

Programming A Data Flow Processor

How to reduce the programming complexities typically associated with network processors, while maintaining the efficiency and control required for real-time, data-driven applications.

September, 2003

Xelerated, Inc.
200 Wheeler Rd
Burlington, MA, 01803
Phone: 1 (781) 505 1921
Fax: 1 (781) 505 1382
info@xelerated.com
www.xelerated.com

Xelerated makes no representations or warranties with respect to the accuracy or completeness of the contents of this publication and reserves the right to make changes to specifications and product descriptions at any time without notice. No license, whether express, implied, arising as a result of estoppels or otherwise, to any intellectual property rights is granted by this publication. Except as set forth in Xelerated's Standard Terms and Conditions of Sale, Xelerated assumes no liability whatsoever, and disclaims any express or implied warranty relating to its products, including, but not limited to, the implied warranty of merchantability, fitness for a particular purpose, or infringement of any intellectual property right.

Copyright © 2003 Xelerated. All rights reserved. All trademarks referenced in this paper are the property of their respective owners.

Introduction

As engineers look for new ways to scale the performance of programmable Network Processors (NPs), they must take a fresh look at the architecture of the individual processing elements they employ in their designs. Traditionally, architects have turned to derivatives of general purpose control flow architectures for their processing elements and organized them into arrays that behave in a data flow manner¹. While this can work for low bandwidths, hardware inefficiency and complexity become an issue when scaling this approach to 10Gbps and above. At these bandwidths, high power dissipation and complex programming models can make cost effective, timely system-level design difficult.

This paper introduces an approach that uses a synchronous data flow architecture for the individual processing elements, greatly simplifying the process of organizing them into arrays that behave in a data flow manner. The simplicity of this approach translates into greater hardware efficiency which translates into smaller die size and lower power dissipation. It then explains a programming model that virtualizes the array of processing elements in a manner that makes them look like a single, synchronous, control flow processor with no data dependencies. This model reduces the programming complexities typically associated with NPs that have large numbers of processors, while maintaining the efficiency and control required for real-time, data-driven applications.

¹ Data flow architectures use the arrival of data (in this case packets) to cause instructions to be executed

Scaling Network Processor Performance (Creating the Beast)

Packet processing applications exhibit a high degree of data parallelism that can be exploited effectively by employing a large number of relatively simple processing elements that operate in parallel. These parallel arrays are usually organized into arrays that exhibit “data flow behavior”. When using this approach, the large number of processors and the complexity of the individual processors can translate into a high degree of programming complexity. A key design decision that impacts programming complexity of the NPU is the topology of the array of processing elements. Processing elements can either be organized as a pool or as a pipeline. Hybrid approaches such as a pool of pipelines or a pipeline of pools are also possible. (See Figure 1) When conventional control flow techniques are used for the processing elements, pipeline topologies tend to trade off ease of programming in favor of hardware efficiency, while pool topologies tend to trade off hardware efficiency in favor of ease of programming. This is because pipelines make the multiple processors visible to the programmer (forcing the application to be partitioned and balanced across the processors), but they have simpler interconnects and they don’t require extra program storage for multiple copies of the same program. Pools can hide the multiple processors from the programmer, but require more complex interconnects and additional instruction memory to store multiple copies of the same program. The respective weaknesses of each topology tend to limit the extent to which the array can be scaled, forcing the utilization of hybrid topologies for high performance NPUs. By definition, hybrids have a pipelined component, which presents a multiprocessor dimension to the programming model. When this multi-processor dimension is combined with the programmer visible artifacts of other traditional processor performance scaling techniques, the programming model can quickly become untenable. In some cases, the programmer is forced to load balance and deal with asynchronous interactions between multiple threads running on multiple processors with multiple instruction sets that execute in multistage pipelines. Although high level tools can hide some of this complexity during program creation, they cause further inefficiencies and often have to be bypassed when it comes time for functional verification and performance tuning.

