shown in Figure 3.6; we can see that perfect balancing of work between the splits is not always possible.

3.4.2.1 Re-ordering Splits

With task splitting, it becomes possible that a lock migrates from one PE to another. Proper migration code has been inserted in the simulator to detect this condition. We must ensure that a lock holder can readily execute on the next PE otherwise all the contexts of the next PE could be occupied by tasks waiting for the lock, creating a deadlock situation.

Ennals et al. [23] propose a task optimization called “PipeIntro” that is similar to what we call “splitting”: they break a sequential task into task splits that will be executed in sequence for all packets. However, task splits from different packets can be interleaved in time. Ennals et al. also discuss correctness issues. Correctness is preserved in our approach as long as synchronized sections are preserved upon a split. Our work evaluates task splitting and presents system issues related to it.

3.4.2.2 Pipelining Splits

A more advanced form of splitting would facilitate *task pipelining*—allowing a task split to be executed in parallel with other splits from the same task. While task pipelining is potentially very useful, automatically pipelining the potential dependences within a task is complex and hence our infrastructure does not support it yet.

3.4.3 Signaling a Task Early

Signaling a task early tries to extract inter-task parallelism within the application task graph by launching multiple tasks processing the same packet as soon as it is determined that their execution would not violate inter-task dependences.

For each task, we determine the next tasks to be executed. If the task graph contains conditional branching, the next tasks considered must exist on all successor branches in the task graph. Using the task graph, we also check if starting the candidate tasks would earlier would incur dependences on memory locations so that we preserve the correct behavior of the program. We also take care to

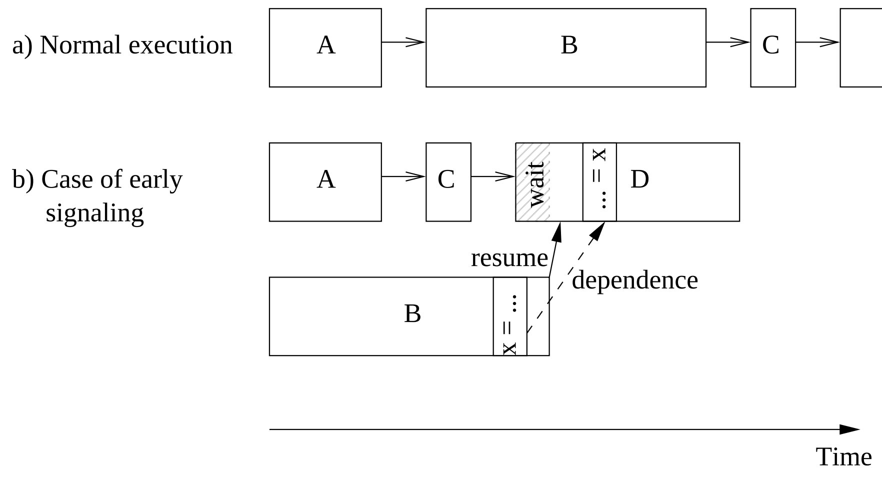


Figure 3.7: Example of early signaling requiring extra inter-task synchronization. Task `B` can start as early as task `A`, however task `D` must wait for a resume signal from task `B` because of an ordered dependence between task `B` and `D`.

remove signals that would be redundant when attempting to start a task early from different points in the task graph.

This transformation requires careful signaling: let us consider the processing chain `A`→`B`→`C` where `A` could signal task `C` to begin execution before task `B` completes (see early signaling in Figure 3.3). Care must be taken to determine that the early signaled task `C` has no dependence with `B`. `A` would then pass the packet to both tasks `B` and `C` upon completion.

Let us now consider the scenario where task `B` could be started as early as task `A`. Suppose also that there is a possible dependence between `B` and `D`, as shown in Figure 3.7. The problem that arises is that `A` is modified to start `C` upon completion, that in turn starts `D`. Hence, `D` could start executing before `B` has completed, thus violating a dependence. We solve the problem by creating two additional types of signals: i) `D` *waits_for* `B` and ii) `B` *resumes* `D` upon its completion. Tasks must also be informed of their time to live: in this example, the number of tasks `B` is allowed to signal upon completion is nil because `A` takes care of signaling `C`. In the case where `D` exists on another sequence of tasks where it is not preceded by `B`, we need another signal so that `D` does not wait in vain for `B`: a task prior to the execution of `D` *announces* that `D` will have to wait for `B`.

There exists cases where the announce/wait_for/resume scenario cannot apply because of the presence of the same tasks on multiple paths of the task graph. For example, a signal announcing a wait condition cannot be issued before the early signaled task if, in the unmodified ordering of the task graph, there is a branch right after the early signaled task. We do not implement early signaling in those those more complex cases because we believe that the hardware support would require non already available primitives.

Finally, sink elements (in particular, any task beyond the `Output` trigger in Figure 4.6) can never be signaled early; otherwise the packet could be prematurely set to be sent out of the NP. Also, we do not launch a task that is a leaf of the control graph because some memory deallocation may be attached to it.

Placement of early signaling in the code is done to make sure the candidate task is started as early as possible (i.e. not after an expansive memory access). Using our compiler pass, we insert the signaling code in the appropriate function, right after the start of a task. We have not implemented early signaling a task at a different point in a routine in part because of the complexities of placing the call while taking into account task dependences and control flow in the task itself. The ’resume’ signal is inserted right before the end of the task for reasons just mentioned.

We envision that, in future work, it would be possible to handle write-after-write dependences in early signaled tasks. We could insert some form of renaming allowing us to discard some write operations (as in the MLCA project [38]).

Impacts of early signaling on simulation Early signaling requires modifications in the simulator. Our implementation has to take into account that there is a possibility of deadlock if tasks occupy executions contexts while waiting for early signaled task to complete. Our solution is to evict the stalled contexts, as explained in section 3.4.1.2. As well, the simulator must provide support for the ’resume’ signal to reach the tasks in a ’wait_for’ state, considering that a task can be mapped to more than one PE. We currently implemented a broadcast algorithm to all the candidate PEs that could be running the task. Destruction of unused signals occurs in a similar broadcast fashion.

3.4.4 Task Speculation

In this work, we parallelize mostly non-loop portions of a program. For this reason, we have not looked at optimizations such as reduction although this might be beneficial to parallelize payload processing tasks that contain more tasks with loops. In general, we need to create parallelism where there is none. So we cannot follow the typical way of making threads by looking at the loop index. Sequential code usually has a lot of self-contained dependences and it is hard to guarantee that computations, when executed in parallel, will not access shared resources.

Figure 3.8 shows a piece of code that counts unique IP addresses (in a rather naive fashion). The concept is used in popular network address translation applications where an entry is created for each packet flow. If no entry is created, then the internal data structures of the element do not get altered and there is no dependence. Melvin and Pratt [53] make some measurements on the frequency of packets belonging to the same flow given a number of packets arriving in sequence. Speculation consists of removing synchronization and monitoring memory accesses for dependence violations, in which case the tasks that were last to enter the formerly synchronized section are aborted and restarted. When a task successfully completes, it must commit to the shared memory all its speculative memory writes buffered locally to the PE.

In the network processing tasks that we consider, we can have multiple synchronized sections for one task. Also, a normally synchronized task does not necessarily enter a synchronized section upon execution. Since we do not know if speculation will happen when the task starts, our implementation rolls back the task upon failed speculation to the beginning of speculation, as opposed to the beginning of the task. The order in which speculative tasks can commit is decided by the order in which they enter the synchronized section (or re-enter, upon violation). Consequently, the processor state has to be saved when entering a synchronized section and all writes to non temporary storage must be buffered until the task safely exits speculation.

In our current infrastructure, the compiler takes care of checkpointing the state of an hardware context at the entry of a synchronized section and restoring this state upon violation. The compiler also inserts code to buffer the writes to local storage and commit them to shared storage when the task completes without violation. Hardware support is however best to detect violations in a manner

```
class Counter {
    int known_cnt;
    int* known_addr[100];
}

void Counter::process(struct packet* p)
{
  int i, sum = 0;
  int addr = p->ip_idhr->ip_dst;

  bool found = false;
  for(i=0; i<known_cnt && known_addr[i] <= addr; i++)
      if(known_addr[i] == addr)
         { found = true; break; }

  if((!found) && (i < 99))
  {
      memmove( &known_addr[i+1],  &known_addr[i], known_cnt-i );
      known_addr[i] = addr;
      known_cnt++;
  }
}

```

Figure 3.8: Rare dependences make opportunities for speculation. In this case, a list is modified every time a new destination IP address is observed in the packets processed. The actual frequency of the modifications depends on the traffic patterns. For example, on a short time scale, all packets may belong to the same connection, thus not creating any dependence between task replicas. The synchronized section, covering most of the task, is highlighted.

that does not slow down the PEs. In our simulation, we assume that there is an engine snooping on the bus connecting the PEs to the persistent heap storage, which is the only memory type where dependences can occur between task replicas (as seen in Figure 3.3. This engine signals a violation to the context with an interrupt when an earlier task reads a memory location written to by a later task or when an earlier task writes to a location read or written to by a later task.

3.5 Managing Threads

Network processors, as surveyed in Section 2.2.2.1, often have processing elements with multiple execution contexts that execute software threads. There exist different flavors of how threads interact on a single processing element. Ungerer et al. [85] explain and survey those different mechanisms. To give an overview of the possible mechanisms and motivate a choice for NP simulation (Chapter 5), we briefly summarize their study.

There are two categories of multithreading when instructions are issued from a single thread every cycle, as surveyed by Ungerer et al. [85]. Fine-Grain Multithreading (FGMT) takes an instruction from each thread in sequence every cycle. One of the advantages of FGMT is to minimize hazards in the pipeline, thus simplifying and speeding up the processor. The alternative is blocked multithreading (BMT): one thread executes until an event occurs and causes a context switching. In some processors, because of replicated fetch units, the context switching time is null. Processors that can issue instructions from multiple threads at the same time perform what is called Simultaneous multithreading (SMT).

Context switching means transferring the control between the threads, and hence, only applies when one thread executes at a time. Context switches can be static, i.e. triggered explicitly or implicitly by a given set of instructions, or dynamic. Dynamic context switching can occur upon a signal, for example, an interrupt, a trap or a message. It can also be triggered in some architectures by the use of a value that has not been loaded yet because of a pending memory access. Finally, dynamic context switches can occur based on a conditional instruction that, for example, triggers a context switch when a register reaches a threshold value.

Modern NPs typically implement blocked multithreading and provide very low-latency switching between hardware contexts within a PE. For example, in Intel's IXP family, a context switch is triggered by the programmer through an explicit context switch instruction (`ctx_arb`). Alternatively, a PE can switch contexts based on its current state, reacting to dynamic events. BMT does not require multiple issue hardware support and allows to execute a task at the full performance of the PE. In our implementation, we model dynamic context switching whenever a long-latency stall occurs. In combination with decoupled, non-blocking loads, this allows the NP programmer to tolerate the significant latency of memory accesses.

We have identified two opportunities to improve on the thread management. When task replication is implemented, the thread that has acquired a lock is on the critical path of other threads waiting for the lock. To minimize this contention, we have added support for the critical thread to preempt other threads. As well, we have tried to load balance the requests of the threads on the shared NP busses so that a particular thread does not congest the bus and leaves the other threads waiting. In summary, our thread management strategy is to execute the first thread that is ready to execute and context switch when that thread can no longer execute. We have implemented some refinements on this that we evaluate in our simulator.

So far in this chapter, we have presented task transformations that we implement based on underlying concepts, namely: packet ordering, memory typing and task dependences localized to certain memory types. Our task transformations involve memory management and also affect the interactions between tasks. Our infrastructure also has some notion of managing thread interactions. Next, we introduce the components of our infrastructure.

3.6 The NPIRE Framework: Overview

Our compilation infrastructure transforms a graph of tasks supplied by the application programmer. To evaluate the resulting network processor system, we have to measure and compare the maximum throughput achievable in a given configuration. We are also interested in understanding the performance-limiting factors. This section presents the integrated suite of tools that we developed for this purpose, starting with a justification for the main component of this large software task: the simulator.

3.6.1 Motivation for a New Simulator

In parallel computing systems, Amdahl's law states that the total execution time improves linearly as we add more processors to take on the parallelizable portion of a program. In its formulation, $Time_{total} = Time_{sequential} + \frac{Time_{parallelizable}}{num\ processors}$, this law also illustrates the concept of diminishing returns: as we invest in more processors, the sequential fraction of an application is not reduced and can dominate the overall latency.

Amdahl's law does not take into account several forms of contention that we can observe in NP systems. For example, two tasks running on two hardware contexts in one PE seldom share the PE so nicely that we can measure a speedup of 2. In reality, one task will typically be able to resume execution before the other task relinquishes the processor. A similar contention is likely to happen whenever shared resources are assigned to tasks. Assuming an infinite availability of hardware, producer-consumer data dependences between tasks are the only limit to speedup. In real life however, a system may experience long periods of time when congestion delays the consumer such that it is not ready to collect the producer's work. Because contention on shared resources depends on the actual arrival and departure times in waiting queues, we have to perform a *simulation* of the interactions between all the main NP components; a theoretical approach is not satisfactory.

Our goal is to provide an integrated compilation framework for network processors. To provide simulation feedback to the compiler, we want our simulator to understand any application we feed to it in a format that is machine-generated. In order to study the architecture of a network processor, we need a simulator flexible enough to do some design space exploration. We need to vary the amount and the kind of on-chip resources and customize the processor behavior to the different transformations we attempt on the application. Also, our simulator needs to execute rapidly so that we have the opportunity to sweep certain design parameter ranges. As presented in section 2.3, no such software could satisfy our requirements, because the alternatives offer a detailed simulation too specific to a given architecture and/or because they could not be adapted to accommodate new

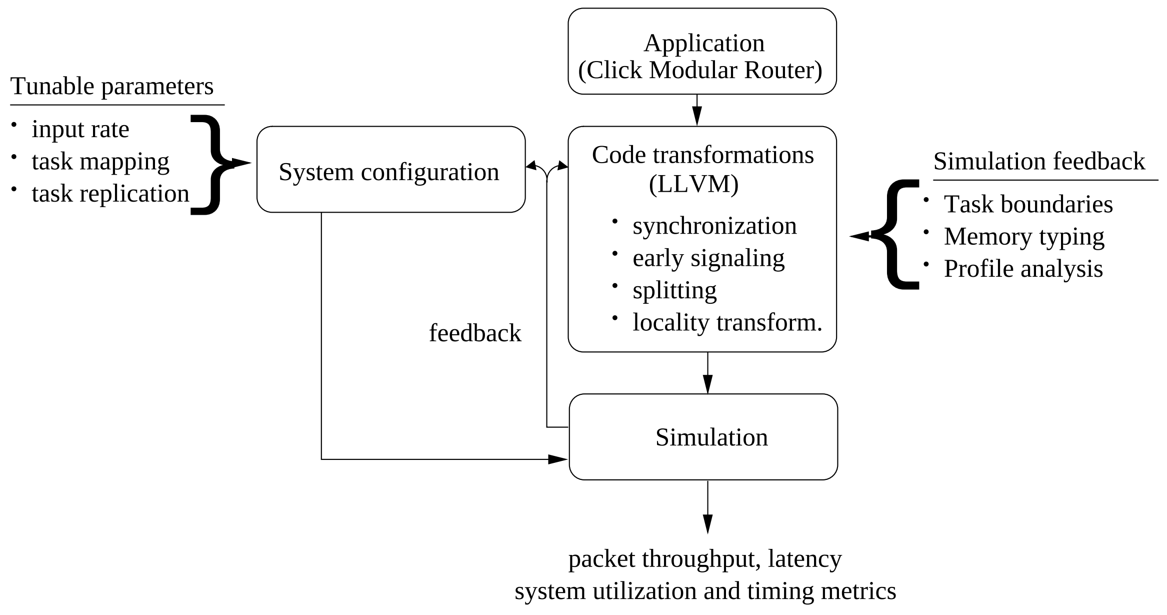


Figure 3.9: Our infrastructure for integrated NP simulation.

behaviors. This justifies the design and implementation of NPIRE, our infrastructure with its own simulator. After a few generations of relying on open-source components (see section A.3), our now full-custom simulator uses real network router applications transformed by some compiler passes of our own.

3.6.2 NPIRE Components

The basic structure of NPIRE, our integrated compilation-simulation environment, is shown in Figure 3.9. NPIRE uses the Click Modular Router [41] in two ways: first, as a programming model and second, as a base for more realistic, complete applications. In this study we evaluate NP architectures using actual Click applications. The NPIRE compiler infrastructure, built on LLVM [46], divides Click's modular *elements* into tasks, inserts synchronization and signaling by analyzing dependences between tasks, and maps tasks to processing resources. It also maps memory to the different potential storage mechanisms. In the NPIRE simulation model, those mechanisms can be: local to the processor, shared at the processor level (e.g., for next-neighbor communication), shared chip-wide (e.g., a scratchpad), or shared in external (off-chip) memory. In addition, the NPIRE simulator allows us to vary the number of *processing elements* (PEs), the number of hardware contexts per PE, and the interconnection between the PEs themselves and with the various types of memory. Finally, the simulator provides feedback that allows the compiler to iteratively mold the application to the supporting hardware by improving the task partitioning and mapping, emulating the efforts of programming by hand.

3.7 Summary

In this chapter, we have presented a programming model based on task graphs and memory type identification. This programming model confines inter-task dependences in packet meta-data, packet buffers and persistent heap. Other memory types include temporary heap, stack and instruction storage. We use this dependence characterization in tasks to parallelize and reorder task execution when dependences are not violated. The NPIRE compiler infrastructure can perform a locality transformation on memory accesses consisting of memory batching and inter-task forwarding. Four task transformations take advantage of the multiple processing elements available on modern network processors. We introduced the compiler techniques to implement, namely: replication, splitting, early signaling and speculation. Finally, two refinements, preemption and priorities on shared busses, improve on our dynamic context switches-based thread management strategy that minimizes long latencies.

To implement and evaluate the proposed optimizations, the NPIRE framework is made out of three components that we will present successively in the next chapters: real network processing applications, a compiler infrastructure and a simulator. We will describe the first two software tools, followed by our simulator, designed to mimic network processor hardware.

4 The NPIRE Framework: Software Components

In this chapter, we describe our software tools that act as an enabling platform to the task transformations presented in Chapter 3.

4.1 Overview

Figure 4.1 shows the software flow that transforms the application to efficiently exploit the simulated NP hardware. Our infrastructure has a compilation unit that analyzes the Click elements and drives the task transformations. The compiler passes generate code allowing us to build our execution environment for our NP application directly into the original Click code itself (as explained in Section 4.3.1.1). As shown in Figure 4.1, the result of this execution is a trace suitable for simulation on our modeled network processor. We will refer to Figure 4.1 throughout this chapter while describing the NP applications that we transform and the compilation process that leads to simulation.

To introduce task graph manipulations, we first give some details on the construction of an application in our infrastructure. We then present how our compiler passes transform the application and generate code suitable for the application's simulation. We conclude this chapter by explaining further application transformations that leverage simulation feedback.

4.2 Expressing Packet Processing with Click

We use the Click Modular Router [41] to build applications by connecting Click's independent modules, called *elements*, as described in section 2.5. In this section, we explain the process of

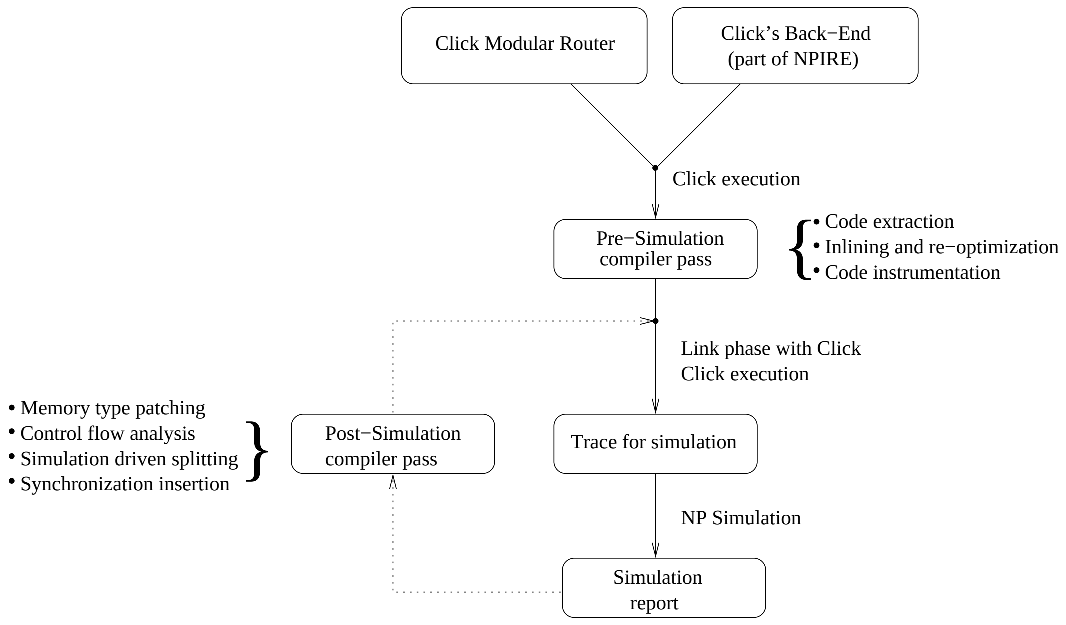


Figure 4.1: Work-flow of the simulation environment.

creating a new application. We also describe the execution behaviors of the application that are relevant to simulation and the modifications made to Click to allow tracing of those behaviors.

