CacheCard: caching static and dynamic content on the NIC

Herbert Bos
Vrije Universiteit Amsterdam
herbertb@cs.vu.nl

Kaiming Huang
Xiamen University, China
kmhuang@xiamen.cn

ABSTRACT

CacheCard is a NIC-based cache for static and dynamic web content in a way that allows for implementation on simple devices like NICs. It requires neither understanding of the way dynamic data is generated, nor execution of scripts on the cache. By monitoring file system activity and potential non-determinism incurred by scripts, we determine all data sources for specific requests. For instance, if a deterministic script opens a set of files or a database tables, these files and tables, as well as the script itself will be in the set of data sources for this URL. Caching the dynamic data is possible, since we can invalidate cache entries when any of the sources changes. Non-deterministic scripts that produce content based on time or random values are automatically recognised and flagged as non-cacheable. We implemented *CacheCard* on Intel IXP2400 network processors.

1. INTRODUCTION

Popular file or web servers are commonly offloaded by *caching* on dedicated machines. Examples include well-known web proxy caches such as Squid [28] and Harvest [6]. Wikipedia, for instance, employs tens of Squid caches for each of its data centers and serves over 80% of the requests from the caches [4,5]. Caching is quite beneficial as page updates are rare (0.03%) compared to page reads, while uploads of binary files are even less common (0.002%) [24]. Locality in web server traces tends to be high [3]. Similarly, Nache [10] and companies like Gear6 and others provide proxy caches for network file servers. To alleviate pressure on the server and provide rapid response times, they commonly place a fast caching appliance in the network.

Proxy-caching as provided by Squid or Gear6 has several drawbacks. First, additional machines imply additional power, cooling, rack space, and management, which together represent most of the total cost of ownership.

Second, it is extremely difficult to combine stand-alone proxy caches with dynamic web content. When data is dynamically generated, for instance by a PHP script, data should be purged from the cache as soon as new values are generated. But the cache does not know when this happens unless (a) the dynamic behavior can be replicated at the proxy, (b) the cache is explicitly notified of all changes, or (c) the cache checks each time whether the content has changed. Currently available solutions are complex and expensive, often beyond the reach of small and medium-sized organizations. However, since these organizations also increasingly generate web content by means of server-side scripts, the problem is urgent.

Third, server capacity is difficult to upgrade without replacing the server. A modular solution, where capacity can be extended in a simple manner (without requiring additional cooling or rackspace) is desirable.

CacheCard. In this paper, we solve all three problems by (a) pushing a cache for file/web content to the network card (Figure 1), so that (b) server capacity can be increased simply by plugging in more cards, each of which is (c) sufficiently integrated with the server to handle dynamic data, (d) without having to change the OS or the server.

The system, known as *CacheCard*, minimizes copying and synchronization both over the PCI bus using techniques originally developed for very high-speed network monitoring [9]. The limited existing work on caching on NICs [7,14], while inspirational, do not handle dynamic content at all *and* require a radical rewrite of the server's OS and/or server software. Unless such changes are pushed upstream, we believe that neither of these approaches is very practical. In contrast, we do not even require access to the existing OS or server source code. We do wrap some of the libc library calls in the server (for instance, to make sure that an entry is purged from the cache on the card when the file content changes), but this is achieved by library interposition and requires no access to the source. Moreover, plugging in a *CacheCard* NIC faithfully provides the illusion of a single (but now much more powerful) server which means that clients *never* see any stale data whatsoever. To the best of our knowledge, we are the first to support caching of *dynamic* data on a network card.

To make *CacheCard* possible, we needed to overcome several hurdles. For instance, having a modest amount of memory on the network card and a much larger amount of active data presents a replacement problem. Most caches use LRU as replacement strategy, but LRU performs poorly when the locality distance (the time between two accesses to the same data) is larger than the length of the LRU stack [22, 32]. Briefly, it leads to frequent eviction of data before it can be used by the next read. For this reason, our replacement strategy is based on an estimate of popularity over time.

Also, to cache dynamically generated data without significant modification to the server or OS, we employ a novel technique to detect which sources (files, time of day, randomness, etc.) make up the reply for a specific request. They form the request's *invalidation set*: if any source in the set changes, the cache entry for the request will be invalidated. The technique to determine invalidation sets is entirely new.

Contributions. In summary, our contributions are:

1. *Transparent proxy cache.* A low-power (15.5W) NIC-based cache that requires no access to OS or server source code.

2. *Modular performance.* Increasing a server's capacity is as simple as plugging in an extra *CacheCard*.

3. *Dynamic data.* A novel way to determine the files used in dynamic content allows us to cache such data.

4. *Hysteresis-based replacement.* A replacement algorithm to handle large locality distances by explicitly weighing in access patterns in the 'distant' past.

By plugging the network cards directly into the server's peripheral bus, we combine the advantages of existing proxy caches with those of server upgrades. While reducing the load on bus, memory, and secondary storage, we also retain the proximity of the cache to the original content. Modern 1U rack-mounted servers often have 2-4 PCI slots, 2U and 3U servers even more. Saving space by using these slots rather than adding new boxes is a big win.

The main thing that we present in this paper is a practical device built on several novel techniques to implement caching. *CacheCard* improves the performance of web servers for most content, including dynamically generated web pages. Moreover, even content that does not lend itself to caching (e.g., a PHP script that displays the current time) is handled correctly; *CacheCard* automatically discovers that it should not be cached and skips it.

Concretely, *CacheCard* functions as a proxy cache for a full **LAMP** (Linux, Apache, MySQL, and PHP) server. The various servers (database and web) may or may not be located on the same machine. *CacheCard* is implemented on a programmable NIC equipped with a 7-year old Intel IXP2400 network processor which has on-chip a single XScale control processor, 8 specialized cores and a modest amount of memory. A single *CacheCard* card improves our apache web server's reply rate by up to a factor four, even with our old hardware.

Outline. We discuss the design and implementation of *CacheCard* in Section 2. We evaluate performance in Section 3. Section 4 places our work in the context of related projects. Conclusions are in Section 5.

2. CACHECARD FOR WEB TRAFFIC

Without caching, HTTP GET request packets arriving at a network interface are DMAed to host memory in order to be reassembled by the CPU. If the content is not in main memory already, it will be read from disk. The server creates the appropriate HTTP headers and hands control to the TCP stack which generates TCP and IP headers for the messages. After that it passes references to headers and content to the network driver. The NIC then DMAs the content and headers to its own buffers, calculates the CRC checksum and transmits.