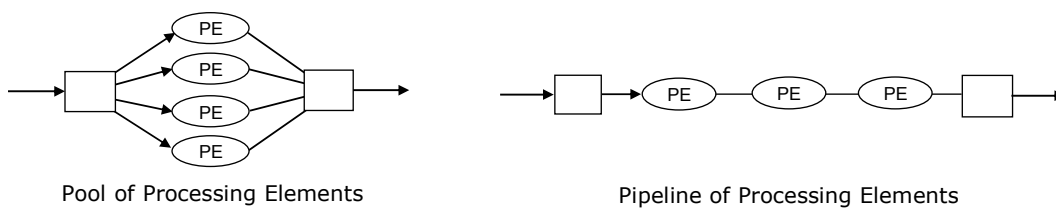


Figure 1: Processing Element Array Topology. Processing Elements (PE) can be organized as a pool or a pipeline. Hybrid approaches such as a pool of pipelines or a pipeline of pools are also possible.

Applying Synchronous Data Flow Techniques to the Processing Elements (Taming the Beast)

Processing elements based on a synchronous data flow architecture can be used to break the traditional tradeoff between hardware efficiency and programming efficiency in NPU design by allowing the designer to build fully synchronous pipelines of processing elements that appear to the programmer as a synchronous uni-processor. Such a processor is shown in Figure 2. The key to this approach is that by using a data flow processor core, it is possible to reduce the number of instructions executed per processor to one. When the number of instructions per processor is reduced to one, then the processor boundaries align with the instruction boundaries which create a uni-processor/single instruction set programming model. Programs can be written in a traditional manner without regard for the multiple processors, because the pipeline of processing elements is automatically balanced by assigning a single instruction to each stage. Resource conflicts and program branching effects are also eliminated allowing synchronous (deterministic) operation, which eliminates the need for cycle counting within a stage and handshakes between stages. The net effect is a programming model that is simple enough to allow

programming in assembly language. This reduces the need for traditional high-level language compilers that hide the complexity of the processor, but reduce efficiency and the degree of direct control over the hardware.

In this type of architecture the challenge is handling I/O in a synchronous manner that is both simple and efficient. This can be done by embedding I/O processors with synchronization FIFOs into the processing pipeline (See Figure 3.). This method has the benefits of being single threaded and hiding the complexities of managing I/O devices, but segments the pipeline in a manner that is visible to the programmer which needs to be accommodated by the programming model.

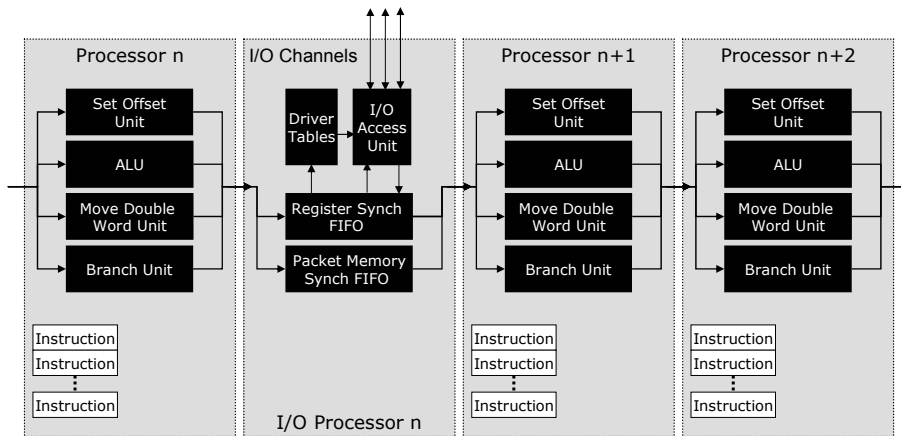


Figure 2: A data flow processor. Data consists of the packet and the program context. The packet flows through a local Packet Memory while the program context flows through a local Register File, including the Condition Flags (N, Z, C) and the Row Instruction Pointer (RIP). Upon arrival of data, the content of the RIP is used to fetch an instruction that is executed by up to four parallel execution units (Set Offset, ALU, Move and Branch). The data then flows to the next processor. I/O is handled by dedicated processors that move data between specified registers and I/O devices and return results from I/O devices by executing one of several I/O drivers.

