Chapter 1

PERFORMANCE IMPACT OF UNCACHED FILE ACCESSES IN SPECWEB99

Krishna Kant
Intel Corporation, Beaverton, OR, USA
kkant@co.intel.com

Youjip Won
Dept. of Elect & Computer Eng., Hanyang University, Seoul, Korea
yjwon@email.hanyang.ac.kr

Abstract

In this paper, we examine the characteristics of SPECweb99 benchmark with respect to its file-caching properties. For this purpose, we develop a simple analytic model of file-cache and use it to study a number of characteristics of the workload. We find that although the memory requirements of SPECweb99 increase quite rapidly with the target throughput, the highly skewed access pattern means that a relatively small amount of memory can provide a very high cache hit ratio and thereby near maximal throughput. In particular, we find that less than 10% caching of the file-set can achieve 95% or more of the throughput that full caching can achieve. The conclusion is that in SPECweb99 benchmarking setups, it pays to carefully evaluate the efficiencies of file cache management, I/O management, and I/O subsystem components instead of attempting to cache huge portions of the file-set. The simple analytic model developed here is also indispensable in developing a performance model for SPECweb99 that can predict maximum achievable throughput for various configurations and platform parameters.

Keywords: Web-server, file-cache, HTTP, dynamic-pages, analytic-modeling, simulation

1. BACKGROUND

Much of the Internet growth is being fueled by Web usage, which makes web-server workload characterization an important and timely issue. Until recently, SPECweb96 was the only widely recognized benchmark for comparing performance of web-servers [7]. SPECweb96 suffered from several weaknesses including lack of dynamic web pages, completely cacheable file-system, very simple request generation process, insufficient skew in access patterns, etc. Many of these deficiencies have been tackled in the new web-server benchmark called SPECweb99 [8]. In particular, the storage requirements and access pattern of SPECweb99 are such that a full caching of the entire file system may not be desirable even in a benchmark configuration. This paper explores file-caching aspects of SPECweb99 to evaluate their impact on disk I/O and achievable throughput. More specifically, the paper concentrates on the buffering of the files in the main memory for quick access. This *file-cache* is usually maintained by the operating system, but could well be managed by a user-space utility. The purpose of the analysis is to identify I/O vs. memory tradeoffs for SPECweb99 workload by examining its file-caching characteristics. We note specifically that while individual disk drives may also do data prefetching and caching internally for efficient I/O, that aspect is not the focus of this work.

The paper shows that in most cases by buffering a fairly small percentage of the file-set in the main memory, it is possible to extract much of the achievable throughput from the system. That is, the benchmark, and hence web-servers running workloads similar to SPECweb99 would not benefit much from a lot of memory; instead, paying particular attention to I/O subsystem inefficiencies might be more important. For the study, the paper develops a simple analytic model of file-caching that is much more efficient than a brute-force simulation of the file-cache. This model is invaluable for developing a benchmark performance model, which is needed for predicting performance of the benchmark for a variety of configurations.

The outline of the paper is as follows. Section 2. provides an overview of SPECweb99 benchmark and discusses file-access and other details relevant for file-caching purposes. Section 3. then develops an analytical model of file-cache performance and shows its validation against simulation results. Section 4. discusses a number of characteristics of the workload derived from the analytic model. Finally, section 5. concludes the discussion and outlines future work on the subject.

2. CHARACTERISTICS OF SPECWEB99 BENCHMARK

SPECweb99 uses one or more client systems to create the HTTP workload for the server. Each client process repeats the following cycle: sleeps for some time, issues a HTTP request, waits for response, and then repeats the cycle. The response is a file in case of the GET request (or a cookie in case of a POST request). The server throughput is measured as the number of cycles (operations or transactions) per second. Stated this way, the behavior is similar to that for SPECweb96. However, SPECweb99 goes a step further and does not retain a one-to-one mapping between connections and requests. That is, (a) HTTP 1.0/1.1

keep-alive feature is used so that a new connection doesn't need to be established for each request, and (b) the number of simultaneous connections is regulated explicitly. For the latter, SPECweb99 specifies that that the aggregate transfer rate (from server to client) on each connection does not exceed 50,000 bytes/sec. (To avoid too many connections, a lower bound of 40,000 bytes/sec is also enforced.) Successive requests can be made over a given connection with sleep time being dynamically adjusted to ensure that the byte rate limitation is not violated. At least 70% of the connections must use keep-alive, with unif[5,15] distribution for the number of HTTP sessions per TCP connection.

