

Overview of IP Fabrics' PPL Language and Virtual Machine

Glen Myers, IP Fabrics

June 6, 2006

PPL (Packet Processing Language) is a very-high-level language for describing the processing of network packets, with the intention (but not requirement) that it be implemented on a multi-core network processor, such as the Intel IXP2xxx family. The primary goal of PPL is to hide the details and complexities of the underlying processors such that the development time of speed-critical or data-plane networking applications can be measured in days rather than months or years.

In approaching PPL for the first time, there are six fundamentally important things to keep in mind, particularly when comparing it to conventional programming languages like C and C++:

1. PPL is applications focused. Where programming languages try to provide abstractions of the underlying machine instruction set, PPL provides representations of things networking applications do. Therefore in PPL you see, as elements of the language, such concepts as packets, encryption, queues, signature searching, and virtual networks.
2. PPL is packet centric. The fundamental data structure in PPL is a packet. Many of the "operators" in PPL perform operations on a packet. PPL provides strong type checking on packets. For instance, it is impossible to refer beyond the extent of a packet, refer to arbitrary memory as a packet, do an IP operation on a non-IP packet, or refer to an IPv6 address in an IPv4 packet.
3. Policies are the work horses of the language. As we shall show shortly, rules and policies are two statement types in PPL. Rules make decisions and apply policies. Policies are where most of the work typically gets done. A PPL policy is a major function. A few examples of the functions that can be performed by policies are encrypting a packet's payload, adding a header to or stripping a header from a packet, creating a new packet, managing a set of packet queues, searching a packet against a large database of signatures, and sending a packet to another PPL program or a program in

a different processor, such as an adjoining Pentium processor.

4. PPL is more of a functional language than a procedural language. PPL has no fixed concept of single-threaded, sequential execution, a concept of procedural languages. While completely abstracting away any parallelism in the underlying processor, PPL provides several concepts of natural concurrency, one being that the arrival of a packet creates a parallel instance of the PPL program.
5. PPL is architecture independent. PPL completely hides the details and nature of the underlying processor. As such, it provides scalability, because the same PPL program will run on a different model of the same processor family. It also provides the opportunity of portability to completely different processor types.
6. Finally, the implementation of PPL is not just a language, but a complete subsystem. For instance, the virtual machine implementation on Intel's IXP family doesn't just process the language; it contains such pre-built things as Ethernet transmitters and receivers, default IP forwarding, and Linux-based control-processor support, allowing one to install the product, write a simple PPL program, and run it on live networking hardware all in the same day.

Language Basics

PPL is comprised of *rules*, *events*, and *policies*. A *rule* lists one or more conditions under which a set of specified actions are performed. For instance the following rule says that if the current packet is an ESP IPsec packet, then policy `in_ipsec` should be applied

```
Rule EQ(IP_PROT,ESP) APPLY(in_ipsec)
```

An *event* causes a designated set of rules to be processed. The typical trigger is the arrival of a packet, although events can also be triggered by timer and from a program or processor outside of

PPL. The event statement represents logical ports 1 and 2. The rules apply a policy if the packet is a TCP packet with just the TCP SYN flag set, and then each packet is unconditionally forwarded.

```
Event(1,2)
Rule EQ(TCP_SYNONLY,1) APPLY(tcpconrate)
Rule FORWARD STOP
```

Note that there could be many instances of this event running concurrently if at any instant in time there are multiple packets available from these two ports. This is managed automatically by the virtual machine.

The third important piece of the language is the PPL policy. A policy is a (typically) complex function, often with internal state. For instance the following policy

```
Enc: Policy CIPHER ENCRYPT(AES,SHA1)
      KEY(keystore(n)) LOCATION(CONTENT,0)
      PAD(SEQ)
```

does the following when applied: the payload of the current packet is encrypted in place using the AES128 cipher with a key from the array keystore. Sequentially numbered padding values are added as needed. The ciphertext is also accumulated in a hash digest using the SHA-1 algorithm.

Much of the power of PPL is in the policies provided, and we will return later to discuss policies in more depth.

