

Improving Local Processing of Network Data With NIC on-chip caching

Karl Naden
kbn@cs.cmu.edu

Bernardo Toninho
btoninho@cs.cmu.edu

December 5, 2010

Abstract

Recent years have witnessed an incredible increase in both the raw computing power of commodity computers as well as an explosion in the network bandwidth available to them as broadband internet access becomes more prevalent. These advances have enabled data intensive applications to be organized in a distributed fashion allowing music and movies to be delivered in realtime. However, memory speeds have not keep pace with these other gains. Increasingly, end-host applications are limited in how fast they can receive and process data not by their connection speed, but by the latency of processing and moving data from the network interface to the application memory.

We propose adding NIC-level caches to keep track of where in memory packets should be delivered, allowing them to skip extra copies through OS memory. We evaluate the high-level feasibility of our proposal using a simple simulation to show that high cache-hit rates are attainable and that the hardware overhead is small. While our scheme would require changes to the operating system and applications in order to utilize the cache, we show that it can remain backwards compatible with existing OS designs. Finally, we discuss simple strategies for applications to use the cache effectively.

1 Introduction

The growth of available bandwidth on the internet in recent years has allowed the development of data-intensive services such as streaming video. On the host side, consumers also have much more processing power and memory space on their machines allowing them to handle this large amount of data reasonably well. However, while processing speeds have seen marked increases as predicted by Moore's Law, memory speeds have not kept up. Because network data is typically handled by the OS before being sent to the target application, network data is copied to and from memory several times, increasing the latency of communications and consuming precious CPU resources.

We propose a general-purpose solution to reduce the impact of this latency. The key idea is to allow applications to pre-allocate memory space for packets it knows are coming and let the NIC store packets from that flow directly into application memory. At the NIC level, we propose a hardware cache to identify flows which can use this bypass. The primary contributions of this paper are

as follows:

- The design of a pre-allocation and flow caching system involving the NIC, OS, and application.
- A high-level simulation of cache hits and cache size for the system
- A discussion of the impacts on the OS and applications developed to use the caching NIC

The rest of this paper is as follows: Section 2 explains existing instances of this problem, the solutions they have used, and why they have not caught on for general-purpose use. In Section 3, we specify the details of our system and rationale for some of our design decisions. In Section 4, we explain our simulation and the results. Section 5 contains a discussion of the impact of our system on operating systems and applications. Section 6 outlines ideas for future work and Section 7 concludes.

2 Existing Approaches

The problem of memory latency on incoming network traffic is not new. High-throughput and large scientific systems where data transport has been high for many years have already had to deal with it. The approach that was developed for them is called Remote Direct Memory Access (RDMA). In order to bypass the operating system and prevent excess transfers in memory, RDMA developed a protocol which directly connected the memory of two separate hosts, allowing each to read or write from a section of memory on the other machine without involving the OS[2]. This scheme worked reasonably well and is still used and deployed in high performance systems and large data centers as a part of the InfiniBand architecture ¹.

However, there are several reason why RDMA has not caught on for general-purpose internet users. First, the protocol is complicated and requires an additional level of explicit setup between clients. Second, because the protocol is built off of TCP, to truly avoid the OS, TCP offload is needed. TCP offload involves special hardware on the NIC which can do the TCP protocol processing, verification, and management typically done by the operating system. In practice, this method has proved expensive and difficult to keep current since hardware encodings of algorithms can only be replaced, not updated. Thus, while

¹<http://www.infinibandta.org/>

useful, RDMA is not an appropriate solution for our problem. Third, in many cases, bulk data transfers do not use TCP because reliability is not needed. As RDMA is built on top of TCP, the extra memory copying overhead for these flows cannot be prevented by RDMA.

Another interesting proposal on this subject is that of zero-copy TCP implementations [3]. Zero-copy TCP aims to improve the way TCP is implemented by the OS by copying packets directly from the application memory to the lower levels of the protocol stack. While there exist several implementations of zero-copy TCP, most of them perform the optimization through sophisticated memory management mechanisms that remap user level memory pages to kernel level pages, so that these can be directly used by the protocol stack without additional copying from the user space to the kernel space. Our approach is somewhat different, in the sense that it does not require such deep changes to the OS implementation. Furthermore, our architecture is mostly focused on improving the processing of inbound packets, regardless of the transport protocol, while zero-copy TCP focuses solely on TCP streams (from what we could gather, some implementations implement zero-copy only for sends, others for both send and receive).

