PPL Packet Processing Language and Virtual Machine

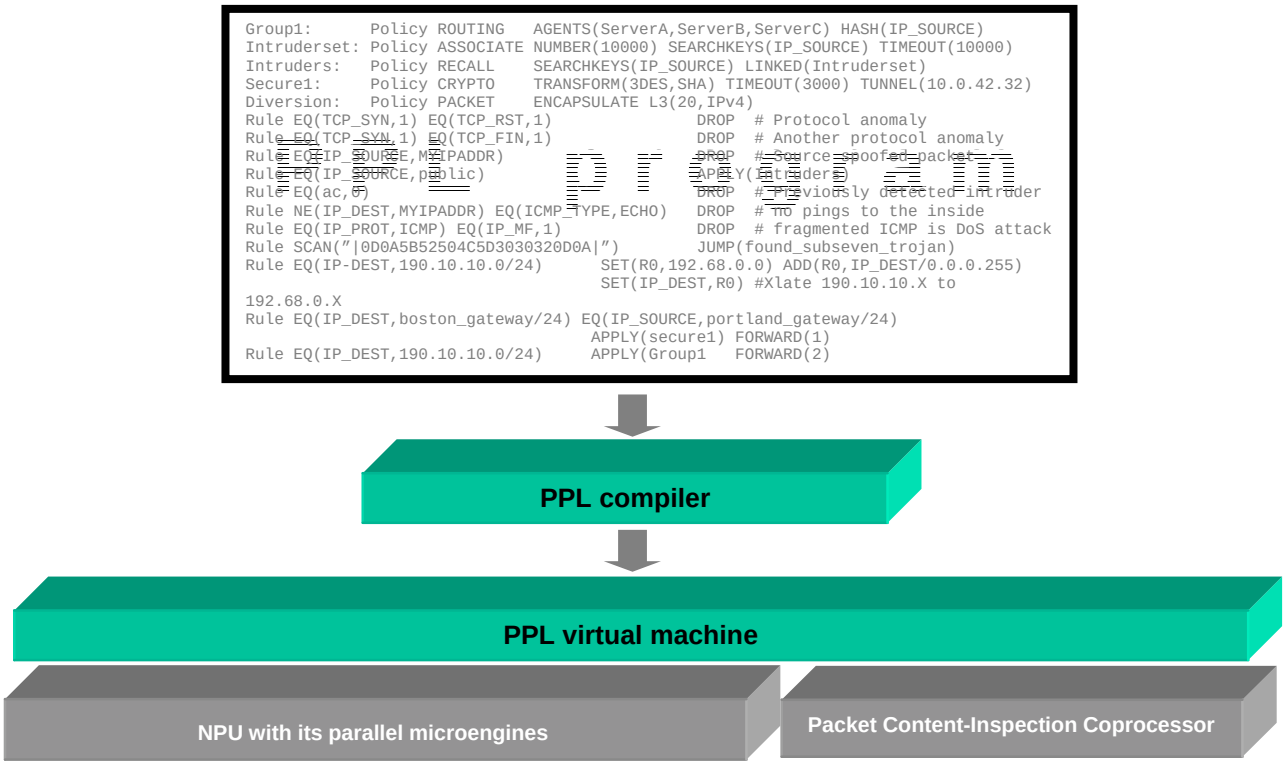
Short Overview

October 27, 2003

PPL is a very-high-level language for describing the processing of network packets. Although it could apply in concept to any type of packet, the language is oriented toward layer 3 IP packets, toward specific protocols at layer 4 (e.g., TCP and UDP), and toward "deep" packet processing at layer 7. In most senses, PPL is a *functional* language as opposed to a procedural language such as C. PPL also defines explicitly several concepts of concurrent processing.

In addition to being applicable to broad types of packet processing, PPL contains specific features oriented toward 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.

PPL is intended to be used in an implementation with one or more network processors with high-speed packet-classification capabilities, and as such it represents a very-high-level language for writing concurrent data-plane software or microcode. Although PPL could be compiled in the traditional sense to the low-level code of these network processors, an implementation that is generally more effective is compiling it to a software virtual machine atop the network processor(s). IP Fabrics has implemented PPL for the Intel IXP2000 NPU family via the virtual-machine approach with the assist of a Packet Content Inspection Coprocessor, as shown below.

