A Case for Vector Network Processors

Madhusudanan Seshadri and Mikko Lipasti
University of Wisconsin - Madison, Electrical and Computer Engineering
http://www.ece.wisc.edu/~pharm

Abstract

The explosive growth of Internet traffic and the increasing complexity of the functions performed by network nodes have given rise to a new breed of programmable microprocessors called network processors. However, network processor design is still in its infancy and has not reached a level of maturity commensurate with currently available commodity microprocessors. A variety of architectures are being proposed and implemented. Designs employing superscalar, chip multiprocessing (CMP), simultaneous multithreading (SMT) and very long instruction word (VLIW) techniques have been developed. In summary, current solutions attempt to exploit the abundantly available packet parallelism by using a multiple instruction multiple data (MIMD) approach. We propose a single instruction multiple data (SIMD) approach to packet processing. We highlight the inefficiencies in current network processor implementations and show how a vector processor can address these problems effectively. We tackle the problem of control flow variations in packet processing using a combination of intelligent hardware techniques and support for variable number of instruction streams (XIMD). Specifically, for the case of layer 3 routing, we show how a moderately sized cache of 2 KB can capture over 99% of routing table lookup results and enable efficient SIMD processing. We demonstrate the feasibility of our approach with vector algorithms for IP forwarding and network address translation. We also show how this approach can be extended to more complex packet processing applications.

1 .0 INTRODUCTION

The need for programmability and in-the-field upgradeability has resulted in network processors replacing fixed function ASICs in the market for network routers and switches. Network processors constitute a fast-growing segment of the microprocessor industry and revenues are expected to increase at a compound annual growth rate of nearly 63% [1]. Network processors being developed today show considerable variation in architectural characteristics. There are several reasons for this phenomenon. The growing market for these processors has attracted many startup companies and there is a natural variety as each company tries to distinguish its product from others. Market pressures have precipitated the development of network processors without a clear understanding of network applications and scalability requirements. Also, the lack of standardized benchmarks has greatly impeded an objective comparison of solutions from different vendors. Most of the proposals to date advocate a multiple-instruction, multiple data (MIMD [8]) approach to packet processing. An analysis of some of the successful architectures today (from Intel, IBM and Motorola) reveals the popularity of chip multiprocessing along with hardware multithreading in the data plane. The generic network processor contains several programmable micro-engines that run the packet processing code [2][3][4]. The micro-engines are assisted by a number of coprocessors whose functionality is fixed [2][4]. This approach limits the flexibility and general purpose programmability of the network processor. We propose to eliminate hard-wired coprocessors and implement their functionality using software executing on the micro-engines themselves.

Furthermore, in the conventional approach, each micro-engine has its own instruction store and operates independently of other micro-engines in classic MIMD [8] fashion [2][3][4]. However, the micro-engines usually perform the same processing function on packets streaming through the switch. In other words, they execute the same instruction sequence. This behavior matches more closely with the single-

instruction, multiple data (SIMD [8]) paradigm**.**

At first glance, there appears to be a requirement for a MIMD approach for the following reasons:

1. A packet is usually subject to several filtering rules and it is possible for the packet to get dropped if it fails any *one* of these rules.
2. A variable number of steps is usually required to search lookup tables organized in a multi-level hierarchy in memory.
3. Packet payload processing applications execute a variable number of instructions depending upon the packet size.

These factors among others contribute to control flow variations between packets**.** In later sections, we describe concepts like a routing cache (for layer 3 routing), vector predicates, and support for variable instruction streams (XIMD [31]) that can effectively counter the effects of control flow variations.

Vector processors usually feature high bandwidth memory subsystems. Such high bandwidth memory subsystems are also necessary in network processors, which have to buffer packets from multiple high-speed physical links. We propose applying existing technology for building efficient, high-bandwidth memory subsystems for vector processors to the network processor problem domain. This approach provides a general purpose programmable system without the shortcomings of existing coprocessor-based and MIMD designs. Specifically, a SIMD or mostly SIMD (under XIMD [31] mode) approach enables high-performance memory subsystems that can easily exploit the predictable memory reference patterns exposed by the vector instruction set. These memory reference patterns are easily obscured in a MIMD system by the chaotic intermingling of references from seemingly unrelated and unsynchronized packet processing threads. Specifying these references using standardized and well-understood vector processor primitives (vector loads and stores, index registers for scatter/gather, etc.) greatly simplifies the implementation of a high-performance, high-bandwidth memory subsystem.

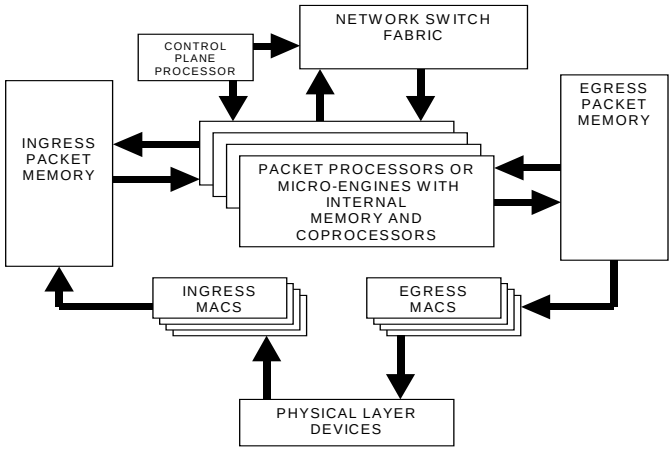

Figure 1. Network Processor Structure.

2 .0 BACKGROUND AND MOTIVATION

The overall structure of a typical network processor is shown in Figure 1 (we use the IBM Rainier design as a representative case study [2]). One or more programmable packet processors are attached to ingress (input) and egress (output) memories that are in turn attached to medium access controllers (MACs) that communicate with the physical line interfaces the network processor is servicing. Figure 2

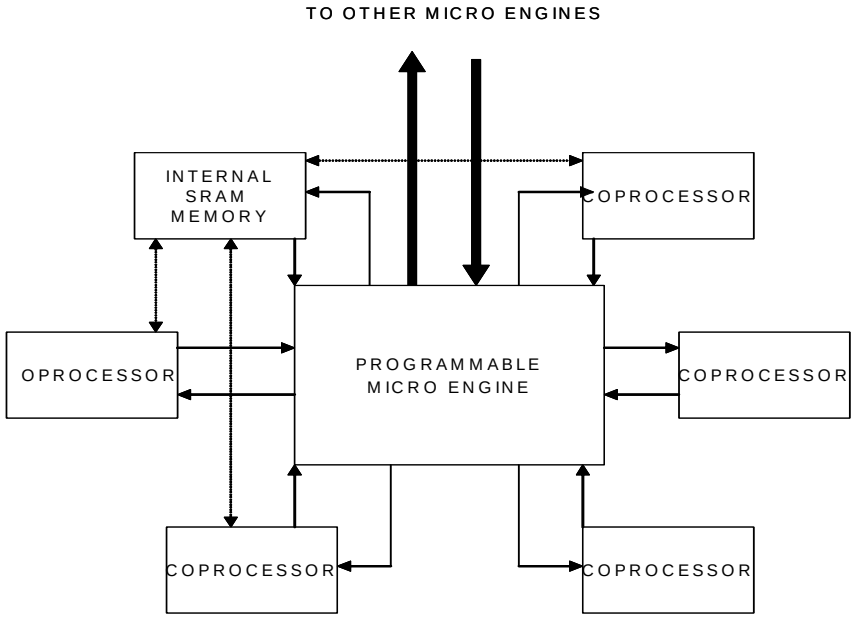

Figure 2. Conventional Packet Processor Details.

illustrates the internal details of a programmable packet processor. It consists of a programmable micro-engine that is assisted by several fixed-function coprocessors for packet processing tasks. The micro-engines perform only very simple operations on the packets while the hardware assists perform important functions such as checksum computation and tree searches. Consequently, even though the micro-engines can run arbitrary code, the programmability of the network processor is constrained by the range of functionality provided by the hardware assists. The only way to support a new application that cannot be handled by these coprocessors is to modify the hardware by adding another widget (and associated control logic) that is specific to the application. This implies a new design and validation cycle for the network processor. In doing so, one of the main objectives for developing a network processor, which is to handle new applications via programmability, is severely compromised. Alternatively, another network processor capable of handling the new application has to be developed. We believe that this is one reason for the proliferation of different varieties of network processors such as storage processors, security processors and traffic managers in the market.