SPECweb99 categorizes the workload into two categories: *static* and *dynamic*. There are four different sub-types in dynamic part of the workload: standard dynamic GET, dynamic GET with ad rotation, dynamic GET with CGI (common gateway interface), and dynamic POST. Except for dynamic POST which makes up only 4.8% of the total workload, the remaining 95.2% of the requests are concerned with obtaining a web-page (or "file") from a predefined file-set. For static GETs (70% of the workload), the requested file is returned to the client as is. In the other 25.2% of the cases, the returned file may be modified (e.g., appended, prepended, or updated with dynamically generated information such as advertisements). For one-half of the dynamic gets, the dynamic information is selected based on the user preference specified via a cookie sent along with the request. The average amount of data appended is about 6 KB. Although the execution of scripts on the server to create and add dynamic information is very CPU-intensive, it does not affect file-caching aspects within the server. In other words, the study here applies to 95.2% of the traffic (i.e., all GET requests) irrespective of the nature of the GET.

Although POSTs form only 4.8% of the entire workload, POSTs could significantly alter the nature of the workload not only in terms of CPU usage (which is not relevant here), but also in terms of memory usage and disk I/O. Every POST must log the information sent by the client in a file called POSTlog. Although each POST appends only 140 bytes to POSTlog, the implementation of POSTlog is required to use a single file, which results in considerable serialization delays due to logging. Furthermore, simple implementations may require closing the POSTlog file on each write, which means that the file may evicted from the file-cache (and written to the disk), only to be read back in on the next POST operation. In this paper we ignore any disk I/O or file-cache pollution caused by POSTs.

As in SPECweb96, SPECweb99 defines the file-set as a collection of "directories", each containing 36 files. The 36 files are divided into 4 classes, where each class contains 9 files with regularly spaced sizes. A class refers to the order of magnitude in terms of file size (class 0 sizes are in 100's of bytes, class 1 in 1000's of bytes, etc.). That is, file no i in class j has the size $S(i,j)$ given by $S(i,j) = 1.024(i+1)10^{j+1}$ bytes, $i \in 1..9$, $j \in 1..4$. The total size of a directory works out to be about 5.0 MB (M=10^6).

In SPECweb99, the access pattern is is specified at 3 levels. The access pattern involves Zipf distribution, which is defined as follows:

Let $\mathcal{N} \in 1..N$ be a discrete random variable with Zipf distribution. Then,

$$P(\mathcal{N} \leq n) = \sum_{i=0}^{n} \frac{\mathcal{C}}{i^\alpha}, \qquad \text{where } \mathcal{C} = \frac{1}{\sum_{i=0}^{N} \frac{1}{i^\alpha}} \tag{1.1}$$

where $\alpha > 0$ is the parameter of the distribution. The SPECweb99 access pattern can now be specified as:

1. Directory level: Zipf distribution with $\alpha = 1$ across directory numbers. Since all directories are identical, the numbering scheme used for directories does not matter.[1]
2. Class level: The relative access probabilities for classes 0-3 are 35%, 50%, 14%, and 1% respectively. That is, there is a strong preference for small files.
3. File level: A *file popularity index* is defined by using the permutation mapping from file-number (1..9) to the list {9,6,4,2,1,3,5,7,8}. This index specifies the relative access popularity of the file. The access probability itself is defined by Zipf distribution with $\alpha = 1$ over the popularity index. That is, the access probabilities for popularity indices 1..9 are as follows:

   0.353, 0.177, 0.118, 0.088, 0.071, 0.053, 0.050, 0.044, 0.039

Thus files 4-6 account for about 65% of all accesses.

From this description, it follows that the average access size is 14.73 KB (where K=1000 rather than 1024 for uniformity), but the median access size is only 3 KB. This points to strong preference for smaller files, which is typical in web applications. We would like to note here that these statistics are for the stored files only and not for actual responses sent out by the server. As noted earlier, dynamic GETs append about 6 KB of additional information, and POSTs only need to return a few hundred bytes. The effective average size (including HTTP and TCP header overheads) is about 15.2 KB, but that is not of much interest in this paper. Another point to note (but not very relevant here) is that SPECweb99 effectively disallows optimizations such a "Jumbo frames" (available in Gigabit NICs) by restricting the maximum packet size of 1460 bytes.

SPECweb99 differs considerably from SPECweb96 in terms of file-set sizing rules and directory access pattern. In SPECweb96, the web server is configured for a *target throughput*, say λ_d. The load generation mechanism attempts to maintain this throughput so long as the server is not saturated. Consequently, for good benchmarking results, the achieved throughput λ_a is very close to and the design throughput λ_d and there is no need to distinguish between the two. λ_d determines the number of directories that the server must service. In order to model the expectation that bigger web servers will perhaps handle larger number of files, SPECweb96 requires the number of directories to increase as the square-root of the target throughput. This, rather slow, increase

[1]However, Zipf access pattern may allow some optimizations in terms of how the directory numbers are correlated to the actual location of directories (and their files) on the disk.

allows caching of the entire file-set in the main memory and thereby avoids disk reads completely during the benchmark run (following the warm-up period during which file-cache is loaded). In particular, it turns out that a few Gigabytes of memory is adequate to fully cache the entire file set even at rather large design throughput. Thus disk I/O can be avoided even for rather large configurations without significantly raising the per op cost of the configuration. Unfortunately, this "trick" hides the inefficiencies associated with file-caching, I/O management, and with I/O subsystem itself. Since complete caching is not practical in reality, this aspect reduces the usefulness of the benchmark in selecting a well-designed web-server.