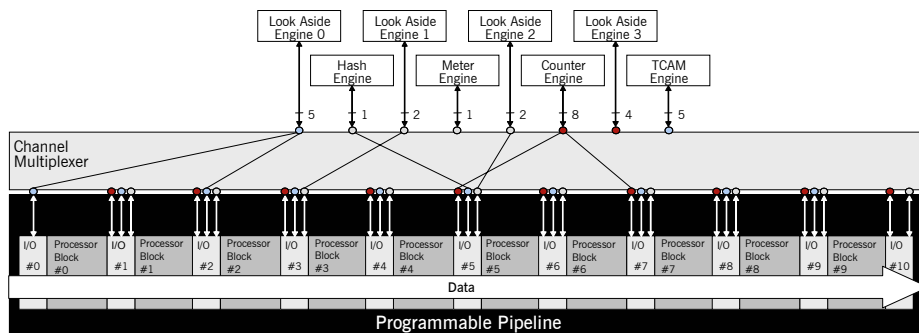


Figure 3: I/O processors can be inserted into the programmable pipeline to offload I/O from the blocks of processors. These processors are connected to I/O devices or engines via I/O channels and a channel multiplexer.

Making Data Flow Look Like Control Flow (Controlling the Beast)

Although data flow processors can be very efficient for data intensive applications such as packet forwarding, control flow processors offer a more intuitive programming model. Based on this, it is desirable to make data flow processors look like control flow processors to the programmer. This can be done by taking an “I/O centric” approach when creating the programming model. With this approach, forwarding plane program sequences are organized around I/O operations resulting in what is called a

“Multi-Stage, Classify/Action Programming Model”. (See Figure 4) This model is a simple extension to the “Classify/Action” programming model popularized by Agere. In this model classification operations map to I/O operations, which include both table search (read) and table update (write) operations. Action sequences map to packet parsing and search key extraction (search preparation), as well as sequences that are triggered by the result returned from the table search operation (such as header modification and packet preparation for counting, metering and dropping). This model naturally accommodates the notion of embedding I/O processors between blocks of data flow processors and maps effectively to layered protocol specifications where classification operations and action sequences at one layer may be dependent on classification operations and action sequences at lower layers.

In this model, the system architect first specifies the data structures for the application, and then creates “virtual processors” that are sized to meet the processing requirements of the action sequences. These virtual processors can correspond to one or more blocks of physical processors. Because the physical processors are synchronous, there is no need to estimate cycle budgets. Instead, the architect estimates instruction budgets, which is easier to do accurately. Graphical tools can be provided to check the resource consumption of these application models against resources available on the network processor to prevent oversubscription and allow quick evaluations of alternative application scenarios. These tools can also be used to automatically generate application specific configuration files that can be downloaded to the NPU and create programming templates for the virtual processors. Within the context of these templates, the forwarding plane programmer sees a traditional single processor, single-threaded, control flow programming model (See Figure 5.) I/O data structures can be pre-defined in a manner consistent with the control plane applications. Once written, these applications can be built, run on a simulator, and debugged in a manner similar to control flow processors. Traditional packet generators and table generators can be provided to facilitate the building of the test environment.