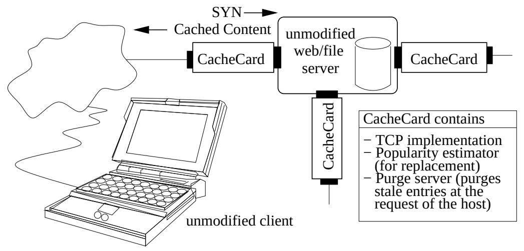

SYN
Cached Content
CacheCard
unmodified web/file server
CacheCard
CacheCard
CacheCard contains
– TCP implementation
– Popularity estimator (for replacement)
– Purge server (purges stale entries at the request of the host)
unmodified client

Figure 1: *CacheCard* **architecture**

Thus, the web request is served by the network card, the peripheral bus, the CPU, main memory, and possibly the disk. Each of these components may become a bottleneck. Caching is intended to reduce the load by minimizing the number of requests that need to be served by these resources.

An ideal cache on the network card assumes all of the above responsibilities for GET request sequences that exhibit sufficient locality. The server's role is reduced to serving cache misses and invalidating cache entries whenever the original file has changed. In the next few sections, we describe the steps we have taken to approximate an ideal cache.

2.1 Basic design

In this section, we give an overview of the basic *CacheCard* design without worrying about invalidations yet. Later sections will look at specific aspects and invalidations in detail.

An important feature of *CacheCard* is that we track the popularity of content (identified by URL) over time in a sorted list and cache the most popular requests on the network card. Our way of estimating popularity will be explained in Section 2.2. To deal with long locality distances, a cache miss does not lead to replacement unless the URL on which it occurred appears in the list of most popular URLs.

If the data is to be cached, caching occurs at the level of the network data. In other words, *CacheCard* intercepts packets transmitted by the server and stores the data in the card's memory. The next time a request for the same URL arrives, it is served by the card. To save repeated processing at runtime, partial checksums are stored with the data.

For all requests, the NIC simply forwards the three-way TCP handshake packets from the client to the original server and *vice versa*. Next, when the NIC encounters a GET request, it checks whether the requested content is present in the cache. If not, it increments the request counter (the number of requests for the URL in the current epoch) by means of an atomic increment, and forwards all remaining TCP segments to the server without touching them. We will show later, that the counter determines which file is cached.

If, on the other hand, the data is found in the cache, the NIC takes over the connection by sending a FIN packet to the original server (closing the connection) and serving the data from the cache, using the appropriate sequence and acknowledge numbers as negotiated in the handshake. It also atomically increments the request counter.

As a consequence, some persistent connections may have to be treated separately. Persistent connections are TCP connections that carry more than one HTTP request. If in a single connection a request that is satisfied out of the cache is followed by another one that misses in the cache, the second request is treated as a special case that incurs the cost of a new connection establishment from the NIC to the server. For transparency, replies must then be proxied to the client by *CacheCard* in the original connection. Fortunately, doing so is a lightweight operation, comparable to network address translation. More importantly, since request sequences tend to exhibit significant locality, in the normal case most objects requested in a connection will be in the cache together.

To minimise the card's work, we optimise checksumming. *CacheCard* must construct the right packet header, link it to the data, and generate the corresponding checksums. As full IP and TCP checksum calculations are fairly expensive, we cache a partial checksum for each segment of data. In most cases only header checksums are calculated.

2.2 Popularity and replacement

Cache replacement is a 'hot' research area, especially in multi-level caching, where LRU is particularly problematic for goals like exclusive caching [31]. Our problem is different. As mentioned earlier, LRU replacement is problematic when the locality distance exceeds the LRU stack length.

Inspired by the LRFU replacement strategy [16] that combines frequency and recency, we add hysteresis to the replacement procedure, so that entries are not evicted when they have shown to be popular. In other words, we add an estimate of popularity *over time*. For explanation purposes, we assume that every request corresponds to a single file, so we can talk about a file's popularity. In reality, of course, multiple files and databases may be used for a single request.

To estimate popularity, we divide time in epochs and calculate the popularity p as a weighted moving average as follows: $p_{t+1} = (1 - \alpha)p_t + \alpha \times rcount$, where $rcount$ is the number of requests for the file during the last epoch. The value of α determines how much weight is placed on requests in the current epoch and how much on earlier epochs. A reasonable value for α can be easily found by periodically feeding an existing trace of the web server in a profiler. Manual tweaking is not needed. For our departmental web server values between $\alpha = 0.6$ and $\alpha = 0.8$ yield good results.

Given the popularity estimates, *CacheCard* calculates a ranking of the most popular files in the epoch. In our experiments, we have used an epoch of 30 seconds. Subject to available memory, *CacheCard* will try to cache the most popular files. Doing so removes some of the problems associated with LRU eviction, as a highly popular file that was not so recently accessed may still be spared eviction at the expense of a more recently accessed, but less popular object.

The NIC has multiple cores. At runtime, the core that is serving a request increments the corresponding URL's request count (using an atomic increment). It does not sort the files in order of popularity. Rather, it checks whether a file should be promoted or demoted across the boundary between popular and unpopular files, where 'unpopular' means that the popularity estimate is less than a threshold τ (initially τ is zero). At the end of an epoch, when we sort the files according to popularity, we only sort the popular files. The threshold τ is intended for regulating the number of files that need sorting. For instance, we can increase τ if the number of popular files grows beyond a high watermark, and decrease τ if it falls below a low watermark. Again, manual tweaking of τ is not needed.

To avoid locking, the cores on the NIC simply append their promotion and demotion decisions to a hardware-assisted FIFO which is read and processed by a thread on the XScale control processor, known as the 'bouncer thread'. The bouncer thread is responsible for popularity calculation and sorting. It also determines which of the popular files fit in the cache and marks each corresponding file, if not in the cache already, 'cacheable'. Next time we encounter a request for a cacheable file, we store the server's reply in memory.

2.3 Invalidations

Besides the replacement strategy mentioned above, the other challenge in caching is deciding when to stop doing it. In principle any data, static or dynamic, can be cached, but when the data changes at the source, we need to invalidate the entry. Problems are caused by changes of content that are not seen by the cache. For instance, a local write to a file or database table. The question is how do we know when the source changes and how do we notify the caches?

If the mapping between requests and source files is obvious, as is the case for static web or NFS requests, solutions are simple and diverse. A well-known technique is to ask the origin server whether the data has changed. Webproxies do so by means of an '`If-Modified-Since`' HTTP header, while centralised NAS/NFS caches typically touch base with the original file server by retrieving the file attributes (e.g., using the NFS `getattr` procedure). These operations are synchronous operations that increase the load on the server and add significant latency for the client. Moreover, for NFS, the open-to-close consistency model is quite weak and may lead to problems when other clients write to a file.

In our case, the cached data is very close to the main processor and in a webserver local writes are relatively rare, so we opted for the inverse solution. A thin wrapper around libc file operations like `open`, `close`, and `write` (and all similar calls like `fopen` and `fclose`) intercepts updates to existing files. When a file is updated, the wrapper sends a purge message to the caches. Assuming the file still appears in the top n popular files, it will soon be cached again, so that subsequent requests are served by the card. A file that is currently opened and memory mapped with write permission is marked not cacheable.