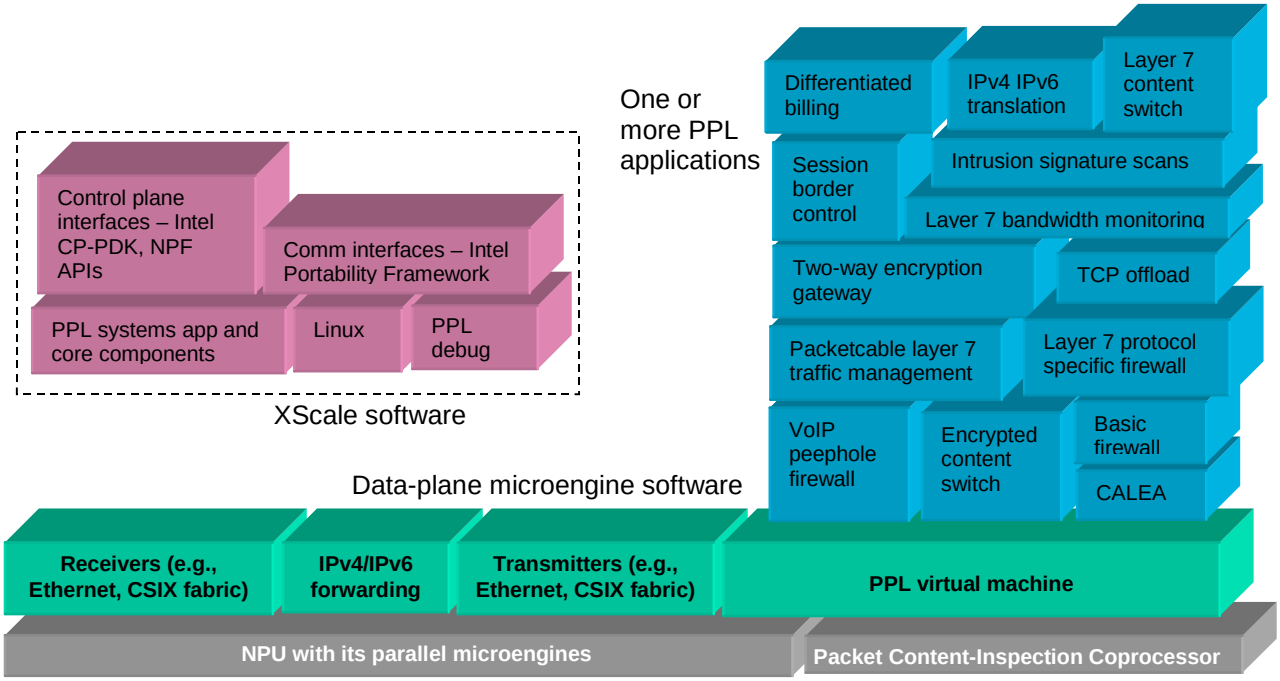


IP Fabrics' product set includes

- PPL compiler
- The run-time virtual machine implemented across the microengines of the NPU
- The remainder of the data-plane environment (e.g., receive, transmit, forwarding micro-blocks)
- PCIC coprocessor
- Interfaces to the Intel Portability Framework

- XScale software for logging, PPL debug, and NPF and Intel CP PDK interfaces
- Networking applications developed in PPL

For the Intel IXP2000 family of network processors, the solution provided is summarized below.



PPL Objectives

As a language, we had several key objectives in mind for PPL. Understanding these objectives is helpful in understanding the language and its implementation.

Productive — PPL provides the means to develop high-performance NPU applications in a *tenth or* less of the time and cost it would otherwise take. As an example, the "bump in the wire" example in Comer's text *Network Systems Design using Network Processors* requires 570 lines of code using a combination of C and microassembler language, requires 76 lines of FPL code for Agere's NPU, but requires only 3 lines of PPL code.

Powerful — "Primitives" in PPL are complex networking operations. Such things as encrypting a packet, removing an outer header, tracking a TCP connection, and scanning for a regular expression are operations in the language.