3 Our Design

In the design of our system, we had three primary goals:

1. **Ease of Use:** require few changes to applications and the OS
2. **Flexibility:** not tied to single protocol
3. **Low cost:** keep the hardware component as simple as possible

3.1 System Overview

To achieve these goals, we propose a simple hardware cache located on the NIC itself which can be configured by applications via an API implemented by the operating system. Figure 1 shows an overview of the system.

At a high level, the flow of a packet through our system is as follows:

- 1) Packet arrives at the NIC
- 2) The packet is partially decoded and checked against the cache
- 3) *HIT*: If the packet hits in the cache, then
 - a) the payload is copied directly to the memory specified by the cache line
 - b) Subsequently, the packet header is sent to the OS
- 3) *MISS*: If the packet misses in the cache, then the whole packet is sent to the OS as normal
- 4) The OS performs standard processing on the packet header (e.g. TCP)

- 5) The application calls into the OS to receive the next available packet.
 - a) cache hits return a special return code to indicate that the data is in pre-allocated space
 - b) cache misses return the data copied into an application buffer as normal
- 6) The application sends any necessary cache updates to the NIC via a separate OS API

We will now give the details of the three important pieces of the system: the application, the OS, and the NIC. We start by explaining how the application will use the cache. Then motivate the information the application needs to provide in order to set up the cache by providing details of the cache itself. Finally, we use these requirements to determine how the OS can act as the middleman.

3.2 Applications

In order to motivate the design of the OS interface and the cache we first explain our model of how applications will use the cache. The key observation is that applications should know when they will be expecting to receive a set of large, data-loaded packets and be able to pre-allocate memory for these packets to be stored into.

The first question is how applications should allocate memory. In order to keep things simple, our model assumes that the applications are aware of the maximum size, N , of the payload of a packet. Now, instead of allocating a buffer of size N and passing it to the operating system via the `recv()` function, it instead allocates a buffer of size $m \cdot N$ where m is the number of packets that the application expects to receive (Note that the question of how large this buffer can be is discussed in section 4).

Once the buffer has been allocated, the application needs to notify the NIC of the buffer and the flow that can be copied there. It will do so through an API provided by the OS (described below).

Finally, the application needs to be notified that data has been placed into the buffer. For this, we use the standard `recv()` call into OS. Potential ways that the OS could signal that data is in a particular buffer are discussed below.

3.3 NIC design

The primary change to the NIC is the addition of the cache. The first question is what information is used to match in the cache and second, what information is needed to carry out the direct copy when a packet hits in the cache.

3.3.1 Cache Tags

Our system associates flows with tuples of IP source address, transport source port, and transport destination port. We chose these because they represent a group of packets coming from a single source, likely for a single