Ignoring some specification differences (see below), the concept of a *target throughput* and file-set size as a function of target throughput also apply to SPECweb99, except that the file system size now increases linearly with ops/sec, which makes full caching prohibitively expensive at large throughput levels. However, the distribution of accesses over directory numbers is Zipf in SPECweb99, as opposed to the uniform distribution for SPECweb96. The high skewness of Zipf distribution makes caching much more efficient for SPECweb99, which indicates that it may not be necessary to maintain a major portion of the file-set in the cache. In effect, this also discourages full caching, and thereby forces I/O system to play a significant role in performance. How significant is this role, is the main subject of this paper.

As mentioned above, SPECweb99 explicitly regulates the number of simultaneous HTTP connections on the server. In fact, the file-set size is specified directly in terms of number of simultaneous connections. Since more connections mean more overhead, a setup would ideally keep the transfer rate close to 50,000 bytes/sec. With an overall average file size of 15.2 KB, each simultaneous connection amounts to at most 3.3 ops/sec. With this translation, we can relate the file-set size ($\mathcal{D}_{reqd}$) and target throughput (λ_d) as follows:[2]

$$\mathcal{D}_{\mathrm{reqd}} = (\mathrm{int})\,(25 + \lambda_d/5) \tag{1.2}$$

Thus, the memory needed to fully cache all directories is given by $125 + \lambda_d$ MB. The descriptors of the files stored in the file-cache may themselves be cached in a separate cache for quick access, and the space occupied by these is again linearly proportional to the number of files, and hence approximately linearly proportional to λ_d. Assuming an additional 53 MB for SPECweb99 application, O/S, and required utilities, and an additional 200 KB/connection for miscellaneous buffers and handles, the minimum memory required as a function of design throughput, denoted $\mathrm{Mem}(\lambda_d)$, can be approximated as:

$$\mathrm{Mem}(\lambda_d) = 178 + 1.06\lambda_d \tag{1.3}$$

[2] In practice, the achievable ops/sec per connection is more like 3.2 or less. In this analysis, we ignore this small difference.

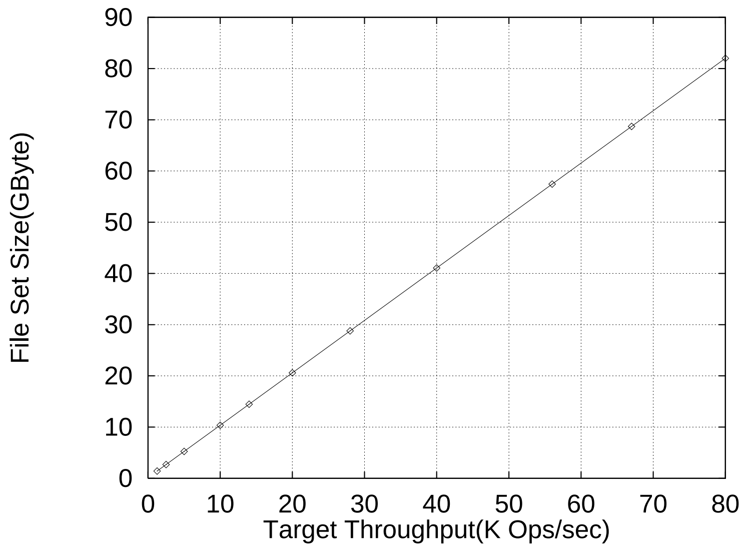

Figure 1.1 Target Throughput vs. File Set Size

Fig. 1.1 shows this relationship graphically. It is seen that 4 GB memory (the maximum directly addressable on a 32-bit machine) can support only about 3800 ops/sec with full caching. Even with a transaction path length (i.e., instructions executed per operation) of twice as much as that of SPECweb96, this performance level is easily exceeded by current generation of SMP (symmetric multiprocessor) systems. Thus, SPECweb99 memory requirements increase fast enough that full caching is not a cost-effective option.

Before closing this section, we briefly address the issue of how representative SPECweb99 is of actual web-servers. In the Internet arena, it is almost a given that there is no such thing as a "typical" workload or configuration. For example, the percentage of dynamic content on a web server is known to vary from almost zero to 100%. Similarly, the sizes and nature of documents served by a web server covers a very wide range from small text files to huge video libraries. In this sense, SPECweb99, or for that matter any other benchmark, could not capture anything about the real world. Yet, SPECweb99 is a valuable benchmark because it includes characteristics found in many large web-servers, such as Zipf (or near-Zipf) distribution of file accesses, strong preference for smaller files, a significant percentage of dynamic web pages, pages served with advertisements, large number of documents, most of which are rarely accessed but some are accessed frequently, etc. In this sense, SPECweb99 is quite realistic (much more so than SPECweb96), and the analysis of its characteristics and optimizations for it should be valuable for real servers as well. However, the nature of the traffic driving a SPECweb99 server does not appear to have been given much attention; for example, there is nothing about the client behavior that would induce properties that are commonly observed in real traffic, e.g., long-range dependence or multifractal behavior [6].

