Report Number: WUCSE-2006-12

2006-01-01

Virtualizing Network Processors

Ben Wun, Jonathan Turner, and Patrick Crowley

This paper considers the problem of virtualizing the resources of a network processor (NP) in order to allow multiple third-parties to execute their own virtual router software on a single physical router at the same time. Our broad interest is in designing such a router capable of supporting virtual networking. We discuss the issues and challenges involved in this virtualization, and then describe specific techniques for virtualizing both the control and data-plane processors on NPs. For Intel IXP NPs in particular, we present a dynamic, macro-based technique for virtualization that allows multiple virtual routers to run on multiple data... Read complete abstract on page 2.

Virtualizing Network Processors

Ben Wun, Jonathan Turner, and Patrick Crowley

Complete Abstract:

This paper considers the problem of virtualizing the resources of a network processor (NP) in order to allow multiple third-parties to execute their own virtual router software on a single physical router at the same time. Our broad interest is in designing such a router capable of supporting virtual networking. We discuss the issues and challenges involved in this virtualization, and then describe specific techniques for virtualizing both the control and data-plane processors on NPs. For Intel IXP NPs in particular, we present a dynamic, macro-based technique for virtualization that allows multiple virtual routers to run on multiple data plane processors (or micro-engines) while maintaining memory isolation and enforcing memory bandwidth allocations.

2006-12

Virtualizing Network Processors

Authors: Ben Wun, Jonathan Turner, Patrick Crowley

Corresponding Author: pcrowley@wustl.edu

Abstract: This paper considers the problem of virtualizing the resources of a network processor (NP) in order to allow multiple third-parties to execute their own virtual router software on a single physical router at the same time. Our broad interest is in designing such a router capable of supporting virtual networking. We discuss the issues and challenges involved in this virtualization, and then describe specific techniques for virtualizing both the control and data-plane processors on NPs. For Intel IXP NPs in particular, we present a dynamic, macro-based technique for virtualization that allows multiple virtual routers to run on multiple data plane processors (or micro-engines) while maintaining memory isolation and enforcing memory bandwidth allocations.

Type of Report: Other

Virtualizing Network Processors

Ben Wun, Jonathan Turner, Patrick Crowley

Dept. of Computer Science and Engineering
Washington University in St. Louis, MO, 63130
bw6@arl.wustl.edu, jon.turner@wustl.edu, pcrowley@wustl.edu

ABSTRACT - This paper considers the problem of virtualizing the resources of a network processor (NP) in order to allow multiple third-parties to execute their own *virtual router* software on a single physical router at the same time. Our broad interest is in designing such a router capable of supporting virtual networking. We discuss the issues and challenges involved in this virtualization, and then describe specific techniques for virtualizing both the control and data-plane processors on NPs. For Intel IXP NPs in particular, we present a dynamic, macro-based technique for virtualization that allows multiple virtual routers to run on multiple data plane processors (or micro-engines) while maintaining memory isolation and enforcing memory bandwidth allocations.

1. INTRODUCTION

Network virtualization holds strong promise as a means for developing and deploying innovative network architectures and services in a commercially viable next-generation global-scale Internet [1]. If networks are to be virtualized in a manner similar to large-scale overlay networks such as PlanetLab [2], then virtualizable routers, such as Washington University's *diversified router* [3], must be developed.

For a router to support virtualization, the primary processing components in the line cards must also support virtualization. Modern routers use specialized processor, referred to as network processors (NPs), to handle data-plane packet processing at router ingress and egress. However, the current generation of NPs is designed to handle traditional network processing workloads, and, as such, their designs do not incorporate mechanisms to support virtualization. Specifically, most NPs are designed assuming that all NP resources will be under the software control of a single application. In the virtualization context, however, we expect a given router to support more than one *virtual router* concurrently, requiring the router's physical resources to be shared. The aim is to allow multiple third-parties to develop software that can run simultaneously in the same router safely with good performance guarantees.

Such sharing introduces new requirements for safety and provisioning. Just as operating systems rely on process scheduling and virtual memory mechanisms to provide isolation and resource allocation between multiple programs, so too must a router support isolation and provisioning of the physical resources to multiple virtual routers.

In this report, we focus on the specific requirements and challenges that arise when network processors are used to support multiple virtual routers within a single physical router. While the issues we raise apply generally to network processors, in order to keep the discussion concrete we will consider the specific examples and solutions for the Intel IXP family of network processors.

Our main contribution is to show that programs executing on the two types of processors available in NPs, the control and data plane processors, require different types of virtualization support. Traditional virtualization techniques appear effective for control processors, since they are organized as traditional processors and host fully featured operating systems. Data plane processors, such as the Intel IXP micro-engines, require other methods, however. In addition to discussing virtualization from this perspective, we describe a dynamic, macro-based approach for virtualizing the data-plane processors and enforcing memory isolation and memory bandwidth provisioning.

The remainder of this paper is organized as follows. Section 2 provides a background discussion of network processors and the Intel IXP network processors in particular. The requirements for NP virtualization are discussed in Section 3. Section 4 surveys mechanisms for

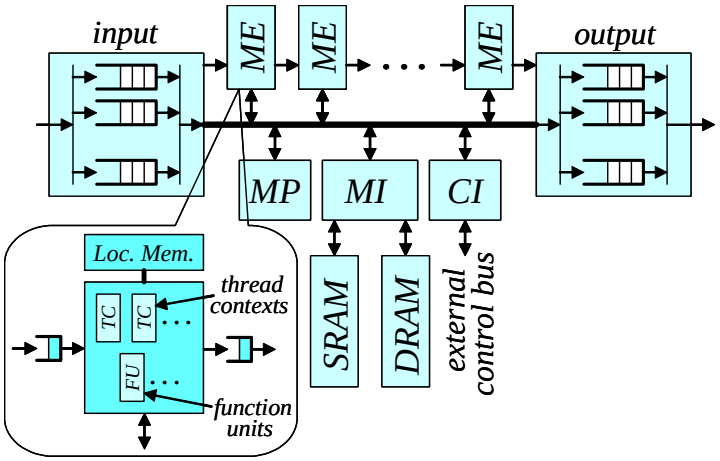

Figure 1. Generic Network Processor

control processor virtualization, while Section 5 discusses the topic of data-plane processor virtualization. The paper concludes with a concise summary of our model of virtualization in Section 6 and acknowledgements in Section 7.