Values

As with many other things, values in PPL are packet centric. There are several ways to refer to data in and about a packet:

Named packet fields. Although PPL can be used with any protocol, PPL “understands” IPv4, IPv6, TCP, UDP, and a few other things better than other protocols. IP_DEST refers to the destination IP address in the current packet. PPL also understands dynamically the difference between IPv4 and IPv6, so the rule

```
Rule EQ(IP_SOURCE,IP_DEST) . . .
```

behaves as expected whether the current packet is IPv4 or IPv6.

Dynamic packet fields. One can index explicitly to data within a packet. For instance PFIELD(2).b

refers to the byte at offset 2 in the current packet. CONTENT(n).q refers to the quadword (16 bytes) beginning at offset n within the packet payload.

Packet state. PPL defines a number of values that represent static information about the current packet. For instance PS_FRAGMENT is a Boolean indicating whether the current packet is a fragment (meaning either bit MF set or non-zero fragment offset present in an IPv4 packet, or presence of a fragment extension header in IPv6). PS_LPN is the logical port on which the packet arrived. PS_VLAN is the virtual network to which the packet belongs.

Even constants are packet centric. For instance the following rule

```
Rule EQ(IP_DEST/24,66.197.248.0)
      NE(IP_PROT,UDP) . . .
```

determines if the first 24 bits of field IP_DEST in the current packet are equal to 66.197.248 and if the IP protocol field is not UDP.

Many policies manage data structures that aren't directly visible to the PPL user. For instance the QUEUE policy manages a packet queue, and the ASSOCIATE policy manages an associative lookup table. The one data structure that is available to the PPL user is an array. For instance

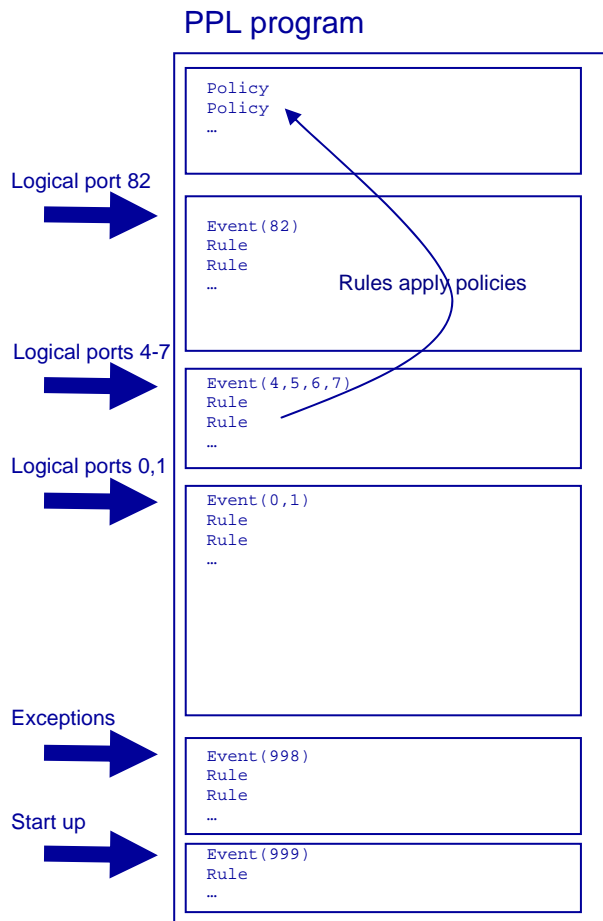
```
Alternate_servers: ARRAY(32).q
```

defines an array of 32 quadword (16-byte) values. Although generally the PPL virtual machine takes care of determining where the array will actually reside, we will see later that an implementation-dependent part of PPL allows one to control, when needed, the actual mapping of an array. Also, since the virtual machine always knows the extent of an array, it will never allow you to refer to a non-existent element.

Values used in rules and policies can be 32- or 128-bits in width, and these can be used interchangeably.

Program Structure

PPL programs thus consist of rules packaged by events, and rules typically refer to policies, as shown in the following diagram. There is a special event that will be triggered by the virtual machine in the event of an exception, and another special event that will be triggered at system startup.



