Forwarding Path Architectures for Multicore Software Routers

Norbert Egi[§] Adam Greenhalgh[‡] Mark Handley[‡] Mickael Hoerdt[§]
Felipe Huici[⋆] Laurent Mathy[§] Panagiotis Papadimitriou[§]

[§]Lancaster University, [⋆]NEC Laboratories Europe, [‡]University College London

ABSTRACT

Multi-core CPUs, along with recent advances in memory and buses, render commodity hardware a strong candidate for building f exible and high-performance software routers. With a forwarding plane physically composed of many packet processing components and operations, resource allocation in multi-core systems is not trivial. Indeed, packets crossing cache hierarchies degrade forwarding performance, since the bottleneck is main memory access. Therefore, forwarding path allocation and input/output processing become challenging, especially when states and data structures have to be shared among multiple cores. In this context, we investigate a set of input/output processing architectures, as well as resource allocation strategies for forwarding paths. For each packet processing operation, we uncover the gains and possible implications by either running different components concurrently or replicating the same components across different cores. [1]

1. INTRODUCTION

Recent research has shown that modern PCs can perform as well as high-performance routers [1, 2]. With multiple, general-purpose multi-core CPUs and high speed interconnects, inexpensive mid-range server machines can sustain aggregate packet rates in the region of 10 Mp/s (millions of packets per second) for minimum-sized packets (64 bytes), while sustaining line rate on multiple 10Gbps interfaces for longer packets (1,500 bytes). Although PCs are never going to challenge high-end hardware forwarding for raw performance, their low price and programmability make them a very attractive platform for many edge-networking tasks.

Previous work [1] has also shown that the main performance bottleneck of such servers was memory access, through a combination of memory latency and memory/front-side bus overload. Modern server CPUs have a Non-Uniform Memory Architecture (NUMA) with one memory controller per CPU. This can reduce the problem somewhat, but the number of cores per CPU continues to increase and memory latency issues persist as the principal performance-limiting factor for software routers.

To get high performance from software routers it is crucial to reduce memory accesses to a minimum. In commodity PCs this can be accomplished by making sure that packets, as well as most of the data structures needed to process them, stay in cache memory as they travel from an input to an output interface. This makes a *cache hierarchy*, the set of multi-level caches present in a CPU, the basic hardware unit of consideration when implementing software routers. Since f exible software routers often depend on a set of interconnected packet processing elements, the challenge is then to decide how to map these elements to a general-purpose, multi-core CPU architecture so that we reduce memory accesses while maximizing the use of the resources available.

The advent of hardware multi-queuing on NICs provides a building block. By allowing several CPU cores to concurrently access the same physical network interface, this technology can allow these cores to work independently from each other, and thus process packets in parallel. However, sometimes coordination is required between queues to implement a desired behavior such as packet scheduling or bandwidth regulation that is not supported by the NIC. Distributed software output processing, requiring access to shared data, is then necessary between cores. The corresponding synchronization primitives and shared data needed impact performance, especially if these are accessed from different cache hierarchies.

In this paper we investigate how to map packet forwarding pipelines onto cores, making use of multi-queuing NICs and maximizing the effectiveness of the cache hierarchy. We show that the simple, obvious ways to use multi-queuing limits performance, and we investigate more effective ways to lay out the forwarding path across cores.

2. RELATED WORK

The past few years have seen many efforts towards building PC-based software routers. In [3] the authors propose an architecture exploiting multi-core and hardware multi-

[1]This work was partly funded by the UK EPSRC PhD+ Programme and the EU FP7 CHANGE (257422) Project.

queuing to some extent, but they do not support coordinated output processing nor do they investigate splitting or replication of forwarding path across cores, as we do.

Authors in [4] off oad output processing from the CPU to a NetFPGA network card. In particular, they enable network I/O fairness across virtual machines (VM) by applying independent, per-VM rate limiters in hardware. While rate-limiting is useful, networks cards are unlikely to support a wide range of advanced scheduling policies in hardware. Handling these in software requires synchronization mechanisms, and our work investigates the performance impact of doing such coordinated output processing.