3. MODELING OF FILE CACHING PERFORMANCE

In order to study file-caching properties of SPECweb99, it is essential to build an accurate and tractable model which can be studied as a function of available memory and various target throughput levels. Unfortunately, an exact model becomes intractable because of the multiplicity of file sizes and a different access frequency for every file of every directory. Therefore, we propose here a simple approximate model that provides reasonably accurate results when the file-cache hit ratio is rather high (the expected case in practice, and certainly in benchmarking runs).

We assume that the file-cache uses the LRU (least-recently used) policy for file replacement. It is well-known that LRU does not work quite as well as some other more sophisticated algorithms in the context of web proxy caching (and hence for file-caching in a web proxy environment)[1]. However, given the SPECweb99 environment where every file has a fixed access frequency, LRU should work fairly well except when the available file-cache is very small.

Consider a SPECweb99 setup with N_d directories, $N_c = 4$ classes, and $N_s = 9$ files per class. A file can be uniquely identified by an integer in the range $1..N_s$. For convenience, we let i_s refer to the popularity index of the file (rather than the file number). Also, let q_{i_c} denote the probability that a class i_c file is accessed. Then, the file access frequency is given by:

$$A(i_d, i_c, i_s) = \frac{q_{i_c}}{C\, i_d\, i_s} \quad \text{where} \quad C = \sum_{i=1}^{N_d} \frac{1}{i} \sum_{j=1}^{N_s} \frac{1}{j} \tag{1.4}$$

Suppose that we arrange all files in the increasing order of access frequency. We henceforth denote the rank of a file f in this total order as r_f. Then, under LRU discipline, the probability of finding the file cached is proportional to the access frequency. In other words, given a file-cache size F, one can determine the rank $r_{\max}$ such that all files with rank $\leq r_{\max}$ can fit the cache. The analysis here is based on the following view of caching: all files with rank $\leq r_{\max}$ normally reside in the cache and can be accessed without disk I/O; however, access to other files requires replacing some of these files temporarily and restoring them later. In this view, it is convenient to label files with rank $\leq r_{\max}$ as "cached" and all others as "uncached". The "uncached files" essentially "pollute" the cache whenever they are accessed, and we shall estimate the effect of this pollution. The total access probability of cached files, denoted α_f, is given by

$$\alpha_f = \sum_{r=1}^{r_{\max}} A(r) \tag{1.5}$$

where $A(r)$ is the access probability of file with rank r. Let η_f denote the *file-count based* file-cache hit ratio. If uncached files are of the same size as cached files, we would have $\eta_f = \alpha_f$. However, because of the preference for smaller files, uncached files are larger, and hence $\eta_f < \alpha_f$. Here we relate η_f to α_f approximately. For this, we first compute the average size of cached and uncached files in units of bytes, denoted Q'_{cb} and Q'_{ub} respectively. Let $S(r)$ denote the size of the file with rank r. Then,

$$Q'_{cb} = \sum_{r=1}^{r_{\max}} A(r)S(r) \Big/ \alpha_f \tag{1.6}$$

$$Q'_{ub} = \sum_{r>r_{\max}} A(r)S(r) \Big/ (1-\alpha_f) \tag{1.7}$$

Let $\beta = {Q'}_{ub}/{Q'}_{cb}$, which can be interpreted as the average number of cached file replacements for each reference to an uncached file. Let γ denote the probability of expulsion of cached files due to references to uncached files of larger sizes. Then,

$$\eta_f = \alpha_f - (1-\alpha_f)\gamma \tag{1.8}$$

In estimating γ, we assume that the file hit-ratio is high enough such that the system effectively comes to a steady state between successive accesses to uncached files. In this case, the probability γ is related to $(\beta - 1)$, the excess replacements to accommodate the larger uncached files. However, γ does not equal $(\beta - 1)$ because the replaced files are not all needed immediately after access to the uncached file. Thus, $\gamma = (\beta - 1) f_r$ where f_r is the probability that the replaced file is needed shortly after. f_r depends on the access pattern and was estimated as 0.333 for SPECweb99 under LRU replacement scheme based on the simulation results.