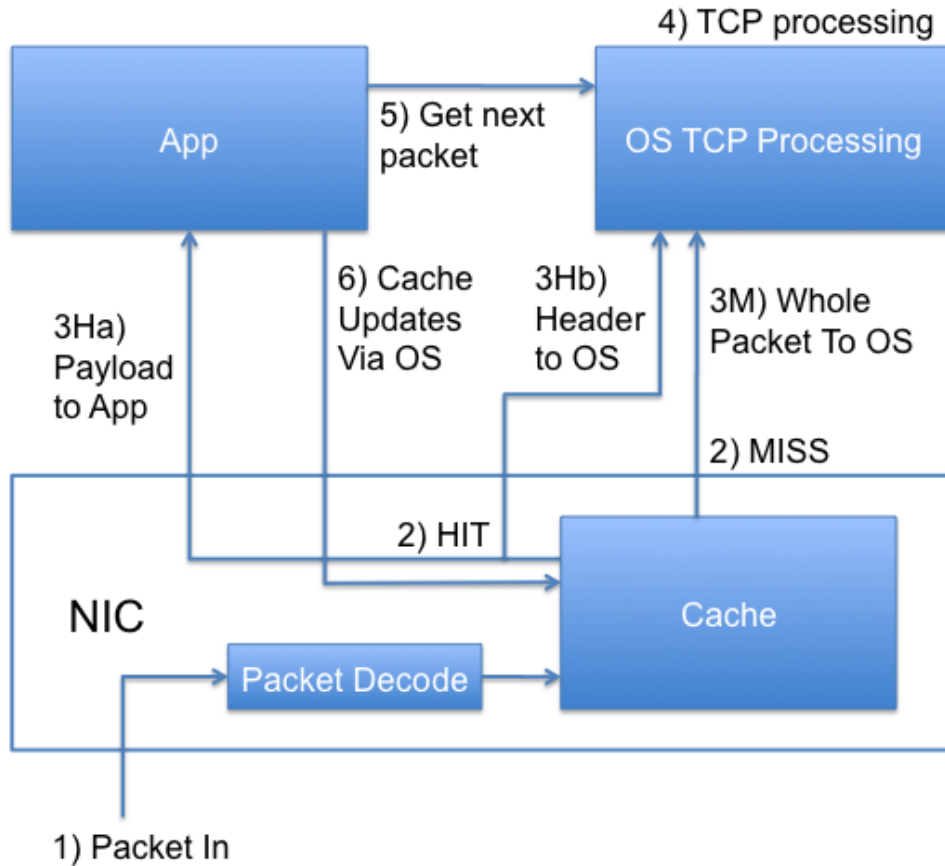


Figure 1: System Overview

purpose. Thus, the application could predict that these packets were coming and set up a cache line for them. It also simplifies our cache structure. By using the concatenation of these values as the cache key, we can create a static and unique key, allowing us to use standard SRAM for our cache instead of the more expensive TCAM hardware needed for IP variable length matching [1]. On the other hand, our cache must be fully associative. While increases the complexity of the cache, our simulation result show that the cache can be small and thus, the cost of this associativity is reasonably low. Given that IP addresses are 4 bytes long and ports are each 2 bytes, we have a total size of the cache tags of 8 bytes. Figure 2 shows the cache tag matching logic. We chose not to match on application-level protocol data contained within packets for two reasons: on one hand, it would greatly complicate the matching procedure (which must be simple enough to be easily and efficiently implemented in hardware); also, it is not clear how to provide a generic mechanism of matching against data inside packets, given the variety of ways that protocols encode information in packets. For instance BitTorrent may encode pieces of several BitTorrent messages within a single TCP packet.

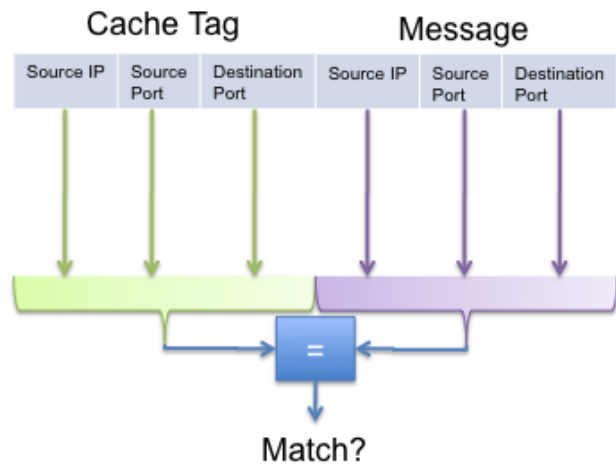


Figure 2: Cache Tag matching

3.3.2 Cache Data

If a given packet hits in the cache, we need to return information to allow the NIC to safely copy the packet's payload directly to the pre-allocated application memory. In order to do this, we need at least the memory

Since we do not have TCP offload, the operating system still handles the processing and any ACKs or other messages required for the TCP connection. It also is still responsible for determining that the packet is well formed and will not tell the application that a packet with data copied into its buffer is available if the packet is not valid. This also allows the OS to control the in order delivery of packets to the application by still controlling the notification.

3.4.2 Cache API

We leave a detailed design of the cache API for later work. However, we know that the cache will need the IP address, source port, and destination port as well as the address of the buffer, the size of each slot and the hit limit to setup a cache line. The OS would provide at least a function to set up a cache line and also one to tear one down.

3.5 Summary

In summary, our system adds a NIC-level cache in which applications can map flows to buffers and thus avoid the latency of copying the data into the OS buffers before sending it to the application. In terms of our goals, it is easy to use because there are no additional protocols to implement and the interface between the OS and application remains mostly the same. It is flexible because it is not specific to TCP or any other transport protocol. It is low cost because it can be implemented with relatively cheap hardware, does not require TCP offload, and does not require changes to the OS or applications for systems that do not utilize the caching feature.

4 Evaluation

Having given a fairly detailed description of our architecture, we now proceed to an evaluation of some of its key aspects, namely the cache hit rates and cache sizes. Cache hit rates determine how much we can actually save by employing this architecture, while estimations of cache sizes give us information about the realizability of cache in hardware.