RouteBricks [2] explores the scaling of software routers by enabling parallelism across multiple servers. This complements our work, since our approach can be used with RouteBricks to scale router capacity by adding more servers.

PacketShader [5] uses GPUs to improve the performance of a software router. GPUs are particularly good at performing computational and memory-intensive tasks. Consequently, workloads like IPSec (computationally-intensive) or running an Openf ow switch (memory-intensive for large numbers of Openf ow entries) are particularly suited to PacketShader, while tasks that f t in the CPU cache, such as IP forwarding, show small or negligible improvement compared to the CPU-only case. Further, GPUs have the additional cost of having to copy data to the GPU memory and back. In contrast, CPUs are good at handling conditional branches and more complex data structures, so we expect that implementing processing like that of an IDS would produce better results on a CPU than on a GPU. All in all, both our approach and the one in PacketShader lessen the memory access performance bottleneck, and we expect any high-performance, PC-based architecture to use a combination of these depending on the type of work load.

3. PLATFORM

To measure performance we need to choose a software architecture and hardware platform. For hardware, we use an Intel Nehalem-based system. This is a typical modern multicore system, with two Nehalem CPUs (see f gure 1). These are 2.8GHz Xeon 5560, quad-core CPUs, with each core having 32KB L1 instruction and data caches and a 256KB L2 cache. In addition, each processor has an 8MB L3 cache shared among all of its cores. Comparable AMD Opteron systems have a very similar architecture, so our results should apply for most current x86 server-class machines.

Our system has two dual-port 10Gb/s Ethernet PCI-express cards (model 82598EB). These cards support hardware multiqueuing, effectively splitting the card into a set of interfaces (64 for receiving and 32 for transmitting). They also support Receiver Side Scaling (RSS), which uses a crude hash on packet reception to load-balance packet f ows across the set of available hardware queues. Although smarter cards are available (for example, some have programmable CAMs that can map specif c f ows to specif c queues), for the results in this paper the difference is not important.

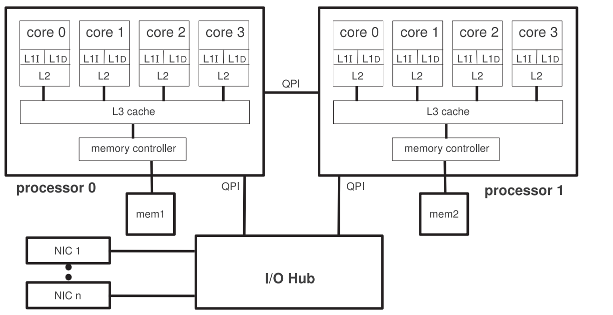


Figure 1: System architecture with two Intel Nehalem four-core processors.

Software Architecture

We use Click [6] in kernel-mode on a Linux 2.6.24.7 system, to perform packet processing, since it provides not only a f exible platform but also yields high performance [1].

Click conf gurations consist of a set of modules (known as *elements*) connected in a data-f ow style graph. Elements provide packet processing functions, for instance queuing or IP table look-ups. We call a *forwarding path* the set of Click elements that any packet traverses from input to output. Essentially a forwarding path consists of chains of pipelined elements, and these chains can be scheduled to run on different kernel threads on specif c cores, giving the parallelism and core aff nity we need to evaluate performance constraints. Click provides a separate scheduler for each Click kernel thread, which schedules execution of the schedulable chains of elements assigned to the thread. We call such a schedulable chain of elements a *task*. A forwarding path is instantiated as a set of tasks interconnected by Click queues.

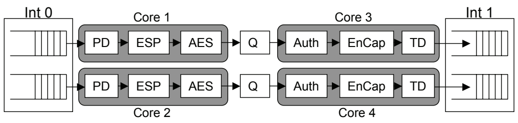
We assign a single Click kernel thread to each CPU core to avoid interference with the Linux scheduler scheduling the threads [7], and then assign the various Click tasks under consideration to these threads as needed. We use Click in polling mode, and all the experiments in this paper use 64 byte packets, as large packets tend to saturate the NICs and mask the differences between solutions.

An important question is whether we can claim generality from our results: would they apply if a different software architecture were used? Essentially this is asking whether the use of Click is artif cially limiting performance, so that a different architecture would not suffer the same limitation.

