Surviving a Search Engine Overload

Aaron Koehl
Department of Computer Science
College of William and Mary
Williamsburg, VA, USA
amkoeh@cs.wm.edu

Haining Wang
Department of Computer Science
College of William and Mary
Williamsburg, VA, USA
hnw@cs.wm.edu

ABSTRACT

Search engines are an essential component of the web, but their web crawling agents can impose a significant burden on heavily loaded web servers. Unfortunately, blocking or deferring web crawler requests is not a viable solution due to economic consequences. We conduct a quantitative measurement study on the impact and cost of web crawling agents, seeking optimization points for this class of request. Based on our measurements, we present a practical caching approach for mitigating search engine overload, and implement the two-level cache scheme on a very busy web server. Our experimental results show that the proposed caching framework can effectively reduce the impact of search engine overload on service quality.

Categories and Subject Descriptors

C.4 [**Computer Systems Organization**]: Performance of Systems—*Reliability, Availability, and Serviceability*

Keywords

Web Crawler, Overload, Caching, Dynamic Web Site

1. INTRODUCTION

Crawler-based search engines are widely recognized as an essential and positive component of the web by both web users and site owners. Specifically, they provide a means for web users to efficiently locate desired content on the Internet, while providing site owners with the means to have their content discovered. Behind the scenes, web crawlers are the software tools that operate on behalf of search engines, responsible for continually sifting through web sites, downloading pages and related links, and ultimately aiming to discover and subsequently index new content. Often underestimated is the impact that these web crawling agents can have on heavily loaded dynamic web sites, and the resultant cost to the user population and site owners. These web crawlers can sometimes overburden dynamic web sites, threatening quality of service for paying clients. It is not uncommon to observe an entire large site being crawled using hundreds of simultaneous hosts, and from a multitude of web crawlers. Such heavy, distributed traffic can manifest as a denial of service from the server's perspective [19].

WWW 2012, April 16–20, 2012, Lyon, France.
ACM 978-1-4503-1229-5/12/04.

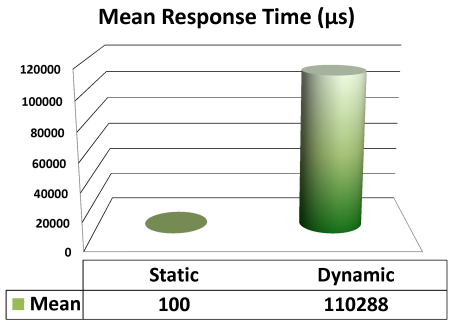

Figure 1: Mean static vs dynamic time to first byte (TTFB) over two weeks, from a production web server.

Dynamic websites are those that serve script-generated content, usually backed by a database, and encompass much of the Web 2.0 we experience today: blogs, forums, social networking tools, and other web applications. These dynamic websites are more costly in terms of server resources than their static counterparts [16]. Current research progress allows static websites to scale well in capacity with the amount of hardware deployed, primarily due to file system optimizations, kernel and network stack improvements, load balancing techniques, and proxy caching. While the processing time required to serve a static file from the file system is relatively constant, serving dynamic content can introduce considerable variability. Aside from the load on the web server, variations in processing time are introduced by script length, number of includes, nature of the computation, database responsiveness, and network latency to other servers. Because of the complexity of these interactions, dynamic scripts are much more expensive than their static counterparts, as shown in Figure 1. When search engine crawlers request more dynamic scripts than the server can reliably handle, we call such a condition *crawler overload*.

A contemporary problem faced by site administrators is how to effectively manage crawler overload on dynamic websites. Table 1 illustrates the overall load induced by crawlers on a very popular site we manage. Although crawlers only represent 6.68% of all requests, they consume an astonishing 31.76% of overall server processing time. Many large dynamic sites operate at or near capacity, providing the motivation for site administrators to selectively introduce software optimizations as the site grows, until additional hardware becomes the only alternative. At peak times, the

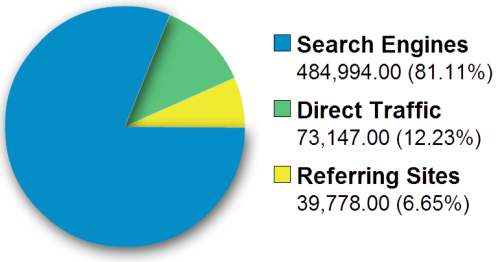

Figure 2: Google Analytics of traffic referrals for a popular site we manage.

	Crawler	Human
Requests (%)	6.68	93.32
Load (%)	31.76	68.24

Table 1: Crawlers compete with humans for server processing time.

additional load caused by search engine crawlers is critical. It competes heavily for resources with human users, and can result in a denial of service, manifesting as unserved requests or unbearably slow responses to a human user. Under non-peak times, crawlers still collectively keep a web server busy. Though not competing with humans for resources, the crawlers nonetheless impact energy consumption, preventing low-power states from being achieved.

To mitigate the overload effects caused by web crawlers, a simple strategy is to ignore requests from search engines, but this can have devastating economic consequences. Google Analytics from one very popular site we manage shows that 81% of human traffic comes as a result of referrals from search engines, as shown in Figure 2. Furthermore, a site which appears unresponsive to a crawler can negatively impact search engine rankings [4].

We therefore seek to quantify the effects and costs that web crawlers pose to the dynamic web community, and subsequently propose a solution for mitigating overload risk. We begin our approach with an analysis of the variations in surfing patterns between humans and crawlers. As expected, the patterns are starkly contrasting in both frequency and distribution, providing insight which leads to our solution. We find that crawlers are indeed "greedier", "hungrier", and "heavier" than their human counterparts. On a busy web server receiving hundreds of page requests per second, and in turn executing thousands of database queries, crawler load from dynamic requests quickly adds up.

As dynamic websites become richer and more personalized, the amount of cacheable content decreases, reducing the applicability of traditional caching methods. For example, clickable links for options such as pagination, "printable versions", and various sorting schemes of interest only to human users are often implemented as distinct URLs. The number of possible combinations to cache can quickly become insurmountable. However, by nature, crawlers do not require this level of personalization and crawl at the lowest security level—that of "guest" users. By studying crawler activity, we expose optimization potential for this class of request, and develop a two-level caching scheme for mitigating crawler-induced load.