One aspect that must be taken into account is how responsive we expect the cache to be, with respect to the addition of new cache lines. This is a factor that can greatly affect cache hit rates, since it might be the case that, between an application receiving a packet and signaling for a new cache entry and the actual cache entry being in place, several packets that would've matched that cache entry were received and therefore the opportunity for optimizing those receptions is lost.

To obtain realistic values for this *cache update latency*, we performed the following simple experiment: since a cache update mainly amounts to a system call to the OS, which then communicates with the NIC to add the new cache entry, we abstract the cache update latency by having two C processes in the same machine simply exchange packets and measuring the average time between a packet

reception and the return of the call to send (which is done immediately after the receive call). This provides a rough estimate of the time it takes for a call to go from an application, to the OS and then for control to be restored to the application (where this last component serves as an approximation of the time it would take for the OS to communicate with the NIC), which turns out to be approximately $20\mu s$. Effectively, the time we really want to measure is the time between a packet arriving on the NIC and the time it would take for the OS to notify the NIC of a cache change (going through the application). Our measurement does not account for the time it takes an inbound packet to be taken from the OS to an application, but can still serve as a reasonable lower bound.

Another fundamental aspect that needs to be taken into account is how large a buffer an application allocates for caching purposes. Buffers that are too large will increase the memory footprint of the application and likely impede several memory pages from being paged out to disk. Buffers that are too small will get filled up by the cache too fast and require adding of new cache lines too often, which is generally undesirable due to the overheads of setting up new cache entries. To account for this in our simulations in a generic way, we set a limit amount of packets that can be matched by a cache entry, thus simulating varying buffer sizes (since when the cache fills a buffer, it discards the cache line).

4.1 Simulation Infrastructure

Our simulator takes as input a Wireshark packet capture, exported as a sequence of byte arrays with timing information. At a high level, our simulator roughly performs as follows: For each packet in the sequence, the cache is consulted to determine whether or not a cache entry for that packet is present. If so, that information is recorded for statistics, if not, a new entry is added to the cache for that packet, in the hope that subsequent packets from that source will be matched.

However, to account for cache latency and buffer sizes, our mechanism to determine cache hits is slightly more sophisticated. When we add a cache entry, we record the time at which the new cache line is registered and add to it the cache update latency (which is a simulation parameter). When we look for a cache hit, if there is a cache entry for that packet information, we compare the time of the packet with the recorded time for the cache line. If the packet time is smaller, this means that a cache line was added by a previously received packet but has not yet become active (due to latency). We call this type of cache miss a *timing miss*, as opposed to a miss due to an inexistent cache entry, which we call a *compulsory miss*. Finally, to account for buffer sizes, we equip each cache entry with a hit limit (also a simulation parameter). When the number of cache hits on a particular entry reaches the limit, we are no longer allowed to match on that cache line. Subsequent match attempts on an expired cache entry are called *expiration misses*. Expiration misses model the fact that cache lines are deleted once buffers are filled.

Finally, we also implemented some packet filtering mechanisms as part of our simulator. In particular, we implemented a clustering filter that discards (for caching purposes) relatively small groups of packets inbound from a given source. Since our simulator naively adds cache lines for all compulsory misses, the clustering filter is intended to model a more realistic use of the cache (and more accurate cache size predictions). In practice, applications will not want to register cache lines for every incoming packet, reserving cache entries for more data intensive packet streams (e.g. data transfers). The role of our clustering filter is to single out these more data intensive time intervals, and only allow for cache lines to be added for these packet streams.

Using our simulator, we mainly evaluated cache hit rates and cache sizes. We accessed what we determined to be the *idealized* hit rates (with no account for hit limits) with various packet capture data from BitTorrent, Skype and Web traffic. We then measured cache hit rates for realistic buffer sizes of 10, 20 and 30 packets. For cache sizes, we determined the worst case estimate for cache size (adding cache lines for every incoming packet) and more realistic values for cache sizes, using our clustering packet filter. For our simulations we used a cache update latency of $30\mu s$, an over-approximation of our experimental findings.

Due to memory constraints (our total capture data consists of approximately 2.6GB of packets), we had to divide our captures into several separate files. However, since each capture contained a specific usage pattern (one for BitTorrent and web traffic, another for Skype and BitTorrent, etc.), the results for the divided capture files are virtually indistinguishable from each other (for each specific traffic pattern).

4.2 Cache Hit Rates

As mentioned above, we began by determining idealized hit rates for our test data. These are presented in Fig. 4.