A truly general purpose solution would obviate the need to design processors targeted for niche applications. At the very least, it is clear that the “programmability” of current network processors is not sufficient to encompass the entire spectrum of network applications. Eliminating application-specific logic and emulating its functionality via software running on simple, clean hardware is the key to extensibility of the network processor to new applications. A new application should need only new software, not new hardware. The challenge then, is to make the general-purpose hardware efficient enough to compensate for the absence of fast hard-wired logic.

Current designs also have to grapple with contention for shared data structures by multiple coprocessors. As these coprocessors usually operate independently of each other, intricate synchronization logic is required to guarantee correct execution. Since every coprocessor that acts on packet data has to access packet memory, the memory interconnect is quite complicated. This directly relates to the complexity of the design itself and affects the scalability of the network processor to larger configurations. Furthermore, splitting the packet processing function among multiple asynchronously operating hardware units makes

pipelining more difficult, if not impossible, and inhibits network processors from benefitting from this most basic performance optimization.

In contrast, a pipelined vector processor with a regular, logically simple structure lends itself to straightforward, well-understood design and implementation. Packet processing is an inherently data-parallel task and is well suited to architectures that exploit such parallelism. More importantly, every packet is processed the same way as it proceeds through the network switch. This implies SIMD parallelism [8]. Hence we believe a vector processor is the most efficient solution to process packets rather than parallel architectures that employ MIMD techniques [8]. SIMD can decrease instruction count and reduce dynamic instruction bandwidth requirement compared to MIMD, and thus mitigate Flynn's bottleneck considerably [7]. This is important for a network switch that has to process millions of packets per second. More importantly, vector processors can greatly ease the design of the memory subsystem. In network processors, packet memory can be accessed up to four times for a packet [5][6]:

1. Receive packet from network and store to memory
2. Retrieve packet from memory for processing
3. Write back processed packet data to memory
4. Retrieve packet for transmission

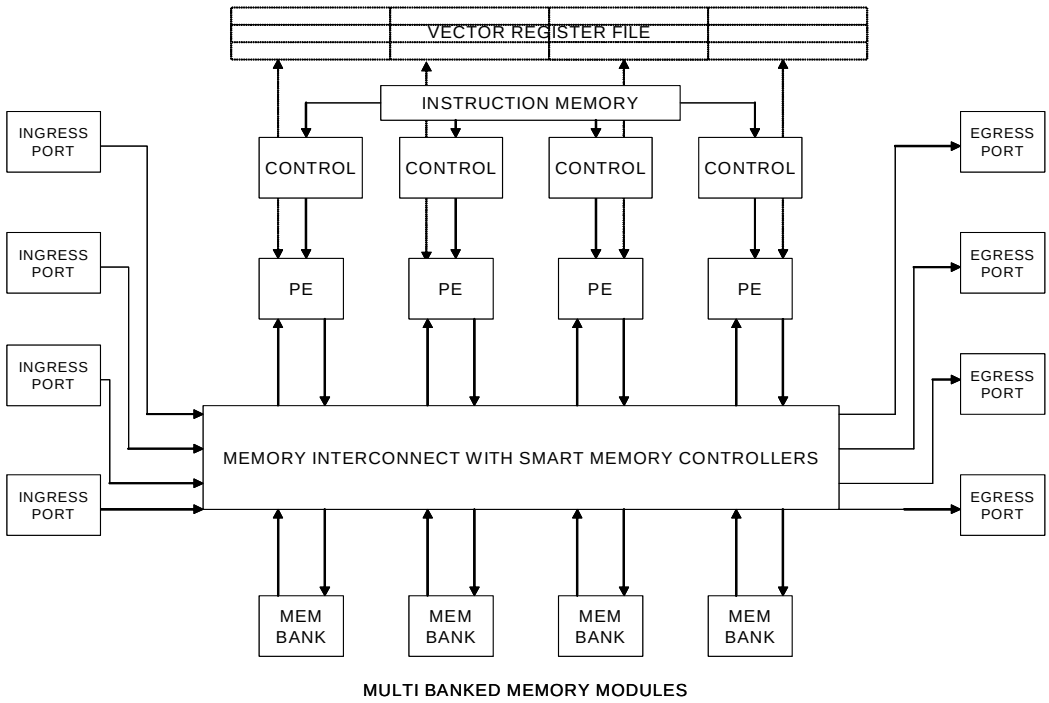

Figure 3. Proposed Vector Network Processor.

With full duplex connections at a link speed of 40 Gbps (OC-768), memory bandwidth requirements could reach 320 Gbps. A router supporting 16 such links would require 5120 Gbps of aggregate bandwidth. Since DRAM bandwidth is expected to trail far behind advances in network speeds, complex memory access schemes have to be devised to bridge this gap. In this scenario, vector architectures that access memory with regular, predictable access patterns [7] can substantially outperform MIMD architectures with unpredictable and seemingly chaotic access patterns. In the next section, we describe our proposed vector network processor architecture.

3 .0 THE VECTOR NETWORK PROCESSOR PROPOSAL

In this section, we outline our proposed approach for building network processors using a XIMD vector approach. Our proposal differs from traditional vector processors in that there are multiple control sequencers for supporting a variable number of instruction streams (as in XIMD [31]). Also, our proposed vector processor operates on arbitrary packet data rather than single- or double-precision floating-point numbers.

3.1 Overview Of The Vector Network Processor

The basic architecture of the vector network processor is shown in Figure 3. High-end routers can have as many as 16 full-duplex (input/output) ports [9]. The ports constitute the physical interface of the processor with the network and are responsible for the electrical transmission and reception of packets. The processor core contains a scalar unit (not shown in Figure 3) and a vector unit. A control sequencer for each processing element (PE) in the vector unit specifies the next instruction to execute. The instruction specifies the type of operation that need to be performed on packet data (header or payload). The packets themselves are fetched by the PEs directly from packet memory. The packets are held in the vector register file while they are being processed. After a packet has been processed, it is written back to memory if it has been modified. The PEs also determine the egress port for each packet and directly manipulates the output queue of each port to ensure that packets are transmitted on the correct line cards. Note that the structures described here are not specific to any network application. The flow of packets through the network processor is described next.

3.2 Packet Dataflow Through The Vector Network Processor

The full duplex ports can be considered as a combination of an ingress port and an egress port. The entire packet memory is statically partitioned among all ingress ports of the network processor. Each partition functions as a logical queue for packets from a particular ingress port. Packets are introduced into the queue only by the ingress port associated with the queue but may be removed by any of the egress ports. The space occupied by a packet is deallocated when the egress port transmits the packet. The vector processing unit can access the packets for the purpose of processing headers or for performing deep processing functions (e.g. encryption or decryption). There is a head and tail pointer associated with each ingress port. These pointers are used to manage the operations on the queue for that port. The ingress port extracts packets from the network and buffers them in packet memory. Of course, the memory capacity must be sufficient to avoid stalling the ingress port often. The ingress port may also maintain some internal buffers to tide over temporary stall conditions without dropping packets.

The PEs in the vector unit fetch these packets and process them in a lock step fashion. Multiple control sequencers can be used to relax this lock step execution dynamically, as in the XIMD architecture [31]. Each PE fetches packets from only one logical partition in packet memory. It uses the head and tail pointers associated with the partition to correctly fetch valid packets. In general, the number of PEs and the number of ports/partitions may not be equal. This implies that more than one PE may be accessing the same partition or a single PE may be serving more than one partition. Appropriate mechanisms must be in place to ensure that (a) all packets get processed in bounded time and (b) a packet is processed by exactly one PE. In the simple case, the PEs operate only on the equally sized packet headers in lock step (vector) fashion and peak efficiency is achieved. But when the PEs operate on packet payloads, some efficiency may be lost as they operate on packets with unequal sizes. In Section 5.5, we provide empirical data to show that this does not happen often. In the case when it does happen and some PEs idle while others are busy processing packets, conditional execution using vector-mask registers or vector ‘predicates’ [7] can be used at the expense of some efficiency. Under pathological conditions, switching to a XIMD execution mode (detailed in Section 3.3) helps preserve packet processing efficiency.

