Understanding Virtual Machine Performance
Part 1

January, 2005

Copyright © IP Fabrics, Inc. 2005Abstract

Abstract

Performance of a programmable network device is characterized by throughput, latency and footprint. In addition to being far easier to develop, code written in PPL[1] and interpreted by the PPL Virtual Machine (VM) can outperform code written in a low–level language, for many applications. This is accomplished by hiding memory access latency, performing data-path optimization, reducing communication between threads and enforcing functional re-use. These optimizations scale very well with logic complexity and far outweigh the virtual machine overhead.

Introduction

The need to simultaneously address productivity and performance while developing increasingly complex network processor based applications, has underlined the need for functional yet high-level programming languages, like PPL. While it is widely accepted that logic modeling in a functional language is much simpler ([1]), there is typically the concern that the interpretation of such logic in the target machine comes at the expense of performance.

In this white paper we argue that this performance penalty can be significantly reduced if the virtual machine that interprets functional logic is designed to take advantage of well known parallel programming optimizations. In many cases virtual machine performance may be better, when compared to software developed using low-level languages. Part 2 of this document will analyze system and application level benchmarks to test this argument.

In the next section, we outline the parameters that are used to characterize network device performance. We also explain the optimizations that are used by the PPL virtual machine for IXP28x0 and argue why these optimizations scale well with complexity. Then we examine the PPL virtual machine overhead (interpreter logic), and end the discussion with a summary.

Characterizing performance of network devices

Network device performance is typically characterized by three parameters – ***Latency***, ***Throughput*** and ***Footprint***. Latency (measured in seconds) is defined as the time-interval between departure and arrival of frames on the output and input ports respectively. Throughput (measured as frames per second) is defined as the maximum rate at which none of the received frames are dropped. The frame size is system or application dependent. For example, very short frames may not make sense for video-over-IP applications.

Footprint is an important parameter because it translates to the device cost to meet throughput and latency objectives. It is important in the virtual machine context as well, because it provides an interesting metric to judge the quality of virtual machine implementation for a target machine. Footprint[2] is measured by the instruction and data space (percentage of capacity) needed in heterogeneous storage elements.

---

[1] PPL is a trademark of IP Fabrics

[2] Note: VM overhead includes impact due to footprint, latency and throughput

A peek into PPL virtual machine optimizations

PPL programs are written and compiled for the virtual machine. At runtime, this application logic is interpreted by the virtual machine to run on the target network processor. The target network processor generally has many RISC-type processors with specialized instruction-set architecture, software-managed interconnect, limited instruction space and heterogeneous storage elements. The virtual machine uses well known parallel programming optimizations described below, to minimize the runtime penalty. The figure below illustrates the PPL virtual machine pipeline.