4.2.1 Creating a new benchmark application

To evaluate the performance of a network processor application inside NPIRE, the user must first describe the application in Click's configuration language, as exemplified in Figure 2.4(a). The result of the application design is a graph of *tasks*, as illustrated in Figure 4.2, where each task represents the processing performed by an element on a packet. Each element has customizable parameters that modify its generic behavior. For example, the `LinearIPLookup` element requires a routing table that must be specified in the Click configuration file. Figure 4.2 also shows that some elements can be present in multiple *instances* that have no sharing in their runtime persistent heap, a data storage type that we defined in section 3.2. For example, we can see in the graph that there are two `Classifiers`.

While we focus the compilation effort on the application's tasks, there are parts of the Click router that we do not analyze. In particular, to be able to execute an application, the Click router needs to parse the user's application description and schedule elements upon packet arrival. Those Click parsing and scheduling components are outside the scope of packet processing on a network processor so they are not processed by our compiler infrastructure.

4.2.2 NPIRE's Support for Creating Suitable Packet Traces

In all our applications, we use the elements `FromDump` and `ToDump` to respectively read and write packet traces for the incoming packets and outgoing packets of the Click router. The user must also supply one packet trace to be read by Click for each input stream of the application. Those packet traces should have the appropriate characteristics (for example, destination identifiers and protocol layers) to exercise the application's tasks in the same frequency as in their expected real world deployment. Failure to match the packet trace to the elements behavior can result in, for example, having all packets discarded because no route is known for them. To help the application writer, our simulator reports the flow of packets along each edge of the task graph as seen in Figure 4.2.

The coverage of our profiling of the program will only extend to the code that is executed inside the elements. In turn, this profiling will affect the inter-task and intra-task dependence analysis in the task transformations we perform. Depending on the application, different packets can stress different control paths: so the simulation must be long enough to cover a representative mix of packets. In our measurements, we try to have a roughly equivalent distribution of packets across all utilized branches in the application's task graph. Some packets may trigger exception handling conditions, which represent worst case latency scenarios: for example, the costly sending of an Internet Control Message Protocol (ICMP) error packet. Similar to a high speed highway, single exceptions in an NP can have a significant impact on throughput. For this reason, in our traces, we keep those normally infrequent behaviors to a negligible fraction of the packets.

We use pre-recorded packet traces to exercise the Click applications. The traces used can be any of the following:

1. a true LAN trace;

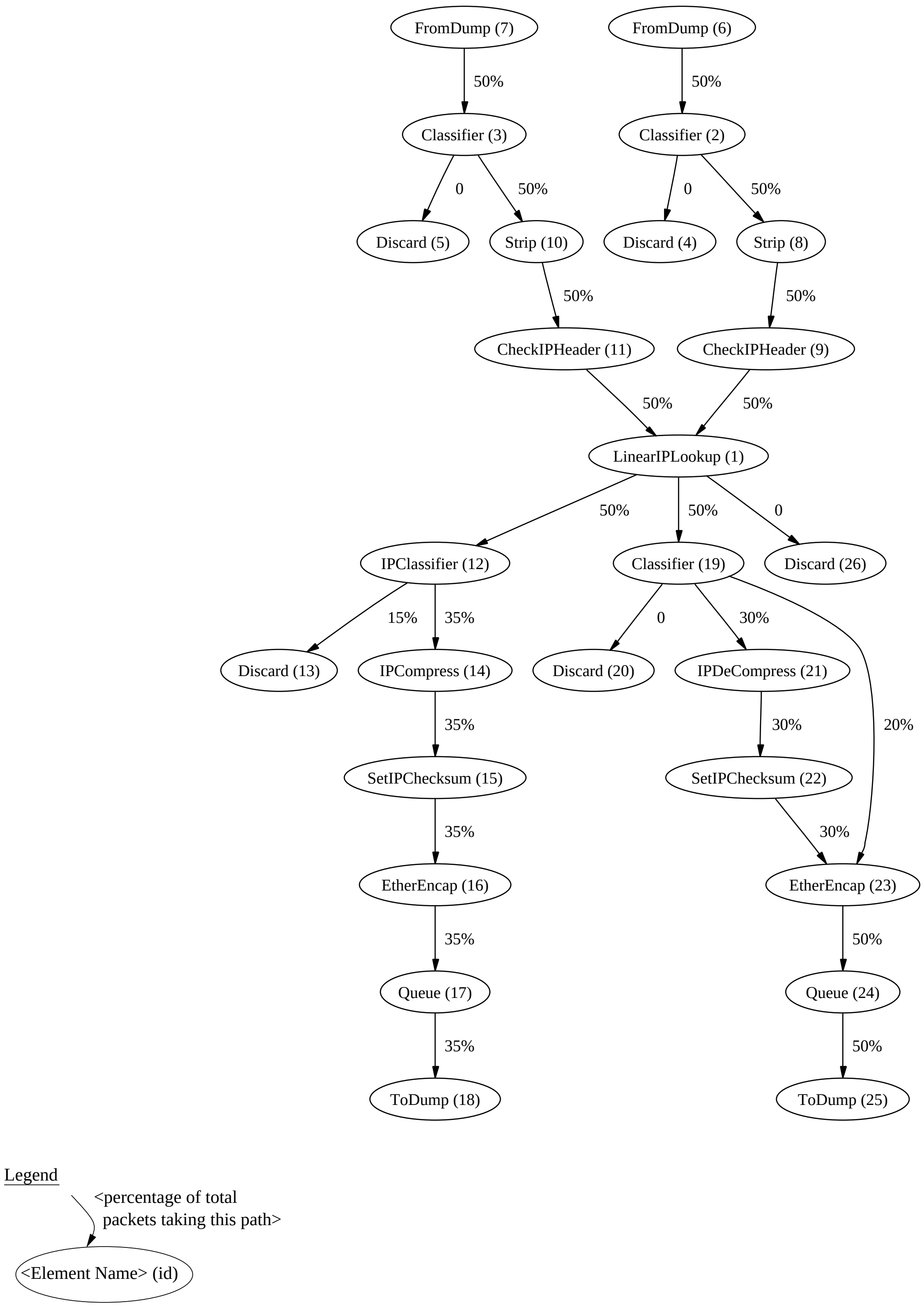


Figure 4.2: Task graph of a compression application generated after a short simulation. The graph shows the number of packets that flow between each pair of consecutive elements.

2. a true LAN trace altered to approximately match the packet mix (size and type) of a reference trace;
3. a reference trace with payloads borrowed from a true LAN trace injected.

We only employ the third option in the list because it makes use of publicly available reference traces from the National Laboratory for Applied Network Research (NLANR) [61]. Those packet traces originate from high speed links (10Gbps) and have been widely used in various other research, including some workload characterization in network processors [66]. Ethereal [68] allows us to convert NLANR traces from Endace's Extensible Record Format (ERF) format to tcpdump [47] traces, understood by Click. However, the reference traces are distributed with their payload removed, for privacy reasons. To alleviate this lack of payload, we modify the packet traces as follows:

1. Packet buffers are padded to make the size described in their header match their actual size. For the payloads, we use a packet trace recorded on a LAN. We inject into each packet of the reference trace the content of a LAN packet that resembles the reference packet in protocol and overall size. If we cannot find a data buffer that is long enough to fully pad the reference packet, we simply repeat the data buffer selected for padding.
2. A checksum is inserted to all known protocols headers that require one in the reference packets. This facilitates the handling of the trace with some packet readers and, most importantly, allows the packets to pass common checks on its IP header.
3. The range of IP addresses of the packets is optionally remapped. The source and destination IP addresses of the reference packets can also be inverted. This is required to simulate different networks and to facilitate the creation of routing tables in our applications.
4. Save the resulting modified trace using tcpdump's API.

NPIRE provides a tool based on libnetdude [42] to do these manipulations on the packet traces. Although there are timestamps in the reference traces, we do not use them: instead we intend to stress our simulated network processor with a rate that matches its processing capability.

4.2.3 Back-End Added to the Click Modular Router

Figure 4.1 illustrates that some code is added to the Click router before we execute our network processing applications. We refer to this software layer added to Click as *Click's back-end*. This section explains what was inserted in Click to produce the application trace file along with some statistics about the basic block execution.

4.2.3.1 Support for Tracking Packets

To identify all accesses to packet data, our back-end must recognize packet data structures inside the Click router. Inside the router, a packet exists as an instance of a class `Packet`; we refer to it as *packet meta data*. This class contains pointers to the actual packet data. The *packet data*, both its header and payload, is referenced from the memory buffer into which the Click router reads large segments from the packet trace file. Some elements can also allocate memory and copy into it the packet data to, for example, extend a packet. On the other hand, the packet meta data is created using the `new` operator in `C++`. When a packet needs to be sent on multiple paths in the task graph, a copy of the meta data is created but the same reference to the packet data is copied over. The packet data only gets replicated across those new packet meta data if the packet data is written to, as any writes to the packet data must be guarded by a call to a buffer unification routine. Because of this efficient memory management inside the Click router, we had to overload the `new` and `delete` operators of the packet meta data and insert code in the elements that source and sink packets to handle respectively the packet data allocation and release.

4.2.3.2 Application Instrumentation Handlers

All the instrumentation calls inserted by the compiler are handled in Click's back-end, making our infrastructure modular inside the router. The back-end handlers are used to monitor instruction count, memory accesses, memory allocation/free, start/end of tasks and signal insertion for task splitting and early signaling. Also, there is a handler to gather the connectivity graph of the elements inside Click and another handler to store a pointer to the stack after the initialization. This pointer will be used to determine if a memory reference belongs to the heap or the stack. In all those instrumentation handlers, the back-end contains an interface to create records in the application trace file.

The back-end also holds a table containing a pointer to all of the application's elements and to their successors in the task graph. As well, the back-end maintains a data structure at runtime to keep track of the identity of the element for which a task is being executed. With this element identifier knowledge, the back-end is able to insert some element specific events in the application trace, such as signals for the early signaling transformation. This conditional insertion allows us to have multiple instances of an element sharing the same code inside the Click router, thus reducing our compiler's work. For each element in execution, we also use the back-end data structures to get a reference on the successor element to which the packet being processed will be forwarded. We use that reference to dispatch the call to the successor element and to insert element transitions in the application trace.

All memory references that reach the back-end are classified by type. The identification is made by traversing a list of allocated buffers. Finally, some validation is made on the addresses of allocated buffers to ensure that no two buffers are allocated at the same address before one of them has been freed.

4.2.3.3 Support for Asynchronous Memory Loads

For NPIRE to model asynchronous (also known as non-blocking) memory operations, we must determine the distance between any load instruction and the corresponding first use of the value loaded. We first present two unsatisfactory approaches at measuring this distance that motivate the third approach that we implemented.

Computing the def-to-use distance of non-blocking loads in a static analysis in the compiler is not possible because of branches in the control flow. However, knowing this distance in the compiler would allow us to break asynchronous loads into their two components: i) the slack time which is the time between the load and its first use and ii) the stall time which is the time after the slack spent waiting for the memory hierarchy to return the value. A static determination of the def-to-use distances would allow the compiler to plan the instruction schedule more carefully and to insert static context switches to minimize the stall times. An alternative way of getting those *def-to-use* relations between instructions would be to do some micro-architectural simulation, which could also be used to introduce different flavors of instruction-level parallelism, such as super-scalar and VLIW. This technique would require executable parallel code generation and simulation support that are beyond the scope of this work. Consequently, we decided to opt for an implementation inside Click's back-end.

Our back-end support for asynchronous memory loads has the advantage that the def-to-use distances can be measured and written to the application trace in a single sequential execution of the application. The back-end has a software component that can buffer non-blocking operations until the slack time is decided. This buffering code is part of Click's back-end module designed to efficiently write the trace. This module handles an instrumentation call on each use of a value loaded, traverses an indexed list of recent memory accesses to find the original load and sets the use/def relations. Only the first use of a value defines the use/def relation. Each load has a default slack time set to infinity in case no use is found. Efficient data structures are critical to generate the trace in a reasonable amount of time. Because non-blocking memory accesses support is a crucial part of the back-end code, validation code is in place to detect any potential error.

4.2.3.4 Support for Identifying Memory Types

Memory types, as defined in section 3.2, are used to categorize memory accesses that have different sharing and dependence characteristics. The back-end is extensible because it can easily accommodate the addition of new memory types. This makes it easier to specialize the processing of a certain kind of data.

Click's tasks are declared using a standard interface and comply to certain requirements that make them easier to run concurrently. The only data passed in an element is a pointer to a packet descriptor that, in turn, contains a pointer to the packet buffer. Packet descriptors flow in and out of elements: if a packet is dropped, then the packet descriptor is destroyed. All elements have an associated configuration and potentially some dynamic data structures. Figure 3.2 presents the memory types found inside Click: the separation of the memory types is important to model the appropriate storage

```
struct packet_desc {
  int ip;
  char* packet_data;          packet data
};

struct ip {
  int addr;
  int access_cnt;
  struct ip* next;
};

class ip_counter {
  int num_uniq_ips;
  struct ip* list_ip_head;          persistent static heap
public:
  ip_counter();
  ~ip_counter();
  struct packet_desc* push1(struct packet_desc* desc);
  struct packet_desc* push2(struct packet_desc* desc);
};

struct packet_desc* ip_counter::push1(struct packet_desc* desc)
{
  bool found = false;          stack
  struct ip* ptr = list_ip_head;
  while(ptr != NULL)
    {
      if(ptr->addr ==
                       desc->ip)        packet meta data
        {
          found = true;
          break;
        }
      ptr = ptr->next;
    }
  if(!found)
    {
      ptr = new struct ip;          temporary dynamic heap
      ptr->addr = desc->ip;
      ptr->next = list_ip_head;
      list_ip_head = ptr;          transition from temporary to persistent dynamic heap
      num_uniq_ips++;
    }
  ptr->access_cnt++;
  return desc;
}
```

Figure 4.3: Illustration of memory types in a task (the critical section is highlighted).

for each buffer and to characterize the task dependences, as summarized in Table 3.3.

Figures 4.3 and 4.4 give an example of two implementations of a fictitious element that counts the number of accesses to distinct IP addresses. The first implementation is a FIFO linked list while the second figure shows the use of a sorted linked list of IP addresses. In both cases, if the modifications to the variable *access_cnt* of the IP record are not atomic, then there is dependence

```

struct packet_desc* ip_counter::push2(struct packet_desc* desc)
{
  bool found = false;
  struct ip* prev, *ptr = list_ip_head;
  prev = ptr;
  while(ptr != NULL)
    {
      if(ptr->addr == desc->ip)
        {
          found = true;
          break;
        }
      else if(ptr->addr > desc->ip)
        break;
      prev = ptr;
      ptr = ptr->next;
    }
  if(!found)
    {
      ptr = new struct ip;
      ptr->addr = desc->ip;
      if(prev)
        {
          ptr->next = prev->next;
          prev->next = ptr;         dynamic persistent heap modified
        }
      else
        list_ip_head = ptr;         static persistent heap modified
      num_uniq_ips++;
    }
  ptr->access_cnt++;
  return desc;
}
```

Figure 4.4: Illustration of memory types involved in the synchronization of a task (the critical section is highlighted).

between replicas of the task. There is also a larger dependence region because of the head of the linked list *list_ip_head* that can be modified by the code. Figure 4.3 shows the memory types involved while Figure 4.4 shows the importance of tracking dynamic heap storage. In Figure 4.4, we can see that the dynamic heap modification is enclosed by a static heap synchronized section solely because the head of the list is written to.

The compiler has some code to do a basic identification of memory types, however this has limitations. For example, it is possible for an instruction accessing memory to refer to different memory types in two different accesses. So we prefer using memory types fed back from Click's back-end and inserted directly in the trace. We next explain how the back-end achieves this function.

The back-end tracks the allocation of memory buffers: these buffers can either belong to packet descriptor storage, packet data or local element storage. Temporary stack and heap storage does not have associated memory allocation events in the application trace because of the difficulty to track each of those individual stack and heap allocations. Our work instead assumes very efficient primitives for memory allocation and release in a thread-safe manner, an example of such is given by Parson [63]. Temporary heap is distinguished from permanent heap storage by determining if memory accesses are within buffers that can be reached by pointers in elements' data structures, using the elements' static data structures as starting points for the search. The process has some similarities with the "mark" heap search performed in some garbage collection systems [94]. We envision that this reachability analysis could also be used to determine if locally allocated heap buffers escape the scope of execution of an element, in order to potentially convert these heap accesses to stack accesses as proposed by Gay and Steensgaard [24].

In simulation, when processing packets in parallel, we may reorder all memory accesses and we need to preserve the fact that accesses to different allocated buffers are independent. Memory allocation in the operating system Click runs on is likely to hand back to the application a buffer that has recently been de-allocated. So we need to remap the locations accessed in memory so that they have no overlap with other unrelated buffers. All non-persistent memory, including the packet meta-data, packet buffers and the allocated stack, must be remapped to unique memory locations for each task. However, persistent heap space is never remapped in order to model system-wide dynamic memory allocations, as opposed to other parallel task infrastructures that only target static structures, such as the SAFL language [60]. Keeping track of allocations and releases in the back-end is required to alter the addresses of memory accesses.

4.2.3.5 Unsupported Behaviors in Packet Handling Routines

Some minor modifications are required for packet handling routines inside a limited number of Click elements, prior to their compilation in our infrastructure. For example, because we work with packet traces, we cannot allow for the router to generate a packet to request the hardware address of another host and wait for the reply, according to the Address Resolution Protocol (ARP) in IP networks. When this situation happens, we assume the router already knows the information that it was about to request. In reality, this is a reasonable assumption because ARP requests occur in the scale of minutes, hours, or not at all, in a high speed network. In summary, in our current implementation, any processing based on a future packet reply or a timer escapes our tracing of the application and must be avoided. This restriction is due to the fact that, at simulation time, we rearrange all the application trace events and we have no good way to position a task that is not bound to an incoming packet event in the modified execution time.

4.2.4 Migration of an Application to the Compiler Infrastructure

After the Click router parses the application file supplied by the user, the elements are connected by Click to form the task graph. To import this task graph into the compiler, we need to give to the compiler the following two files:

- A connectivity graph of the elements in the application. This connectivity graph is taken from the execution of the Click router: the file is generated after the Click router parses its configuration file.
- A file describing the binding of elements to `C++` classes. This information is obtained by examining the Click router source files.

A script reads the connectivity graph and generates the second configuration file in the above list, given the root directory of the source code for all the elements. Because the compiler needs specific information on the location of an element's code, we need to identify whether there is inheritance between elements (for example, `StaticIPLookup` is derived from `LinearIPLookup`) and what functions are present in the (derived) classes. Care must be taken to find the right function when an element is derived from another element and does not provide any of the basic element routines.

The entry points for the execution of an element that must be identified by the compiler are one of the following routines: `run_task()`, `push()`, `pull` or `simple_action`. All those functions are overloaded functions of the primitive implementations in the base class `Element`. `run_task()` is a function used primarily by elements that source and sink packets. Those elements are typically scheduled by Click instead of being called by other elements. This distinction is important: the

```
void Element::push(int port, Packet *p)
{
  p = simple_action(p);
  if (p) output(0).push(p);
}

Packet* Element::pull(int port)
{
  Packet *p = input(0).pull();
  if (p) p = simple_action(p);
  return p;
}
```

Figure 4.5: Default implementation of the `push()` and `pull()` element entry points.

compiler can connect elements that call each other but lets the simulator handle the elements that Click schedules upon arrival of a packet in the router. `simple_action()` is a one-parameter version of the processing and is called in the default implementation of `push()` and `pull()` as shown in Figure 4.5.

4.3 NPIRE's Compiler Support

The compiler support in NPIRE enables task transformations as well as the insertion of all the instrumentation calls in the elements' code that help Click's back-end generate the application trace and some execution statistics.

The compiler infrastructure, based on LLVM [46], is currently composed of two passes as illustrated in Figure 4.1. The *Pre-Simulation* pass modifies each of the application's elements from the source code and emits a compiled binary version. Click is compiled with those transformed elements and executed to generate the application trace. This trace is played through the simulator. The pre-simulation compiler pass and the simulator produce input files for the *Post-simulation* pass of the compiler. This second compiler pass transforms the elements' code for a more efficient execution of Click on the simulated architecture while leveraging simulation feedback. While support for task transformations was presented in Chapter 3, we now describe in more detail the other components of those two compiler passes.

4.3.1 Pre-Simulation Pass

The goal of the pre-simulation pass is to analyze the memory accesses because we need to know where to physically send data in a realistic/simulated NP implementation. Another objective of this pass is to instrument the elements' code to get the execution frequency of basic blocks when the application is exercised by a representative packet trace. We next describe these steps in the compiler pass, starting with the transformations made by the compiler to the original Click code.

4.3.1.1 Code Transformations

An initial run of Click with its back-end creates a connectivity graph of the elements contained in the application's task graph. This information is useful to identify which functions are on the packet forwarding path of our application. If two elements with different names have the same function because they both are derived from a common element, we duplicate the code of one of the elements to avoid any ambiguity in the instrumentation we will insert. The initial compilation phase includes compiling all of Click's files through LLVM . We then extract from the voluminous binary only the functions that belong to the packet forwarding path of the application. In fact, only one function is extracted for each element except for the `Queue`, that preserves both `push()` and `pull()`. When extracting the functions, we put them in a different module, under a different name to avoid naming collisions with the original Click.

Our compiler passes assume that only inlined code should be instrumented. Inlining functions called by elements as much as possible creates long sections of straight-lined code which are easier to optimize. This simplifies the instrumentation by eliminating shared code across elements and hence shared tags in the instrumentation. The alternative would be to have elaborate tracking abilities in Click's back-end to distinguish a code execution in different calling contexts. In an actual NP implementation, inlining would have to be balanced by the corresponding tradeoffs in performance and instruction storage space. Although inlining reaches deeply inside Click, the compiler passes only touch code that is on the packet forwarding path: the initialization/configuration phase of the router can safely be ignored at simulation time because we are only concerned with the runtime NP behavior. We assume that the data loaded at Click's initialization time would be loaded inside the memory of the modeled NP.

