ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

TIK Institut für Technische Informatik und Kommunikationsnetze

Adrian von Bidder

A Framework for Programming Packet Processors

Diploma Thesis DA-2004-01
April 2004 to August 2004

Supervisor: Lukas Ruf
Professor: Bernhard Plattner

Abstract

To satisfy demands in modern networks, routers employ specialised *network processors*, usually consisting of a general purpose CPU and several packet processors. With their high processing power and versatility they are well suited for use as components in an active network infrastructure. However, up to now the software running on network processors in typical implementations consisted of a fixed firmware image. This diploma thesis aims to implement a plug-in loader for the packet processors of the IBM PowerNP network processor to allow run-time code deployment on active network nodes, and to provide a means to pass network packets between the packet processors and other processing elements in the active network node.

Um den Anforderungen moderner Computernetzwerke gerecht zu werden, benutzen Router heute spezialisierte *Netzwerkprozessoren*, welche üblicherweise aus einem standard-Mikroprozessor sowie mehreren Paketprozessoren bestehen. Dank ihrer Flexibilität und hoher Leistung eignen sich Netzwerkprozessoren auch sehr gut als Komponenten in aktiven Netzwerken. Bisher wurden diese Prozessoren typischerweise mit einem feststehenden Firmware-Image eingesetzt. Diese Diplomarbeit implementiert nun einen Plug-in Lader für die Paketprozessoren des IBM PowerNP Netzwerkprozuessors, womit zur Laufzeit Programmcode auf die Paketprozessoren geladen werden kann. Ausserdem wird eine Kommunikationsinfrastrucktur bereitgestellt, um Netwerkpakete zwischen allen Komponenten eines PowerNP-basierten Netzwerkknoten ausgetauscht werden können.

Preface

When I entered the ETH in 1998, my primary interest in computers was in programming languages and operating systems. Over time, this has certainly broadened, I've become interested in cryptography and databases as well, briefly touched on Web services and also enjoyed the courses dealing with actual hardware very much. Working on the PowerNP processor can once more completely satisfy this interest in fiddling with single bits and dealing with the nice world of word endianness problems and funny memory addressing modes (and, as icing on the cake, I got so much documentation that I had serious problems sorting out the relevant bits of it.)

While this kind of thing is probably not what I will be doing in the near future, I've still learned a lot from this project (about both actual hardware oriented programming and project and time management), and despite the occasional moments of frustration I've enjoyed the time here at the TIK. So, I'd like to thank Prof. Bernhard Plattner and Lukas Ruf for letting me play with this funky hardware, and the support staff for their everlasting patience with our frequent requests for more network cards, screwdrivers and the keys to unlock the computers and move the PowerNP boards around. Finally, the staff of the Gloriabar and the other students in the room were always a good source for entertainment for those times when I needed a short break.

Zürich, 4.8.2004

Contents

Chapter 1

Introduction

This chapter starts with a few words on Network routing in general, then the PromethOS router platform is introduced to provide a reference frame for the rest of this document. After a section about the goals of this diploma project, a short outline of the rest of the thesis closes this introductory chapter.

1.1 Internet Routing

The way we use computers has changed fundamentally in the last 15 years, with calculator and typewriter functionality being replaced by communication facilities as the most important application. This has only been possible by the almost universal adaption of a single networking standard, the IP protocol suite [17], and the transition from individual corporate and academic networks to today's Internet, with network infrastructure enabling communication between virtually any two devices on the network.

With the increasing number of Internet users and complexity of Internet application, data traffic has also increased by several orders of magnitude and network infrastructure had evolve to handle the high volume. The underlying technology, however, has remained virtually unchanged from the earliest days of computer networking: data is transferred in packets which are individually sent to their destination, often passing multiple intermediate systems. These intermediate systems, called routers are the principal topic of this thesis. This report, and especially the implementation developed in this thesis, will only use today's most widespread Internet protocol, IPv4 over Ethernet [17], but it is important to note that the underlying theory applies to all packet-based network systems in an analogous way.

1.1.1 Routing Requirements

Until now, Internet routers have done little more than look at the destination address of incoming data packets and decide where to send them. This is quickly changing as new network protocols are deployed: A starting point is the deployment of multicast [2], where routers distribute network packets not just to one, but to multiple destinations, and manage information about who is interested in which packets.

Other applications that require more intelligent routers may be things like statistics gathering, intrusion detecting on the router or, in conjunction with multicast, adaptive video scaling: the provider of a video stream distributes high-resolution data, and the router may scale this down for redistribution over low-bandwidth data channels. Environments enabling such advanced functionality are called *active networks*, as opposed to conventional "passive" networks where network routers merely send packets to their final destination, and where all actual data processing happens at either the source or destination of the data stream.

1.1.2 New Routing Architectures

Common to many of these possible applications of intelligent router technology is that the required CPU bandwidth (and associated items like memory capacity and internal communication bandwidth) far exceeds what is available on todays routers.

In the past, when requirements were higher than what was available on standard hardware, two approaches have been used very successfully: distributing the task over multiple computers, and building special hardware optimized for the requirements. The framework developed in this thesis focuses on integrating the two: enabling load distribution in a distributed router architecture with special-purpose network processors offering a part of the computing power by dynamically changing the software running on the routers in response to changing requirements of the network.

1.2 The PromethOS High-Speed Router Framework

The PromethOS NP project [16, 18] implements a high performance network router for an active network environment. Multiple processors are combined to build a distributed router with a common management software. True to the active network philosophy, plug-ins can be loaded onto the components of this router at run-time to adapt to changing requirements, also, the management software tracks resource usage of all components of the router platform and considers this information when deciding where to load plug-ins. Processors can either be general purpose processors or special purpose network processors, and the PromethOS NP framework offers communication facilities so that data and control information can be exchanged between all components of such a distributed router.

PromethOS NP is implemented as a combination of Linux kernel modules and userspace applications, and application specific code running on the packet processors of network processors in the router architecture. A schematic overview of a PromethOS router is shown in Figure 1.1. The Linux kernel module (the *proxy device driver*) receives and sends the network packets to be handled by the PromethOS NP framework at a very low level to keep processing overhead as low as possible, bypassing most of the Linux networking code.

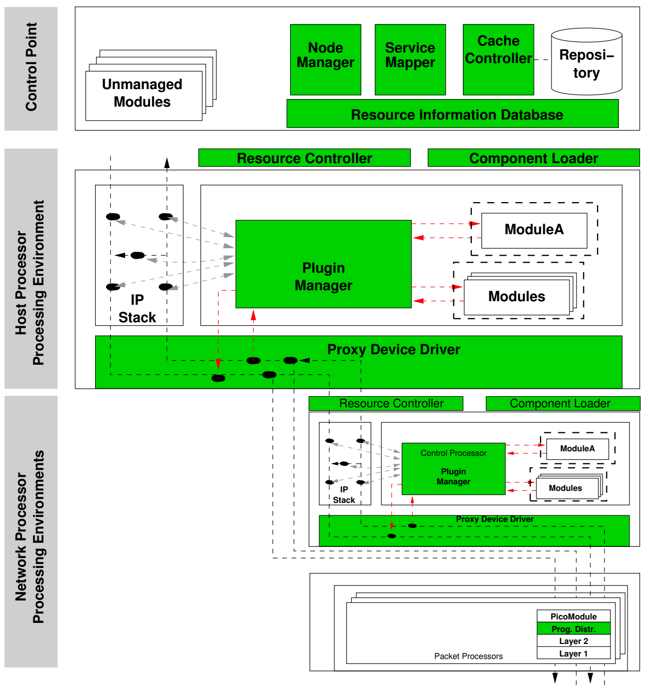


Figure 1.1: PromethOS NP router architecture

A hardware abstraction layer provides a uniform interface to all software components (such as plug-in loaders for different hardware implementations, communication facilities over various channels between components of a router etc.) to keep the core logic of PromethOS NP – the resource management facility and overall plug-in management – easy to adapt to new types of processing elements.

1.3 This Diploma Thesis

Much of the PromethOS NP framework has already been implemented and tested in earlier work by Pascal Erni, Lukas Ruf and others [3, 16, 18], including partial integration of the IBM PowerNP network processor (the PowerNP will be described in chapter 2). The task set for this thesis is the completion of the PowerNP network processor integration with the rest of the PromethOS architecture, namely

the plug-in loader: PromethOS plugin are loaded onto the packet processors of the IBM PowerNP at run time, without requiring a break in processing of network packets.

plug-in integration: Network packets are processed by the packet processor plug-ins, and can then be dispatched to be routed onto the network or for furhter processing by other packet processor plug-ins or by PromethOS plug-ins on other processors in the distributed router.

communication paths: All components of a PromethOS router need to be able to exchange data and control information. This is implemented for a router environemnt employing IBM PowerNP network processors and standard-issue general purpose CPUs as processing elements.

In addition to the implementation, corresponding documentation is provided to allow others to get acquainted with the system as quickly as possible. The original task assignment (in German) for this thesis can be read in appendix A.

1.4 Structure of this Document

The rest of this document describes the implementation outlined above in great detail, starting with chapter 2 which introduces the architecture of the PowerNP network processor used in this thesis. It can and does not intend to replace any of the original documentation, but rather should make it easier for the reader to locate the relevant parts of the extensive original documentation. The main part of the thesis, the plug-in loader, is addressed in chapter 3, which will also describe the mechanism to redirect network packets from the packet processors to the other components in the system. In chapter 4 an easy way for communication between the embedded PPC405 general purpose CPU of the PowerNP and the host CPU (i.e., the CPU of the computer the PowerNP board is installed in) is introduced. The evaluation (chapter 5) and a few closing remarks (6) conclude the main body of the thesis, with the original task assignment, the planned project schedule, an overview of the contents of the included CD and some instructions on how to get started with all the tools making up the appendices.

Chapter 2

The PowerNP Network Processor

This chapter does not intend to be a complete documentation of the IBM/Hifn[1] PowerNP processor [10], or to replace any part of the documentation provided by IBM. Instead, it is an extensively commented collection of pointers to those parts of the original documentation which are relevant to this diploma thesis. To be precise: described is not only the PowerNP processor itself, but the PowerNP and the S3 (Silicon & Software Systems) *Application Reference Board* (ARB) [20] as it was used in this thesis – things like the boot process or the memory layout depend on how the PowerNP is connected to the surrounding circuitry.

2.1 Network processors

But first, let's answer the question what a network processor is. As Niraj Shah and Kurt Keutzer note in [19], there is no single set of features that determine what a network processor is – their broad definition includes *any processor able to efficiently process packets for network communication.* Nonetheless, they identify five important areas where network processors differ from general purpose CPUs.

Parallel Processing In a typical network router, many independent packet streams are present simultaneously, with no or almost no interdependencies, which makes it very easy to schedule tasks in parallel. Consequently, designers of network processors often equip their products with a number of parallel processing units.

Special Purpose Hardware Network processors often are able to perform certain frequently needed tasks in hardware, such as sophisticated bit field manipulation, checksum operations, table lookups or queue management. The number of special-purpose function units, their design, and the way they are controlled varies wildly from product to product.

[1]The PowerNP architecture was originally developed by IBM but later bought by Hifn. IBM discontinued support of the PowerNP end of March 2004.