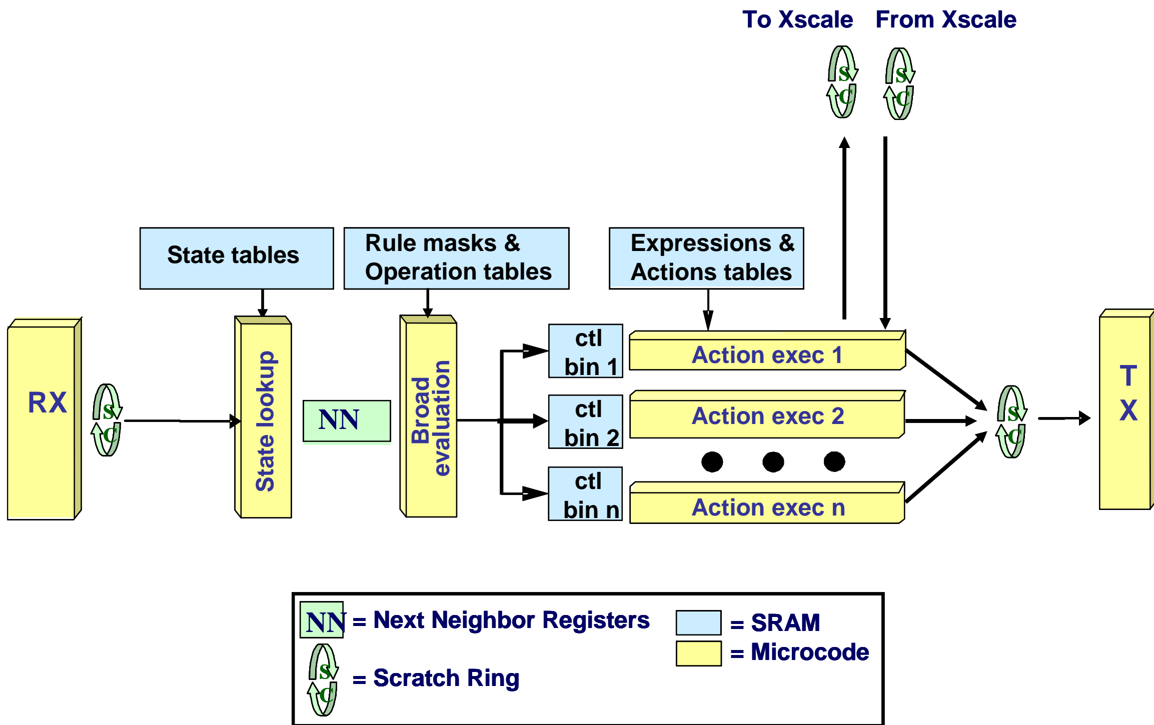

Figure 1. PPL Virtual Machine

Memory access latency: Scratch[3] ring trips are more expensive than general-purpose or next-neighbor register trips, SRAM trips are more expensive than scratch ring trips and DRAM trips are more expensive than SRAM trips. The virtual machine spreads these storage element accesses to maximize microengine and Xscale utilization, and also organizes the movement of data between heterogeneous storage elements, to reduce latencies.

Data-path optimization: PPL provides a rich set of highly optimized network processing primitives written in microcode (e.g. Patterns, Connections, Monitor, Associate, Packet,

[3] Read and Write latencies are not same

Crypto, Rate etc.). These primitives are called *policies* in PPL literature. Policies are used to perform complex modifications to packets selected for processing by the PPL program ([1]). The virtual machine has basic logic primitives as well, but these are generally used in wrapper logic around the core primitives. Also, the language syntax and virtual machine implementation allows user specified "custom" microcode or off-loaded remote logic (e.g. DSP on the PCI bus) to inter-work with PPL code.

Communication between threads: Communication between threads of the same microengine, different microengines of the same cluster, or even different clusters -- have varying degrees of negative impact on the overall throughput and latency. To reduce this impact, specific strategies are adopted while mapping the various stages of the virtual machine pipeline to microengine threads. For example, to optimize memory utilization in the virtual machine pipeline, all the "broad evaluator" threads are co-located, BE and SL threads have next-neighbor register adjacency and AE (early and late processing) threads are co-located in the same microengine ([1]).

Functional re-use: In a PPL program, one could associate a particular type of policy (e.g. Associate or Connections) with multiple rules. This type of re-use helps keep the footprint or virtual machine overhead low. Only 25% of the footprint (instruction memory portion) is non policy related, and this number will shrink further as new policies are added, and BE and/or SL is further optimized ([1]).

Optimizations scale well with complexity

For smaller applications, one could easily apply these time tested parallel programming techniques and write programs in a low-level procedural language. But as complexity grows, this becomes a challenge. In IXP28x0, there are 15,452 software visible registers to manage, besides the numerous accelerators and heterogeneous storage elements.

PPL virtual machine automates this process – logic specified in PPL is compiled to byte-code, which includes the expression/action, policy-descriptor, array-descriptor and other user-defined tables. At runtime, in a continuous loop, events (arrival of packets on ports) direct AE[4] computations that are driven by these tables. The SL (metadata extraction stage) and BE (expression evaluation stage to prune computation space) precede AE and can be considered as the only *fixed* overhead for the virtual machine implementation ([1]), since Rx and Tx stages are intrinsic to any implementation.

Applications with little or no AE processing (as in core network elements) are not good candidates for any virtual machine approach, since it would be hard to hide the BE and SL overhead. However, applications with dense AE computation (like in edge-network elements), benefit from the virtual machine approach, since the SL and BE overhead is easily masked.

Tuning virtual machine performance

Besides keeping the instruction memory footprint due to SL and BE stages low, other measures with varying degrees of payoff[5] can be adopted to limit the virtual machine overhead:

[4] AE=Action Executor, BE=Broad Evaluator, SL=State Lookup (Details in Figure 1 and [1])
[5] Note: This varies from application to application

- By minimizing the wrapper logic (e.g. set, and, or, scan, lshift etc.), and instead taking advantage of policies (e.g. patterns, connections, association, crypto etc.), computation latency can be improved, because policies are highly optimized micro-code.

- By configuring virtual machine parameters in device-map (e.g. array-memory, association-memory etc.) to reduce lookup time (e.g. SRAM as opposed to DRAM), overall latency is reduced.

- By sequencing the rule-elements based on traffic-analysis (e.g. putting expressions that evaluate to false mostly ahead of others), overall throughput and latency objectives can be improved.

- By appropriately distributing expression evaluation in the PPL program to *early* and *deferred* AE processing phases of the virtual machine (e.g. using *act* to break the rule-element chain), throughput through the virtual machine can be improved. This allows computations for two separate packets to be interleaved.

These are simple tuning steps that one can employ to improve virtual machine performance.

Summary

Today's network processors and applications that run on these parallel machines are becoming increasingly complex. While it is widely accepted that high-level, functional programming languages like PPL provide more than 100 fold productivity gain, there is a common myth that this productivity comes at the cost of performance penalty. In this white paper we have argued that this is not always true. Disciplined virtual machine implementations, like PPL virtual machine for IXP28x0, take advantage of proven parallel programming optimization techniques to hide the virtual machine overhead. The gains become more evident with bigger and more complex applications.

References

[1] Leveraging PPL to reduce TTM for Network Processor applications