

Implementation and Evaluation of A QoS-Capable Cluster-Based IP Router

Prashant Pradhan Tzi-cker Chiueh
Computer Science Department
State University of New York at Stony Brook
{prashant, chiueh}@cs.sunysb.edu

Abstract

A major challenge in Internet edge router design is to support both high packet forwarding performance and versatile and efficient packet processing capabilities. The thesis of this research project is that a cluster of PCs connected by a high-speed system area network provides an effective hardware platform for building routers to be used at the edges of the Internet. This paper describes a scalable and extensible edge router architecture called *Panama*, which supports a novel aggregate route caching scheme, a real-time link scheduling algorithm whose performance overhead is independent of the number of real-time flows, a highly efficient kernel extension mechanism to safely load networking software extensions dynamically, and an integrated resource scheduler which ensures that real-time flows with additional packet processing requirements still meet their end-to-end performance requirements. This paper describes the implementation and evaluation of the first *Panama* prototype based on a cluster of PCs and Myrinet.

1 Introduction

A key design principle for next-generation Internet architecture is to push packet processing complexity to network edges so that backbone routers can be both simple and fast. However, this architectural decision also places more demanding requirements on edge router designs in that they have to not only forward generic IP packets at a rate scalable with the link speed, but also support a wide array of packet processing functions for value-added services. Examples of such services range from low-level router enhancements that improve end-to-end performance such as active queue management and congestion state sharing, to application-specific payload processing functions such as content-aware forwarding and media transcoding. As a result of these new service requirements, edge routers become compute-intensive as well as I/O-intensive systems. In view of this new architectural requirement, the following principles guide the design of next-generation Internet edge routers:

1. To support a wide variety of packet processing functionalities, an edge router architecture should base its high-level packet processing on a general-purpose computing platform and support high-level programming

abstractions to facilitate the development of router extensions.

2. Router extension functions should be added to an edge router statically or dynamically without compromising the router core's integrity. In addition, these functions should be embodied as computation interposed along the packet datapath in an *efficient* and *safe* manner.
3. The performance of an edge router architecture should scale up linearly with increasing processing hardware, switching capacity, and memory size. Moreover, it should be possible to grow the router incrementally over a wide range of system sizes.

This paper describes the design, implementation, and evaluation of a scalable and extensible edge router architecture, called *Panama*, whose goal is to show that with efficient algorithms and system software, PC clustering hardware provides an effective platform for building highly scalable and extensible edge routers. To achieve this goal, *Panama* decouples packet forwarding from packet computation whenever possible, and provides an *asynchronous* linkage between them. The basic hardware building block of the *Panama* architecture is a *router node*, which includes a set of programmable network interfaces and a general-purpose processor. Router nodes are interconnected through a high-speed system-area switch-based interconnect that serves as the router's system backplane. The scalability of the *Panama* architecture comes from the incremental expandability of both the number of router nodes and the total switching capacity of the router backplane. The current *Panama* prototype uses general-purpose PCs with programmable network interfaces as router nodes, and a Gbit/sec switch as the router backplane. The only requirement on the router node's network interfaces is that they provide a low-end processor (henceforth called the *network processor*), a small amount of memory and a DMA engine over which the network processor has direct control.

In *Panama*, if an IP packet does not require additional processing, its *forwarding path* is an interrupt-free pipeline coordinated by network processors lying on the packet's forwarding path. However, when a packet needs high-level processing, it is asynchronously *posted* to the *computation path*, which in turn asynchronously returns the packet to the forwarding pipeline after processing it. High-level packet computation is modeled as a flow of control through a function graph, and is guided by a CPU scheduler that integrated

with output link scheduling to guarantee the end-to-end performance objective of network connections that require additional packet computation.

The decoupling of forwarding and computation paths has several desirable characteristics. First of all, neither component bottlenecks the other as long as their capacities are sufficient to sustain their input demands. For instance, at high packet rates, involving the CPU (via interrupt handling, for example) in the forwarding path of network packets that do not require computation may affect the performance of the forwarding and computation paths, even though in isolation, the throughputs of the forwarding and computation paths would have been adequate to support their respective demands. Moreover, because of decoupling, improvements in any one component allow the router to service higher input loads for that component, independent of the other component. Another advantage is that in typical architectures, coupling of the computation and I/O components introduces a *fixed* overhead, which leads to an unnecessary bottleneck to the overall performance. For example, in a general-purpose architecture like a PC, this overhead is the interrupt overhead, which is determined by the time taken to receive and acknowledge the interrupt. Also note that on a node with several interfaces, such a coupling precludes any parallelism in packet processing since all interrupts are to be handled by a single CPU.

The main contribution of this work is a set of architectural and implementation techniques that we developed to construct scalable and extensible edge routers based on PC clusters. Although PC clusters have been used in the context of Web proxies, firewalls, and media gateways, the architectural tradeoffs of applying PC clustering hardware to network packet forwarding and computation has never been explored before. We believe this paper reports one of the first, if not the first, experimental result along this direction. The other major contribution of this work is the integration of real-time link scheduling and packet processing scheduling in a single framework that achieves end-to-end performance guarantees when real-time packets require protocol-specific or application-specific computation.

The rest of this paper is organized as follows. Section 2 reviews related work in extensible and high performance edge router design. Section 3 presents *Panama*'s system architecture, including the routing table caching algorithm, the kernel extension mechanism, and the packet processing computation scheduling for performance isolation. In Section 4, we describe the implementation details of the current *Panama* prototype. Section 5 presents the results of a comprehensive performance evaluation study of this prototype, as well as some lessons from this performance study. Finally we conclude with a summary of the main research contributions of this work.

2 Related Work

The Extensible Router [13] project in Princeton has a design goal very similar to *Panama*. Specifically, it attempts to make routers open and general purpose computation and communication systems so that they can support a wide variety of protocol-specific and application-specific packet processing functions. The Extensible Router uses an explicit

path abstraction, introduced in [2], that models data flow from input device to output device, possibly with computation interposed along the way. Moreover, a hierarchy of such paths is proposed with different combinations of hardware, kernel and user-level subpaths. A classification hierarchy is also proposed, to support increasing degrees of complexity of the classifiers. The idea of caching classification decisions was mentioned but no concrete algorithms were described. Two proposed solutions on the issue of how extension functions are incorporated into packet paths safely and efficiently are Java bytecodes and ahead-of-time compiled code, but neither of them is quite satisfactory on both fronts. Details about the datapath implementation, in particular, the optimized hardware paths are not described. Datapath primitives like real-time link scheduling are also missing. It also seems that the interaction between the computation and data paths is through interrupts, which we have eliminated in the *Panama* architecture to achieve better packet rate performance. The hardware platform proposed in the Extensible Router project is also a cluster of PCs, interconnected by a fast switch. A similar platform, with a different system software architecture, was independently proposed in [14] at the same time.

Router extensions have been proposed through the use of kernel modules in [10]. While this mechanism is efficient in minimizing the performance overhead of invoking router extensions, it could lead to a compromise in system integrity when the extensions are buggy or malicious. Specifically, kernel modules are part of the kernel and there is no protection boundary between kernel and extension code. Our architecture uses a safe extension model, proposed by the authors of [6], that prevents misbehaving or runaway extensions from compromising the router kernel, while keeping protection domain crossing overhead to the minimum. In addition to memory protection, *Panama* also supports performance isolation through real-time packet computation scheduling.

Click [15] is a recently proposed modular software architecture for routers based upon general-purpose PCs. Two important features of this architecture are *pull processing* and *flow-based router context*. Pull processing decouples the source and the sink of a packet forwarding path by allowing the sink to decide when to pull data from the source. Pull processing can recursively propagate towards the source, if needed. Flow-based router context allows a module to perform a *closure* operation over the modules in the computation graph that are reachable from it, and extract important information about them. The focus of Click is on showing that a modular and low-overhead structure can be built for router software, and the techniques proposed therein can be utilized in the computation model of any router architecture based on a general-purpose computing platform. However, Click does not support any real-time scheduling among computation modules. In contrast, computation in *Panama* proceeds under the control of a scheduler that attempts to match the computation requirements of various flows with the performance requirements of their data paths.