Memory Architectures Computing tasks in network processors are mostly relatively simple, but need to be done on many data elements, thus memory bandwidth quickly becomes a main bottleneck. Most network processors therefore feature a memory subsystem designed to either avoid or at least hide memory latencies, by holding data close to the computing circuitry or by transparently executing multiple tasks on the same processing element. In addition, some designers also offer hardware implementations of memory management functions normally found in the operating system kernel.

On-Chip Communication As outlined above, most network processors feature a wide array of specialized processing circuits – implying that data needs to be moved between them as fast as possible. With most network processor architectures, this has resulted in a very specialized on-chip communications infrastructure, often directly supporting features of particular network protocols. This, together with the specialized memory layout, has proved to make true understanding of a network processor design quite a challenge.

Peripherals In addition to quickly process a large number of network packets, network processors need to move packets on to and off the chip. Network processors therefore often directly feature industry standard network interfaces like Ethernet, ATM or SONET.

Network processors are manufactored by many vendors; examples (other than the IBM PowerNP) are the Motorola C5 series [14] and the Intel IXP [15].

2.2 PowerNP Architecture Overview

The main components of the PowerNP as shown in figure 2.1 are the *Embedded Processor Complex* (EPC) with the embedded PowerPC 405 CPU (ePPC) and the Picoengines, the switch interfaces, the *Enqueuer/Dequeuer/Scheduler* (EDS) units and the *Physical MAC multiplexer* (PMM). Most components exist twice, on the ingress and the egress data path, except for the EPC, which handles both directions in one unit and the ePPC which is not in the main data path through the PowerNP: packets usually enter the PowerNP at a PMM and are enqueued to the ingress EDS by the corresponding *Data Mover Unit* (DMU). The ingress EDS passes control to the EPC which processes the packet and signals completion back to the EDS which then enqueues it to the ingress *switch interface*. In the S3 ARB, packets are passed directly to the egress switch interface, which signals the arrival to the egress EDS. The egress EDS lets the packet be processed by the EPC again and then enqueues it for delivery on a DMU, from where it is written to the physical network by the PMM. The following sections describe these functional units in more detail, roughly following the flow of a packet through the PowerNP processor.

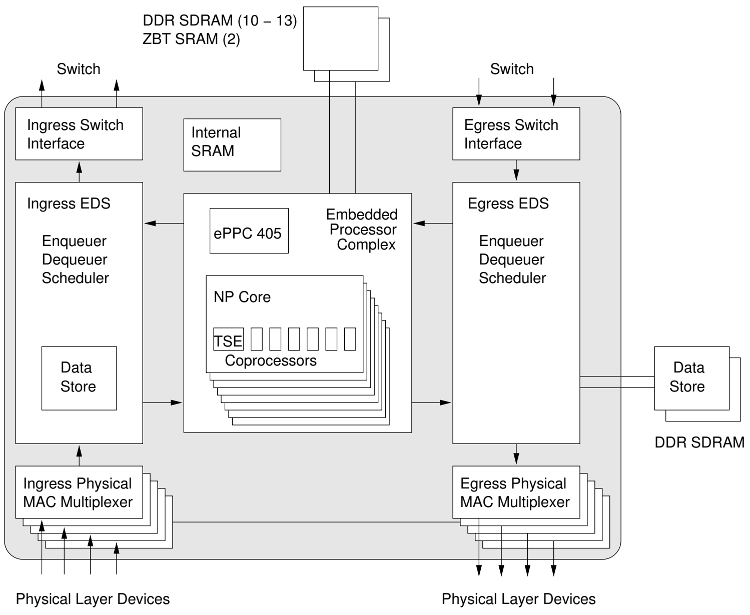


Figure 2.1: Components of the PowerNP processor

2.2.1 Data Mover Units and Physical MAC Multiplexer

The PowerNP (in its NP3GS3 version, as used on the Application Reference Board) offers connectivity to 4 gigabit Ethernet or OC-48 SONET links or 40 100Mbps Ethernet links, or combinations thereof, via four *Data Mover Units* (DMU A through DMU D). On the ARB, three Gb Ethernet connectors are available, while DMU A is directly wired to a Broadcom Gb Ethernet chip and is used as connection to the PCI bus of the host computer. To access this Ethernet chip, the *copernicus* driver is used (included in the subversion repository on the PromethOS server); there was no problem whatsoever with this driver throughout this thesis[2].

Additional to the aforementioned Data Mover Units, there is a special DMU E (or *wrap DMU*), which injects all packages sent to it (from the egress side) back to the ingress side for processing. As will be seen in chapter 3, this is an essential feature for some application.

2.2.2 Enqueuer Dequeuer Scheduler

The Ingress and Egress Enqueuer Dequeuer Scheduler (EDS) units deal with moving network packages between the DMUs, the Embedded Processor Complex (EPC, see below) and the switch interfaces. The units handle receiving the packets, storing the data in a data store memory and starting the processing in the EPC. When the EPC has finished processing each package, the EDS is responsible for enqueueing it to its destination (the switch interface on the ingress side, the Data Mover Units on the egress side), respecting

[2]Use of jumbo-frames (which are supported by the PowerNP) was not tested; just increasing the MTU of the Ethernet interface failed, though, so maybe some work is necessary in this area.

policy decisions made by the EPC (rate limiting, for example, or whether packet ordering should be kept for certain packages.)

The important thing about the EDS units is that once they are properly set up, they operate fully automatic, and the application programmer (programming the EPC) does not have to handle most queue and memory management issues. While this may make the task of writing routing code much easier (reducing it to the part where the routing decisions are actually felled), it makes understanding the PowerNP architecture a lot more complicated – a big part of the package handling within the PowerNP is done in these units, and most of it happens “behind the scenes”.

2.2.3 Embedded Processor Complex

The EPC is the central part of the PowerNP – and probably the most complex one. As soon as the EDS (on either side) has received a network packet[3], processing is started in the EPC. After processing is finished, handling of the packet is passed back to the EDS unit it came from. IBM recommends to do as much processing as possible on the ingress side because network packets in the ingress EDS are stored on memory directly on the PowerNP chip (with a very fast link to the EPC) while the memory of the egress EDS consists of external, slower, DRAM. On the other hand, the ingress EDS's memory is much smaller than the egress EDS's memory, so there may be cases where processing should be done on the egress side to allow many packets to be buffered for some time.

Because of its complexity, the EPC is described in more details below; the Core Language Processors are described in section 2.3 and the coprocessors in 2.4.

2.2.4 Switch Interface

Once a packet has been enqueued for delivery in the ingress EDS, the ingress switch interface sends it off the PowerNP to either the egress switch interface of the same PowerNP (as is the case with the Application Reference Board used here), to a second PowerNP's egress switch interface or to a switch fabric interconnecting many PowerNP processors in a parallel router architecture with more than 2 PowerNP processors.

On the egress side, a packet received on the switch interface is handled by the egress EDS, run through the EPC and then enqueued to a DMU for delivery; the ingress and egress data paths are quite similar in this regard, even though there are many details that are different since the tasks to be done differ.

2.2.5 Embedded PowerPC 405

Outside of the typical data flow through the PowerNP is the embedded PowerPC 405 CPU which is included on the PowerNP chip. This PowerPC CPU is a standard general purpose

[3]Actually, already when a part of the packet has been received, but this is mostly invisible to the application programmer, even when he requests data that has not yet been received.

processor and can run a normal Linux system. Communication between the PowerPC and most other units of the PowerNP chip is possible through the *Coprocessor Access Bus* (CAB, [12] p. 405) for direct (but slow) access to most memory regions and through a *mailslot interface* (p. 414 of the same manual) to efficiently pass network packets from the EPC to the ePPC and back.

The Linux system used on this processor is described in more detail in appendix D – it is a minimal Linux system, based on a 2.4 kernel and usees busybox to replace all of the standard system binaries, so the full system fits on a 8M ram disk. No substantial changes to this system were made in this thesis.

2.3 Packet Processors

As mentioned in section 2.1, a distinguishing features of network processors is the availability of special hardware for network packet processing, often in the form of specialized processing units generally called packet processors. In the IBM/Hifn PowerNP architecture these are called *Core Language Processors* (CLP) or simply picoengines, derived from the name of their instruction set, the *picocode*. The PowerNP features 16 picoengines in 8 *Dyadic Protocol Processor Units* (DPPU) with 2 picoprocessors and 10 specialised coprocessors shared between them in each DPPU. Also, each picoengine can execute 2 threads semi-concurrently, similar to the "Hyperthreading" feature introduced in the DEC Alpha EV8 and offered in modern AMD or Intel general purpose CPUs, for a total of 32 threads being executed in parallel.

Of those 32 threads, 28 are *General Data Handler* (GDH) threads and used for network packet processing. The other 4 threads have additional capabilities and are used for managing the PowerNP: the *Guided Frame Handler* processes commands issued to the EPC in so-called *guided frames* (see section 4.1.) The *General Table Handler* thread processes special commands relating to data structures of the *tree search engine* (see below), and the *General PowerPC Handler Request* and Response threads handle the communication between the embedded PowerPC and the packet processors via the mailslot interface. If no request is pending which requires one of these special threads, they can also process data packets like the GDH threads.

In most cases, application programmers do not need to deal with parallel programming at all, but write a normal single-threaded program to handle a single network packet - most thread scheduling and resource control issues are handled by the hardware.

The picocode instruction set (described in the Assembler Language Programmer's Guide [9] and also in the data sheet [12] p. 199ff) is similar to most other RISC platforms: 32 bit word size and 32 bit instruction length and the usual arithmetic and control instructions are present. Instructions dealing with half words and single bytes as well as bit fields make it relatively easy to deal with typical network packet headers. However (and more about that in section 2.5), the address modes available are more complex than in most other CPU architectures, and much of the work is done asynchronously on the various coprocessors (see below), so that reading picocode assembler as provided, for example, in the IBM *Advanced Software Offerings* can not be compared with reading code for most general purpose computers.

2.4 Coprocessors

One of the key strengths of the PowerNP platform are the many coprocessors available to accelerate tasks frequently used when processing network packets. This section contains an overview of the available coprocessors and a few details about the functionality used in this thesis, but please refer to p. 228ff of the data sheet [12] for a complete description of all the features. While some coprocessors (for example the string copy and checksum coprocessors) are local to each DPPU, other coprocessors interface with a corresponding global unit within the PowerNP (the enqueue coprocessor interfaces with the completion unit, the counter coprocessor with the counter manager etc.)

Coprocessor instructions can be executed synchronously or asynchronously[4] and take arguments directly in the opcode and (for some coprocessor instructions) in a general purpose register. Furthermore, many coprocessor instructions require data structures in certain memory locations to be set up correctly. Again, for a detailed list of input and output parameters of coprocessor instructions, please see the descriptions in the data sheet. For asynchronous coprocessor calls, synchronisation is done with the `wait` family of instructions – these can either just wait for completion of coprocessor operations, or can conditionally branch on either success or failure of a specific coprocessor operation.

While the Assembler Language Programmer's Guide [9] describes the `cpx` instruction to execute a coprocessor instruction, this is generally done via specific mnemonics which imply the `cpx` instruction. For example, instead of using `cpxa enqe,` *`arguments`* it is recommended to use just `enqe` *`arguments`*. Since picocode programs usually use quite a lot coprocessor calls, this greatly improves readability of the code.