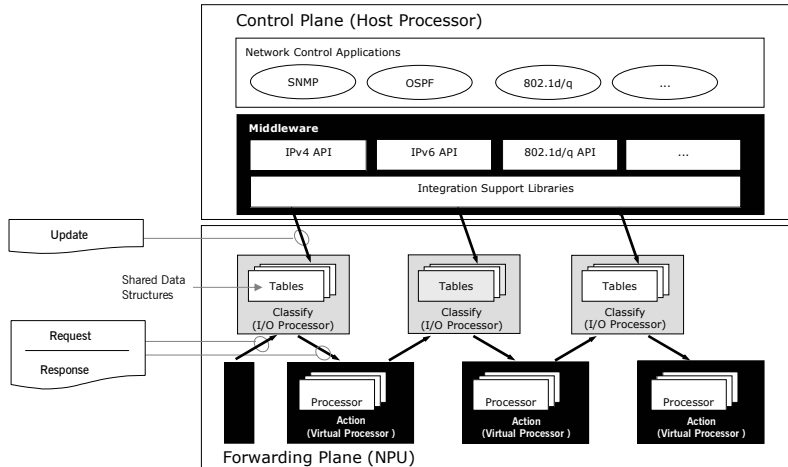


Figure 4: The System Architect's View of the Programming Model. The programming model utilizes a conventional partitioning between control and forwarding planes, where the two planes share a set of data structures (such as forwarding tables) that the Control Plane writes (updates) and the Forwarding Plane reads (via a request/response protocol). Control plane interactions with the data structures can be performed by “packetizing” the requests via a messaging protocol and feeding it through the pipeline, which returns the results via the original packet. The forwarding plane model maps Classify operations to I/O processors and Action sequences to virtual processors which consist of groups of data flow processors.

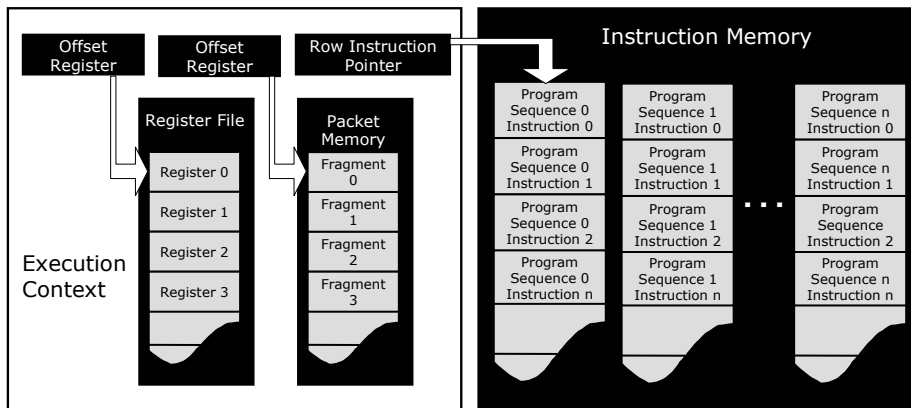


Figure 5: The Forwarding Plane Application Programmer's View of the Programming Model. Action sequences are written as conventional sequential programs with direct addressing of instruction memory via the Row Instruction Pointer and indirect addressing of the Packet Memory via an offset register. Deep inspection of the Packet Memory is accomplished by increasing the value of the offset register. The Register File can be addressed either directly or indirectly via an offset register. I/O drivers are "called" by writing to a device register that controls I/O.

Application Verification and Tuning (Getting the Beast to Perform)

A primary difference between the Multi-Stage Classify/Action programming model and traditional programming models is that applications are not done until the table entries are in place and the program cannot execute until data (packets) are input. This makes application development very suitable for a test case driven development method. It also makes it easier to tightly integrate the test environment into the development environment. Packet programs consist of linear sequences of instructions triggered by packet and table entry attributes. (See Figure 6) This means that input packets and table entries can be associated with each sequence facilitating test case generation as well as a simple graphical means of measuring and displaying code coverage for application verification. Because the processor is synchronous, application tuning becomes a spatial rather than a temporal problem, greatly simplifying the process. Since there is no notion of cycle counting, a traditional performance analyzer is not required. The linker checks the program's memory requirements and compares it to the available memory. As long as the program fits in the available instruction slots within the instruction memory matrix, the program will compile successfully and the application is guaranteed to run wire speed. To facilitate tracking of resource utilization a resource map (See Figure 7) can be generated by a graphical headroom analyzer.