A key observation is the relatively low frequency with which individual pages are crawled for new content. While humans always expect the freshest content, our analysis shows that we can treat search engine crawlers differently, by generating the requests offline and amortizing the load. On average, we find that a static file ($102\mu s$) can be served three orders of magnitude faster than the same dynamic request ($170,100\mu s$). Thus, in order to mitigate the risk of crawler overload, a static caching scheme is implemented, and populated during off-peak hours. When a request is identified as coming from a web crawler, the URL is rewritten to serve the page from the cache, avoiding the database and server side scripting altogether. The site is still properly indexed and remains responsive to both crawler and human, and the crawler has no idea the URL is being rewritten. More importantly, due to the access patterns we identify, the crawler always receives fresh content from its perspective.

The remainder of the paper is structured as follows. Section 2 presents the measurement-based characterization of web crawling agents. Section 3 details our caching framework. Section 4 discusses our implementation. Section 5 evaluates the proposed solution on a real site. Section 6 highlights related work, and Section 7 concludes.

2. WEB CRAWLER CHARACTERISTICS

We motivate our analysis by mining web server logs on a very high traffic web server serving roughly 47,000 unique visitors per day. The server is used for a popular online forum running the vBulletin forum software [18]. The access logs, collected over a 2-week period between February 26 and March 13, 2011, produced about 17 GB of data, representing over 41.8 million requests. Using specialized scripts, the logs were parsed and stored into a MySQL database. Malformed requests were discarded, URLs were decoded and parsed, and request strings were further cross-referenced with post and thread identifiers from the online community to support our analysis.

The logs were provided in the Apache common log format, and include the host IP address, timestamp, url, http status code, the number of bytes transferred, the referring URL, and the user agent string provided by the browser. We also configured the web server to log each request's total processing time for further analysis. However, in our first effort on data collection conducted in January 2011, we found the total processing times to be extremely heteroscedastic: the values collected by default include in aggregate both the server processing time and the time to transfer the content to the user, and with a high degree of variation in connection speed, this adversely reduces the utility of the data collected. Thus, the Apache server was modified to log the time to first byte (TTFB) for each request, providing a much better indicator of script processing time [2].

The load generated by search engine web crawlers can be considerable. In one instantaneous snapshot from January 2011, of the one thousand connected clients, we identified close to 300 continuous guest sessions from 26 unique search engine crawlers.

2.1 Crawler Identification

Whereas sophisticated methods of crawler detection have been proposed, our requirements for crawler identification differ slightly from previous research. Our goal is to develop a classifier with extremely low overhead, as it must be executed at every request, and the classifier is accurate enough

Σ Processing Time	Cumulative (%)	Rank
242425514986	18.95	1
30270206599	30.23	5
3796494907	50.26	36
59815901	90.00	1900
13655020	95.00	4041
100	100	47928

Table 2: Distribution of user agent strings on cumulative server processing time is extremely long-tailed.

to identify only the subset of crawlers which contribute high load to our server.

Several existing systems attempt to classify search engine crawlers by producing a list of hosts that request the `robots.txt` file from the server [1, 6]. In assessing this approach, we found that of the 374 positively identified crawlers in our logs, only 155 (41%) bothered requesting the file. Robots.txt allows site administrators to provide guidance to crawlers on how a site should or should not be crawled. Ideally, this file is used to protect non-public data from being indexed, to prevent deep search in virtual trees, to prevent crawlers from being fed duplicate or transient content, and to limit the rate of the crawl. In addition to accuracy, there are two other problems with using this approach for identification: (1) a hash of hosts must be maintained and checked at every request, which can grow exceedingly large, and (2) the hosts continually change and become stale.

The approach we used is classification based on the user agent string, which is provided by the browser for each request. The following includes a common user agent string from Microsoft's Bing. We found that it is typical to include a URL for server administrators to obtain more information about the crawler.

```
Mozilla/5.0 (compatible; bingbot/2.0;
  +http://www.bing.com/bingbot.htm)
```

For the period of time we studied, we found 47,927 distinct user agents in the server logs, with varying frequency. Most of these belong to human browsers and include various combinations of browsers, operating systems, and software identifiers like the following:

```
Mozilla/5.0 (Macintosh; U; Intel Mac
  OS X 10_6_6; en-us) AppleWebKit/533.19.4
  (KHTML, like Gecko) Version/5.0.3
  Safari/533.19.4
```

In order to identify the subset of user agents which contribute the highest impact on our server, we grouped all of the user agent strings and ranked them in descending order by total aggregate processing time. Ranking them by frequency of requests proved unfruitful, as the requests with the highest frequency are dominated by static file requests issued by human users, and contribute little to overall processing load. We then proceeded in a manual effort to identify user agents as crawler or human, stopping when we reached a reasonable threshold representing a cumulative 97.5% of the total processing time for the two and a half week period. A histogram of the user agents appearing in our logs shows an extremely long-tailed distribution, and is summarized in Table 2. Due to this long-tailed distribution, a manual screening of the remaining 3.5% of processing load would require analyzing an additional 38,090 user agents, for very little additional benefit. Of the 47,928 user agent strings, 30% of all processing time is attributed to the top five hosts, and perhaps unsurprisingly, Googlebot consumes 18.95% all on its own. Furthermore, of the top 5 individual consumers of processing load, four are known crawlers (Google, Bing, Msnbot, Google Adsense)—the remaining user agent string is the one listed above. In terms of request frequency, only one bot is even in the top 18 user agents, further illustrating the fallacy in using hits alone when optimizing for performance, an effective metric for static websites.

Through this manual classification effort, we discovered an incredibly simple heuristic to identify user agents as crawler or human, achieving our goal for fast classification. Recall that our primary motivation is to identify the largest consumers of processing time; the following heuristic meets this goal with 100% accuracy for currently dominating user agents.