The ordering of the coprocessors in the overview below is somewhat arbitrary and is taken from the IBM documentation – it can be seen as a rough ordering by importance.

Tree Search Engine The tree search engine is a generic table lookup unit, implemented completely in hardware. Since the format of the table entries (to be more exact: leaf nodes - tables are implemented as search trees) is user defined, the tree search engine can be used for many different purposes. The tree search engine is one of most powerful units of the PowerNP and also one of the most complex ones.

Data Store The data store coprocessor provides an interface between the EPC and the ingress and egress EDS. It is used to transfer network packet data between the EDS and the EPC whenever a packet needs to be processed.

Control Access Bus Interface The control access bus (CAB) is an interface that allows access to almost all components of the PowerNP processor and for some memory

[4]and also with or without retaining priority; a feature which was not used and is not further discussed in this thesis.

regions is the only method for the application programmer to access them directly (for example, the picocode instruction memory; see section 2.5.2). But the bandwidth is very low, and all 32 threads and the embedded PowerPC CPU share just one CAB, so it is not usually used during network data processing. It is, however, crucial during the set up of the PowerNP at boot time.

Enqueue The enqueue coprocessor is responsible for passing a package back to the EDS after processing and freeing the allocated resources in the EPC. How the package is further processed by the EDS depends on which queue the package was enqueued to (this is specified as an immediate operand to the enqueue instruction) and on the content of the *Frame Control Buffer* (FCB) Page, which is the data structure interpreted by the EDS hardware (the data fields are described in the data sheet [12] starting at section 7.4.4.1 on p. 250.)

Checksum It generates and verifies IPv4 header checksums. The IP header is read from or written to in one of the memory areas managed by the Data Store coprocessor.

String Copy The string copy coprocessor has only one command, `strcopy`, which copies data between any two memory locations within the EPC. (In contrast to the data store coprocessor which copies data between memory locations in the EPC and the EDS.)

Policy The PowerNP supports the notion of a *flow* as a sequence of network packets that are somehow related, and has several options to deal with flows[5] In this thesis, the policy handling capabilities of the PowerNP have not been used, so please refer to the manual for more details.

Counter The counter coprocessor provides an interface to the counter manager in the EPC and is used for statistics gathering. There are counters which are automatically updated by the hardware and a number of counters controllable by the picocode software.

Coprocessor Response Bus The coprocessor response bus coprocessor can be used to attach additional coprocessors to the PowerNP – a feature which is not used at all on the application reference board.

Semaphore The semaphore manager offers the possibility for inter-process synchronisation between threads running concurrently on the CLPs, it can be used to manage shared data structures. Most of the data structures supported directly by the hardware are already properly protected from concurrent access by multiple threads, so the semaphore coprocessor is only required to manage additional, application-defined data structures in the PowerNP.

[5]The algorithms implemented by the policy manager are the 'Single Rate Three Color Marker' and the 'Two Rate Three Color Marker' algorithms as specified in RFC 2697 and 2698 [4, 5].

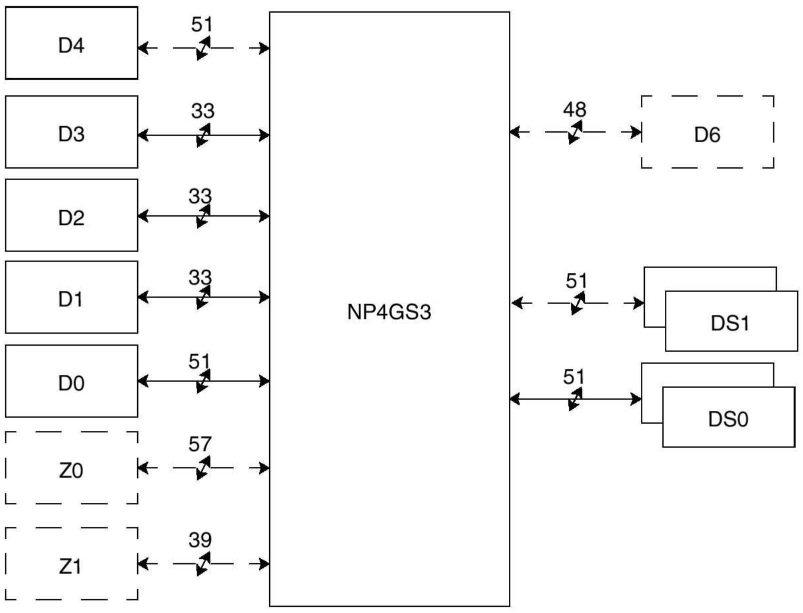


Figure 2.2: External memories controlled by the PowerNP

2.5 Memory Layout

The PowerNP does not implement the “flat” memory address space present in most general purpose CPU. Instead, it provides an array of many different, specialised memories with very different characteristics. Many of these are addressable via the CAB bus; the data sheet [12] shows in section 7.4.3.2 (p. 247) how a CAB address is structured. Other areas can be directly addressed from assembler instructions, please see the Assembler Language Programmer’s Guide [9] p. 65ff (chapter 4) for more information about memory addressing.

Not usually referred to as “memory”, each of the 32 threads has access to a register set consisting of general purpose registers of the CLP itself and a register set for each coprocessor[6]. Additionally, there is a thread-local memory pool of 1kb which is heavily used in packet processing, its layout is described in section 7.2.4 of the data sheet (p. 198).

2.5.1 Global Data Storage

Most of the memory areas are global to the whole PowerNP, these are quickly described below. Fundamentally, there are on-chip and external memories, and some of the external memories can be omitted if some features of the PowerNP are not used (showed as dashed boxes in figure 2.2). Details are explained in various places in the data sheet ([12], sections 8.1.1 p. 315 and 13.1 p.473), in the hardware reference manual ([13], explains where each data structure is stored) and in the Application Reference Board Technical Description ([11], sections 2.3 to 2.5, p. 20f.)

[6]some of these coprocessor registers, called *array registers*, are really small memory regions and not what is usually associated with the term.

A short overview of all the memories is given here: **H0** and **H1** are two fast but small on-chip SRAMs, **Z0** is an optional external SRAM memory, **D0**, **D1**, **D2** and **D3** are external DRAM memories, a lot slower than the SRAM ones, but considerably larger. All these are used for data structures of the tree search engine. There are various constraints on where data structures can be stored (listed in the Hardware Reference Manual [13] and in chapter 8 of the data sheet [12]), but generally it is always possible to store the most frequently accessed structures into a fast static RAM. The external **D4** DRAM area is a grab-bag for all hardware-supported data structures which are not held in other (dedicated) memory areas. **DS0** and **DS1** are external DRAMs which hold the egress data store. The optional **Z1** memory is an fast external static RAM for use by the egress scheduler. And, finally, the **ingress data store** is built into the ingress enqueue dequeue schedule unit and is, consequently, fast but quite small.

2.5.2 Picocode Memory

There are two memory areas not usually directly accessed by the application programmer: the flash memory and the picocode instruction memory. The latter contains the instructions executed by the picoengines; it has a capacity of 32k Instructions (or 128kB) and is initialized at system boot, exactly how depends on the way the PowerNP is used in the hardware. In the S3 application reference board, the procedure is according to section 10.9.5 of the data sheet [12]: The picoinstruction memory is loaded from the flash memory, and the code for the embedded PowerPC is provided through guided traffic.

For this thesis, the flash memory has not been modified, so please refer to the IBM documentation and to the diploma thesis of Pascal Erni [3] for more details. The picoinstruction memory, however, is a central component for the plug-in loader described in this thesis. The description is in the Hardware Reference Manual [13], section 3.11 (p.304), the plug-in manager accesses it through the CAB as described in section 3.3.

2.5.3 Area D6

As described before, the embedded PowerPC 405 runs a Linux system. The PowerPC has its own memory area, D6, which is the only memory area directly accessible to it. Communication with the other parts of the PowerNP happens in two ways: the control access bus described in section 2.4, and the mailbox interface which is described in the data sheet [12], section 10.8 (p. 414ff).

The D6 memory can also be accessed by the picoprocessors, and (at a higher level) this is how the PowerPC is booted: the `d6load` tool uses the Block_Write_Raw guided command as described in IBM's ASO Control Application Programming Reference [8] on p. 74 (the *Guided Frame Format* section on p.58 of the same manual is probably necessary to understand how to use that command) to load the Linux kernel and an initial ram disk into the D6 load before starting the PowerPC processor. More information about the Linux system on the embedded PowerPC is contained in section D.4 in the appendix.

Chapter 3

Plug-in Loader

Network processors so far have been deployed with a statically compiled firmware containing the software to route and filter packets, with run-time configuration being limited to activating or deactivating certain features by adding and modifying routing tables and firewall rules. In section 1.1.1 the idea of active networking was described, where network routers need to provide much more functionality than just forwarding or filtering packets. Also, when a node in for an active network is designed or even deployed, it may be unknown what functionality this node should execute. Hence the requirement to be able to load plug-ins onto the network processor at run-time.

In this chapter the plug-in loader architecture is explained in detail. The first section of this chapter highlights the special difficulties of writing a plug-in loader on the PowerNP architecture, the remainder contains an in-depth description of the plug-in interface and describes how to use the tools to assemble and load plugins.

3.1 Constraints

With the ubiquity of architectures supporting shared libraries (using ELF, PE or some other executable file format), an architecture to dynamically load executable code into a runtime environment does not sound very exciting. But the core language processors on the IBM PowerNP do not implement a modern general purpose architecture, so that implementing a plug-in loader on this architecture was not as straightforward as it seems at first. Also, as much as possible of the available software, the IBM *Advanced Software Offerings* (ASO), should be used, as writing a new runtime environment from scratch would be a dauntingly complex task – and this ASO was not designed to support dynamically loaded code.

Additionally, as described in section 2.5, the instruction memory of the PowerNP is only 32k instructions big, at 4 bytes per instruction. Roughly 27k instructions are already used by the ASO firmware, so that only 4k instructions are available for loadable plug ins. Setting the size of a plug-in arbitrarily to 1k instructions, this means that 4 plug-ins can be loaded simultaneously – further evaluation will tell how many plug-ins will typically be needed, and how big plug-ins typically will be. This is, however, not within the scope of this thesis.

As mentioned above, the IBM ASO is not designed with dynamically loadable code in mind. Furthermore, the PowerNP hardware platform lacks some features that are present in today's general purpose CPUs which ease the implementation of dynamically reconfigurable programs considerably.

3.1.1 No MMU

The first such feature is the memory management unit, MMU. In light of dynamically loadable plug-ins, the main use of a MMU is that it creates a virtual address space and allows any arbitrary mapping to physical addresses. This allows the repeated allocation and deallocation of physical memory regions of different sizes into the virtual address space without fragmentation in the physical memory becoming a problem (fragmentation within the virtual address space is not usually a problem because the virtual address space typically is several orders of magnitude bigger than the physical address space, and in many cases is only relatively short-lived.)