We use the Click router to read and write packet traces but this occurs outside the scope of our instrumentation. Because we do not model any kind of media access controller to act as source and sink of packets, we consider that the packets are allocated in memory by some kind of direct memory access. Also, we do not account for fixed sized packet buffers (called *mbuf* in the Intel IXP NP chips, and *skbuf* in the Linux operating system). We assume that the packets are transparently accessible starting at a certain memory location. Because we rely on Click to load the packets, we use the packet alignment in memory that is given to us. Since word sized unaligned memory accesses are allowed on an x86 architecture, we assume that our simulated implementation has packets initially aligned to avoid the overhead of unaligned memory accesses on a simple NP architecture. This packet alignment assumption is realistic because modern NPs, such as the IXP2800 [35], have specialized hardware to automate byte alignment.

Click is coded in `C++`, a high level language that, because of its overheads, is not a traditional choice for NPs. However, the modularity and encapsulation of this object oriented language make it a good candidate for parallelization and source-level optimization. We limit the overhead of `C++` by inlining method calls whenever possible. We have not encountered cases where Click makes calls to library functions that could not be inlined. Exceptions are the primitive `memset()` and `memcpy()` that are assumed to be handled by some hardware primitive. Recursive functions as well as function pointers that could not be inlined for instrumentation are negligible in dynamic instruction count. As a guideline to NP programmers, we would recommend transforming those routines so that they could be inlined. We also tried inlining together tasks that always occur in sequence. On average over our experiments, we found that this operation does not save any memory accesses, saves at best 5 instructions and incurs extra communication between the elements involved. Those poor instruction count improvements show that little code is reusable to, for example, access boxed data structures [12], a characteristic overhead of object oriented programs. Inter-element inlining was

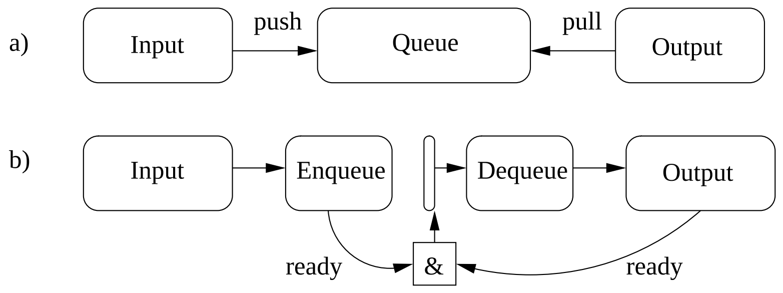


Figure 4.6: Modifying Click's push and pull queue operation a) the original application using push and pull; b) the transformed application with push and rate-controlled pull.

also abandoned because it presents several challenges to subsequent compiler passes if we want to preserve the identity of the separate elements. Finally, we argue that working from a high-level language description of the elements is acceptable, considering that other real network processor systems have also some source of overhead while binding tasks at a high level decomposition: for example, the Netbind project [9] has an overhead that grows linearly with the number of software tasks.

In our infrastructure, we adopt Click's programming model [41]: elements are connected by `push` (send a packet to the next element) and `pull` (request a packet from the previous element) links, as shown in Figure 4.6(a). The elements that initiate a push or a pull without corresponding requests are scheduled by the Click router. Any scheduled task can trigger an arbitrary sequence of push and pull requests. Section 2.5 gives more details on the Click Modular Router. In this compiler pass, we transform transitions between elements in explicit calls to our instrumented methods. This has the benefit of eliminating virtual calls through various layers of the Click software. Also, this call transformation along with the above mentioned inlining allows us to build a set of functions that constitute the totality of the processing for a packet.

In Click, a `Queue` element has both a `push` component on the enqueue and a `pull` component on the dequeue operation. In Figure 4.6(a) the element `Output` that triggers the dequeue (that `pulls`) can by default pull even if no packet is ready to be pulled. In NPIRE, we transform `pull` operations by adding a token indicating the readiness of the `Output` element so that the `Dequeue` operation does not occur before the `Output` element is ready. This additional control allows us to model sorting packets on the output interface. In NPIRE, we use Click's `Queue` element to specify that packet ordering must be preserved at the point where it is employed. As in most real network processor applications, our applications therefore precede all of their output ports by a `Queue` element to preserve order.

In a separate part of this pre-simulation compiler pass, we reduce all `C++` `try`/`catch` sequences to the content of the `try` block. We can remove this overhead because, in Click's code, exceptions are only for error/debugging purposes. We also automatically remove all trace of debugging information printed by the elements. As well, Click is compiled with the `assert()` calls disabled (i.e. they do not generate any code). After those code alterations, we ask LLVM to re-optimize the instrumented set of functions extracted from the Click source code. LLVM was modified to prevent the unwanted inlining of element functions, because we want control on where the code for each element is located.

4.3.1.2 Code Instrumentation

We use LLVM [46] to insert instrumentation on a basic block basis. In this step of the pre-simulation compiler pass, we insert instrumentation to track instruction counts and memory load and store instructions. Extra information is also passed to Click's back-end to record the sequence of basic blocks executed.

Since our simulation infrastructure does not target yet any form of micro-architectural evaluation, we are not interested in the actual instructions behavior, except for instructions that lead to some form of communication. Our intent is to model a RISC processor completing one instruction per cycle. Similar simplifications are done in such works as the Modeling Framework [14] and the Intel Architecture tool [26]. The main caveat of this approach is unmodeled pipeline stalls. This is partially alleviated if we assume that our compiler can hint accurately branch prediction. Also, multithreaded processors usually have instructions from multiple threads in the pipeline to hide most stall time. Instruction counts are obtained using a rough conversion factor from the intermediate representation to RISC instructions: 3.5, according to the LLVM intermediate representation (IR) design paper [1]. Interestingly, this number is roughly equivalent to the number of internal decoded micro operations obtained from an x86 instruction stream: each IR instruction generates 2 or 3 x86 instructions [1] and the rePLay paper [78] indicates a conversion of 1.4 from x86 to micro-ops, thus 2.8 to 4.2 (average 3.5) micro-ops for an IR instruction.

Start and end of task markers are inserted to generate the trace information. Those end of task markers allow the execution to resume/continue on an element that would launch another element before its completion (a technique also known as non-blocking dispatch). Note however that Click disallows processing a packet once it is sent to another element. At this point in the code preparation, all distinct instances of an element present in an application share the same code in our modified binary.

4.3.1.3 Code Generation

Once the instrumentation calls have been inserted at the compiler intermediate representation level, we have a file that contains all the static processing of the elements. Our compiler can optionally generate different copies of a function shared among different instances of an element, in the case where the code would be dispatched to different control stores in an actual NP implementation. The remaining challenge is to incorporate all of the new element code inside Click.

We export the intermediate representation from LLVM to an x86 assembly file using the LLVM infrastructure. We also create an include file containing the information to size static arrays and per-application file name information for the back-end. Files generated by Click will we be recuperated later by our LLVM passes. After compiling Click using `g++`, we need to link Click with the LLVM generated x86 assembly. An issue that arises is global symbol duplication: some global data structures exist in g++ and LLVM object files. To solve the problem, we first cut all globals from the generated assembly. We then select the missing globals and append them to the cut assembly file. Finally, we disassemble the `g++` object files from the schedulable elements (i.e. with a `run_task()` function, as defined in Section 4.2.4) in the original Click. At the entry of these functions, we insert a bypass to their transformed copies renamed to avoid name collisions. By combining our code with allows us to reuse the Click skeleton, in particular the initialization and configuration code and

Click's scheduler, to trigger our modified elements. All of the above steps are performed automatically by custom made scripts. Next, the LLVM generated assembly file is linked with Click that is now ready for execution and trace generation.

4.3.2 Post-Simulation Pass

The second compiler pass occurs after the application has been prepared by the pre-simulation compiler pass and the simulator has made initial measurements on this application. We need some simulation feedback on the timings for a sequential execution with one element per processing engine using a special execution mode in the simulator, while distinguishing the timing of an element in each of its instances. This information is relevant in task transformations and is be especially useful when assigning tasks to processing elements in the simulated network processor (as presented in Section 5.2.2).

The post-simulation stage extracts and summarizes profile information useful to the task transformation passes described in Chapter 3. An important component of the post-simulation pass is to read information produced by the pre-simulation pass: the compiler needs to know which elements are connected and in what functions it can find the code for all instances of an element. We also read a file containing the execution count for each basic block and a file containing the sequence of basic blocks executed. Generated by the simulator, we read read-after-write, write-after-read and write-after-write dependence information matrices for the elements in each memory type as well as the memory dependence mask that limits the number of elements that can have dependences even if accessing the same locations (see section 5.1.5). Also from the simulator, we get latency statistics for each element, a summary of paths taken on the task graph and a maximum list of all memory references (read and write) accessed by an element for all packets and memory types.

In the post-simulation pass, we build control flow and use-to-def structures for the intermediate representation in a format suited to our analysis. We indicate for each basic block the next most likely block(s), with the associated probabilities. Transition points across elements are also marked as such. Then we determine the frequency of occurrence of basic block sequences for each function and for the complete processing of a packet. Next, we attempt to revert the sequences of basic block to sequences of elements. Obviously, because multiple instances of an element can share the same code, there is an indetermination to which element a trace can belong to. Using a combinatorial search on the expanded chains, we can solve the problem quickly and identify what chains completely share code because of identical or different instances of the same element. We found that we can attain an important reduction of the execution time of further compiler passes by summarizing the above work in a file what we call a `trace cache`.

4.4 Summary

In this chapter, we have explained how, once we have an application described in Click's programming language, very little work is needed for the application writer to import his application into the NPIRE framework. NPIRE even provides some support to the programmer to tune the packet trace that will exercise the application. Once the application is in place, a fully automated flow instruments the element's code, generates executable code, recompiles Click with the modified elements, runs Click to produce an application trace and collects the results to refine the task optimizations that the compiler provides. In NPIRE, to facilitate NP design space exploration, these steps are performed without knowing the precise details of the target network processor. The simulator takes care of finalizing the resource allocation as we will see in the next chapter.

5 The NPIRE Framework: Simulated NP Architectures

The simulator is the largest component of the NPIRE framework. The purpose of the simulation is to evaluate our task transformations on parameterized network processor (NP) architectures. To perform this evaluation, the simulator has to understand the contents of the application trace and simulate the work of each concurrent thread. In this chapter, we present the devices that our simulator can model. We also expose the key configuration components required to make simulation measurements that reflect the maximum performance of the NP under test. Lastly, we give a brief description of the simulator's internal organization.

5.1 Simulated Hardware

Network processors are part of an active field of research in computer architecture. We are interested in studying the compilation of applications on parametric architectures and identify which architectures respond the best to our compilation techniques. In this section, we give an overview of the network processors considered and we present a list of the major simulated NP hardware structures modeled by our simulator.

5.1.1 Overview

In network processing, several packets can be processed in parallel to improve the computation throughput. As seen in the surveyed processors in Section 2.2.2.1, the majority of commercial network processors try to take advantage of this thread level parallelism. The support for paral-

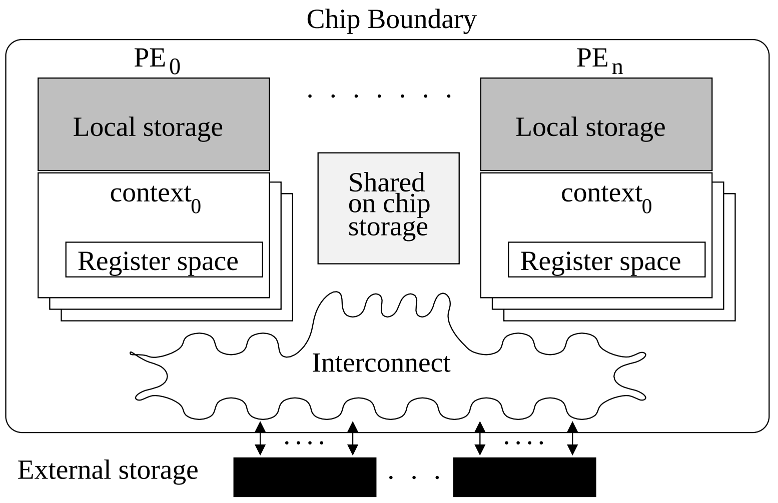


Figure 5.1: Generalized architecture of simulated NPs.

lelism exists mainly at two levels: in multiple processing elements (PEs) and in multiple execution contexts within each PE. Multithreading within each PE limits the number of stalls in each PE by increasing the number of instructions ready to be issued. When deciding the architecture of the PEs for our simulation, we identified that the processors surveyed favor in-order single issue PEs, that we will qualify as *simple*. Two reasons explain this preference: i) it is possible to put more PEs on chip because simple cores are more compact that superscalar ones; ii) multithreading leaves little room for issuing more instructions out-of-order. Recently, simple processor cores have also been found to outperform superscalar cores in *chip multithreaded multiprocessors* (CMT) on web and database transactional workloads [17]. Finally, because simple cores are easier to simulate, we are following the trend of commercial NPs in favoring the evaluation of simple core multithreaded multiprocessors. For the high-level organization of the tasks executed on PEs, we adopt the hybrid model consisting of a network of PEs. As defined in Section 2.2.2, this organization offers the most flexibility between the run-to-completion and pipelined models.

The overall architecture of the NPs that we can simulate with NPIRE is shown in Figure 5.1. A given NP has a configurable number of PEs, each with a number of hardware contexts and a local store for both instructions and data. Contexts provide the architectural support to execute multiple processes: each context can store its temporary state in a reserved section of the register file of its PE. NPIRE does not yet model the micro-architecture of PEs in detail. Instead, assuming a basic single-cycle-per-instruction model, we focus on instructions that involve communication. This approximation is also used by Crowley and Baer [14] and in the Intel IXP architecture tool [26]. The micro-architectural organization of the PEs could be explored further in another study that would nonetheless require the simulation framework and compiler assisted transformations presented here to efficiently program the NP. Our simulated NPs are also equipped with a shared on-chip SRAM and external memory, including SRAM and SDRAM channels similar to the ones found in popular network processors (as seen in the survey in Section 2.2.2.1). The design of the on-chip interconnect as well as the number of channels to external memory are both adjustable.

5.1.2 Queues & Scheduler

In our infrastructure, tasks represent the processing performed by an element on a packet, as explained in Section 4.2.1. Once tasks have been assigned to PEs, there still remains some flexibility in the scheduling of tasks to the hardware contexts of each PE. We capitalize on the fact that only heap data is persistent across task instances in a programming model such as Click's—this gives us greater flexibility in task scheduling, since a given task need not be bound to a certain hardware context. Instead, as shown in Figure 5.2, an instance of a *work unit*, consisting of a pointer to a packet and an identifier for the task to execute, is queued before each PE. A task is executed when it reaches the head of the queue and a hardware context is free to execute it. In the case of task replicas (defined in Section 3.4.1) that are assigned to multiple PEs, we model queuing such that the replicated tasks may execute on whichever of the target PEs first has a free hardware context (i.e., for task `C` in Figure 5.2). This model assumes that the code for each task or task replica has been loaded into the local instruction store of each potential target PE.

The simulator has several kinds of queues: for example, tasks, signals, and bus requests can be queued before being executed. We model some cost in terms of cycle counts and bus utilization to access all the queues. Shared work unit queues are more flexible and their access time proportionally reflects this complexity. The exact latency values that we select for our experimentations are shown in Table 6.3.

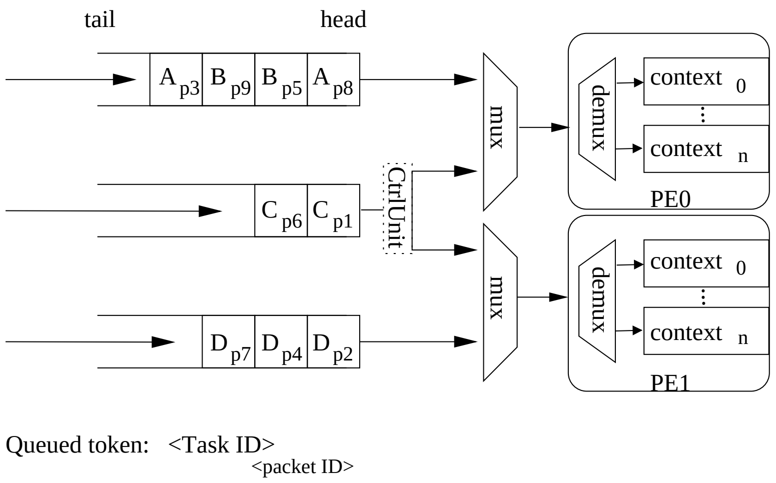


Figure 5.2: A model of task queues that allows some tasks (`A`, `B` and `D`) to be pinned to a specific PE, while a replicated task (`C`) can be dynamically scheduled onto multiple PEs. The `CtrlUnit` is a proposed controller to improve the distribution of task replicas to PEs.

With the queue system presented in Figure 5.2, a processing element can request an additional work unit as soon as it has a vacant hardware context. One optimization that we evaluated to improve on this ad hoc schedule of tasks to PEs is to interpose a small controller between the work unit queues and the PEs (the `CtrlUnit` in Figure 5.2). This controller handles the requests for work units and is free to return a work unit according to a user-defined policy or return nil. In our implementation, the controller only has the flexibility to supervise task replicas (defined in Section 3.4.1) that can execute on more than one PE. This limitation is imposed by the fixed assignment of tasks to PEs, presented in Section 5.2.2. The algorithm that we adopt in the controller attempts to load-balance the work among PEs that can execute the same task replicas. Consequently, between the PEs that can execute a given work unit, the PE that acquires the work unit is the one that has a lower or equal number of tasks executing on its contexts. Because of the number of requests for work units that are denied and retried, this proposed controller incurs more traffic on the shared on-chip bus.

5.1.3 Processing Elements

The target network processor is made of a parametric number of processing elements each having a multi-threading capability. Changing processor control from one thread to another is assumed to have a fixed latency. In our implementation, we do a context switch by transferring the control from a task as soon as it is unable to execute more instructions on the next processor cycle. The context that is either ready to execute instructions or the most likely to become ready first is then selected as the current context of the PE. Evicted tasks on a PE are reloaded before requesting new work units.

Table 5.1 shows the break down of a processing element state. Those states represent the state of the current hardware context execution in the processing element. Consequently, the average time spend in each state by a PE over a simulation allows us to infer the most frequent states of the contexts, given that contexts with computations to perform have priority over waiting contexts. We take care of breaking down the latency of the instructions in terms of memory access time and bus usage.

Keeping track of the PE state activity is non-trivial: for example, multiple requests originating from different contexts can be in flight at the same time, on the bus or in the memory controllers. This state representation is also extremely helpful while debugging to determine why a PE is not committing instructions at a certain time. Finally, the processor states breakdown enables us to add PE behaviors specific to new task transformations. For example, the `stalled` state in Table 5.1 only exists when the early signaling transformation is enabled.

For the user's convenience, the compiler can generate an application trace with the code for all the different signaling patterns related to our task transformations. The simulator can be configured to execute any combination of the transformations available.

5.1.4 Interconnect

Our simulation infrastructure must be flexible enough to experiment with different processor organizations. The interconnect is a key component of the NP architecture because it propagates messages across and off the chip, allowing PEs to communicate with memory devices and other PEs. To model a wide range or modern NPs, our simulated NP interconnect has four important features: it is

Table 5.1: Hardware context states tracked by the simulator on a per-PE basis. The state of the PE is the state of the context executing on the PE.

State	Processing element activity	State entry condition
1. **inactive**	unused processing element	initial state
2. **active**	executing instructions	computing required by a task
3. **sleeping**	waiting for memory operations to complete	slack of non-blocking memory read expired and memory access is processed by a memory device
4. **blocked**	waiting because of a load/store queue rejection (too many requests pending)	the memory controller exerts a back-pressure on a context to prevent the issue of more requests or the maximum of pending reads has been reached by the context
5. **ready**	has received a signal to execute a task	the context is ready to handle the signal
6. **stalled**	has to wait for another task to complete (only in early signaling)	task reaches a compiler inserted wait condition
7. **locked**	waiting for a lock at the entrance of a synchronized section	a lock has been requested and is unavailable or a speculative task waits to commit in order
8. **bus**	waiting because of a bus transfer	a request or waited reply is in transit one of the NP busses
9. **idle**	no activity	task completed

customizable, scalable, configurable, and modeled synchronously with the simulated time. We next explain what these concepts mean and how they benefit our experimentations.

The interconnect between memory channels and processing elements is designed to be *customizable* by modeling a network of heterogenous busses that have different latency and bandwidth characteristics. Busses can be connected in a hierarchy to experiment with different bus organizations. For example, it is possible to implement an interconnect that scales linearly in the number of hops between processors as presented by Karim et al. [39].

Our interconnect is also *configurable* such that the compiler can allocate bus segments to carry exclusively specific messages. The interconnect then autonomously decides where to route messages, such as incoming packets, data from/to external RAM, shared RAM accesses, and synchronization signals.

Because requests may have to travel in different wires and because data may go through inter-PE bypasses or may be found in a local storage, we need to handle the interconnection in a *scalable* fashion. For this reason, we have a software routing layer that takes care of putting event packets on the wires. This indirection layer allows for arbitrary wiring scenarios.