Simple PC-based router designs have been experimented with in [11] and [12]. [11] gives a good overview of the limitations of PC hardware and possible suggestions for performance optimizations. [12] proposes the use of peer-to-peer

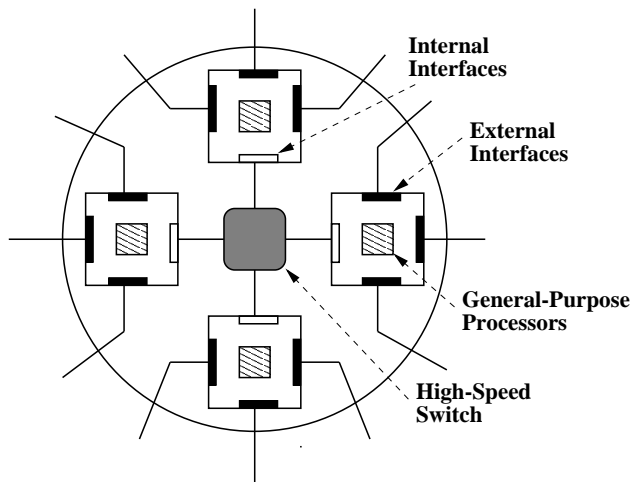


Figure 1: An example Panama router that consists of 4 nodes and 12 ports. Internal interfaces connect router nodes to a high-speed interconnect, whereas the external interfaces connect Panama to the rest of the network.

DMA to optimize throughput by avoiding multiple trips between the I/O bus and memory. This technique is also used in *Panama*.

The Scalable IP Router Project [16] proposes to use a cluster architecture to speed up complicated IP route computation such as OSPF, rather than IP packet forwarding or processing. In *Panama*, PC clustering is viewed as a means to scale the performance of both packet forwarding and computation paths, and the challenge is to develop efficient software primitives to effectively harness the raw bandwidth and processing power contained therein.

3 System Architecture

Figure 1 shows the architecture of a 4-router-node and 12-port *Panama* router. On each router node, there is a CPU and several network interfaces with one network processor per interface. On each router node, there is an interface (henceforth called the *internal* interface) that connects the node to the router's backplane. All the other interfaces on a node are *external* interfaces, and connect the router to the rest of the network. Hence, the fan-out of a *Panama* router is the total number of external interfaces on its router nodes. The typical path that a packet takes is from an input interface of an ingress node, through the internal interface of the ingress node, over the router backplane, into the internal interface of the egress node, and eventually out of the router via an output interface of the egress node. When a packet does not require any packet processing beyond forwarding, the network processors on the interfaces along the path are responsible for moving the packet toward the output link by performing necessary route lookup, packet classification, and DMA data transfer operations. If a packet needs additional processing, the network processor on the internal interface of the egress router node posts a request for service to its local CPU *asynchronously*. When this packet processing is completed, the packet is sent out on the output link according to a schedule

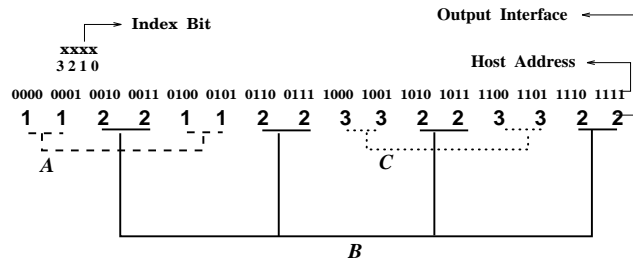


Figure 2: An example routing table whose address is 4 bits wide. Ignoring bit 1 permits merging non-contiguous address ranges and reduces the number of ranges to be distinguished from 8 to 3 (marked A, B and C).

as determined by the link scheduler.

Panama employs a *function graph* computation model to support packet processing of arbitrary complexity. In addition, the CPU scheduler carefully schedules concurrent packet computation threads so that the end-to-end performance guarantees of flows that require high-level packet computation are maintained. Packet forwarding and processing paths are running independently on physically distinct processors, but they are synchronized with each other using queues, through which each *posts* work to the other. A novel aspect of the *Panama* architecture is that it supports this synchronization in a *lock-free* and *interrupt-free* manner.

3.1 Packet Forwarding

3.1.1 Routing Table Lookup and Packet Classification

Panama implements an aggregate route caching scheme [4] at the network processors to perform route lookup for most packets, and in the case of cache misses, posts a request to the main CPU for a full-blown search through the routing table data structure. While there has been some debate about the applicability of caching on backbone routers, the results reported in [9] demonstrated that caching is very effective for edge routers. An efficient route caching algorithm, recently proposed in [4] exploits the structure in the routing table to improve individual cache sets' coverage of the IP address space, thus further improving caching performance.

An important observation for deriving efficient routing table caching algorithms is that the number of distinct outcomes of routing table lookup is small, irrespective of the size of the routing table or the IP address space. Consider the example routing table in Figure 2 for a 4-bit address space. Each routing table entry corresponds to a range of addresses in the address space. A routing table lookup algorithm's goal is to find out which address range a given packet's destination address lies in. The key novelty of the IP address range caching algorithm [4] is to merge non-contiguous address ranges into a single cacheable unit. In Figure 2, a naive merging method, in which only adjacent address ranges are allowed to merge, would yield a set of 8 address ranges whose maximum size is 2 addresses. Naive merging dictates that only address ranges that share the same most significant address bits and routing table lookup results, be allowed to be combined. However, there is no fundamental reason to limit the algorithm to combine only address

ranges that have identical most significant address bits. In general, address ranges that share some *subset* of address bits can be merged. For example, if the merging algorithm focuses on the 0-th, 2-nd, and 3-rd address bits in the example of Figure 2, then non-contiguous address ranges (e.g., 0000, 0001, 0100, 0101) can be combined, leading to a reduction of the number of cacheable address ranges from 8 to 3 (marked as A, B, C).

The address range merging algorithm aims to identify a subset of the address bits as the index bits into the cache and starts the merging process with an empty index bit set. Intuitively the index bits are bits in which mergeable address ranges differ. Assume that a cache has 2^K sets and thus K index bits need to be chosen. Assume that K' bits, where $K' < K$, have been chosen already. These K' bits decompose the address space into $2^{K'}$ partitions, each of which contains a set of distinct address ranges. Within a partition, address ranges that are not adjacent to each other in the original IP address space may be adjacent in the partition's address space. If neighboring address ranges within a partition share the same lookup result, they can be merged into a larger address range. Thus, the $K' + 1$ -th index bit is chosen to be that bit which minimizes both the total number of address ranges and the variance in the number of address ranges, across the $2^{K'+1}$ induced partitions.

Once the routing table cache data structure is constructed and put in the external network interfaces, at run time the network processors simply select the K chosen index bits from the incoming packet's destination address and use these bits as an index into a locally maintained cache. The network processor then performs tag matching on the retrieved cache entry to verify that the fetched address range indeed contains the given destination address. If so, the routing table lookup operation is completed and the packet is forwarded to the output router node. Otherwise, a cache miss occurs. Cache misses are serviced by a *classifier* module running on every router node's main CPU. The classifier exposes a request queue to each external network interface on its node, services requests from these queues in a round-robin fashion, and eventually posts packets back to the forwarding path. After routing table lookup, if a packet is destined to a remote router node, it is posted to the local node's internal interface. Otherwise, it is posted to an output queue on the local node's main memory.

Computation of the routing table cache is done by a separate administrative process, running on one of the CPUs. This is to ensure that a single consistent copy of the routing table is computed from routing protocol updates. Upon a change to the global routing table, the network processor's caches are invalidated. If the new routing table yields a new set of index bits that should be used, then the new set is also made known to each of the network processors.