2. BACKGROUND ON NETWORK PROCESSORS

Figure 1 shows the logical organization of a typical NP. Packets arrive at the input block at the left, where they are stored in one of a number of separate queues. They leave from the output block at right. The *Micro Engines* (ME) perform the bulk of the data processing. This multi-core architecture allows the NP to efficiently exploit packet-level parallelism [4,5]. Packets may be passed through the MEs in a pipelined fashion, with each ME performing a portion of the processing, or they may be distributed to different PEs over the central bus, with each packet being completely processed by a single ME. The central bus also provides access to external SRAM and DRAM through the *Memory Interface* (MI) and to an external control bus via the *Control Interface* (CI). Each ME typically has a local memory used to store program instructions and several distinct hardware *Thread Contexts* (typically 4-8), each of which includes a register file, program counter and other essential per-thread information. MEs are typically able to switch from one thread context to another in a single clock cycle, allowing the processor to continue processing, even when some threads are blocked waiting for a memory access to complete. Each ME may also have multiple functional units, allowing it to exploit instruction-level parallelism within a single thread. The device is managed by a central *Management Processor* (MP).

To see how this generic organization corresponds to a real commercial NP, we now consider the Intel IXP.

2.1 Intel IXP Network Processor Organization

The IXP family of NPs consists of networking-specific chip-multiprocessors (CMPs) with processor core counts ranging from 4 to 17. The IXP2850 is the high-end offering, targeting a line rate of 10 Gbps; its organization is shown in Figure 2. To provide this level of performance in software, the IXP2850 runs at a 1.4 GHz clock frequency and features one XScale control processor (which corresponds to the management processor in Figure 1), a high-performance ARM-compatible processor which hosts embedded Linux or a real-time OS and implements management and control functions, and 16 MEs, which supply the data plane packet processing and do not host an OS.

The MEs are six-stage, RISC-like processors with an instruction set architecture enhanced for network processing. The MEs are multithreaded, with hardware support for 8 thread contexts, providing single-cycle context switches via a non-preemptive hardware thread scheduler;

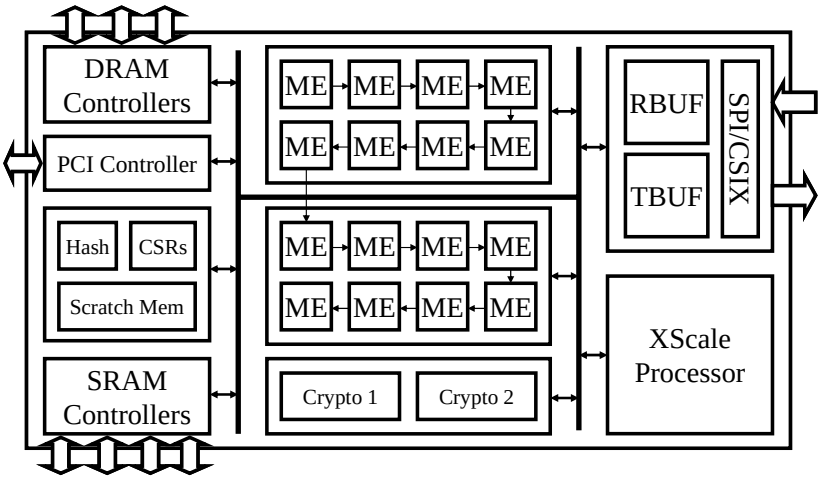

Figure 2. IXP2850 Organization.

accordingly, the ME features 256 general-purpose registers and an addressing mode that allows one thread to access them all. The MEs each have a number of hardware assists that are accessible to programs as instructions, including a 16 entry content addressable memory (CAM), and a hash unit. The arrows in Figure 2, connecting neighboring MEs, represent next-neighbor (NN) registers, which can be used to implement high-bandwidth, low-latency communication between neighbors (the registers are typically managed as a ring). Like many other embedded systems, the MEs have no hardware support for floating-point operations. In place of instruction and data caches, MEs each have an 8K-entry control store, which is organized as an SRAM, and a local data memory 2KB in size. The ME architecture is illustrated in Figure 3.

Additionally, the IXP2850 supports several external SRAM and DRAM channels to meet the high memory bandwidth requirements of high-performance networking workloads. The SRAM controller supports advanced operations, such as queue/ring semantics and atomic operations. The presence of the crypto engines, which are capable of line rate 3DES, AES, and SHA-1 (de)encryption of packet data, distinguishes the IXP2850 from the IXP2800; otherwise, the two are identical.

The collection of blocks in the IXP diagrams labeled SPI/CSIX, RBUF, and TBUF are part of the *media switch fabric (MSF).* The MSF is the unit that that provides network I/O. SPI is a chip-to-chip protocol that, for example, connects the IXP to one or more Gigabit Ethernet media access controllers (MACs). CSIX is a standard protocol for connecting chips to switch fabrics. The MSF provides packet I/O via receive and transmit buffers (RBUF, TBUF); packets are staged through these buffers under the control of software running on the MEs.

The collection of blocks labeled Scratch, Hash, and CSRs are part of the SHaC unit which provides 16 KB of on-chip scratch SRAM and controller supporting message queues and rings as well as atomic operations, a configurable Hash unit, and control-status registers (CSRs) for controlling and querying IXP operation.

The PCI interface allows the IXP to attach to control plane devices, such as disks and Ethernet ports. All units on the chip are connected by a collection of high-speed, unidirectional buses and arbitration units.