In the NPIRE simulator, we model the interconnect *synchronously with the simulated time*, i.e. the state of the NP is changed when requests are processed. Other simulators, such as the out-of-order Simplescalar simulator [4], can change the state of the memory before any request reaches the memory device. Our delayed execution allows us to, for example, reorder transactions on a bus and postpone accordingly a synchronization message between PEs. Our modelization hence makes it easier to implement extensions or interfaces between simulated hardware modules. There are two main aspects of simulation that allow us to model accurately bus delays: request sizing and queue modeling. First, when we add a request on to a bus controller's queue, we need to send enough information so that the request can be issued on the serving peripheral and the result returned to the appropriate entity. In fact, there is enough information encoded with the request to allow for its proper execution and to have a reasonable appreciation of the size of the transfer on the interconnect. Second, the simulator handles cases when a request needs to be queued in several queue units in sequence. For example, for a memory read, the bus controller and memory controllers make it impossible for the simulator to predict upon issue when a request will reach its handler in the future. For context switching purposes, when the completion time for an event is not known in the simulator, it is set to infinity. This completion time and the processor state (presented in Section 5.1.3) is updated as the event message becomes in transit on a bus or is being handled.

5.1.5 Modeling Memory Devices

To handle load and store task instructions, our simulation framework models memory units at three levels of a hierarchy: there exists storage local to PEs, shared on-chip and external. i.e. off-chip, as illustrated in Figure 5.1. As seen in the network processor surveyed in Section 2.2.2.1, most NPs do not have hardware caches because of the low locality of the data accessed by tasks. Our work adheres to this absence of cache and lets the software control when the data should be copied from one memory unit to another. NPIRE's memory modeling is divided in two parts: first, we present memory simulation aspects, then we describe the memory access profiling performed in the simulator.

When considering the timing of the memory devices, modern network processors utilize various memory technologies, such as SRAM, DDR, RDRAM and RDLRAM, that have different timing specifications. Also, several memory access optimizations specific to DRAM have been published in the literature. For example, Hasan et al. [30] propose a smart memory controller to exploit row locality in the context of network processing using SDRAM devices. Our goal with NPIRE's simulator is to make generalizable observations targetted at identifying performance bottlenecks. We next explain how we can achieve this objective by modeling our memory units in a technology-independent fashion.

Although our simulator has detailed models of memory devices, we realized that those models alone do not improve the accuracy of our simulation. As explained in Intel's discussion on DRAM performance on an IXP processor [34], the latency of memory operations is relatively constant for random memory accesses until the memory unit is over-subscribed. After experimenting with the Intel IXP SDK, we realized that the memory access time on an external memory device only represents a small fraction of the total latency experienced by the application. Reasons include the latency incurred by the communication on system busses and the crossing of the chip boundary. Consequently, the exact timing of the memory devices need not be replicated exactly to obtain a good appreciation of the memory access time.

To be able to simulate timing accurately for adequate bottleneck identification, we realized the importance of having a parametric breakdown of the latency of memory operations. On network processors, because of pipelining in the interconnect as well as in the memory units, we have to model an overall latency, of which a large fraction can be overlapped with other transactions. In particular, the total latency has a *pipelined component* and an *un-pipelined component*. The memory access latency that is serialized between requests, i.e. unpipelined, is based on the address (aligned/unaligned), size, storage type and kind (read or write) of an access. The latencies can be either deterministic or with a random jitter. In our simulator, statistics are collected on all memory and bus transactions in terms of in their average duration and queue time.

In our simulator, memory reads and writes are always handled in an asynchronous fashion as explained in Section 4.2.3.3. This is a characteristic of modern network processors such as the Intel IXP NPs [35]. To keep track of the pending memory loads, each hardware contexts as a limited number of structures analoguous to 'miss information/status handling registers" (MSHRs) [43]. Also, to model realistic hardware provision, memory device controllers have limited size queues. When a memory device can not accommodate an additional request already issued on a memory bus, the request is dropped and the sender task must re-send its request on the memory bus when the memory controller signals the task that space is available in the queue. Our memory controllers do not yet support complex memory operations such as "atomic test-and-set".

The simulator can have multiple channels of external memory. In fact, this is a requirement to reproduce popular network processors. When distributing the packets on multiple equivalent external memory channels, we need to create an association between a packet and an external memory device. This association does not change over time and is performed when the packet is first made available to the application. In our simulation, we save the compiler-inserted tags of pending memory access in the devices they get executed on. This can be used to provide feedback to the compiler to, for example, identify expensive loads from off-chip memory.

Integrated with the memory modeling, the NPIRE simulator tracks memory accesses as they are dispatched to different memory devices. This accounting code does not model any structure in the processor; as we explain next, it generates part of the profiling data fed back to the compiler.

Memory Accounting In the simulator, there is a *memory manager* that records all accesses to each buffer. For the packet data, packet meta data, and heap references, the simulator attempts to create a set consisting of the union of all references. Obviously, for allocated buffers, like packets, the references are always aligned to the start of the buffer. For the heap references, we try to merge consecutive regions of accesses as they get recorded. There are sanity checks on memory accesses that verify that the memory allocation and free are executed in proper sequence with the memory accesses. As well, we inserted code in the simulator to verify that the packet memory has been freed after each packet has been processed.

While doing this memory accounting, some information is exchanged with a *dependence checker*. This dependence checker identifies which tasks access the same memory locations. Another of its goals is to find locations that are accessed by a task while processing the majority of packets. After a few packets, the memory classifier tries to establish if there is a recurrent number of memory references performed by a task. If a consensus is reached on the most frequent memory accesses between task executions, the selected accesses are regrouped and are used to implement memory batching (defined in Section 3.2.1).

In collaboration with the memory manager, the dependence checker establishes a *dependence mask* which is a matrix used to determine which elements will never have a dependence between each other. Inter-task dependences are used to identify candidate tasks for early signaling, as explained in Section 3.4.3.

5.2 Configuration Components

This section describes how our infrastructure binds our benchmark applications to the simulated network processors by controlling the packet input and output and assigning tasks to processing elements. All the techniques below rely on an iterative process between a configuration generator, or a manual intervention, and a cost evaluation in the simulator.

5.2.1 Rate Control

Packets flow in and out of network processors. The maximum rate at which packets can arrive and depart is determined by the physical links of the network. For this reason, our simulation infrastructure must provide abstractions external to the NP under test to control packets.

Input packet buffering Because NP input buffers for incoming packets are typically small, they can overflow rapidly. To prevent the loss of packets, we prioritize the acceptance of packets over tasks pending on processing elements on which the packet receiving tasks are mapped.

When packets arrive, work units are created and inserted for the tasks at the input of the task graph. Another approach, *input reordering* [49], assumes that a classification of packets could be applied at the input of the system to dispatch packets to PEs such that the number of dependences between the tasks on those PEs is minimized. We currently do not have compiler support for the later in our infrastructure; in fact, the head-of-line classification [19, 77] could very well be described in a Click element and integrated to the application's task graph.

Input Packet Rate "Headroom" is defined by the authors of the Netbind paper [9] as the amount of instructions that can be executed in the critical path beyond the minimum application requirements. Although headroom is desirable, to evaluate a processor, we have to stress the application to a point where the headroom is nil. In such a setting, we can determine reliably what is slowing down the application most, i.e. the *bottleneck*. Hence our infrastructure must find the operating point where the system is saturated.

It has long been understood that self-similarity in real network traffic implies bursty-ness [70]. However, when we try and characterize our system, we are interested in the maximum sustainable rate and not by the fact that occasional slow inter-arrivals and packet loss help regulate traffic bursts. Also, in our case, because we can accommodate variance in task durations by buffering incoming tasks, we are not concerned with realtime guarantees as are El-Haj-Mahmoud et al. [21]. Furthermore, since our applications have unpredictable control flow and, in some cases, loops in the task graph, the worst-case execution time [79] is an inappropriate metric.

The saturation point should be carefully selected because if a large number of work units are queued, the system appears to have less idle time and has always more work to overlap with system latencies. In a real system, this is not necessarily the case: in the Nepsim project [50], important levels of idleness were measured. On the other hand, measuring a system with slack, i.e. not rate determined by a critical path, does not reflect its actual potential performance.

In kernel benchmarks, packets are often taken one by one at the same rate as they are processed. Making packets arrive in the system at exactly the time when the previous packet exits is highly inefficient because it disregards the available task pipelining possible. In our case, the arrival process models packet arrival on potentially multiple network interfaces.

Because we store tasks to be executed in a queue at the input of a processing element, our system could be modeled as a network of queues. Given a packet input rate, proving the stability of a network of queues where a task can be replicated and modeling the contention of multiple concurrent tasks on different shared resources is very complex.

The “maximum loss-free forwarding rate” used by Crowley et al. [14] relies on not exceeding 80% of utilization on any shared resource. This disallows high utilization ratios, that may be desirable even at low packet rates if the programmer wants to make the most of his system.

In our simulation infrastructure, the input packet rate can be bandwidth controlled or controlled by a packet-per-cycle metric. To find the point where the system operates at saturation, our first approach was to find the rate at which the number of packets queued in the system was not steadily increasing over a large period of time. We found that this was a good method in general but can run into corner cases due to the discrete nature of the measurements made. We next explain the approaches used to make our measurements in header-based and payload processing.

For header processing applications, such as routing, a fixed packet rate with respect to processor cycles is more appropriate because the size of the packet is irrelevant. Figure 5.3 shows how we can find the saturation point for a given NP—in this case an NP with 12 PEs, executing a routing application with replication supported. Since there are two packet input sources, the mean packet

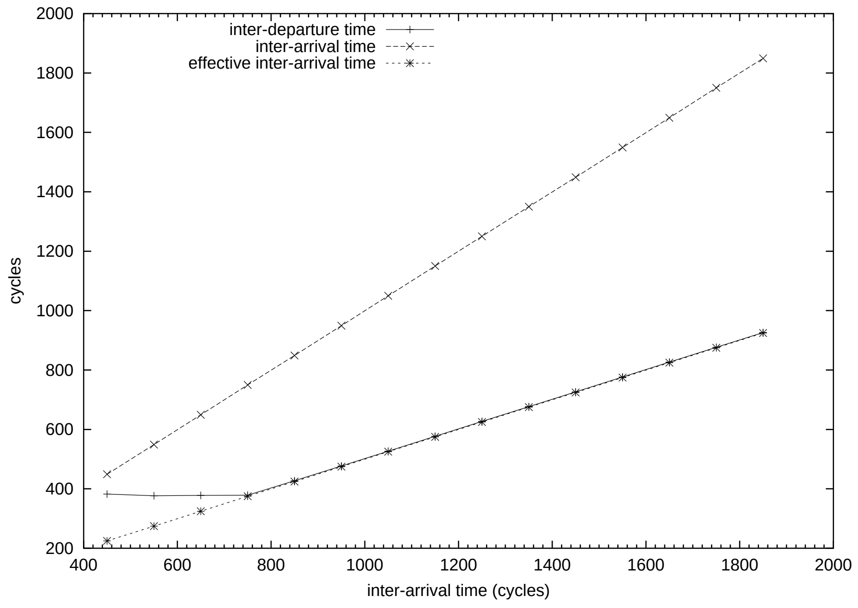


Figure 5.3: Finding the saturation point. Since there are two packet sources, their mean inter-arrival time (the line $y = x$) results in an effective inter-arrival time of half as much. The saturation point is the smallest effective inter-arrival time where the inter-departure time is equivalent (i.e., the NP can keep up).

inter-arrival time (the line $y = x$) of each source results in an overall effective packet inter-arrival time of half as many cycles. Hence the saturation point for that NP is the smallest effective packet inter-arrival time where the packet inter-departure time is equivalent (i.e., the NP can keep up). In the figure, we see that the inter-departure time begins to deviate at the effective inter-arrival time of about 750 cycles, which is the saturation point. NPIRE uses this method for finding the saturation point by running a bisection search on potential packet inter-arrival times. Because the inter-departure has a large per-packet variance, we use statistical hypothesis testing to determine if we have enough inter-departure samples to make our rate measurements. Our packet inputs sources are configured in a text file: we insert a time offset between them so that packets do not arrive at the same time on all interfaces, which would be a worst case because of the bursty demand on the processing elements.

For payload oriented applications, such as compression and cryptography, byte rate control is required because the size of the packets is relevant to the amount of processing. We use the same approach as for headed-based processing of comparing the output rate with the input rate to determine the saturation input rate.

Packet Departure Output packet control has been described in Section 3.1. Each output is preceded by a `Queue` element. The dequeue operation can take into account the output bit rate per port. So our output control delays the start of the dequeue operation until the output unit is allowed to resume.

5.2.2 Task Mapping

To share the load between the different processing elements, parallel tasks have to be dispatched to processing engines in a careful fashion. Wild et al. [93] explain that finding the best assignment of tasks to processing elements is an NP-complete problem. Although, we have to map tasks instead of clusters of basic blocks, we also use an iterative approach involving a cost evaluation at each step.

Plishker et al. [65] formulate mapping as an integer linear program problem where the task instructions have to fit in the in the PE control stores which leads them to consider instructions that can be shared or not between contexts. Because, at the simulation level, we want the application to drive the architecture, space available for instructions PEs is not considered in out mapping process. As opposed to Weng et al. [92], we allow to map more than one task per PE.

Each element of the application must be assigned to at least one processing engine. The objective of mapping is to allocate enough processing resources to each task and maximize the system throughput by hopefully increasing parallelism at the packet level. In our measurements, this assignment is done only once for the duration of the simulation. Because our simulated network processor does not have any implicit coherence scheme (such as a MESI protocol), we assign a task to a PE and other PEs wanting to access its data have to do so remotely via an on-chip communication channel.

Automatically mapping the specified task graph to an NP's PEs is perhaps the most challenging problem of all. In such a mapping, there is a strong tension between locality and parallelism.

Locality is optimized by mapping related tasks to few PEs so that storage and communication are minimized, while parallelism is improved by mapping tasks to many PEs and exploiting more resources.

Mapping file Our methodology is similar to what Ennals et al. [23] describe as a “new approach”: we separate the high-level application functionality from the architectural details of the NP. We also have a component related to the other authors’ “Architecture Mapping Script (AMS)” that describes the mapping of tasks to hardware contexts. Our approach is also to make the code portable and let the compiler do the bulk of the chip-specific transformations. Contexts can be allocated to elements so that an element can benefit from a custom number of contexts. For now, we however allocate the same number of contexts to all PEs.

Our mapping process in NPIRE is iterative, based on feedback from simulation, and proceeds in the following steps:

1. An initial measurement is made where each task runs on its own PE, assuming an infinite number of PEs.
2. Using a greedy algorithm, we then re-assign tasks to the actual PEs available while trying to minimize the expected utilization of each PE.
3. We then try to alleviate the worst bottleneck by replicating the task with the largest queue time. Replicas can optionally be assigned to the same PE, or to different PEs. We repeat this step until the NP is well utilized.
4. Once the base mapping is decided, we attempt to improve it through simulated annealing which uses a fast, coarse-grain simulator to provide fast feedback. Given an initial seed, the *fast simulator* makes random modifications in the mapping and commits them according to a certain probability depending if they improve and worsen the throughput. During this process, the algorithm avoids moving any task replicas to the same PE. This algorithm was found to give better results than other algorithms (see the section A.2), and appeared to converge very rapidly. For this reason, our criteria for termination is a time limit.

5.2.3 Automated Bottleneck Identification

To iteratively improve the binding of the task to the simulated NP, we need to identify the causes of bottleneck. To help the process, we identified a list of the sources of bottleneck: Table 5.2 also reports what we can do to alleviate each of them. The last three sources of bottleneck in the table, i.e CPU, bus and memory utilization, are strictly architecture-specific. We can see that the objective of parallelizing the application is balanced by the resulting dependences between task replicas.

Because the synchronization is the only limit to parallelization assuming infinite hardware, our simulator has code to quantify the overheads of synchronization. Figure 5.4 presents possible dependence situations. The protected accesses can be either reads or writes. By recording the memory access to persistent heap for each synchronized task, we can track the time between dependences. In the case `A` of the figure, we can observe overly conservative synchronization because there is not enough parallelism to have more than one task accessing shared data at a time. In the case `B`, the access occurs before the last task attempts to enter the synchronized section. In case `C`, synchronization is effective in protecting a memory access. The simulator can measure the time distance between the accesses to shared data when the synchronization primitives stall a task.

5.3 Simulator Internals

To explain how we obtained our results and give an overview of the simulator organization, we next describe some important aspects of the simulator, with a focus on the trace manipulations, the main loop in the simulator and the iterative simulation flow effect on the simulator configuration.

5.3.1 Loading the Application Trace

Because the application trace (introduced in Section 4.2.3) can be arbitrarily long and its in-memory representation is even more voluminous, we implemented a streaming mechanism to go through the trace file. At the beginning of the trace is the connectivity information of the configuration. Next there are task start markers followed by memory allocation, read, write and free nodes. Each node is marked with a time stamp: the difference in consecutive record time stamps indicates the number of

Table 5.2: Sources of bottleneck and our approaches to alleviate them.

Symptoms	**Remedy techniques available**
Load imbalance due to uneven-ness in the mapping.	Task splitting along with our elaborate mapping approach.
Context switching resulting in a low utilization of all the contexts available.	Our state tracking of the PEs tries to overlap computation with communicating instructions in the PE.
Signaling overheads.	Next-neighbor links, inter-PE bus and broadcast techniques can be used. Also, the mapping can try an add locality to the task dispatching.
Input interface not fast enough.	Our saturation process makes sure that packets arrive fast enough.
Bad partitioning (long tasks and short tasks).	We have a software component that requests splitting of individual tasks when it is found that load imbalance may be the dominant bottleneck.
Contention on the output ports, a symptom that the output rate is too slow.	Unavoidable for rate-controlled interfaces.
Output ordering.	Can require to adjust the output rate control.
Synchronization limits parallelism.	Our task transformations try to expose as much parallelism as possible.
CPU resources exhaustion, when to be observed on all PEs.	We are able to simulate additional PEs and contexts.
Shared bus utilization.	We are able to add additional interconnect capacity.
Memory utilization saturation.	We are able to add memory bandwidth.

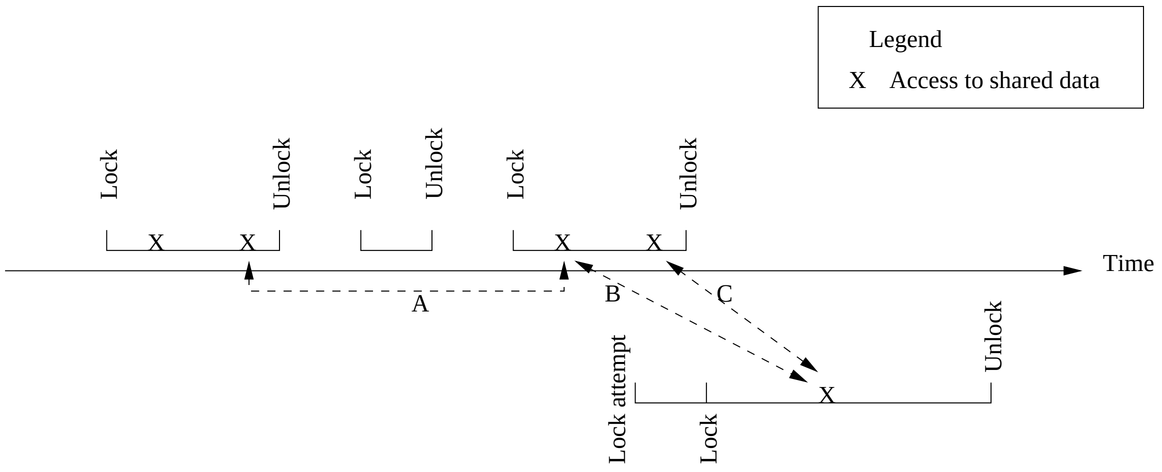


Figure 5.4: Dependence scenarios. `A` measures the time distance between dependent memory accesses. `B` and `C` measure the delay imposed by dependences and can be compared to the delay imposed by the `lock` and `unlock` primitives.

instructions to be executed between records, assuming a single cycle per instruction. Trace entries loaded in memory are linked in a hierarchical fashion. End of task markers complicate the reading because tasks are often called within another. Because time stamps are used to evaluate the latency between operations, care must be taken to timestamp properly the end of task markers.

The trace loader can be used independently for debugging purposes and to get statistics on the content of the trace, such as the number of instructions or memory references per element.

As soon as the first section of the trace is loaded, the simulator starts executing it. Because of memory constraints, the simulator must be able to delete the trace nodes as it progresses. Memory deallocation for the loaded segments of the trace is non trivial, and even more so with early signaling, because trace records are can be referenced from multiple locations in the simulator. The use of the Standard Template Library (STL) structures was found to be problematic in some cases because STL has its own pooling mechanisms that does not always favor reuse to allocation.

5.3.2 Main Simulator Loop

Because interrogating all processing engines on all cycles would be too time consuming, our simulator is event driven, i.e. it only executes cycles where events happen. The main processor loop updates all the devices and processor states as events happen. The performance of a simulation can be estimated by looking at the total number of packets processed and the simulated time required to do so.

5.3.3 Configuring the Simulator

Our current exploration methodology starts with trying to find a scalable execution context for a benchmark. First, the time length of the simulation is manually found so that the simulation can be measured in a steady state. All simulation statistics are reset after a configurable number of cycles to get rid of the initial NP filling with tasks transient effect. To find a representative configuration for our benchmark, we need to stress the network processor to the limit of its capabilities in that given configuration. Finding the best configuration of a given architecture makes the problem cyclic as shown in Figure 3.9. Indeed, the incoming rate of packets in the simulator can be made faster when, by changing the configuration, we also improve the throughput. A criteria to validate our methodology is that the throughput of the network processor should scale well with a varying number of PEs.