Heuristic 1 IdentifyBotFromUserAgent

Input: *ua*, a user agent string.
Output: *true* if probably a bot, *false* otherwise.

```
 1: if ua contains bot or crawler then
 2:     return true
 3: if ua contains slurp and not Firefox then
 4:     return true
 5: if ua contains Mediapartners or PubSub then
 6:     return true
 7: if ua contains http then
 8:     if ua contains bsalsa or browser then
 9:         return false
10:     else
11:         return true
12: return false
```

We also found two scenarios where the user agent string was not reported in the request, or was simply reported as "-", but found the frequency of these cases to be too insignificant (0.25%) to warrant special treatment. Through our manual screening effort of over 9,600 user agent strings, we identified 374 user agents as crawlers, all of which were accurately classified by our heuristic with no false positives. It is worthwhile to mention that in our experience, new search engines appear in our logs every month. In the extremely likely event that a new search engine is developed, our heuristic can be easily modified to accommodate its detection. Of course, this would not be necessary until the new search engine begins to exhibit anomalously high load, which as the long-tailed distribution of user agents shows, is not all that likely. Should a user masquerade as a bot by changing its user agent string to match our heuristic, it could be misclassified, but that may be the user's intent. Nonetheless, we have found this classifier to be very useful in practice.

2.2 Crawler Access Pattern

While the forum serves requests from 189 countries and territories, the majority of the subscriber base is located in anglophonic North America. Figure 3 shows a predictable, diurnal pattern of usage, with a daily peak exhibited at noon, a slight valley at 6:00pm building up to a crest at 9:00pm, and dropping off rapidly thereafter. Search engine crawlers, in contrast, exhibit a relatively uniform frequency of access (with a slight spike at noon), as the figure also shows. Two observations can be reaped from Figure 3. First,

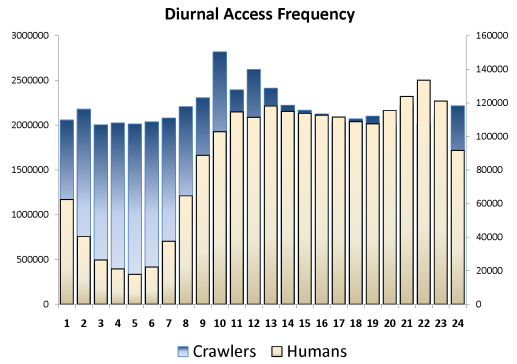

Figure 3: Daily traffic pattern for a popular website, peaking at 4M hits. Humans exhibit a predictable, diurnal access pattern, while crawler access is relatively uniform.

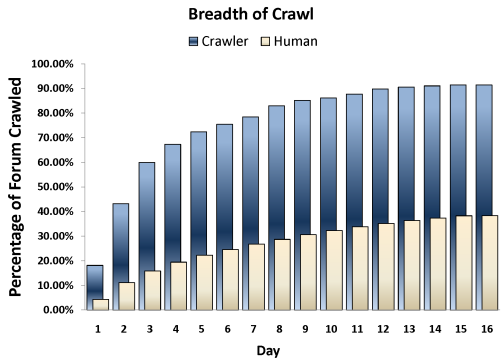

Figure 4: Crawlers exhibit lower reference locality compared to humans, crawling a full 92% of the entire forum in about 2 weeks.

4:00am provides an ideal circumstance for batch processing, which we can utilize to populate our cache. Second, due to the uniform nature of crawler accesses, a targeted reduction in crawler load theoretically provides an overall increase in server capacity.

A straightforward application of Amdahl's Law to the measurement results in Table 1 and Figure 1 provides an upper bound on the reduction in overall processing load. If static requests are served three orders of magnitude faster than dynamic requests, we can approximate the potential improvement in performance from servicing crawler requests from the static cache.

$$\text{Overall Improvement} = \left[\frac{31.76\%}{1000} + \frac{68.24\%}{1}\right]^{-1} = 1.46$$

Based on this workload, we would be able to decrease aggregate processing time by factor of 1.46. For a server operating at capacity, this overall reduction could shield human users from overload conditions during peak times. Clearly, the peak hours from 10am to 10pm would benefit greatly from a uniform reduction in server load.

2.2.1 *Distribution of Load*

As indicated in Table 1, crawlers are responsible for under 7% of all requests, yet are nonetheless responsible for around 32% of overall server load. This implies the types of requests made by crawlers are fundamentally different than those made by humans. We found that crawlers are "heavier" than humans, consuming much more expensive scripts in general, and by a large margin. Indeed, crawlers download dynamically generated HTML as human users do, but ignore most of the sub-requests for CSS, banners, small images, bullets, and icons embedded in the page that provide an enhanced visual experience for human users. These are of little use to web indexes. Instead, the crawlers prefer dynamic requests containing a high degree of textual content, which are typically the most expensive to generate.

Human users tend to take time to read a page of content before moving on to a new page, exhibiting what Barford and Crovella identified as an "active-inactive" behavioral pattern [10]. Crawlers, on the other hand, are "hungry", in that they tend to request expensive pages at a much higher rate per host than human users. Whereas a human user might space out their requests by five minutes or more, crawlers make multiple requests within seconds. Table 3 shows the sustained crawl rate (interarrival times averaged over 2 weeks) for several crawlers found in our logs. Dozens of crawlers had mean interarrival times under 1 second, exerting heavy pressure on the server during the crawl, yet did not continuously crawl the site; rather their behavior was bursty. These types of crawls tend to be more likely to introduce server instability under load, as the requests are not spaced appropriately. Even Googlebot, with an average interarrival time of $4.8s$ per visit, frequently had periods of activity with 3 to 5 requests per second for several minutes at a time. Note that the `robots.txt` file in all of these cases had a `crawl-delay` set to 3 seconds, indicating the degree to which this approach can be relied upon.

2.2.2 *Breadth of Crawl*

In order to validate the intuition that crawlers exhibit fundamentally reduced locality by nature, we resolve user `GET` requests from the web server logs to individual thread and post identifiers from the forum database. In this way, we are able to analyze the breadth of the crawl as a result of sustained visits from search engine crawlers.

As of June 2011, our forum hosts about 153,100 threads containing 1.67 million posts. Each thread is accessible through a unique thread ID. Our results show that a full 92% of the forum threads were crawled within the two week monitoring window, with earlier data sets showing 99% of the forum crawled within 3 weeks. Figure 4 is illustrative of the difference between human users and crawlers over time. The crawlers exhibit reduced locality, continuing to follow links deeper into the site, while human users tend to dwell on popular topics of discussion.