Applications developed for the IXP are typically structured as packet processing pipelines, thus the IXP has many features to support pipelining across MEs, including: next-neighbor registers between MEs, inter-thread and programmer-controlled inter-core signals in hardware, and on-chip and SRAM-based message rings and queues.

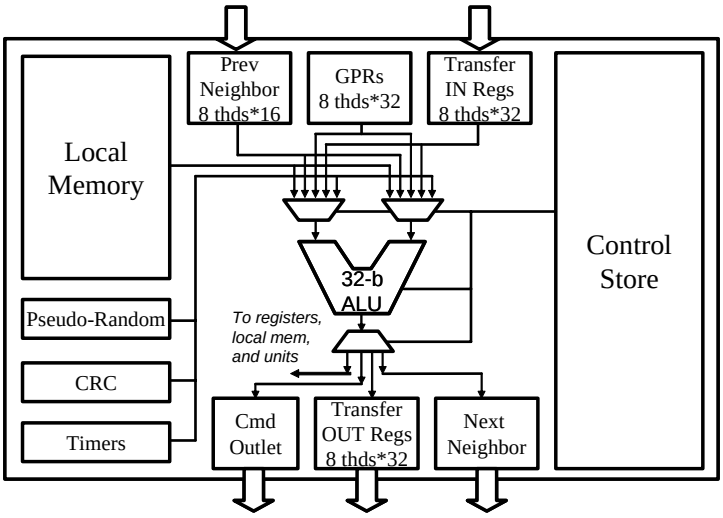

Figure 3. IXP Micro-engine (ME) Architecture.

The XScale is programmed like any other Linux-based system. The MEs can be programmed in either their assembly language, or with a variation of the C programming language called Micro-C. Micro-C is neither a superset nor a subset of ANSI C. It allows C-language syntax for data and control structures, but requires special annotations to variable declarations, for example, to accommodate the heterogeneous memories provided by the IXP. Other C-language restrictions include: no support for floating-point data; and no support for recursive functions since the MEs do not have fast access to a stack. Many IXP features must be accessed with assembly macros, since not all features can be expressed in Micro-C semantics.

The IXP2850 is just one instance of a family of IXP NPs. As an additional example, the IXP 2400 is architecturally similar to the 2850, but targets 4 Gbps line rates; it features a 700 MHz clock rate, 8 MEs, fewer memory interfaces, and no crypto engines. As we will see, our experimental results use the IXP 2400 since that was the platform we had access to at the time these experiments were conducted.

2.2 Intel IXA Software Architecture

Networking applications consist of three levels of functionality: management, control, and data. The *management plane* exports user interfaces for: system configuration, service management, organization and user-level policies, statistics, and system startup and shutdown. The *control plane* implements the protocols and applications that govern the operation of applications and network services; resource provisioning protocols and reservation and management of shared data such as route tables are examples of control plane functions. Finally, the *data plane* is responsible for normal packet and flow processing and forwarding at line rates.

The *Intel Internet Exchange Architecture (IXA)* software architecture [6] reflects this structure, as does the IXP hardware organization itself. Management plane operations are typically implemented on a host computer attached to the router. Control plane operations are typically implemented on the XScale processor. Data plane functions are primarily implemented on MEs, except for rare and error conditions which are usually implemented on the XScale. This overall structure is illustrated in Figure 4. As can be seen, a generic receive, process, transmit application is shown.

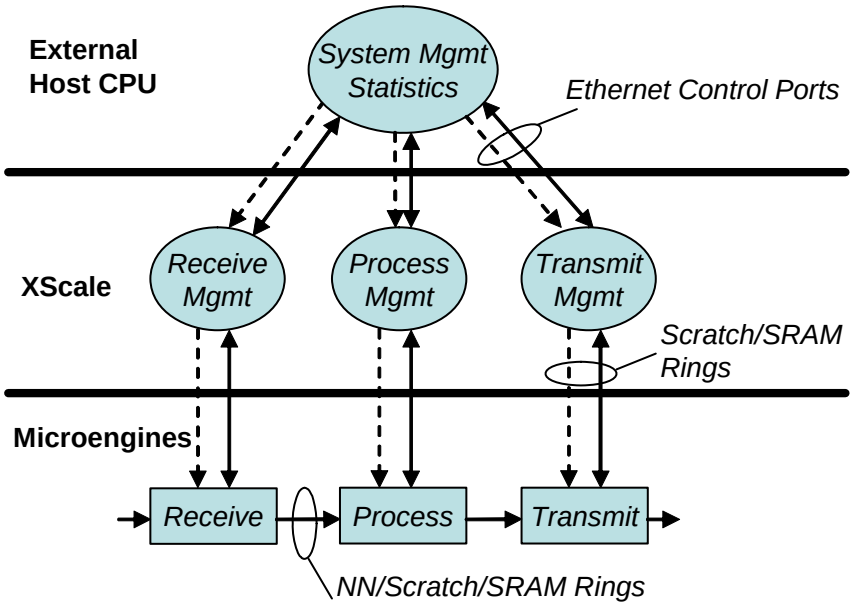

Figure 4. Generic Structure of Software Architecture.

3. REQUIREMENTS FOR VIRTUALIZATION

Since our aim is to consider NP virtualization within the specific context of network virtualization, we have the option of either considering NP virtualization in a completely generic way, or, alternatively, in the context of supporting virtual routers.

A generic approach to NP virtualization would consider all resources to be available for shared access and would call for mechanisms to provide isolation and provisioning in each instance. A more specific view of virtualization could define a subset of resources to be shared, with the rest considered to belong to the system and unavailable for direct end-user access. This specific view would restrict the placement of third-party, allowing it only to be added to well-defined portions of the infrastructure.

Consider, for example, the system illustrated in Figure 5. This software architecture is based on software plugins [7], which allow the programmer to add application-specific packet filters and associate them custom processing blocks. In this instance, rather than virtualizing the entire NP, a subset of the system is made available via the plugin API. Such an API could conceal many details and those corresponding subsystems of the NP would not therefore need to be virtualized. This example is attractive since the plugin model has proved to be simple and effective in both software and hardware [8]; thus, the extension to network processor software is a natural one. On the other hand, such an API would limit the flexibility given to developers. To see how such an API might limit the portion of the NP touched by authors of virtual router code, consider the following operational description.