Egress Queue Management. The final step in packet processing is to schedule the packet to the

correct output port. This process is illustrated in Figure 4. To facilitate this operation, each port maintains a queue that is two dimensional in nature. The PEs (producers) write into the queue and the entries are consumed by the port (consumer). A head and tail pointer are provided to manage the queue operation. An entire row of the queue may be written during each processing cycle. The number of columns in a queue is equal to the number of PEs in the network processor. This will allow all PEs to schedule their packets to the same port in the same cycle if necessary. The number of rows in the queue will depend on the speed mismatch between the PEs and the port. Ideally, enough rows are available so that the PEs do not wait for the port to free up queue entries. Each queue entry consists of a packet address in memory. During each processing cycle, the PEs write into a row of the queue and update the tail pointer to point to the next row. The port walks through the queue in row major order, collecting packet addresses, fetching them from

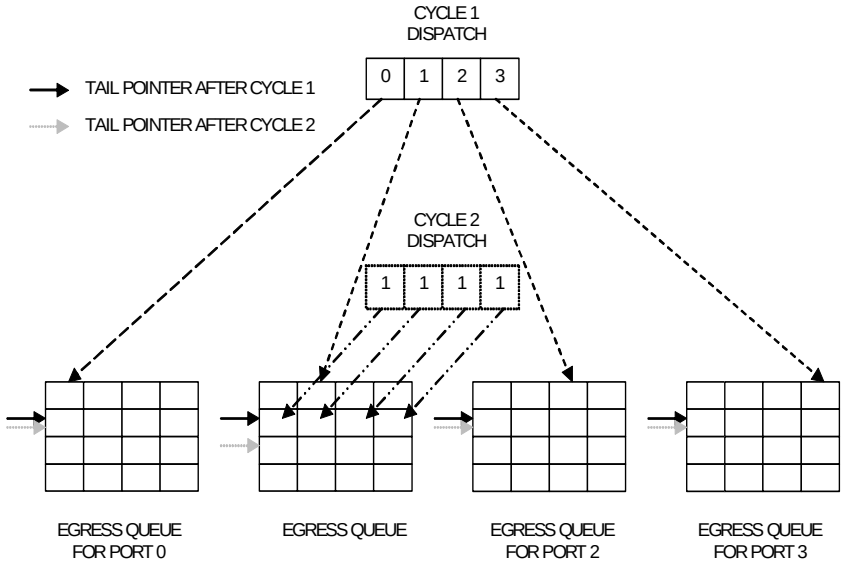

Figure 4. Egress Queue Tail Pointer Updates.

memory and dispatching them over the network. The port updates the head pointer when it has finished processing all the packets in a row. Thus the process of packet retrieval (from memory) and dispatch (to output ports) is achieved directly by the vector unit without any glue logic. This provides a scalable general purpose solution. It should be noted that packets always stay in packet memory and are copied only to the vector register files for processing by the PEs. The egress ports fetch the packets directly from memory. The entire packet processing task is completed in an essentially ‘copy-free’ or ‘zero-copy’ manner. This is in contrast with conventional designs where packets are copied frequently to facilitate independent operation on packet fields by different functional units. This reduces the network processor’s memory bandwidth requirement substantially.

As mentioned earlier, the PEs update the tail pointer of each output port queue at the end of a processing cycle. If packets are not scheduled to a particular port during a particular cycle, then updating the tail pointer of that port in that cycle is wasteful of queue entries. Instead, tail pointers can be selectively updated using the following simple algorithm. At the start of the algorithm, assume that the output port numbers have been determined by each of the PEs and stored in vector register vr1. Also assume that there is a bit-mask vector which determines if the tail pointer of a particular port is updated during a particular

```
cmpv vr1, “0”   /* compare and set element 0 of bit-mask vector if there is a match */
cmpv vr1, “1”
cmpv vr1, “2”
cmpv vr1, “3”
cincrv vr2      /*conditional increment of tail pointers stored in vector register vr2*/
```

Figure 5. Vector Assembly Language for Tail Pointer Update.

cycle. The number of elements ‘n’ in the bit vector equal the number of egress port queues.

1. Compare the port numbers in vr1 with port 0. If there is at least one match, set the bit corresponding to port 0 in the bit-mask vector to 1. Otherwise, set the bit to 0.
2. Repeat step 1 for each of the ports 1 to n-1.
3. Update the tail pointers of each port depending upon the values in the bit-mask vector.

At the end of the iterations in step 2, the bit vector would be fully set up to determine which of the tail pointers should be updated. Each of the tail pointers can then be conditionally updated. A kernel which implements the algorithm for 4 output ports using a pseudo vector assembly language is shown in Figure 5.

3.3 XIMD Vector Approach - A Robust Solution To Control Flow Variance

Many packet processing applications exhibit substantial inter-packet data parallelism and are highly amenable to simultaneous processing of multiple packets. A traditional vector processing approach exploits this parallelism but achieves peak efficiency only when dealing with applications that process packets in a control flow invariant fashion. This is a fundamental limitation of a pure SIMD model that requires the same sequence of instructions to be executed on all elements of a vector group to achieve data parallelism. Even with this constraint, efficiency can still be affected by instructions with non-deterministic execution times like memory operations in the presence of caches. In Section 5, we provide empirical data and evaluate architectural techniques that reduce control flow variance between packets. However, it is decidedly advantageous to use an architecture that can exploit data parallelism even in the presence of control variations. This can greatly benefit applications such as server load balancing (SLB).

In SLB, a load balancer distributes incoming network traffic among several machines constituting the server. Each packet is subjected to filtering rules that determine the particular machine to which the packet should be delivered. Filtering rules can be complex and many such rules could potentially be applied to a packet. Packets in a vector group may not be subjected to the same set of rules. This implies control flow variance between packets. This can be handled in a traditional vector processor by using predicated execution. Though instructions for all rules are fetched by each processing element, if a particular rule is to be applied to only one packet in a vector group, execution of the rule on other packets can be inhibited by setting the vector-mask register appropriately. This permits the vector processor to keep processing packets in parallel although at reduced efficiency. The key bottleneck here is the presence of only a single instruction stream to process packets.

In contrast, the XIMD architecture allows multiple instruction streams to execute simultaneously in the machine [31]. The number of such streams can be varied dynamically at run time. XIMD essentially extends the SIMD model of data parallelism to accommodate control parallelism as well. XIMD differs from MIMD in the ability to vary the number of instruction streams and the number of functional units executing in each instruction stream. As mentioned earlier, there is a separate controller for each processing element. The controller governs the instruction stream executed by the associated PE. Each PE can

potentially execute a unique instruction stream. It is also possible for a subset of PEs to be grouped together, executing a common instruction stream.

There are several advantages to this configuration. Consider the case of a network processor executing a single application with little or no inherent control flow variance like layer 3 forwarding. All the PEs could start out operating as a single group on a set of packets. During the route table cache lookup step, if one or more PEs encounter a cache miss, they could be dynamically combined together to form one "slow" execution group while the others are combined into another "fast" group that completes the packet processing quickly. The "fast" group, upon completion, moves on to the next set of packets without waiting for the "slow" group to complete. If the "slow" group happens to complete while the "fast" group is in the middle of processing a set, then it just waits for the "fast" group to catch up before the two groups are combined together to process the next set of packets. This operation is illustrated in Figure 6. Since, on average, the number of PEs in the "slow" group is expected to be far less compared to that in the "fast" group, the efficiency of the network processor is not compromised. This solution with a XIMD processor is an improvement over a pure SIMD processor where all PEs operate as a single group in each processing cycle. As we have described it, the "slow" group is synchronized with the "fast" group as quickly as possible. This is done to preserve a uniform packet data access pattern to the memory subsystem. If the synchronization were not enforced, then the memory access pattern degrades to that in a MIMD system where access patterns are seemingly random and chaotic.