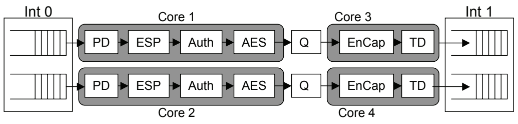
We found one case where this was true, and address it in Section 5.3. In all other cases we show that the dominating effects are due to memory accesses, and this appears to be fundamental; any forwarding path architecture performing the same task will need similar memory accesses, so the question then becomes how to map packet processing functionality to cores, irrespective of what software architecture is used to perform this mapping.

4. FORWARDING PATH ALLOCATION

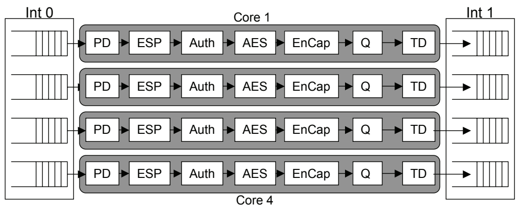
Sharing packet forwarding between multiple CPU cores allows us to forward more packets or to increase the amount of processing done per packet. There are essentially two



(a) Core-pipelined, even in/out task split



(b) Core-pipelined, uneven in/out task split



(c) Parallel forwarding paths, each run on one core.

Figure 2: Computation-intensive allocation scenarios

ways to allocate forwarding functionality to cores:

1. Split the forwarding path into several tasks (connected by queues) each assigned to a different CPU core.

2. Split the traff c (using hardware multi-queues) between multiple parallel forwarding paths, each running on its own core.

To examine task distribution, we consider a computation-intensive IPSec scenario. We instantiate the IPSec forwarding path on four CPU cores in different ways shown in Fig. 2[2].

These three scenarios give throughputs of approximately 1.1, 0.9 and 1.6 Mpps respectively. Even for a CPU-intensive task, we can see that having a packet switch cores results in a signif cant performance penalty. Splitting the forwarding path horizontally is also very prone to a few cores becoming a bottleneck if the split cannot be uniformly distributed across the cores (E.g. Fig. 2b). Such a pipeline bottleneck causes the other cores of the pipeline to stall, wasting precious computing power. It is rarely easy to balance a single pipeline across several cores.

We get similar results with pure IP forwarding: having a packet switch cores should be avoided. For the rest of this paper we treat this as given, and examine the remaining issues to be addressed to get good performance while keeping a packet on a single core.

5. INPUT PROCESSING

The very f rst element of any Click forwarding path is responsible for fetching packets from a NIC and is called a PollDevice (PD).

Until recently, the input processing architecture was rather straightforward: there was a strict one-to-one association between an interface and a PD. However, with the advent

[2]These graphs have been simplif ed to ease the presentation by removing low-impact elements.

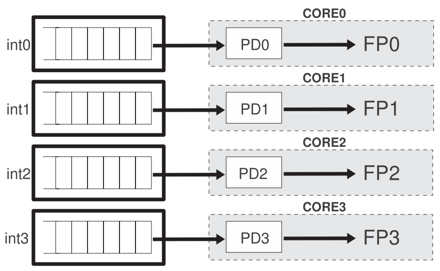


Figure 3: A simple software router strategy for input processing (four forwarding paths shown).

of hardware multi-queuing, new possibilities have emerged which we explore below.

5.1 Simple Input Approach

Figure 3 shows the original input architecture used by a default Click conf guration. On a multi-core system, to increase aggregate input bandwidth each interface can be "bound" to a specif c core by assigning the corresponding PD to the Click thread with aff nity to that core. Without hardware multi-queuing, it is hard to use multiple cores in parallel to handle the input task for an interface because this would result in unacceptable packet re-ordering.

This simple approach has two consequences from a performance point-of-view. First, the maximum processing power that can be dedicated to processing an input in a software router is one core. Indeed, using a single core to fetch 64 byte packets from a 10 GbE interface we obtained 6.97Mpps. This is only 49% of the nominal bandwidth. Performing additional per-packet processing on the core beyond this fetch operation will reduce performance further.

A further downside of this approach arises when traff c load on different interfaces is unequal. In this common case some input tasks will be idling, polling empty queues and wasting cycles on the associated cores.