We implement our wrappers by means of library interposition so no modifications to or recompilations of the original libc library are needed. Instead, we provide a wrapper library which exports the same functions as those of libc that we want to intercept. We instruct the loader to load our library functions instead of their original libc counterparts by setting the LD_PRELOAD environment variable to point to our library. The wrappers do whatever is needed on behalf of *CacheCard* in addition to calling the original libc versions.

A file that is modified very frequently may lead to frequent purges. Although purges are cheap, they do have some overhead. More importantly, the file that is really too dynamic to cache keeps consuming space in the cache, which could have been used for a slightly less dynamic file. Although we do not do so, this is easy to determine and if the purge rate exceeds a threshold, we can mark the file 'not cacheable'.

Dynamic content and invalidation sets. Replies on the web are increasingly dynamically generated. Even if the actual content pushed to clients is mostly static, the data is gathered for instance by (PHP-like) scripts from files or a database. A simple solution is not to cache dynamic data.

However, this would be not be very satisfying. Unless the script is non-deterministic for a specific URL[1] (e.g., generating content that is random or that varies depending on the time of day), the data that is served is often sufficiently static to warrant caching and only changes when a file or database table is updated, which may be infrequent[2]. Caching dynamic data is therefore desirable, but difficult. The challenge is to decide when to stop caching, because the data has changed.

For this we need to know the sources that generate the server's reply. Unfortunately, truly dynamic content does not have a straightforward mapping between the URL and its files. For instance, a PHP script may open several different files, contact a back-end database, and so on. Nothing in the URL tells us which sources are involved.

In this paper, we refer to the set of sources involved in satisfying a GET request for a URL as that URL's *invalidation set*. Any changes to sources in the invalidation set should lead to invalidation of cached entries of the data. Examples of sources can be the files on disk, tables in a database, or sources of non-determinism.

Non-determinism is introduced in server-side scripts, for instance when they use time, random numbers, or connections to external sources beyond our control. However, any connections to other servers under *CacheCard*'s control (such as our own MySQL back-end running on a machine nearby) do not count as sources of non-determinism. Any URL with an invalidation set that does not contain sources of non-determinism can be cached.

Besides sources of non-determinism, we did not need to consider any sources other than files in our LAMP configuration. Apache and PHP are naturally file-based, but even MySQL represents its tables as files. Knowing the files in the invalidation set is sufficient to allow us to invalidate the appropriate cache entries. Other database systems with different internal formats may require tracking other sources also. We will explain how we determine the sets shortly.

We will explain how we establish invalidation sets in the next section. For now, let us assume that we have a reliable invalidation set. We then maintain a simple hash table containing each file that is present in at least one invalidation set with a per-file pointer to all invalidation sets in which the file is present. This allows us to invalidate efficiently all corresponding cache entries.

Invalidation set contraction. We developed a generic method to establish invalidation sets dynamically without knowledge of the internals of the server software (e.g., for us it works both for PHP and MySQL). For this purpose, *CacheCard* monitors file system activity as well as calls that may introduce non-determinism. Between when a request enters the system and the corresponding reply, we trackwhich files in the data directories are open(ed), which calls are made to functions like `gettimeofday()`, which database tables are opened, etc. These sources form the initial invalidation set. Again, we wrap libraries rather than syscalls, because it is cheaper.

As multiple requests are handled simultaneously, not all of the sources in the initial invalidation set correspond to its URL. For instance, some of the files may have been opened on behalf of other URLs. The other URLs and the files opened on their behalf are the context in which a particular GET request appears. The smaller the context, the more accurate the invalidation set. Note, an invalidation set that is too large is only a problem for efficiency, not for correctness. The worst that can happen is that a cache entry is invalidated when it need not have been, when an unrelated file changes.

[1]In *CacheCard*, a generalised URL includes header fields such as language specifications and all parameters passed via the URL.

[2]We saw previously that files too volatile to be cached can be detected automatically.

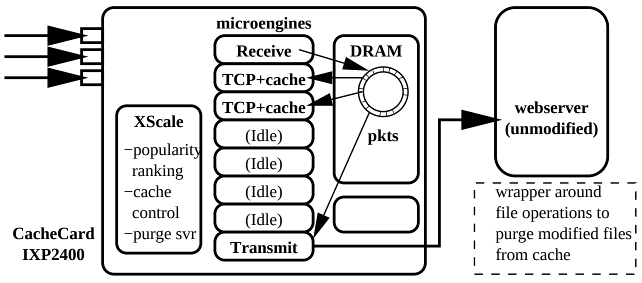


Figure 2: ***CacheCard*** **architecture**

However, if the URL is popular, it will occur in different contexts over time. We therefore employ a process known as *invalidation set contraction* which determines the minimum common subset among different estimations of the invalidation set. This way, we reduce the number of spurious sources over time until we arrive at an invalidation set that is close to the optimal invalidation set (see the evaluation in Section 3).

Because correctness is not affected by the optimality of the invalidation set, we are at liberty to limit the overhead of invalidation set contraction to an arbitrary amount, for instance 1% of the CPU time. Alternatively, we can choose not to run contraction at all unless the server load is below a certain threshold. Finally, we may 'pickle' invalidation sets that have not changed in a long time. Pickling invalidation set means that we will not change them anymore, so that there is no more overhead due to contraction that is probably not going to shrink the invalidation set anyway. None of these optimizations have been applied in our prototype where we simply (and arbitrarily) run invalidation set contraction every minute.

2.4 Implementation details

We implemented *CacheCard* on a Radisys ENP2611 board built around the Intel IXP2400 network processor (NPU), as the multi-core processor allows us to exploit parallelism to handle high-speed links at modest clock rates and low power consumption[3], and the board has generous amounts of memory (the IXP supports up to 1 GB of DRAM and 16 MB of SRAM). The IXP2400, shown in Figure 2, is a heterogeneous multi-core with an XScale control processor and 8 muti-threaded stream processors known as *micro-engines* (MEs) that run without OS. Introduced in 2002, it is by no means the fastest NPU in the IXP family. For instance, the IXP28xx series offers twice as many micro-engines and a much higher clock rate as well as other improvements. At the same time, the IXP28xx is identical in instruction set, so that all our code also runs on the 28xx. Even more impres-

[3]The entire board consumes a maximum of 15.5 Watts.

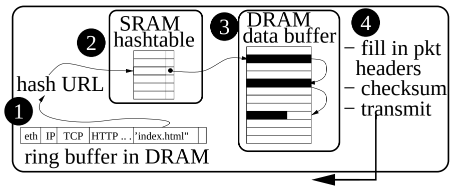


Figure 3: Cache lookup and memory organization

sive, Netronome's NFP32-xx (backward compatible with the IXP28xx), has no fewer than 40 multithreaded cores. For our project, however, we explicitly for cost-effective (low-end) processors.