5.4 Validation of the Simulation Infrastructure

In the interest of validating our infrastructure, we have support to record traces out of an IXP1200 simulator's benchmarks, Nepsim [50], and replay them on our system. We traced this other simulator to extract the main simulation parameters for our experimentations, as shown in Table 6.3 (Section 6.3). We used program counter sampling to determine tasks boundaries and recorded packet arrival events. We were unable to recover the NAT benchmark from Nepsim, because busy waiting in that benchmark blurs our interpretation of the tasks transitions. While simulating the IPfwdr, MD4 and URL benchmarks from Nepsim, we obtained comparable relative packet throughput numbers.

We also observed scaling with the number of processing elements similar to the numbers presented in Nepsim's paper [50]. However, this comparison does not go much deeper because we were unable to fully reverse-engineer the use-to-def relations of memory operations to replicate exactly the context switching points in our simulator.

NPIRE can also export its applications to the Intel Architecture Tool [26] where we can simulate them on most recent IXP processors. However, because in this Intel tool it is uneasy to saturate the input rate of the application and the utilization of devices is computed using the utilization of the device arbiters rather than the shared devices themselves, we have not found a solid basis for comparison.

5.5 Summary

In this section we gave some insight on how the simulator works to shed some light on our experimental results. We gave an overview of the parametric NP architecture that we use and of our simulated hardware structures. We also presented the importance of making measurements when the NP was working at a saturation input rate and how this rate is obtained experimentally using a bisection algorithm. Some emphasis was put on the mapping process because it is key to load balancing, along with task scheduling. We showed the different symptoms of bottleneck that our simulator could detect and concluded with a high level description of the simulation process. In the next chapter, we present the results obtained with the conciliated efforts of software tools presented in Chapter 4 and the simulated NP hardware.

6 Evaluting NPs with NPIRE

In this chapter, we evaluate the impact of our task transformations using the NPIRE compilation infrastructure presented in Chapter 4 and our network processor simulator described in Chapter 5. Our observations are aimed at evaluating how our application transformations in terms of memory, thread and task management affect the packet throughput of our simulated NPs. As well, we want to identify the performance bottlenecks in our applications, i.e. determine if it is the application's resource usage or specific NP architectural components that limit the packet throughput. After describing our benchmarks and simulation parameters, we proceed to analyze the data collected.

6.1 Applications

In this chapter, we evaluate our task transformations using NPIRE's simulation infrastructure. For this, we selected a representative set of applications. In selecting benchmark applications, we attempted to fulfill the three important goals of NP benchmarks identified by Tsai et al. [84] : (i) our benchmarks model applications that are representative of network processor usage, (ii) they provide results that are comparable across network processors, and (iii) they provide results that are indicative of real-world application performance.

Table 6.1 describes the four applications that constitute our benchmark suite. The first two applications perform IP header processing—i.e., applications for which the packet payload is irrelevant. We use a RFC1812-compliant router (ROUTER) and a network address translation application (NAT, described in Section 3.1), both adapted from those created by Kohler *et. al.* [41]. Those two benchmarks are also considered as a reference by Tsai et al. [84].

Our payload processing applications have the same element connectivity as the task graph shown

Table 6.1: NP applications studied: Router and NAT are header-based processing applications while LZF and DES are payload-based processing applications.

Application	Num. Dyn. Tasks		Instrs per Packet		Loads per Packet		Stores per Packet		Instrs per MemRef	Synch. ratio	
	Avg.	+/-	**Avg.**	+/-	**Avg.**	+/-	**Avg.**	+/-	**Avg.**	**Avg.**	+/-
ROUTER	16	1.12	2353	258	127	21	24	3	16	0.17	0.07
NAT	12	3.37	4857	2206	234	92	58	39	17	0.32	0.20
DES	10	3.93	564925	326813	23189	13403	2024	1169	22	0.08	0.17
LZF	10	3.93	190026	213897	2138	1672	4215	4581	30	0.11	0.23

Table 6.2: Memory requirements of our benchmark applications, and the storage devices to which each memory type is mapped. For each memory type, we show the average amount of data accessed per packet.

Device:	External SRAM	External DRAM	Local storage	Registers	External SRAM
Type / **Application**	**Packet Descriptor**	**Packet Payload**	**Persistent Heap**	**Stack**	**Temporary Heap**
ROUTER	42B	23B	5B	0B	44B
NAT	36B	45B	22B	96B	49B
DES	36B	1500B	48B	2100B	20B
LZF	36B	1500B	48B	200B	600B

in Figure 4.2. DES performs packet encryption and decryption and LZF, packet compression and decompression. The DES cryptographic elements originate from the Click [41] element library. Encryption is a popular NP application, used in NP benchmarking by Ramaswamy et al. [67] and Lee et al. [48]. In our other payload processing application, the compression elements are custom made from the LZF library adapted from the ADOC project [37]. The packet related aspects of this compression are inspired from the IPComp–RFC2393 standard, and the Linux kernel sources. The compressed packets generated by LZF comply with the IPComp standard and can be decompressed independently from other packets. We do not support the more recent LZS (RFC2395) compression algorithm because its source code was not available to us at the time of writing. NP-based packet compression is interesting because Jeannot et al. [37] have shown that host-based compression alone could improve the latency of distributed computations by 340%.

For each application, we report in Table 6.1 the average (Avg.) and standard deviation (+/-) across a large number of packets of the total number of dynamic tasks, number of instructions, loads, and stores executed per packet, the number of instructions executed per memory reference, and the fraction of execution time spent on synchronization (synch. ratio). While not shown in the table, some tasks have a large fraction of dynamic instructions inside a synchronized section, specifically the `TCPRewriter` from `NAT` (83%) and the `Queue` elements (50%) used in all benchmarks. In contrast, the IP header checksum task is free of synchronization. Finally, Table 6.2 shows the average amount of data accessed for each buffer type per packet, as well as the devices to which each memory type is mapped.

We measure our benchmark applications using modified packet traces (see section 4.2.2) from the Teragrid-I 10GigE NLANR trace [61]. All of our applications have two input and two output packet streams, as exemplified in Figure 4.2. This choice in the number of packet interfaces, resulting in an equal number of tasks, is a balance between two organizations of the work in an NP. In some processors, the same task must process all incoming packets, while on other processors, such as the Motorola C-5e (introduced in Section 2.2.2.1), processing resources can be allocated to each input packet stream. As a result of this work organization, in our applications, the two input packet streams can have non-trivial interactions when they contend for the same elements, for example, the `LinearIPLookup` element in Figure 4.2.

6.2 Architectural parameters

With our simulator, we attempt to model resource usage and delay characteristics similar to what we could observe when executing the same code on a physical network processor. Hence, in this work, to allow for representative simulations, we use realistic parameters to configure our simulated chips. As network processors evolve and become equipped with faster processing elements, they are confronted to the fact that the off-chip memory throughput does not scale as fast as the clock rate of processing elements, a reality known as the “memory wall” [97]. To show this trend, we perform our evaluation on two network processors, *NP1* and *NP2* respectively modeled after the IXP1200 and IXP2800 network processors introduced in Section 2.2.2.1. These Intel processors have respectively 6 and 16 PEs; however we will consider a variable number of PEs to identify scalability limits. The simulation parameters in Table 6.3 were obtained as a result of our validation experiments with the IXP SDK and Nepsim, presented in Section 5.4.

As shown in Table 6.3, an important difference between the NP1 and the NP2 processors is their PE clock frequency. The NP2 has a clock rate 4.3 times faster, however, the latency of its memory and bus operations is between 2 (remote PE access) and 40 times (on-chip shared SRAM bus access) higher than on the NP1. The NP2 has 3 DRAM and 4 SRAM external memory channels along with the doubled number of contexts per PE (8, versus 4 for the NP1). The NP1 has one bus for DRAM and one for SRAM. On the other hand, in the NP2, DRAM transactions transit on 4 busses: 1 bus for reads and writes for each half of the PEs. The SRAM is also accessed through 4 on-chip busses that have the same organization as the DRAM busses. This additional supply of hardware on the NP2 are intended to compensate for the relative increased memory latency by increasing the packet processing throughput.

Table 6.3 summarizes our simulation parameters, in particular the latencies to access the various storage types available. Each PE has access to shared on-chip SRAM, external DRAM and SRAM through separate buses, and certain shared registers on remote PEs through another bus. Usage of the buses and storage each have both non-pipelined and pipelined components. Each PE also has faster access to local storage, its own registers, and certain registers of its next-neighbor PEs.

The initialization/configuration phase of our benchmarks can safely be ignored because we are concerned with the steady state throughput of our applications. After initialization, the simulation is run for 6 Mcycles for payload-processing applications, and 20 Mcycles for the payload processing application. Those run times were the shortest run times empirically found to give a reliable estimate of the steady state throughput of the NPs.

Table 6.3: Simulation parameters. The base total latency to access a form of storage is equal to the sum of all parts. For example, to access external DRAM takes $10+2+17+26=55$ cycles, 43 of which are pipelined. An additional amount of 1 pipelined cycle is added for each 4 bytes transfered (to model 32 bits busses).

Storage Type		NP1 Non-pipelined (cycles)	NP1 Pipelined (cycles)	NP2 Non-pipelined (cycles)	NP2 Pipelined (cycles)
External DRAM	access	10	17	12	R 226 / W 0
	bus	2	26	4	59
External SRAM	access	4	8	5	81
	bus	2	10	4	51
On-chip shared SRAM	access	1	1	3	R 21 / W 8
	bus	0	1	3	37
Remote PE registers	access	1	2	1	12
	bus	0	1	1	1
Local store		0	1	4	11
Registers		1	0	4	0
Next-neighbor PE registers		1	1	4	4

Other Parameters	**Value** NP1	**Value** NP2
processing element frequency	232 MHz	1 GHz
hardware contexts per PE	4	8
rollback on failed speculation	15 cycles	40 cycles
queue size for bus and memory controllers	10	40
pending loads allowed per context	3	
context switch latency	0 cycle	

6.3 Measurements

One of the main metrics that we use to measure the performance of the simulated NP is the maximum allowable packet input rate of the processor—that is, the point where it operates at saturation (as explained in Section 5.2.1). For convenience, we will refer to this metric as the I_{max} rate. We define the fraction of time that a bus or a memory unit is servicing requests as its *utilization*. This definition applies also to the locks used in synchronization: their utilization is the fraction of time that they are held by a task. In this section, we define a number of constant parameters for our simulations and present our task transformations implementation results.

6.3.1 Choice of fixed parameters

To present consistent results, we performed some preliminary experiments to fix a number of simulation parameters. Our preliminary experiments were performed on 2 benchmarks, `NAT` and `Router` and 2 reference systems: NP1 equipped with 6PE, called `REF1`, and NP2 with 16PE, called `REF2`. This number of PEs corresponds to the resources present on the corresponding IXP1200 and IXP2800 NPs. For those preliminary experiments, replication is enabled for all tasks and tasks can execute on any context on all PEs so that no mapping is required. In this section, we explain our settings for packet sorting, the thread management techniques, the scheduling controller in the queues, our queue timing modeling and our iterative splitting experiments.

We observed that packet sorting on the output of the NP has a very small impact on the I_{max} rate. However, its support, as presented in Section 3.1, requires extra communication and complicates the early signaling transformation because of the task reordering that takes place. To be able to easily identify to factors impacting throughput, we disabled packet sorting in the `Queue` elements (introduced in Section 3.1).

To perform our experiments, we had to select which thread management techniques we would adopt. Two techniques were proposed in Section 3.5: a priority system allowing threads to have a balanced utilization of the on-chip busses; and a preemption system that favours tasks inside synchronized sections. For all simulations, we found that the bus priority system improved the throughput of our benchmarks on REF1 by 1% and of REF2 by 25%. However, the preemption system only improved the `NAT` benchmark on REF1, while either affecting negatively or leaving unaffected the other benchmarks. The performance improvement was on the order of 4% for `NAT` on REF1. Consequently, we decided to enable the priority system and disable the preemption system in our experiments.

We evaluated the controller in the work unit queues presented in 5.1.2. This controller does not assign a work unit to a context if assigning the task to a context on another PE would improve the load balance of PEs. The objective of this controller is hence to tentatively improve on the ad hoc scheduling of tasks to PEs. Adding the controller improves the throughput of NAT by 50% over both reference network processors but degrades the throughput of Router by 17%. We observed that making slight input rate variations could change significantly those throughput figures. In consequence, adding the control unit reduces the noise in our throughput results due to the dynamic assignment of work units to PEs. We can readily see that scheduling can significantly impact the results, and future work should target this aspect of the processing. The adverse affect of the control unit on Router is due to the fact that, while the control unit improves the load balance, its decisions add latency to the work unit dispatching. As a result, the controller limits the number of concurrent packets by 15% on Router. Nonetheless, because of the gains seen on NAT, we chose to use the scheduling control unit for all the following experiments.

Our simulation uses queues in a shared on chip storage to distribute tasks to processing elements. The manipulation of work units (presented in 5.1.2) is achieved by writes and reads to the on-chip storage to respectively enqueue and dequeue work units. Those memory accesses to shared memory penalize important large-scale replication by increasing the contention on the shared on-chip storage. For our other experiments, we do not model those memory and bus accesses due to the task queues because we want large-scale replication to be a reference best case scenario. This scenario indeed provides the highest number of PEs for tasks to execute on. Modeling the contention on work unit queues counters the benefits of replication and complicates the characterization of other on-chip contention factors impacting the throughput of our applications, such as high demand on a particular memory unit.

For our splitting experiments, we iteratively find the I_{max} for the task graph with one to five iterations of splitting, and retain the best throughput value. In our test cases, we found that there was no benefit in doing more than five iterations of splitting. One explanation for this diminishing return is that each task split incurs inter-split communication overheads. The other explanation is that, as explained in Section 3.4.2, the task splitting compiler pass is constrained on where it can insert splits: it does not insert splits in a tight loop inside a task.

6.3.2 Impact of the Task Transformations on Packet Processing Throughput

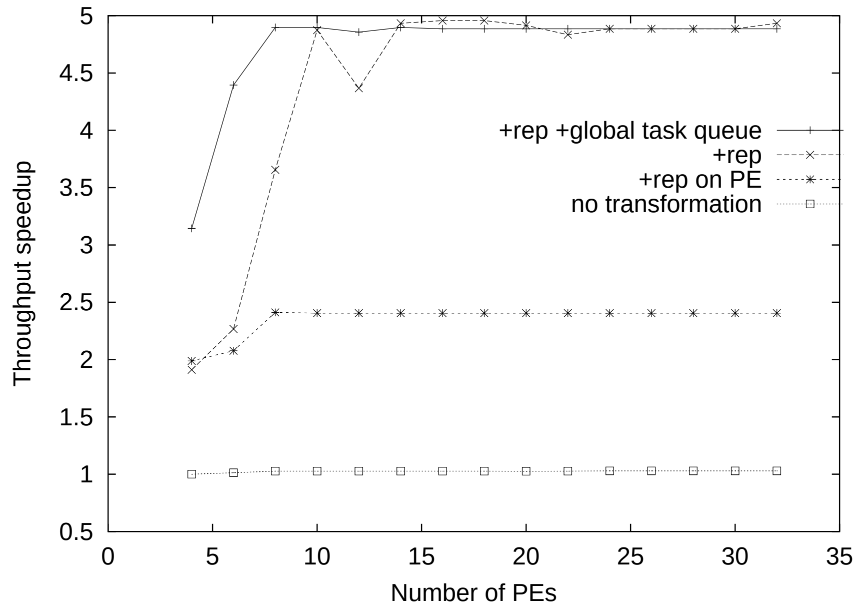
In this section, we evaluate NPIRE's task transformations in a way that allows our conclusions to be generalized to a large number of NPs. First, we simulate two NP architectures, NP1 and NP2, presented in Section 6.2. Second, we measure the impact on packet throughput as the underlying architecture scales to larger numbers of PEs. These two architectural axes, chip organization and number of PEs, allow us to evaluate NPs with support for different communication over computation ratios.

In our graphs, we normalize all our measurements to the application with no transformation on four PEs, the minimum number of PEs evaluated. This number is small compared to the other processors surveyed in Section 2.2.2.1 and it is the minimum number allowing us to bind one PE per input and output interface (we have two input and two output streams as explained in Section 6.1).

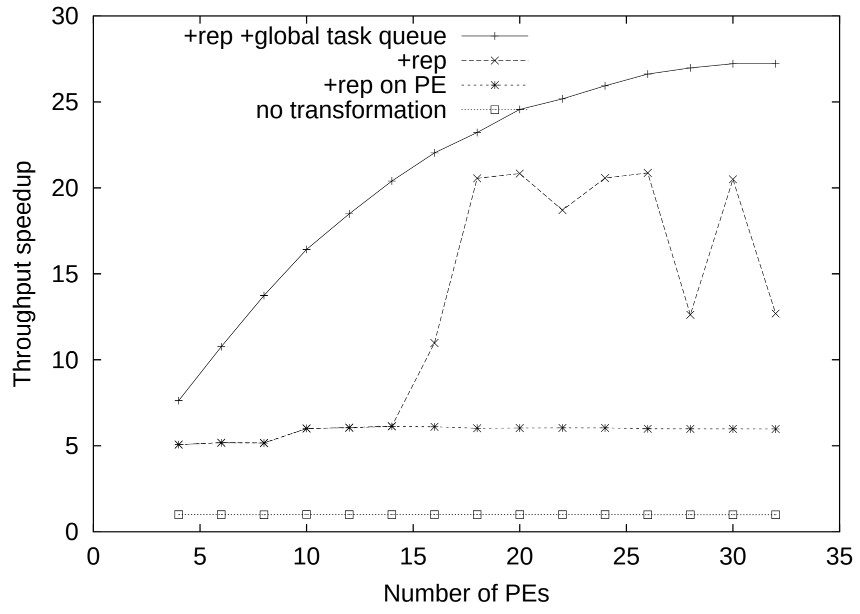
6.3.2.1 Replication

In this section, we evaluate four different replication scenarios for the replication task transformation presented in Section 3.4.1. The simplest scenario has no replication and simply extends the mapping to the number of available PEs. Next, we present the case where replication of a task is limited to each PE, meaning that a task can execute on any number of available contexts on a PE. More task replication leads to the case where one task can execute on any context of a selected set of PEs, as determined by the mapping process (presented in Section 5.2.2). We call this last replication scheme *subset replication*. The final case that we examine is where one task can execute on any context of any PE: we refer to this model as having a *global task queue*.

Router on NP1 For `Router` on NP1, we can see in Figure 6.1(a) that simply spreading the tasks of the application with no replication on a greater number of PEs only improves the throughput by 2.9%. The maximum throughput is reached with 24 PEs: this low performance gain with a large number of PEs underlines the need for efficient replication. The replication limited to a PE and the global task queue schemes reach their maximum throughput with 8 PEs. The subset replication has

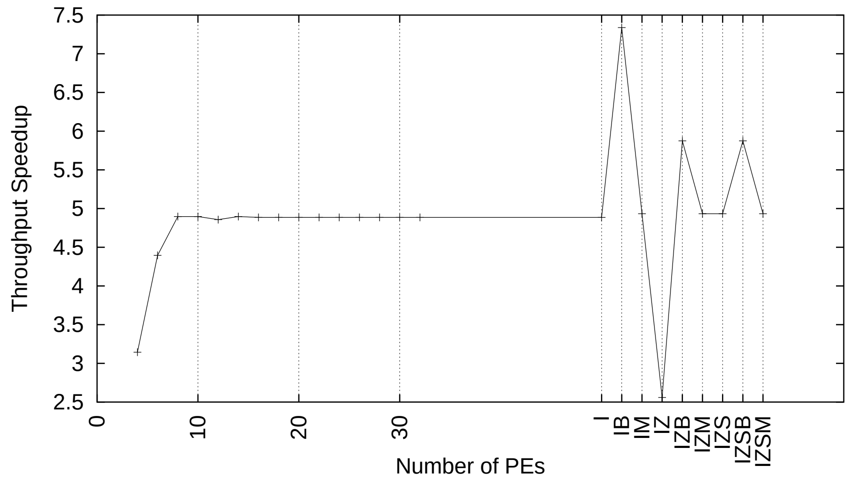


(a) Router on NP1

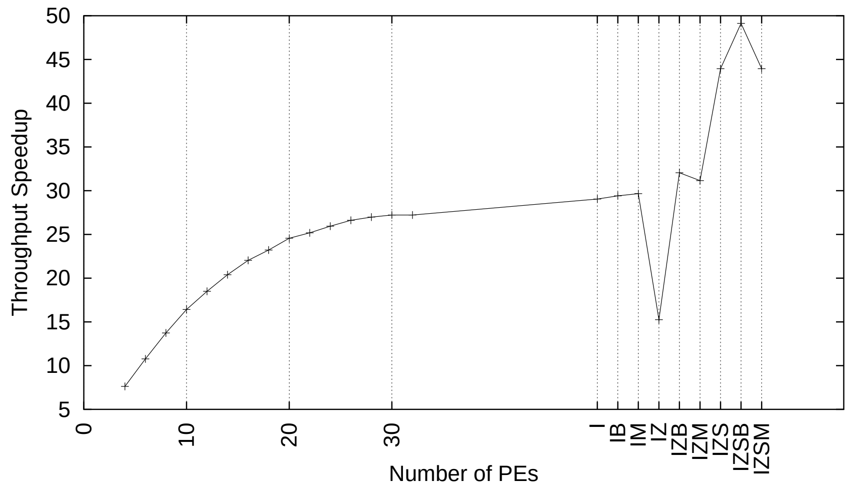


(b) Router on NP2

Figure 6.1: Throughput speedup of Router for several transformations and varying numbers of PEs, relative to the application with no transformation running on 4 PEs. The throughput indicated is a measure of the maximum sustainable input packet rate. `rep` means replication, `rep on PE` means replication where the replicants are limited to execute on a specific PE.



