General purpose application software for network processing applications

By Curt Schwaderer

IP Fabrics, Inc. recently announced availability of a new high level language called the Packet Processing Language (PPL) and supporting software for the development of a variety of applications targeted for network processor based boards and systems. At first glance, it appears that the product is a development tool, but it is really a general purpose application software product, signaling a new era of software products available for network processing applications.

What is general purpose application software?

Is a spreadsheet classified as an application or a development tool? Is a database program an application or a tool? Well, technically they are application programs with a high degree of programmability by people who really do not have to be *programmers* in the purist sense. For example, a spreadsheet is an application program used by financial professionals to keep track of balance sheets, employee records, or company expenses. In an analogous way, PPL is an application program used by network engineering professionals to create complex packet processing equipment. Figures 1a and 1b illustrate the difference in these two approaches.

Figure 1a illustrates how a developer uses a development environment to write code such as C, microcode native, or higher level representation, builds it using the compilation tools for the network processor, then loads and tests on the target system. So, in essence, a development tool product runs entirely or nearly entirely on the development workstation. The user then writes the software to be run on the target system. In many cases software libraries can be incorporated into the application being developed for the network processor. In some cases they may be directly usable. In other instances developers may need to modify these libraries to fit the ultimate application's purpose. In addition to being networking engineers, users must also be fluent in the network processor language being used in order to develop the application.

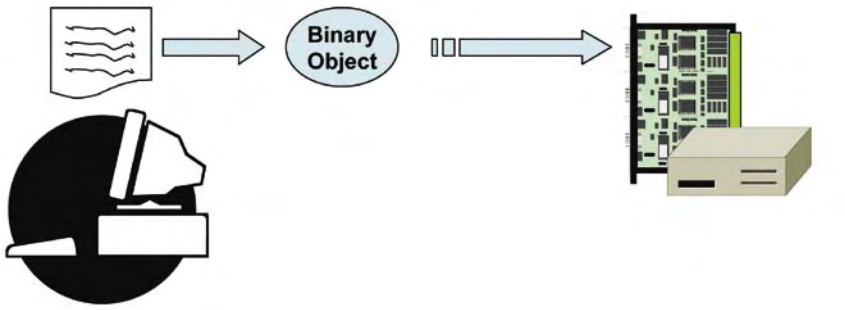

Figure 1a

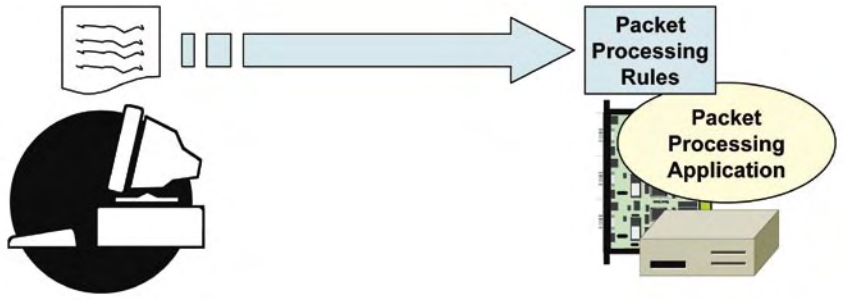

Figure 1b

Figure 1b illustrates the use of a general purpose application paradigm for the development of network processing products. The packet processing engine itself is an already built executable. With the product's tools, the user can specify a set of expressions, actions, and policies that govern packet processing for the application. In a similar way, an accountant might fill in formulas in spreadsheet cells when developing a payroll application. The resulting network operation application is then loaded onto the target and executed by the packet processing application. In this case, the network engineer need not be fluent in the network processor language. The engineer need only understand the syntax and semantics of specifying expressions and actions for packet processing required for their product. Think of the financial professional who must understand how to create mathematical formulas within the spreadsheet program to create an employee payroll spreadsheet.

But do network processors really need a general purpose program environment? Well, it probably depends on what is being built. The very basics of Network Processor Unit (NPU) programming are fairly easy to understand and, in general, the tools are pretty good. Therefore, if you are writing a simple application using the *common* instructions and do not have to be concerned too much about the parallel processing interaction, the development tool approach may work just as well as the general purpose program approach. But when faced with complex packet processing on many parallel micro engines and multiple interactions and critical sections within these sections, a general purpose program environment can be a lifesaver.