Lacking memory address virtualisation, we decided to use plug-ins of a fixed size. That way, repeated allocations/deallocations of memory will never fragment the free memory region in the instruction memory in a way to make it impossible to load other plug-ins. As an added benefit, using fixed-sized plug-ins at fixed addresses simplifies the code to call the plug-ins considerably.

The other main feature of a MMU, access protection between memory regions of different owners, is less important in this context, even though it could probably be useful when plug-ins should be executed which are not completely trusted (for example in a router architecture that offers CPU cycles to process certain network packets but does not necessarily trust the provider of the plug-in code.)

It should be noted that it is completely reasonable for IBM not to have equipped the core language processors with a memory management unit: the very small address space of the instruction memory is a convincing argument for omitting address virtualisation there and the complex memory layout of the data memories would have made an MMU for the whole PowerNP memory architecture not only a likely performance problem and waste of silicon space but would certainly also have made understanding the PowerNP architecture still more difficult.

3.1.2 Limited Support for Relative Branches

On most very small embedded systems, code is compiled and linked statically to a fixed address. On the other hand, on all modern environments which support the notion of an operation systems separate from the application programs, code is compiled to be relocatable, so that it can be loaded to an arbitrary address in memory – all of these platforms also support the notion of a dynamic library, i.e. code that is loaded into the address space of a program after the program is started[1]. To allow relocatable code, the instruction set of the platform needs to have branch instructions that address their target relative to the current program counter (called *relative* branches).

The PowerNP has several branch instructions, namely the unconditional branch (`b`), conditional branch with condition codes (`b{cond}` – the condition codes are similar to most other CPU architectures), "branch and link" subroutine call (`bal`) and conditional branches testing on a specific bit in a register (`b0bit` and `b1bit`). While most of these can be used for a relative branch, the `bal` (branch and link) instruction apparently cannot. To be more precise: according to the assembler documentation documentation the `bal` instruction does perform a relative branch ([9] p. 204f), while the data sheet ([12] p. 201) clearly says that the immediate value is directly loaded into the program counter, which would mean an absolute branch.

This limitation does not mean that relocatable plug-ins are impossible, but it does mean that using subroutine calls within plug-ins needs special thought, and it limits the use of the picocode C compiler provided by IBM, which generates normal (absolute) `bal` instructions[2]. There are two possibilities to work around this problem: first, storing the return address for a procedure call manually into the link register and then using an unconditional branch to jump to the procedure, or storing the relocated target address into a register and using the branch and link operation with a register argument.

3.2 User Interface

The Plug-in Loader is built on top of the `NPCtrlD`/`NPCtrl` interface written by Pascal Erni in [3] and has functions to upload and activate and to deactivate picocode as well as to set rules in the *multi field classifier* (MFC – see p. 213 of [7]) to call the plug-ins. The command syntax is identical with the original `NPCtrl` command, except that a new 'load' command is used instead of the 'addRule' command. A plug-in id in the range 0 to 3 (inclusive) has to be specified. Unloading of plug-ins is done through the 'flush' command of the `NPCtrl` command. A sample session starting the networking on the embedded PowerPC Linux shell and loading a plug-in is shown in figure 3.1.

3.3 The NPCtrlD daemon

To load, activate and deactivate picocode plug-ins, the `NPCtrlD` daemon was extended to recognize the 'load' command. As manually loading a plug-in in this way is only expected

[1] Although from the view of the application programmers, dynamic libraries are loaded at program start up, it is important to note that as far as the kernel is concerned, libraries are loaded entirely under the control of the application program.

[2] In the author's opinion, the many coprocessor calls and the special code to deal with the complicated memory model makes the C code unreadable to the point where it does not have any advantage over writing plain assembler code.

```
# cd /usr/bin
# npcp
command output omitted
# NPCtrlD &
command output omitted
# NPCtrl -u 2 -p tcp -s 192.168.92.0 -S 192.168.92.255 -l 23 -L 25
most command output omitted
Added Rule successfully.
#
```

Figure 3.1: Sample shell session: load a plugin

to occur within a proof-of-concept implementation, with a full implementation either integrating the functionality of the node manager within the control point (`npcp`) or at least with the `NPCtrlD` daemon, a simple implementation was chosen: the message between `NPCtrl` and `NPCtrlD` does only contain the plug-in id, and the actual plug-in code is read from the file system by `NPCtrlD` from a file named `pico_`*x*`.bin` (where x is the plug-in id) in the working directory of the NPCtrlD daemon.

As explained above, picocode plug-ins currently have a fixed size of 1024 instructions and the picocode firmware is about 27k instructions long, which means that 4 plug-ins can be loaded simultaneously, starting at addresses `0x7000`, `0x7400`, `0x7800` and `0x7c00` in the instruction memory. These four slots are directly mapped to the plug-in ids 0 to 3.

The `AddPlugin` function of `NPCtrlD.c` in `aso-134/src/wrapper/linux/NPCtrlD` contains most of the added code between r288 and the HEAD revision in the subversion repository; the `NPCtrl` frontend program source code is in `aso-134/src/NPCtrl`. To access the instruction memory, the CAB (co-processor access bus) Driver API is used, which is described on p. 706 of [8]. Information about the layout and access to the picocode instruction memory can be found in [13] on p. 304, most importantly its CAB address range and addressing mode (quadwords at 0x24000000 to 0x2401FFF0, with the lowest nibble being used to address the word within every quadword.)

The cited documentation is very brief, so as additional reference the source code of the CAB driver API in the `cabdd` directory and the code in `linux/powerl2/pci_init.c` was used, which showed that the CAB driver API does not access the CAB directly but merely handles the CAB bus protocol but requires user-provided funtions to access the CAB configuration registers. The CAB configuration registers are mapped into the physical address space of the ePPC and so they are accessible to `NPCtrlD` via the `/dev/mem` Linux character special file. While this access method works with the software in use for this thesis project, it should be pointed out that accessing the CAB registers in this way is unsafe and does not protect against other programs using the CAB at the same time.

To verify that the plugin code is indeed written into the instruction memory, a small demonstration program to read the whole instruction memory was written, it can be found

```
ORG 0x0000;

    do something here ...

; no further plugin to be called --> set r28 to -1
    xor r28, r28
    sub r28, #1
    ret

; cause the plugin file to be 1024 instructions long
ORG 0x03FF;
    dw 0xFFFFFFFF
```

Figure 3.2: A minimal picoplugin

in `testprog/flashdump.c` in the subversion repository. This program accesses the instruction memory by using guided commands (more about guided frames and guided commands in chapter 4), which would be an alternative in case using `/dev/mem` should prove problematic. The reason that guided commands were not used to upload the plug-in to the instruction memory is that the current implementation of the control point does not allow sending these guided commands, so an extension to `npcp` would have had to be written to allow this.

3.4 Picoplugin API

In addition to upload the picocode plug-in to the instruction memory, the load command of `NPCtrl` also adds a filter rule to the multi-field classifier so that the plug-in is called for packets matching the rule (figure 3.1 shows what a rule typically looks like.) A minimal plugin (which does absolutely nothing) is shown in figure 3.2; it can be assembled with `npasm -ram -l -v NP4GS3B -d0_128` *plugin.asm*, which will produce a `plugin.ram` file containing the plugin code and a `plugin.lst` file containing a commented source listing. The ram file, stripped of its leading two lines, can be converted into a raw binary image with `xxd -r -p`, the resulting file can then be used by `NPCtrlD`. This is also shown by the `Makefile` in the `Code/sampleplugin` directory (the `buthead` command used in the `Makefile` is available in a package with the same name in Debian; the `xxd` command is part of the vim package.)

The call to the plug-in can be found in the `npdd/pico/src/data` directory of the ASO, around line 800 of `14.asm` (it was inserted at r375 in the subversion repository). The original idea was to use the 'redir' action flag (p. 183 of [7]) to distinguish picocode PromethOS plug-ins from external plug-ins (this would make sense, given that an unset 'redir' flag is interpreted as "process this packet locally" throughout the firmware, but in the end it was easier to just statically allocate the plug-in ID's 0 to 3 for picocode plug-ins and jump into the picocode plug-in in `l4.asm` according to the plug-in id: it turned out that bigger modifications to the NPCtrlD daemon would have been necessary to add a rule with unset 'redir' action flag, and the documentation of the control point API is not very verbose on this topic (chapter 11 of [8]), so that this implementation directly calls the plug-in if the plug-in id is 3 or lower.

The environment available to the plug-ins is described in the Forwarding Picocode Design Reference [7] on pages 70 (general overview), 99ff (Interfaces A3, A5) and 210ff, and particularly 217ff (detailed description of the code surrounding the call to the plug-in). All the corresponding code is contained in the `l4.asm` file in the picocode firmware in `aso-134/src/npdd/pico`.

After return from a plug-in, the value of the `r28` general purpose register is interpreted: if is set to -1, plug-in execution is finished and the packet continues its way through the picocode firmware. Any other value is interpreted as a plug-in id, with values smaller than 4 causing a further picocode plug-in to be called and other values causing the packet to be redirected to the control CPU for processing by a Linux kernel level PromethOS NP plug-in.

3.5 The Way Out – ePPC to Network Code Path

Unfortunately, the code to call the picocode plug-ins is only executed for packets entering the PowerNP from external network interfaces, and not for packets going out from the embedded PowerPC. Due to time constraints, the code to call packets for these packets, too, could not be written, but the code path where this has to happen has been identified and possible solution are proposed below.

As described in section 2.3, the communication between the embedded PowerPC and the packet processors is handled by the GPH threads. The corresponding code is contained in `gph.asm` at `GPH_Transmit_Frame` for packages leaving the ePPC. The packet is read from D6 RAM and written into the egress data store, then immediately enqueued for delivery to the ingress side (line 808 of that file). On the ingress side, the packet is received in `cp_gdh.asm` as a *D205* encapsulated frame, which will bypass the multi-field classifier code which calls the plug-ins. To deliver these packets to the multi-field classifier and thus cause the plug-ins to be acalled, a *D203* frame is required, which has quite a different header and contains only the layer 3 (IP) frame instead of a complete Ethernet packet. These frame formats are described on p. 112ff of the Picocode Design manual [7]; table 3.1 contains an overview which shows that there are not many common header fields between D203 and D205 encapsulation.

Possible solutions to this problem include the three outlined below, of course without claiming that other solutions are not possible or even better.

Plug-ins on Egress Side: Plug-ins could be called directly from the GPH thread in `gph.asm`. This alternative was discarded quickly as it would require the plug-ins

field name	Length	D203	D205	Description
DA	6B	yes	yes	Destination MAC (of the PowerNP), ignored
DA	6B	yes	yes	Source MAC (of CP), ignored
Ethertype	2B	0xD200	0xD200	The Ethertype
Subtype	1B	0x03	0x05	The frame encapsulation type
SP	1B	yes	yes	Source port. Used only for the data wrap port, 64 = ePPC, 65 = PCI host, 0 otherwise.
IP DA	4B	yes		The IP dest. address to use instead of the IP DA in the frame.
IP SA	4B	yes		Only for IP multicast: IP source address
Protocol type	2B	yes		Layer 3 protocol type, as for r1 in @A1
Ingress context	16b	yes		For Layer 4 classification
Layer 4 skip	1b	yes		L4 classifier lookups should be skipped.
BA flag	1b	yes		Perform BA classification in L4
BA number	3b	yes		BA table number
ETB	8b		yes	Encoded Target Blade
TP	8b		yes	Target port. Goes to the TP field in the FCB
OC-48	1b		yes	Is target port an OC-48 link?
LID	21b		yes	LID (meaning varies depending upon frame type.) Usually contains queue index and port number.
FHF	4b		yes	The frame type to build. Refer to the Ux and UJx frame formats.
FCInfo	4b	yes	yes	Flow control information
TTL no Decr	1b	yes		Reserved (?)
Priority	1b		yes	Switch priority
FHE	4B		yes	FHE (meaning depends on frame format)
Data		yes	yes	L3 frame or raw frame, respectively.