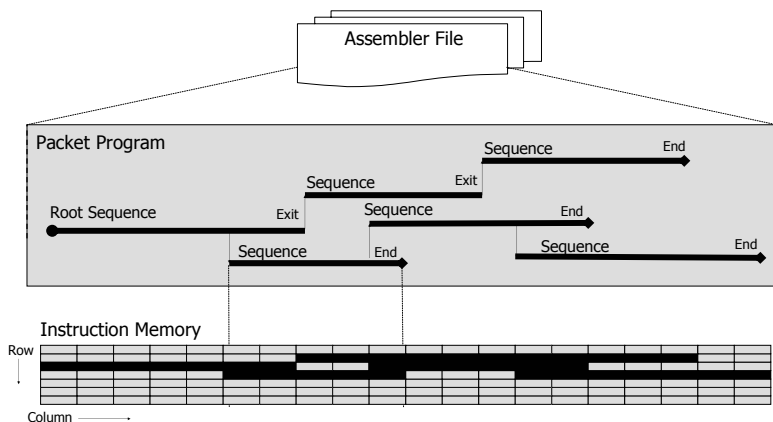


Figure 6: Program Structure. Packet programs consist of linear sequences of instructions that are mapped to the physical layout of the instruction memory of the processor array. Sequences are stored in the same row of instruction memory. A

packet program may leave the current sequence if a branch operation is executed and the branch condition is fulfilled. Every packet passes through every processor, so a NOP is executed if the processor is not required for the application.

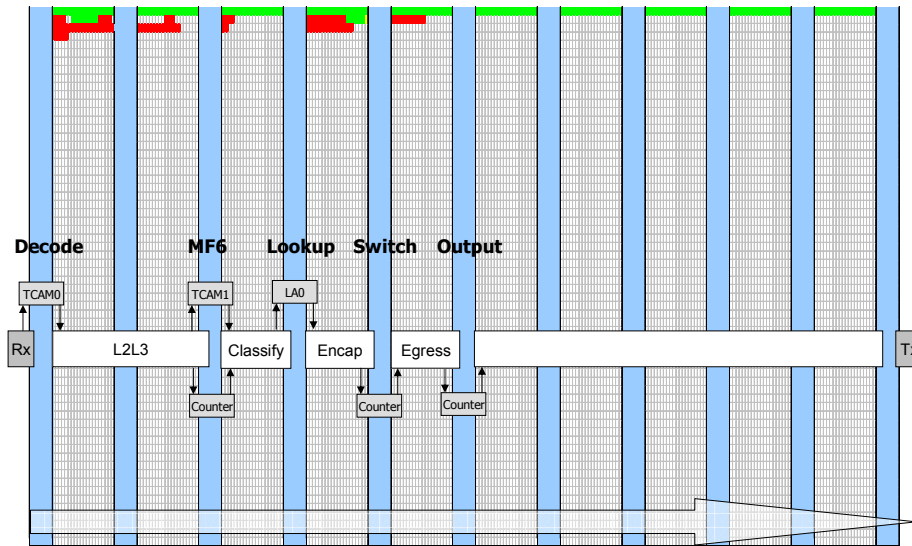


Figure 7: A Resource Map for an IPv4 Differential Services Forwarding Application. Data flows from left to right. Solid columns represent I/O processors. The 20 column x 64 row matrix between I/O processors represents Instruction Memory slots. The filled in boxes at the top of the matrix represent instruction slots that are used by the sequences of the packet program. In the case of this application, 65 of the 200 sequential slots are utilized and 125 of the 12,800 total available instruction slots are utilized depicting the available headroom. The top row of the instruction matrix contains NOPs and is called the null sequence. NOPs can be filled in automatically by the linker.

Summary

This paper describes a technique for applying synchronous data flow principles to the processing elements of network processors to scale performance in a manner that is both hardware efficient and easy to program. Hardware efficiency comes from the utilization of a pure pipeline topology for the processing element array, while ease of programming comes from a synchronous, uni-processor control-flow programming model. The simplicity of the programming model facilitates assembly language programming, which can reduce the requirement for higher level languages, which further increases efficiency. In contrast to traditional control flow development methods, the approach presented results in a higher level of confidence in hardware resource requirements determined during the architectural phase of a project (reducing the risk of not meeting the performance requirements), as well as saves time during the application verification and tuning phase of a project (which can be longer than the actual coding phase).