Robust — Banes of networking software are system hang-ups and security holes. PPL's high-level primitives, protection mechanisms, and functional nature create far fewer opportunities for subtle bugs. Its implicit protocol-dependent checks on packets eliminate a common source of security holes. One of our favorite examples is the Comer's text example mentioned above; the PPL code is correct where the C code and Agere FPL code contain a serious flaw (they would behave incorrectly given a fragmented packet).

Self-contained	It is possible to write 100% of an NPU application (the data-plane part) in PPL and avoid other languages, learning a new suite of tools, and reading literally thousands of pages of manuals on the NPU.
Open	On the other hand, we realize a PPL application needs to have mechanisms to interface with control-plane software and important standards such as the Network Processing Forum's API, and also that one may want to write 90% of his application in PPL and 10% in a lower-level form.
High performance	In spite of all of the above, PPL programs need to provide performance comparable to other alternatives. We believe this is the case and this complex subject is left for the final section of this overview.
Intrinsically parallel	Most NPUs consist of a set of highly parallel engines, which is one of the many things that makes writing software for them very difficult. PPL embeds several types of parallelism directly in the language.
Dynamic	A major dilemma is that networking equipment needs to stay up 100% of the time, yet it needs to amenable to software changes to add new functions, detect new security risks, etc. PPL is designed to be dynamically compiled - that is, a PPL program may be changed on the fly.
Portable	PPL is not tied to any specific NPU model or even architecture family.

Language Basics

PPL is comprised of *rules*, *events*, and *policies*. A rule is a statement of one or more actions to be performed if a set of conditions are true. PPL defines the evaluation of rules to be concurrent. An event is a description of the set of rules that apply to a particular stimulus. The most-common stimulus is the arrival of a packet on a particular network port, but events can also be triggered by programs, including programs outside of the domain of PPL (in the Intel IXP implementation, this uses the mechanisms in the Intel IXA Portability Framework). Policies are extensions to rules in the sense that one or more of the actions of a rule might be to apply a particular policy.

At the core of PPL, then, is the concept of a rule. A rule basically says "make these tests, and if all tests are true, take these actions." It has the format

Rule *expression expression … action action …*

For instance, a simple rule might be

```
Rule   NE(IP_DEST,sys_ipaddr) EQ(IP_PROT,TCP) APPLY(secure1)
```

which says – if the current packet is an IP packet and the destination IP address is not equal to some value called sys_ipaddr and the layer-4 protocol is TCP, apply a policy called secure1. The above rule is written to be oblivious to whether the packet is IPv4 or IPv6, and thus it works for both. It is also possible to distinguish between IPv4 and IPv6 as needed.

Values in rules can be a variety of things, including literal numbers, names of fields in the packet, predefined state values, and registers (short-term values). A value can also have a mask applied to it, such as sys_ipaddr/24 (24 leading one's). Actions in rules are such things as dropping a packet, applying a policy, forwarding a packet, some simple arithmetic operations, and some complex operations such as hashing, computing a CRC, and converting a character IPv4/IPv6 address to a binary one.

Policies are broad, often complex, actions on or using a packet. They are an adjunct to rules, because specific rules decide to apply specific policies. For example, the first policy below describes a particular mode of IPSec processing and the second describes a set of agents to which a packet might be rerouted and the means for selecting which.

```
Secure1: Policy CRYPTO TRANSFORM(3DES,SHA) TUNNEL(my_VIP,172.18.12.0)
                       TIMEOUT(one_hour)
Group1:   Policy ROUTING AGENTS(ServerA,ServerB,ServerC) HASH(IP_SOURCE)
```

Other policies in PPL are related to tracking connections, managing packets other than the current packet, creating packets, using a general associative table structure, and interfacing to outside programs.

The third basic construct is the event. Rules are grouped as events, such as

```
Event(1,2)  # Rules to be processed for incoming packets on ports 1 and 2
Rule EQ(TCP_SYNONLY,1) APPLY(TCPconnectionrate) FORWARD STOP
Rule                   LOG DROP
```

Events serve two purposes. First, they identify the occurrences that cause rules to be processed. The most typical circumstance is the arrival of a packet into the system, but events can also be associated with system startup, exceptions, and internal program communications (from PPL or from another program in the system). Second, as discussed in the next section, events represent a level of concurrency in PPL; rules associated with multiple events can be processed concurrently, and even multiple occurrences of the same event can be processed concurrently unless the program indicates otherwise.

