



## **Leveraging PPL to reduce Time-to-Market for Network Processor based Applications**

**January, 2005**

Copyright © IP Fabrics, Inc. 2005

## Abstract

*PPL is a high-level, functional language that is very easily grasped due to its packet processing focused syntax and semantics. Programs written in PPL<sup>1</sup> are interpreted by the PPL Virtual Machine to run on a target network processing machine, like the IXP2850 or IXP2350. This allows network engineers to develop, deploy and maintain complex logic rapidly, without noticeable performance degradation or the need to know target machine specifics. The end result is not only shorter development time, but also reduced software lifecycle costs.*

## Introduction

Increasingly, to accommodate attributes like performance, flexibility and extensibility, networking and communication equipment makers are embracing network processors as opposed to ASICs or general-purpose processors. However, the very strength of network processors — being a "soft" solution via software — is also the key challenge in deploying network processors.

A typical network processor (NPU) has many parallel low-level RISC-type processors that need to be programmed by the system builder. Typically these processors have one or more types of low-level, software-managed interconnect, specialized instruction-set architecture, heterogeneous memory arrangements, limited instruction space, and no operating system support in the fast-path interconnects. As a result, development effort can be several orders of magnitude greater than what is required for general-purpose processor. Although NPU manufacturers provide support for C language coding, in addition to assembly language, NPU programs written in C tend to look much like assembly code because the program needs to deal with hardware specifics.

To overcome these issues, a different approach is needed that (a) enables rapid prototyping for proof-of-concept designs, and also (b) reduces product development cycles.

## Need for Functional Packet Processing Language

One alternative to writing low-level, machine-dependent code (be it assembler or C) would be to devise a high-level functional language for expressing a wide variety of packet processing applications. The primitives of this language would include fundamental network processing capabilities like tracking connections, removing an outer header, translating IP addresses, encrypting a packet, scanning the payload for a regular expression, and so on. Such high-level of abstraction enhances productivity and enforces re-use.

Functional languages require that programmers specify functionalities, not the dependencies between them. The actual parallelism extraction is automated, by first sequencing the functionality based on dependencies and then optimally mapping the workload to the NPU underneath.

---

<sup>1</sup> PPL is a trademark of IP Fabrics

One approach to accomplish this is to develop a virtual machine to interpret the application logic and a compiler to translate programs written in the language into the virtual machine representation. This approach abstracts the network processor, allowing the application developer to focus his or her attention on packet processing. The strengths of the virtual machine approach are not completely free, and the obvious tradeoff would be performance. However, the performance penalty is surprisingly small or virtually non-existent, because the virtual machine capitalizes on optimizations in the NPU's parallel processing environment.

PPL, which stands for Packet Processing Language, is a very high-level, functional programming language for describing the types of packet processing found in many of today's networking applications. PPL contains functionality to process layer 3 IP packets, specific protocols at layer 4 (e.g., TCP and UDP), and is highly optimized for "deep" packet processing at layers 5-7. It has many "built-in" algorithms/state machines oriented toward complex packet processing applications such as encryption, authentication, content inspection, stateless and stateful firewall filtering, detection of intrusions and denial-of-service attacks, layer 7 filtering, traffic management, and content-based load balancing.

### A quick peek into PPL

A PPL program consists of a set of *rules* consisting of expressions and *actions*. Rule expressions are evaluated in parallel and the actions of true rules are then executed sequentially and/or in concurrent groups. Actions range from simple (forward or drop the packet) to more complex actions (encrypt, encapsulate), that are performed by routines called *policies*. Rules are grouped as *events* to (a) imply processing of a group of rules and (b) to control concurrency in PPL. The code segment below illustrates a simple bidirectional NAT with static address assignment, as defined in RFC 2663. Note that actions due to incoming and outgoing events execute concurrently, provided the corresponding rules evaluate to true.

```
Event(incoming)
. . .
Rule EQ(IP_DEST,199.100.73.15) SET(IP_DEST,200.0.0.100)
Rule EQ(IP_DEST,199.100.73.16) SET(IP_DEST,200.0.0.101)
Rule EQ(IP_DEST,199.100.73.17) SET(IP_DEST,200.0.0.102)
Rule EQ(IP_DEST,199.100.73.18) SET(IP_DEST,200.0.0.103)

Event(outgoing)
. . .
Rule EQ(IP_SOURCE,200.0.0.100) SET(IP_SOURCE,199.100.73.15)
Rule EQ(IP_SOURCE,200.0.0.101) SET(IP_SOURCE,199.100.73.16)
Rule EQ(IP_SOURCE,200.0.0.102) SET(IP_SOURCE,199.100.73.17)
Rule EQ(IP_SOURCE,200.0.0.103) SET(IP_SOURCE,199.100.73.18)
```