Table 3.1: Picocode D203 and D205 frame encapsulation; reserved and padding fields have been omitted.

to handle two completely different interfaces: packets are stored in the egress datastore in the GPH thread, but are stored in the ingress datastore in the multi-field classifier in `l4.asm`, also most of the data structures described in the previous section are absent or totally different.

Rewrite D205 to D203 Packets: Packets entering the ingress side as described above could be converted from D205 to D203 packets by rewriting the encapsulation header and removing the Ethernet packet header from the packet data. This is probably the most realistic approach, since only a small part of the firmware would have to be changed.

Move Packets to Ingress in GPH: A more elegant solution would be to modify the GPH thread to move incoming packets to the ingress datastore instead of the egress datastore and handle them in the same way as packets coming in from the data mover units. Care would have to be taken to re-enqueue these frames to a GDH thread to avoid contention on the GPH thread. While this solution is more elegant than the previous one, it is certainly also much more complex to implement.

3.6 Direct Packet Dispatch

Sometimes a network packet may need to be dispatched from a picocode plug-in directly to the host processor or the control processor[3], bypassing all further handling by the ASO picocode. For this purpose the *redir* plug-in (in `Code/sampleplugin/redir.asm`) was written which directly enqueues the packet after manually constructing the necessary `FCBPage` data structure.

The FCBPage, which is documented on p. 250 of the data sheet [12], is the data structure interpreted by the Enqueuer/Dequeuer/Scheduler units (there is a data structure with this name on both the ingress and the egress side, but the actual contents differ) to decide where a packet is to be sent to by the hardware after it has been processed by the EPC. Since the plug-in processing occurs on the ingress side, an ingress FCBPage is assembled which will direct the packet via the switch interface to the egress side, where it will be processed in `softjump.asm` (the egress processing is directly started at `swj_unicast` by the hardware classifier.) Two new software jumps were added: `promethos_egress_gph` and `promethos_egress_host`, which directly enqueue the packets to the embedded PowerPC or to DMU A (and therefore to the host CPU), respectively.

This is the extent of the current implementation – to actually use this code, the proxy device driver will have to be modified to handle these packets, since they don't match any of the frame formats that driver is prepared to handle.

[3]At the time this code was written the more elegant plug-in intercall API via the return code – now described in 3.4 – did not exist yet.

Chapter 4

Host to PowerPC communication

It was decided quite early that the IBM PowerNP control point (`npcp`) software would be running on the embedded PowerPC and not on the host CPU, so that all possible conflicts between instances of the npcp on a host computer running with multiple PowerNP Application Reference Boards could be totally avoided. Obviously, the control software on the host CPU needs to communicate with the control point on the embedded PowerPC – for this, and for the author to familiarize himself with the PowerNP architecture, a simple test program was developed allowing data exchange between the host CPU and the embedded PowerPC without involving any part of the plug-in architecture.

4.1 Communication Paradigm: Guided Frames

First, a short introduction on the software architecture of the IBM Advanced Software Offering (ASO). After power up, the PowerNP application reference board only runs a minimal firmware which was loaded from flash memory as described above in section 2.5.2. The only way to communicate with the picocode firmware is by using *guided frames*: commands wrapped in special IP frames over UDP port 5555. The format of these guided

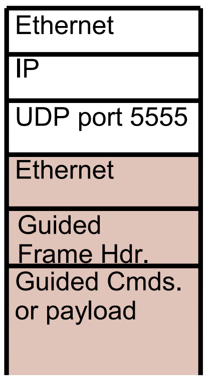


Figure 4.1: Guided commands encapsulated in a guided frame.

commands is shown in figure 4.1 and described in the ASO Control API Programming Reference [6], p. 51ff. These guided frames are run through the functional blocks of the picocode firmware and interpreted; response data and error codes are embedded into the guided frame and the same data is then returned to the originating control point if an acknowledgement was requested (an in depth description of guided frame handling is provided in chapter 9 of [6]). As described below, the fact that the frame is mainly just passed through by the firmware, with commands interpreted where necessary, could be used for a simple yet reliable communication mechanism between host CPU and ePPC.

4.2 Implementation Details

The header fields of a guided frame are described in details on p. 58/59 of the API documentation [6], three fields are of particular interest for the communication path to be introduced here (quoted directly from the IBM documentation):

resp/$\underline{\textbf{req}}$ Response and Not Request Indicator. Differentiates between request (unprocessed) and response guided frames.

ack/$\underline{\textbf{noack}}$ Acknowledgment or No Acknowledgment. Controls whether the GFH picocode acknowledges the guided frame. Guided frames that are not to be acknowledged must not contain any form of guided command that performs a read.

use/$\underline{\textbf{learn}}$ Use or learn the Source Line Card/Source Port (SB/SP) information. The SB/SP information is required to route the frame back to the originating the CP. If the use/learn bit is set to 1, the SB/SP field in the frame is expected to be provided by the originating CP. However, if it is set to 0, these values are deduced by the GFH at the ingress side.

Thus, by setting ‘req’, ‘ack’ and ‘use’, on a guided frame, it is possible to cause the picocode firmware to pass a frame through unchanged – and by specifying the destination address for the frame in the *source* address fields (source blade, source port) – since the ‘ack’ causes the firmware to pass the frame “back to the originator” – it is possible to pass frames between the embedded PowerPC and the host CPU. Luckily, not even the internal frame structure is parsed, so payload data may be inserted directly after the guided frame header, i.e. the packet does not need to adhere to the guided command format.

While the PowerNP hardware – according to the documentation – does support big (9kB) jumbo-frames, at least some parts of the software available for this thesis do not. So, through experimentation, a maximal payload size of 1458 bytes was determined. As described below in section 5.3, some performance testing was done when the program was written. In the current state of the source code repository (r473; see also the software archive content overview in appendix C), the program is in `NP/picocode/Code/testprog` in `bench.c` and `gf.h`. To compile, issue a `make bench` command in that directory or (to compile for the embedded PowerPC platform – see also appendix D) `CC=ppc_405-gcc make bench`. The tool is invoked with either ‘s’ or ‘r’ as its only command line argument to time the sending or receiving of a number of frames (it may be necessary to compile the program intended for the receiving end with a lower number of frames as the picocode firmware tends to drop a few frames during a benchmark run.) Also, note that the control point software must not be started to run this test, just boot the ePPC with the Linux system image (again, appendix D.)

4.3 Two Control Points

The router infrastructure that is supposed to surround this implementation is supposed to consist of a control point (`npcp`) process on the embedded PowerPC processor for each PowerNP board, with the management and communication software running on the host CPU and some interface software on the PowerPCs to interface with the control point being specific to PromethOS NP. Of course, this is not the only way to build a distributed router infrastructure with the PowerNP platform, and IBM equipped the Advanced Software Offerings bundle with the possibility of attaching more than one control point to a single PowerNP chip – to quote from the IBM ASO API Reference manual [6], on p. 70:

> This API is used to set the physical port parameters for a secondary CP port. A secondary CP is a CP attached to the Network Processor using an Ethernet connection, but it is not the CP that is used to load the operational picocode on the Network Processor.

But, as mentioned at the start of the chapter, to avoid conflicts between multiple control point instances, and to avoid complicating the software set up even more, it was decided that running just the one control point in the default configuration and using custom glue code was preferable to running several control points.

Chapter 5

Evaluation

This chapter summarizes what has been achieved in this project. The structure of this chapter follows the structure given in section 1.3 in the introduction.

5.1 Plug-in Loader

The original firmware provided in IBM's advanced software offerings (ASO) does not provide facilities to load packet processor code at run-time – the firmware is loaded once, at control point start up, and is not modified after that. As described in sections 3.2 and 3.3, an extension was written to allow plug-ins to be loaded by software running on the embedded PowerPC of the PowerNP.

The IBM firmware takes most of the available space in the instruction memory for the packet processors, so that only 4k Instructions are available to store plug-ins; this has arbitrarily been divided into 4 slots of 1k Instructions each, so that the current plugin loader can load upto 4 plug-ins simultaneously.

5.2 Plug-in Integration

Packet processing in the picocode firmware needs to be altered so that the plug-ins are called to process data packets. This has been done for packets originating from the network directed to other network interfaces or the embedded PowerPC. Traffic originating on the ePPC is not currently processed by the plug-in architecture, the reasons and possible solutions are given in section 3.5.

The environment available during plug-in execution is described in section 3.4, as is the API which allows plug-ins to specify that a network packet needs further processing in another plug-in, either on the packet processor, the control CPU or, using a slightly different method (3.6) on the host CPU. The implementation of the latter functionality does, however, only exist on the packet processor side, the driver implementation in the Linux kernel does not currently handle arriving packets.

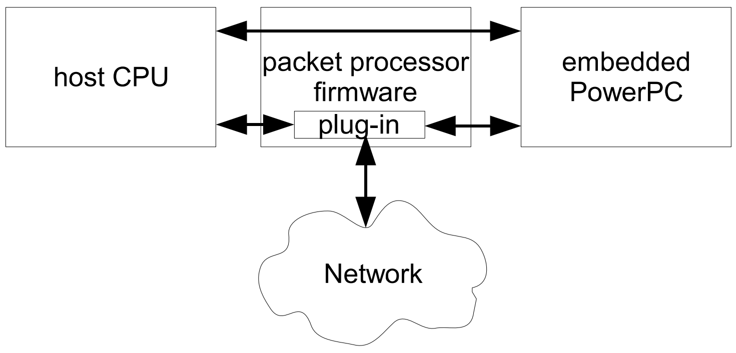


Figure 5.1: Communication paths in the Promethos NP architecture

5.3 Communication Infrastructure

All the possible communication paths on a router node with the PowerNP processing are shown in figure 5.1[1] The communication channels between the plug-ins and the other components in the system have been described above, this section describes the direct communication between the host CPU and the control CPU.

A simple way to exchange packets between the host CPU and the embedded PowerPC has been found, and is even usable with the default IBM ASO firmware – so there is no need to rebuild the boot firmware of the application reference board, which according to Pascal Erni is not trivial (section 4.4.4 of [3]). The implementation of this communication path is described in chapter 4.

Communication bandwidth between the host CPU and the embedded PowerPC is rather low: measurments show a rate of less than 100Mbps (or ca. 6300 packets per second), which is also confirmed by the work of Pascal Erni [3] and Michel Dänzer [1]. The path from the host CPU to the packet processors is certainly not the problem: the PowerNP's Data Mover Unit which interfaces to the PCI connector of the board has a capacity of 1Gbps. The bottleneck is in the GPH thread which handles communication between the packet processors and the ePPC: regular data traffic (running from ingress DMU to egress DMU and not to the embedded PowerPC) can use any of the 32 threads of the EPC, but there is only one GPH thread for each direction.