The NPU offers various kinds of memory ranging from fast but small memory local to each ME, to larger and slower ones. The ME's local memory has 640 4B words, while shared on-chip Scratch space is 16KB. Off-chip SRAM is 8-16MB, and off-chip DRAM is 256MB-1GB. Asynchronous memory accesses make it possible to hide memory latency by overlapping accesses with processing (and/or more memory requests). IXPs offer hardware support for expensive computations like hashing, and CRC calculations.

2.4.1 *Mapping on memories*

The performance of the caching strategy is dependent on the way we organize memory. Figure 3 illustrates (schematically) the way in which *CacheCard* handles cache hits.

When a request packet comes in (in DRAM), we extract the URL and file name (①) and check whether the data is stored in the cache. To do so and to ensure fast access, we maintain a hash table to map the file name to a pointer to the data (②). The hash table is kept in SRAM memory, while Scratch memory is used as an explicitly managed cache for temporary data. For efficiency, we calculate the hash over the file name by means of the processor's hardware hashing unit. Next, we check whether an entry exists for that file. If so, the table yields a pointer to the first slot of data.

To avoid expensive memory allocation at runtime and external fragmentation, the memory area dedicated to the cached data itself is statically partitioned in fixed-size slots (③), each large enough (1600 bytes) to hold a maximum-size packet. Each file larger than a single packet is stored as a linked list of data slots. The slots need not be contiguous in memory. For example, in the figure, the file consists of three slots that are not adjacent. Fixed-size slots suffer from internal fragmentation (the white area in one of the slots in Figure 3), but managing the memory is greatly simplified. In particular, they exhibit no external fragmentation.

As mentioned earlier, the slots store packet data with partial IP and TCP checksum, rather than raw files. Memory for headers is preallocated and some of the fields are already filled in for the common case reply in which no additional HTTP reply headers are requested. If such a request comes in, we fill in a handful of TCP and IP fields, calculate checksums and transmit the packets (④). In case additional headers are needed, these should be added first.

2.4.2 *Mapping on micro-engines*

As shown in Figure 2, we dedicate two micro-engines (MEs) to packet reception and transmission. The number of threads on each of these MEs is chosen so as to maximize performance. ME_0 uses two threads to receive packets. The first thread polls whether packets are available. If so, the second threads stores them in a circular buffer in DRAM using slotted buffers similar to those used in Beltway Buffers [9].

ME_1 employs three threads for forwarding. The threads handle three transmission queues which are kept in DRAM. The first two of these queues contain TCP handshake messages. These are simply forwarded. Queue number 3 contains two kinds of packets: FIN packets generated by ME_0 heading to the web server, and ACK messages to acknowledge the last connection-terminating packet from the web server. These are also simply transmitted.

All other MEs are available for TCP and cache handling. In practice, however, we use only two MEs, each with 2 threads. We implemented TCP including the protocol's fairly complex dynamic part for transmissio n (acknowledgment handling, timeouts, bulk transfer, and retransmissions). For reception we employ an observation that holds true for all browser we encountered (including IE and Firefox) to avoid full TCP reassembly: new requests arrive in separate segments. This is true, even in the case of persistent connections. In practice every GET request is always sent in a separate segment.

Employing more threads increases the amount of parallelism in *CacheCard*, but at the cost of extra strain on the memory subsystem. As a result, using two MEs appears to be the sweetspot. Thus, ME_2 and ME_3 process a queue containing only headers and are responsible for matching them up with data from the cache. In addition they handle checksumming for IP and TCP and send replies to the client.

2.4.3 *Invalidation sets*

Libc interposition to determine invalidation sets is kept as lightweight as possible. When a relevant function is called (e.g., `fopen()`), an asyncronous event is sent to a daemon known as the *ISContractor*, containing operation, filename and file-descriptor (and a few other fields to cope with packet loss and synchronization). The ISContractor periodically performs invalidation set contraction and shares the resulting sets with a purger process. The purger process keeps track of file changes and sends out invalidation notifications to caches for the appropriate invalidation sets. By default, events are sent over Unix domain sockets if the communication is local and UDP if the ISContractor is on a different machine.

The simplest way to do so from userspace is either by means of library interposition (as discussed in Section 2.3), or by means of the `inotify` API in Linux which is good for alerting the 'purger' when files have been modified. As library interposition is both faster and more immediate than inotify actions, we have chosen to detect file changes by means of library interposition.

Intercepting calls by means of library interposition arguably requires more implementation effort than modifying system call behavior or intercepting system calls by means of `ptrace`. However, it requires no context switching and is thus quite efficient. Another thing one may argue is that libc interposition misses direct system calls made by applications that bypass libc. In practice, this is not a problem and we did not encounter any real applications, besides malware, that bypass libc to make system calls directly. A more complex issue presents itself for memory mapped files. Whenever a file is memory mapped with write permission, we need to mark it uncacheable, because updates would bypass the interposer. Given the application domain (network-attached web and file servers), we do not see this as a problem. Moreover, library interposition is simple, fast and requires no privileges other than those of the process itself. Finally, we only need to intercept a handful of calls and only for applications that may update the files served by the server.

2.4.4 Limitations

So far, we have assumed that the path of a file can be used as a (unique) identifier. This is not true in the presence of aliasing [13]. For instance, both hard and symbolic links may point to the same file via different paths[4]. A write to any of these paths should lead to a purge of all entries that refer to that file.

We consider this a minor issue, as in a web server links and mounts that lead to the same file via different paths are probably not very common. A simple solution is to use the `stat` system call to obtain the inode and use a (device, inode number) combination as a unique identifier. Maintaining a mapping from the (device, inode number) tuples to the cached entries is straightforward and allows us to purge the stale content from all caches. Moreover, we can obtain the mapping in advance by scanning the file system. As the amount of aliasing is likely to be small, and a mapping entry is needed only for the files that exhibit it, the expected size of the mapping table is small also.

Aliasing may also lead to the same file appearing in the cache twice. As this is a rare situation that does not lead to incorrect results anyway, we do not address this issue.

Finally, *CacheCard* is quite conservative. For instance, if an unrelated row is modified in a database table, all entries in the cache that also use that table are purged. While this is no problem for correctness, it may lead to suboptimal performance. In future work, we hope to go beyond file granularity and look at individual records. This is also important for larger databases that work with raw disks or do not equate files and tables.

3. EVALUATION

Our evaluation consists of three parts. First, we consider the case for caching: is *CacheCard*-based caching beneficial in terms of hit rates? Second, we briefly look at invalidation set contraction: how many invalidation sets are sub-optimal? Third, we consider to what extent *CacheCard* improves throughput and latency. Before we start measuring, however, we need to decide what to use for comparison. It makes no sense to compare against Squid proxies, because the two configurations have completely different properties (e.g., in terms of staleness, dynamic data, cache size, management costs). Rather, we compare against an off-the-shelf web server which exhibits the same (desirable) behavior as offered by *CacheCard*: immediate invalidation, no additional management, power or rack-space, etc. Finally, we mention that in all experiments the interposition of libc did not have noticeable performance overhead.