The language supports policies to examine information in the content part of the packet, create/use associatively searched tables, create/manage groups of packets, configure crypto engines and calculate flow-rates. These policies are useful for applications like virus detection, intrusion prevention, content-based load balancing, network security, traffic shaping, policing, active or passive monitoring and many other applications.

If there is a need from within a PPL program to communicate directly with other applications running in adjoining microengines or elsewhere in the system, a policy called

PROGRAM can be used (i.e. generating alerts for control/management software or invoking “secret sauce” customer written data-plane microcode).

### The role of PPL Virtual Machine

The PPL Virtual Machine interprets application logic written in PPL to run on the target network processors, which are parallel machines. Like all virtual machine implementations, there is a performance penalty. This penalty is managed by hiding memory-access latency, performing data-path optimizations, reducing inter-thread communication and keeping the virtual machine overhead low.

To illustrate, the figure below shows an implementation of such a virtual machine<sup>2</sup> upon the 16 microengines of an Intel IXP28x0 NPU ([1]). Each block in the figure represents a micro-engine and divisions within a block represent threads of the micro-engine. Packets and control information are passed between blocks using 1) a scratch ring, 2) the Next Neighbor ring, or 3) SRAM “bins”. There are also various structures stored in DRAM, SRAM, and scratch memory that are generated by the PPL compiler and/or allocated dynamically by the system application.

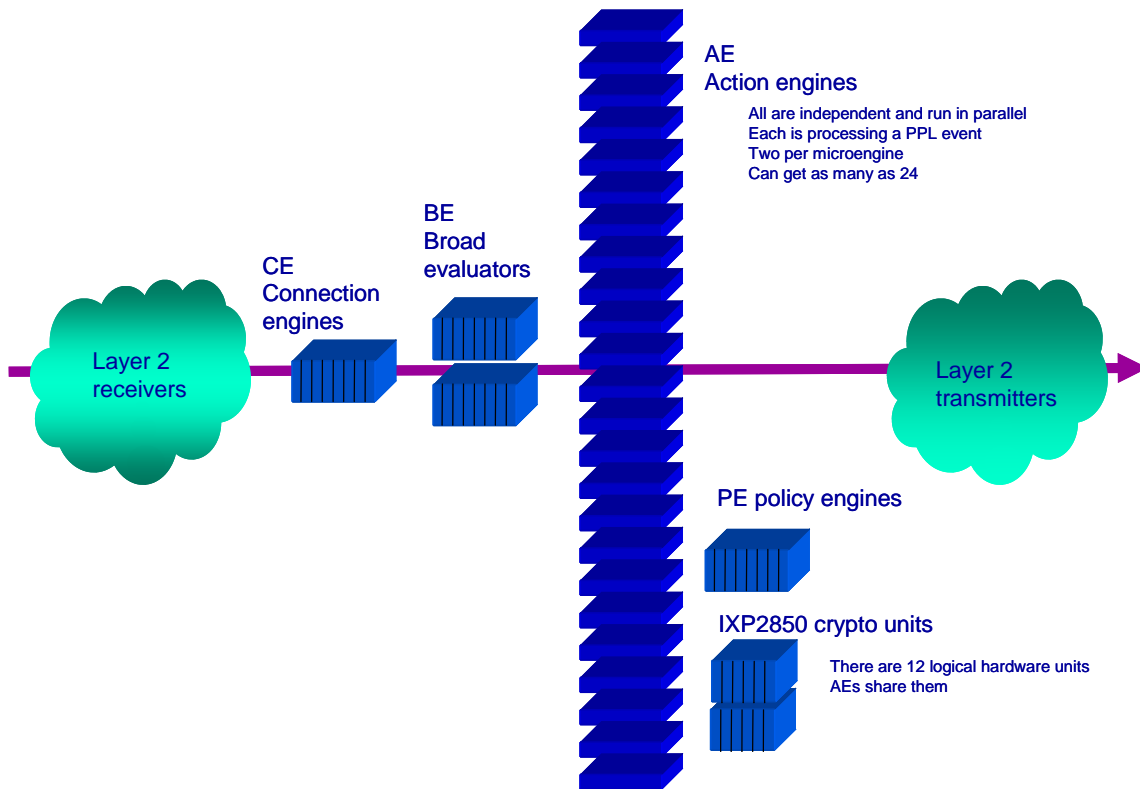


Figure 1. PPL Virtual Machine

Besides receive and transmit, there are three stages in the pipeline:

<sup>2</sup> PPL Virtual Machine configuration is specified in a special file called the devicemap file.