While this performance may seem disappointing, it should be pointed out that the embedded PowerPC in this network processor is a PPC405 running at 133MHz, with only 64MB RAM, so any nontrivial data processing or bookkeeping would probably soon severely test the limits of that architecture.

The source code of the benchmark program can be found in the subversion repository at `Code/testprog/gf.c` in r271, or in the HEAD revision in the same directory as `bench.c`; an example run is shown in figure 5.2. The author found it necessary to use a lower packet count on the receiving side, since some packets get dropped when the PowerNP is flooded.

[1]The case of multiple PowerNP processor boards in a single host computer is not substantially different: traffic between the PowerNP boards is routed by the host CPU.

```
# gf s
Sending...
50000 packets of 1458 bytes sent in 7896ms
this means: 9232346.240592bps at 6332.199068pps
# gf s
Sending...
50000 packets of 1458 bytes sent in 7925ms
this means: 9198094.013605bps at 6308.706457pps
# gf s
Sending...
50000 packets of 1458 bytes sent in 7911ms
this means: 9214551.154031bps at 6319.993933pps
# gf s
Sending...
50000 packets of 1458 bytes sent in 7895ms
this means: 9233231.425816bps at 6332.806191pps
```

Figure 5.2: Performance measurements between control CPU and host CPU.

The current implementation of this communication channel uses the `npctl0` interface of the procy device driver exclusively, so no control point software can be run in parallel. Either modification of the proxy device driver or of the control point software will be necessary to handle operation of the control point in parallel with the use of this communication channel.

Chapter 6

Conclusion and the Road Ahead

To get started with this last chapter, let's first look back at the original task assignment (here translated into English, the original German version can be seen in appendix A):

> In this diploma thesis, a framework for the packet processors shall be developed enabling their dynamic programming with PicoPlugins. Also, the framework should implement a programmable distributor [16] to dispatch network packets to the various processors in the system.

The following section will look back and relate this thesis to the task assignment. The rest of the chapter will then describe how the work done in this project could be further improved.

6.1 Summary

Looking back at the past 4 months, most of the goals of the task assignment could be implemented, albeit sometimes only as proof-of-concept code. For the biggest missing piece, plug-in invocation on network traffic originating on the embedded PowerPC, it could be shown where exactly further modification is necessary. The most important conclusion for the author is that, even with the extensive documentation that was available, the complex architecture required a lot of time until the architecture was understood enough, so that any actual work could be done. With the switch to 6 months-long master thesis instead of the 4 months diploma thesis, this problem should have a less heavy impact on the work of future students.

6.2 The Future

As mentioned above, it was not possible to implement a full production-ready network router, so there is much room for improvements in many areas. One obvious project is writing actual plug-ins to make use of the possibility to load pico-plugins. Other projects include:

Traffic from ePPC An implementation of one of the proposed solutions of section 3.5 is required to handle traffic originating from the embedded PowerPC and thus make the plug-in framework fully usable.

Extend Linux Kernel Driver The network packets directed to the Linux kernel running on the host or control CPU as described in section 3.6 need to be handled and passed on to a software component to be processed.

Evaluate Plugin Size The free area of the instruction memory was arbitrarily divided into space for 4 plug-ins. This choice needs to be reevaluated by looking at plug-in implementations and playing the possibility to load more plug-ins against the possibility to load bigger plug-ins. Additionally, it may be possible to omit or replace parts of the IBM firmware to free instruction memory space for more plug-ins.

Configuration Data for Plug-ins The current plug-in framework does only load plug-in code into the instruction memory and run this code on network packets. No way is provided to supply configuration data to these plug-ins. This could be achieved by reserving parts of the D6 memory for this purpose or by building data structures in the tree search engine.

Appendix A

Task Assignment

The next three pages contain the original task assignment for this diploma thesis.

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Institut für
Technische Informatik und
Kommunikationsnetze

Sommersemester 2004

Diplomarbeit
für
Adrian von Bidder

Betreuer: Lukas Ruf

Ausgabe: 05.04.2004
Abgabe: 04.08.2004

A Framework For Programming Packet Processors

1 Einführung

Moderne Router setzen zur Steigerung der Verarbeitungskapazität einen oder mehrere Netzwerkprozessoren (NP) [2, 3, 4] pro Netzwerk Port ein. Diese Netzwerkprozessoren können durch Programmcode programmiert werden. Eine Fähigkeit, welche die NPs für den Einsatz in Aktiven Netzen (AN) [7] prädestiniert. NPs bestehen üblicherweise aus einem General Purpose Processor Core und mehreren spezialisierten Paket Prozessoren, welche bei IBM Core Language Processors, bei Intel MicroEngines genannt werden. Sie stellen somit ein Chip-Multi-Processor System dar, welches die Aufgaben entsprechend der zu lösenden Komplexität auf die verschiedenen Prozessoren verteilt.
Die Programmierung von Paket Prozessoren ist nicht trivial, setzt sie doch ein sehr profundes Verständnis der Prozessor-Architektur voraus. Um die Programmierung von Services zu vereinfachen, wurde im Rahmen von PromethOS NP [6] das Konzept der *PicoPlugins* eingeführt, welches es gestatten soll, Service Komponenten nach einem einheitlichten Konzept auch auf den Paket Prozessoren zu installieren. Eine Uebersicht der Gesamtarchitektur wird in [5] präsentiert.

2 Aufgaben: A Framework To Program Packet Processors

Im Rahmen dieser Diplomarbeit soll ein Framework für die Packet Processors entwickelt werden, welches deren dynamische Programmierung nach dem Konzept der PicoPlugins ermöglichen soll. Das Framework soll zudem in der Lage sein, Pakete nach dem Konzept der Programmable Distributors [5] an verschiedene Prozessoren zu dispatchen.

3 Vorgehen

- Richten Sie sich eine Entwicklungsumgebung (GNU Tools) unter Linux ein.
- Machen Sie sich vertraut mit den Unterlagen (Dokumentation und Source Code) zum PromethOS NP Framework und dem Netzwerkprozessor.

- Erstellen Sie einen Zeitplan, in welchem Sie die von Ihnen zu erreichenden Meilensteine Ihrer Arbeit identifizieren.
- Entwickeln Sie eine Architektur für das zu erstellende Packet Processor Framework.
- Implementieren Sie Ihre Architektur.
- Verifizieren, evaluieren und demonstrieren Sie das Erreichte durch eine Beispielapplikation.
- Dokumentieren Sie die Resultate ausführlich.

Auf eine klare und ausführliche Dokumentation wird besonders Wert gelegt. Es wird empfohlen, diese laufend nachzuführen und insbesondere die entwickelten Konzepte und untersuchten Varianten vor dem definitiven Variantenentscheid ausführlich schriftlich festzuhalten.

4 Organisatorische Hinweise

- Am Ende der zweiten Woche ist ein Zeitplan für den Ablauf der Arbeit vorzulegen und mit dem Betreuer abzustimmen.
- Mit dem Betreuer sind regelmässige, zumindest wöchentliche Sitzungen zu vereinbaren. In diesen Sitzungen sollen die Studenten mündlich über den Fortgang der Arbeit und die Einhaltung des Zeitplanes berichten und anstehende Probleme diskutieren.
- Am Ende des ersten Monates muss eine Vorabversion des Inhaltsverzeichnis zur Dokumentation dem Betreuer abgegeben und mit diesem besprochen werden.
- Nach der Hälfte der Arbeitsdauer soll ein kurzer mündlicher Zwischenbericht abgegeben werden, der über den Stand der Arbeit Auskunft gibt. Dieser Zwischenbericht besteht aus einer viertelstündigen, mündlichen Darlegung der bisherigen Schritte und des weiteren Vorgehens gegenüber Professor Plattner.
- Am Schluss der Arbeit muss eine Präsentation von **20 Minuten** im Fachgruppen- oder Institutsrahmen gegeben werden. Anschliessend an die Schlusspräsentation soll die Arbeit Interessierten praktisch vorgeführt werden.
- Die Arbeit muss regelmässig auf dem PromethOS SVN-Server <`https://svn.promethos.org:8443/svn/PromethOS`> gesichert werden. Es ist darauf zu achten, dass die **richtige Branch** verwendet wird.
- Bereits vorhandene Software kann übernommen und gegebenenfalls angepasst werden.
- Die Dokumentation ist mit dem Satzsystem LaTeX zu erstellen. Illustrationen müssen mit einem OpenSource Programm unter Linux erstellt werden.
- Es ist ein mit Bindespiralen gebundener Schlussbericht über die geleisteten Arbeit abzuliefern (4 Exemplare). Dieser Bericht besteht aus einer Zusammenfassung , einer Einleitung, einer Analyse von verwandten und verwendeten Arbeiten, sowie einer vollständigen Beschreibung der Konfiguration von den eingesetzten Programmen. Der Bericht ist in Deutsch oder Englisch zu halten. Die Zusammenfassung muss in Deutsch und Englisch verfasst werden.
- Die Arbeit muss auf CDROM archiviert abgegeben werden. Stellen Sie sicher, dass alle Programme sowie die Dokumentation sowohl in der lauffähigen, resp. druckbaren Version als auch im Quellformat vorhanden, lesbar und verwendbar sind.
  Mit Hilfe der abgegebenen Dokumentation muss der entwickelte Code zu einem ausführbaren Programm erneut übersetzt und eingesetzt werden können.
- Diese Arbeit steht unter der GNU General Public License (GNU GPL) [1].
- Für die Arbeit werden jedoch Informationen eingesetzt, welche nur durch das Unterzeichnen eines NDA (Non-Disclosure Agreement) mit IBM Corp. erhalten wurden. Die Arbeit als ganzes (Programmcode und Dokumentation) sowie alle Informationen, die unter das NDA fallen, dürfen nur an Dritte weitergegeben werden, wenn eine schriftliche Einwilligung von IBM Corp. vorliegt. Falls NDA-Informationen auf einem anderen Rechner als <`www.promethos.org`> gespeichert werden, muss sichergestellt werden, dass kein unberechtigter Zugriff auf diese möglich ist.
- Diese Arbeit wird als Diplomarbeit an der ETH Zürich durchgeführt. Es gelten die Bestimmungen hinsichtlich Kopier- und Verwertungsrechte der ETH Zürich.

Literatur

[1] GNU General Public License v2. http://www.gnu.org/copyleft/gpl.html, June 1991.

[2] IBM Corp. IBM PowerNP NP4GS3 Databook. http://www.ibm.com, 2002.

[3] Intel Corp. Intel IXP1200 network processor – datasheet. http://www.intel.com, 2000.

[4] Intel Corp. Intel IXP2xxx hardware reference manual. http://www.intel.com, 2003.

[5] L. Ruf, R. Keller, and B. Plattner. A Scalable High-performance Router Platform Supporting Dynamic Service Extensibility On Network and Host Processors. In *Proc. of 2004 ACS/IEEE Int. Conf. on Pervasive Services (ICPS'2004), Beirut, Lebanon*. IEEE, Jul. 2004.