Next, consider the case of a network processor that supports multiple applications simultaneously. In this case, each application could be assigned a subset of the available PEs that execute the instructions corresponding to that application only. This is not possible with a purely SIMD vector network processor. Though the memory access pattern across applications is bound to be chaotic, within an application, it should still be very predictable. A vector memory subsystem with resources to handle more than one type of access pattern can service multiple applications effectively. In a network processor, packet data transfer constitutes the bulk of memory operations. Application code is typically small and is expected to fit in on-chip caches. This prevents the chaotic instruction access patterns caused by the intermingling of several independent instruction streams from polluting the uniform access patterns of packet data.

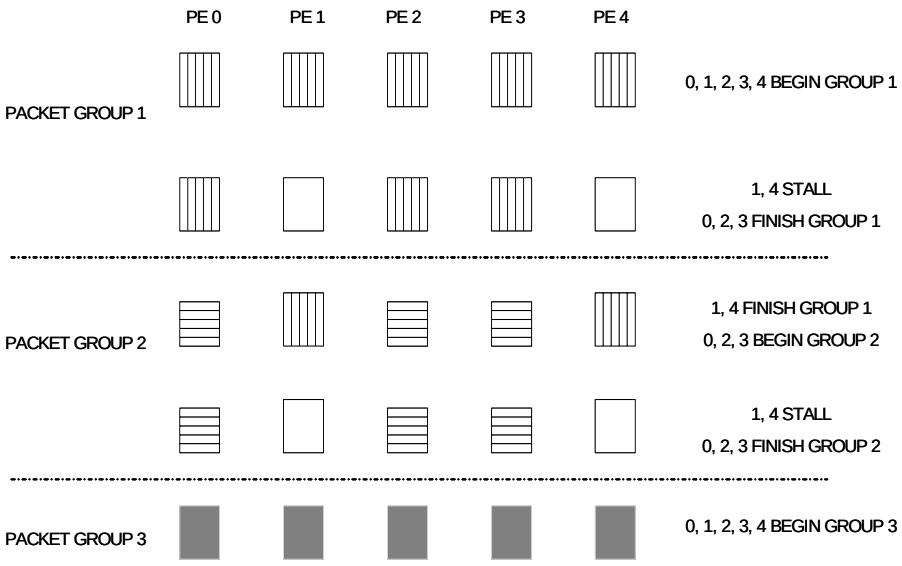

Figure 6. Packet processing in the presence of control flow variance.

3.4 Memory Subsystem

Memory architectures for vector machines have been studied extensively for many years and a variety of techniques have been developed to optimize their performance. One of our main motivations for a vector processing approach is to leverage this existing body of work to enhance the design of a network processor. The memory bandwidth requirements for a network processor are tremendous and it is not clear if DRAM technology will scale to meet these demands. In any case, it is not expected to do so in the near future. Hence the design of the memory subsystem is crucial to the performance and overall cost of a network processor. Predictability of memory references goes a long way in improving memory system performance, since the memory controller can effectively exploit bandwidth-enhancing DRAM features if it can accurately anticipate future reference behavior. We treat packet memory as a set of queues (one for each ingress port). Each ingress port streams through its queue filling it up with packets thereby making the ingress references very predictable. Memory operations issued by the vector unit can be efficiently supported by the use of *scatter-gather* operations implemented using intelligent memory controllers. Rau et al. advocate a pseudo-random interleaving scheme to make the memory system stride-insensitive [11]. Mathew et.al. describe a parallel vector access unit for SDRAM memory systems [10]. Their scheme uses a small set of remapping controllers and multiple interleaved DRAM banks each with its own bank controller (BC). The remapping controllers support three types of scatter-gather operations and can be configured by applications. The controllers handle vector operations directly without the need to serialize the stream of element addresses. The bank controllers implement hardware algorithms to efficiently support reads and writes to distributed memory locations. Iyer et.al. analyze a memory architecture for fast packet buffers [12]. They propose using reasonably sized SRAM buffers to quickly move packets from and to packet memory. In their scheme, the packet memory is managed as a set of queues similar to ours. Both techniques described above are orthogonal and can be used simultaneously. While the former technique can be used to handle memory operations by the vector unit, the latter can be used for references by the ingress and egress ports. We leave detailed study of memory system design alternatives to future work; however, we are confident that the existing body of knowledge for vector processor memory subsystem design will apply to and greatly benefit our proposed vector network processor.

4 .0 ALGORITHM CASE STUDIES

We consider three common network applications in detail and show how they are both amenable to vector processing. These case studies--layer 3 IPv4 forwarding, layer 3 IPv6 forwarding and network address translation--are representative of the kinds of applications that network processors execute. In the following sections, we describe how to implement these algorithms in our vector network processor.

4.1 IPv4 Layer 3 Forwarding

First we describe IPv4 forwarding at layer 3 of the OSI model. RFC 1812 lays down the requirements for IP version 4 routers [13]. The IP forwarding algorithm operates as follows:

1. Validate IP header – this involves validation of the IP version number, checksum, IP header length and total packet length.
2. Process IP options
3. Determine if packet has to be forwarded or consumed locally or both
4. Lookup routing table to determine next hop IP address
5. Apply filtering rules for packets
6. Decrement the packet’s time to live field
7. Recalculate the IP checksum
8. Perform fragmentation if necessary
9. Determine link layer address of the packet’s next hop and encapsulate IP in a link layer frame

IP checksum computation is discussed in RFC 1071 [14] and efficient checksum updating in RFC

1141 [15]. We now proceed to develop an assembly language implementation of the forwarding algorithm. Our vector instruction set is loosely based on the DLXV instruction set described in [7]. We do not consider multicast/broadcast packets because RFC 1812 maintains that forwarding of IP multicasts is still experimental. We assume a routing cache to speed route lookups (refer to Section 5.2 for more details). We neglect packets with IP options; processing of IP options is a source of control flow variability between packets, but the fraction of IP packets with header options is typically less than 0.003% in the internet [16]. It is also not unusual for routers to ignore these options even if they are present. Even in the case where they are present and the router does process them, they occur so infrequently that the performance penalty should be very minimal. For the same reason, we do not consider fragmented IP packets which constitute less than 0.2% of the packets in the Internet [16]. We also neglect the case where packets are destined for the router itself because it can be trivially handled without table lookups. We consider only IPv4 packets. Due to space constraints, we do not implement functions such as 'martian' address filtering, link layer encapsulation and enforcing router administration rules. However, SIMD efficiency is not sacrificed because *all* packets are subjected to these rules.

In summary, our implementation of layer 3 IP forwarding, shown in Figure 7 and Figure 8 captures the essence of the routing algorithm while ignoring some of the uninteresting corner cases. Consequently, some of the processing steps outlined above may not appear in our code. It is possible for a packet to be discarded for a variety of reasons in the course of processing it. In such cases, the bit corresponding to the packet is cleared in the bit-mask vector and further processing of the packet is inhibited. Discarded packets can be scheduled to a dummy egress port which removes the packet from memory. Note that the bit-mask vector is a common feature used to implement conditional execution in vector processors.

We assume a big-endian machine with byte-addressable memory and 32-bit wide datapaths. Scalar registers are designated as R<number> and vector registers appear as V<number>. For the purposes of this code, the following assumptions are made: (1) As many registers as are needed are available. (2) Base address for packet memory is stored in register R0. (3)The starting addresses of packet headers specified as offsets from the base address are stored in register V0 (the index vector). We do not show the code which synchronizes with the ingress ports to obtain the addresses stored in V0. Also note that the code is only meant to be a proof-of-concept demonstration and is not optimized for performance. The IPv4 header format appears in Figure 9.

4.2 IPv6 Layer 3 Forwarding

IPv6 is a new Internet protocol designed to supersede IPv4. It provides expanded addressing capabilities, a simplified header format and improved support for header options and extensions. IPv6 specifications can be found in RFC 2460 [30]. Figure 10 depicts the IPv6 header format. IPv6 makes layer 3 forwarding even more amenable to vector processing compared to IPv4. IPv6 requires IP options to be specified in extension header fields which are examined almost exclusively by the destination node only. Unlike IPv4, fragmentation in IPv6 is performed only by source nodes and not by routers along a packet's delivery path. These features allow the router to support IP options and fragmentation without introducing control flow variance in processing packets. The routing algorithm for IPv6 is similar to that for IPv4 and so, we do not provide a assembly language implementation for IPv6 but refer the reader to the one for IPv4 instead.