5.2 A More Flexible Approach

To resolve these issues, we use hardware multi-queuing on modern NICs. This splits an interface into multiple "sub-interfaces", each consisting of a hardware queue mapped into the OS as its own device that can be "bound" to a CPU core. Combined with Receiver-Side Scaling (RSS) to load balance incoming f ows onto these queues within the NIC, this allows more f exible input processing allocation.

Using multi-queuing, we can recycle unused input processing power by automatically reassigning it to busier interfaces. To do so, we bind a set of interfaces to a set of cores as illustrated in Fig. 4. Each physical interface is split into as many hardware queues as there are cores bound to the set of interfaces. To poll these queues, each core runs one input task per sub-interface. Each core then fetches packets from all the interfaces in turn, simply "skipping" an empty queue. As RSS hashes on a per-f ow basis, the problem of packet reordering within f ows is eliminated.

We tested this scenario with small packets and "discarding" PDs, and measured a rate of 6.25Mpps per interface

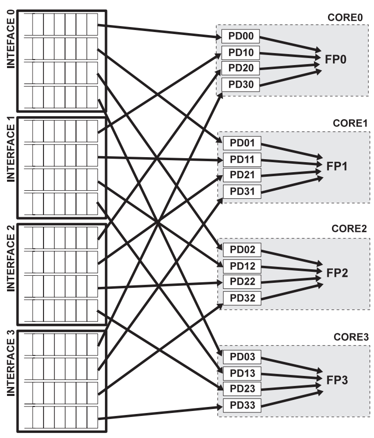


Figure 4: A strategy using hardware multi-queuing.

when all the interfaces were fully loaded using 4 cores and 4 NICs. This is actually a reduction in the per-core forwarding performance compared to the naive approach in section 5.1.

That there is no benef t is not surprising, as the aim of this approach is to allow automatic redirection of unused input processing cycles amongst a set of interfaces: because each core polls several interfaces it is less likely to run out of data to feed its forwarding paths. In the experiment above, there are no underloaded interfaces, hence no benef t. Indeed, we also observed that 3 cores could achieve a line rate of 10Gbps with 64 byte packets on a single interface using 3 hardware queues.

5.3 Reducing Context Switch Costs

The input architecture above suffers from a subtle performance issue. Each core hosts several input tasks; when these get scheduled on an empty hardware queue, they not only waste CPU cycles while checking the queue, but they also incur a Click context switch in exchange for no useful work. This is not fundamental, but is purely related to the default Click model of mapping one poll device to each interface.

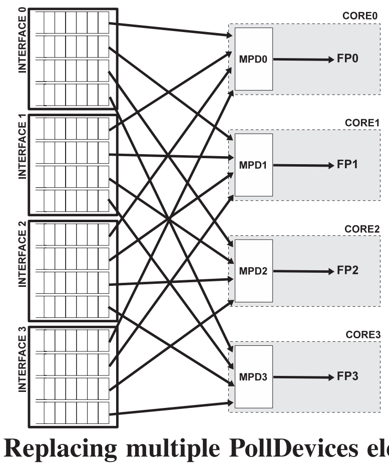


Figure 5: Replacing multiple PollDevices elements with a single MultiplePolldevice element.

To amortize the overhead of accessing ring buffers on the NIC, a Click PD reads a burst of packets, necessary for high performance. However, in the Fig. 4 conf guration, should any hardware queue have fewer packets than the allowed burst, the corresponding PD is limited to the number of packets in that queue. This sharing reduces eff ciency, as the polling overhead is then "paid" for fewer packets.

To mitigate these effects, a number of hardware queues can be assigned to a single element (Fig. 5). This reduces the likelihood that an element will process no packets, since it has more queues to try to poll packets from. Where a polled queue has fewer packets than the burst limit, the element can then poll the next queue to make up the burst. Polling several queues in succession does incur several times the overhead related to accessing the hardware, but these are incurred within a single element context switch, thus reducing the overall cost. In practice, when fully loaded, the input queues build to a reasonable size anyway.