For higher dimensional packet classification, every dimension of the filter rule is considered as an independent address space. *Ranges* in each address space are the results of projecting the filter expressions along each dimension, and a set of index bits is selected for each address space. Given an incoming packet, the lookup algorithm then looks for each packet header field in the appropriate address space, and finally takes an intersection of each dimension's results. Classification is only done up to a fixed number of fields in the

network processor. If more dimensions are required for some rules, they are mapped to a special class that is equivalent to a "forced cache miss." This causes packets lying in this special class to be posted to the classifier function, which then uses additional fields to perform the complete classification.

Classification of real-time flows is simpler and requires only a lookup into a flat mapping table to translate flow IDs to output queue IDs. This is possible because we assume that real-time flows have a fixed-length and fixed-structure header, e.g., an RSVP header, from which flow IDs can be easily extracted.

The results of all forms of classification is a queue. For instance, if the packet should simply be forwarded after route lookup, this is the output queue on the output interface. In case the packet should be processed by a function, this is the *argument queue* of that function (as explained in section 3.2.2). Clearly, queues should be identified in a location-dependent manner. In particular, queue identifiers in our system are the logical shift-and-OR of a router node id, a type and a queue id. The type can be the argument queue of a function, or a packet queue, and is understood by both the network processors and the CPUs.

3.1.2 Output Link Scheduling

To guarantee guaranteed QoS, *Panama* allocates a separate queue for each real-time connection, and implements *discretized fair queuing scheduler* (DFQ), which maintains fairness at a fixed-granularity time scale, rather than at an infinitesimally small time scale. Given a chosen time granularity T , it can be proved [5] that the differences between Fluid Fair Queuing (FFQ) and DFQ, in terms of per-hop and end-to-end delay bounds, are proportional to T , but is independent of the number of real-time connections at each hop. This is an important result because the deviation from ideal FFQ bounds is constant, even though the implementation of DFQ is much simpler than FFQ emulation schemes.

Each flow f contending for an output link makes a resource reservation characterized by three parameters, a per-hop delay bound D_f , a long-term bandwidth requirement B_f , and a maximum data burst size P_f , which is used to derive the temporal allocation granularity of B_f . Thus, the flow f is eligible for at most P_f units of service over a time period of $\frac{P_f}{B_f}$, which we call the *service cycle*. During a service cycle, the flow is served with an instantaneous bandwidth of $IB_f = \frac{P_f}{D_f}$. We define the *quantum size* of the flow, Q_f , as $IB_f * T = \frac{P_f * T}{D_f}$. Thus, the quantum size of a flow is the amount of data from it that gets served in one scheduler cycle of length T . This separation of long-term bandwidth (B_f) and instantaneous bandwidth (IB_f) allows the decoupling of delay and bandwidth guarantees, and is particularly important for low-latency and low-bandwidth real-time connections such as Internet telephony applications.

The link scheduler decomposes time into scheduling cycles of length T , and in each scheduling cycle, retrieves a quantum worth of data from every eligible flow to form a transmission batch and forwards it over the link, as shown in Figure 3. Flows become *ineligible* if they have already received the maximum amount of service that they should get in a service cycle. DFQ link schedulers also achieve a cut-

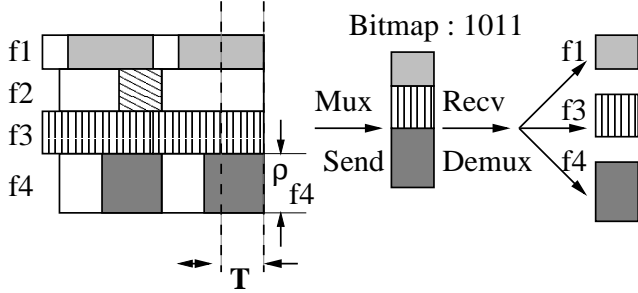


Figure 3: In this example, the scheduling cycle time of DFQ is T and the link capacity is C . In every cycle, a quantum worth of data is picked from each flow’s queue to form a batch. Every flow’s reservation is depicted as a channel that allows a maximum data volume P to be served over an interval $\frac{P}{B}$. The instantaneous bandwidth that can be received by a flow is represented by the height of its channel $\frac{P}{D}$. For every batch, a bitmap is used to represent the information that the downstream router needs to demultiplex quanta and map them to their corresponding flows.

through forwarding effect through a network path and thus improve end-to-end delay bounds [5]. Together with each transmission batch is a bitmap that indicates the flows that contribute a quantum to the batch. The bitmap is of fixed size and is specified with respect to the first flow id that has a quantum in the batch. Operationally, neighboring routers need to agree on a simple multiplexing/de-multiplexing protocol on the connecting link, *because DFQ does not observe packet boundaries*. Given a starting flow id and a bitmap, a downstream router can derive the set of flow IDs that have a quantum in the batch, followed by a map-table lookup to yield the output flow IDs (Figure 3).

3.2 Packet Computation

Two design issues in supporting extensible packet processing functions in Internet routers are *memory protection*, i.e., buggy router extensions cannot compromise the router core, and *performance isolation*, i.e., the router core’s performance should not be adversely affected because of the existence of compute-intensive router extensions.

3.2.1 Safe and Efficient Packet Processing Function Invocation

The key design challenge for supporting memory protection is to develop a *safe* and *efficient* extension mechanism in the router OS. Although safety can be achieved through various hardware/software protection schemes such as using separate address spaces, sand-boxing, or type-safe programming languages, *Panama* exploits the segmentation hardware protection features in the Intel X86 architecture [7] to protect the router OS from programmable router extension functions. In *Panama*, the router OS is put at the highest segment privilege level (0), while all router extensions are placed in separate segments at a lower segment privilege level (1). As a result, *Panama* allows extension functions to reside in the same address space as the router OS and thus avoids the full context

switching overhead, such as TLB flushing, when the router OS invokes router extension functions. On the other hand, the built-in segmentation hardware check in X86 architecture guarantees that extensions cannot corrupt the router OS, because extensions and the router OS occupy different segments at different protection levels.

Since X86 hardware does not allow more privileged domains to initiate an inter-protection-domain control transfer, an efficient emulation of an inter-segment control transfer operation has been implemented, as described by [6], which takes 142 CPU cycles for an empty call and return between two protection domains. Our current intra-address-space protection scheme only protects the router OS from extensions, but does not protect extensions from one another.

3.2.2 Composing and Scheduling Packet Computation

The main CPU on each router node performs three main types of packet computation: servicing classifier cache misses, output link scheduling, and router extension processing. The link scheduler on every router node is driven by a periodic timer to ensure its timely and deterministic invocation. *Panama* chooses to service classifier cache misses at a higher priority than router extension functions, because classifier cache miss handling is considered part of a router’s core function, and thus should be protected from “add-on” functions such as router extensions. When there are no pending classifier cache miss requests, *Panama* runs a potential-based CPU scheduler [1] to allocate computation resources among router extensions.

The set of extension functions is physically organized as a *function graph*, where each node represents a self-contained extension function and links between nodes represent control flows. However, unlike function calls, control transfer from one function to another is *asynchronous*, in that invocation arguments are *posted* to an *argument queue* of the called function, but the time of actual invocation is determined by the CPU scheduler. To compose packet processing computation for a flow, a path is first constructed in the function graph to achieve the desired functionality (Figure 4). This path of functions forms the *compute path* of the flow. Network packets are *bound* to such compute paths at run time through the classifier module, which maps packets to an index into a table of compute paths. The CPU scheduler itself supports a *computation request queue* for each flow requiring packet processing computation, which represents the starting point of each flow’s compute path. After a given packet is classified, a request is enqueued to the computation request queue associated with the packet’s flow.