In Figure 5, the input processing block: performs link-level (e.g., Ethernet) decapsulation and examines incoming packet headers; forwards packet payloads to the packet storage manager, which validates the packets, and stores payloads in DRAM; and passes a buffer descriptor containing much of the known header, which includes an ID for the virtual network that is carrying the packet, plus a pointer to the payload, on to the packet classification (PC) block. The PC block classifies the packet, using a key constructed in a manner specific to the virtual network, and passes along the resulting queue ID and descriptor to the queue manager (QM) block. The output processing block receives buffer descriptors for outgoing packets from the QM block, and performs validation, link-level encapsulation, and packet transmission. For example,

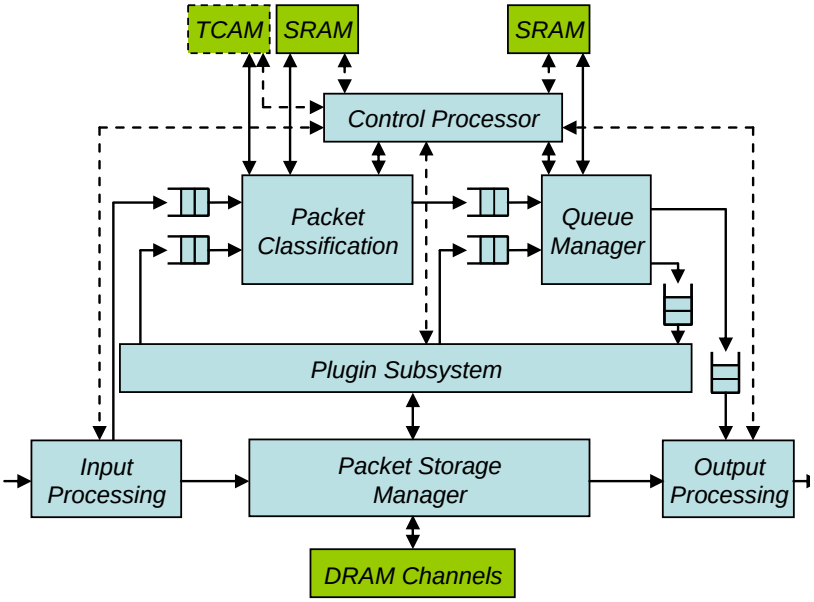

Figure 5. Sample Virtual Router Infrastructure.

an IP-based virtual network could be implemented by a third party using a series of plugins that provided: IP header extraction (and initial classifer key construction routine), IP packet verification, IP lookup, and IP header and checksum construction. Note that in this example, each packet would take multiple round trips through the packet classifier, plugin, and, possibly, queue manager subsystems. All blocks shown are implemented on micro-engines, with the exception of the control processor which refers to the XScale processor and its Linux operating system. In this example, the third-party developer would not need to have access to the MSF or buffer allocation subsystems since those are handled by the infrastructure.

Whether we consider generic or specific virtualization, we must consider virtualizing access to resources on both the control and data plane processors. In each instance, we must consider virtualizing: thread execution, access to memory, memory bandwidth, and access to special-purpose hardware assists. In the case of the IXP, special-purpose hardware assists such as the cryptography units are accessed in the same way that memory is accessed. Thus, we need not consider it in its own category. There are examples of hardware resources that cannot effectively shared through memory mapping techniques, such at the IXP's MSF, so in those cases we will assume that these specialized resources are part of the system infrastructure and not made available to virtual routers for direct manipulation.

As we will see, control processor virtualization is relatively straightforward due to the presence of a modern operation system. Virtualizing MEs, however, is less clear and will require new software or hardware mechanisms.

Note that the granularity of virtualization can have a strong influence on the difficulty of this problem. If a virtual router is allocated, at a minimum, one dedicated IXP, then we need not consider IXP virtualization at all. If the minimum allocation is a dedicated ME, rather, then inter-ME isolation and provisioning must be provided. As we will see, one straightforward way to accomplish this is simply to partition the MEs among the different virtual routers and provide protection registers to regulate which part of the physical memory is available to each ME. This approach makes it easy to provide a static memory partitioning, but will need to be extended to enable the memory space available to different MEs to be adjusted while the system is operating. In conventional processors, paged virtual memory systems allow programs to operate securely within their own virtual memory system. However, paged virtual memory systems have too much

overhead to enable the fine-grained context switching that must take place in a virtual router. An alternative is to use some form of segmented virtual memory instead.

If individual ME contexts are the unit of allocation, then we must consider virtualizing the internal ME resources as well. Since our intended usage does not require multiple virtual routers to share a single ME, however, we do not explore this degree of virtualization.

In the following two sections, we discuss control and data-plane processor virtualization in turn.

4. CONTROL PROCESSOR VIRTUALIZATION

Virtualizing the control processor involves dividing up resources, such a memory, I/O devices, and CPU cycles among different domains. This must be accomplished in such a way as to provide isolation and resource accounting, scalability, and fairness. Three ways to accomplish this are system virtualization via virtual machines (VMs), and specific OS image virtualization such as Linux vservers [9].

One approach to virtualizing the control processor is to provide each virtual router with its own virtual machine and operating system, which allows it to view itself as the only occupant of that CPU. Two related ways of accomplishing this are virtual machines and virtual machine monitors. VMs, such as VMware [10], provide a fully emulated virtual machine on which an operating system and its associated applications to run unmodified, but often at the expense of performance. VMMs, such as Xen [11], improve performance by using paravirtualization, where the underlying processor is only partially virtualized. This requires porting the operating system to run on the VMM, but no modifications to user applications are necessary. While both VMWare and Xen are currently only available for the x86 architecture, the creators of Xen claim that x86 represents the worst case for virtualization; it should be relatively easy to achieve similar results on another architecture.