[6] L. Ruf, R. Pletka, P. Erni, P. Droz, and B. Plattner. Towards High-performance Active Networking. In *Proc. of 5th Annual Int. Working Conf. on Active Networking (IWAN), Kyoto, Japan*, number 2982 in Lecture Notes in Computer Science. Springer Verlag, Heidelberg, Dec. 2003.

[7] D. Tennenhouse, J. Smith, W. Sincoskie, D. Wetherall, and G. Minden. A Survey of Active Network Research. *IEEE Communications*, January, 1997.

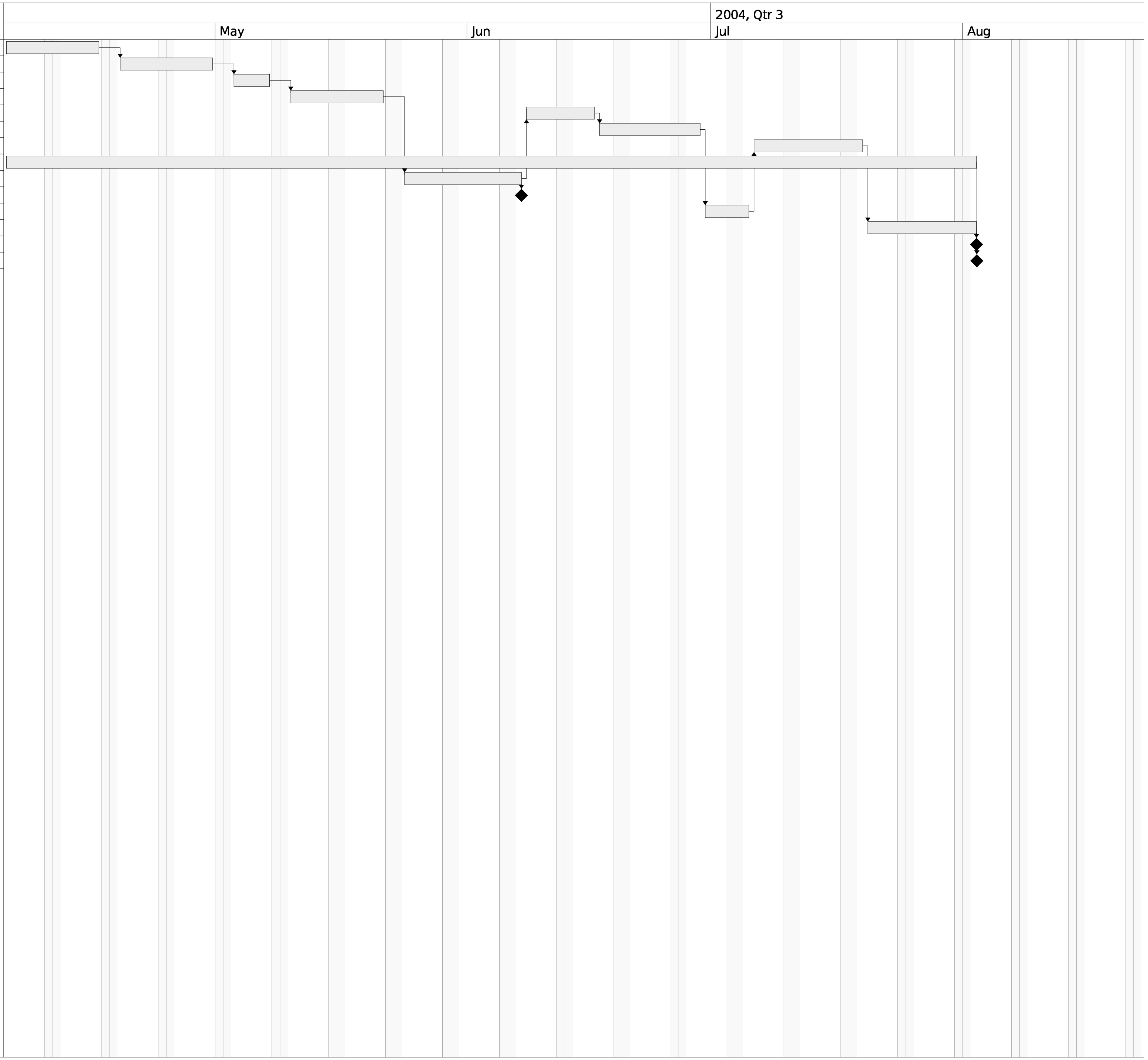
Zürich, den 05.04.2004

Appendix B

Schedule

The next page contains the original project schedule. The diploma thesis started on April 5th 2004, and ended on August 4th of the same year.

Name	Work
requirements analysis, familiarize with environment	10d
set up application reference board	10d
draft design communication framework	5d
communication framework implementation	10d
draft design of plugin loader	7d
proof-of-concept implementation of plugin loader	9d
test, fix bugs & evaluate	10d
continuously document progress	86d
documentation, intermediate presentation	11d
PRESENTATION I	
plugin loader documentation	4d
finish docs, fix bugs	10d
PRESENTATION II	
FINISHED	



Appendix C

Software Archive Contents

The *subversion*[1] version management software was used to store the software used in this project, and references to specific versions of files are made in some places in this document. Consequently, to continue the work started in this project, it is recommended to coninue to use this repository – for access permissions, the supervisor of this project, Lukas Ruf, should be contacted at `<lukas.ruf@promethos.org>`. A snapshot of the latest version of the software is also available in a tar file; again, contact Lukas Ruf for access. Both the tar file and the subversion repository have the same directory layout. A number of additional subdirectories were omitted, especially all directories in the Linux kernel tree and most subdirecotires of the ASO code. All files which were changed in this project are listed in *`monospace italics`*.

```
+-- NP
    |-- copernicus_drv
    |      The copernicus driver (Broadcom gigabit Ethernet)
    |-- hardhat
    |   +-- devkit
    |       +-- lsp
    |           +-- ibm-np4gs3-ppc_405
    |               +-- linux-2.4.17_mvl21
    |      The Linux kernel for the embedded PowerPC Linux
    +-- picocode
        |  Everything that was changed in this project is contained in this directory
        |-- Code
        |   |-- aso-134
        |   |   +-- src
        |  The IBM Advanced Software Offerings source code
        |   |       |-- NPCtrl NPCtrl.{c,h}
        |  The client program of the plugin loader
        |   |       |-- cabdd
```

[1] `http://subversion.tigris.org/`

```
|   The CAB driver API
|   |         |-- linux
|   |         |   +-- em405
|   |         |       +-- sonic
|   |         |           +-- utils
|   The d6load utility
|   |         |-- npdd
|   |         |   '-- pico src/data/{l4.asm,softjump.asm}
|   The picocode firmware
|   |         +-- wrapper
|   |             +-- linux
|   |                 +-- NPCtrlD NPCtrlD.{c,h}, NPCtrlMsg.h
|   The plugin loader daemon
|   |-- hardhat
|   |   +-- devkit
|   |       +-- ppc
|   |           +-- 405
|   |               +-- bin
|   The MontaVista PowerPC cross development tools (executables)
|   |-- sampleplugin all files
|   The sample picocode plugin, the packet dispatcher
|   +-- testprog all files
|   Test programs: bench, flashdump
|-- Confdata all files
|   The load.sh helper script; configuration files used in the project
+-- Docu all files except 'ibmdoc'
    +-- ibmdoc
    IBM ASO documentation (pdf) from IBM
```

Appendix D

Tools Setup

This chapter describes how to set up the entire PowerNP development environment, starting with a standard Debian GNU/Linux PC on an Intel based computer. While it is fairly detailed, it does not intend to describe each keystroke.

D.1 Tools on Debian GNU/Linux

There are two different sets of tools to be installed: the tools provided by IBM for Picocode development and the MontaVista Hardhat Linux cross compiler for the embedded PowerPC.

D.1.1 PowerPC 405 cross compiler environment

MontaVista provides a pair of complete Linux distributions for embedded software development on PowerPC 405-based systems; one (Red Hat-based) system for the host system and one for the embedded platform, mainly intended to be used for systems able to mount their root file system via NFS. In this thesis, however, only the cross development tool chain was needed,

In theory, the `NP/hardhat` directory in the subversion server contains the necessary executables and support files in the `devkit/ppc/405/bin` subdirectory. However, using this toolchain for anything that requires linking with libc (i.e. anything except compiling the Linux kernel) showed that a lot of the header and object files necessary to compile programs which link to the C library are missing. The necessary parts of the `hardhat` directory structure were therefore duplicated in `NP/picocode/Code/hardhat` and supplemented with the missing files from the original MontaVista ISO images[1]. Thus, to use the cross compiling environment, `NP/picocode/Code/hardhat/devkit/ppc/405/bin` must be added to the `$PATH` environment variable.

[1] `host-mvl3.0.0.iso` and `ppc_405-mvl3.0.0.iso` from the PromethOS web page with SHA1 sums `e70b0a8450cd3176f28a48436f3adb479e34a819` and `e30b52cb45bd75a1ea334cd7e06ce94717782402`.

D.1.2 IBM Advanced Software Offerings

The Advanced Software Offerings (ASO) version 3.1.0 used in this thesis is contained in the `ibm.restricted.red_hat72.npbkit310-linux24-i386.tgz`[2] file on the PromethOS web server at `https://www.promethos.net/IBM/Restricted/` which is an archive containing the ASO itself and its dependencies as RPM packages.

The following is a description how the IBM ASO bundle was set up on the author's Debian GNU/Linux installation. The software bundle was originally intended to be used on Red Hat Linux 7.2, so if the instructions below should fail[3], using that environment and setting up the software according to the instructions in the `README.TXT` in the tar file should work. But because the author prefers Debian and because Red Hat 7.2 is an ancient system by today's standard (especially since modern graphics and network adapter hardware and SATA disk controllers are likely not to be supported at all), this was not tested.

Before the installation, make sure the following Debian packages are installed: *alien* (version 8.44 was used by the author), *gcc* , *make*, *debhelper* and *dpkg-dev.* The *fakeroot* package is useful, too; alternatively `alien` can be called from the root account. Then, the RPM files can be converted into Debian packages:

```
avbidder@papillon:~/tmp$ fakeroot alien blt-2.4u-1.i386.rpm
blt_2.4u-2_i386.deb generated
avbidder@papillon:~/tmp$ fakeroot alien Tcl-8.3.4-1.i386.rpm
tcl_8.3.4-2_i386.deb generated
avbidder@papillon:~/tmp$ fakeroot alien Tk-8.3.4-1.i386.rpm
tk_8.3.4-2_i386.deb generated
avbidder@papillon:~/tmp$ fakeroot alien IBMnpbkit-3.1.0-0158.i386.rpm
ibmnpbkit_3.1.0-159_i386.deb generated
avbidder@papillon:~/tmp$ ls
IBMnpbkit-3.1.0-0158.i386.rpm  blt_2.4u-2_i386.deb
README.TXT                     ibmnpbkit_3.1.0-159_i386.deb
Tcl-8.3.4-1.i386.rpm           tcl_8.3.4-2_i386.deb
Tk-8.3.4-1.i386.rpm            tk_8.3.4-2_i386.deb
blt-2.4u-1.i386.rpm
avbidder@papillon:~/tmp$
```

These .deb files can then be installed as usual, though it may be necessary to either remove all installed Tcl/Tk based software from the system or to ignore conflicts and dependencies (and risk that the other Tcl/Tk based programs stop working) – or (if only the picocode assembler is needed and not the `npsim` and `npscope` tools), only install the `ibmnpbkit_3.1.0-159_i386.deb` package and ignore the Tcl/Tk and *blt* requirements. In

[2]SHA1 checksum `b90e92b9a318358c5063320480358b922d4a6fdd`

[3]A lot of software from the Debian testing and unstable distributions was used, so the author's installation presented a moving target. It is therefore, unfortunately, conceivable that the software installation may fail on different Debian installations.

any case, the IBM ASO seems not to work with the Tcl/Tk and *blt* packages that come with Debian. All files are installed in `/opt/nptools`.