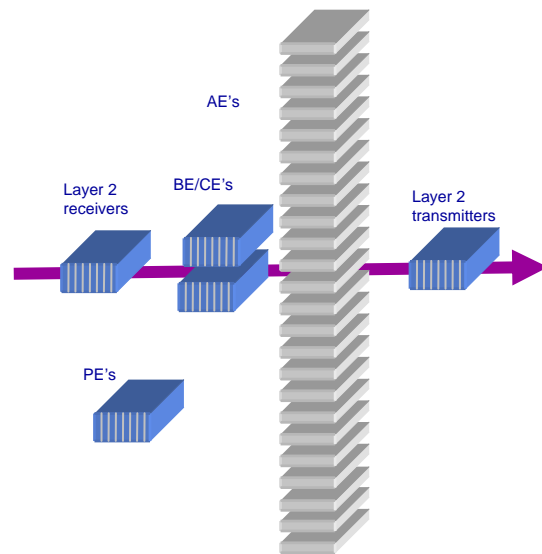
IXP2805/2855 Virtual Machine Mapping

At this point we will leave the language for a short while and discuss the implementation of PPL on the Intel IXP28xx NPUs, because it gives one considerable insight about the power of the language and the resultant performance.

Although PPL could theoretically be compiled to a machine instruction set, the current implementation on the Intel IXP28xx compiles PPL to a low-level representation which is then interpreted by the virtual machine. Much of the design of the virtual machine is aimed at extracting the full power of the NPU's resources for optimal performance. The virtual machine is designed very much like one would design a high-end CPU, meaning it is pipelined, uses many concurrent execution units, does asynchronous (overlapped with execution) memory operations, and has received careful cycle-by-cycle optimization. As such, it is much different than, say, a Java virtual machine running on a Pentium.

The processing of PPL events is done by the virtual machine's action engine (AE). An AE is typically

presented with an arrived packet and the set of rules to run on behalf of that packet. The IXP28xx has 16 independent microengines or cores, and each has 8 hardware-switched threads. The virtual machine allocates most of the microengines to AE's, and allocates two AE's per microengine. Two is a good tradeoff given resource constraints, because when a microengine is stalled as a result of an AE doing a memory read or write, the other AE on the microengine runs. Typically, 24 AE's get allocated (on 12 microengines), so 24 PPL events get processed in parallel (or, at the other extreme, 24 occurrences of one PPL event are running).



PPL execution is also pipelined to a certain degree, which is the role of the BE and CE. These do certain preprocessing on each packet and evaluate any rule expressions that can be evaluated safely ahead of execution (e.g., those that refer to packet state or contents). Thus the BE/CE can often "rule out" some of the rules of the PPL event to be run on an AE.

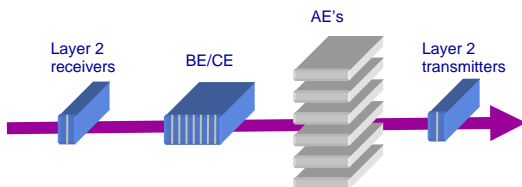
Most policies, when applied, run directly on the AE of the event invoking the policy, but a few policies have such a large internal state that they need a separate microengine when used (PE's).

There is an important point here – that with the PPL virtual machine, *processing power is dynamically assigned*. In a conventional IXP2xxx software design, one assigns a fixed role (by programming it) to each microengine, and the microengines operate in pipelined fashion to process a packet. Thus the allocation of processing power is predetermined by the software designer, cannot be dynamically changed, and thus at virtually any instant in time is suboptimal. In the PPL virtual machine, processors move from program to program (event to event) as the need arises.

Anticipating some questions, a few notes here. There is a way to ensure that packets from the same flow are processed sequentially so that they do not get out of order. Also, if there is a PPL event that must be processed serially, there is a way to designate such.

IXP2350 Virtual Machine Mapping

Before returning to the language, we might as well show how the PPL virtual machine maps onto an Intel IXP2350, since it has many fewer microengine cores (4, not 16). Because the IXP2350 has a larger local memory per microengine than the IXP28xx, three AEs are allocated per microengine. Therefore, in the typical IXP2350 configuration, there are six AE's allocated in two microengines, a combined CE/BE take a third microengine, and the fourth microengine runs the receiving and transmitting threads and some incidentals (including a PE function if needed).