Once the CPU scheduler chooses to schedule a function operating on a packet, the function runs to completion and posts a new invocation to the argument queue of the next function on the packet’s compute path. The control then returns back to the CPU scheduler. The intermediate state describing the progress of a packet along its compute path is kept in the associated flow’s compute path table entry. Note that a packet from a flow must complete its compute path before the next packet from the same flow may be processed. Also, in the process of compute path execution, the packet’s payload is never copied; only references to the payload are created and consumed.

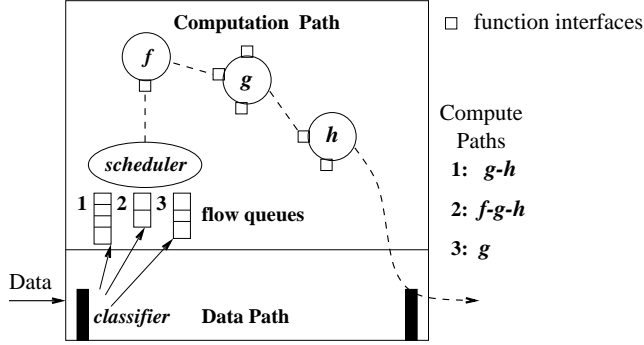


Figure 4: An example of composing computation on three flows. Each flow is bound to a compute path, shown on the right, with Flow 2’s compute path marked by a dashed line. The classifier posts every packet of a flow requiring packet processing to the flow’s computation request queue at the scheduler. The scheduler receives control after each function invocation and selects the next function to invoke according to a scheduling policy.

The compute paths associated with multiple flows can intersect in the function graph. Essentially, this means that the function at the intersection point is being used by multiple flows. However, invocations of functions for separate flows are kept physically disjoint through an independent argument queue for each flow, as shown in Figure 4. Figure 4 shows an example in which three connections are bound to three different compute paths. A function has as many interfaces to accept invocation requests as the number of compute paths that pass through it. The compute path of Flow 2 is illustrated in the figure. Control is logically passed between functions in the compute path. However, in reality, every function invocation returns control to the CPU scheduler, which then decides the next function to invoke.

A packet processing function, that is running in response to an invocation on one of its interfaces, can only be preempted by invocation requests to *other* functions, but not by an invocation on another interface of the same function. This means that it is safe for packet processing functions to be non-reentrant. Although this greatly simplifies implementation complexity, it also introduces preemption interdependence among compute paths that share a function. Note that since the CPU scheduler gets control on timer interrupts, no packet processing function can prevent the service of classifier cache misses or output link scheduling. In particular, the classifier function’s service latency is bounded by the minimum of the time taken by the longest function invocation, and the timer period.

To ensure that packets of a real-time flow that requires high-level packet processing receive sufficient CPU resource to meet their end-to-end performance objective, the CPU scheduler uses the flow’s reservation and packet processing requirements to automatically determine their CPU reservations, and schedules them accordingly. Assume B_f and P_f are the long-term bandwidth requirement and maximum data burst size for the flow f . Also assume that the number of CPU cycles required by the functions in f ’s compute path, for processing P_f bytes worth of packet data, is X . Then connection f implicitly requires a CPU bandwidth

of $\rho_f = \frac{X}{T_f}$, where $T_f = \frac{P_f}{B_f}$, to maintain its guaranteed throughput through the router.

The CPU scheduling algorithm attempts to equalize the *potential* of all competing flow. We borrow the notion of potential from [1], and give appropriate interpretations to system potential, per-flow potential and re-calibration instants as follows. Given a CPU bandwidth reservation ρ_f for connection f , if connection f is backlogged, its potential at time t is $W_f(t)/\rho_f$, where $W_f(t)$ is the number of processor cycles spent on processing packets of connection f during the time period $[0, t]$. For a non-backlogged connection, the potential is set to the *system potential*, which is the total amount of CPU time spent on packet processing for any of the flows. *Re-calibration instants* for the system potential are all time instants at which some function returns. Thus, whenever the CPU scheduler takes control after a function invocation returns, it updates the system potential according to the definition above, and picks the flow with the least potential as the next function to run.

3.3 Forwarding and Processing Path Synchronization

In *Panama*, there is a packet forwarding path, mainly through network processors, and a packet processing path, mainly through CPUs, and they need to synchronize with each other from time to time. All interactions between the forwarding path and the computation path are producer-consumer interactions. Instances of such interactions arise throughout the system. For instance, the CPU allocates a buffer pool for each external interface, whose corresponding network processor *consumes* by posting packets and computation requests. On the other hand, the CPU *produces* space in the buffer pool after it completes the service of the requests or packets. Other examples include network processors at external interfaces *producing* classification miss handling requests that the classifier module on CPU *consumes*, and internal interfaces *producing* scheduling or packet computation requests that the CPU scheduler *consumes*.

To minimize the performance overhead of synchronization, *Panama* uses a *lock-free queue* as the basic building block to support all the concurrent accesses from the packet forwarding and processing paths. The key idea here is to break all concurrent accesses into *one-to-one* producer-consumer interactions. For instance, in the case of the interaction between network processors on external interfaces and the classifier module at a CPU, the classifier module is a single consumer for multiple producers. *Panama* provides one request queue for each external interface so that the interaction is now one-to-one producer-consumer. Similarly, in case of the per-external-interface buffer pool, free buffers would be produced by the link scheduler running at various output interfaces, and by several extension functions. In this case a one-to-one interaction is maintained as follows. A single link scheduling function performs scheduling for all the output interfaces at a node. Likewise, the CPU scheduler performs buffer accounting for packets that require processing. Thus there are actually only two producers for the free buffer pool and each buffer pool is split into two queues to form two one-to-one interactions. Whenever an interaction is broken to one-one in this manner, a simple round-robin ar-

bitration is performed among the split request/data queues. In case of the argument queues and per-connection packet queues, the interaction is already one-to-one since there is a unique queue for every flow.

Given a one-to-one producer-consumer relationship, concurrent accesses to a shared queue can be resolved without locks by always keeping at least one element in every queue, where the *type* of the element distinguishes an empty queue from a queue with one or more elements. Specifically, when the consumer consumes the last queue element, it leaves an element of type *void* in the queue. With this setup, the consumer’s criterion for queue emptiness is changed to the following: a non-empty queue either has a non-*void* descriptor at its head, or a *void* descriptor with a successor. If a queue is found to be non-empty by the second criterion, the *void* descriptor is discarded and the next element is consumed. Note that an invariant holds that only the first element of a queue can be *void*, since a producer always produces non-*void* elements. It can be shown that as long as *inserting* an element to the tail of a queue (which is a single memory write for simple queues) is atomic, this lock-free mechanism eliminates any inconsistency between the producer and the consumer.

Although the above mechanism ensures that no shared queue element is garbled due to a race condition, there remains a subtler issue of triggering relevant computation on a synchronization event. For example, when a queue changes from empty to non-empty, its consumer should be scheduled to run on the CPU. This problem is solved by ensuring that all consumers of shared queues that need to be woken up on the empty-to-non-empty transition be *polling* consumers. For instance, the link scheduler is guaranteed to get woken up on timer interrupts, picking up any missed empty-to-non-empty queue transitions. Similarly, the CPU scheduler (and hence the classifier) is guaranteed to get control after every function return and/or timer interrupt.

4 Prototype Implementation

The current *Panama* prototype consists of four 400-MHz Pentium-II PCs as router nodes, connected by a 8-port Myrinet switch with a 10-Gbit/sec backplane and full-duplex 1.28 Gbps links. Each router node has two Myrinet [18] interfaces, each of which is equipped with a Lanai 4.X network processor. The *Panama* kernel is derived from Linux and runs on the network processor and the main CPU.

4.1 Intra-Cluster Packet Movement

As shown in Figure 5, in the most general case a packet traverses a path from the external interface of the ingress router node, through the ingress node’s internal interface and the egress node’s internal interface, and eventually to the egress node’s external interface. The external interface of the ingress node performs a packet classification operation to decide a queue to which the packet should be enqueued. The queue could be a packet processing request queue, or a real-time/best-effort packet queue at an output interface. Further, this queue could be local to the ingress node, or may reside on the egress node. In the case of a local target queue, the ingress node also plays the role of the egress node.