Based on this estimate, we can now obtain a more refined estimates of average sizes of cached and uncached files, denoted Q_{cb} and Q_{ub} respectively. These are given by

$$Q_{cb} = \sum_{r=1}^{r_{\max}} A(r)S(r) \Big/ \eta_f \tag{1.9}$$

$$Q_{ub} = \sum_{r>r_{\max}} A(r)S(r) \Big/ (1-\eta_f) \tag{1.10}$$

In addition to file-count based hit ratio, it is also important to estimate the byte-count based cache hit ratio, denoted η_b, which gives the fraction of bytes that are delivered out of the file-cache. Let $\bar{S(r)} = 14.73$ KB denote the overall average access size. Obviously,

$$\eta_b = Q_{cb} \Big/ \bar{S(r)} \tag{1.11}$$

From a disk I/O perspective, the number of IOs per transaction is usually a lot more important than the number of bytes transferred. We conservatively assume that a single IO can transfer up to $\mathcal{B} = 16$ KB of sequential data (some IO systems can transfer up to 32 KB of data in one IO). The average number of IOs per transaction, denoted Q_{uB}, is thus given by

$$Q_{uB} = \sum_{r > r_{\max}} A(r) \left\lceil \frac{S(r)}{\mathcal{B}} \right\rceil \Big/ (1 - \eta_f) \tag{1.12}$$

By multiplying Q_{ub} and Q_{uB} by the throughput, we obtain the disk IO rate in bytes/sec and IOs/sec, respectively.

Another issue of interest is the impact of disk I/O on the throughput. Because of the lack of well-tuned measurement results at this time, we estimate this impact only relative to what we call *maximum achievable* or *target* throughput, i.e., the throughput that can be achieved if the entire file-set was contained in the file-cache. To approach this issue, we consider the "path length" L, i.e., the number of instructions executed per transaction. For the static part of the workload, the path length in the fully cached case should be similar to that for SPECweb96 because of almost identical average access size. In fact, because a new connection is not needed in SPECweb99 for each transaction, the path length could even be smaller. From these considerations, an optimized "web-cache" type of solution could deliver path lengths of 50,000 instructions/op or lower.[3] However, depending on the implementation details of the dynamic part, the path length could increase substantially. We believe that it is possible to approach overall path lengths of as little as 100,000 instructions/op with optimized software, e.g., ISAPI (Internet Service Application Programming Interface) [4] instead of CGI. Let L_0 denote this path-length estimate assuming full caching of the files.

Now, if I/O is needed, this path length will increase by two factors: (a) disk read path length multiplied by average number of IOs per transaction, and (b) file-cache management path length because of the need for making replacements. Let ζ_D and ζ_F denote, respectively, disk read and file cache management path lengths. Based on available measurements on Intel/NT platforms, we assume that both are $\zeta_F = 6250$ instructions and $\zeta_D = 12,500$ instructions. Thus, the actual path length is given by:

$$L = L_0 + (1 - \eta_f)(\zeta_F + Q_{uB}\zeta_D) \tag{1.13}$$

It follows that the actual throughput T can be related to the target throughput T_0 as $T = T_0(L_0/L)$. It may be noted here that an optimistic estimate of L_0 is the most conservative from the achieved throughput perspective. Thus, if the systems are not able to achieve a path length of 100,000 instructions/op (which is likely in the short run based on the current measurements), the relative impact of I/O on throughput will be even smaller.

[3]The idea of a web-cache for static workloads such as SPECweb96 is to minimize context switches and system calls. For example, Microsoft's solution attempts to do everything in the user-space.

Target tput (ops/s)	File-set size (MB)	cache size (MB)	cached frac (%)	file hit ratio (%)	Disk I/O /sec in KIOs	MB	Effec. tput (ops/s)	ana or sim
2500	2688	67.2	2.50	0.744	1.54	17.1	2312	S
				0.765	1.97	20.2	2229	A
2500	2688	134.4	5.00	0.849	1.06	12.7	2378	S
				0.857	1.25	13.9	2324	A
2500	2688	268.8	10.0	0.925	0.69	9.0	2432	S
				0.925	0.69	8.5	2401	A
10000	10367	259.2	2.50	0.783	5.25	58.8	9330	S
				0.807	6.53	67.9	9087	A
10000	10367	518.3	5.00	0.870	3.70	44.6	9551	S
				0.883	4.15	46.7	9410	A
10000	10367	1037.	10.0	0.933	2.41	31.5	9737	S
				0.938	2.32	28.6	9669	A
40000	41084	1027.	2.50	0.810	18.84	213.0	37154	S
				0.838	22.14	232.8	36865	A
40000	41084	2054.	5.00	0.885	13.18	158.7	38054	S
				0.901	14.09	159.9	37987	A
40000	41084	4108.	10.0	0.941	8.42	109.4	39097	S
				0.948	7.84	97.5	38868	A

Table 1.1 Comparison between Simulation and Analytic Results

We validated the model by a simulation that emulates SPECweb99 static workload along with an LRU file-cache and a server represented by a single server queuing station. Table 1.1 shows a comparison of analytic and simulation results for a few cases. In particular, it shows the comparison for 3 target throughput levels, namely 2500 ops/sec, 10,000 ops/sec, and 40,000 ops/sec. In each case, we consider 2.5%, 5.0% and 10.0% of the file-set to be cached, thereby resulting in 9 cases. For each such case, the first row shows the simulation results, whereas the second row shows the analytic results. It can be seen that analytic results overestimate the cache hit ratios somewhat in all cases. (This is to be expected since a real LRU scheme does not result in a clear distinction between so called *cached* and *uncached* files.) However, disk I/O and achievable throughput are pessimistic for small caching levels and optimistic for high caching levels. In all cases, the accuracy is good enough to use the model for investigating file-caching behavior further.