More on Rule Expressions

Rules can do one or more equality and magnitude comparisons on pairs of values (optionally with masking, as in the use of the subnet mask in the example earlier). One additional expression is SCAN, which provides payload scanning. For instance the following will search the current packet for the designated string (which happens to be a signature for the subseven Trojan horse).

```
Rule SCAN(" |0D0A5B52504C5D3030320D0A| ")
```

We can also use a regular expression with scan. 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(L4_DPORT,80) . . .
```

Rule Actions

Rules can contain one to many actions, which are performed sequentially if the rule evaluates to true. We have already seen the most-potent action - APPLY(x) - where x is the name or value of a policy. Policy names have values, which means that policies can be selected dynamically by computing a value.

Some of the other actions are summarized below.

SET computes the value of a simple (one operator) expression and assigns the value. Like most other things, SET can operate on 32- or 128-bit values or a combination of the two.

FORWARD transmits the current packet somewhere. The somewhere depends on the values expressed with the action.

DROP drops the current packet.

LOCK and UNLOCK manipulate a specified lock and are useful when concurrent events need to update, for instance, a shared array.

COMPUTE performs a more-complex function on one or two values. Examples of the functions that can be expressed are converting endian representation, hashing, get random number, get current time, compute a checksum, multiply and accumulate, and compute a CRC.

Policies

Typical PPL programs consist of rules that make some decisions and perform some actions, but where most of the logic is embedded in the policies used. Below we review briefly most of the policies in PPL.

CIPHER allows one to encrypt and decrypt part or all of the current packet in a manner that is not tied to any specific protocol (e.g., IPsec, SSL, TLS, 3GPP, RTP encryption, XML encryption). Options exist for different algorithms, whether to cipher in place or not, and for different types of padding. It also allows one to accumulate data into a hash digest and calculate an HMAC.

ASSOCIATE and a few related policies create and manage a content-addressable data structure such that one can look up values by search keys. It has a wide range of uses, such as looking up IPsec

security policies, doing NAT, maintaining flow-based traffic counts, and others.

PACKET performs certain functions on the current packet or a different packet for which one possesses a handle, such as dropping it, making it the current packet, and inserting or stripping header or trailer space at different places within the packet.

NEWPACKET creates a new packet, with options relating to its initial value, whether it encapsulates the current packet, etc.

DEFRAG collects packets deemed to be related fragments until all the fragments have been collected or a reassembly time is exceeded.

SUPERPACKET manages and operates on a "superpacket," which is an arbitrary ordered set of whole packets whose collective payload one wants to treat as a single payload. Superpackets are especially useful in detecting signatures that span multiple IP packets.

PROGRAM is a policy that allows a PPL program to communicate with a program outside of the PPL virtual machine. We will return to this important subject in the next section.

PATTERNS has several different flavors. One does a multi-pattern search of the packet (or superpacket) content against a database. In the current implementation, a further-optimized form of the Wu-Manber algorithm is used; the algorithm can determine that the payload doesn't match a database of 1000s - 100,000s of patterns in a remarkably short time. The second form compares a value (typically an IP address) to a database, looking for the longest-prefix match. In the current implementation, a further-optimized Eatherton tree-bitmap algorithm is used for the latter.

RATE maintains time-based rates (e.g., rates of occurrences, bit rates). In the example below, we use it to inhibit more than 1000 TCP connection attempts per 30 seconds over a time period of a day.