Our goal here was mainly to determine the significance of cache update latencies on overall performance, as well as to have an upper bound on the worst case cache sizes (which we defer for the next section). As expected, the hit rates are substantially high, with most of the cache misses being the result of compulsory misses. A few timing misses were observed throughout the data set, but they were always in a very small number.

We then proceeded to measure cache hit rates for more realistic buffer sizes, through our cache hit limit parameter. We considered buffer sizes of 10, 20 and 30 packets. The results of these simulations are presented in Fig. 5, 6 and 7, respectively. As expected, we observe a decrease in hit rates due to an increase of expiration misses and timing misses that result from the adding of new cache lines. Note that even for a relatively small buffer size of 10 packets, we still obtain fairly high hit rates (approx. 89%). For larger buffer sizes, we obtain results that are very close to the idealized hit rates (94% and 95% versus 98%).

4.3 Cache Size

As with the evaluation of cache hit rates, we begin by determining the worst-case cache size for several types of packet traffic. These are obtained by naively adding a cache line after every compulsory miss, and are presented in Fig. 8. We can easily observe that, even in the worst case scenario, the number of cache lines end up not being prohibitively high, needing at most 2.2% of the total number of packets as cache lines. Considering each cache line is relatively small, this supports our claim that the cache could be reasonably implemented in hardware, even in the worst case.

We now show what happens in a more reasonable use of our cache in the same circumstances (Fig. 9). By only adding cache lines for reasonable sized contiguous flows (which is an approximation of how applications would actually use our caching mechanism), we can see that the actual amount of cache lines is fairly insignificant, furthering our claims of a reasonable hardware implementation of the cache. Note that the amount of packets that are filtered out is fairly small, so our results are not due to simply eliminating a substantial amount of the packet stream. We also show the cache hit rates obtained in this scenario (considering a 30 packet buffer). As is to be expected, the hit rates are slightly lower due to discarded packets, but they remain substantially high (above 90%). This shows that in more realistic cache utilizations, we still obtain a good amount of cache hits and therefore are able to potentially speedup the packet reception process.

5 Discussion

In this section we discuss how our results support the viability of our proposed architecture, as well as potential improvements that could be made through increased cooperation with the operating system.

We have shown that with relatively small buffers assigned to our cache, we can obtain a high volume of cache hits. This is important because we should not expect applications to allocate overly large buffers in order for our cache to be effective. Furthermore, we have seen that with the expected cache update latency (which is supported by preliminary measurements), we end up observing low numbers of timing misses, even in the presence of small buffers that push up the total number of cache misses.

There is also a point to be made in the simplicity of our matching procedure. Since we only compare packet attributes (IP addresses and ports), matching is a fairly simple procedure that should be feasibly implemented using SRAM (we'd essentially need a table against which to match a bit string consisting of the source IP address, source port and destination port). Given the somewhat expensive nature of this type of memory, our observation that the number of cache entries is actually fairly small is of the utmost importance, as well as the fact that each entry itself is of a very small size.

Even though we do not have simulation results to back this claim, we posit that we could push cache hit rates

even higher with some additional OS support. Our current design only requires fairly minimal support from the OS. Obviously, the OS must be aware of the cache (to be able to insert new cache entries and to deal with the way matched packets are processed). However, If an application could register a cache line preemptively, specifically during the TCP three-way handshake, the initial compulsory misses would not be present, as well as potential timing misses that could be avoided given the relatively slow nature of the TCP initial setup.

We must also address the actual impact on application design that using our architecture would entail. We tried to minimize this by incorporating our design with the standard send/receive patterns of socket programming. While it is inevitable that application code would have to change to use our caching mechanism, the programming discipline itself would not change substantially. Every received packet is still signaled by the return of a receive call, with the caveat that packets that are copied directly from the NIC end up in the assigned buffer, instead of the buffer passed to the receive call. In principle, other than reading out from the caching buffer as instructed by the return of the receive calls, the only other significant change would be the actual explicit adding of cache lines (through a system call).

Finally, we address the issue of what is actually gained by our proposal. Even though we do not completely remove OS participation in packet processing (due to the implementation of TCP being at the OS level and for ease of use of the actual framework), we substantially cut down the amount of data that needs to be copied to OS buffers and then to the application memory space. Essentially, for packets that match a cache line, the payload is directly copied into application memory, with the OS only receiving packet headers (which are a very small portion of the actual packet). Assuming an efficient matching procedure and a reasonable use of the cache, the hit rates that can be obtained should easily produce performance increases (which is supported by the hit rates obtained in Fig. 9), given that most packet payloads would not be copied to OS buffers, thus substantially decreasing overheads that are the result of memory latency.