Network processors: What makes them a programming challenge?

Network processors came into being as a flexible programming yet wire speed performance alternative to ASIC development. ASICs can require up to $1 million or more to develop and build, making them cost prohibitive. What has made ASICs even more problematic is that the rate of change within packet processing

has been so fast that given ASICs became obsolete even as they were being developed. Many companies limit, or even prohibit, the development of ASICs for this very reason. In order to continue to advance networking products, features, and capabilities, a more flexible solution is required.

Network processors solve the fixed-ASIC-function problem by providing a highly parallel compute environment consisting of a general purpose processor and a number of parallel-processing compute engines, typically called micro engines. These micro engines perform data plane processing that an ASIC would have otherwise handled. This approach enables the software running on the micro engines to change with minimal cost, making them far more adaptable to the *rate of change* problem that ASICs have. Figure 2 shows the ingredients of a NPU.

The flexibility and wire speed support capabilities of the network processor meet data plane packet processing needs

powerfully and flexibly. However, the power and sophistication of these processors can also make them challenging to program.

An NPU comprises multiple, independent micro engines. Additionally, each micro engine may itself run multiple threads of execution. The NPU can also have thousands of programming registers that provide:

- Local and global storage for the data plane program
- Configuration for the I/O
- Queue and ring support
- Data/control registers that facilitate passing packets and control information between micro engines

There are also multiple memory types, each having its own storage capacity and access speed benefits and drawbacks. For example, dynamic random access memory is typically used for large memory storage (hundreds of megabytes to gigabytes range). Static random access memory is used for less storage but faster access (single to tens of megabytes). On-chip memory on the order of a kilobyte provides a very fast access alternative; however, it must be used carefully. Each micro engine has its own control store

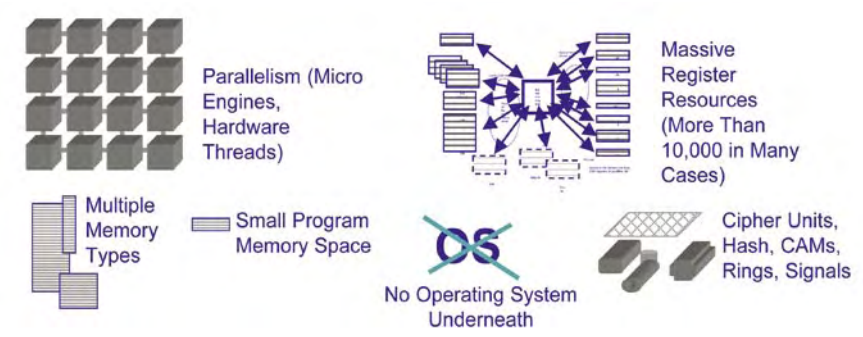

Figure 2

area where the packet processing program executable resides and typically supports less than 10,000 instructions for each micro engine. There may also be:

- Onboard support peripherals
- Cyclic-Redundancy Check (CRC) units
- Hash generators
- Content Addressable Memory (CAM) units

These onboard NPU peripherals can accelerate performance of some compute-intensive operations. But it also takes time to develop software that controls these on-chip peripherals.

In order to take advantage of this environment, the network processor micro engine language syntax and semantics must express the parallelism inherent in the environment, memory access types, and on-chip peripheral functions. So, as network applications take advantage of more and more of the capabilities of the network processor, the more complex the programming becomes.

General purpose application software approach

The PPL application software approach is somewhat of a hybrid of an operating system, virtual machine, and *spreadsheet* language. PPL controls and abstracts the onboard NPU components and peripherals. This allows the programmer to specify and use these facilities without having to know the implementation details. For example, PPL specifies a CRC compute action. The PPL programmer can invoke that action. Details of synchronization between users, use of the onboard CRC unit, or software implementation is thus hidden from the network engineer developing the PPL application.

The control program created by the PPL programmer contains all the expressions and actions to be executed on all packets coming into the system. While the programmer must think of all expressions in a given PPL program being executed simultaneously for a match/no match result on a given packet, the PPL general purpose application directs how the packets are fanned out among the many parallel micro engines and their coordination through the micro engines during processing. In this situation the PPL software application and the PPL program behave in a fashion similar to a Java applet and a Java virtual machine.

Another good by-product of the language is that it allows much more complex packet processing programs to be written than the control store of the NPU would normally allow.