```
Define day = "86400000" #Msec per day
Define sec30 = "30000"
Con_rate: Policy RATE
           RESETTIME(day)
           TIMEBASE(sec30)
           COUNTING(1)
. . .
Rule EQ(TCP_SYNONLY,1) APPLY(Con_rate)
      GE(Rr0,1000) LOG DROP
```

(A few added notes – many policies return a value in Rr0. LOG and DROP are two additional actions. And this is written correctly as a single rule.)

QUEUE defines a set of packet queues and performs an operation on a queue (namely enqueue, dequeue, and query). An option exist to assign arbitrary weights to packets, and thus queues can be weighted, allowing one, for instance, to dequeue from the heaviest queue of a set. Another option is whether active management of the queues should be done. If this is selected, a concurrently running function within the virtual machine dequeues automatically (WRR, DRR, weighted, or priority can be selected) and transmits the dequeued packet to a user-specified place (e.g., an Ethernet port, the IP forwarder, a PPL event, the Linux stack or program on an adjacent Pentium processor).

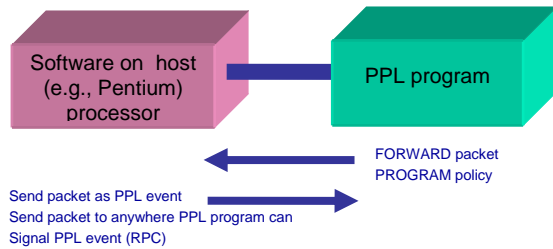
CONTROL is a policy whose definition allows for easy extension of control functions. Today only one function is defined – enabling or disabling the processing of a specific event on a periodic (timed) basis.

CLASSIFY is a general multi-field, multi-criteria searching mechanism to look up a set of values in a database. An implementation-dependent provision exists to map the database into a TCAM. CLASSIFY is useful in comparing a set of values, such as a 5- or 6-tuple from the current packet, where the comparisons aren't exact matches or where the comparison operators are different for each item in the database. CLASSIFY isn't restricted to using fields within the packet; it classifies whatever set of named values it is pointed at.

External Program Interfaces

We have deferred discussion of the PROGRAM policy to here because it is a piece of a more-general subject of how PPL can interact with a variety of non-PPL programs.

In general, a PPL program can forward a packet to an outside program or invoke an outside program via the PROGRAM policy, and vice versa. We will use the diagram below and assume a typical network appliance design consisting of one or more Pentium cores as the central resource and the microengine cores doing the heavy lifting and real-time processing of network traffic.



Looking at it from the perspective of the PPL program, the PPL FORWARD action can be linked to a variety of things, such as a network port or to an external program, and the PROGRAM policy can be linked to an external program to do a remote procedure call with parameters. Thus a PPL program can send a packet up to the host or can invoke a function on the host.

In the opposite direction, a software interface or API is provided to allow software in the host to send a packet to the PPL program (with a logical port number), send a packet to anywhere the PPL FORWARD action allows a packet to be sent, and signaling a PPL event with parameters.

Today the primary way to attach the Pentium(s) and the NPU microengines is through a PCI or PCI Express bus. An extension of the PPL virtual machine called PXD provides this packet-level and RPC abstraction over PCI or PCI Express.

For completeness, we note that PPL programs can also interact with other PPL programs in separate devices, with a local control processor, and with data-plane microcode (using Intel IXP terminology), for instance “secret sauce” microcode that a user might need. In the latter case, communication is via rings and microthread signals.

Handling Hardware Dependencies

Although PPL is independent of a particular processor model or architecture and particular board or blade design, there are relationships that need to be defined. This is done via the PPL DeviceMap statement, which isolates the physical and implementation dependencies to one spot in the PPL program. DeviceMap is chip specific, and a separate specification will exist for each chip type. Here we will illustrate using the IXP2xxx DeviceMap.