Parallelism

A major attribute of PPL is concurrency. In general it is more appropriate to think of all of the rules in a PPL program as being processed concurrently than to think of the rules as being executed sequentially. However, as we will show, there are ways to have coarse- and fine-grained control over concurrency. There are four separate concepts of concurrency:

1. Concurrent evaluation of expressions in rules
2. Concurrent run groups (smaller designated groups of rules that can be processed simultaneously with other groups within an event).[1]
3. Different events running concurrently (multiple sections of a PPL program that process asynchronously and concurrently)

[1] Run groups are not discussed further in this overview; refer to the full PPL specification.

4. Multiple concurrent instances of the same event

With the exception of something called a deferred expression, the expressions of all rules are evaluated concurrently and then the actions are performed sequentially. When the PCIC coprocessor is used, the concurrent expression evaluation is literally true.

Events provide another layer of concurrency. As noted earlier, a typical use of an event is to associate a set of rules to the arrival of a packet on a particular port. For instance, if one's system was a "bump in the wire," it would typically have two ports, and one would structure one's PPL program into rules that handle incoming traffic on each port. This would be represented by two events. In PPL each event runs concurrently with all other events, and thus in this example there are two parallel "threads of execution." In addition, the PPL program can specify whether concurrent instances of a *single* event can occur.

Layer 7 Inspection

An important aspect of PPL that is worth highlighting is its mechanisms to examine information in the content part of a packet. These can be used for a variety of purposes, for example, detecting viruses, locating intrusion signatures, scrubbing application protocols for DoS attacks, and using packet content for purposes of load balancing.

One capability is represented by the SCAN expression, which examines the content part of a packet for a string. SCAN can express a simple character or hexadecimal string, or it can also bring the power of regular expressions to bear. A simple example of SCAN is

```
Rule SCAN(”|0D0A5B52504C5D3030320D0A|”) LOG(found_subseven_trojan)
```

SCAN can express a character string, and this string can be a regular expression. For instance, suppose we wish to examine the payload of each packet going to TCP port 80 to see if it is a GET HTTP transaction with a URL ending with redirect.html and containing a session cookie. The PPL rule would be

```
Rule SCAN(re”GET.*?redirect.html\s.*?HTTP/1.*?Cookie:”,0,0)
     EQ(IP_PROT,TCP) EQ(IP_DPORT,80)                    APPLY(group1)
```

Because SCAN is an expression, it follows the rules of concurrency in PPL, meaning that it is intended to be done concurrently with all other expression evaluations (including other SCANs).[2]

Examples

To highlight some other areas of PPL, we have included a few examples below. For instance, references to certain layer 3 and 4 protocol fields generate implicit expressions to avoid creating programs with basic security holes, so, for instance, saying

[2] The initial implementation of PPL on the Intel IXP2xxx NPU family uses an auxiliary coprocessor (PCIC) capable of performing concurrently hundreds of string scans as well as other expression evaluations.

```
Rule EQ(TCP_SYN,1) APPLY(x)
```

is identical to saying

```
Rule EQ(TCP_SYN,1) EQ(IP_VERSION,IPv4) GE(IP_HDRLEN,5) EQ(PS_NOORFIRSTFRAG,1)
                   GE(PS_IPDATASIZE,20) EQ(IP_PROT,TCP)
                   APPLY(x)
Rule EQ(TCP_SYN,1) EQ(IP_VERSION,IPv6) EQ(PS_NOORFIRSTFRAG,1)
                   GE(PS_IPDATASIZE,20) EQ(IP_PROT,TCP)
                   APPLY(x)
```

If we wish to maintain a table that tracks unique combinations of IP addresses and ports on packets that have passed through. We could say

```
Sourcedestcombos: Policy ASSOCIATE NUMBER(100000)
                         SEARCHKEYS(IP_SOURCE,L4_SPORT,IP_DEST,IP_DPORT)
. . .
Rule . . . APPLY(Sourcedestcombos) FORWARD(out_the_other_side)
```

Here the rule will create a unique entry in the table each time the values of these four fields in the current packet don't match a current entry. If the entry already existed, no change would be made.