4.3 Network Address Translation

The next case study we consider is Network Address Translation (NAT). NAT addresses the problem of IP address depletion through the concept of address reuse. A stub domain is defined as a domain that only handles traffic originated or destined to hosts in that domain. NAT proposes splitting up the IP address space into global and local addresses. Each stub domain is assigned a global address (the NAT address). The global addresses are unique throughout the Internet. Hosts within a stub domain have 'local'

```
1.  lvi    V1, (R0+V0)          ; load V1 using V0 as index vector - gather 1st word of packet
2.  addvi  V0, 4                ; increment index vector to point to next word
```

Repeat steps 1 and 2 four times to gather the rest of the words in V2, V3, V4 and V5. The entire five words of the IP header is thus brought into vector registers.

```
/* IP header validation */
3.  andvi  V6, V1, #0xf000000   ; extract Internet Header Length (IHL)
4.  srlvi  V6, V6, #24          ; shift right logical operation to enable comparison
5.  sgevi  V6, #20              ; set bit in bit-mask only if IHL >= 20 (min valid IHL
                                  length)
6.  andvi  V7, V1, #0xffff      ; extract total length of IP packet
7.  sgtv   V7, V6               ; total length should be > IHL
8.  andvi  V7, V1, #0xf0000000  ; extract version number
9.  srlvi  V7, V7, #28
7.  seqvi  V7, #4               ; version number should be 4
8.  addv   V7, V1, V2           ; begin 16-bit checksum validation
9.  addvc  V7, V7, V3           ; 1's complement addition – add with carry
10. addvc  V7, V7, V4
11. addvc  V7, V7, V5
12. andvi  V6, V7, #0xffff0000  ; extract most significant 16 bits of 32-bit checksum
13. srlvi  V6, V6, #16
14. andvi  V7, #0xffff          ; retain least significant 16 bits alone in v7
15. addv   V7, V7, V6           ; checksum reduction operation
16. andvi  V6, V7, #0x10000     ; extract 17th bit of checksum – this is the carry
17. srlvi  V6, V6, #16
18. andvi  V7, V7, #0xffff      ; retain only 16 bits of checksum
18. addv   V7, V7, V6           ; add carry - complete checksum computation
17. seqvi  V7, #0xffffffff      ; the checksum must be all 1's to be valid
```

/* route table lookup to determine egress port number - the V5 register currently holds the destination IP addresses currently being processed, hence it is used as an index vector to perform the lookups; R1 is assumed to hold the base address of route table in memory – lrvi is a special route lookup load operation*/

```
18. lrvi   V10, (R1+V5)
```

Figure 7. layer 3 Routing Example (steps 1-18).

addresses which are unique only within the domain. These addresses can be reused within another domain. With this scheme, stub domains can be allowed to grow without depleting IP addresses. More details can be found in [17]. NAT is implemented as follows. A NAT box at the edge of a stub domain replaces the source IP field of every outgoing packet with the NAT address. A unique port number (UPN) is assigned to every unique {source IP, source port} pair within the domain. This implies that a new mapping has to be performed for only the first packet of a connection. Subsequent packets belonging to the same connection get the UPN assigned previously. The source port field of every outgoing packet is replaced with the UPN. To the external world, all the connections from the stub domain appear to come from a single IP address. The {source IP, source port} to {UPN} mappings are maintained in a NAT table. To route an incoming packet to the correct host, the NAT table is looked up with the objective of matching on the destination port field. If a UPN matching the destination port is found, then the packet is destined for that stub domain and the destination IP and destination port fields are replaced with the corresponding source IP and source port values in the table. If a match is not found, the packet is simply discarded [18].

```
/* check time-to-live (TTL) field; decrement if TTL is valid */

19. andvi  V7, V3, #0xff000000  ; extract TTL field
20. sravi  V7, V7, #24          ; perform arithmetic shift to retain sign
21. sgtvi  V7, V7, #0           ; check that TTL > 0
22. subvi  V7, V7, 1            ; decrement TTL by 1
23. andvi  V3, V3, #0xffffff    ; clear TTL field in word 3 of packet
24. sllvi  V7, V7, 24           ; prepare TTL field prior to merging with header
25. orv    V3, V3, V7           ; rewrite TTL field in the header

/* update the checksum field */

26. andvi  V7, V3, #0xffff      ; extract checksum field
27. addvi  V7, V7, #1
28. andvi  V6, V7, #0x10000     ; extract 17th bit of the preceding addition
29. srlvi  V6, V6, #16
30. addvi  V7, V7, V6
31. xorvi  V7, V7, #0xffffffff  ; do ~(V7) – this is the checksum
32. andvi  V3, V3, #0xffff0000  ; clear checksum field
33. orv    V3, V3, V7           ; write the checksum field

/*  special output port store instruction to update the ports – the output ports are in vector V10 – the load
register for the route lookup operation*/
34. srv    V10

/* Code to update the tail pointers of egress queues  - detailed in Section 3.2 */
```

Figure 8. layer 3 Routing Example (steps 19-33).

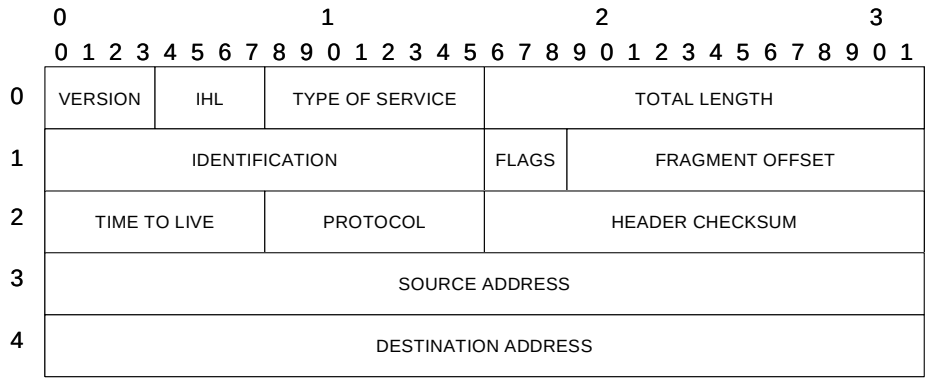

Figure 9. IPv4 Header format excluding OPTIONS fields.

Due to space constraints, we restrict ourselves to a high level analysis of the algorithm. The first observation is that *every* outgoing packet is subjected to the remapping process described, providing plenty of inter-packet SIMD parallelism. The {source IP, source port} pair of the packet is looked up in the table. The NAT table is usually small and entries are deleted every 2-3 minutes to keep the size of the table small [18]. The lookup process yields one of two results. If the packet is the first packet of a connection, then the lookup fails and *scalar* code needs to be executed to set up a new mapping for this connection. This will reduce the efficiency of the vector processor but it has to be noted that this will occur for only the first packet. The lookup will yield a UPN for all subsequent packets of the connection. Hence, the percentage of