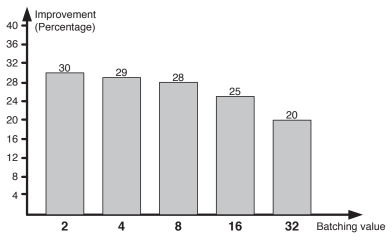


Figure 6: Improvement with four interfaces polled by each core

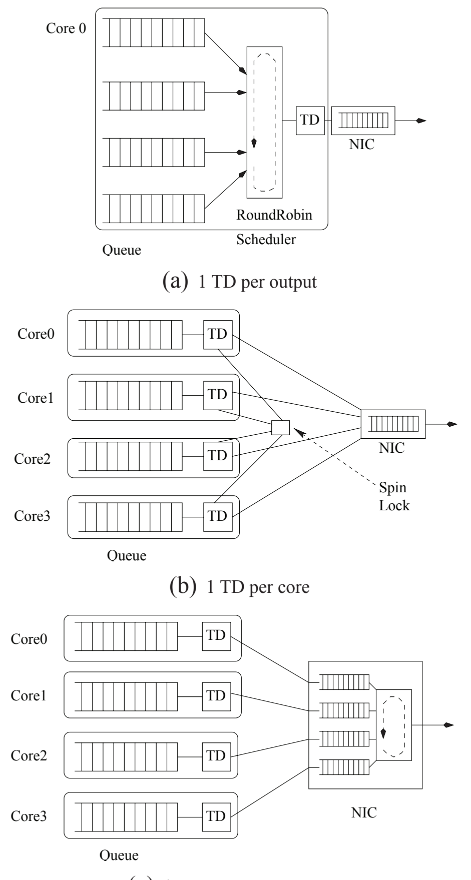
To quantify these effects, we implemented an extended PollDevice element that polls multiple hardware queues on different NICs within a single scheduling cycle. Figure 6 shows the increase in packet rate of the conf guration with the extended PD element polling 4 queues on 4 separate interfaces (1 queue per interface), as in Fig. 5, compared to the conf guration in Fig. 4. This approach performs between 20% and 30% better, depending on the batching value for this four interface scenario. With more interfaces we would expect to have an even bigger improvement, as both the reduction in schedulable elements and context-switches per packet would be reduced. Finally we can extract the performance the underlying hardware is capable of.

6. OUTPUT PROCESSING

The schedulable element of a Click output task is called a ToDevice (TD). This is usually the last element of a forwarding path and its basic job is to transmit packets to an output interface. The performance of a multi-core router depends critically on how the cores divide up this task.

6.1 Output Processing Architectures

Before the era of hardware multi-queuing, there were two basic ways to do output processing. The simplest (Fig. 7(a)) uses one TD to handle an interface. Typically a round-robin scheduler multiplexes the various Click queues associated with the TD onto it. In a typical router, these queues will come from different incoming interfaces. While simple, this output architecture requires that packets often switch cores at least once to reach the TD, since the corresponding PDs may be assigned to different cores.



(a) 1 TD per output

(b) 1 TD per core

(c) 1 TD per core per queue

Figure 7: Uncoordinated output processing

Several TDs can also share an output interface, as shown in Fig. 7(b). This approach avoids the need for packets to switch cores, and also allows the processing power dedicated to an interface to be increased by allocating several cores to it. However, it also requires low level locking to coordinate access to the interface hardware from several elements.

Finally, with the advent of hardware multi-queuing, several cores can access one output interface without the need for software locking by binding each corresponding TD to one of the interface's multiple hardware output queues (Fig. 7(c)).

To compare these alternatives, we ran an experiment with increasing numbers of simple forwarding paths containing just an input task, an output task, and a Click queue connecting the two. For the scenario in Fig. 7(a), the single output task, shared by all FPs, is assigned to a core on CPU1. The input tasks are allocated to free cores, starting with those on CPU1. For the other two scenarios, the full simple FPs are allocated to single cores, again populating CPU1 f rst.

The aggregated throughput is shown in Fig. 8. Clearly a single TD per output port yields the worst performance. This is to be expected: since a single output task is servicing all FPs, the core handling the task cannot keep up and so this scenario is CPU-limited. When only one FP is used, however, we see that this conf guration still performs slightly worse than the two others. This is caused by the packets switching cores, albeit through the shared L3 cache.