As in [10], our human users reliably follow the well-known Zipf distribution [12], visiting only a popular subset of the overall forum content per day. The presence of this distribution is usually a good indicator of cache potential. Unfortunately, each page is customized for individual human users, reducing cacheability. When the data is analyzed over a two week period, we find that crawlers also exhibit a Zipf-like distribution in their access pattern, as shown in Figure 5.

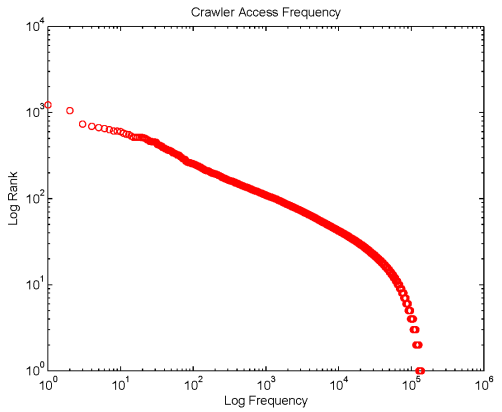

Figure 5: Crawler access pattern is Zipfian as a result of human access patterns. Humans heavily influence content of top-level pages, where crawlers begin their crawl.

Crawler	Interarrival (s)	Requests
Googlebot	4.8	1,500,815
Google Adsense	1.50	172,416
Yahoo Slurp/3.0	1.02	64,125
Yahoo Slurp	33.90	88,455
Exabot	5.20	14,484
Unknown Crawler	0.17	2,613

Table 3: Sustained crawl rate for several bots. Crawl-delay was set to 3 seconds.

These results may be surprising to some, but do consider that a crawler begins its crawl on the top-level pages of a site; content of top-level pages is directly influenced by human access patterns in many online communities. Namely, in the majority of implementations, the top level pages consist of the most popular (or most recently modified) threads, so the most popular pages for humans on our forum are also where crawlers begin their crawl. Still, the crawlers do dig far deeper into the site than human users.

3. CACHING FRAMEWORK

In this section, we describe our caching framework, whose design is mainly based on our measurement observations summarized below.

- Static files can be served from the file system two to three orders of magnitude faster than an equivalent dynamically generated file, even with many optimizations already in place.
- Crawlers consume a high overall server processing time with a uniform access pattern, which makes crawlers an ideal target for reducing server load.
- High-load crawlers are easy and economical to identify using efficient heuristics.
- Crawlers exhibit lower reference locality and differ from humans in their access characteristics, which means a separate caching scheme is ideally suited to this class of request.

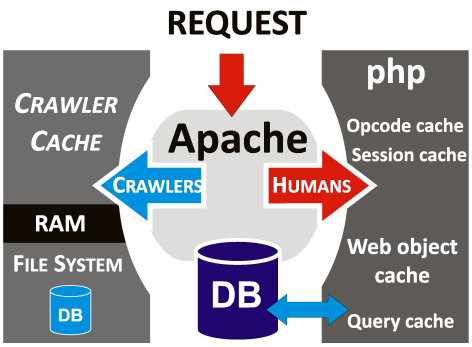

Figure 6: Overview of our framework with respect to caching schemes in dynamic web sites.

- Crawlers do not log in, so their accesses are those of "guest" users. Due to the archival nature of their accesses, crawlers do not require the same content that generally makes dynamic requests unsuitable for caching.

In order to mitigate the overload risk posed by web crawlers on a typical dynamic website, we propose a two-level, lightweight static cache which sits on the originating server, and interposes on behalf of web crawlers. The static cached pages are generated during off-peak hours, and reside in the file system in gzip-compressed state. When a page is requested, the cache need only serve the pre-compressed request to the crawler with little additional processing overhead. Storing in compressed state provides several benefits including reduced latency when the crawler requests the file, and more importantly, it increases the number of RAM-cacheable objects by 5 to 6 times. Cache storage is split into two-levels; a least recently used (LRU) cache in RAM which is continually maintained on each request, and a file-system cache which is populated daily, or more frequently if desired. Figure 6 provides a high-level overview of where our approach fits in with existing cache points for dynamic websites.

3.1 Existing Cache Points

The study of caching for dynamic websites is not new, but different approaches for caching are continually proposed for various points in the web serving process and for specific classes of users. Our proposed cache framework sits directly on the web server, and caches content generally provided to the lowest-security accounts on a dynamic website (so called "guest" accounts which do not have login capability), providing publicly available content for search engines to index. The right side of Figure 6 illustrates the caching mechanisms typically deployed in servers hosting dynamic websites.

Opcode cache. Since php is an interpreted language, each script must be read, parsed, and compiled into intermediate opcodes which are then interpreted by the Zend engine. This overhead is repeated needlessly for millions of requests. The opcode cache allows new requests to be served faster, by jumping directly to the interpreting phase. Opcode caches are shared across user sessions, and only repopulated when the server is reloaded or the source file is changed.

Session cache. As each request arrives, a session identifier for the user is passed to the web server via url or cookie.

Random	12.1 ms
Trace-driven	7.8 ms
Trace-driven + DB	4.5 ms
Sequential	1.2 ms

Table 4: Mean seek times for L2-only cache simulations.

Sessions allow for statefulness on top of `http`, and by default are generally stored in the file system. For fast lookup and retrieval, session caches are often stored in RAM-resident tables in the database or mounted temporary RAM file systems, and periodically garbage collected.

Query cache. The query cache is a database level cache, which selectively stores SQL queries along with the query results sent to clients. Caching avoids the parsing and repeated execution of identical queries, a frequent occurrence with template-based dynamic websites. For tables that are not updated often, the query cache can be very efficient. Query caches are typically shared across sessions.

Web object cache. The web object cache is a memory resident cache for small chunks of arbitrary data. Object caches such as [3] are typically used directly by web applications to alleviate database server load for small, repeated requests. For content management systems (CMS), template and configuration data are often generated from the database and stored here for speedy retrieval over multiple requests.