3.1 The case for caching: hitrates

[4]While UNIX path names are often also slightly different from URL pathnames, the difference tends to be a prefix only, which presents no serious problems.

For large web servers like Wikipedia, caching has already proved itself quite useful with hitrates well above 70% [24]. But while hit rates are high, they are achieved with Squid-like servers with large amounts of memory (say, 16GB) and sometimes fast disks. How beneficial is caching with smaller caches and with non-Wikipedia-like workloads?

We conduct a set of experiments with real workloads of our departmental web server to measure hitrates for different cache sizes, by applying *CacheCard* during a *quiet* (out of term), and a *term* (normal, in-term) period. In addition, to see how traffic has changed in the last ten years, we also considered *CacheCard*'s effectiveness for the EPA and ClarkNet *traces* of the Internet Traffic Archive [1] trace from 1995[5]. The results are shown in Figure 4 (for $\alpha = 0.75$).

The cs-quiet and cs-term each correspond to a week of HTTP traffic to the web server of our computer science department. Even in the quiet period the server handled 2.8 million requests for 402.000 unique objects. In the normal period, the number rose to 3.4 million requests. In both cases, file popularity was approximately zipf distributed. Of all requests, about 12% were PHP requests. Other dynamic content (cgi-bin, JSP) contributed less than 5% of the requests. The Clarknet and EPA trace-based results correspond to a week and a day of traffic to busy web servers, respectively, and contain little dynamic content (around 1%).

We see that the hitrates for the CS server in the busy period (with a peak at 65%) is much higher than in the quiet period (peaking at 40%), reaching 60% for cache sizes of $2^{17} = 128 \times 1024$ slots and higher. For the Clarknet and EPA traces the hitrates are very high indeed as these traces exhibit good locality and the files are relatively small compared to those of the CS web servers (file sizes and number of files on desktops doubled and tripled in 4 years, respectively [2]). We were surprised to see how quickly returns diminish when we increase the cache size. Sizes beyond 2^{17} slots (200MB) hardly improve overall hit rate at all, and even 25-50MB may be sufficient to achieve reasonable hitrates.

The amount of data that is served from the cache for a cache size of 512k slots is 54% for cs-quiet and 89% for for cs-term. Surprisingly the size of the cache does help in reducing the amount of traffic across the bus. For instance, a cache size of 128k slots reduces the amount of data served by the cache to 20% for the quiet period and 69% for busy period. This suggests that the files added to the cache are big compared to the average reply size.

3.2 Invalidation set contraction

In our second experiment we determine the number of sub-optimal invalidation sets and the maximum amount of over-estimation. To do so, we load a web server with randomly generated requests for PHP scripts that each open exactly one file. The number of scripts in the system is set to one thousand and script popularity has a zipf distribution.

Requests are sent from three clients with a Poisson distribution and an average inter-arrival time of 1ms. The requests are sent asynchronously, so multiple GET requests will be outstanding at the same time. Figure 5 shows the number of sub-optimal invalidation sets as well as the maximum amount of over-estimation for two cases. In the first case ('fast script'), the script immediately returns the content of a minimum-sized file. In the second case ('slow script'), the script explicitly sleeps for one second (simulat-

[5]Publicly available traces are very scarce and often old.

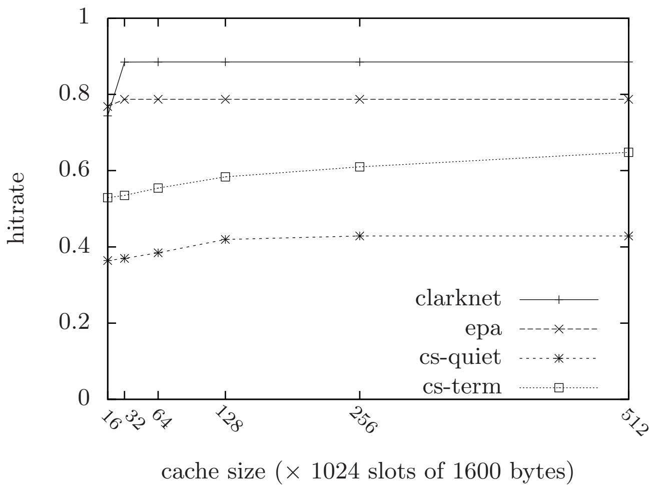


Figure 4: Hit rates for web servers (α is 0.75)

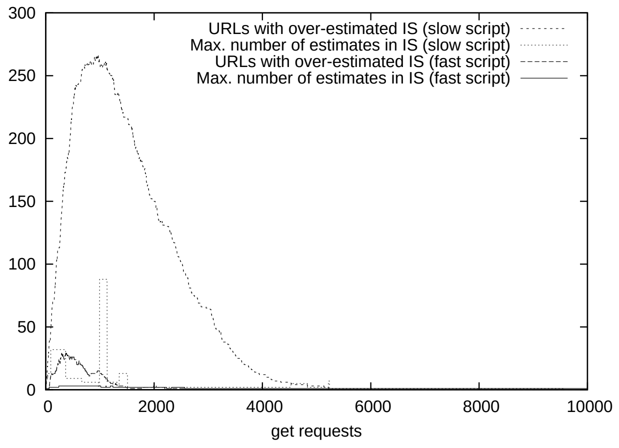


Figure 5: invalidation set contraction

ing a long processing time). In the latter case, we have more overlap in requests and thus more sub-optimal invalidation sets. It is representative of a (not very realistic) worst-case situation.

The figure confirms this. We see that in the worst case up to approximately 15% of the invalidation sets have overestimations and that in the first 1200 requests the amount of overestimation can be as high as 260. However, we also see that the amount of overestimation drops quickly and is minimal after approximately 1700 requests. Even the number of invalidation sets with over-estimations is negligible after 4000 requests. For the fast scripts, the amount of overestimation is minimal throughout the experiment.

3.3 Web server performance

We have seen that caching on the card is effective, and that invalidation sets contract quickly. In this section, we show micro-benchmarks to measure the performance of *CacheCard* for web servers. The question we will try to answer in the remainder of the evaluation section is: given a good hit rate, how well does *CacheCard* perform compared to the native configuration when all popular objects are in memory (e.g., when the webserver is slash-dotted)? We used httperf to load our webserver and measure the scalability.

3.3.1 Experiment setup

In our experiments, we evaluate *CacheCard* by comparing the throughput of a webserver (along different dimensions) for a varying number of connections. As a baseline, we first establish the throughput of the native server. This is the maximum capacity of our web server and web clients without caching.