In addition to the software installation, it is also necessary to set some environment variables. This is also explained in the `README.TXT` file which was already mentioned above. The author added the following section to his ~/`.bash_profile`; users of other shells will need to adapt this, obviously.

```
if [ -d /opt/nptools ]; then
    export NPTOOLS_HOME=/opt/nptools
    export NPTOOLS_USER_HOME=/home/avbidder/.nptools
    export NPASM_OPTS="-q -co"
    export TCL_LIBRARY=$NPTOOLS_HOME/tcl8.3.4/bin
    export TK_LIBRARY=$TCL_LIBRARY
    export LD_LIBRARY_PATH=$NPTOOLS_HOME/tcl8.3.4/lib:$NPTOOLS_HOME/lib
    export MANPATH=:$MANPATH:$NPTOOLS_HOME/tcl8.3.4/man
    PATH=$PATH:/opt/nptools/bin
fi
```

D.2 Drivers for the Application Reference Board

To access the PowerNP board from the host system, the *copernicus* driver (to access the Broadcom gigabit interface) and the *proxy device driver*need to be loaded. Please see the next section on how to compile the proxy device driver; this section explains how to get the copernicus driver.

The source code of the copernicus driver resides in `NP/copernicus_drv` in the subversion repository. To compile, it expects the configured Linux kernel code in either `/usr/src/linux` or `/lib/modules/‘uname -r‘/build`. In this thesis, a 2.4 kernel was used (the author used a 2.4.26 kernel, Michel Dänzer used 2.4.17-mvl21-promethos; any 2.4 version should work.), 2.6 kernels were not tested and will probably not work. A simple `make` then builds the copernicus.o kernel module. It is important that a gcc 2.95 version is used to compile the kernel and the drivers; the source code contains some non-standard C constructs which are not supported anymore in newer versions. For this, either install the *gcc* Debian package from Debian 3.0 (this worked on the author's machine – it may not work on newer Debian installations because of dependency problems), include a symlink to gcc-2.95 in a directory early in the `$PATH`, or change the `/usr/bin/gcc` symlink to point to the correct compiler (using `dpkg-divert` to make sure the change is not overwritten on software updates.)

`NP/picocode/Confdata` contains the `load.sh` shell script which loads the copernicus driver, the proxy device driver and initialises the `npctl0` pseudo Ethernet interface which is the primary interface to the PowerNP processor. It initialises network interfaces to the IP addresses 3.3.3.3 and 4.4.4.4[4] which are hard coded into some of the tools.

[4]Which are assigned to General Electric and Level 3 communications, respectively. Why IBM did not

D.3 Modifying and Compiling the ASO

The important parts of the advanced software offerings from IBM, as modified by Pascal Erni [3] and in this thesis, are the PowerNP control point daemon (`npcp`), the proxy device driver, the `npcp` control daemon and its control program (`NPCtrlD` and `NPCtrl`) and the picocode firmware. The complete ASO bundle is contained in `NP/picocode/Code/aso-134` (or without the author's modifications, in `NP/aso-134`.)

As explained in the previous section, gcc 2.95 must be used; additionally, the control point software includes some components written in C++, so g++ 2.95.x needs to be available too. The software kit can be compiled either for the host platform or for the Linux system on the embedded PPC405 (see below). The configuration is contained in `Linux-debian.mk` (compilation for the host PC) and `Linux-sonicehh.mk` (compilation for the embedded Linux platform) in `Code/aso-134/src`; you will need to set the `BASE` variable in these files to match your directory layout[5]. To activate a configuration, execute either `./configure debian` or `./configure sonicehh` and compile the software with `make`. The picocode firmware needs to be compiled separately by executing `make np3b` in the `src/npdd/pico` directory. The `d6load` utility in `linux/em405/sonic/utils/` also needs manual compilation, with just `make`. The final executables can be found in `exe/`***`platform`*** (the executables) and `obj/`***`platform`***`/npdd/pico/link` (the picocode firmware), respectively; ***`platform`*** is either `i386-debian-linux` or `ppc405-hardhat-linux`.

The sample picocode plug-ins, contained in `NP/picocode/Code/sampleplugin`, are closely related to this software bundle. A simple `make` command compiles the *redirect* plug-in, a trivial change in the `Makefile` will enable compilation of the *settl* plug-in. More information about the plug-in infrastructure is contained in the main part of this thesis in chapter 3

D.4 Linux on the embedded PowerNP CPU

As described in the main part of this document, plug-ins are loaded from the Linux system running on the embedded PowerPC. An adapted Linux kernel which runs on the PowerNP board is stored in `NP/hardhat/devkit/lsp/ibm-np4gs3-ppc_405/linux-2.4.17_mvl21` in the subversion repository. The PowerPC Linux root file system must be present in the kernel directory in the ram disk at `arch/ppc/boot/images/ramdisk.image.gz`, the ramdisk used in this project is provided in the `Confdata` directory. With the kernel configuration in `config-2_4_17_mvl21` (rename this to `.config` in the same directory), a bootable kernel image which includes the ram disk can be built with `make zImage.initrd`, and the resulting `arch/ppc/boot/images/zvmlinux.initrd.vxboot` file can be loaded onto the PowerPC with the `d6load` tool.

The `ramdisk.image` contains a minimal Linux system on a 8M ext2 file system image.

chose IP addresses from the RFC 1918 private IP ranges is a mystery to the author.

[5]Michel Dänzer did some more work on the build system to eliminate this in [1]. This unfortunately didn't get merged into this tree.

All system binaries are provided by the *busybox* system[6]. To edit this file system, it can be mounted on the host Linux system with the *loop* option to mount: `mount -o loop ramdisk.image /mnt` (after uncompressing it with gzip.) After making the changes and compressing it again, regenerate the boot image as described above and load it with `d6load`.

After boot, the Linux console is accessible via the serial port of the application reference board[7] (the necessary serial cable was provided by Lukas Ruf; ordering information can be found in the technical description of the application reference board [11] in section 1.2.) After boot up, networking can be started by loading the `pci_rethmod.o` driver module and starting the control point software (the `npcp` program) which will load the picocode firmware from `/etc/npcp.d` into the packet processors' instruction memory (note that the npcp program needs his helper binaries in the current working directory, so be sure to change to `/usr/bin` beforehand.) To use the plug-in loader, the `NPCtrlD` daemon must be started and can then be controlled with the `NPCtrl` program. Please read the previous section on how to modify and compile these components.

Please note that there are several issues that need to be taken into consideration when working with the Linux system on this PowerPC processor – there was unfortunately not enough time to trace these to their roots.

Initial boot problems In some cases, `d6load` fails after a cold boot. Retrying a second time may succeed, or doing a warm boot may be necessary. Unfortunately no systematic pattern for this behaviour could ever be established.

Does only boot once After the control point has been started and has loaded the firmware onto the PowerNP (from the host CPU or from the Linux on the embedded PowerPC), (re)booting the ePPC with `d6load` does not work anymore, so the host computer needs to be restarted.

"Ping of death" From the PPC Linux, after the control point has been started to activate network, ping does only work for a few packages – after approx. 6 packets, the PowerPC kernel will crash (which will manifest itself either by causing a segmentation fault or illegal instruction error in the ping program or by turning the kernel belly-up in a kernel panic.)

[6]`http://www.busybox.net/`

[7]The author prefers the `ser2net` program to access the serial port via telnet, but any serial console emulator will do.

Bibliography

[1] M. Dänzer. Resource Controlled Interprocessor Communication On NP-based Active Network Nodes. DA 2004-17, ETH Zürich, Switzerland, Jul. 2004.

[2] S.E. Deering. *RFC–1112: Host extensions for IP multicasting*, August 1989.

[3] P. Erni. *Einsatz und Programmierung des IBM NP4GS3 Netzwerkprozessors für Aktive Netzwerkknoten unter Linux.* Computer Engineering and Networks Laboratory (TIK), Mar. 2003.

[4] J. Heinanen and R. Guerin. *RFC–2697: A Single Rate Three Color Marker*, September 1999.

[5] J. Heinanen and R. Guerin. *RFC–2698: A Two Rate Three Color Marker*, September 1999.

[6] IBM Corp. IBM PowerNP advanced software offering control application programming interfaces reference. http://www.ibm.com, Sep. 2002.

[7] IBM Corp. IBM PowerNP advanced software offering forwarding picocode design reference. http://www.ibm.com, Sep. 2002.

[8] IBM Corp. IBM PowerNP application programming interfaces reference. http://www.ibm.com, Sep. 2002.

[9] IBM Corp. IBM PowerNP assembler language programmer's guide and opcode summary. http://www.ibm.com, Dec. 2002.

[10] IBM Corp. IBM PowerNP NP4GS3 databook. `http://www.ibm.com`, 2002.

[11] IBM Corp. IBM PowerNP NP4GS3C and NP2G application reference boards technical description. http://www.ibm.com, Sep. 2002.

[12] IBM Corp. IBM PowerNP NP4GS3. http://www.ibm.com, Jan. 2003.

[13] IBM Corp. IBM PowerNP NPGS3 hardware reference manual. http://www.ibm.com, Jan. 2003.

[14] Motorola Inc. C5-DCP. http://www.mot.com, 2000.

[15] Intel Corp. Intel IXP1200 Network Processor – Datasheet. `http://www.intel.com`, 2000.

[16] R. Keller, L. Ruf, A. Guindehi, and B. Plattner. PromethOS: A Dynamically Extensible Router Architecture Supporting Explicit Routing. In *Proc. of 4th Annual Int. Working Conf. on Active Networking (IWAN), Zürich, Switzerland*, number 2546 in Lecture Notes in Computer Science. Springer Verlag, Heidelberg, Dec. 2002.

[17] J. Postel. Internet Protocol Specification. RFC 791, ISI, Sept. 1981.

[18] L. Ruf, R. Keller, and B. Plattner. A Scalable High-performance Router Platform Supporting Dynamic Service Extensibility On Network and Host Processors. In *Proc. of 2004 ACS/IEEE Int. Conf. on Pervasive Services (ICPS'2004), Beirut, Lebanon.* IEEE, Jul. 2004.

[19] Niraj Shah and Kurt Keutzer. Network processors: Origin of species. In *Proceedings of ISCIS XVII, The Seventeenth International Symposium on Computer and Information Sciences*, October 2002.

[20] Silicon Software System. Application Reference Board for the IBM PowerNP NP4GS3 Network Processor User Manual. `http://www.s3group.com`, 2002.

```
:wq
```