Xen incurs very little overhead; on most workloads, it is able to achieve over 90% of the performance of a non virtualized Linux host. It is designed to be scalable to over 100 domains running simultaneously, and provides strong resource and performance isolation among them. Even malicious programs such as fork bombs do not greatly affect the performance of other domains. Domains can be created and destroyed dynamically, and resource accounting for each domain is provided.

Linux vservers provide an alternative to system virtualization. Vservers are a modification to the Linux kernel that allows user space processes to be given their own root file system and allocated their own resources, but with lower overhead, since each instance does not need its own kernel or virtualized machine. They can be used to provide protection for everything from an individual process to an entire distribution and are attractive because they are more scalable and perform better. Fewer resources are used since each domain does not need its own kernel and emulated machine environment, and further savings can be had by applying the concept of unification- resources such as shared libraries, which are read only and are used in different domains, can be shared by those domains. Depending on the application, this may be a better approach than a VM or VMM. However, if a particular application needs to run a modified kernel, or different domains need different versions or configurations of the kernel, the vserver approach would not work.

There are many options for control processor virtualization, depending on the exact requirements. Linux vservers are lightweight, easier to implement, and more portable, but VMMs such as Xen provide more functionality, and can still be quite scalable and fast. All these methods provide the isolation, resource accounting, and scalability needed in any virtualization scheme. However, neither of these solutions have, to our knowledge, been ported to the IXP's

XScale processor. While this may involve a more difficult that usual port of the software, there is no clear technical reason why either scheme could not be made to work in this context.

5. DATA PROCESSOR VIRTUALIZATION

As described in the following sections, virtualizing data-plane processors involves three categories of virtualization. The first concerns scheduling threads of execution to ensure that the real-time requirements of individual virtual routers are satisfied. The second issue considers how the memory architecture can be managed to provided memory access isolation between virtual routers. The third concerns provisioning memory bandwidth among virtual routers. Section 5 concludes with a discussion of the performance overheads associated with our NP virtualization approach.

5.1 Virtualizing Threads

There are two natural methods of using the parallel processing resources in a modern NP. One is to organize the processing as a pipeline, with each processor handling one stage in the pipeline. While pipelining has some important advantages, it can be difficult to break down programs into a large number of computationally balanced steps. The second common way to use the parallel processing resources is to distribute packets to processors, letting one processor fully handle each packet. This approach is inherently more flexible and makes programming far more straightforward. It does raise other issues, some of which may require architectural extensions to NPs in order to realize the full benefits.

One of the key issues is how to schedule the use of processing resources to ensure fair treatment of different flows or to satisfy real-time requirements. While there are some similarities between scheduling the use of link bandwidth and scheduling the use of processing capacity, there are also some significant differences. First, when scheduling the processors in an NP, there are multiple processors to choose from and certain packets may be constrained in their choice of processor (they must run on a processor that has the required code available and if there is per flow state information that is retained between packets within a processor's local memory, all packets of the flow must run on the same processor). Second, packet execution times are highly variable and much less predictable than packet transmission times on a link. This requires adjustments to scheduling policies that depend on knowing in advance, how much time is needed to process a given packet [12].

An alternative to scheduling packets is to associate application data flows with processing threads and schedule the threads, rather than scheduling packets. There are two big advantages to this approach. First, some packets take much longer to process than others (a 4 KB packet requiring 500 instructions per byte of data will take 2 ms to process, assuming instructions can be processed at the rate of one per ns). We may prefer to interrupt processing of such long packets to avoid imposing excessive delays on packets that require only a little processing. Second, some applications may not follow the strict packet-in, packet-out paradigm. They may generate multiple output packets for each input packet, or may produce fewer packets than they receive. It can be difficult to handle such applications in the context of a purely packet-driven scheduling mechanism.

Thread (or process) scheduling has been used extensively in conventional time-shared computing systems. There are some important differences that arise when we consider thread scheduling in the NP context. First, to achieve the requisite real-time performance, the time-slice duration must be much shorter in an NP than in a conventional time-sharing system (say 100 μs, instead of 10 ms) and the overhead for switching contexts must be a small fraction of the time-slice duration. Second, thread scheduling must take into account the quality of service requirements of the associated packet flows. While both of these issues arise in real-time systems

contexts, the requirements are more stringent for NPs than in most real-time systems (typical real-time systems process hundreds or thousands of events per second, while NPs serving 10 Gb/s links may process hundreds of millions of packets per second). It's important to recognize that the availability of multiple hardware thread contexts does not address the larger thread scheduling issue. Hardware thread contexts are only meant to reduce the impact of memory latency on processor performance. They do not address the real-time aspects of thread scheduling. Also, if individual flows are assigned separate threads, the total number of threads handled by a processor can be much larger than the number of hardware contexts.

To apply time-slicing to processors with multiple hardware thread contexts, we must generalize the notion of time slice to reflect the resources used by the thread rather than simply elapsed time (this could be measured simply in terms of the number of instructions executed by the thread, or using a more complex metric that reflects the varying costs associated with different instructions). A thread selected for execution would be allowed some fixed amount of computation (a *processing slice*) before being suspended and replaced with another thread. Of course, a thread that becomes blocked waiting for input would be suspended and replaced with another thread. When a thread context becomes free, we can select from among the ready-to-run threads using the same type of virtual-time methods used for scheduling packets. Indeed, because processing is no longer associated with specific packets, the scheduling decision can be made somewhat simpler.