The CONNECTIONS policy provides a means to track flows of related packets. This includes tracking TCP connections, splicing them, and proxying them. The following PPL program tracks TCP connections in a "bump in the wire" device where agents on either side are allowed to initiate a connection.

```
Cxtable: Policy CONNECTIONS NUMBER(100000) EVENTS(incoming,outgoing)
                      TIMEOUT(one_min) CLOSEWAIT(one_sec) FINALWAIT(one_sec)
Event(incoming)
Rule EQ(TCP_SYNONLY,1) EQ(CX_STATE,NONE) APPLY(Cxtable)
Rule EQ(TCP_SYNONLY,1) NE(CX_STATE,NONE) NE(CX_STATE,CONNECTING) DROP
      # A SYN by itself with no connection creates one in the connecting
      # state.  A SYN when we're already in the connecting state means a
      # simultaneous connect and is just forwarded.  A SYN in other than the
      # none or connecting state is wrong and is dropped.
Rule EQ(TCP_SYNACK,1) NE(CX_STATE,CONNECTING)   DROP
Rule EQ(TCP_SYNACK,1) EQ(CX_STATE,CONNECTING)   SET(CX_STATE,ESTABLISHED)
      # ACKONLY's in the ESTABLISHED or CLOSING states are fine and
      # just fall through
Rule EQ(TCP_ACKONLY,1) EQ(CX_STATE,FINALCLOSE)  SET(CX_STATE,NONE)
Rule EQ(TCP_ACKONLY,1) EQ(CX_STATE,CONNECTING)  DROP
Rule EQ(TCP_ACKONLY,1) EQ(CX_STATE,NONE)        DROP
Rule EQ(TCP_RST,1)                              SET(CX_STATE,NONE)
Rule EQ(TCP_FIN,1) EQ(CX_STATE,CLOSING)         SET(CX_STATE,FINALCLOSE)
Rule EQ(TCP_FIN,1) EQ(CX_STATE,ESTABLISHED)     SET(CX_STATE,CLOSING)

. . .
Event(outgoing)
   # place exactly the same code here
```

Here is another example that would entail a large number of statements in a language such as C but is a single rule in PPL. We are processing some layer-7 protocol that embeds IP addresses in character form. We wish to search for something that appears to be an IPv6 address in the packet payload and convert it to a binary IPv6 address. It accounts for the IPv6 "::" zero-compression notation and leaves a 128-bit binary IPv6 address in Rr0q.

```
Define nums = "[0-9A-F]{1,4}?"
Define CONVERT="0"
Rule SCAN(reul"(nums:){7,7}?nums |nums(:nums){0,6}?:: |::(nums:){0,6}?nums |
nums(:nums){0,5}?::(nums:){0,5}?nums ")
     NE((Rr0,FuF)) SET(Rr0q,Rr0) COMPUTE(CONVERT,Rr0q)
```

Mapping to NPU(s)

Although PPL is a language that is independent of a specific processor model or architecture, there are relationships that need to be expressed between PPL and a specific hardware implementation underneath of processing units, memories, physical network ports, etc. This is accomplished by the PPL DeviceMap statement, whose form is implementation specific. The format of the DeviceMap for the Intel IXP 2000 family allows you to specify such things as NPU model, memory configuration, whether any memory or microengines need to be reserved, network interfaces, switch-fabric interfaces, external program interfaces, and PPL debugging controls. This allows one to describe a full NPU environment in PPL without having to learn and use what is otherwise a large set of tools and procedures.

PCIC Packet Content-Inspection Coprocessor

A general characteristic of a typical PPL program is that it may contain thousands of rules but, for any given packet, only a handful of rules evaluate true and thus typically only a few actions are performed per packet. So an important aspect in making PPL fast is evaluating the rules very rapidly on the arrival of a packet. One aspect of making this fast, far faster than one could even do it with hand-optimized microcode, is to use a specialized processor.