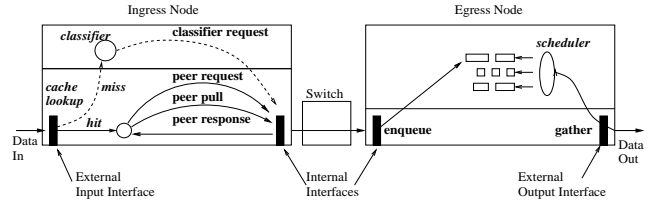


Figure 5: A generic packet forwarding path through Panama includes two router nodes and four interfaces. The ingress node’s external interface classifies the packet and performs a request-response protocol and peer-to-peer DMA with its internal interface to send the packet to the egress node. Misses in the classification cache at the external interface trigger a request to be posted to the CPU, which asynchronously services the request and forwards the packet to the ingress node’s internal interface. The egress node’s internal interface receives and enqueues the packet and its CPU performs output link scheduling to actually puts the packet on the wire.

In the general case, the ingress and egress nodes are different; an incoming packet is first moved from the ingress node’s external interface to its internal interface through a *peer-to-peer* DMA transaction, and then to the egress node’s internal interface through a network transfer transaction over Myrinet. The peer-to-peer DMA transaction allows the packet to appear on the I/O bus (PCI in this case) exactly once, and proceeds using a request-response protocol. The external interface DMAs a request to a designated area in the memory of the internal interface and blocks waiting for a notification. The internal interface, which may get multiple such requests from multiple external interfaces, services these requests in a round-robin fashion by pulling the data from the source interface, again through a peer-to-peer DMA, and finally DMAs a notification to a designated area in the external interface’s local memory.

Upon receipt of a packet over the router’s backplane, the egress node’s internal interface enqueues it to an appropriate queue in the node’s main memory, based upon the classification decision that the packet carries with it. Finally, the link scheduler schedules data for transmission, by setting up the corresponding external interface to perform a gather DMA from the node’s memory out on the output link.

In the event that the external interface of the ingress node encounters a classifier cache miss, it posts a classification request to the ingress node’s CPU, which, after servicing the lookup request, writes a pull request to the node’s internal interface, exactly like a peer-to-peer DMA request made by external input interfaces to the internal interface. This mechanism allows the packet to “flow” back to the forwarding path. Note that the CPU is *not* interrupted regardless of whether a packet encounters a classification cache miss or not. Instead, each router node’s CPU polls the lookup request queue periodically to determine whether there are classification cache misses to process. Similar “posting” mechanisms are used for output link scheduling and packet computation requests at the egress nodes. Therefore, *Panama*’s data packet movement process is *interrupt-free*.

The movement of a packet through the forwarding path is heavily pipelined among the network processors on network interfaces and CPUs on the ingress and egress nodes. Figure

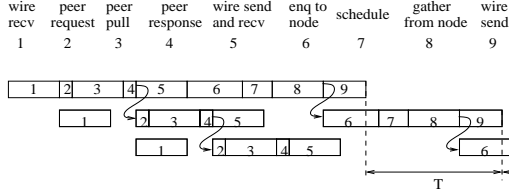


Figure 6: Pipeline boundaries for the four-interface forwarding path in Figure 5. The pipeline’s critical path in this example is the combination of stages 6, 7 and 8, and determines the pipeline’s cycle time T . The arrows between pipeline stages represent dependencies among individual pipeline stages due to resource contention.

6 shows the nine steps involved in moving a packet from the input link to its corresponding output link. Each step takes a different amount of time. Since some steps may use the same hardware resources, the final design is a 5-stage pipeline rather than a 9-stage one. Specifically, steps 2, 3 and 4 are combined into one pipeline stage because they all need to use the ingress node’s PCI bus, and steps 6, 7 and 8 are combined into one pipeline stage because steps 6 and 8 need the egress node’s PCI bus. In case of our hardware platform, the “cycle time” of this pipeline, i.e., the critical path delay, is dictated by the combined delay of the 6-th, 7-th and 8-th steps.

However, note that pipeline boundaries are enforced by the network processor code and are not hard-coded. In particular, on different hardware platforms, the pipeline cycle time may be different.

Apart from explicit synchronization where network processors defer their next request until the previous invocation request is complete, there is an implicit pipeline synchronization between the egress node’s internal interface and its main CPU: the internal interface’s enqueueing operation may be stalled if the free buffer pool is exhausted, to wait for the CPU to release buffers. Similarly, there is an implicit pipeline synchronization requirement between the egress node’s CPU and the output interface that the packet takes to leave the router: the link scheduler may get blocked if the output interface has not finished the gather DMA of the previous cycle.

4.2 Integrated Packet Queuing and Link Scheduling

In the 6-th stage of the pipeline described in Section 4.1.2, a batch of quanta arrives at the egress node’s internal interface. Enqueueing each quantum in the batch to its respective per-connection queue requires a separate short DMA to the current tail of the queue. As a result, the overhead of this short DMA burst would seriously degrade the efficiency of the overall data transfer pipeline. To solve this performance problem, the internal network interface is required to send a summary data structure about each arriving batch of quanta to the CPU, which then performs the enqueueing operations directly against main memory. Consequently, only a single DMA is required to transmit the summary data structure and the batch.

The data structure that summarizes which flows a batch

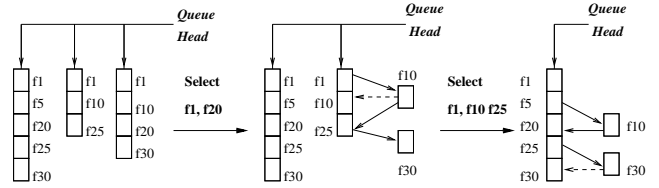


Figure 7: Three batches arrive at an output interface. An example 2-cycle run of the consume-and-thread algorithm. Unsent quanta of a batch are threaded both vertically and horizontally into the subsequent batches, which are to be serviced by the link scheduler subsequently. Solid and dotted lines represent vertical and horizontal threading respectively.

of quanta belong to is a run-length encoded bit map. For example, assume there are 30 connections sharing an output link, and a batch is received in which flows 1 through 10 and flows 20 through 30 have quanta present. Logically this information can be represented as $(1, 10, 1)(0, 10, _)(1, 10, 1)$. Each triple corresponds to a run. The first field in a triple indicates whether the flows in the run have quanta, 1 meaning yes and 0 meaning no. The second field represents the run’s length and the third field represents the number of quanta from each flow if the first field is 1. The same representation is used to represent the per-connection queues associated with an output interface. So when a batch of quanta arrives at the internal network interface of an egress node, the interface forwards the batch’s corresponding encoded bitmap to the CPU, and the CPU simply logically “ORs” the new bitmap with the bitmap representation at the corresponding output interface. For example, if the current status of the per-connection queues is $(0, 5, _)(1, 10, 1)(0, 15, _)$, then the logical OR of the batch’s bitmap and per-connection queues’ bitmap is $(1, 5, 1)(1, 5, 2)(1, 5, 1)(0, 5, _)(1, 10, 1)$. Similarly, when the output link scheduler sends out a batch of quanta, the set of quanta should be “subtracted” from the per-connection queues’ bitmap accordingly.

The subset of quanta that are not sent out in the current scheduling cycle are “threaded” into the next batch. This threading operation is the physical realization of the bitmap merge operation: quanta that belong to the same connection are horizontally threaded, and quanta that belong to connections between which there are no other backlogged connections are vertically threaded. Figure 7 illustrates how the threading operation works in two consecutive scheduling cycles. In the first cycle, the scheduler selects only flows 1 and 20 for service. This leaves quanta for flows 10 and 30 to be carried over to the next batch. A merging scan through the next batch would thread flow 10’s quantum horizontally with its fellow packets, and the quanta from flow 10 and 30 vertically with their neighboring connections. In the next cycle, quanta from flows 1, 10 and 25 are selected, causing quantum from flow 30 to be horizontally threaded, and quanta from 10 and 30 to be vertically threaded. Without consume-and-thread, a total of $4 + 3 + 5 = 12$ short DMAs would be required for this example. However, with consume-and-thread, only 3 short DMAs are required, one for each batch.