3.2 Two-Level Caching

With the large disparity between static and dynamic requests, some might question why we include a two-level cache in our design, rather than relying on the static filesystem cache alone. In fact, this approach commands our attention initially; we suspected that crawler locality would be so low that caching in RAM would be pointless. Our measurement study in the previous section invalidates this intuition, as Figure 5 shows that crawlers are indeed influenced by human access patterns. Thus, we can take advantage of this locality in a two-layer cache design.

In our early simulation studies, we conducted four tests to investigate using only the L2 cache on the file system[1]. The first test is a sequential read of all of the static cache files, followed by a randomized read of all the files in the L2 cache, which should represent two bounds of the hard drive seek time spectrum. The sequential read yields a mean seek time of 1.2ms per file, while the random read yields 12ms per file—an order of magnitude slower and close to the disk's expected seek time.

With these bounds as a starting point, we conducted two trace-driven tests. First, we read all the files in the order in which they were originally requested in the web server traces, resulting in a 7.8ms mean seek time per file. Then, we loaded all the files into a table in a MySQL database, which includes the benefit of a query cache and other optimizations, such as not having to continually retrieve the status of new files. This reduces the average seek time per file by nearly half, to 4.5 ms per file, as listed in Table 4.

Note that the time reduction between the random read and trace-driven read tests further evidences the locality of the request pattern. The addition of a RAM-resident cache allows us to reduce processing time even further, below the (ideal) sequential hard drive seek time. Ultimately, we augmented our approach with a RAM-resident cache for a further reduction in processing time.

[1]The L2 seek tests are performed on the SATA drives, not the solid state drives used in our production server.

	120 MB	250 MB	500 MB
LRU	54.76	61.22	68.67
LRU-human	60.67	67.93	76.55

Table 5: Comparison of cache replacement policies and L1 cache sizes on cache hit ratio.

3.3 Cache Replacement Policy

Using three reasonable cache sizes (120MB, 250MB, and 500MB), we investigated several cache replacement policies for our L1 cache. The engineering goal for our replacement policy is that it be very fast, as the algorithm must be executed on every incoming dynamic request. Accordingly, we quickly abandoned the *least frequently used* policy, as its $O(n \ lg \ n)$ behavior is much too slow for our needs.

Similarly, we tried several experiments with both online and offline replacement algorithms, with varying degrees of success. One online policy requires maintaining the cache with only those pages that have been requested within the last hour. This produces a meager 16% cache hit ratio. We then extended this notion to an offline replacement policy: caching only the previous day's most popular requests, which improves the cache hit ratio to 49% at 500MB. As an upper bound, it is worthwhile to note that a perfect cache (unlimited size, fully clairvoyant) on our data set yields a 90% cache hit ratio.

Nonetheless, we ultimately found that an LRU (least recently used) cache eviction policy provides an ideal hit ratio. This policy is implemented using a queue to keep track of cached entries, augmented with two fast hash tables indexing into the queue data structure to reference count each entry. This system allows older entries to be renewed in the event that a request comes in for a page about to be evicted, preventing an accidental cache eviction, and to prevent the same page from taking up multiple cache slots. Ultimately, the LRU policy meets our goals of being both fast ($O(1)$ operations) and capable of producing a desirable hit ratio: 68% at 500MB.

3.3.1 Human-Induced Prefetching

Building upon this success, we were able to achieve a considerable bump in cache hit ratio with the inclusion of human-induced prefetching into the cache monitor. Initially, we applied the crawler identification heuristic early in the process, completely ignoring all human requests with respect to the crawler cache. Rather, we found that if we passively monitor page identifiers from human requests and use them to continually pre-fetch files into the L1 cache, we can achieve an additional 8% cache hit ratio for crawlers. Table 5 summarizes the two LRU policies (with and without human-induced prefetching) for each of the cache sizes previously identified.

3.4 Crawler Cache Structure

Our approach can be readily implemented on a variety of web servers. We have chosen to implement our cache system using the combination of Apache [5], memcached [3],

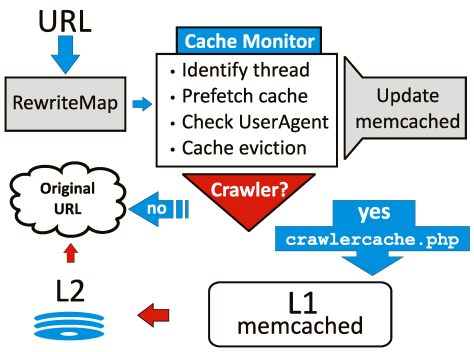

Figure 7: Cache operation

c, and php [8]. Figure 7 illustrates our cache structure, which consists of two main components: the cache monitor and the crawler cache.

3.4.1 Cache Monitor

The cache monitor is a resident daemon started by the web server, and consists of data structures that control the cache contents and record useful statistics such as the cache hit ratio. Each dynamic request is passed to the cache monitor first for processing and possible URL-rewriting. The daemon is single threaded and is used by all web server threads. It does not terminate until the web server is terminated. As each dynamic URL request comes into the web server, its query string (if any) is augmented with a URL-encoded key/value pair including the user agent string provided by the client. If no user agent is present, the URL processing stops and the request is served normally. The augmented query string for the request is then passed to the cache monitor via Apache's `RewriteMap` mechanism, which communicates with the daemon using the standard input and output.

The cache monitor itself can be written in any language that supports reading and writing from the standard input and output. We implemented versions of the cache monitor in both php and C. The daemon reads a new-line terminated URL and query string from the standard input, which is then parsed for the user agent and identifier for the page being requested. The cache monitor is thus aware of every dynamic request coming into the web server. If a request is malformed or otherwise unrecognized, the daemon returns a sentinel value indicating failure and then the original URI is handled by the web server without further involvement of the cache.

When the cache monitor is being initialized for the first time, it establishes a connection with a memcached server. Memcached [3] is a simple, configurable RAM-resident web object cache, which can be interfaced using APIs for numerous programming languages. We deployed our memcached daemon directly on the web server, with an ample 500MB, though it is common to scale memcached with several dedicated servers. We used memcached to store any L1-cached pages, with the LRU cache-eviction policy being managed directly by the cache monitor. The cache monitor communicates with the memcached server using asynchronous (non-blocking) requests, to provide failover for the web server if the memcached server is unavailable. The cache monitor is easily configured to use multiple memcached servers.