Figure 3 shows an eight micro engine NPU, each micro engine having space to store 4,000 instructions (4K control store). If the overall packet processing program takes 17,000 instructions, the programmer needs to split the program up into stages of less than 4,000 instructions, allocate them across the micro engines, and provide queuing for passing packets and control information across each stage of the packet processing application. In this case, you could only run one instance of the program because you would need five micro engines to store one instance of the program, leaving three micro engines unused. If the program instructions exceeded the total number of control store instructions available in the NPU, the program would not be able to be run without being optimized using fewer instructions. Figure 3 illustrates the development of a native data plane application and allocating it across micro engines of an NPU.

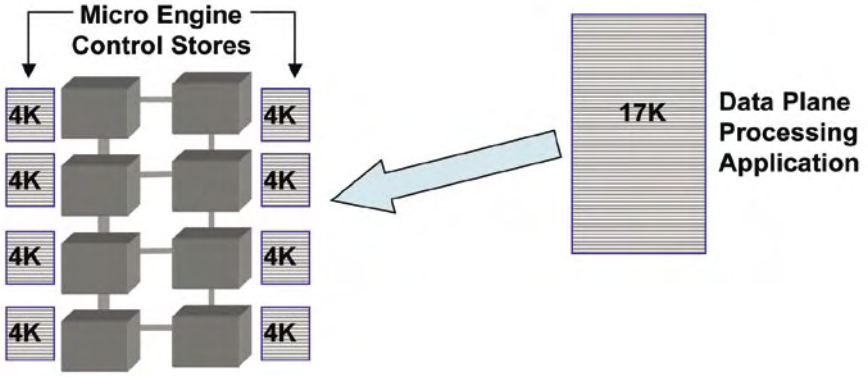

Figure 3

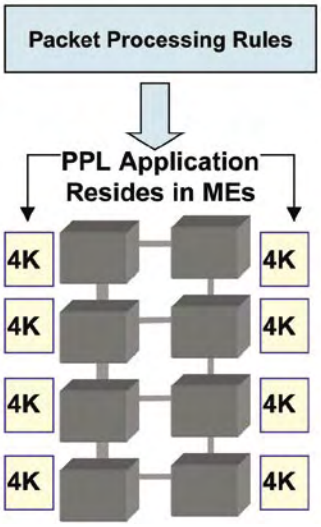

Figure 4

Figure 4 depicts packet processing application execution with the generic software application model. In this case, the control stores are filled with the generic application, not with the packet processing rules. As a result, the programmer partitions the program across multiple micro engines without concern. Also, large packet processing programs no longer depend on the micro engine control store size.

The paradigm that Figure 4 illustrates can lower the development complexity since no consideration need be given to breaking up the packet processing into smaller stages, chaining the processing together, and taking into account overall packet processing application size.

Performance costs and considerations

Of course an abstracted packet processing programming environment must come at a cost of some overhead and decrease in performance. Let's compare relative performance between writing native microcode with a development tool and using a general purpose application environment, such as PPL, by looking at the spectrum of applications.

First, consider a simple direct receive and forward to a port application. The reference design provided by Intel for the IXP2800 network processor uses two receiver micro engines, a transmit scheduler, two transmitter micro engines, and up to four IPv4 forwarding engines that can forward at approximately two Gpbs per micro engine, or about eight Gbps total. The IP Fabrics packet processing application uses the same number of receive and transmit micro engines but is flexible as to how many micro engines the user employs for the PPL generic application. The number of micro engines allocated to the packet processing application could be as few as three or as many as 15. Using 12 micro engines for the PPL applications, the same aggregate performance can be achieved when compared with the native microcode applications. So using a general purpose application for a simple design does not make a lot of sense unless the developer knows things are going to get more complicated over time and wants to reuse code. For more information, visit www.ipfabrics.com.

However, as the complexity of the network processing increases, the performance gap between writing native microcode and using the PPL application narrows. The implementation of the PPL application has been optimized for highly parallel, complex packet processing applications. Thus, for multifunction equipment requiring deep packet inspection, the PPL application could indeed perform much better than a native application, depending on the amount of time the developer is willing to spend on optimizing the native microcode program.