4. RESULTS

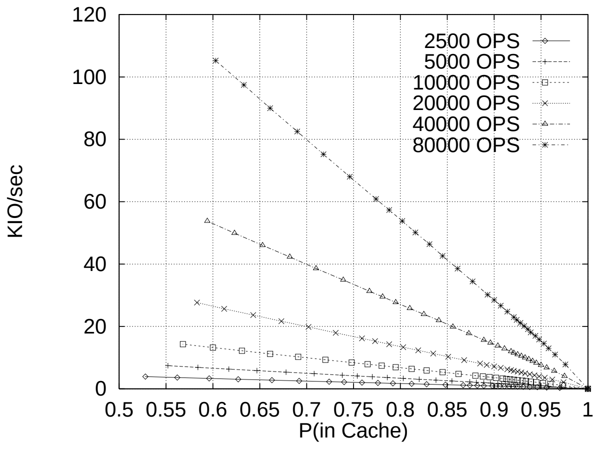

Figure 1.2 Cache hit ratio and KIO/sec

Based on the formulations provided in section 3., it is possible to estimate the amount of disk I/O operations when file cache size is not large enough to hold entire file set. Fig. 1.2 and Fig. 1.3 illustrate the relationship between the two factors: Cache hit ratio vs. amount of disk I/O for a set of target throughputs ranging from 2500 ops/sec to 80,000 ops/sec. The X-axis in both figures denotes cache hit ratio, i.e., the probability that the requested file is in the file cache. Note that the amount of disk I/O decreases almost linearly with the increase in cache hit ratio until cache hit ratio reaches about 95%.

Evidently, small size file cache implies more I/O operations, which means that a higher performance I/O subsystem is required to keep the CPU busy. Figs. 1.4 and 1.5 illustrate the IO system requirements as