PCIC is a highly parallel (within itself) coprocessor that handles the bulk of the PPL rules evaluation, including string searches and regular expressions. PCIC can compare 128 bits of a packet (320 bits if IPv6) and current state to several lists of values with masks, and in parallel with this can scan the payload for a large number of strings, and then "AND" the results such that it reports a list of true rules. In other words, given a packet, the PCIC will rapidly say, for example, that rules 41-44 and 46 in event 7 are true. So where concurrent rule evaluation in PPL is conceptual, PCIC actually implements it as a highly concurrent operation.

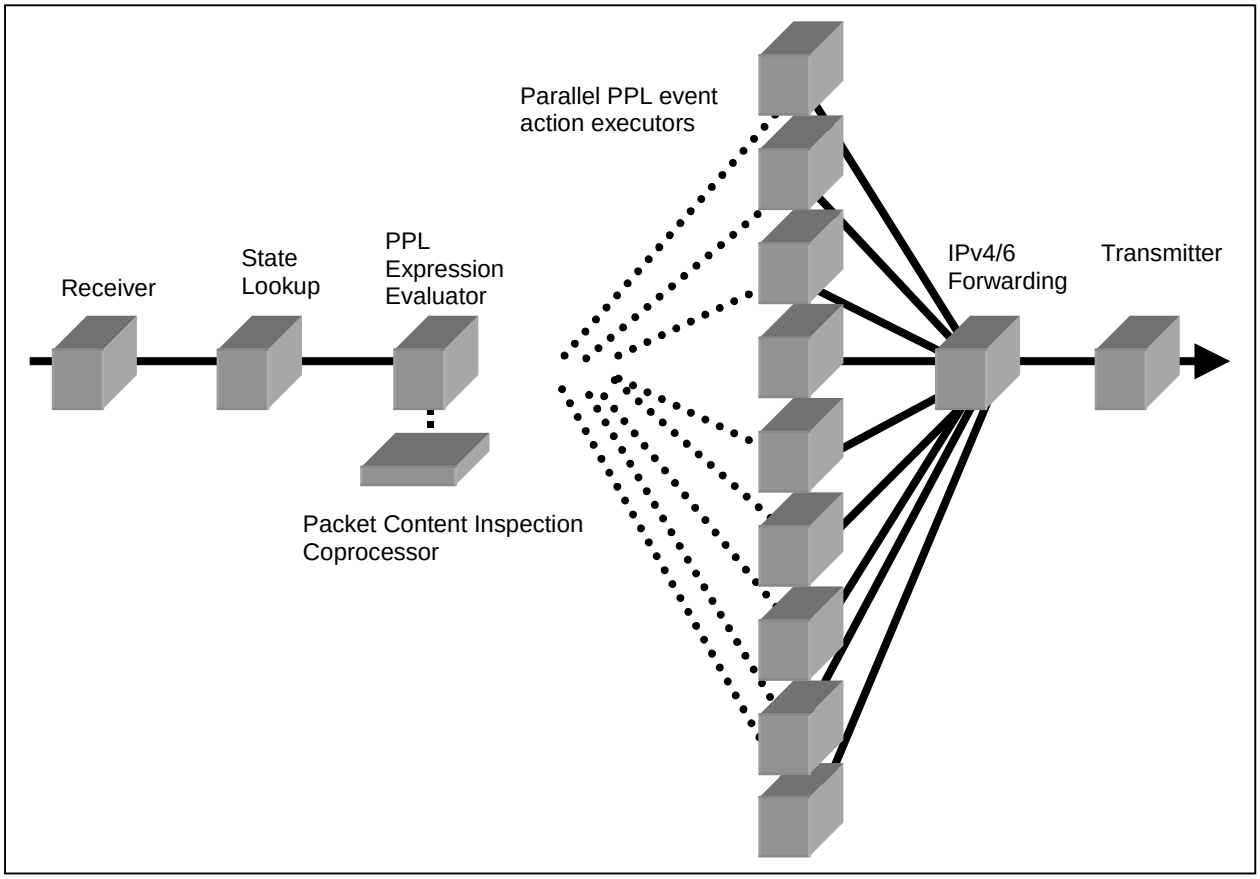
In FGPA form, the PCIC can support about 4000 PPL rules and 150 strings. String scanning performance is capped at about a 1 Gb/s throughput rate (although if an event has no rules containing scans, then this limitation does not apply). Rule evaluation capacity can be traded off with capacity. Assuming an average packet size of 256 bytes, the FPGA form of PCIC can evaluate 4096 rules at an aggregate network rate of 1.5 Gb/s, or 1024 rules at an aggregate network rate of 6 Gb/s. An ASIC version of PCIC is anticipated in the future, and this will significantly increase the capacity and the scan/span bandwidth rate.