In the second scenario (Fig. 7(b)) aggregate performance decreases signif cantly when the f fth FP is added. This is because, up until that point, the lock used for synchronization was only accessed from CPU0, and thus its value stayed fresh in the L3 cache of that CPU. But with the addition of the f fth FP, FPs are now running on both CPUs, and this causes the cached value of the lock to be invalidated whenever a TD on the other CPU changes its value, forcing main memory accesses and slowing all FPs down.

From these results, it is clear that hardware multi-queuing affords vastly superior performance for output processing by enabling a true parallel output architecture. However, it should be noted that once a packet has been placed in a hardware queue on an interface, the card's hardware will transmit it on the link using the NIC's internal policy (e.g., simple round-robin across the hardware queues). Once the packet is in the interface buffer, it is out of the control of the software router. While this might be perfectly f ne sometimes, there are cases where a software router requires more advanced traff c control at the output; we study the implications of this in the next section.

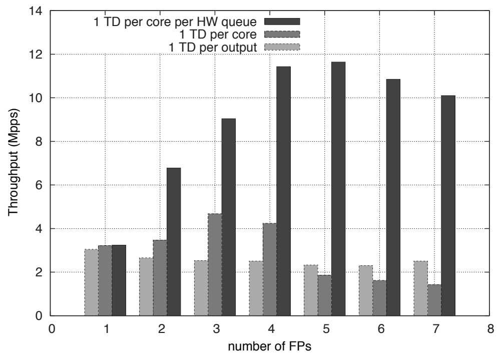


Figure 8: Performance of basic output processing

6.2 Coordinated Output Processing

As we have seen, hardware multi-queuing is great at supporting FP parallelization. However, if the NIC does not support features such as traff c management (e.g., bandwidth regulation) or advanced scheduling policies (e.g., weighted fair queuing), these must be handled in software, requiring software synchronization mechanisms (e.g., locks) even when HW multi-queue is used (see Fig. 9).

To assess the impact of such extra mechanisms, we implemented an interface access control mechanism based on a simple token bucket element. The bucket has a depth equal

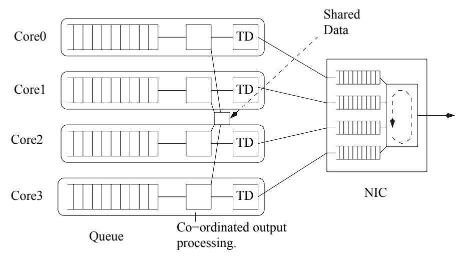


Figure 9: Coordinated output processing (1 TD per core)

to the greatest burst allowed through an interface, and f lls at a rate equal to the long term average throughput allowed. The token bucket element is designed to be inserted in output tasks in-between the Click queue and the TD. These elements must acquire a lock to access the shared data structure that represents the state of the bucket. We examined two strategies for acquiring tokens from the bucket: one token at a time ("1-token") or multiple tokens. In the latter case, the number of tokens acquired is equal to the minimum of the number of tokens in the bucket and the number of packets waiting in the Click queue ("min(N,Q)"). In both strategies, TDs send a packet for each acquired token *after* releasing the lock.

For our experiments we set the depth of the token bucket to 100 and the f lling rate to 20 Mpps (just over line rate); the results are shown in Fig. 10. First, we observe that in both cases the aggregate throughput achieved is lower than that achieved in the scenario of Fig. 7(c) where access to the hardware queues was uncoordinated; this is caused by the use of a locking primitive. However, both token bucket scenarios achieve better performance than the scenario of Fig. 7(b) that also uses locks for synchronization.

In the 1-token scenario, each output task must acquire the lock before sending each packet, just as in scenario 7(b). It might therefore seem strange that the 1-token scenario performs signif cantly better. The reason for this is that, in all token bucket cases, all packets are sent *outside of the lock*, and as the output tasks are bound to different hardware queues on the interface, more parallelism is achieved than in the case of 7(b) where the lock imposes strict serial access to the NIC.