Our test setup consists of Intel P4 CPU 2.8GHz PCs with 512KB cache and an 800MHz front-side bus running running Fedora 4 Linux with kernel version 2.6.17. The server and clients are connected to the network via Intel 82545EM Gigabit Ethernet cards. The server in our test bed is a stock Apache-2.2.8 server with PHP-5.2.4 and MySQL-5.0. The cache contains PHP generated data from a database as well as static files.

In the *CacheCard* configuration, an IXP2400 evaluation board running our software is plugged into one of the server's PCI slots. We used a minimal web-cache-size of 16K slots of 1600 bytes, making the size of the cache (approx. 25MB) of the same order of magnitude as that of the cache used by Kim et al. [14]. However, the size of the cache does not influence the following micro-benchmarks as we ensure that all accesses are cache-hits. The maximum number of concurrent TCP connections in *CacheCard* is configurable. For all the experiments in this section, it was set to 1024, the same value as the maximum number of open file descriptors on each of the Linux PCs.

We conduct our experiment under worst-case conditions for *CacheCard*, where all data is static (so apache need not execute any scripts either). While it would be more favorable to test *CacheCard* for data generated by scripts (which generate significant overhead for apache, but none for *CacheCard*), it raises a difficult issue: what is the right script against which to test (in terms of complexity, number of file and database accesses, etc.)? Rather than trying to solve this issue, we used up to three clients with `httperf` as web performance benchmark [19]. Moreover, each client connection sends HTTP GET requests for data that is either in *CacheCard*, or in the native web server's main memory. In other words, we compare Apache on equal terms, without disk I/O. In all experiments, the connections are distributed among the clients as evenly as possible.

3.3.2 Throughput

Perhaps the most interesting metric in web server performance is its scalability in number of connections. We compare the performance with (CC) and without (noCC) *CacheCard* when hitting the server with increasing request rates and for different file sizes (1 byte, 100 bytes, 10 kilobytes, and 1 megabyte). The sizes 1 byte and 1 megabyte were chosen as extreme cases. The 10 kilobyte file is approximately the average size of a JPEG image [17]. It is also close to the average size of PNG images and external scripts, and in the same order as the average HTML file (which is 25 kilobytes). The 100 byte value is a lower-bound on the size of files with real content as found in practice on the web. We repeat experiments at least 31 times to derive meaningful statistics.

Figure 6 shows the server's reply rate for increasing rates

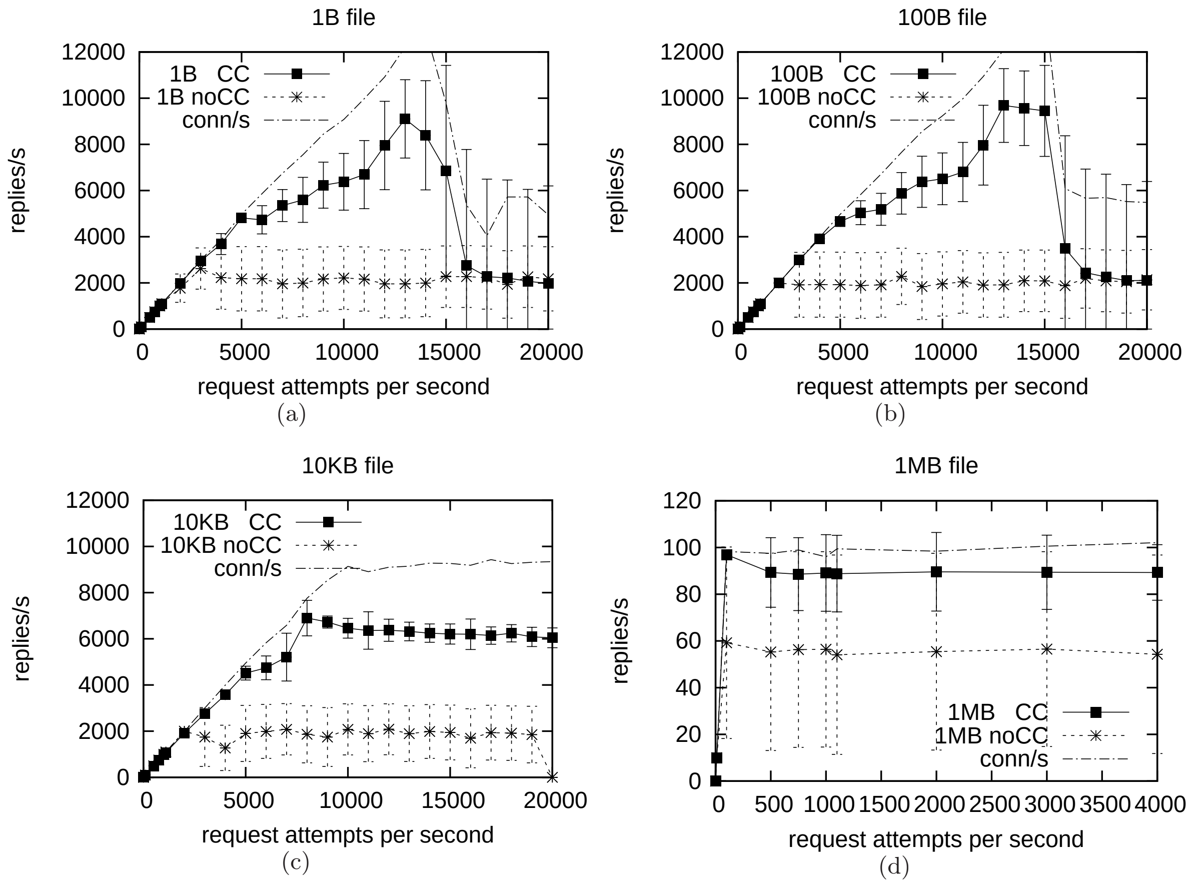


Figure 6: Reply rate with and without *CacheCard*, for different file sizes. The figure shows both the average reply and the 1-standard deviation interval for the configuration given the desired request rate as indicated on the y-axis. As the client was not always able to achieve the desired request rate, the plots also show the TCP connection rate achieved.

of attempted requests. As the request rate grows beyond a few thousand per second, TCP connections are sometimes dropped (or do not complete their handshake in the server). We therefore also show the average TCP connection rate that the client achieved in the *CacheCard* configuration (the top line without error bars in each of the figures). For the first three plots, up to about 10 to 13 thousand connections per second (depending on file size), the connection rate almost keeps pace with the attempted request rate (e.g., for 10.000 attempts to retrieve the 1 byte file, about 9.000 are successful). Beyond 13.000 attempts per second, the performance plunges. For a file size of 10KB, the actual rate of successful connections tops out at 9000 connections per second in the case of *CacheCard*. For the 1MB file, we achieve about 100 requests per second. This is expected. One hundred connections with 1% of 1Gbps each would take a bit less than 1s to complete the download of a 1MB file. With one thousand connections the download would take roughly 10 times as long, so the connection rate would be the same.