5 Performance Evaluation

Performance measurements are made on the *Panama* prototype described above. All the reported results are based on measurements from a single packet forwarding path between two router nodes through four network interfaces. Because the router backplane supports significantly higher bandwidth than can be saturated by individual paths, the aggregate performance of a *Panama* router is N times the measurements reported below, where N is the number of disjoint pairs of router nodes. Two other PCs are used as the source and sink hosts that drive traffic into and receive packets from the *Panama* prototype. Byte and packet throughput are measured by sending packets back to back from the source to the sink. Inter-packet gap is measured at the sink and if it is within a small percentage of the inter-packet gap at the source, the source further reduces the inter-packet gap. This adjustment continues until *Panama*'s throughput saturates and no further reduction in inter-packet gap at the sender side is possible. All timing measurements have been taken using the cycle count register on the Pentium-II processor of the source and sink hosts. Each cycle is worth 2.5 nsec.

5.1 Throughput Results

Panama supports both FIFO and DFQ output link scheduling. In this section, we evaluate the overhead of DFQ compared to FIFO. We also evaluate the scalability of DFQ with respect to the number of concurrent real-time connections.

Non-FIFO link scheduling such as DFQ ensures that the output link bandwidth be shared among competing flows based on their resource reservations, but may incur additional scheduling overhead. The scheduling overhead of DFQ is due to the multiplexing and de-multiplexing operations required for quantum-size rather than packet-size transmission. Figures 8 and 9 show the differences in byte throughput and packet throughput between FIFO link scheduling and DFQ link scheduling, as the quantum size used in DFQ and the packet size vary. Each data point in these figures represent the measured throughput, in bytes/sec and packets/sec respectively, when a sequence of packets, each of the corresponding packet size, are transferred from the source host to the sink host, through the *Panama* prototype. For DFQ link scheduling, we also vary the quantum size, which is the unit of "discretization" as compared to Fluid Fair Queuing. For a given quantum size, we measure the router throughput only for packet sizes that are larger than the quantum size.

For FIFO scheduling, the byte throughput increases with the packet size, because each DMA transaction at the sender side amortizes its per-transaction overhead over a larger packet and is therefore more efficient. For DFQ scheduling, the byte throughput is independent of the packet size but depends on the quantum size, because the size and thus efficiency of each DMA transaction in DFQ is dependent on the quantum size, regardless of the packet size. The larger the quantum size is, the more efficient DFQ's DMA transactions are. Compared to a DFQ scheduler whose quantum size is the same as the packet size in the input traffic, the FIFO scheduler actually shows a lower byte throughput because at the *receiver* end of a network link, DFQ only needs

to perform a single DMA transaction for a batch of quanta whereas FIFO requires one DMA transaction for each independent packet. However, FIFO can continue to exploit the increasing packet size to improve the DMA transactions' efficiency at the sender end and eventually out-perform all DFQ instances with a fixed quantum size, in terms of byte throughput.

For the FIFO scheduler, as the packet size increases, the byte throughput increases but the number of packets transmitted within a unit time decreases, and the overall net effect is that the packet throughput decreases. For the DFQ scheduler, only the number of packets transmitted within a unit time decreases with increase in packet size, but the byte throughput remains unchanged. Therefore, the slope of the decrease in packet throughput for DFQ is steeper than that for FIFO.

The byte and packet throughput differences between DFQ and FIFO represent the cost of real-time link scheduling. As shown in Figure 8 and 9, this cost is less than 50% for packet sizes smaller than or equal to 1000 bytes. For even smaller packet sizes, this cost is actually *negative*, because DFQ's batching improves the receiver side's DMA efficiency. In addition to low scheduling overhead compared to FIFO scheduling, DFQ is also more scalable in that its per-flow scheduling overhead does *not* depend on the number of real-time connections that share the same output link, which has been the deficiency for other real-time link schedulers based on packetized weighted fair queuing. To substantiate DFQ's claim of $O(1)$ scheduling overhead, we measure the packet rates versus the number of real-time connections for a sequence of 128-byte real-time packets and a quantum size varying from 32 bytes to 128 bytes, increasing by factors of 2. Figure 10 shows that the overall packet throughput remains almost constant when the number of real-time connections varies from 320 to 3200.

5.2 Latency Results

Quantum Size (bytes)	Ingress Node Latency (cycles)	Egress Node Latency (cycles)	Pipeline Cycle Time (cycles)
16	56,580	79,298	56,994
32	61,064	85,229	59,279
64	70,644	95,197	65,302
128	88,852	124,195	98,794

Table 1: *Latency measurements on the ingress and egress nodes for real-time packets. The ingress node latency is the combined latency of pipeline stages 1 through 5 in Figure 6 and the egress node latency is the combined latency of stages 6 through 9. The last column shows the pipeline cycle time for the entire operation, which is always less than the latency due to overlap between pipeline stages.*

Besides throughput, latency is another important performance metric, because it shows how effective the pipelined datapath implementation is. In Figure 6, the combined latency of stages 1 through 5 is the latency at the ingress node, whereas the combined latency of stages 6 through 9 is the latency at the egress node. Table 1 shows the latency mea-

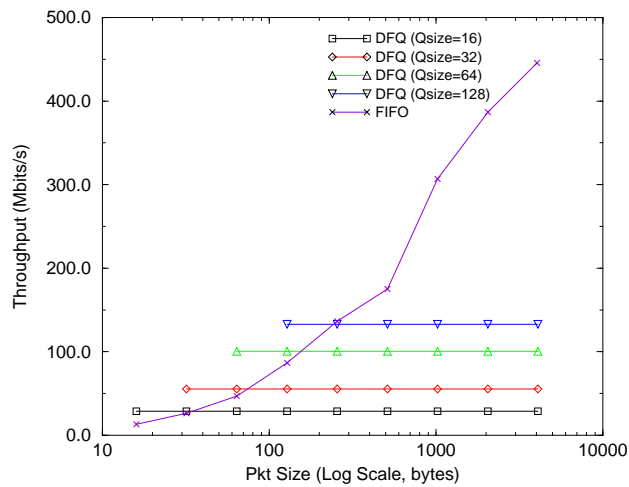


Figure 8: Throughputs in bytes/sec for FIFO and DFQ schedulers with varying packet size and quantum size.

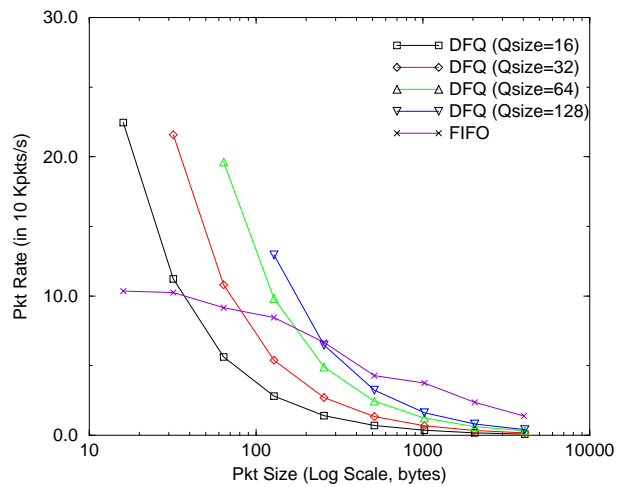


Figure 9: Throughputs in packets/sec for FIFO and DFQ schedulers with varying packet size and quantum size.