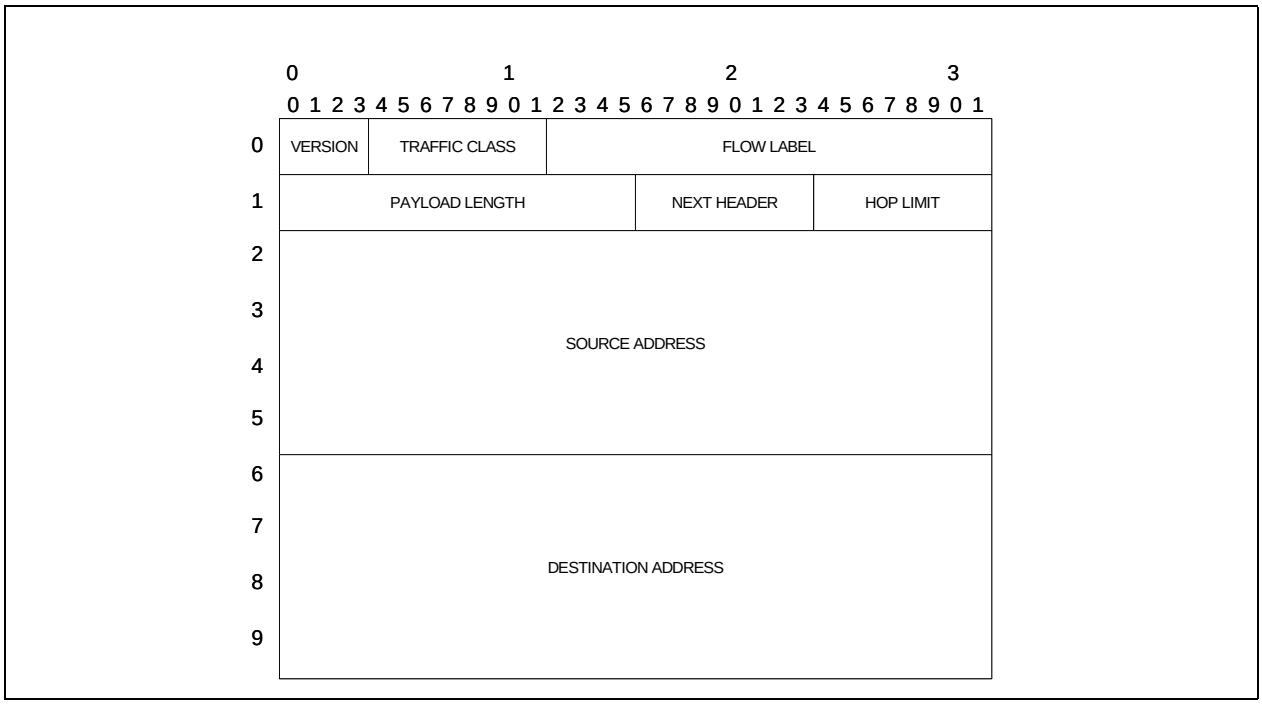

Figure 10. IPv6 Header format.

lookups that fail will be small. Also, the small size of the table implies that a single level hash-based lookup is probably sufficient. This is in contrast to routing tables which are huge and are organized as hierarchical tables. A single level lookup eliminates differences in lookup time between packets that are being processed simultaneously. Consequently, the efficiency of the vector unit is not compromised. Even if single level lookups are not sufficient, a generic cache (discussed in Section 5.2) could effectively eliminate this bottleneck. Qualitatively, there will be plenty of temporal locality considering that a new mapping will be used for *all* future packets from the same connection.

Operations on incoming packets are even more homogenous. If the lookup fails, the packet is simply dropped without affecting the processing of other packets. Hence, the lookup process can be vectorized efficiently. Following the lookup process, functions like modification of certain fields and recalculation of checksums are performed, which are very similar to the operations for layer 3 routing detailed above, and are easily vectorizable.

Chandra et.al. note that complete applications that run on real systems typically include multiple functions like the ones described above [29]. Since the functions are vectorizable individually, it follows that the applications are also vectorizable.

5 .0 EXPERIMENTAL RESULTS

In this section, we report empirical results that support our proposed vector network processor. First of all, in Section 5.1, we describe our methodology for collecting and analyzing packet traces. In Section 5.2, we describe the problem of routing table lookups, explain how it appears to be non-vectorizable, and then propose and evaluate a routing table cache that enables efficient vectorization. In Section 5.3 and Section 5.4 we present characterization data that supports our assertion that routing table caches are a scalable solution, even in the presence of increasing levels of traffic. Finally, in Section 5.5 we report on the distribution of packet sizes in the traces we collected and argue that deep packet data processing applications that require processing of the entire packet body (not just the header) can still be vectorized with relatively high vector efficiency, hence obviating the need for MIMD-like control flow.

5.1 Packet Trace Collection And Analysis Methodology

We collected packet header traces from the Computer Systems Laboratory (CSL) edge router at University of Wisconsin-Madison. The traces were collected in a non-intrusive manner by snooping on a mirror port of a switch connecting the CSL router with the outside world. We collected about 60 traces, one for each day of June and July, 2001. The traces are all of the same size (2 Gigabytes). Each trace file contains about 48 million equally sized entries. Each entry contains a 64 bit timestamp and 20 bytes of IP header (IP options, if present, were discarded). The traces captured a router load variation from 1.36 million packets/hr to 9.34 million packets/hr. To compensate for artificial locality in our traces as a result of our being at the edge of the Internet, we created an interleaved trace of 90 million packets for our caching study. We used 23 trace files to create the interleaved trace. The trace was created using the following algorithm. At the start of the algorithm, the file pointers for each of the 23 files point to the start of the file.

1. Pick a packet at random from file 1
2. Repeat step 1 for each of the remaining 22 files
3. Repeat steps 1 and 2 always making sure that no packet is picked twice and that every picked packet is subsequent in time to all previous packets picked from the same file.

We believe that this scheme is robust in that while it captures the interleaving of packets likely to be seen in backbone routers, it does not completely destroy the natural locality that is bound to exist among packets belonging to a particular trace. The results of our studies are detailed in the following sections.

5.2 Routing Cache Study

The basic algorithm for layer 3 IP forwarding can potentially reduce the efficiency of the network processor during the routing table lookup step. This is because routing tables are quite large and are frequently organized as hierarchical tables. Hence, a single lookup operation will not be sufficient in most cases. The nondeterministic number of lookups implies that without additional effort, the efficiency of the vector unit will be badly affected, as PEs which complete their lookups in a single step wait for other PEs which take longer to complete. A simple cache that holds the results of route table lookups will allow lookups to complete in a single step and solve this problem. Such a cache can also hold the results of NAT table lookups and enhance the performance of that application. The viability of route table caches has been studied extensively in the literature [19][20][21]. We validate previous studies and present data that indicate trends favorable to caching schemes.

We collected miss ratio data for a few points in the design space for routing caches indexed simply by the destination IP address. We present the results for a line size of 1 block because the results for a larger line size are uninteresting. This is expected considering that there is no spatial correlation among IP addresses for route lookups. Each block is a byte wide. We chose this size because the ultimate result of a routing table lookup is a port number which is unlikely to exceed 256 even for large switching fabrics. Note that the cache parameters chosen here may appear specific to the layer 3 IP forwarding application. However, this was done merely to simplify our study. The results we present are easily extensible to a ‘generic’ cache with different parameters. We performed the study using the cache simulation library ‘libcheetah’ – a part of the SimpleScalar 3.0 toolset. The results are shown in Table 1.

The results indicate the effectiveness of the routing cache even at small sizes. As one might expect, the importance of increased associativity decreases with increasing cache sizes. The miss ratios are down to 0.22% with only a 2KB 4-way associative cache. For a 32KB 4-way set associative cache, the misses are all compulsory (0.015%) and further improvements cannot be obtained. Even after accounting for our location at the periphery of the Internet and routing cache flushes due to periodic updates, these results are very encouraging. Furthermore, the indexing scheme used is the simplest possible. This leads us to believe that there is significant locality in the destination IP address stream that can be effectively exploited.

Table 1: Routing Cache Results

Cache Size	Associativity	Miss Ratio
1K	1	2.5%
	2	1.11%
	4	0.80%
	8	0.19%
2K	1	1.85%
	2	0.51%
	4	0.22%
	8	0.16%
32K	1	0.04%
	2	0.02%
	4	0.015%
	8	0.015%

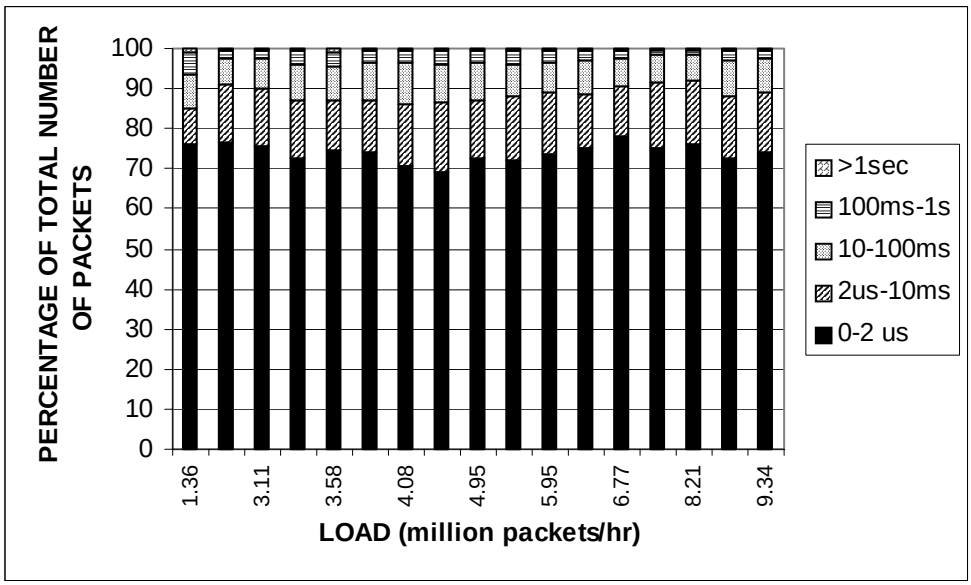

Figure 11. Temporal Locality of Destination IP Addresses.