For maximum flexibility, we would like to avoid having to implement the thread scheduler in hardware. At the same time, we need to keep the scheduling overhead very small. The key to minimizing the overhead of a software scheduler is to speed up the saving and restoring of the thread context. Block copies of the thread context to and from external SRAM are essential for this purpose. One elegant way to structure the scheduling operation is to equip each thread context with two protected registers, a *thread image* register and a *next thread* register. Whenever a thread is suspended, it is copied to memory at the location specified by the thread image register (typically, thread images will be stored in external SRAM) and then the next thread register is used to retrieve a new hardware thread context. Once the new context is loaded, the new thread begins executing. The system software can use this mechanism to start the thread scheduler whenever a normal processing thread completes. Once the scheduler decides which thread should go next, it writes this value in its next thread register and suspends itself, triggering the switch to the new thread. Using fast Quad Data Rate SRAM, thread contexts supporting 16 general purpose registers and other required per thread state can be transferred between SRAM and memory in under 50 ns, leading to an overhead of about 200 ns to switch from one processing thread to another (including the switches to and from the system thread). If the scheduler data structures are located in SRAM and the scheduler makes no more than a few tens of independent memory accesses, it should be possible to complete the whole thread switch in less than a microsecond. This is small enough to allow processing slices as short as the equivalent to about 10 μs of uninterrupted computation. Longer slices (equivalent to say 100 μs of computation) will reduce the overhead further and are probably still short enough to satisfy most real-time requirements. Flows consisting of widely spaced packets requiring only a small amount of computation can incur significant overhead due to flow switching, since these flows will process a single packet at a time and may require less that 10 μs of computation per packet. For such flows, it may not be practical to associate each individual flow with a separate thread. By structuring the program so that a single thread handles multiple flows, we can reduce the effective overhead of thread switching, at the cost of less effective traffic isolation among flows.

5.2 Virtualizing the Memory Address Space

We first consider the memory architecture of a typical NP. NPs such as the Intel IXP have a fixed instruction memory, a small amount of local scratch memory, plus off-chip SRAM and

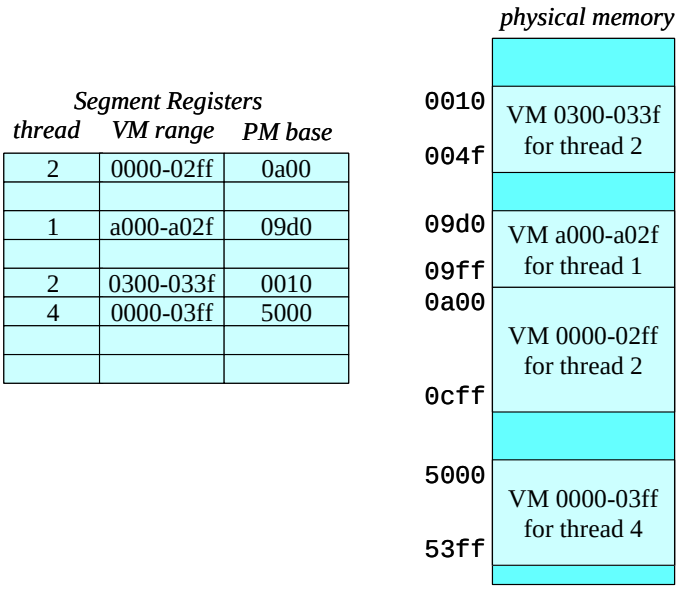

Figure 6. Segmented VM with pooled segment registers

DRAM. Rather than attempting to reduce memory access latency, each processor has multiple hardware thread contexts to tolerate the potentially long latency of memory access. Typically, none of the memory is organized as a hardware cache. It's up to the programmer to manage the different types of memory (registers, on-chip SRAM, off-chip SRAM, DRAM). This gives the programmer great flexibility, but at the cost of a more complex memory model which makes programming much more difficult. One consequence of a fixed control store is a maximum program size – typically, each ME has access to a store of 2-8K instructions. This fact makes an instruction cache all the more appealing, since small programs would execute entirely from cache, but larger programs could still be executed. One complicating factor is the interaction between the cache and multiple thread contexts. Unless the cache associativity matches the number of active contexts, the overall performance can be poor. Caches can also represent a hidden cost, when switching among different flows, since flows can experience poor performance until the cache "warms up."

5.2.1 Hardware Enhancements for Address Space Virtualization

Another important issue regarding the memory architecture is protection. An NP hosting multiple virtual routers will execute a variety of different programs at different times. While network operators are expected to strictly control which programs can be executed, there is still a need for memory protection to prevent faulty programs from corrupting data belonging to other programs. In conventional processors, memory protection is provided through the virtual memory (VM) subsystem. NPs need a lighter-weight alternative than the conventional paged VM system. The real-time nature of NPs rules out swapping threads to and from disk, so conceivably one could use paged VM, but eliminate swapping. Unfortunately, the large page tables that define the memory maps in paged VM systems are too large to fit in a processor's local on-chip memory. The standard solution to this problem is to cache memory mapping information in a *Translation Look-aside Buffer* (TLB), but it's not clear how effective a TLB can be in processors supporting rapid switching among multiple hardware thread contexts.

An alternative is to provide a segmented virtual memory is to equip each ME with a set of segment registers, each consisting of a pair of virtual memory addresses (defining a range of memory locations) and a physical memory address defining the first location in physical memory corresponding to the first virtual memory address in the range. When a program executing on the

```
scratch_op[ args ]
sram_op[ args ]
dram_op[ args ]
  with args := op, xfer, src_op1, src_op2, size_arg, indirect_st,
               indirect_arg, sig
```

- *op*: read or write
- *xfer*: the transfer register(s) to store to/read from
- *src_op1*, *src_op2*: These 2 are added together to form the starting address of the operation.
- *size_arg*: either a number (1-8) or if doing an indirect ref, max_1 to max_16
- *indirect_st*: the statement to execute just before the memory operation if doing an indirect ref.
- *indirect_arg*: 0 if doing an indirect_ref, 1 otherwise
- *sig*: sig_done[signal name] or ctx_arb[signal name]

Figure 7. Macro interfaces for memory API.

processor attempts to access a memory location, the virtual address of the memory location is compared to the ranges of all the processor's segment registers and the associated physical address is then computed using the segment register with the matching range. If there is no matching range (or multiple matching ranges), a memory access error occurs. The flexibility of this approach is limited only by the number of segment registers. Assuming that the allocation of memory to processors is not extremely dynamic, a fairly modest number of registers (say 16) can provide ample flexibility at a low implementation cost. Alternatively, each hardware thread context in the processor can be equipped with its own set of segment registers, allowing each to operate within a separate virtual memory space. Figure 6 shows a more scalable version of this approach, in which the processor provides a pool of segment registers that can be dynamically assigned to thread contexts. This allows some thread contexts to use more segment registers than others to accommodate more dynamic virtual memory usage patterns. Given enough segment registers per thread, different segments can be used for the stack, other private data, shared data, and code, allowing different threads to easily share the same code in a read-only segment.