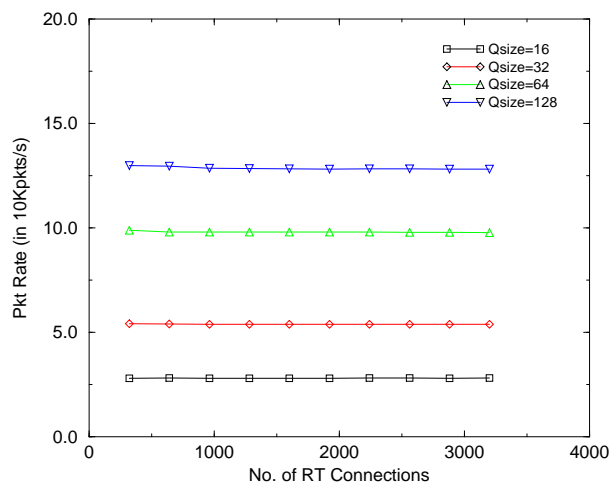


Figure 10: The constant scheduling overhead of DFQ allows Panama's packet rate to remain unaffected by increasing numbers of real-time connections.

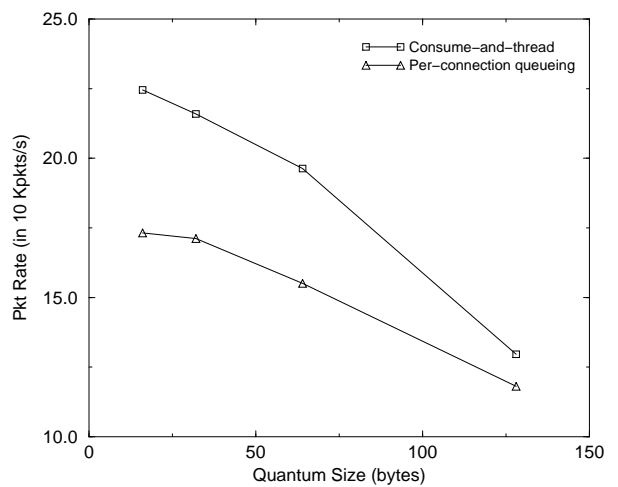


Figure 11: Performance improvement in packet rate from the consume-and-thread algorithm, which avoids unnecessary DMAs, decreases with increases in quantum size.

measurements on the ingress and egress nodes for real-time data transfers with varying quantum size. These numbers correspond to a batched transfer of 32-quanta batches. These measurements demonstrate the effectiveness of *Panama*'s pipelined implementation by showing that the pipeline cycle time in each case is less than the overall latency as well as the latency of the bottleneck node (viz the egress node in this case). The ratio between the sum of ingress and egress node latencies and the pipeline cycle time reflects the extent of pipelining at work.

To reduce the delay of the 6-th pipeline stage in Figure 6, the consume-and-thread algorithm was developed to avoid short DMAs that would have been required if the internal interface at the egress node appends each incoming quanta to its corresponding queue. By relaying relevant information to the CPU through a summary data structure, and letting the CPU perform the queue appending operation, the overall packet throughput increases significantly, especially for small packets, as shown in Figure 11. However, as the quantum size increases, the overheads of short DMAs are more and more dominated by the per-byte data transfer cost, and therefore the significance of this optimization diminishes.

5.3 Routing Table Lookup Performance

When a network processor on a *Panama* router node's external interface fails to classify an incoming packet, it posts a request to its associated CPU, which then services the request and posts the packet back to the forwarding path. In this section, we evaluate the performance of routing table lookup in our prototype.

To evaluate the routing table lookup operation in isolation, we measure the hit access time and the miss penalty of a lookup operation. I Lanai processor's cycle time is worth approximately 12 Pentium cycles. A lookup resulting in a hit takes 4705 Pentium cycles, or ~ 392 Lanai cycles. This overhead is mainly due to the fact that selecting the index bits and the tag from a packet's destination address has to be done in a 32-iteration loop, since there is no hardware bit selection primitive. The cache miss penalty is the extra overhead incurred in posting a request to the classifier, and in a full search through the routing table data structure, and is measured to be 7560 Pentium cycles for 64-byte packets.

To understand the effectiveness of the proposed aggregate route caching scheme, a packet trace was run through the *Panama* prototype, which has been collected from an Internet edge router (the main router of Brookhaven National Laboratory), and its cache hit ratio and routing table lookup rate were measured. To be conservative, we used a 1024-entry cache in the network processor, which corresponds to 10 index bits. The measured cache hit ratio is 92.36%. Therefore, the average service time for a routing table lookup is 4,923 cycles. If the routing table lookup is the bottleneck, then a packet rate of 81.25 Kpackets/sec can be sustained for this trace. This less-than-stellar routing table lookup performance is because long hit access time, which in turn is due to lack of an efficient bit selection primitive, and the low clock rate the Lanai network processor. If the network processor's clock rate improves from 33 MHz to 330 MHz, the overall routing table lookup performance per interface would be close to 1 million packets per second.

5.4 Packet Computation Overhead

Panama features a highly efficient safe kernel-level router extension mechanism. To evaluate the performance impacts of the invocation overhead of this router extension mechanism, we measured the packet throughput of a 64-byte packet sequence, some of which need to call a router extension function. The extension function used in this experiment first calls a null kernel service, writes a word to a memory region shared by the kernel and the router extension, and returns to the kernel. Thus, in every extension invocation, two protected function calls are being made. Recall that the overhead of a null protected function call is 142 cycles. Figure 12 shows the packet rates versus the percentage of packets in the input sequence that need additional "null" packet computation. The packet rate decreases slowly with the increasing percentage of input packets that need additional packet processing. The small slope in this figure demonstrates that the invocation of *Panama*'s safe router extension mechanism is indeed quite efficient. The extension function used in this experiment does nothing but write a word to a memory region shared by the kernel and the router extension. As the number of extension function calls that each packet calls increases, the packet rate decreases, with a constant slope that corresponds to the overhead of invoking a router extension function. An alternative way to measure the effectiveness of the

Figure 12 shows how the packet rate changes as the percentage of flows that require computation worth one extension function increases. The slope will be given by single extension overhead divided by the number of connections considered in the experiment. The experiment shows that the incremental overhead of computation is small and does not significantly degrade the overall packet rate.

5.5 Importance of Packet Computation Scheduling

A key feature of *Panama* that does not exist in other extensible routers is its ability to integrate CPU scheduling for packet computation with output link scheduling to achieve end-to-end performance guarantees. To demonstrate the usefulness of this integrated resource scheduling scheme, we conducted an experiment in which there are two competing flows, flow 1 and 2, each reserving $\frac{2}{3}$ and $\frac{1}{3}$ of the the link bandwidth and CPU capacity respectively. In addition, while flow 1 sends data in accordance with its reservation, flow 2 sends at a rate twice its reservation. Figure 13 shows the time line of packets received for each flow, with and without scheduling. In the absence of packet computation scheduling, the CPU simply serves packets from these two flows as they arrive, without regard to their reservations, and thus divides its capacity equally between the two flows. As a result, the aggressiveness of flow 2 causes flow 1 to receive less CPU resource than it needs and to leave some of its reserved bandwidth unused. Since the link scheduler is work-conserving, the unused bandwidth actually becomes available to flow 2. Consequently, the link bandwidth is also equally divided between the two flows. However, in the presence of packet computation scheduling, flow 2 only gets $\frac{1}{3}$ of the processing cycles and flow 1 gets $\frac{2}{3}$ of the processing

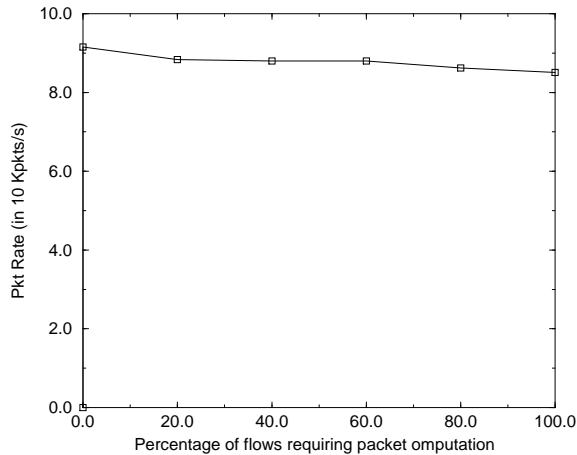


Figure 12: The performance impact of the percentage of packets in the input traffic that require additional packet computation on Panama’s packet throughput. The slope is indicative of the invocation overhead of Panama’s safe router extension mechanism.