The other lines in the plots show the average reply rates of CC and noCC with the standard deviation. For all sizes, CC significantly outperforms noCC, on average about 3 times (sometimes more than 4 times) as good. Taking a file size of 1B as an example, we see that the noCC tops at about 2000 replies/s, while CC keeps rising until approximately 9000 replies/s. Nevertheless, for a request rate between 6000 and 13.000 requests per second the slope of the curve drops from 1.0 to 0.6. Beyond that rate, the system overloads and performance collapses, until it stabilizes at the rate of noCC; mainly because of a lower rate of successful TCP connections.

For files of 10KB the CC reply rate remains stable at approximately 6000/s. It does not collapse, because the TCP connection rate remains approximately 9000 connections per second. The size of the file is relevant, because large transfers stay active longer. As the maximum number of open file descriptors is capped at 1024 (default for Linux), the actual rate that connections are generated is moderated. As expected, for 1MB files (Figure 6.d), the connection and reply rates are low. Even so, CC outperforms noCC by 30-40%, probably because the latency and transmission rate of the CC configuration is better than that of noCC, so fewer connections overlap.

To obtain a feel for the latency, we measured the total length of individual TCP connections. The average and median results for 1B and 10KB files are shown in Figure 7 (the results for the other sizes are similar and have been omitted for space efficiency). For noCC, the standard deviation ranged from 0.1 to 82.9 ms for connection rates up to 1100 req/s, and shot up to over 2000 ms immediately after.

For CC, it ranged from 0.1 to 54.3 milliseconds for connection rates up to 16000 req/s (which is much better than in noCC). Beyond that rate, it shot to about 2000 ms also. We were unable to draw standard deviations in the plots while keeping them readable. We see that the connection time for individual requests is much better in the *CacheCard* configuration than in the noCC setup. Long connection times, means more and prolonged conflicts with other connections, explaining some of the poor results achieved with noCC.

In the final experiment, we download a large file and look at the throughput in bits per second. For small numbers of connections (less than 10), CC is slower than noCC (830 Mbps vs. 920 Mbps). This is expected, as ach individual IXP core is slow compared to the host processor. For 600 simultaneous connections the throughput is exactly the same, while for 1000 simultaneous connections, CC outperforms noCC by a factor 3 at least, although absolute rates are quite low in both cases (104 Mbps vs. 34 Mps).

4. RELATED WORK

Web. Web caches have been popular since the 90s. Many existing proxy caches be they from industry (for instance, Cisco's 7300 Content Engine [8]), the open source community (Squid [28]), or research (projects like Squirrel [12]), serve transparently redirected HTTP requests. Well-known projects like Squid [28] and Harvest [6] are based on the Internet Cache Protocol which does not address the problem of dynamic content. Most work on caching dynamic content requires the cache to understand the back-end applications, sometimes to the extent that it needs to distinguish between static and dynamic elements of a page (e.g., Domproxy [26] handles dynamic content by caching both content and applications).

Cache coherence for web pages is a complex trade-off between performance and freshness [29], and different approaches exist, including time to live and invalidation [11]. Most commonly, polling is used in the web by means of '`If-Modified-Since`' HTTP headers, while NAS/NFS caches touch base with the original file server upon receiving open or read requests to retrieve the file attributes.

Content distribution solutions often do not guarantee freshness and neither does Wikipedia. As a result, users experience and occasionally complain about, staleness [25].

Storage. Nache [10] provides a proxy cache for NFS by leveraging file delegations provided by NFSv4. It aims at WAN configurations and pushes the proxy cache to the clients. While NFSv4 may grow in importance, version 2 and 3 are currently more popular. Server-side caching is common primarily in larger storage systems [30], probably beyond the budget of many medium-sized organizations.

Caching on the NIC. Few projects have used NICs for caching. To the best of our knowledge, only two papers come close to our approach [7,14]. Both differ from *CacheCard* in that they do not handle dynamic content and require extensive modification of the operating system and/or web server. An additional, but smaller difference is that we cache network data rather than disk blocks. Similar payload caching was applied by Yocum and Chase for retransmissions [33].

Arguably the most influential of these projects is the system by Kim et al. at Rice University [14]. The authors show that even a small amount of cache memory (16MB) improves throughput by 7-31%. As mentioned above, the performance comes completely at the cost of transparency. Kim et al modified the core of the FreeBSD operating system (syscalls, drivers, etc.) to take advantage of the programmable hardware. The Myrinet cache for clusters likewise requires significant modifications to the web server [7].

In contrast, we do not even need access to or recompile server source code, let alone make changes. Since *CacheCard* is independent of the web server itself, it should be straightforward to combine our approach with new server architectures like SEDA [27].

The work by Yao [32] is also related to *CacheCard*. In their work they emulate on a PC a NIC that caches data for iSCSI storage servers with cache sizes ranging from 8 to 128MB yielding hit rates up to 50%. The current state of the project appears to be a PC-based emulator rather than an implementation on actual hardware. Also, because of the application domain, there is no notion of dynamic data.

Finally, *CacheCard* is related to TCP and checksum offloading [15, 21]. Whether TCP offloading is good idea or not, is still under debate [20, 23], but some suggest that it would be better we offloaded more involved tasks than just TCP [18]. This is certainly true for *CacheCard*.

5. CONCLUSIONS

We have shown that caching dynamic content on the network card is made possible by modern programmable network cards and argued that doing so offers several advantages over caching in separate machines. *CacheCard* strives to give an illusion of a single file/web server of 'arbitrarily' large capacity and one that can be dynamically expanded by means of *CacheCard* plugin modules.

Acknowledgments. This work was partly sponsored by the Intel IXA EDU programme. The authors thank Leonid Ryzhyk, Kevin Elphinstone (NICTA, Australia), Willem de Bruijn (VU), and all reviewers for feedback on earlier versions of the paper, Guillaume Pierre and Guido Urdaneta (VU) for providing us with traces, and Lian Lin (Xiamen University) for his help with the experiments.

6. REFERENCES

[1] Internet traffic archive. `http://ita.ee.lbl.gov`, 2008.

[2] N. Agrawal, W. J. Bolosky, J. R. Douceur, and J. R. Lorch. A five-year study of file-system metadata. In *Proc. FAST'07*, San Jose, CA, 2007.