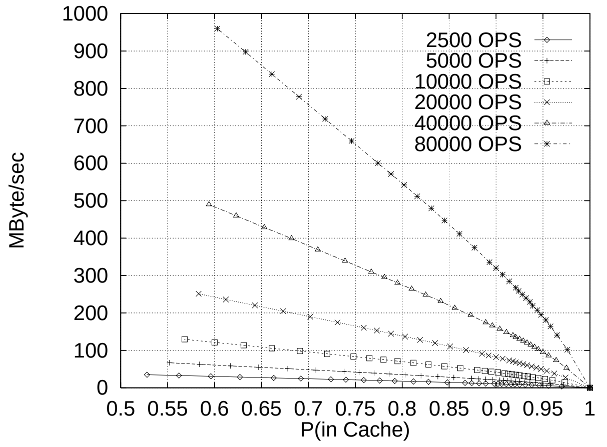

Figure 1.3 Cache hit ratio and MByte/sec

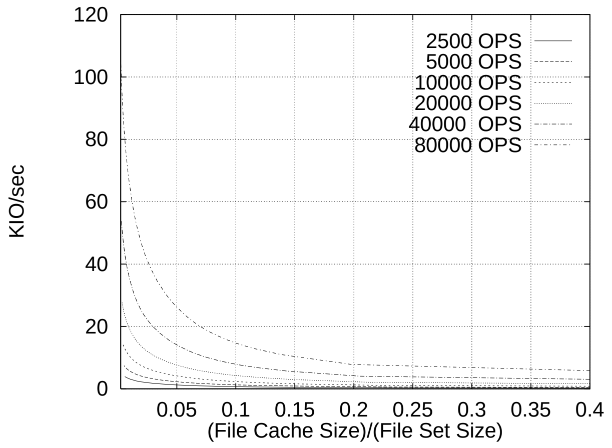

Figure 1.4 Fraction cached vs. KIO/sec

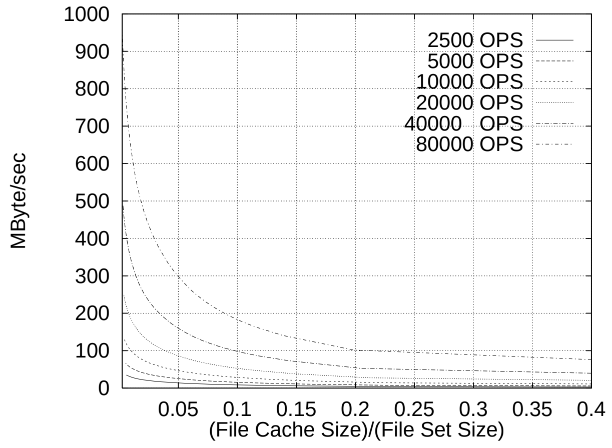

Figure 1.5 Fraction cached vs. MByte/sec

a function of cached fraction (i.e., ratio of file cache size and the file set size). This is done for a number of target throughput levels. The first graph shows the requirements in Kilo-IOs/sec, whereas the second one shows it in terms MB/sec. (Generally, IOs/sec is a much more relevant capacity metric for an IO subsystem than MB/sec.) Both of these graphs imply that beyond a certain point (about 10-15% caching level), adding memory does not reduce disk IO significantly, and a very large file-cache may not be worthwhile. It may also be noted that the curves become more skewed as the target throughput increases.

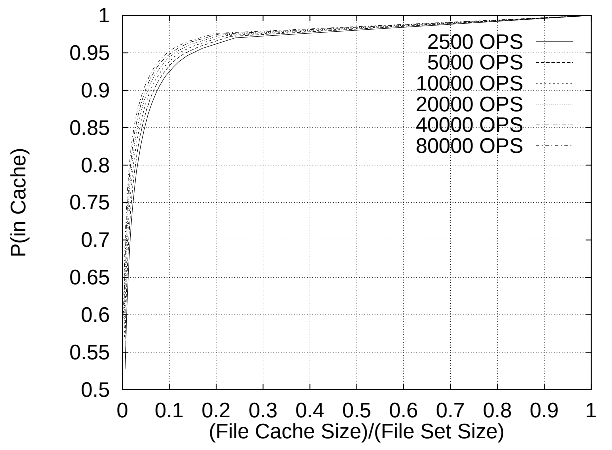

Figure 1.6 Cache hit ratio vs. Fraction cached

Fig. 1.6 shows the relationship between the cache hit probability as a function of cached fraction for a number of target throughput levels.

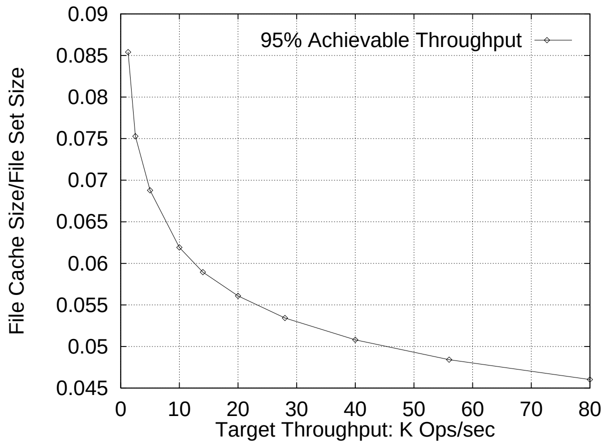

Figure 1.7 Target Throughput vs. Fraction cached

There are a number of important observations in the graph. The first issue is the effect of the *skewed access pattern*. Due to highly skewed access pattern, it is possible to achieve high cache hit ratio with a small file cache, e.g. Cache hit ratio is greater than 90% when the file cache size is approximately 10% of the file set size. Furthermore, as target throughput increases, the access pattern gets more skewed. Fig. 1.7 illustrates this skewness by plotting the caching fraction needed to achieve a fixed 95% of the target throughput, as a function of the target throughput. It is seen that the cached fraction for achieving 95% of target throughput is less than 9% in all cases and decreases with target ops/sec. In particular, at 80,000 ops/sec, only 4.5% of the entire file set needs to be cached to achieve 95% throughput.

Fig. 1.8 shows the file cache size needed to achieve a given fraction of the target throughput. The important observation from this graph is that as the achievable throughput becomes closer the target throughput, the file cache size increases very rapidly. For example, with a target throughput of 40,000 ops/sec, we can achieve 39,400 Ops/sec (98.5% of target throughput) with a file cache size of only 8.4 GBytes, which is only 20% of the total file set size.

The results above also indicate an interesting tradeoff between memory and I/O. Suppose that we want to achieve a throughput of T_0 ops/sec. We can achieve this in two ways (a) design a server with target throughput of, say, 1.02 T_0 and lots of memory, or (b) design a server with target throughput of, say, 1.1 T_0, with a small amount of memory but with a much better I/O subsystem and a somewhat faster CPU. This tradeoff may be particularly relevant for large servers in view of the current limitation of 4 GB of virtual address space on 32-bit machines. We note in this regard that while some 32-bit machines can support more than 4 GB physical memory (e.g., Intel processors that have 36-bits for address lines), accessing large memory regions is