```
DeviceMap
NPU(2350,900,21)
PACKET_MEM(DRAM,64000,20,32)
ARRAY_MAP(serv_list,ext_$pdk servlist)
LINK(0,INOUT,GE_INT,0)
LINK(1,INOUT,GE_ON_SPI,0,8,1514,0,0,
      0,0,"IXF1104")
LINK(2,OUT,PCI)
PROG(lin_stk_dr,REMOTE)
```

To dissect this, it says

- The NPU is an IXP2350 with a microengine clock speed of 900 MHz, and of the many modes the IXP2350 supports, this one is configured with one internal gigabit Ethernet MAC enabled and MSF channel 0 as a 16-bit SPI-3 SPHY interface (mode 21).
- Allow 64 MB of DRAM for packet buffers, leave at least 20 bytes of space in front of every packet and use 32 bytes of metadata per packet.
- There is a specific PPL array `serv_list` that we want located in memory at the same place as external symbol `ext_$pdk servlist`.
- PPL logical port 0 maps to the internal gigabit Ethernet controller 0.
- PPL logical port 1 maps to a port in an IXF1104 Ethernet controller on MSF channel 0. The other values are some Ethernet controls.
- PPL logical port 2 maps as an output to the PXD mechanism over the PCI Express bus.
- A PPL PROGRAM policy that refers to the symbol `lin_stk_dr` causes, when applied, an inter-program communication to a program of that name on a host processor on the PCI Express bus.

A number of other capabilities exist in the Device-Map section of PPL, such as

- Controlling what memory the PPL virtual machine does and doesn't use
- Similar LINKs for other types of layer 2 interfaces
- Automatic tests for malformed packets
- Debug controls

Exception Handling

An early user of PPL considered resultant system robustness as its most-important benefit. Because of its power of expression and its run-time checking, networking functions written in PPL have less exposure to latent system hang-ups and undiscovered security holes.

Event 998 is always defined to be an exception handling event. When an exception occurs, that PPL event is invoked, along with the type of exception, the rule causing the exception, and the current-packet handle. Types of exceptions include extent errors (e.g., relative to a packet or array), invalid packet handle, insufficient storage, lock timeout exceeded, and others.

Roadmap

One of the important characteristics of PPL is independence from the underlying silicon. Thus, for instance, PPL is the means to develop a solution today for hardware deploying discrete network processors, and then port one's application seamlessly to higher-integration SoC devices.

Other considerations include

- Porting the virtual machine to different multi-core architectures
- Implementing a language expansion to ATM AAL2 and AAL5
- Implementing a finer degree of concurrency in the language called run groups (where individual rules or groups of rules can be designated as running concurrently)
- Dynamic compilation (changing the PPL program while running)
- Direct compilation to a machine instruction interface

Summary

PPL has proven itself to be a highly effective language for the development of data-plane software in a multi-core environment. In conventional NPU approaches, because of the high complexity of NPU architectures, the software developer typically spends 90% of his or her time on the many details of the NPU and its tools, and very little time thinking about the application itself. With PPL, the tables are reversed; the focus of the software developer is on the application, and the total time from starting to having a working system on live hardware literally goes from multiples of months or years to a few days.

Lunch is reasonably priced but not free. The theoretical penalty is performance; the actual penalty depends on a number of factors. For the Intel IXP2xxx family, typically we find that a PPL program has a 20-30% disadvantage over the same application written by NPU experts (over a much longer period of time) in microengine assembly lan-

guage. A middle ground is using the C language, but this approach still requires considerable NPU expertise by the C programmer, and the resultant C code is highly NPU-model-dependent and thus not portable. Generally, PPL should perform as well or better than NPU C programs written by engineers not having substantial expertise in the many details and aspects of the NPU architecture.

There are other situations where PPL in fact can be faster. One is related to the dynamic processor assignment discussed earlier. Using the conventional approach, one typically fixes the function of each microengine. For instance, if there are both ingress and egress paths, some microengines are pipelined in the ingress path and some for egress. At any instant in time, it is unlikely that all microengines have work to do. In fact, if at an instant in time, the ingress traffic is heavy but the egress traffic is light, half of the NPU is going to waste (and perhaps packets in the heavy ingress load are being dropped). In PPL, events get dynamically assigned to a processor short-term as their stimuli (usually a packet) arrive.

Another way that PPL programs can actually be faster than the alternatives is when the PPL program spends most of its time in policies. Policies are major functions that have been carefully written and optimized by NPU experts for maximum performance.

As one last comment on performance, many believe that the best answer to performance optimization is to have a running system as early in the product cycle as possible in order to have the maximum amount of time to study it and optimize it in real-life situations. If you are in this school of thought, PPL has big performance benefits.

PPL and its initial implementation on Intel's IXP microengines have proven to be an effective way to develop data-plane software without investing significant cost in microengine expertise, yielding substantial gains in time to market, good performance, and better product robustness.