For each page identified (whether human or crawler), the cache monitor determines whether or not the request is contained within the L1 cache (memcached). If not, the monitor sends a request to the memcached server to prefetch content from the L2 (file-system) cache. When the cache size is exceeded, the cache monitor will selectively evict older cache entries from the memcached server until the cache size is within the desired threshold. By allowing both humans and crawlers to trigger prefetches into the cache, we observed an average 8% increase in the L1 cache hit ratio for crawlers, over using crawler requests alone.

Finally, a heuristic is applied to the user agent string to determine if the request is likely coming from a high-load crawler. If not, the cache monitor strips the user agent key/value pair from the augmented query string, returns the original URL back to the web server, and the request is served normally. If the request is coming from a crawler, the URL is rewritten internally to be served from the crawler cache (without an external redirect).

3.4.2 Crawler Cache

The rewritten URL is then passed back to the web server to be processed by a worker thread. The crawler cache itself is very simple, and is implemented as a php script. The script first parses the crawler request for the key identifying the cached entry, and then establishes a connection with the memcached server. The script does not interface with the cache monitor, so it must query the memcached server (L1 cache) to serve the requested content back to the crawler. If the L1 cache misses for some reason (the prefetch failed or has not yet completed before a second identical request comes in), the crawler cache loads the requested content from the file system (L2 cache). In the event of an L1 cache miss, the crawler cache does not propagate L2 cache content into the L1 cache—the crawler cache does not have access to the cache monitor data structures. If for some reason the desired content does not exist in the L2 cache (newly created pages), a proxy sub-request is initiated to dynamically generate the content. These results are then gzip-compressed and stored in the L2 cache for future retrieval.

4. IMPLEMENTATION

Our prototype system is implemented on a modest 8-core Intel Xeon, 1.6GHz 64-bit Ubuntu Linux web server with 2 GB RAM and SATA drives, loaded with Apache, MySQL, php, and memcached. The production system is a quad-core Intel Xeon, 2 GHz 64-bit CentOS dedicated Linux server with 12GB RAM, and two solid-state drives arranged in mirrored RAID configuration. The php installation makes use of the memcached dynamic extension, while the Apache installation requires both the php and rewrite engine extensions. The rewrite rules are straightforward, and are used to escape and subsequently feed the user agent and URL to the cache monitor.

We originally implemented the cache monitor in php, for several reasons: the language allows for rapid prototyping, the memcached libraries are readily available, URI processing functions are included in the language, and the internal array hash implementation is both convenient and fast. Unit testing on the php implementation takes about 180 seconds to process an entire trace. Unfortunately, the amount of metadata stored in the cache monitor combined with the size of our inputs result in nearly 1GB of RAM usage by the

end of the test. Thus, we re-implemented the cache monitor using C, the libmemcached library [7], the UriParser [9] library, and straightforward C data types. The C implementation, by comparison, requires only 2.5 seconds to process the trace, and peaks at 39MB RAM usage for the 2 weeks worth of web server logs. With this implementation, the overhead from the cache monitor is negligible. The crawler cache script is implemented in php, and since it is not in the critical path, it is not optimized for performance.

Our use of the memcached server allows both the C cache monitor and php crawler cache implementations to communicate with the same server with no language translations. Memcached is not a requirement, however, as the cache could easily be implemented using another web object cache with appropriate language bindings, or even shared memory. On the other hand, memcached is scalable to multiple servers, available on a variety of platforms, and supports multiple language interface libraries.

4.1 Space Requirements

The test site in our case study consists of about 153,100 accessible thread pages, ranging in size from 50K to well over 250K for the HTML alone. After gzip-compression, the average page size is around 14K. Most (if not all) high-load crawlers accept gzip encoding, and as such, the cached pages are all stored in a pre-compressed state, rather than wasting cycles by compressing pages at the time of the request. As an added advantage, we found that the compression enables storage for 5-6 times more content in the L1 cache, dramatically improving hit ratios.

As each php process consumes anywhere between 30MB and 300MB of RAM, we deemed 120, 250, and 500MB to be reasonable sizes for the memcached server. Most importantly, the cache size should be set so as not to cause memcached to swap, which can produce very undesirable results (one order of magnitude slower processing time). Our chosen cache size is 500MB for our production server. With 12GB of RAM available, this amount is only 4.1% of the total primary memory resource. Recall that crawlers account for 33% of all server processing time, and this number is easily justified.

To store all of the static pages in the L2 cache, it requires 2.2 GB of storage in the file system for our test site. This number is small considering our test site produces nearly 4GB worth of access logs per week, and serves 15GB of image attachments. In order to prevent running out of inodes, the numeric ID of each page is used to determine a file's location in the file system modulo some maximum number of files. For instance, an ID of 20593 (mod 250) $=$ 93 can be stored in the file system under the directory `/cache/9/3/20593.gz`, saving inodes in the process and providing some degree of organization. Of course, the crawler cache can easily interface with a lightweight database if desired.

5. EVALUATION

We evaluated the effectiveness of our approach using a live, busy web server. Our test site is a very busy online community with over 65,000 members, running the popular vBulletin [18] forum software. The site receives on average of 2 million hits per day, with around 900 users online continuously. Like many dynamic websites, the server hardware can be excessively taxed when crawled by multiple search engines. For instance, in January 2011, overload conditions were introduced to the server when Microsoft Bing 2.0 crawling agents were released to crawl the site. Yahoo's Slurp has been routinely blocked around the Internet in previous years for similar reasons. Excessive slowdowns appeared sporadically for nearly three weeks until the crawl subsided to a more manageable rate.

The two primary benefits of our approach from the perspective of performance are reduced response time and increased throughput. Under extremely heavy crawler load, targeting crawlers directly with caching is very effective at staving off overload conditions, enabling uninterrupted service to human users.

5.1 Experimental Setup

We developed a parallel tool written in php to test our cache setup, which takes as input the web server traces gathered over a two week window. The script is parameterized by the number of child threads to use, with each child thread responsible for a portion of the trace, and allows a test with nearly 90% CPU utilization on all cores. A snapshot of the online community database is configured and used along with the traces pulled from the web server logs. As a result, we created a live and accurate environment for experimental tests. We limited our trace to only dynamic requests for page listings. The trace includes $2,208,145$ bot requests and $1,052,332$ human requests for the `page.php` script, which is the primary workhorse for our case study.