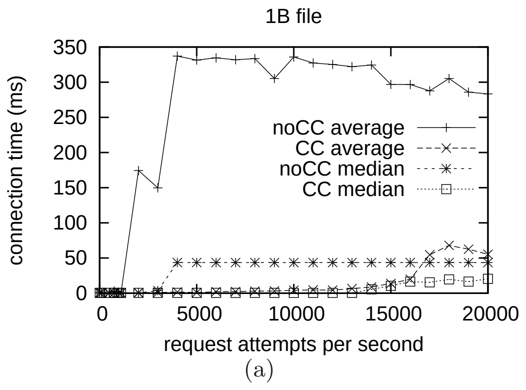
[3] M. F. Arlitt and C. L. Williamson. Internet web servers: workload characterization and performance implications. *IEEE/ACM Trans. Netw.*, 5(5), 1997.

[4] M. Bergsma. Wikimedia's (cache) network. In *Proceedings of Wikimania HD2006*, MIT Media Lab, Cambridge, MA, USA, August 2006.

[5] M. Bergsma. Wikimedia architecture. `http://www.scribd.com/doc/43872/Wikimedia-architecture`, April 2007.

[6] M. Bowman, P. Danzig, D. Hardy, U. Manber, M. Schwartz, and D. Wessels. The Harvest Information Discovery and Access System. Internet Research Task Force - Resource Discovery, `http://harvest.transarc.com/`.

[7] G. S. Choi, J.-H. Kim, D. Ersoz, M. Yousif, and C. Das. Exploiting nic memory for improving cluster-based webserver performance. In *IEEE Cluster*, pages 1–10, Burlington, MA, Sept. 2005.



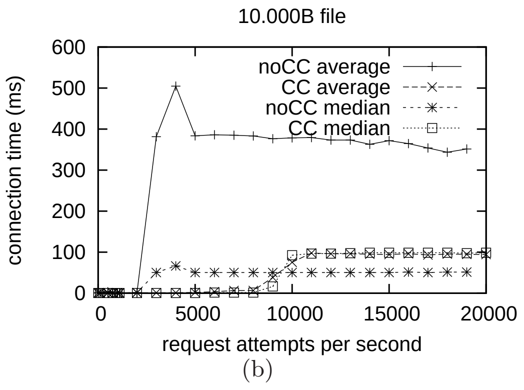


Figure 7: TCP connection time downloading two different file sizes (minimal and typical) for the configuration given the desired request rate as indicated on the x-axis. Standard deviations are discussed in the text.

[8] Cisco. Cisco enterprise content delivery network solution - cisco content engines (data sheet). `http://www.cisco.com/warp/public/cc/pd/cxsr/ces/prodlit/cdnhw_ds.pdf`, 2003.

[9] W. de Bruijn and H. Bos. Beltway buffers: avoiding the OS traffic jam. In *INFOCOM*, Phoenix, AZ, 2008.

[10] A. Gulati, M. Naik, and R. Tewari. Nache: design and implementation of a caching proxy for nfsv4. In *Proc. of FAST'07*, pages 27–27, Berkeley, CA, USA, 2007.

[11] J. Gwertzman and M. Seltzer. World-wide web cache consistency. In *USENIX*, San Diego, CA, 1996.

[12] S. Iyer, A. Rowstron, and P. Druschel. Squirrel: A decentralized peer-to-peer web cache. In *Proc. of PODC*, Monterey, California, July 2002.

[13] T. Kelly and J. Mogul. Aliasing on the world wide web: prevalence and performance implications. In *Proc. WWW'02*, pages 281–292, Honolulu, Hawaii, USA, 2002.

[14] H. Y. Kim, V. S. Pai, and S. Rixner. Increasing web server throughput with network interface data caching. In *ASPLOS-X*, pages 239–250, San Jose, California, 2002.

[15] K. Kleinpaste, P. Steenkiste, and B. Zill. Software support for outboard buffering and checksumming. In *Proceedings of SIGCOMM*, pages 87–98, 1995.

[16] D. Lee, J. Choi, J.-H. Kim, S. H. Noh, S. L. Min, Y. Cho, and C. S. Kim. On the existence of a spectrum of policies that subsumes the least recently used (LRU) and least frequently used (LFU) policies. *Sigmetrics Perform. Eval. Rev.*, 27(1), '99.

[17] R. Levering and M. Cutler. The portrait of a common html web page. In *DocEng '06*, pages 198–204, Amsterdam, The Netherlands, 2006.

[18] J. Mogul. TCP offload is a bad idea whose time has come. In *Proc. of HotOS IX*, Lihue. Hawaii, USA, May 2003.

[19] D. Mosberger and T. Jin. httperf - a tool for measuring web server performance. *SIGMETRICS Perform. Eval. Rev.*, 26(3):31–37, 1998.

[20] M. O'Dell. Network front-end processors, yet again. *Communications of the ACM*, 52(6):46–50, June 2009.

[21] M. Rangarajan and A. Bohra. TCP servers: Offloading tcp processing in internet servers. design, implementation and performance. Technical Report DCS-TR-481, Rutgers University, 2002.

[22] G. M. Sacco and M. Schkolnick. A mechanism for managing the buffer pool in a relational database system using the hot set model. In *Proc. VLDB*, San Francisco, CA, USA, 1982.

[23] P. Shivam and J. S. Chase. On the elusive benefits of protocol offload. In *ACM SIGCOMM NICELI'03*, pages 179–184, Karlsruhe, Germany, 2003.

[24] G. Urdaneta, G. Pierre, and M. van Steen. Wikipedia workload analysis for decentralized hosting. Technical Report IR-CS-041, Vrije Universiteit Amsterdam, 2008.

[25] Various. Is Wikipedia's caching screwed up?, Failure to synchronize? Wikipedia Review, `http://wikipediareview.com/index.php?showtopic=6994`, February 2007.

[26] M. Veliskakis, J. Roussos, P. Georgantas, and T. Sellis. DOMProxy: Enabling dynamic-content front-end web caching. In *IEEE WCW*, pages 56–61, Sophia Antipolis, sept 2005.

[27] M. Welsh, D. E. Culler, and E. A. Brewer. SEDA: An architecture for well-conditioned, scalable internet services. In *SOSP*, pages 230–243, 2001.

[28] D. Wessels. The Squid Internet Object Cache. NLANR, `http://squid.nlanr.net/Squid/`.

[29] J. W. Wong, D. Evans, and M. Kwok. On staleness and the delivery of web pages. In *CASCON '01*, page 17, 2001.

[30] T. Wong and J. Wilkes. My cache or yours? making storage more exclusive. In *Usenix ATEC*, pages 161–175, 2002.

[31] G. Yadgar, M. Factor, and A. Schuster. Karma: know-it-all replacement for a multilevel cache. In *Proc. FAST'07*, pages 25–25, San Jose, CA, 2007.

[32] X. Yao and J. Wang. Toward effective nic caching: A hierarchical data cache architecture for iscsi storage servers. In *ICPP '05*, pages 492–499, 2005.

[33] K. Yocum and J. S. Chase. Payload caching: High-speed data forwarding for network intermediaries. In *Proc. USENIX ATEC*, pages 305–317, 2001.