Packet Processing Language basics

A PPL program consists of independent blocks of rules called *events*. Each event can be invoked by an incoming packet or perhaps an action of a previously executed event. Each event contains some number of *rules*. A rule has two parts, the expressions list and the actions list. Each expression in the expressions list yields a true or false result depending on whether the packet matched the criteria in that expression. The actions list within a rule is executed if all the expressions for that rule evaluate *true*.

Conceptually all expressions for all rules in a given event are processed simultaneously. So for a given packet/state input to an event, all the expressions are evaluated in parallel, resulting in one true or false result for each rule. Once the expressions evaluation is complete, the *actions list processing* begins. The actions list processing is performed sequentially starting with the first true rule in the event and continues on until a STOP action is executed. The language allows for jumps forward and backward in the PPL program and *late expressions evaluation,* such as evaluating an expression during actions list processing resulting in a true or false result that may or may not result in continued actions list execution for that given rule.

Expressions include:

- Equality
  - Equal/not equal
  - Less than or equal to
  - Greater than or equal to
- Variations on scanning for strings or regular expressions within the content of the packet

Actions include:

- Arithmetic and logical
  - Set
  - And
  - Or
  - Xor
  - Add
  - Subtract
  - Shift
- Computational
  - Variety of CRC and checksum calculations
  - String-to-value-to-dot-notation conversion functions
- Applying a class of packet processing
  - Called a policy

Policies include:

- CAM-like state/variable storage and access called associate tables
- Rate
- Packet monitor
- Crypto services
- Packet generation/transformation
- Patterns comparisons

PPL also defines a connections policy where the PPL programmer can attach a connection state to a flow of packets, and that state will be automatically looked up and carried with the packet for a given connection. This enables the programmer to operate not only on the packet fields themselves, but also on generic state variables for packet streams (for statistics, type of service, rate calculations, and other network management applications).

Listing 1 gives a small flavor of what a PPL program might look like:

```
tcpPktRate: Policy RATE RESETTIME(day) timebase(sec30)
counting(1)EVENT(0)
Rule EQ(TCP_SYNONLY) APPLY(tcpPktRate) GE(Rr0,1000) LOG DROP STOP
Rule FORWARD STOP
# End event 0
```

Listing 1

This particular code segment identifies incoming packets with the SYN bit set, indicating a Transport Control Protocol (TCP) connection request. The packet rate policy is applied in this case, and the policy returns the number of occurrences within the time base period in a register called *Rr0*. This register is then compared to 1,000. If this packet represents more than 1,000 connection attempts within a 30 second time base, the packet is logged and dropped. Notice that the second rule has only an actions list: FORWARD (forward the packet) and STOP (stop this event). So, for a TCP SYNONLY packet, both rules will be true. As the actions list for the first rule is processed (first the rate policy, then the comparison of the rate to 1,000) if this *late expression* is false, the LOG, DROP, and STOP actions will not be performed, resulting in the packet being forwarded. For all other non-TCP packets, only the second rule is true, so these packets will be forwarded as normal.

The scan expression is an interesting feature of the language as it provides simple programming for content inspection for applications such as intrusion detection, content switching, and Denial of Service (DoS) attack recognition and mitigation. A simple SCAN expression might look like:

```
Rule SCAN("|0D0A5B52504C5D3030
320D0A|") LOG(subseven_trojan)
```

This rule scans the content for a byte string representing a possible attack. The scan operation acts similarly to an expression in that it yields a true/false result and also provides the location of where the string matches. In addition, the SCAN expression takes regular expressions as input. These language constructs enable development of a wide range of network processing applications.

Both of the above examples are written in just a few lines of PPL, but would take many hundreds or thousands of lines of microcode to perform the same tasks. So, the time savings of writing in a language like PPL can be considerable.

Summary

It appears that the generic application program concept has made it into the network processing arena with the promise to harness all the power, parallelism, and capabilities of the network processor into an easy to use application environment where networking engineers can implement complex packet processing equipment quickly and easily. In such an environment, networking engineers will be able to implement complex packet processing equipment quickly and easily. Network applications for this kind of environment include:

- Content switching and load balancing equipment
- Dynamic peephole firewalls
- Session initiation protocol proxy, Voice over IP processing
- Cable modem termination systems
- Encryption gateway, security appliances
- Intrusion detection, DoS attack recognition systems

The concept is not without successful precedent in other software disciplines. As long as the environment can keep up with bandwidth requirements for these systems, the benefits of faster, lower cost development send a very compelling message to network equipment providers in all industry segments.