5.2 Reduced Response Time

We first run the trace through the web server to gather baseline measurements, and we found the mean page generation time to be $162,109\mu s$, on order with our production server. This is the average latency experienced for all human users, as well as crawlers without the cache. With the addition of the crawler cache, we gained a reduction in latency of three orders of magnitude for crawler requests, to $1,367\mu s$. As an additional benefit of the crawler cache, the user-perceived response time for human users is reduced by 31%, to $111,722\mu s$. These results are summarized in Figure 8.

5.3 Increased Throughput

Achieving the reduced latency, we also observed a considerable increase in server throughput, as shown in Figure 9. The throughput for the traces without the crawler cache is 117 satisfied requests per second. With the addition of the crawler cache, throughput nearly doubles to 215 requests per second. With a higher throughput, overall aggregate response time for the entire trace is cut in half. Under periods of heavy load, this added capacity would be enough to stave off overload conditions caused by benign-yet-heavy crawlers.

5.4 Cache Generation

Ideally, most sites dependent on human activities (blogs, online communities, social networks) will have considerable dips in server load, providing an opportunity for L2 cache generation. In our case, based on diurnal usage patterns, we chose 4am to generate or update any new cache entries, though this parameter will depend heavily on the site's usage characteristics. To fully generate our cache from scratch on our test machine, we spent 1 hour and 22 minutes in using a parallel php script with 8 threads. This initialization need only be performed when the cache is deployed.

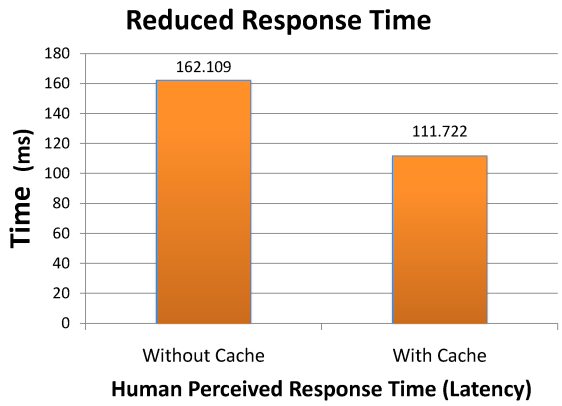

Figure 8: Crawler cache reduces server latency, improving user response time for human users.

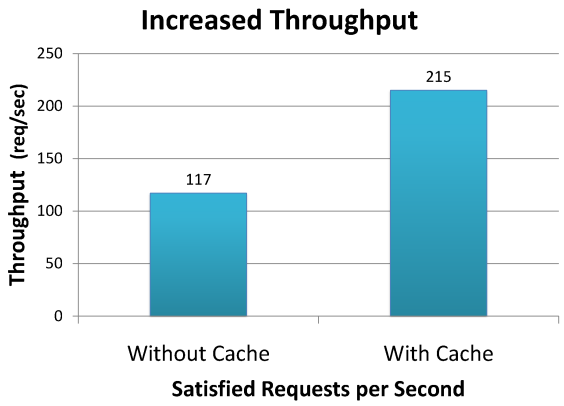

Figure 9: Crawler cache reduces server workload and as a result increases throughput.

Cache refreshing will also depend heavily on the site's database write characteristics. In our case, human users only modify around 200 unique threads per day on average (with multiple updates to those threads throughout the day). With only 200 modified pages to generate daily, this implies that the L2 cache can be refreshed in under 1 second. For sites like ours, the L2 cache can be updated multiple times per day, always providing fresh content to the crawlers as threads are updated.

5.5 Limits to Our Approach

For busy dynamic sites such as online communities, forums, and blogs, caching can be very beneficial to mitigate crawler overload risk. Unfortunately, for sites with an extremely high degree of transient data, such as event times and stock prices, static caching may not be the best approach. However, given the archival nature of the search engines, sites with a large amount of transient data are not well suited to crawling in general. These sites might better benefit from rewriting the URL to a static page explaining the possible benefits of visiting the page live.

In our live site, each dynamic request requires loading session and site configuration data, validating the security of the request, and making several trips to the database to assemble a complete page from various compound templates. This induces a considerable disparity between the time required to serve dynamic and static requests, between two and three orders of magnitude. Very simple, lightly-featured templated systems may have a smaller gap, and might not benefit as drastically from our approach. However, the current trend is toward richer, more complex, programmable content management systems.

In our case (as in most dynamic communities), the static cache is not applicable to human users. We rely on the property that crawlers are fed "guest-level" content, which makes this segment of the population cacheable. For instance, each page loaded by a logged-in human user includes a check for private messages and online chat updates, as well as filtering out posts from "ignored" users, and an applied security model to control access to paid-subscription areas; this transient data and high degree of customization make the pages uncacheable for human users with these techniques.

6. RELATED WORK

Caching has long been studied and recognized as an effective way to improve performance in a variety of environments and at all levels of abstraction, including operating system kernels, file systems, memory subsystems, databases, interpreted programming languages, and server daemons. Caching in general is a method for transparently storing data such that future requests for the same data can be served faster. Our work, to our knowledge, is the first to methodically study crawlers in the context of caching to reduce server load, and to suggest how these crawler overload can be mitigated as a result of a few readily observable crawler properties. A distinguishing feature of our work is a uniform increase in throughput without resorting to caching for human users.

Caching techniques for static websites have been studied thoroughly [12, 21, 24]. Most of these techniques do not apply generally to dynamic websites, due to the inherent customization in dynamic sites. As a result, many different caching approaches for dynamic websites have been proposed. Research into caching for dynamic websites is usually implemented at various layers of abstraction. For instance, a dynamic website may include a half dozen cache mechanisms: at the database layer [11, 25], data interface layer [11], scripting layer, virtual file system, and the network proxy layer [14]. Several cache systems for dynamic websites attempt to map underlying queries to cache objects for intelligent invalidation [13, 20, 25]. The web application as a whole may also include several programmatic caches to cache repeated function results, web objects, and templates [3].