(a) Router on NP1, with global task queue



(b) Router on NP2, with global task queue

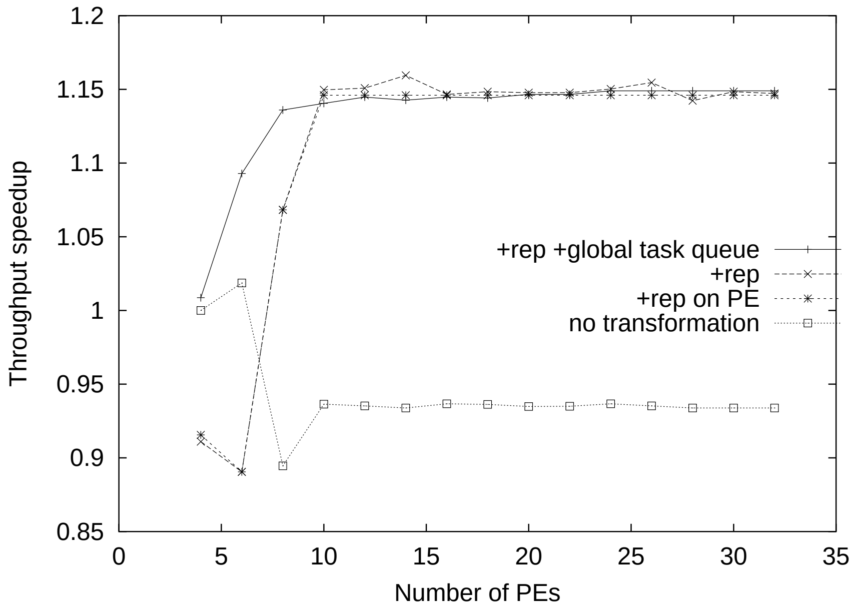
Figure 6.2: Throughput speedup of Router for varying numbers of PEs, relative to the application with no transformation running on 4 PEs. The throughput indicated is a measure of the maximum sustainable input packet rate. Combinations of idealized executions are plotted to the right of the graphs: *I*: infinite number of PE; *B*: maximum bus pipelining; *M*: maximum memory pipelining, i.e. the unpipelined time for a request is 1 cycle; *Z* zero instructions; *S*: no synchronization.

a throughput improving up to 10 PEs. It is evident that for the subset replication, the saturation throughput does not change smoothly with the number of PEs: this is a result of the mapping algorithm, which is discrete in nature. Also in Figure 6.1(a), we can see that subset replication can outperform the global task queue for 14 to 20 PEs. At 16 PEs, subset replication provides 1.5% more throughput than the application with the global task queue. In this configuration, the two replication schemes have the same memory utilization: 63.5% utilization for DRAM and 83.3% utilization for SRAM with the maximum bus utilization being in the SRAM bus with 66%. This relatively high utilization allows us to hypothesize that the SRAM and its bus are limiting the router throughput. At 16 PEs, the global task queue processes 18% more packets in parallel than the subset replication and the lock with the maximum utilization is taken 87.5% of the time. Hence, another possible limiting factor is synchronization. We can verify those assumptions on `Router`'s bottlenecks in Figure 6.2(a). This figure shows simulations with global task queue replication on NP1 having between 4 and 32 PEs. On the right side of the figure, we measure the throughput of NP1 with an infinite number of PEs available. The figure shows throughput improvements when the bus pipelining is maximized, i.e. the unpipelined request time of transactions on all busses is reduced to 1 cycle. This non-realistic parameter allows us to determine the impact of removing constraints on the NP. Indeed, we have verified that when the bus pipelining is maximized, the SRAM utilization reaches close to 100%, thus becoming the next bottleneck. It is logical that the SRAM (bus and memory) accesses dominate the latency of `Router` because this is where the routing element maintains its routing table in the temporary heap. In Figure 6.2(a), it is evident that removing the synchronization does not improve the throughput because the synchronization is only on the critical path for this application after SRAM memory and bus contention are resolved.

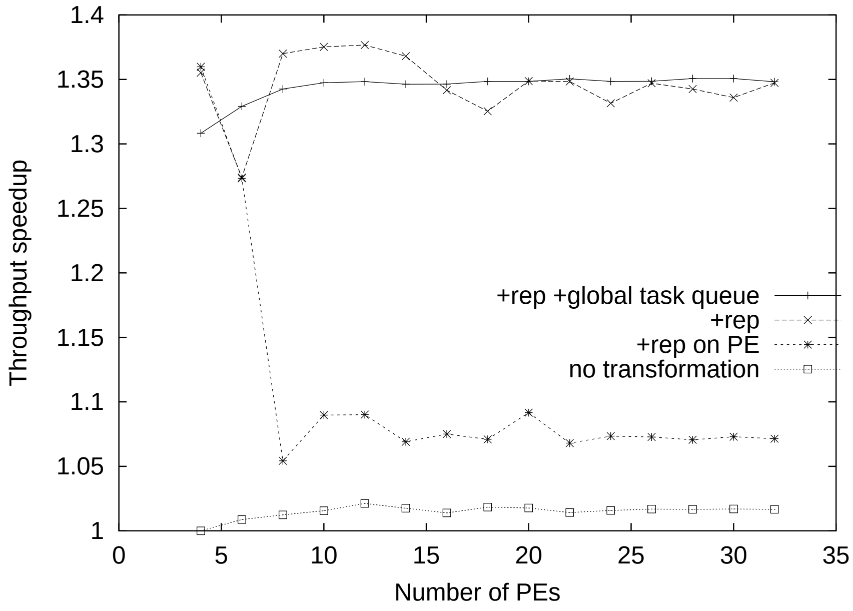
Router on NP2 When running on NP2, as seen in Figure 6.1(b), with the global task queue replication scheme, the `Router` application scales asymptotically up to 30 PEs. The PE computing utilization is progressively reduced because of increasing contention on the external SRAM and DRAM memory and their associated busses. The maximum throughput obtained on 30 PEs is 27.2 times the throughput of the application with no transformation. In Figure 6.1(b), we can see that subset replication cannot improve on the task mapping until the number of PEs approaches the number of active tasks in the application. Subset replication also limits the number of replicas: with 14 PEs, the average number of concurrent packets in the NP is 67% of the average number with the global task queue. We can also observe that our mapping technique does not perform well with 28 and 32 PEs. Figure 6.2(b) with an infinite number of PEs shows clearly that removing synchronization gives the most speedup to the application. In that configuration, the SRAM bus utilization reaches over 80% and the DRAM utilization reaches over 60%. Removing the instructions alone forces memory accesses to be executed one after another, thus putting more pressure on the SRAM and DRAM busses. As seen in Figure 6.2(b), those busses are less important bottleneck sources for the application.

For `Router`, we saw that the bottlenecks were different on NP1 and NP2: respectively the SRAM bus utilization and the synchronization. We observed that replication was effective in taking advantage of the computing power provided by a large number of PEs until architectural bottlenecks limit the application performance.

NAT on NP1 For `NAT` running on NP1, in Figure 6.3(a), we can see that the application without any transformation runs 7% slower with 10 or more PEs than on 4 PEs. An increased contention on synchronization indicates that the added task parallelism is not sufficient to compensate for the added latency on the NP busses and memory units. The global task queue scheme reaches its plateau the fastest and plateaus approximately at the same performance as when replication is limited to a PE and when subset replication is used. `NAT` has several synchronized tasks and the lock with the maximum utilization is taken 63.2% of the time, on average across the replication schemes. The infinite PE graph in Figure 6.4(a) shows that removing the synchronization improves the throughput. As well, computations are also on the critical path since removing computations seems to improve the throughput the most and increases the number of concurrent packets by 7%. However, removing synchronization increases the number of concurrent packets by a factor 3. Hence synchronization is a more significant bottleneck for `NAT` than the computations, which explains the very slow scaling of throughput with the number of PEs.

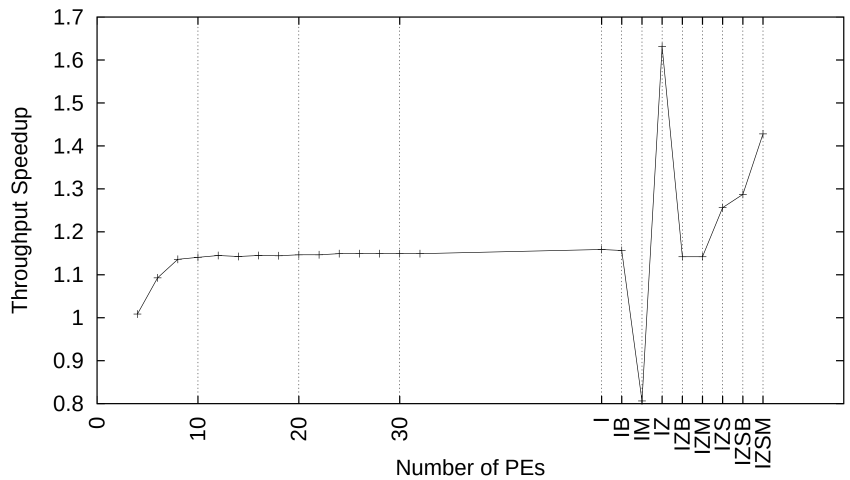


(a) NAT on NP1

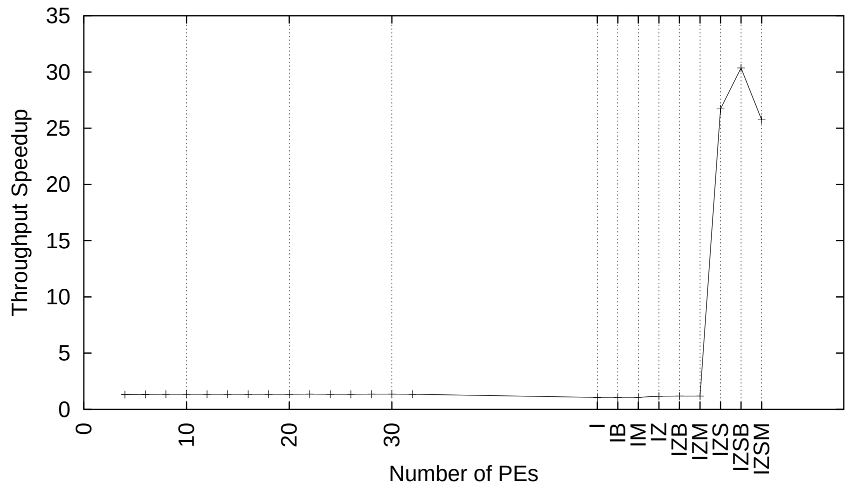


(b) NAT on NP2

Figure 6.3: Throughput speedup of NAT for several transformations and varying numbers of PEs, relative to the application with no transformation running on 4 PEs. The throughput indicated is a measure of the maximum sustainable input packet rate. `rep` means replication, `rep on PE` means replication where the replicants are limited to execute on a specific PE.



(a) NAT on NP1, with global task queue



(b) NAT on NP2, with global task queue

Figure 6.4: Throughput speedup of NAT for varying numbers of PEs, relative to the application with no transformation running on 4 PEs. The throughput indicated is a measure of the maximum sustainable input packet rate. Combinations of idealized executions are plotted to the right of the graphs: *I*: infinite number of PE; *B*: maximum bus pipelining; *M*: maximum memory pipelining, i.e. the unpipelined time for a request is 1 cycle; *Z* zero instructions; *S*: no synchronization.

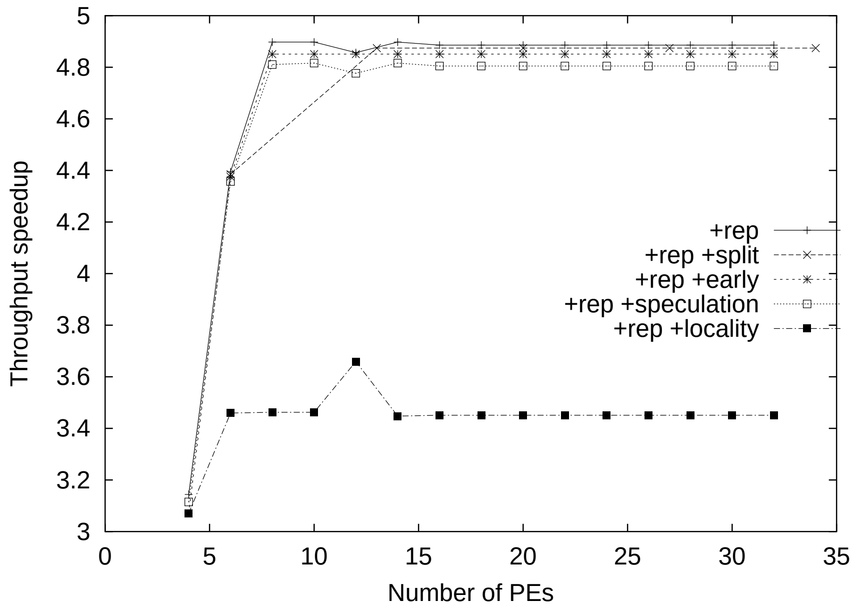
NAT on NP2 On the NP2 processor, in Figure 6.3(b), the curves of replication limited to a PE and subset replication no longer overlap. The performance of `NAT` with the replication limited to a PE decreases and becomes relatively stable with more than 8 PEs. This performance reduction is attributed to the reduction in locality when accessing persistent heap mapped to the local storage in the PEs. Further compiler work could potentially alleviate this problem by replicating read-only data to the PE's local storage. In Figure 6.3(b), the subset replication again outperforms in certain cases the global task queue due to a different task scheduling. In Figure 6.4(b), the performance of `NAT` on the NP2 is increased by a factor ranging from 25 to 30 when synchronization is disabled, thus indicating that synchronization is a significant bottleneck for `NAT`.

Because replication has shown to be especially useful, we use it in conjunction with the other task transformations. To show an upper bound of the transformation benefits, we only present the experiments with the global task queue. For the reader's convenience, we reproduce the global task queue curves with no other transformations on the graphs to serve as a comparison point.

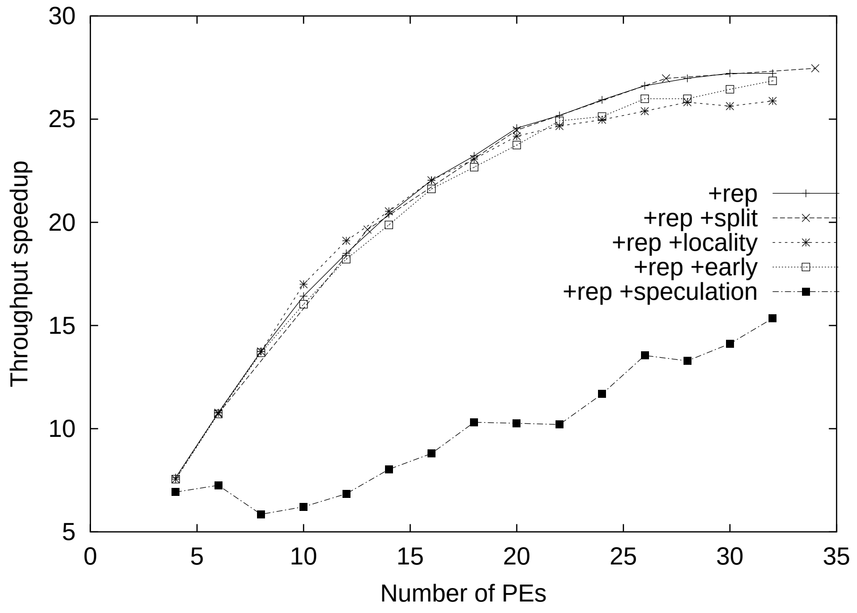
6.3.2.2 Locality Transformation

In this section, we present experiments evaluating the locality transformation that consists of both memory batching and forwarding, as presented in Section 3.2.

Router on NP1 In Figure 6.5(a), on the NP1, we can see that the locality transformation with the global task queue in fact limits I_{max} to a maximum speedup of 3.45 over the application with no transformation. The maximum speedup in that configuration without the locality transformation is 4.89 as seen in the same Figure 6.5(a). Experiments in Figure 6.6(a) show that the locality transformation improves the throughput of the application without replication by 20% starting at 8 PEs. Replication limited to a PE benefits from the locality transformation by a 0.8% throughput increase. This leads us to conclude that the locality transformation has diminishing returns when there is more computation to overlap with memory accesses. Indeed, the locality transformation sends bursts of requests on the memory busses at the start of tasks, thus temporarily increasing the congestion on the busses.

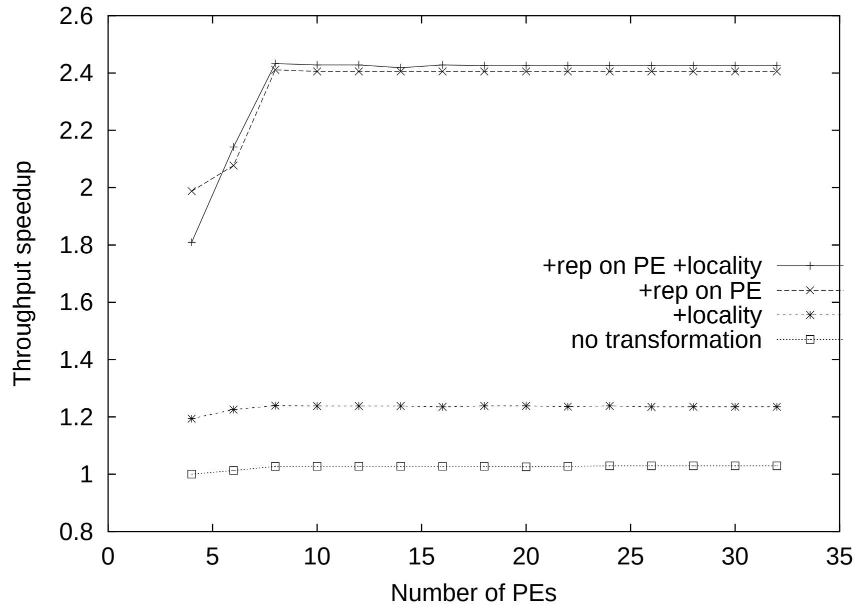


(a) Router on NP1 with a global task queue

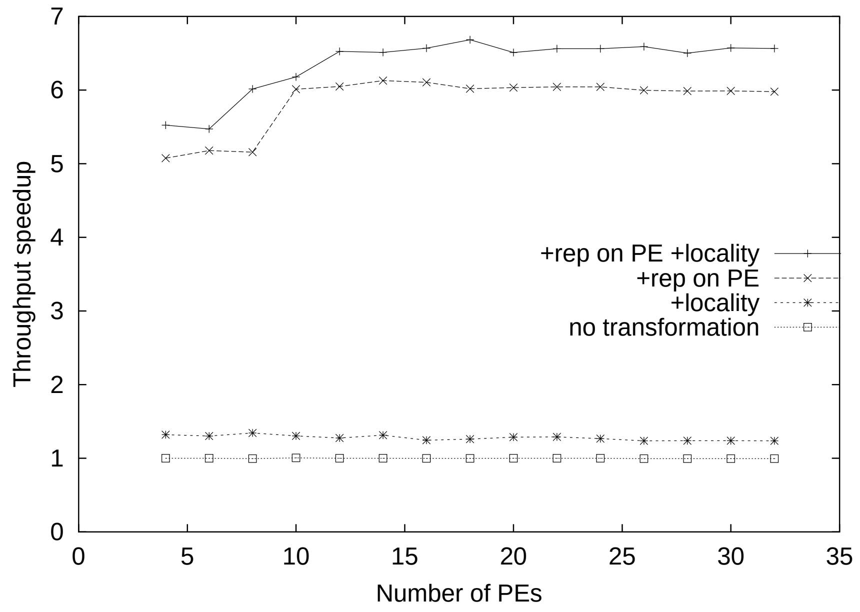


(b) Router on NP2 with a global task queue

Figure 6.5: Throughput speedup of Router for several transformations and varying numbers of PEs, relative to the application with no transformation running on 4 PEs. The throughput indicated is a measure of the maximum sustainable input packet rate. `rep` means replication. `locality`, `early`, `split` and `speculation` refer to the locality, early signaling, task splitting and speculation transformations.



(a) Router on NP1



(b) Router on NP2

Figure 6.6: Throughput speedup of Router for several transformations and varying numbers of PEs, relative to the application with no transformation running on 4 PEs. The throughput indicated is a measure of the maximum sustainable input packet rate. `rep on PE` means replication where the replicants are limited to execute on a specific PE, `locality` refers to the locality transformations.

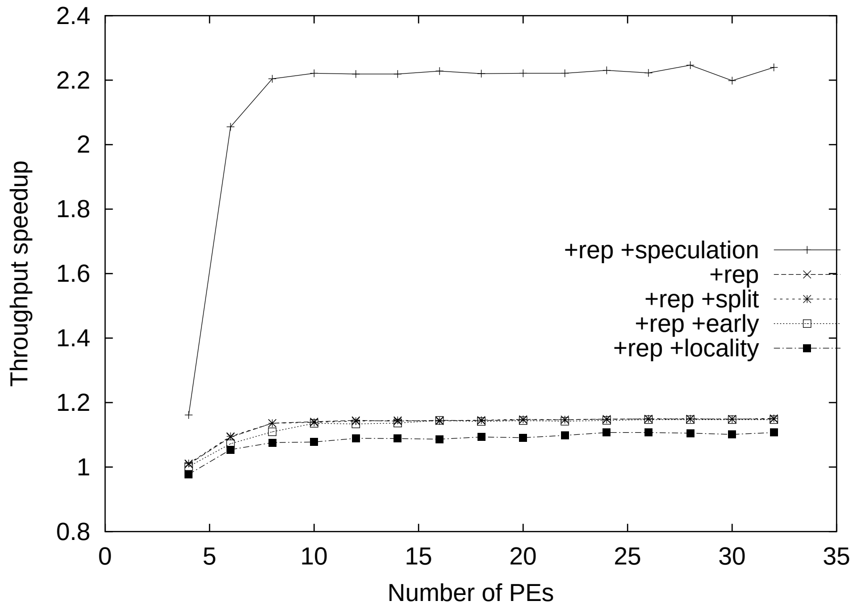
Router on NP2 As seen in Figure 6.5(b), the NP2 is able to accommodate the traffic burstiness of the locality transformation and improve `Router`'s throughput when the number of PE is between 10 and 20. The maximum throughput increase is by 3% with 12 PEs. The average fraction of the time a PE waits for memory is reduced from 1.16% to 0.59% while the DRAM utilization is reduced by 5%. However, the SRAM read busses and the shared on-chip bus utilization are increased by 10%, and the DRAM read busses utilization is increased by 3%. This shows that the burstiness of the locality transform degrades the performance of the busses more than the external memory units. The explanation for this difference is that there is less pipelining in the interconnect than in the external memory units, as shown in Table 6.3. On NP2, Figure 6.6(b) reports increased performance due to the locality transformation with no and PE-limited replication of respectively 23 and 9%. This performance increase is attributed to a reduction in the number of accesses required to external memory.