6 Future Work

Now that we have shown that our approach is potentially feasible, our future work in this area will focus on refining the idea and building towards and more realistic simulation including the new hardware and more generalized flows. These simulation will undoubtedly reveal and clarify some of the important design decisions that we had to gloss over given our high level of abstraction.

One question that will need to be addressed and better tested is the question of how to handle out of order packets. Our current architecture makes no assumptions about the ordering of packets and will insert packets into the application buffer out of order if that is how they are received. However, as this puts a higher burden on the OS

because it needs to know where in the buffer each packet it. Given that actual out of order delivery is rare, it would be interesting to evaluate our design against one that assumes in order delivery and simply treats out of order delivery as a miss in the cache. This would require some shifting of complexity from the OS to the NIC, but might result in better performance on average. As our current simulation is not complete enough to evaluate this difference, we leave it for future work.

7 Concluding Remarks

We have presented a potential new way of handling packet processing, by exploiting DMA and some speculative changes in NIC hardware. Our main goal was to design a reasonable architecture that would result in gains in terms of the number of copies that each packet goes through, while being processed up the protocol stack. We analyzed and discussed the viability of this hypothetical architecture through what we claim to be an adequate simulation environment, having obtained results that back up our hypothesis that such an architecture is both viable (in terms of actual implementation) and efficient.

References

- [1] Daekyeong Moon Scott Shenker Martin Casado, Teemu Koponen. Rethinking packet forwarding hardware, 2008.
- [2] Allyn Romanow and Steven Bailey. An overview of rdma over ip, 2003.
- [3] Karl-Andre Skevik Thomas, Thomas Plagemann, and Vera Goebel. Evaluation of a zero-copy protocol implementation. In *In 27th Euromicro Conference*, 2001.

Traffic Type	Hits	Misses	Hit Rate	Comp. Misses	Timing Misses
BitTorrent and Heavy Web	18166	289	98%	287	2
Heavy Web	16930	59	99%	59	0
Skype, BitTorrent and Web	18502	217	98%	217	0
Skype and Web	19808	108	99%	108	0

Figure 4: Idealized Cache Hit Rates

Traffic Type	Hits	Misses	Hit Rate	Comp. Misses	Timing Misses	Expiration Misses
BitTorrent and Heavy Web	16468	1987	89%	287	76	1624
Heavy Web	15390	1599	90%	59	13	1527
Skype, BitTorrent and Web	16827	1892	89%	217	7	1668
Skype and Web	17820	2096	89%	108	241	1747

Figure 5: Cache Hit Rates - 10 Packet Buffer

Traffic Type	Hits	Misses	Hit Rate	Comp. Misses	Timing Misses	Expiration Misses
BitTorrent and Heavy Web	17293	1162	93%	287	31	844
Heavy Web	16126	863	94%	59	5	799
Skype, BitTorrent and Web	17625	1094	94%	217	6	871
Skype and Web	18746	1170	94%	108	154	908

Figure 6: Cache Hit Rates - 20 Packet Buffer

Traffic Type	Hits	Misses	Hit Rate	Comp. Misses	Timing Misses	Expiration Misses
BitTorrent and Heavy Web	17575	880	95%	287	24	569
Heavy Web	16388	601	96%	59	3	539
Skype, BitTorrent and Web	17909	810	95%	217	6	587
Skype and Web	19079	837	95%	108	115	614

Figure 7: Cache Hit Rates - 30 Packet Buffer

Traffic Type	Packet Total	Cache Size	Percentage
BitTorrent and Heavy Web	17101	302	1.7%
Heavy Web	16845	114	0.6%
Skype, BitTorrent and Web	17536	391	2.2%
Skype and Web	17867	116	0.6%

Figure 8: Cache Sizes - Worst Case

Traffic Type	Packet Total	Filtered Packets	Cache Size	Percentage	Hit Rate
BitTorrent and Heavy Web	17101	957	89	0.5%	91%
Heavy Web	16845	130	99	0.5%	95.5%
Skype, BitTorrent and Web	17536	875	107	0.6%	91.4%
Skype and Web	17867	88	75	0.4%	95.7%

Figure 9: Cache Sizes - Flow Filtering