## Leveraging PPL to reduce TTM for Network Processor Based Applications

1. **Connection Engine (CE):** This stage performs lookups of state & event information.
2. **Evaluation Stage (BE):** This stage is responsible for identifying a specific range of rules to be executed based on the event value passed in by the SL stage. This stage also does a certain amount of “pre-qualification” on the rules within the event in an attempt to further reduce the number of true rules to be executed on the packet.
3. **Action Execution (AE):** This stage evaluates expressions and performs actions for the event based on the list of true rules provided it by the evaluation stage.

Packet flow starts with the receiver and moves progressively to the right, with a packet ultimately being dropped, transmitted, passed to the XScale or queued on another event for later processing. Scratch ring interfaces pass packets and control messages between the PPL virtual machine and XScale core software.

### Summary

It is easier to model, debug and modify packet processing logic, written in a functional language like PPL, than in a low-level language. The optimization burden is shifted to the virtual machine which hides the performance penalty by adopting strategies that are known to work well in parallel programming environments. Using the PPL virtual machine developers can prototype new applications rapidly, assess architectural bottlenecks before investing significant software development resources, reduce schedule risk and accelerate completion of product milestones. The result is accelerated time to market and lower software life-cycle costs for network processor applications.

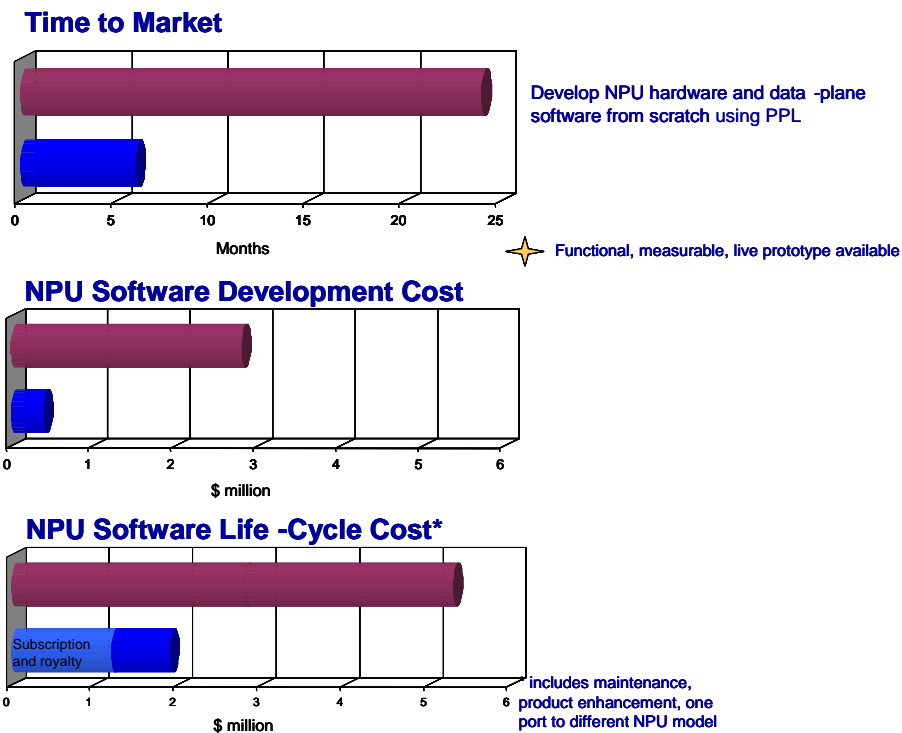


Figure 2. TTM and Software Life-Cycle Cost<sup>3</sup>

<sup>3</sup> Data based on past project experience

## References

- [1] <http://www.intel.com/design/network/products/npfamily/ixp2850.html>