NAT on NP1 and NP2 For `NAT` on the NP1, shown in Figure 6.7(a), the locality transformation reduces the maximum throughput by almost 4%. On NP2, in Figure 6.7(b), the throughput is decreased at 12 PEs because the parallelism does not compensate for the extra contention brought by the task transformation. This extra contention leads to an increase from 3% to 65% in the fraction of the time spent in the most utilized critical section.

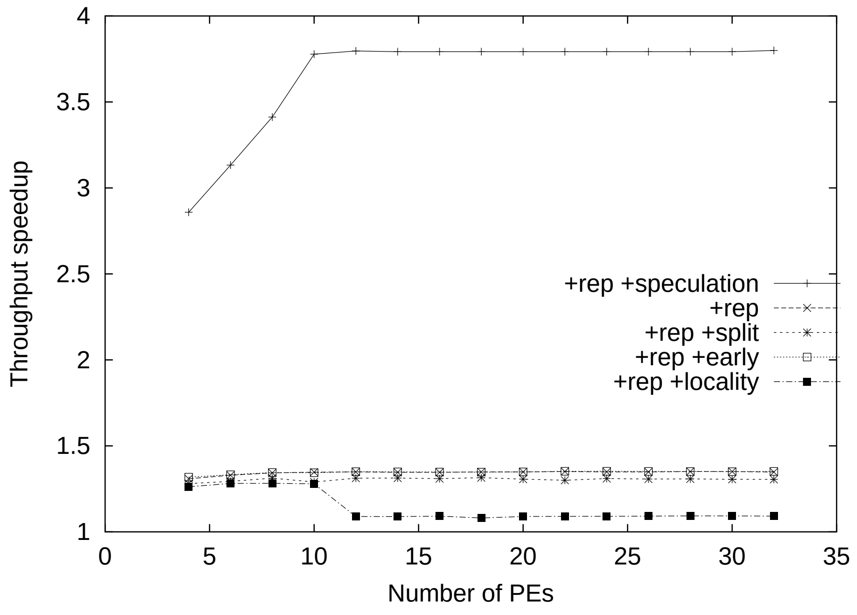
6.3.2.3 Early Signaling

When performing the early signaling task transformation on our payload processing applications, all the possible cases of early signaling complied with the announce/wait_for/resume system presented in Section 3.4.3. `Router` was able to signal early several small tasks, while `NAT` could only signal early a few tasks of average size.

Router on NP1 and NP2 For `Router` on NP1, in Figure 6.5(a), signaling tasks early limits the maximum speedup to 4.85, versus 4.88 obtained with the global task queue alone. Nonetheless, we observed a reduction in the packet processing latency of 42%, averaged over all PE configurations. The slight I_{max} reduction can be explained by the negative impact of context eviction when tasks



(a) NAT on NP1 with a global task queue



(b) NAT on NP2 with a global task queue

Figure 6.7: Throughput speedup of NAT for several transformations and varying numbers of PEs, relative to the application with no transformation running on 4 PEs. The throughput indicated is a measure of the maximum sustainable input packet rate. `rep` means replication. `locality`, `early`, `split` and `speculation` refer to the locality, early signaling, task splitting and speculation transformations.

need to wait for other early signaled tasks. Also, `Router`'s high contention on the SRAM memory unit prevents I_{max} improvements, as explained in Section 6.3.2.1. We realized that the early signaled tasks had very low SRAM access requirements, indicating that those tasks could execute without interference on idle contexts of the NP. Also, we observed that the early signaled tasks were mostly located in the last stages of the processing, thus not allowing tasks with high SRAM demands to execute according to a different schedule than without early signaling. Consequently, in `Router`, because the packet processing latency is greater than the packet inter-arrival time, reducing the packet processing latency does not necessarily improve the application throughput. On NP2, early signaling decreased the maximum throughput of `Router` by 1%, as showed in Figure 6.5(b). However, for the same reasons as mentioned for NP1, we observed a reduction in the packet processing latency of 8%. This smaller latency improvement shows that the early signaled tasks, because of their low usage of SRAM memory, account for a less important fraction of the processing on NP2 than on NP1. As explained in Section 6.2, the NP2 has processing elements proportionally faster than the memory compared to the NP1.

NAT on NP1 and NP2 With `NAT` on NP1 showed in Figure 6.7(a), the maximum throughput achieved with early signaling is decreased by 1% while on NP2 in Figure 6.7(b), the maximum throughput is increased by 0.1%. For `NAT`, we did not see any significant packet processing latency improvement. Consequently, this application is unable to overlap a significant amount of processing without any dependence with other tasks of the application.

6.3.2.4 Speculation

Speculation involves optimistically letting tasks cooperate in their dependences as presented in Section 3.4.4. We next describe the impact on `Router` and `NAT` of this transformation that also requires hardware support to detect dependence violations.

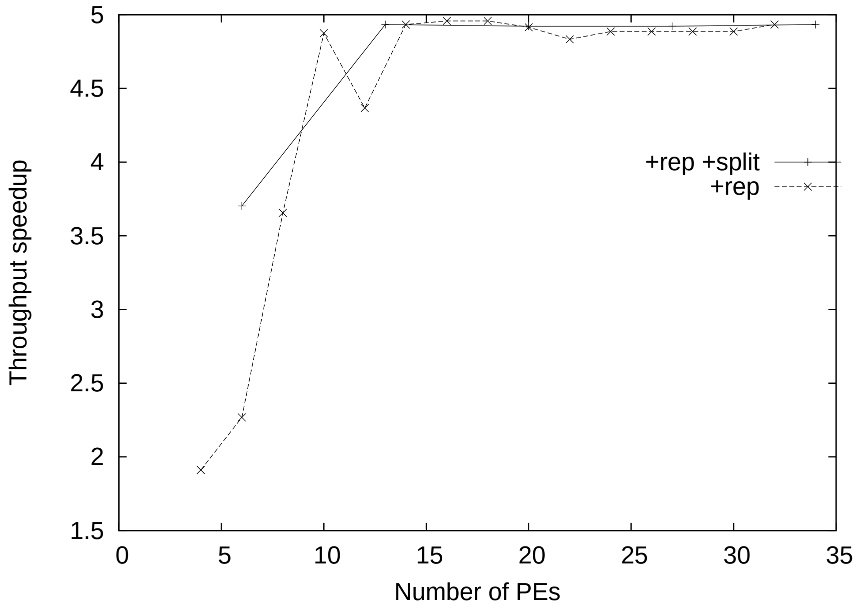
Router on NP1 We evaluated the impact of using speculation for our header processing applications. For `Router` on NP1 (Figure 6.5(a)), speculation has a negative impact on `Router` because synchronization is not on the critical path for this application and the local buffering along with the committing of the speculative writes (as presented in Section 3.4.4) simply adds overheads.

Router on NP2 The impact of the transformations is more pronounced when `Router` is executed on the NP2. We can see that speculation has a negative impact although it does not prevent scaling. For all PE configurations in Figure 6.5(b), the worse performance is obtained for 8 PEs. In this configuration, the *violation rate*, i.e. the number of total violations over the number of synchronized task executions, is of 5%. With 32 PEs, the violation rate is 6.5%. This low violation rate indicates that speculation incurs significant re-execution overheads for `Router`. The non-smooth scaling of the performance of `Router` on NP2 is symptomatic of a less deterministic processing time per packet.

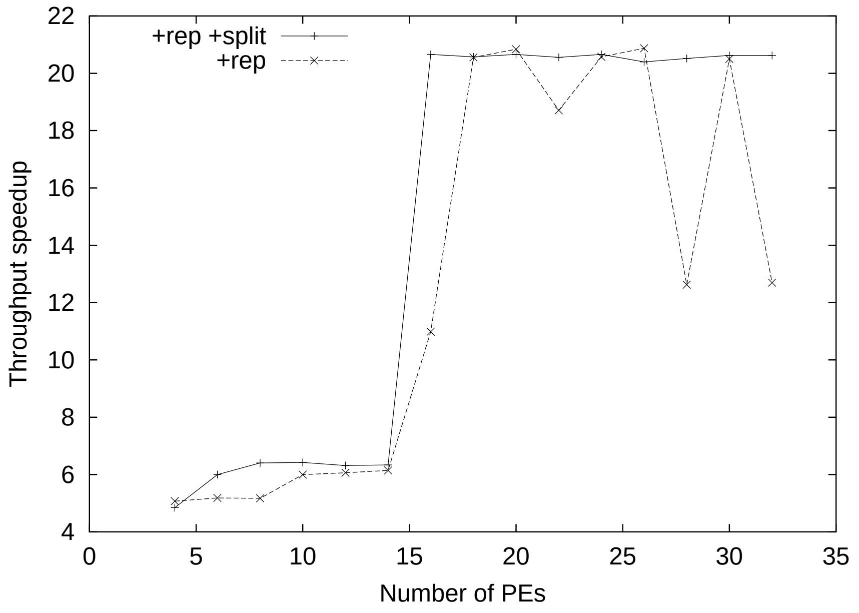
NAT on NP1 and NP2 For `NAT` on NP1 in Figure 6.7(b), speculation improves the maximum throughput by 96% over the global task queue alone. Hence, speculation allows `NAT` to execute synchronized tasks without dependence violations in the common case. On the NP2, speculation allows I_{max} to scale slowly up to 30 PEs, in which case `NAT` has a throughput 183% higher than the global task queue alone. With 12 PEs, we observed that the violation rate was 1.8%; with 32 PEs, this rate was increased to 2.2%. In consequence, we can see that a greater supply of PEs can compensate for the re-execution penalties of an increased violation rate.

6.3.2.5 Task Splitting

As it can be seen on Figures 6.5(a), 6.5(b), 6.7(a) and 6.7(b), task splitting (first introduced in Section 3.4.2) does not significantly impact the throughput of `Router` and `NAT` on NP1 and NP2. In fact, the communication overhead between the task splits lowers the throughput of `NAT` on NP2 by 3% (Figure 6.7(b)). Splitting does not increase the parallelism inside an application because the task splits execute in sequence and have the same mapping, i.e. assignments to PEs, as the unsplit task. The impact of splitting can best be seen on the scheduling of tasks when replication is limited. In Figure 6.8, splitting with subset replication improves the throughput with a small number of PEs with `Router` on NP2 (Figure 6.8(b)) and achieves a better load balance because of



(a) Router on NP1



(b) Router on NP2

Figure 6.8: Throughput speedup of Router for several transformations and varying numbers of PEs, relative to the application with no transformation running on 4 PEs. The throughput indicated is a measure of the maximum sustainable input packet rate. `rep` means replication, `split` refers to the task splitting transformation.

the rescheduling of the splits. We can see that splitting also corrects mapping problems by leveling the throughput for more than 16 PEs in the same figure. Similar benefits are observed for `Router` on NP1 in Figure 6.8(a). Consequently, splitting is effective in load-balancing the utilization of PEs when tasks are not free to execute on any PE.

6.4 Summary

In this section, we presented how our infrastructure can be used to characterize benchmarks and identify what hardware support and compiler techniques are best to exploit the resources of a network processor. `Router` was found to be performance limited by its SRAM accesses while for `NAT`, synchronization is the bottleneck factor. For this reason, we found that our transformations had different impacts on these benchmarks. Replication was found to be able to extract the most parallelism. Our locality transformation increases the utilization of the NP memory busses but can improve the NP performance when the number of concurrent tasks is low. Early signaling was found to improve the throughput marginally or not at all. However, early signaling significantly improved the packet processing latency of `Router`. Speculation was found to be helpful for `NAT` by removing its synchronization bottleneck. Finally, splitting helps load balancing tasks with little replication by breaking tasks into multiple re-schedulable splits. Splitting gives the most speedups when the amount of task replication is small. Replication can be limited by the number of hardware contexts, the task scheduling overheads or the PEs' instruction store capacity.

Our experiments have shown that the NPIRE simulator helps uncover architectural bottlenecks by giving numerous system statistics for the user to analyze. Furthermore, we observed that an ad hoc schedule of memory accesses and computations between threads leads to an imperfect overlap of latencies with computations. For this reason, individual utilization metrics do not always reveal the system bottleneck; the performance limiting factor is often best found when measuring the benefits of tentatively removing potential bottleneck factors. In conclusion, NPIRE provides a suite of powerful tools to build a feedback-driven compilation infrastructure for network processors.

7 Conclusions and Future Work

We have presented NPIRE: a simulation infrastructure that compiles a high level description of packet processing, based on Click, and transforms it to produce a form suitable for code generation for a network processor. In addition, we presented a programming model based on buffer type identification and separation that allows the compiler to insert synchronization and perform several task transformations to increase parallelism. A list of high-level topics that were studied using our simulator include:

1. mapping tasks to processing engines;
2. task transformations to achieve pipelining inside an application;
3. memory organization;
4. scheduling multiple threads;
5. signaling and synchronization strategies;
6. allocation of resources (in particular, data layout).

NPIRE provides compiler support to transform an application. Using execution feedback, our infrastructure can compile an application such that its memory access patterns are closer to that of a finely tuned application. In particular, our support for the combination of the following memory operations makes our infrastructure more realistic over other related NP studies:

1. memory typing and simulation of a memory hierarchy matching the memory typing;
2. improved on-chip communication;
3. non-blocking memory operations;
4. automated synchronization.

To evaluate and compare different task transformation and mapping techniques and their ability to effectively scale to many PEs, we devised a method for finding the maximum sustainable packet input rate of an NP. We selected full-featured network processing applications and measured their throughput using modern realistic NP architectural parameters. Our analysis extends to a range of NPs with different ratios of processing versus memory throughput.

Of the automatic compiler transformations proposed, we demonstrated that replication was the task transformation allowing to extract the most parallelism out of an application. Early signaling was found to help reduce the packet processing latency while splitting was able to load balance tasks with low replication. Our locality transformations were able to improve the throughput when the on-chip communication channels were not the bottleneck. Finally, speculation reported dramatic throughput improvements when the amount of synchronization in the application was important and the amount of violation dependences low. We showed that transformation pairs such as replication/locality transformations and replication/splitting are complementary and allow the application to scale to a greater number of processing elements, resulting in packet throughput that is very close or exceeds the idealized global task scheduling.

Today, requirements for packet processing range from bare routing to the interpretation of packets at the application layer. Programming network processors is complex because of their high level of concurrency. With NPIRE we have shown that the programmer can specify a simple task graph, and that a compiler can automatically transform the tasks to scale up to the many PEs and hardware contexts available in a modern network processor.

7.1 Contributions

This dissertation makes the following contributions: (i) it presents the NPIRE infrastructure, an integrated environment for network processor research; (ii) it describes network processing task transformations and compilation techniques to automatically scale the throughput of an application to the underlying NP architecture; (iii) it presents an integrated evaluation of parallel applications and multi-PE NPs.

7.2 Future Work

Similarly to other work [50], we have shown that there remains idle periods of time in our task schedule and that the speedups due to our transformations are far from matching the investment in the number of PEs, mostly because of contention on SRAM busses and memory channels. For this reason, we present further task transformations that could improve the packet processing throughput. Next we present features of our simulation that could be improved upon.

7.2.1 Further Task Transformations

We now present transformations that are not yet (fully) implemented but offer potential throughput improvements for an application automatically mapped to a network processor.

Task Specialization Because of Click's modularity, tasks may perform more work than is desired for a specific application. For example, it is possible that a classification engine removes the need for another element downstream in the task graph to consider a certain type of packets. It is hence possible that large portions of the tasks are revealed dead (unused) code that could be eliminated. This elimination could lead to savings in instruction space and further code optimizations. One common example of code specialization is to replace some variables by their observed run time constant value.

Head of Line Processing This transformation assumes that we can build different specialized versions of tasks that individually process faster different kinds of packets. In order for the packets to reach the specialized task, we need a way to determine what characterizes the packets that can benefit from more efficient processing. The earlier we get that information, the earlier the packet can be handled by a specially tuned task. The approach that we use is to find points in the task graph where there are branches. We then look at the code in the basic blocks that create a transition between elements. From there, we analyze the conditional statements. The next step would be to evaluate early those conditions. Slicing the condition code and bubbling it upwards the task graph poses some significant challenges.

Out-of-band Tasks In our infrastructure, computations start with the arrival of a work unit (defined in Section 5.1.2), consisting of a packet and a task identifier. We could extend our simulator to support tasks that are timer triggered or run continuously to do maintenance.

Re-Partitioning Task repartitioning involves moving task boundaries between consecutive tasks. Ennals et al. [23] show that this can be achieved by successive task splitting and merging (the authors refer to those transformations respectively as “PipeIntro” and “PipeElimin”).

Further Speculation In this work, we presented speculation to enter a synchronized section without waiting for all other tasks to have exited it. Another form of speculation that we could explore would be to start elements even before they are guaranteed to execute.

Intra-Task Pipelining We could attempt at parallelizing the task splits defined in Section 3.4.2. Hence, each task split would wait for a synchronization message and would not need to wait for its predecessor splits to complete.

7.2.2 Improving Simulation

Here is a non exhaustive list of features that can be improved/added in the infrastructure.

7.2.2.1 Improving Mapping

As explained in Section 5.2.2, we wrote a *fast simulator* to be able to test a large number of mappings. It is fast because it only simulates scheduling tasks of the duration measured in the simulator with no context switching and no architectural simulation. We used this fast simulator to do extensive searches of mappings. Because the number of possible mappings gets very large with respect to the number of tasks and PEs, we introduced the concept of seeding an initial mapping to the fast simulator. In that case, our mapping tools only has to place the remaining tasks. To be able to trust the results of the fast simulator, we compared the throughput of the fast and the real simulators. We found that the throughput trends were similar between the two simulators and were especially close when contention on bus, memory or synchronized resources was not a performance bottleneck in the detailed simulator. We envision that better mapping results could be attained by improving the accuracy of the fast simulator.

7.2.2.2 Improving Other Simulation Aspects

Here is a list of approaches that could make our simulation even more realistic:

- Introduce a micro-architectural simulation of the processing elements. Different flavors of instruction -level parallelism could be examined like in Seamans and Rosenblum's work [74].
- Model with more accuracy memory allocation of packet memory and temporary heap. This memory allocation must be supported for our compiler to generate executable code.
- Implement our techniques on a real NP or an FPGA fabric.

A Appendix

In this chapter, we present some simulator features that make of it a powerful tool suited for further system research. We then motivate the current organization of the NPIRE by giving some information on the task mapping techniques that were tried and on how our infrastructure was iteratively built.

A.1 Simulator features

The goal of the simulator is to mimic the execution of our application's recorded trace on a parametric network processor. The simulator allows to see the performance impacts of the modifications that we make to the application and to the architecture of the simulated NP.

Our infrastructure has numerous scripts that automate simulations and the generation of traces and simulator configurations. The simulator is also equiped with multiple scripts that make it easy to change between benchmark environments very quickly. Consequently, the NPIRE simulator can be deployed and installed rapidly on x86-class machines. Multiple simulations in parallel are supported: we even ported our simulator to a Condor [82] cluster.

A few data sets collected by the simulator are best represented graphically. The NPIRE simulator can generate a plot of an application task graph with the edges labeled with their usage count and a graph of the element mapping. The simulator user can display and save as a picture file a colored map of memory references to a buffer, or a memory type, where the color corresponds to the frequency of the accesses. In the simulator, a large number of events occur concurrently. To give a global view of the NP activity to the user, the simulator can produce, for a limited time interval, a diagram showing context switching, task signaling and processing elements state. For example, this diagram allows the user to see graphically where PEs are waiting because of contention on shared resources. The simulator can also produce coarser graphs of tasks execution in time and packet processing in time. This customizable granularity of graphs allows for easier debugging. Finally, at the end of each run, the simulator prints a block diagram of the network processor simulated, annotated with the most important rate and utilization statistics.

The simulator has support for interactive debugging by being able to report the simulated time and its full state at any time. The simulator can also print a file containing the size of all element queues in order to follow the evolution of a potential congestion in the control graph. All statistics in the simulator are connected to a global reset allowing to start and end measurements at any time. Some statistics can be set to be periodically re-normalized at runtime to account for fluctuations in the processing.

To assist the compiler in identifying frequently executed code, the simulator dynamically builds a suffix tree of sequences of elements executed on a packet. This data structure was found to be complex to build considering that multiple packets can be in flight and appending/branching in the suffix tree. We envision that this monitoring could be used in the future to provide some simulator responsiveness to changes in packet flow patterns.

A.2 Mapping

We implemented several strategies for task assignment to processing elements that are still included in our infrastructure. The following techniques are not used in the results presented in this document because they provide inferior mapping results on average to the technique presented in Section 5.2.2. The quality of a mapping can be measured in terms of load balance between processing elements utilization and overall system throughput.