Segmented VM is rarely used in conventional computer systems, in large part because it can lead to fragmentation of the memory space. This is particularly an issue in systems that swap segments to and from disk. In the NP context, these concerns are reduced, since there is no swapping and since NP programs can be more reasonably constrained in their memory usage (static memory and bounded stack size) than can programs in a general-purpose time-shared computer system.

5.2.2 Software-based Memory Protection

We implemented a technique for memory protection in software on the IXP that is similar to a software enforced segment register. Each ME has a set of 6 registers- two each for DRAM, SRAM and the Scratchpad, representing the upper and lower bounds of the memory segments allocated to that ME. Wherever a raw memory access is made, it is replaced with a macro that dynamically checks to make sure that the request stays within the allocated segments. If it does not, an interrupt is raised and the ME is paused. In a fully implemented system, a process on the Xscale will then handle the error. As we will see in Section 5.4, these macros involve very little overhead. The macro API is shown in Figure 7.

```
.while(i < 1000)
  sram_op[read, $xfers[0], 0, sram_begin, 8, X, FALSE, sig_done[sig1]]
  sram_op[read, $xfers[8], 30, sram_begin, 8, X , FALSE, sig_done[sig2]]
  sram_op[read, $xfers[16], 40, sram_begin, 8, X , FALSE, sig_done[sig3]]
  sram_op[read, $xfers[24], 50, sram_begin, 8, X , FALSE, sig_done[sig4]]
  sram_op[read, $xfers[32], 60, sram_begin, 8, X , FALSE, sig_done[sig5]]
  ctx_arb[sig1, sig2, sig3, sig4, sig5]
  alu[i,i,+,1]
.endw
```

Figure 8. Test code.

5.3 Virtualizing Memory Bandwidth

In addition to providing memory protection, a virtualized NP must provide bandwidth guarantees for each separate virtual router. A good virtualization scheme will guarantee a certain amount of bandwidth to each ME, provide for fair allocation under high loads and properly account for the amount of resources used by each ME. Ideally, excess capacity could be given to MEs that can utilize it.

Resource allocation on the MEs is not as easy to achieve as it is on the control processor, since there is no operating system or other centralized entity to arbitrate among them. The most direct way to implement bandwidth virtualization would be with enhancements to the memory controller. Such a smart controller could be configured to allocate different amounts of bandwidth to different MEs, and could then schedule memory requests from each ME accordingly.

We implemented a technique for bandwidth rationing in software on the IXP that uses macros to replace memory access instructions. Each ME is allocated a set of tokens sized in proportion to the percentage of allocated bandwidth; at any given time, the number of remaining tokens is kept in a well-known register. When the ME makes a memory access, the macro checks to see if there are enough tokens left to satisfy that request before issuing the memory operation. If there are, that number of tokens is subtracted from the total and the memory instruction is executed. If not, it must wait for the number of tokens to be replenished. Tokens are replenished when the MEs receive a signal from a process running on the Xscale whose sole task is to periodically set this replenishing signal for all the MEs. The macro API is the same as the bounds-checking macros, and is shown in Figure 7.

In the context of virtualizing an ME to support virtual routers, it is assumed that the system administrator will have tools to parse all the programs before loading them into the hardware to ensure that these virtualization macros are being used (i.e. there are no bare accesses to memory) and that global registers needed to support them are not accessed by the user programs.

5.4 Implementation & Evaluation

As described above, we have explored software methods for implementing some of the virtualization requirements on a current generation Intel IXP network processor. We report results for the IXP 2400, since at the time these experiments were run our laboratory only had access to those hardware platforms. Specifically, we have implemented software schemes for memory protection and bandwidth rationing on a per microengine basis. The implementations are described above in the relevant sections; this section describes their effectiveness and impact on performance. Our example results illustrate SRAM access, but comparable results can be obtained for the other memory types.

Our virtualization mechanisms seem to work well. The memory protection scheme adds 8 instructions per memory access. The bandwidth rationing scheme adds 6 instructions per access. Together, they represent 14 more instructions per access. While this might appear to be a

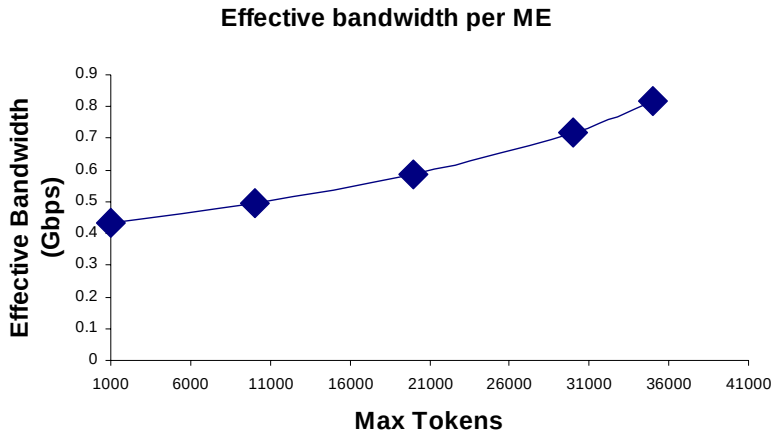

Figure 9 Average effective bandwidth per ME (8 total) as a function of token count.