One phenomenon related to our work is the Flash Crowd; non-malicious, sudden onslaught of web traffic that can cripple server performance [19]. While burdensome to servers, high load crawlers are relatively uniform in their accesses and do not fall under the guise of flash crowds. Other recent works study the mitigation of flash crowds [15, 26], but these techniques rely on CDN's and additional servers to disperse load. Furthermore, our work targets crawlers specifically, which allows server throughput to increase uniformly while still providing dynamic content for human users.

Our work includes a measurement study of web crawler access characteristics on a busy dynamic website to motivate

our two-level cache design. Previous work measures crawler activity [17] in detail, but do not study dynamic sites at our scale, and as a result, the crawlers behave differently. Our work shows that crawlers can consume a considerable percentage of overall server load, and hence, should be handled differently than human users. Other works include detection of crawlers through examination of access logs and probabilistic reasoning [22, 23]. Our requirements are more relaxed, only that we can detect high-load crawlers quickly and efficiently.

7. CONCLUSION

Search engines are essential for users to locate resources on the web, and for site administrators to have their sites discovered. Unfortunately, crawling agents can overburden servers, resulting in blank pages and crawler overload. Fortunately, high load crawlers are easy to identify using simple heuristics. We conducted a measurement study to show that crawlers exhibit very different usage patterns from human users, and thus can be treated differently than humans. By generating a static version of a dynamic website during off-peak hours, crawlers can be adequately served fresh content from the crawler's perspective, reducing load on the server from repeated dynamic page requests. Crawlers are archival in nature and do not require the same level of updates as human users, and this property should be taken advantage of by site administrators. Since static requests can be served two to three orders of magnitude faster than dynamic requests, overall server load can be practically reduced by serving crawlers using a static cache mechanism. We have developed a two-level cache system with an LRU policy, which is fast, straightforward to implement and can achieve a high cache hit ratio. Through a real website, we have demonstrated that our caching approach can effectively mitigate the overload risk imposed by crawlers, providing a practical strategy to survive the search engine overload.

8. REFERENCES

[1] Robots.txt - standard for robot exclusion. `http://www.robotstxt.org`, 1994.

[2] Apache ttfb module. `http://code.google.com/p/mod-log-firstbyte`, 2008.

[3] Memcached - open source distributed memory object caching system. `http://memcached.org`, 2009.

[4] Google official blog: Using site speed in web search ranking. `http://googlewebmastercentral.blogspot.com/2010/04/using-site-speed-in-web-search-ranking.html`, 2010.

[5] Apache httpd server. `http://httpd.apache.org`, 2011.

[6] Controlling crawling and indexing with robots.txt. `http://code.google.com/web/controlcrawlindex/docs/robots_txt.html`, 2011.

[7] libmemcached client library for the memcached server. `http://libmemcached.org`, 2011.

[8] Php: Hypertext preprocessor. `http://www.php.net`, 2011.

[9] Uriparser, rfc 3986 compliant uri parsing library. `http://uriparser.sourceforge.net`, 2011.

[10] P. Barford and M. Crovella. Generating representative web workloads for network and server performance evaluation. In *ACM SIGMETRICS'98*, pages 151–160, Madison, WI, 1998.

[11] S. Bouchenak, A. Cox, S. Dropsho, S. Mittal, and W. Zwaenepoel. Caching dynamic web content: Designing and analysing an aspect-oriented solution. In *Middleware'06*, Melbourne, Australia, 2006.

[12] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker. Web caching and zipf-like distributions: Evidence and implications. In *IEEE INFOCOM'99*, New York City, NY, 1999.

[13] K. S. Candan, W.-S. Li, Q. Luo, W.-P. Hsiung, and D. Agrawal. Enabling dynamic content caching for database-driven web sites. In *ACM SIGMOD'01*, pages 532–543, Santa Barbara, CA, 2001.

[14] P. Cao, J. Zhang, and K. Beach. Active cache: caching dynamic contents on the web. In *Middleware'98*, London, UK, 1998.

[15] C.-H. Chi, S. Xu, F. Li, and K.-Y. Lam. Selection policy of rescue servers based on workload characterization of flash crowd. In *Sixth International Conference on Semantics Knowledge and Grid*, pages 293–296, Ningbo, China, 2010.

[16] E. Courtwright, C. Yue, and H. Wang. Efficient Resource Management on Template-based Web Servers. In *IEEE DSN'09*, Lisbon, Portugal, 2009.

[17] M. D. Dikaiakos, A. Stassopoulou, and L. Papageorgiou. An investigation of web crawler behavior: characterization and metrics. In *Computer Communications*, 28:8, 80–897, Elsevier, May 2005.

[18] JelSoft, Inc. vBulletin Forum Software. `www.vbulletin.com`.

[19] J. Jung, B. Krishnamurthy, and M. Rabinovich. Flash crowds and denial of service attacks: characterization and implications for cdns and web sites. In *WWW'02*, pages 293–304, Honolulu, HI, 2002.

[20] Q. Luo, J. Naughton, and W. Xue. Form-based proxy caching for database-backed web sites: keywords and functions. *The VLDB Journal*, 17:489–513, 2008.

[21] P. Rodriguez, C. Spanner, and E. Biersack. Analysis of web caching architectures: hierarchical and distributed caching. In *IEEE/ACM Transactions on Networking*, 9(4):404–418, 2001.

[22] A. Stassopoulou and M. D. Dikaiakos. Crawler detection: A bayesian approach. In *ICISP'06*, Cap Esterel, France, 2006.

[23] A. Stassopoulou and M. D. Dikaiakos. Web robot detection: A probabilistic reasoning approach. In *Computer Networks*, 53:265–278, 2009.

[24] J. Wang. A survey of web caching schemes for the internet. In *ACM Computer Communication Review*, 29:5, 36–46, 1999.

[25] I.-W. T. Yeim-Kuan Chang and Y.-R. Lin. Caching personalized and database-related dynamic web pages. In *International Journal of High Performance Computing and Networking*, 6(3/4), 2010.

[26] K. Yokota, T. Asaka, and T. Takahashi. A load reduction system to mitigate flash crowds on web server. In *International Symposium on Autonomous Decentralized Systems '11*, Kobe, Japan, 2011.