One-to-one Tasks are assigned in a round-robin fashion to the available processing engines.

Theoretical This mapping scheme uses a statistical (mathematical) model of queue lengths. This technique computes the probability of packet loss according to the task latencies and frequency of occurrence.

Avoidance based This algorithm compacts elements on the smallest number of PE. The way we proceed is by recording a window of operation in which all used elements are given a chance to execute. During this execution, we record all PE activity periods in a mapping of one task per PE. We then determine which elements were active at the same time and conclude that they cannot be mapped on the same PE. We later realized this task compaction on PEs is incorrect. Making concurrent tasks execute sequentially only hurts performance if this reordering delays the execution of other tasks on a given PE.

Limit bin packing This algorithm requires a user-defined target number of PEs. Each element instance has to be assigned at least one PE. For this first placement, we add tasks to a PE as long as the utilization of the PE does not exceed 100%. At each step, our greedy algorithm selects the PE on which, when the task is added, has the most remaining idleness (i.e. headroom). We also have the option to consider placing the most active tasks first. If there is still room in the PEs (determined by the sum of activity), we replicate the elements that have a waiting time that is over the average waiting time.

A.3 Early versions of NPIRE

The current NPIRE design is in its third version. In this section, we briefly explain why the earlier versions had to be modified to motivate the current compiler/simulator organization of our infrastructure.

Our first attempt at creating the simulator was using Augmint [62] to instrument all memory reads and writes inside the disassembled application code. We used a call graph generated by Doxygen [86] to select the functions to be instrumented. Multiple software threads were declared in our back-end (presented in Section 4.2.3) and we would execute a tasks on packets on threads taken from the thread pool. In that case, our network processor simulation was done concurrently with the Click router execution. This imposed several limitations on the parallelization/reordering techniques we could use.

We then evolved the simulator to generating a trace still using Augmint and an instruction count obtained by converting the Click code to micro-ops using Bochs [7] coupled with rePlay [78]. Using an application trace turned out to be more flexible from the simulation point of view but the binding between the application and the architecture was inexistent. With the compiler support that we inserted in the current version of the infrastructure, we have some knowledge of what memory references point to and we have more powerful ways of achieving custom partitioning, instrumentation, scheduling and resource allocation.

In the early versions of the simulator, we selected the RLDRAM II [58] [33] as the technology for our first memory device model implementation. On some RLDRAM devices, since the write operation has a shorter latency than the read, data on the data bus can be reordered (if this doesn't incur any violations). We added a small controller to the RLDRAM code to try and improve on memory transactions batching by looking at the requests queued. We attempted to eliminate redundant accesses and merging corresponding write/read accesses. Those optimizations had very little returns. Timing of an RLDRAM access is a non trivial problem because we need to account for the off-chip transition as well as the different clock frequency with respect to the PEs. For example, in the IXP NPs [35], the latency of a memory access can be 10 to 100 longer in PE cycles that the number of memory cycles for the operation, although the clock frequencies differ by a factor of roughly 2. This motivates our current memory timing model presented in Section 5.1.5.

Bibliography

[1] ADVE, V., LATTNER, C., BRUKMAN, M., SHUKLA, A., AND GAEKE, B. LLVA: A Low-level Virtual Instruction Set Architecture. In *Proceedings of the 36th annual ACM/IEEE International Symposium on Microarchitecture (MICRO-36)* (San Diego, California, Dec 2003).

[2] ADVE, V., AND SAKELLARIOU, R. Application representations for multi-paradigm performance modeling of large-scale parallel scientific codes. In *International Journal of High-Performance and Scientific Applications* (2000), vol. 14, pp. 304–316.

[3] ALLEN, J., BASS, B., BASSO, C., BOIVIE, R., CALVIGNAC, J., DAVIS, G., FRELECHOUX, L., HEDDES, M., HERKERSDORF, A., KIND, A., LOGAN, J., PEYRAVIAN, M., RINALDI, M., SABHIKHI, R., SIEGEL, M., AND WALDVOGEL, M. PowerNP network processor hardware software and applications. *IBM Journal of Research and Development 47*, 2 (2003).

[4] AUSTIN, T. SimpleScalar LLC. http://www.simplescalar.com/.

[5] BIEBERICH, M. Service providers define requirements for next-generation IP/MPLS core routers. The Yankee Group Report, April 2004.

[6] BLUMOFE, R. D., JOERG, C. F., KUSZMAUL, B., LEISERSON, C. E., RANDALL, K. H., , AND ZHOU, Y. Cilk: An efficient multithreaded runtime system. In *5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming PPOPP '95* (Santa Barbara, California, July 1995), pp. 207–216.

[7] BUTLER, T. R. The open source IA-32 emulation project. http://bochs.sourceforge.net/, 2005.

[8] C-PORT. *C-5 Network Processor Architecture Guide*, c-5 np d0 release ed., 2002.

[9] CAMPBELL, A. T., CHOU, S. T., KOUNAVIS, M. E., AND STACHTOS, V. D. NetBind: A binding tool for constructing data paths in network processor-based routers, 2002.

[10] CARR, S., AND SWEANY, P. Automatic data partitioning for the Agere Payload Plus network processor. In *ACM/IEEE 2004 International Conference on Compilers, Architecture and Synthesis for Embedded Systems* (2004).

[11] CHEN, B., AND MORRIS, R. Flexible control of parallelism in a multiprocessor PC router. In *2001 USENIX Annual Technical Conference (USENIX '01)* (Boston, Massachusetts, June 2001).

[12] CHEN, K., CHAN, S., JU, R. D.-C., AND TU, P. Optimizing structures in object oriented programs. In *9th Annual Workshop on Interaction between Compilers and Computer Architectures (INTERACT'05)* (San Francisco, USA, February 2005), pp. 94–103.

[13] CISCO SYSTEMS. The Cisco CRS-1 carrier routing system. http://www.cisco.com, May 2004.

[14] CROWLEY, P., AND BAER, J.-L. A modeling framework for network processor systems. *Network Processor Design : Issues and Practices 1* (2002).

[15] CROWLEY, P., FIUCZYNSKI, M., BAER, J.-L., , AND BERSHAD, B. Characterizing processor architectures for programmable network interfaces. *Proceedings of the 2000 International Conference on Supercomputing* (May 2000).

[16] CROWLEY, P., FIUCZYNSKI, M., BAER, J.-L., AND BERSHAD, B. Workloads for programmable network interfaces. *IEEE 2nd Annual Workshop on Workload Characterization* (October 1999).

[17] DAVIS, J. D., LAUDON, J., AND OLUKOTUN, K. Maximizing CMP throughput with mediocre cores. In *PACT '05: Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques (PACT'05)* (Washington, DC, USA, 2005), IEEE Computer Society, pp. 51–62.

[18] DECASPER, D., DITTIA, Z., PARULKAR, G., AND PLATTNER, B. Router plugins: a software architecture for next-generation routers. *IEEE/ACM Trans. Netw. 8*, 1 (2000), 2–15.

[19] DITTMANN, G., AND HERKERSDORF, A. Network processor load balancing for high-speed links. In *International Symposium on Performance Evaluation of Computer and Telecommunication Systems (SPECTS)* (San Diego, California, July 2002), pp. pp. 727–735.

[20] EHLIAR, A., AND LIU, D. Benchmarking network processors. In *Swedish System-on-Chip Conference (SSoCC)* (Bâstad, Sweden, 2004).

[21] EL-HAJ-MAHMOUD, A., AND ROTENBERG, E. Safely exploiting multithreaded processors to tolerate memory latency in real-time systems. In *2004 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES'04)* (September 2004), pp. 2–13.

[22] ENNALS, R., SHARP, R., AND MYCROFT, A. Linear types for packet processing. In *European Symposium on Programming (ESOP)* (2004).

[23] ENNALS, R., SHARP, R., AND MYCROFT, A. Task partitioning for multi-core network processors. In *International Conference on Compiler Construction (CC)* (2005).

[24] GAY, D., AND STEENSGAARD, B. Fast escape analysis and stack allocation for object-based programs. In *9th International Conference on Compiler Construction (CC'2000)* (2000), vol. 1781, Springer-Verlag.

[25] GEORGE, L., AND BLUME, M. Taming the IXP network processor. In *PLDI'03, ACM SIGPLAN Conference on Programming Language Design and Implementation* (2003).

[26] GOGLIN, S., HOOPER, D., KUMAR, A., AND YAVATKAR, R. Advanced software framework, tools, and languages for the IXP family. *Intel Technology Journal 7*, 4 (November 2003).

[27] GRÜNEWALD, M., NIEMANN, J.-C., PORRMANN, M., , AND RÜCKERT, U. A framework for design space exploration of resource efficient network processing on multiprocessor socs. In *Proceedings of the 3rd Workshop on Network Processors & Applications* (2004).

[28] HALL, J. Data communications milestones. http://telecom.tbi.net/history1.html, October 2004.

[29] HANDLEY, M., HODSON, O., AND KOHLER, E. XORP: An open platform for network research. *Proceedings of HotNets-I Workshop* (October 2002).

[30] HASAN, J., CHANDRA, S., AND VIJAYKUMAR, T. N. Efficient use of memory bandwidth to improve network processor throughput. *Proceedings of the 30th annual international symposium on Computer architecture* (June 2003).

[31] HENRIKSSON, T. *Intra-Packet Data-Flow Protocol Processor*. PhD thesis, Linköpings Universitet, 2003.

[32] HIND, M., AND PIOLI, A. Which pointer analysis should I use? In *ISSTA '00: Proceedings of the 2000 ACM SIGSOFT international symposium on Software testing and analysis* (New York, NY, USA, 2000), ACM Press, pp. 113–123.

[33] INFINEON TECHNOLOGIES AG. Infineon technologies announces availability of 256-mbit reduced latency dram. http://www.infineon.com, 2002.

[34] INTEL. Intel IXP2800 Network Processor Optimizing RDRAM Performance: Analysis and Recommendations Application Note, August 2004.

[35] INTEL CORPORATION. *IXP2800 Network Processor: Hardware Reference Manual*, July 2005.

[36] JAKOB CARLSTRÖM, T. B. Synchronous dataflow architecture for network processors. *IEEE Micro 24*, 5 (September/October 2004), 10–18.

[37] JEANNOT, E., KNUTSSON, B., AND BJORKMANN, M. Adaptive online data compression. In *IEEE High Performance Distributed Computing (HPDC'11)* (Edinburgh, Scotland, July 2002).

[38] KARIM, F., MELLAN, A., NGUYEN, A., AYDONAT, U., AND ABDELRAHMAN, T. A multilevel computing architecture for multimedia applications. *IEEE Micro 24*, 3 (May/June 2004), 55–66.

[39] KARIM, F., NGUYEN, A., DEY, S., AND RAO, R. On-chip communication architecture for OC-768 network processors. *Proceedings of the 38th conference on Design automation* (June 2001).

[40] KARLIN, S., AND PETERSON, L. VERA: An extensible router architecture. *Computer Networks 38*, 3 (February 2002), 277–293.

[41] KOHLER, E., MORRIS, R., CHEN, B., JANNOTTI, J., AND KAASHOEK, M. F. The Click modular router. *ACM Transactions on Computer Systems 18*, 3 (August 2000), 263–297.

[42] KREIBICH, C. Libnetdude. http://netdude.sourceforge.net/doco/libnetdude/.

[43] KROFT, D. Lockup-free instruction fetch/prefetch cache organization. In *Proceedings of the 8th Annual International Symposium on Computer Architecture* (1981), pp. 81–85.

[44] KULKARNI, C., GRIES, M., SAUER, C., AND KEUTZER, K. Programming challenges in network processor deployment. *Int. Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)* (October 2003).

[45] KWOK, Y.-K., AND AHMAD, I. Static scheduling algorithms for allocating directed task graphs to multiprocessors. *ACM Comput. Surv. 31*, 4 (1999), 406–471.

[46] LATTNER, C., AND ADVE, V. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)* (Palo Alto, California, March 2004).

[47] LBNL'S NETWORK RESEARCH GROUP. Tcpdump and libpcap. http://www.tcpdump.org/.

[48] LEE, B. K., AND JOHN, L. K. NpBench: A benchmark suite for control plane and data plane applications for network processors. In *21st International Conference on Computer Design* (San Jose, California, October 2001).

[49] LIU, H. A trace driven study of packet level parallelism. In *International Conference on Communications (ICC)* (New York, NY, 2002).

[50] LUO, Y., YANG, J., BHUYAN, L., AND ZHAO, L. NePSim: A network processor simulator with power evaluation framework. *IEEE Micro Special Issue on Network Processors for Future High-End Systems and Applications* (Sept/Oct 2004).

[51] LUO, Y., YU, J., YANG, J., AND BHUYAN, L. Low power network processor design using clock gating. In *IEEE/ACM Design Automation Conference (DAC)* (Ahaheim, California, June 2005).

[52] MANNING, M. Growth of the internet. http://www.unc.edu/depts/jomc/academics/dri/evolution/net.html, August 1997.

[53] MELVIN, S., AND PATT, Y. Handling of packet dependencies: A critical issue for highly parallel network processors. In *International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)* (Grenoble, France, October 2002).

[54] MEMIK, G., AND MANGIONE-SMITH, W. H. Improving power efficiency of multi-core network processors through data filtering. In *International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES)* (Grenoble, France, October 2002).

[55] MEMIK, G., AND MANGIONE-SMITH, W. H. NEPAL: A framework for efficiently structuring applications for network processors. In *Second Workshop on Network Processors (NP2)* (2003).

[56] MEMIK, G., MANGIONE-SMITH, W. H., AND HU, W. NetBench: A benchmarking suite for network processors. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)* (San Jose, CA, November 2001).

[57] MICHAEL, M. M., AND SCOTT, M. L. Nonblocking algorithms and preemption-safe locking on multiprogrammed shared — memory multiprocessors. *Journal of Parallel and Distributed Computing 51*, 1 (1998), 1–26.

[58] MICRON. RLDRAM memory components. http://www.micron.com/products/dram/rldram/, 2005.

[59] MUNOZ, R. Quantifying the cost of NPU software. Network Processor Conference East, 2003.

[60] MYCROFT, A., AND SHARP, R. A statically allocated parallel functional language. In *27th International Colloquium on Automata, Languages and Programming* (2000), Springer-Verlag, pp. 37 – 48.

[61] NATIONAL LABORATORY FOR APPLIED NETWORK RESEARCH. Passive measurement and analysis. http://pma.nlanr.net/PMA/, February 2004.

[62] NGUYEN, A.-T., MICHAEL, M., SHARMA, A., AND TORRELLAS, J. The Augmint multiprocessor simulation toolkit for Intel x86 architectures. In *Proceedings of 1996 International Conference on Computer Design* (October 2002).

[63] PARSON, D. Real-time resource allocators in network processors using FIFOs. In *Anchor* (2004).

[64] PAULIN, P., PILKINGTON, C., AND BENSOUDANE, E. StepNP: A system-level exploration platform for network processors. *IEEE Design & Test 19*, 6 (November 2002), 17–26.

[65] PLISHKER, W., RAVINDRAN, K., SHAH, N., AND KEUTZER, K. Automated task allocation for network processors. In *Network System Design Conference* (October 2004), pp. 235–245.

[66] RAMASWAMY, R., WENG, N., AND WOLF, T. Analysis of network processing workloads. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)* (Austin, TX, 2005).

[67] RAMASWAMY, R., AND WOLF, T. PacketBench: A tool for workload characterization of network processing. In *Proc. of IEEE 6th Annual Workshop on Workload Characterization (WWC-6)* (Austin, TX, Oct. 2003), pp. 42–50.

[68] RAMIREZ, G., CASWELL, B., , AND RATHAUS, N. *Nessus, Snort, & Ethereal Power Tools*. Syngress Publishing, September 2005, ch. 11–13, pp. 277–400.

[69] RINARD, M., AND DINIZ, P. Commutativity analysis: A technique for automatically parallelizing pointer-based computations. In *Proceedings of the Tenth IEEE International Parallel Processing Symposium (IPPS'96)* (Honolulu, HI, April 1996), pp. 14–22.

[70] RITKE, R., HONG, X., AND GERLA, M. Contradictory relationship between Hurst parameter and queueing performance (extended version). *Telecommunication Systems 16*, 1,2 (February 2001), 159–175.

[71] ROBERTS, L. G. Beyond Moore's law: Internet growth trends. *IEEE Computer 33*, 1 (January 2000), 117 – 119.

[72] RUF, L., FARKAS, K., HUG, H., AND PLATTNER, B. The PromethOS NP service programming interface. Tech. rep.

[73] SCHELLE, G., AND GRUNWALD, D. CUSP: a modular framework for high speed network applications on FPGAs. In *FPGA* (2005), pp. 246–257.

[74] SEAMANS, E., AND ROSENBLUM, M. Parallel decompositions of a packet-processing workload. In *Advanced Networking and Communications Hardware Workshop* (Germany, June 2004).

[75] SHAH, N., PLISHKER, W., RAVINDRAN, K., AND KEUTZER, K. NP-Click: A productive software development approach for network processors. *IEEE Micro 24*, 5 (September 2004), 45–54.

[76] SHERWOOD, T., VARGHESE, G., AND CALDER, B. A pipelined memory architecture for high throughput network processors, 2003.

[77] SHI, W., MACGREGOR, M. H., AND GBURZYNSKI, P. An adaptive load balancer for multiprocessor routers. *IEEE/ACM Transactions on Networking* (2005).

[78] SLECHTA, B., CROWE, D., FAHS, B., FERTIG, M., MUTHLER, G., QUEK, J., SPADINI, F., PATEL, S. J., AND LUMETTA, S. S. Dynamic optimization of micro-operations. In *9th International Symposium on High-Performance Computer Architecture* (February 2003).

[79] SYDIR, J., CHANDRA, P., KUMAR, A., LAKSHMANAMURTHY, S., LIN, L., AND VENKATACHALAM, M. Implementing voice over AAL2 on a network processor. *Intel Technology Journal 6*, 3 (August 2002).

[80] TAKADA, H., AND SAKAMURA, K. Schedulability of generalized multiframe task sets under static priority assignment. In *Fourth International Workshop on Real-Time Computing Systems and Applications (RTCSA'97)* (1997), pp. 80–86.

[81] TAN, Z., LIN, C., YIN, H., AND LI, B. Optimization and benchmark of cryptographic algorithms on network processors. *IEEE Micro 24*, 5 (September/October 2004), 55–69.

[82] THAIN, D., TANNENBAUM, T., AND LIVNY, M. Distributed computing in practice: The Condor experience. *Concurrency and Computation: Practice and Experience* (2004).

[83] THIELE, L., CHAKRABORTY, S., GRIES, M., AND KÜNZLI, S. *Network Processor Design: Issues and Practices*. First Workshop on Network Processors at the 8th International Symposium on High-Performance Computer Architecture (HPCA8). Morgan Kaufmann Publishers, Cambridge MA, USA, February 2002, ch. Design Space Exploration of Network Processor Architectures, pp. 30–41.

[84] TSAI, M., KULKARNI, C., SAUER, C., SHAH, N., AND KEUTZER, K. A benchmarking methodology for network processors. In *1st Workshop on Network Processors (NP-1), 8th Int. Symposium on High Performance Computing Architectures (HPCA-8)* (2002).

[85] UNGERER, T., ROBIČ, B., AND ŠILC, J. A survey of processors with explicit multithreading. *ACM Computing Surveys (CSUR) 35*, 1 (March 2003).

[86] VAN HEESCH, D. Doxygen. http://www.doxygen.org, 2005.

[87] VAZIRANI, V. V. *Approximation Algorithms*. Springer, 2001. Section 10.1.

[88] VIN, H. M., MUDIGONDA, J., JASON, J., JOHNSON, E. J., JU, R., KUNZE, A., AND LIAN, R. A programming environment for packet-processing systems: Design considerations. In *Workshop on Network Processors & Applications - NP3* (February 2004).

[89] WAGNER, J., AND LEUPERS, R. C compiler design for an industrial network processor. In *5th International Workshop on Software and Compilers for Embedded Systems (SCOPES)* (St. Goar, Germany, March 2001).

[90] WARNER, M. K. Mike's hardware. http://www.mikeshardware.co.uk, 2005.

[91] WARNER, W. Great moments in microprocessor history. http://www-128.ibm.com/developerworks, December 2004.

[92] WENG, N., AND WOLF, T. Pipelining vs. multiprocessors ? Choosing the right network processor system topology. In *Advanced Networking and Communications Hardware Workshop* (Germany, June 2004).

[93] WILD, T., FOAG, J., PAZOS, N., AND BRUNNBAUER, W., Eds. *Mapping and scheduling for architecture exploration of networking SoCs* (January 2003), vol. 16 of *International Conference on VLSI Design*.

[94] WILSON, P. R. Uniprocessor garbage collection techniques. In *Proc. Int. Workshop on Memory Management* (Saint-Malo (France), 1992), no. 637, Springer-Verlag.

[95] WOLF, AND T. Design of an instruction set for modular network processor. Tech. Rep. RC21865, IBM Research, October 2000.

[96] WOLF, T., AND FRANKLIN, M. CommBench - a telecommunications benchmark for network processors. In *Proc. of IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)* (Austin, TX, April 2000), pp. 154–162.

[97] WULF, W. A., AND MCKEE, S. A. Hitting the memory wall: Implications of the obvious. *Computer Architecture News 23*, 1 (1995), 20–24.

[98] XORP PROJECT. XORP design overview, April 2005.

[99] ZHAI, A., COLOHAN, C. B., STEFFAN, J. G., AND MOWRY, T. C. Compiler optimization of scalar value communication between speculative threads. In *Tenth International Conference on Architectural Support for Programming Languages and Operating Systems* (San Jose, October 2002).