Table 1. Average effective bandwidth in bytes/sec with effectively infinite tokens

	No Macros	Bounds checking	Bandwidth limiting	Both
Effective Bandwidth (Gbps)	0.819	0.819	0.817	0.817

significant multiplier, we observe that two factors mitigate the effect. First, the real cost of the additional instructions must be weighed against the 100-300 cycles spent waiting for the operation to complete. Second, even programs that spend most of their time executing memory operations are not necessarily dominated by these additional instructions. In fact, when running a memory bound test program, shown in Figure 8, they combined for 0.07% increase in instruction execution cycles.

Overall, there is no significant performance reduction when using the macros. When executing the program in Figure 8 in a single ME, with both bounds checking and rate limiting enabled, the ME is able to consume nearly the same amount of peak bandwidth, assuming that enough credits are provided. As required, the memory protection scheme successfully pauses an ME whenever it attempts to access memory outside its allocated segment.

To demonstrate that bandwidth can be accurately provisioned, we ran the test program on eight MEs on an IXP 2400. The token refresh rate used was 3 us and the maximum number of tokens was varied. Figure 9 shows the effect on effective bandwidth per microengine as a function of the maximum number of tokens. The fact that the plot is fairly linear over a long range implies that we should be able to obtain very fine grained control on the amount of bandwidth allocated to each ME. We note that at around 35K tokens, the aggregate SRAM bandwidth consumed approximately equals the total available bandwidth. The average effective bandwidth achieved assuming infinite tokens is shown in Table 1. Further tests show that changing the amount of tokens given to each ME results in a different effective bandwidth for each.

6. VIRTUALIZATION SUMMARY & CONCLUSION

In this paper, we have considered the problem of virtualizing network processor resources. In the context of providing support for network virtualization in Intel IXP-based routers, we have presented a model for virtualization that can be summarized with the following points.

- Virtual routers are allocated an execution context on the XScale processor via either Xen or Linux vservers;
- Virtual routers are allocated one or more MEs;
- Each ME is allocated a percentage of the *address space* and *bandwidth* at each external resource (e.g., SRAM, DRAM, Scratch);
- Third-party programmers use macros that do bounds checking and rate limiting;
- The router administrator scans ME binary/assembly/source for compliance, verifying that there are no illicit "raw" accesses to external resources.

The use of Xen or Linux vservers for XScale virtualization is straightforward since it is a traditional example of processor virtualization. While porting either system to the IXP may pose a challenge, the strategy is clear.

In order to ensure that software running on a given ME only accesses its allocated memory region, our memory access macros perform dynamic bounds checking on each memory reference. The bounds are established when the program is loaded and are kept in reserved system registers.

To enforce bandwidth provisioning, these macros also perform rate limiting. Our approach uses a credit-based scheme to limit each ME to a specified number of memory operations within a window of time. Credits are decremented with each operation and refereshed periodically by a system process running on the control processor. In our implementation, these credits are maintained in reserved system registers. Dynamically at each reference, the macro checks for refresh a signal and to see if enough credits remain to carry out the requested memory operation (the current version blocks the code until enough credits are available).

Safe use of these macros requires an administrative policy that A) forbids memory accesses that do not use the provided macros and B) requires that programs leave reserved system registers unmodified. Such a policy can be enforced by scanning all provided software for compliance.

Our initial experimental evaluation on the IXP 2400 network processor suggests that this approach is feasible and demonstrates the two key aspects of our system. First, bounds checking and rate limiting can both be performed with negligible overheads. Second, our rate limiting scheme can effectively provision memory bandwidth: the scheme can be used to support uneven allocations (nearly all bandwidth allocated to one ME) as well as balanced allocations (in which all available bandwidth is shared dynamically).

7. ACKNOWLEDGMENTS

This work was supported by NSF grant CNS-0435173 and by a gift from Intel Corporation.

REFERENCES

[1] Anderson, Tom, Larry Peterson Scott Shenker and Jonathan Turner. "Overcoming the Internet Impasse through Virtualization," *IEEE Computer Magazine*, April 2005.

[2] Peterson, Larry, Tom Anderson, David Culler and Timothy Roscoe. "A Blueprint for Introducing Disruptive Technology into the Internet," *Proceedings of ACM HotNets-I Workshop*, 10/02.

[3] Turner, Jonathan, Patrick Crowley and John Lockwood. "MRI:Development of a Diversified Router for Experimental Research in Networking," NSF Grant CNS- 0520778.

[4] Crowley, Patrick, Marc Fiuczynski, Jean-Loup Baer, and Brian Bershad. "Workloads for Programmable Network Interfaces," *IEEE 2nd Annual Workshop on Workload Characterization*, 10/99.

[5] Crowley, Patrick, Marc Fiuczynski, Jean-Loup Baer, and Brian Bershad. "Characterizing Processor Architectures for Programmable Network Interfaces," *Proceedings of the 2000 International Conference on Supercomputing*, Santa Fe, N.M., 5/00.

[6] Carlson, Bill. *Intel Internet Exchange Architecture and Applications: A Practical Guide to Intel's Network Processors*, Intel Press, 2003.

[7] Decasper, D., Dittia, Z., Parulkar, G., and Plattner, B. 1998. Router plugins: a software architecture for next generation routers. In *Proceedings of the ACM SIGCOMM '98 Conference on Applications, Technologies, Architectures, and Protocols For Computer Communication,* Vancouver, British Columbia, Canada, August 31 - September 04, 1998.

[8] Taylor, D. E., Turner, J. S., Lockwood, J. W., and Horta, E. L. Dynamic hardware plugins: exploiting reconfigurable hardware for high-performance programmable routers. *Comput. Networks* 38, 3 (Feb. 2002), 295-310.

[9] LINUX VSERVERS PROJECT. http://linux-vserver.org/.

[10] VMWare. http://www.vmware.com/.

[11] Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., and Warfield, A. Xen and the Art of Virtualization. In *Proc. 19th SOSP* (Lake George, NY, Oct 2003).

[12] Pappu, Prashanth and Tilman Wolf. "Scheduling Processing Resources in Programmable Routers," *Proceedings of IEEE Infocom*, 6/02.