5.3 Temporal Locality Of Destination IP Addresses

To further justify our assertion that routing table lookups can be cached, we characterized the sensitivity of destination IP address temporal locality to router load. To do so, we measure the time elapsed between two consecutive references to the same destination IP address. This indicates the 'temporal locality' in the destination IP address stream. We define six categories or 'buckets' that cover unequal ranges of

time. The time resolution of our measurements is no finer than a microsecond due to the limitations of the trace collection tool. Each packet goes into exactly one bucket depending upon whenever the next packet that contains the same destination IP address happens to be seen. For example, if packets 1, 3 and 5 share the same destination IP address, and less than 2 µs elapse between packets 1 and 3, then packet 1 would go into the 0-2 µs bucket; packet 3 would be temporarily unassigned. Similarly, if 5 ms elapse between 3 and 5, then packet 3 would go into the 2µs-10 ms bucket only.

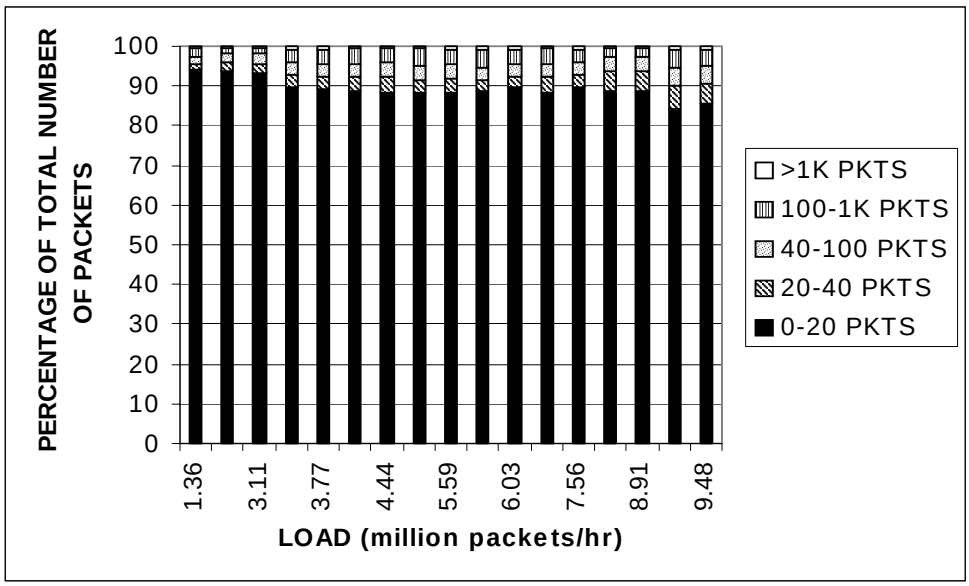

Figure 12. Working Set Analysis of Destination IP Addresses.

Our findings are summarized in Figure 11. The Y axis shows the fraction of total number of packets in each bucket. The X axis plots the router load variation. The graph shows that despite an almost order of magnitude variation in router load, the temporal locality characteristics remain nearly constant. A large fraction of packets (70% – 80%) occupy the 0-2 µs bucket. This shows that temporal locality is insensitive to router load, at least in the microsecond scale.

5.4 Working Set Analysis

To further characterize temporal locality of destination IP addresses, we performed a similar study that complements the results reported in Section 5.3. We measure the number of packets elapsed between two consecutive references to the same destination IP address. As before, we define a set of buckets that cover the packet count range. The mechanism of assigning a packet to a bucket, the X axis and Y axis semantics are all similar to Section 5.3. The results are shown in Figure 12, and show that a destination IP address in the trace has an 85%-95% (depending on router load) probability of matching a destination IP that occurred within the last 20 packets in the trace. Again, this probability is relatively insensitive to router load variation shown on the X axis. This implies that a simple FIFO cache holding the routing table lookup results for just the last 20 destination IP addresses (a working set of 20) would be able to capture 85% - 95% of the references. This further implies that a cache sufficiently large enough to capture several possibly overlapping working sets caused by a dramatic increase in the packet processing rate would still be very effective.

5.5 Packet Size Analysis

Packet sizes are a source of control flow variation in applications like encryption that perform

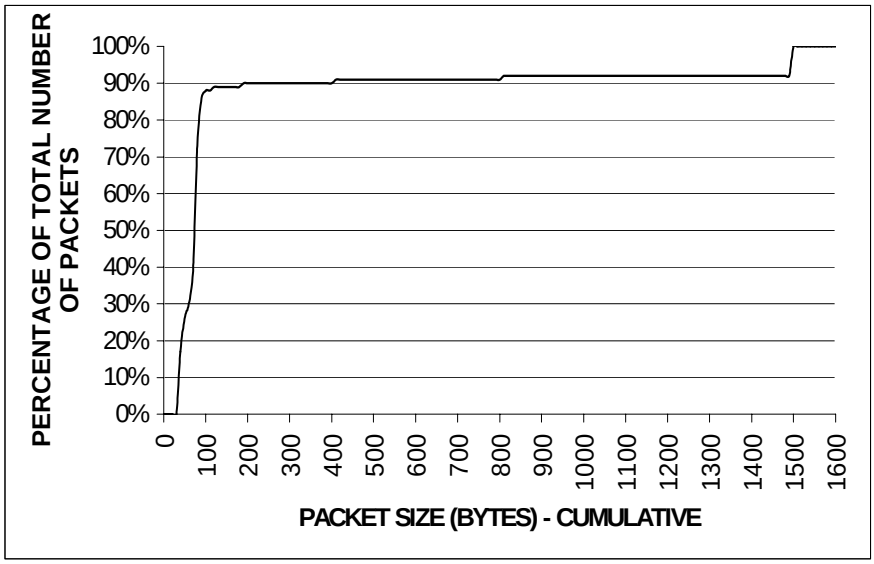

Figure 13. Packet Size Cumulative Distribution.

'deep processing' of packets. In such applications, variable-size packet payloads have to be processed in contrast to applications like layer 3 IP forwarding which only deal with fixed size headers. If packet sizes in the internet show substantial variation, then the efficiency of the vector unit will be reduced as different processing elements take varying amounts of time to process packets of different sizes. We collected packet size statistics on our interleaved trace of 90 million packets and present the results in Figure 13. The X axis plots the size of packets in bytes while the Y axis shows the cumulative percentage of total number of packets that fall within that category. So if at an X axis value of 100, the corresponding Y axis value is 80%, then it implies that 80% of packets have size 100 or less. The graph indicates that a substantial number of packets (88%) are small (less than 100 bytes). Around 8% of packets are 1500 bytes in size (corresponding to the Ethernet maximum transmission unit (MTU) value) while 4% of packets are of intermediate size. It can be inferred that vector processing efficiency is unlikely to be seriously reduced by variations in packet size, since the majority of packets are similar in size (i.e. 90% are under 100 bytes). Variations in packet size, if deemed problematic, can also be handled by falling back on the XIMD execution mode.

6 .0 SURVEY OF EXISTING NETWORK PROCESSOR DESIGNS

Networking functions can be divided into two broad categories – the data plane and control plane. Tasks such as routing table updates, statistics gathering and execution of network management protocols are relegated to the control plane. The data plane is concerned with processing the bits within each packet. Our analysis and our proposal for a vector network processor have focused on the data plane. Control plane functions are relatively infrequent and constitute the overhead of managing the network. Data plane functions have to be performed at "wire speed" as they apply to all packets flowing through the switch. Network processors usually handle the data plane and control plane functions separately. All network processors handle the data plane functions on chip. Some chips like the Intel 1XP 1200 and IBM PowerNP are stand-alone products that incorporate a general-purpose core to take care of control plane tasks. They can also serve as coprocessors to a conventional microprocessor. Others like the Vitesse IQ2000 and Agere Payload Plus chipset do not handle control plane tasks and have to be used along with a conventional microprocessor that can handle those functions. The instruction sets for network processors contain special

instructions for field extraction, byte alignment, string comparisons and boolean computations.