Finally, the reason why the min(N,Q) scenarios perform better than the 1-token when there are up to 3 FPs is because the lock is kept in CPU1's cache hierarchy at all times and because lock contention is reduced by the fact that TDs acquire multiple tokens at once, while still exploiting the parallelism afforded by hardware multi-queuing. However, with FPs on both CPUs, the performance is limited by cache invalidation on the lock.

While locks clearly have their costs, other experiments (not presented here due to space constraints) show that it is cheaper to use a lock than to split forwarding paths across CPU cores. In all cases, we see that locking on a lock that stays fresh in the local cache at all times is much cheaper than locking across cache hierarchies.

7. CONCLUSION

To fully exploit the substantial computational capacity of recent multi-core server architectures for software routers, a suitable forwarding architecture is needed that is capable of high-speed packet forwarding. We have shown the crucial role hardware multi-queueing plays in enabling high degrees of parallelism, f exibility and performance. However, we have also demonstrated that hardware multi-queueing does not completely eliminate the need for task synchronization in software routers, especially in the case where coordinated

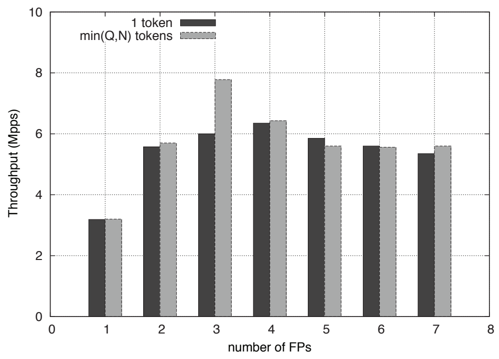


Figure 10: Performance w/ coordinated output processing

output processing amongst the hardware queues is required.

Replication of full forwarding paths on several CPU cores usually offers the best allocation strategy, primarily because it is better at using spare CPU cycles through higher parallelization. Nevertheless, when the forwarding paths are numerous and complex, hybrid solutions using lower levels of parallelization may be preferable. This points to the need for identifying metrics to quantify the computing and performance requirement of forwarding paths to use in more f exible allocation schemes. Our work also conf rms that the prime hardware unit to consider for all data touched by forwarding paths (including system features such as locks) is the cache hierarchy.

Finally, we note that commodity-hardware architectures have evolved at a very high pace recently, and we believe this will continue for some time ahead. In the future, NICs could become integrated with CPUs, adding a new dimension to the already complex problem of forwarding path allocation and aff nity. We hope the work presented in this paper gives important directions towards building high-performance software router platforms.

8. REFERENCES

[1] N. Egi, A. Greenhalgh, M. Handley, M. Hoerdt, F. Huici, and L. Mathy, "Towards high performance virtual routers on commodity hardware," in *Proceedings of ACM CoNEXT 2008*, Madrid, Spain, December 2008.

[2] M. Dobrescu, N. Egi, K. Argyraki, B.-G. Chun, K. Fall, G. Iannaccone, A. Knies, M. Manesh, and S. Ratnasamy, "Routebricks: Exploiting parallelism to scale software routers," in *Proceedings of USENIX SOSP 2009*, Big Sky, MT, USA, October 2009.

[3] R. Bolla and R. Bruschi, "Pc-based software routers: High performance and application service support," in *Proceedings of PRESTO'08*, Seattle, USA, August 2008.

[4] B. Anwer, N. Feamster, A. Nayak, and L. Liu, "Network i/o fairness in virtual machines," in *Proceedings of ACM SIGCOMM VISA 2010*, New Delhi, India, September 2010.

[5] S. Han, K. Jang, K. Park, and S. Moon, "Packetshader: a gpu-accelerated software router," in *Proceedings of ACM SIGCOMM 2010*, New Delhi, India, September 2010.

[6] E. Kohler, R. Morris, B. Chen, J. Jahnotti, and M. F. Kasshoek, "The click modular router," *ACM Transaction on Computer Systems*, vol. 18, no. 3, pp. 263–297, 2000.

[7] N. Egi, A. Greenhalgh, M. Handley, M. Hoerdt, F. Huici, and L. Mathy, "Fairness issues in software virtual routers," in *Proceedings of PRESTO'08*, Seattle, USA, August 2008.