PCIC is similar in some ways to the few commercial classification or inspection chips on the market, although PCIC is more powerful in the following respects:

- Can handle IPv6 (some commercial classifiers don't)
- Can classify on more than just fields in the packet. For instance, PCIC can use the connection state, the event, and complex relationships among some packet fields
- Can classify against more than masked constants (e.g., against run-time values)
- Not just scans for strings, but locates them (important for layer-7 protocol processing)
- Accumulates results into a list of true rules
- Higher packet-per-second classification rate
- Far shorter latency than some string-searching classifiers

PPL Performance

The power of PPL has a certain price, and the obvious guess is performance. However, because of the nature of PPL and the innovative implementation of the virtual machine, the performance penalty is typically small. First, unlike, say, a Java virtual machine running on a Pentium, the PPL virtual machine on an NPU is designed as both a pipelined engine (very much the same way one would design a high-performance CPU) as well as one with true parallelism, as shown in the following figure.



Depending on the number of microengines available (16 in the IXP2800 and IXP2850), the virtual machine can simultaneously be receiving new packets from network ports, doing state lookup (e.g., finding connection-table entries) on one or more earlier packets, evaluating rules on one or more packets, processing true rules on multiple packets, doing next-hop lookups on another set of packets, and transmitting yet an earlier set.

As described previously, a major part of the processing is the evaluation of expressions in the rules, and the majority of this is handled by the PCIC coprocessor. Therefore the PPL virtual machine does not have the structure of a software interpreter that proceeds sequentially through a stream of instructions or commands. All of the expressions in PPL rules are evaluated in parallel and in advance of the "execution" of the program for a particular packet. When the program is ready to be processed, the action executor is presented with a list of actions to be performed and the data for those actions. So the typical "execution" of a PPL program on behalf of a packet is a list saying "do this, this, and this, and then end."

Several other key aspects of the virtual machine's performance are

- Overhead is minimal because many of the actions in the program represent complex functions that are invoked (e.g., encrypt this packet and encapsulate it as a tunnel-mode packet, forward the packet, pick an IP destination address having the fewest TCP connections currently established, add an entry to this connection table.
- A significant design consideration in an NPU is the memory/processing speed differential. In the IXP2800/2850, the average read time from memory is 150 to 300 cycles, where the instruction execution time is usually one cycle. So one can execute, on just one of the processors, several hundred instructions in the time it takes to do one memory read. One consequence is that performance is largely a function of how much data is moved between processor and memory per packet. Another is the importance of "Nth degree" optimization of the memory accesses, which has been done extensively in the virtual machine.
- PPL contains several escapes when one wants to interface to an external program (e.g., such a program could be a microblock (in Intel's terminology) that was written independent of PPL to optimize a procedural algorithm that is difficult or impossible to express in PPL.

To compare PPL to alternatives, we need to determine whether the alternative is just a programmed NPU, or a programmed NPU with classification logic (e.g., TCAMs, network search engines or a full-scale packet inspector). Compared to the alternative of a programmed NPU, we believe that for applications which represent a good match to PPL, the PPL/PCIC-based solution approach will show a net aggregate throughput (e.g., packets per second) of 50-100% higher. The other case is where the programmed NPU has some external hardware assist similar to PCIC. Our belief is that for what PPL is designed for (IP-based processing) and on the Intel IXP NPU family, the application expressed in PPL will always outperform the typical attempt to write the application from scratch in C, and almost always beat the typical attempt to write in microassembler. Comparing PPL to writing an application that is carefully optimized for the microengine architecture, for the best use of multiple microengines, and for the best memory performance, PPL is expected to rival the performance of these approaches, while providing huge benefits in cost, development time, and extensibility.