The microarchitecture of network processors is marked by emphasis on streaming data throughput and heavy architectural parallelism. Designs span all the way from non-pipelined such as IBM's Rainier [22] (NP4GS3) and Motorola's C-5 to aggressively pipelined like Agere's Payload Plus, SiByte's SB-1250, EZChip's NP-1 and Cisco's Toaster 2. As mentioned earlier, many processors exploit chip multiprocessing with hardware multithreading. Vitesse's IQ2000 has four 32-bit scalar cores with 64-bit memory interfaces, so each core can perform a double-word load or store in a single clock cycle. Each core has five identical register files (32 registers, 32 bits wide, triple-ported). This arrangement allows each core to run five concurrent threads of execution with fast context switching, because they don't have to save their register states in memory [23]. SiByte's SB-1250 contains multiple integrated SB-1 cores. Each core is a four-issue in-order superscalar design [24]. IBM's Rainier integrates 16 picoprocessors with one PowerPC core in a single chip. Each picoprocessor has support for two hardware threads. Two picoprocessors are packed into a dyadic protocol processor unit and share a tree search engine and internal memory [22]. Intel's IXP1200 integrates six RISC micro-engines with a StrongArm core. Each micro-engine has its own physical register file, a 32-bit ALU with a basic five stage pipeline, a single cycle shifter, an instruction control store (4K of SRAM), a micro-engine controller and four program counters – one for each thread that a micro-engine can execute in parallel [25]. Lexra's LX8000 features a cluster of MIPS-like cores with support for fast context switching and chip multiprocessing [1]. Cisco's Toaster 2 contains 16 XMC (express microcontroller) cores. Each core is an enhanced ARM-7 LIW design executing a 64-bit instruction word (that consists of two RISC instructions) per cycle [26]. EZChip's NP-1 has four different processor cores, each of which is optimized for performing a specific portion of the packet processing task. The NP-1 has 64 cores altogether [27]. XStream Logic's network processor is based on the dynamic multistreaming (DMS) technique (i.e. simultaneous multithreading or SMT). The processor core can support eight threads. Each thread has its own instruction queue and register file. The core is divided into two clusters of four threads each. Every clock cycle, each cluster can issue up to 16 instructions – four from each thread and four of the sixteen are selected and dispatched to one of the four functional units in that core for execution [27]. The core has nine pipe stages and features a MIPS like ISA [28].

7 .0 CONCLUSION

This paper proposes a vector architecture for a network processor. To take advantage of packet level parallelism in network applications, conventional network processors use a MIMD approach where a set of micro-engines assisted by dedicated fixed-function hardware perform packet processing functions. This results in a complicated architecture that is not easily scalable and is difficult to adapt to new applications. This problem can be addressed by using a vector processor. A vector processor can exploit the available parallelism by using a SIMD approach while keeping the architecture simple and efficient. The architecture of a vector network processor has been described and the feasibility of the approach demonstrated using vector algorithms for common network applications. Techniques to eliminate performance degradation in a vector processor due to control flow variations between packets have been discussed. Numerical data that support and prove the effectiveness of those techniques have been presented. A XIMD architecture which can handle control flow variations without degrading efficiency via support for multiple concurrent instruction streams has also been discussed.

References

[1] P. Glaskowsky, "Network Processors Multiply," *Microprocessor Report*, (January 2001), pp. 37-39.

[2] "Network Processor 4GS3 Overview," *IBM PowerNP NP4GS3 Technical Library*, http://www-3.ibm.com/chips/techlib.

[3] T. Halfhill, "Intel Network Processor Targets Routers," *Microprocessor Report*, (September 1999), page 1.

[4] *C-5 Network Processor Architecture Guide.*

[5] "Challenges in Designing 40-Gigabit Network Processors," http://www. ezchip.com/html/tech_40Gchallenges.html.

[6] W. Bux et.al., "Technologies and Building Blocks for Fast Packet Forwarding," *IEEE Communications Magazine*, (January 2001), pp. 70-77

[7] John. L. Hennessy and David.A.Patterson, "Appendix B: Vector Processors," *Computer Architecture: A Quantitative Approach*, 2nd. ed., Morgan Kaufmann Publishers, c1996.

[8] Michael J. Flynn, "Some computer organizations and their effectiveness," *IEEE Transactions on Computers*, vol. C-21, no. 9, 1972, pp. 948-960.

[9] "Cisco 12416 Internet Router Data Sheet", http://www.cisco.com/warp/public/cc/pd/rt/12000/12416/prodlit/itro_ds.htm

[10] B. K. Mathew, S. A. McKee, J. B. Carter and A. Davis, "Design of a parallel vector access unit for SDRAM memory systems," *In Proceedings of the Sixth International Symposium on High Performance Computer Architecture*, January 2000.

[11] B. R. Rau, M. S. Schlansker and D. W. L. Yen, "The Cydra 5 Stride-Insensitive Memory System," *In Proc. Int Conf. on Parallel Processing*, 1989, pp. 242-246.

[12] S. Iyer, R. R. Kompella, and N. McKeown, "Analysis of a memory architecture for fast packet buffers," *In Proc. IEEE HPSR*, Dallas, Texas, 2001.

[13] RFC 1812, "Requirements for IP Version 4 routers," http://www.faqs.org/rfcs/rfc1812.html

[14] RFC 1071, "Computing the Internet Checksum," http://www.faqs.org/rfcs/rfc1071.html

[15] RFC 1141, "Incremental Updating of the Internet Checksum", http://www.faqs.org/rfcs/rfc1141.html

[16] Sean McCreary and kc claffy, "Trends in Wide Area IP Traffic Patterns - A View from Ames Internet Exchange, "http://www.caida.org/outreach/papers/2000/AIX0005/AIX0005.html

[17] RFC 1631, "The IP Network Address Translator (NAT)," http://www.faqs.org/rfcs/rfc1631.html

[18] "Network Address Translation FAQ,", http://www.vicomsoft.com/index.html?page=http://www.vicomsoft.com/knowledge/reference/nat.html*track=internal

[19] T. Chiueh and P. Pradhan, "High performance ip routing table lookup using cpu caching," *IEEE INFOCOM*, April 1999.

[20] C. Partridge, "Locality and route caches," *NSF Workshop on Internet Statistics Measurement and Analysis*, http://www.caida.org/outreach/isma/9602/positions/partridge.html

[21] O. Feldmeier, "Improving gateway performance with a routing-table cache," *IEEE INFOCOM*, pp. 298–307, March 1988.

[22] K. Krewell, "Rainier Leads PowerNP Family," *Microprocessor Report*, January 2001, p 10.

[23] T. Halfhill, "Sitera Samples Its First NPU," *Microprocessor Report*, May 2000, p 53

[24] T. Halfhill, "SiByte Reveals 64-bit Core for NPUs," *Microprocessor Report*, 2000

[25] T. Halfhill, "Intel Network Processor Targets Routers," *Microprocessor Report*, September 1999, p 1

[26] L. Gwennap, "Cisco Rolls Its Own NPU," *Microprocessor Report*, November 2000, pp 19-22.

[27] Linda Geppert, "The New Chips on the Block," *IEEE Spectrum*, January 2001, pp 66-68

[28] P. Glaskowsky, "Networking Gets XStream," *Microprocessor Report*, November 2000

[29] Prashant R. Chandra et.al., "Benchmarking Network Processors," *Workshop on Network Processors*, pp. 10-17, February 2002.

[30] RFC 2460, "Internet Protocol, Version 6 (IPv6) Specification," ftp://ftp.ipv6.org/pub/rfc/rfc2460.txt

[31] Andrew Wolfe and John P. Shen., "A variable instruction stream extension to the VLIW architecture," *International Conference on Architectural Support for Programming Languages and Operating Systems*, pp 2-14, 1991