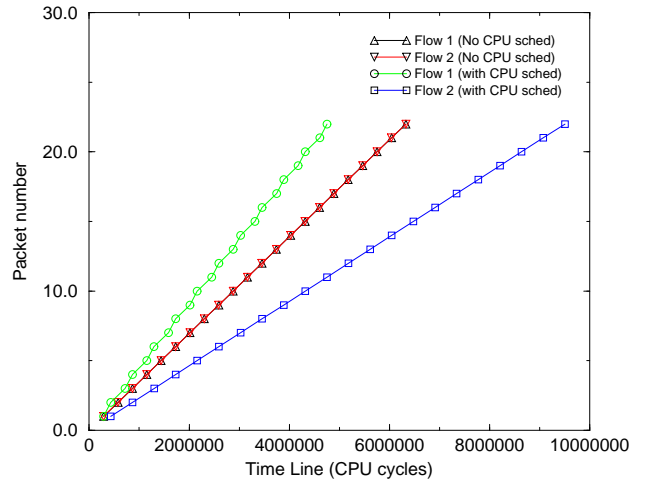


Figure 13: Without packet computation scheduling, flow 1 and 2 progress at the same rate. With packet computation scheduling, flow 1 progresses at twice the rate as flow 2, exactly according to their original link/CPU reservations.

cycles. Eventually each flow is progressing at a rate exactly according to its reservation, as shown in Figure 13. This experiment shows that true end-to-end performance guarantees can not be achieved without integrating packet computation scheduling and output link scheduling.

5.6 Lessons

The experiences from designing and implementing the *Panama* prototype teach us several important lessons about using PC clusters as the underlying computing platform for edge routers. First, the most performance limiting factor in the *Panama* hardware architecture is the PCI bus, in particular, the inability to efficiently execute short DMA transactions due to fixed per-transaction bus arbitration overhead. Although improving PCI’s raw bandwidth might help, i.e., by upgrading to 66-MHz and/or 64-bit PCI, we believe such upgrades would not improve *Panama*’s packet or byte throughput by proportional amounts. What is needed is a more intelligent DMA engine on the network interface to fully exploit the pipelining capabilities of the PCI bus. Second, in *Panama* the processors on the network interfaces assume the responsibility of routing and forwarding network packets, and do not appear to be a performance bottleneck. This result demonstrates that by keeping the generic packet forwarding path simple, relatively low-performance RISC processors, 33 MHz processors in our case, can produce adequate packet forwarding performance. Therefore, instead of designing customized processors with special-purpose instruction, a more promising optimization strategy is to increase the clock rate of generic RISC engines. Finally, to further minimize the performance overhead due to the interaction between the network processors and node CPU’s, network interfaces should be equipped with more high-speed

memory so that the network processors can assume more responsibilities, like link scheduling. Alternatively, network interfaces could be provided with a faster access path to the main memory, much as the AGP bus for 3D graphics cards.

6 Conclusion

The main contributions of this work are the development of a scalable and extensible edge router architecture, its realization, and its performance evaluation based on a functional prototype. We believe this is the first set of empirical results ever reported on a cluster-based software router. In addition to the decoupled system architecture that cleanly separates packet forwarding and packet computation paths, we also believe *Panama* boasts several unique architectural and algorithmic features not available in existing router implementations. These features include

- A safe and efficient extension mechanism to support programmable router functionalities,
- An aggregate route caching scheme to increase the effective coverage of a routing table cache,
- A Discretized Fair Queuing link scheduler that provides good delay bounds, while incurring an overhead that is independent of the number of real-time flows sharing an output link,
- A highly efficient interrupt-free and lock-free data movement pipeline, and
- An integrated resource scheduler that does both packet computation scheduling and link scheduling to ensure that the end-to-end performance guarantee of a real-time flows with packet computation is indeed met.

Finally, we have completed a fully-operational cluster-based *Panama* prototype and performed a comprehensive performance study on the prototype, which provides a better understanding of the performance characteristics and design tradeoffs of PC cluster-based edge router architecture, and design lessons on useful features for next-generation network interface processors.

Acknowledgement

This research is supported by NSF awards MIP-9710622, IRI-9711635, EIA-9818342, ANI-9814934, and ACI-9907485, USENIX student research grants, as well as fundings from Sandia National Laboratory, Reuters Information Technology Inc., Computer Associates/Cheyenne Inc., National Institute of Standards and Technologies, Siemens, and Rether Networks Inc.

References

- [1] D. Stiliadis, A. Varma; "Rate-proportional servers a design methodology for fair queueing algorithms"; IEEE/ACM Transactions on Networking, April 1998, Volume 6, Number 2, pp. 164 - 174.
- [2] D. Mosberger, L. Peterson; "Making Paths Explicit in the Scout Operating System"; Proc. OSDI 1996.
- [3] N. C. Hutchinson, L. L. Peterson; "The x-Kernel: An architecture for implementing network protocols"; IEEE Transactions on Software Engineering, 17(1):64#76, Jan. 1991.
- [4] P. Pradhan, T. Chiueh; "Cache Memory Design for Network Processors"; Proc. IEEE HPCA 2000.
- [5] "Discretization in Fluid Fairness : Formulation and Implications"; Technical Report (Author names hidden).
- [6] T. Chiueh, G. Venkitachalam, P. Pradhan; "Integrating Segmentation and Paging Protection for Safe, Efficient and Transparent Software Extensions"; Proc. ACM SOSP 1999.
- [7] "Intel Architecture Software Developer's Manual" (<http://developer.intel.com/vtune/cbts/refman.htm>)
- [8] J. Liedtke; "Improved Address Space Switching on Pentium Processors by Transparently Multiplexing User Address Spaces"; GMD technical report (<http://i30www.ira.uka.de/publications/pubcat/As-pent.ps>).
- [9] T. Chiueh, P. Pradhan; "High Performance IP Routing Table Lookup Using CPU Caching"; Proc. IEEE INFOCOM 1999.
- [10] D. Decasper, Z. Dittia, G. Parulkar, B. Plattner; "Router Plugins: A Software Architecture for Next Generation Routers"; Proc. ACM SIGCOMM 1998.
- [11] J. Wroklawski; "Fast PC Routers"; (<http://ana-www.lcs.mit.edu/anaweb/pcrouter.html>).
- [12] "The ATOMIC-2 Project"; (<http://www.isi.edu/div7/-atomic2>).
- [13] L. Peterson, S. Karlin, K. Li; "OS Support for General-Purpose Routers"; Proc. IEEE HotOS 1999.
- [14] P. Pradhan, T. Chiueh; "Operating Systems Support for Programmable Cluster-Based Internet Routers"; Proc. IEEE HotOS 1999.
- [15] R. Morris, E. Kohler, J. Jannotti, M. F. Kaashoek; "The Click Modular Router"; Proc. ACM SOSP 1999.
- [16] V. Vuppala, L. M. Ni; "Design of A Scalable IP Router"; Proc. IEEE Hot Interconnects 1997.
- [17] Michigan University and Merit Network. Internet Performance Measurement and Analysis (IPMA) Project. (<http://nic.merit.edu/ipma>).
- [18] Myricom Inc.; "LANai4.X specification"; (<http://www.myri.com/scs/documentation/mug/-development/LANai4.X.doc.txt>).