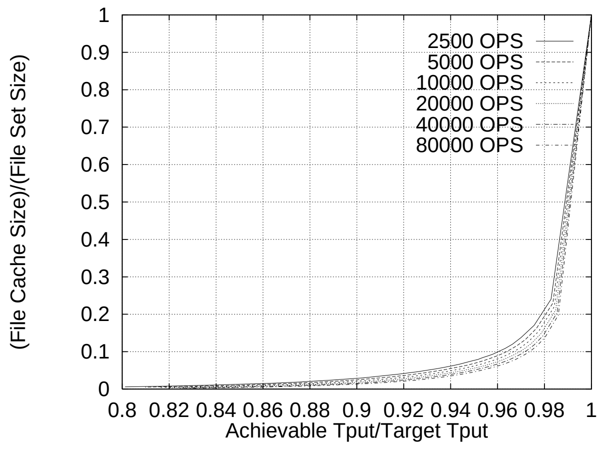

Figure 1.8 Frac of throughput achieved vs. Fraction cached

not very efficient and requires software changes. At the same time, the CPU performance of 32-bit machines is expected to continue climbing and at least remain competitive with 64-bit machines.

5. CONCLUSIONS AND FUTURE WORK

In this paper, we examine the characteristics of SPECweb99 benchmark with respect to its file-caching properties. We find that although the memory requirements of SPECweb99 increases quite rapidly with the target throughput, the highly skewed access pattern (Zipf distribution for access across directories and across files in a class) means that a relatively small amount of memory can provide a very high cache hit ratio and thereby near maximal throughput. In particular, 95% of the target throughput can be achieved by caching less than 10% of the total file size. Further increase in caching fraction increases throughput only very marginally and may not be cost effective. Thus, the 4 GB memory limitation for current 32-bit machines should not pose a significant performance hurdle for high-end web-servers that run workloads not too dissimilar from SPECweb99. It is to be noted in this regard that SPECweb99 was defined based on data from a number of major web-sites, and thus the results for SPECweb99 are of significance to real web servers in general, even if no real web server may behave exactly like a SPECweb99 server. In particular, the Zipf distribution for file accesses in a web environment is well-established [2], and so is the fact that large web servers typically host a very large number of documents, most of which are rarely accessed.

A further lesson from this work is that instead of using a lot of memory on the web-server and thereby increasing per op cost, it might be preferable to keep the amount of memory moderate and instead concentrate on tuning the I/O subsystem for optimum performance. For example, reduction in disk I/O path length by using larger transfer sizes or via more clever arrangement of file-set on the disk would directly contribute to the throughput. Similarly, a more efficient file-cache management would also contribute to the throughput. In this context, several well-known techniques can be examined. As stated earlier, a LRU file-replacement is not necessarily optimal, and other replacement policies should be evaluated. Similarly, operating system control of file-caching (and indeed O/S controlled I/O) are generally expensive, and corresponding user-level schemes should help significantly [3].

Future work on the topic includes validation of the analytic/simulation results against actual measurements. We already have a measurement setup available, however, a significant amount of tuning is needed in order to obtain good performance. The major stumbling block is the use of CGI for the dynamic content, which is extremely inefficient. As the benchmark implementation matures, especially by using ISAPI implementations instead of CGI, the focus should shift to extracting better efficiencies out of the I/O subsystem. We are also in the process of building a detailed model of the benchmark in order to enable performance projections as a function of various platform parameters (CPU speed, memory pipeline characteristics, CPU cache size and latency, etc.) along the lines of a similar model for SPECweb96 [5]. A validated, simple analytic model of file-caching is indispensable in this endeavor since the actual simulation of file-cache is very expensive.

It was mentioned in section 1. that POSTs could have a substantial influence on file-cache performance and disk I/O. A major remaining task is to understand POSTlog (including its efficient implementation) and modeling of file-caching and disk I/O impacts of it.

One issue that has not been addressed in this paper is the effect of multiple server threads on the file-caching behavior. Our analysis implicitly assumed a single-thread case. With multiple threads, file cache performance could be worse; it would be interesting to come up with a file-caching model that accounts for this and validate it against measurements.

References

[1] M. Arlitt, R. Friedrich, and Tai Jin, "Performance Evaluation of Web Proxy Cache Replacement Policies", Technical Report, HP Labs, 1998.

[2] M.E. Crovella and A. Bestavros, "Explaining World-wide web self-similarity", Technical Report, Dept of Computer Science, Boston University, Oct 1995.

[3] D. Dunning, G. Regnier, et. al., "The virtual interface architecture: a protected, zero copy user-level interface to networks", IEEE Micro, March 1998, pp66-76.

[4] "The ISAPI developers site" available at www.genusa.com/isapi.

[5] K. Kant, "A Server Performance Model for Static Web Workloads", submitted for publication.

[6] K. Kant, "On Aggregate Traffic Generation with Multifractal Properties", to appear in GLOBECOM 99.

[7] "An explanation of the SPECweb96 benchmark", available at SPEC web site www.specbench.org/osg/web96.

[8] "An explanation of the SPECweb99 benchmark", available at SPEC web site www.specbench.org/osg/web99.